CHAPTER
# ONE

# INTRODUCTION TO PARALLEL PROCESSING

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

## 1.1 EVOLUTION OF COMPUTER SYSTEMS

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

To design a powerful and cost-effective computer system and to devise efficient programs to solve a computational problem, one must understand the underlying

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

### 1.1.1 Generations of Computer Systems

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

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

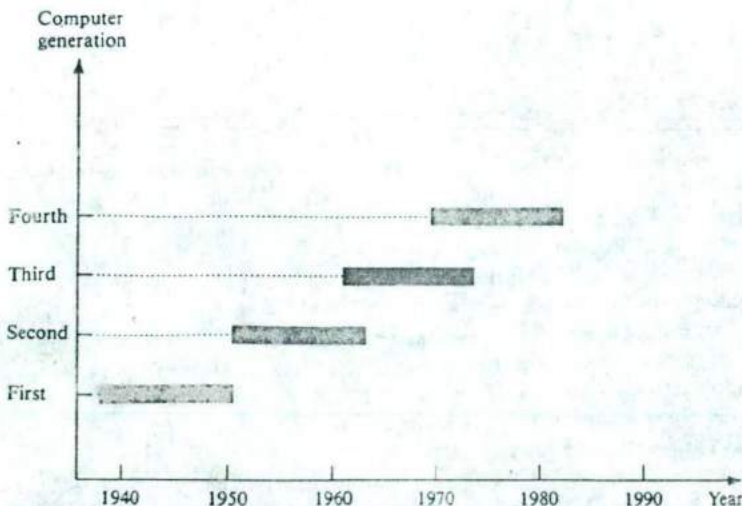



Figure 1.1 The evolution of computer systems.

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

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

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

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

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

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

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

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

## 1.1.2 Trends Towards Parallel Processing

According to Sidney Fernbach: "*Today's large computers (mainframes) would have been considered 'supercomputers' 10 to 20 years ago. By the same token, today's supercomputers will be considered 'state-of-the-art' standard equipment 10 to 20 years from now.*" From an application point of view, the mainstream usage of computers is experiencing a trend of four ascending levels of sophistication:

- Data processing
- Information processing
- Knowledge processing
- Intelligence processing

The relationships between data, information, knowledge, and intelligence are demonstrated in Figure 1.2. The data space is the largest, including numeric numbers in various formats, character symbols, and multidimensional measures. Data objects are considered mutually unrelated in the space. Huge amounts of data are being generated daily in all walks of life, especially among the scientific, business, and government sectors. An information item is a collection of data objects that are related by some syntactic structure or relation. Therefore, information items form a subspace of the data space. Knowledge consists of information items plus some semantic meanings. Thus knowledge items form a subspace of the information

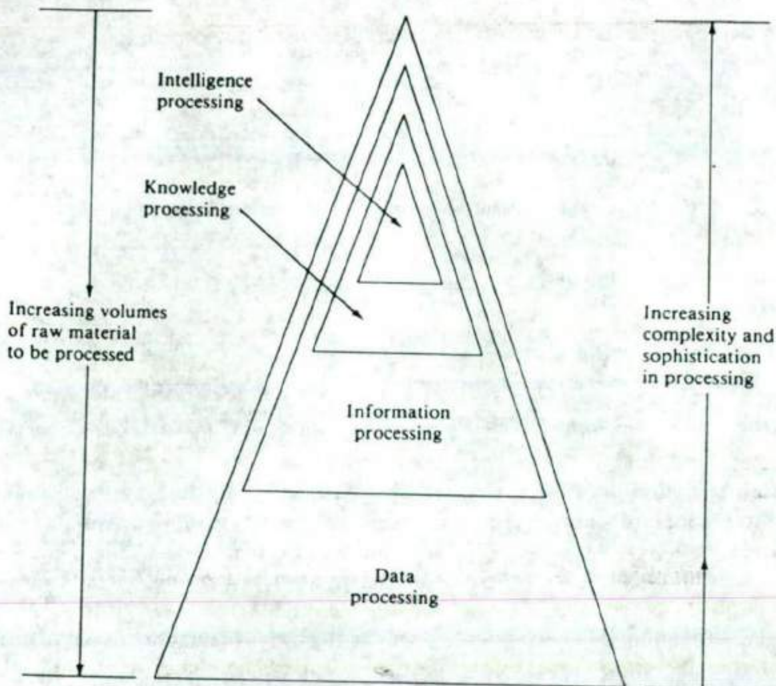



**Figure 1.2 The spaces of data, information, knowledge, and intelligence from the viewpoint of computer processing.**

space. Finally, intelligence is derived from a collection of knowledge items. The intelligence space is represented by the innermost and highest triangle in the Venn diagram.

Computer usage started with *data processing*, which is still a major task of today's computers. With more and more data structures developed, many users are shifting to computer roles from pure data processing (mainly number crunching) to *information processing*. Most of today's computing is still confined within these two processing levels. A high degree of parallelism has been found at these levels. As the accumulated knowledge bases expanded rapidly in recent years, there grew a strong demand to use computers for *knowledge processing*. For example, the various expert computer systems listed in Table 1.1 are used for problem solving in specific areas where they can reach a level of performance comparable to that of human experts. It has been projected by some computer scientists that knowledge processing will be the main thrust of computer usage in the 1990s.

Today's computers can be made very knowlegeable but are far from being intelligent. Intelligence is very difficult to create; its processing even more so. Today's computers are very fast and obedient and have many reliable memory cells to be qualified for data-information-knowledge processing. But none of the

**Table 1.1 Some existing expert computer systems for knowledge processing**

| System name | Expertise |
|---|---|
| AQ11 | Diagnosis of plant diseases |
| Internist, casnet | Medical consulting |
| Dendral | Hypothesizing molecular structure from mass spectrograms |
| Dipmeter, advisor | Oil exploration |
| EL | Analyzing electrical circuits |
| Macsyma | Mathematical manipulation |
| Prospector | Mineral exploration |
| R1 | Computer configuration |
| SPERIL | Earthquake damage estimation |

existing computers can be considered a really intelligent thinking system. Computers are still unable to communicate with human beings in natural forms like speech and written languages, pictures and images, documents, and illustrations. Computers are far from being satisfactory in performing theorem proving, logical inference, and creative thinking. We are in an era which is promoting the use of computers not only for conventional data-information processing, but also toward the building of workable machine knowledge-intelligence systems to advance human civilization. Many computer scientists feel that the degree of parallelism exploitable at the two highest processing levels should be higher than that at the data-information processing levels.

From an operating system point of view, computer systems have improved chronologically in four phases:

- Batch processing
- Multiprogramming
- Time sharing
- Multiprocessing

In these four operating modes, the degree of parallelism increases sharply from phase to phase. The general trend is to emphasize parallel processing of information. In what follows, the term *information* is used with an extended meaning to include data, information, knowledge, and intelligence. We formally define *parallel processing* as follows:

**Definition** Parallel processing is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Concurrency implies parallelism, simultaneity, and pipelining. Parallel events may occur in multiple resources during the same time interval; simultaneous events may occur at the same time instant; and pipelined events may occur in overlapped time spans. These concurrent events are attainable in a computer system at various processing levels. Parallel processing demands concurrent execution of many programs in the computer. It is in contrast to

**sequential processing. It is a cost-effective means to improve system performance through concurrent activities in the computer.**

The highest level of parallel processing is conducted among multiple jobs or programs through multiprogramming, time sharing, and multiprocessing. This level requires the development of parallel processable algorithms. The implementation of parallel algorithms depends on the efficient allocation of limited hardware-software resources to multiple programs being used to solve a large computation problem. The next highest level of parallel processing is conducted among procedures or tasks (program segments) within the same program. This involves the decomposition of a program into multiple tasks. The third level is to exploit concurrency among multiple instructions. Data dependency analysis is often performed to reveal parallelism among instructions. Vectorization may be desired among scalar operations within DO loops. Finally, we may wish to have faster and concurrent operations within each instruction. To sum up, parallel processing can be challenged in four programmatic levels:

- Job or program level
- Task or procedure level
- Interinstruction level
- Intrainstruction level

The highest job level is often conducted algorithmically. The lowest intrainstruction level is often implemented directly by hardware means. Hardware roles increase from high to low levels. On the other hand, software implementations increase from low to high levels. The trade-off between hardware and software approaches to solve a problem is always a very controversial issue. As hardware cost declines and software cost increases, more and more hardware methods are replacing the conventional software approaches. The trend is also supported by the increasing demand for a faster real-time, resource-sharing, and fault-tolerant computing environment.

The above characteristics suggest that parallel processing is indeed a combined field of studies. It requires a broad knowledge of and experience with all aspects of algorithms, languages, software, hardware, performance evaluation, and computing alternatives. This book concentrates on parallel processing with centralized computing facilities. Distributed processing on physically dispersed and loosely coupled computer networks is beyond the scope of this book, though a high degree of concurrency is often exploitable in distributed systems.

Parallel processing and distributed processing are closely related. In some cases, we use certain distributed techniques to achieve parallelism. As data communications technology advances progressively, the distinction between parallel and distributed processing becomes smaller and smaller. In this extended sense, we may view distributed processing as a form of parallel processing in a special environment.

To achieve parallel processing requires the development of more capable and cost-effective computer systems. This book emphasizes the design and application

of parallel processing computers, including various architectural configurations, functional capabilities, operating systems, algorithmic and programming requirements, and performance limitations of parallel-structured computers. The ultimate goal is to achieve high performance at lower cost in performing large-scale scientific-engineering computing tasks in the various application areas to be introduced in Section 1.5.

Most computer manufacturers started with the development of systems with a single central processor, called a *uniprocessor system*. We will reveal various means to promote concurrency in uniprocessor systems in Section 1.2. Uniprocessor systems have their limit in achieving high performance. The computing power in a uniprocessor can be further upgraded by allowing the use of multiple processing elements under one controller. One can also extend the computer structure to include multiple processors with shared memory space and peripherals under the control of one integrated operating system. Such a computer is called a *multiprocessor system*.

As far as parallel processing is concerned, the general architectural trend is being shifted away from conventional uniprocessor systems to multiprocessor systems or to an array of processing elements controlled by one uniprocessor. In all cases, a high degree of pipelining is being incorporated into the various system levels. We will introduce these parallel computer structures in Section 1.3. After learning the parallelism in both uniprocessor and multiprocessor systems, we will then study several architectural classification schemes based on the machine structures and operation modes.

## 1.2 PARALLELISM IN UNIPROCESSOR SYSTEMS

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

### 1.2.1 Basic Uniprocessor Architecture

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

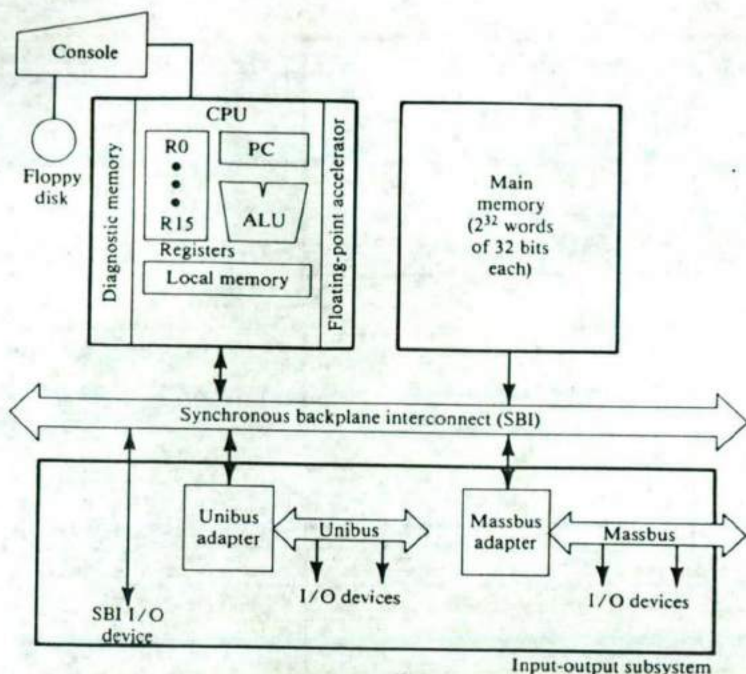



**Figure 1.3 The system architecture of the supermini VAX-11/780 uniprocessor system (Courtesy of Digital Equipment Corporation).**

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

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

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

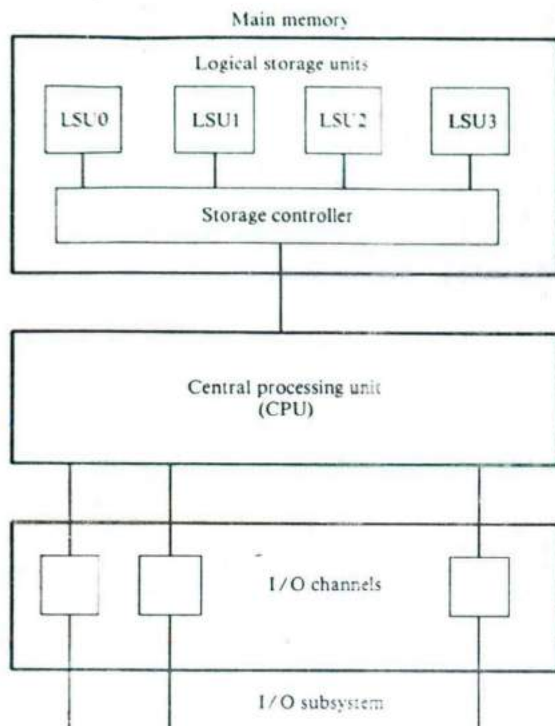



**Figure 1.4 The system architecture of the mainframe IBM System 370/Model 168 uniprocessor computer (Courtesy of International Business Machines Corp.).**

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

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

### 1.2.2 Parallel Processing Mechanisms

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

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

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

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

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

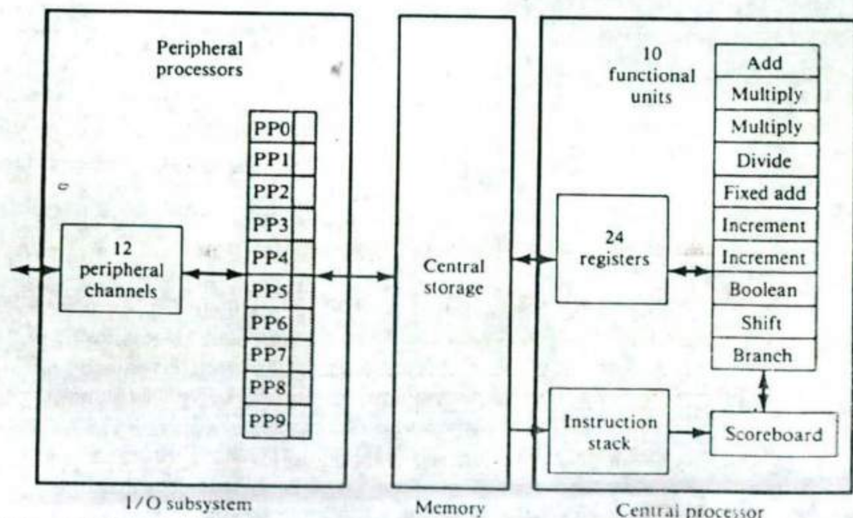



Figure 1.5 The system architecture of the CDC-6600 computer (Courtesy of Control Data Corp.).

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

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

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

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

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

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

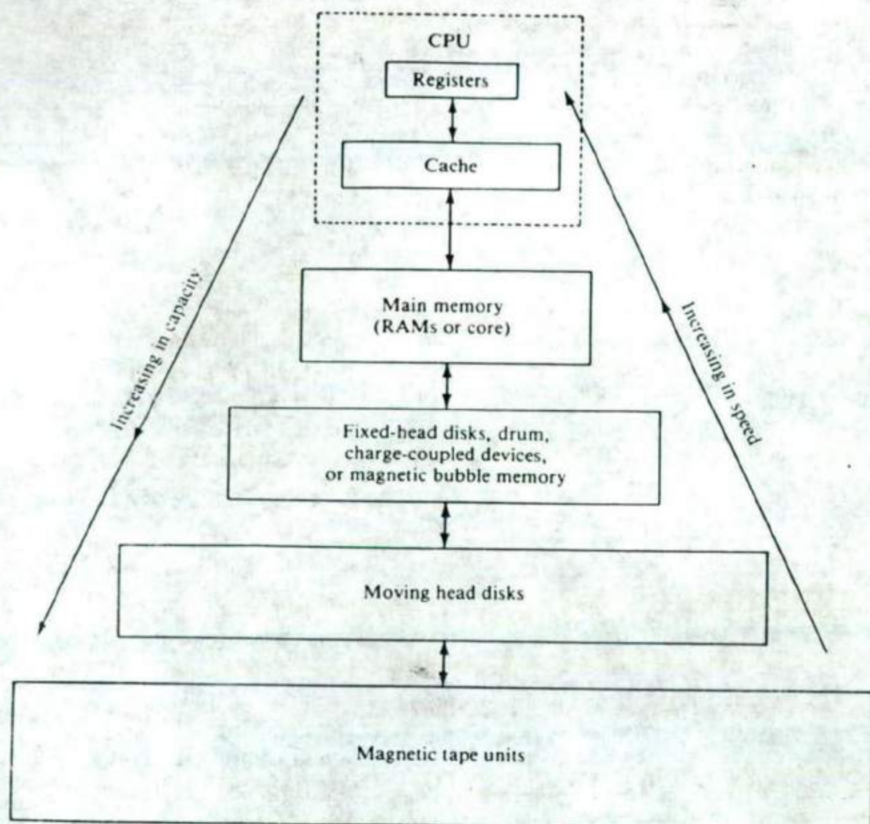



**Figure 1.6 The classical memory hierarchy.**

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

### 1.2.3 Balancing of Subsystem Bandwidth

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

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

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

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

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

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

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

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

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

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

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

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

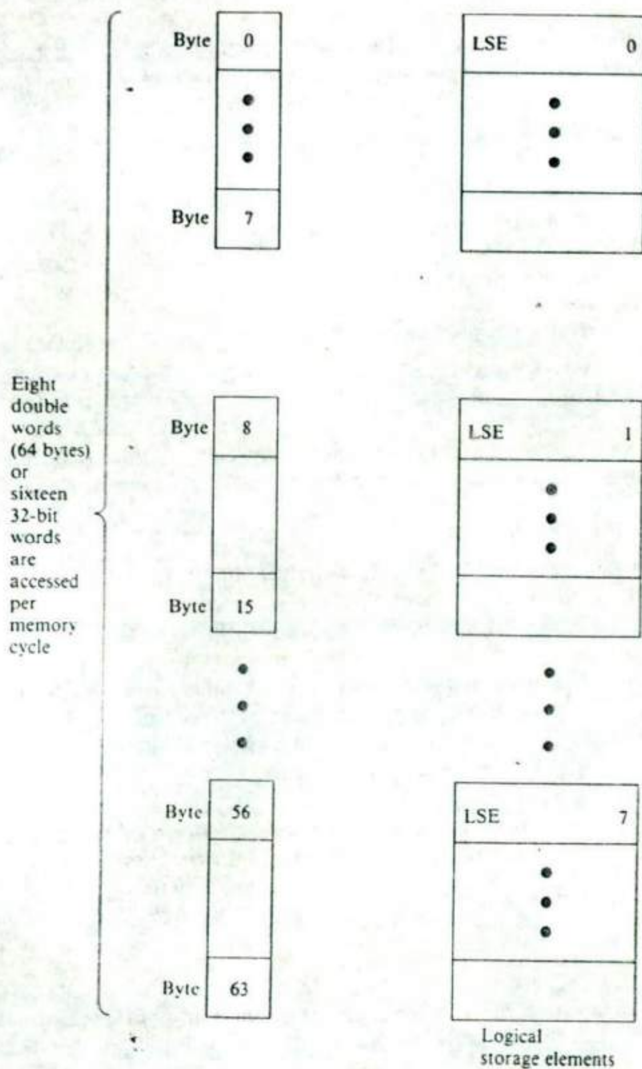



Figure 1.7 The interleaved memory structure in IBM 3033 uniprocessor.

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

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

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

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

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

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

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

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

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

### 1.2.4 Multiprogramming and Time Sharing

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

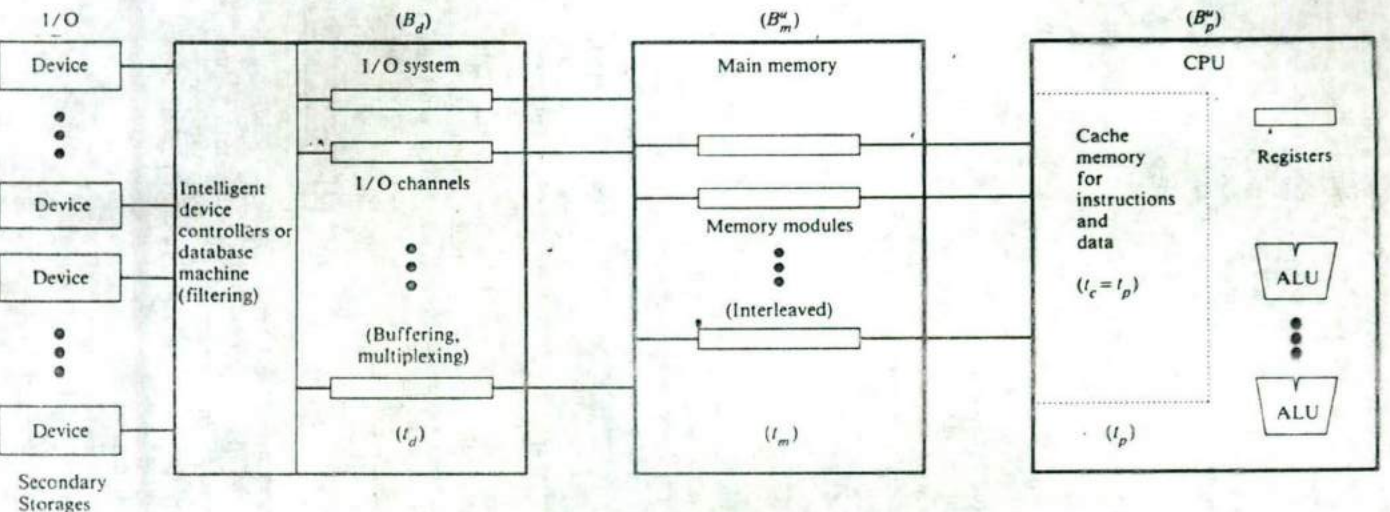



**Figure 1.8 Bandwidth balancing mechanisms between CPU, memory, and I/O subsystem in a uniprocessor computer.**

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

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

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

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

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

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

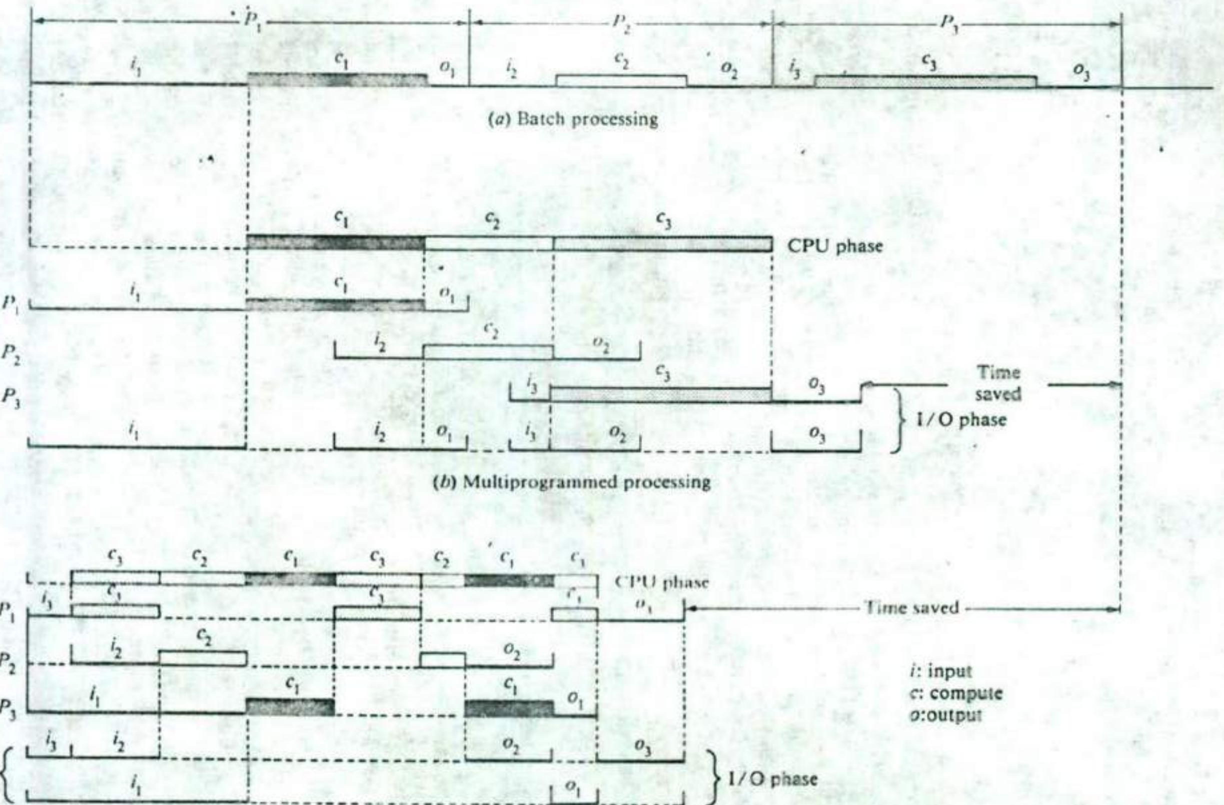



Figure 1.9 Operating system approaches to achieve parallel processing in a uniprocessor computer.

## 1.3 PARALLEL COMPUTER STRUCTURES

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

- Pipeline computers
- Array processors
- Multiprocessor systems

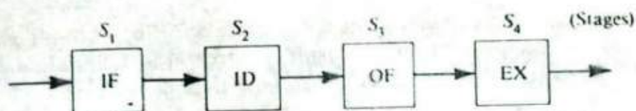
A pipeline computer performs overlapped computations to exploit *temporal parallelism*. An array processor uses multiple synchronized arithmetic logic units to achieve *spatial parallelism*. A multiprocessor system achieves *asynchronous parallelism* through a set of interactive processors with shared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

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

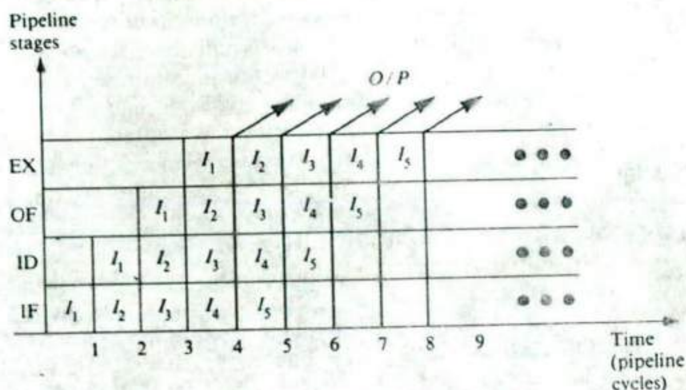
### 1.3.1 Pipeline Computers

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

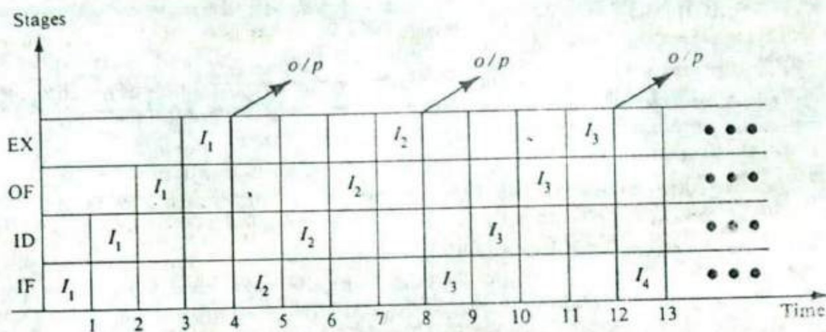
An *instruction cycle* consists of multiple pipeline cycles. A pipeline cycle can be set equal to the delay of the slowest stage. The flow of data (input operands, intermediate results, and output results) from stage to stage is triggered by a common clock of the pipeline. In other words, the operation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




(*a*) A pipelined processor




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




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

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

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

Theoretically, a $k$-stage linear pipeline processor could be at most $k$ times faster. We will prove this in Chapter 3. However, due to memory conflicts, data dependency, branch and interrupts, this ideal speedup may not be achieved for out-of-sequence computations. What has been described so far is the *instruction pipeline*. For some CPU-bound instructions, the execution phase can be further partitioned into a multiple-stage arithmetic logic pipeline, as for sophisticated

floating-point operations. Some main issues in designing a pipeline computer include job sequencing, collision prevention, congestion control, branch handling, reconfiguration, and hazard resolution. We will learn how to cope with each of these problems later.

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

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

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

### 1.3.2 Array Computers

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

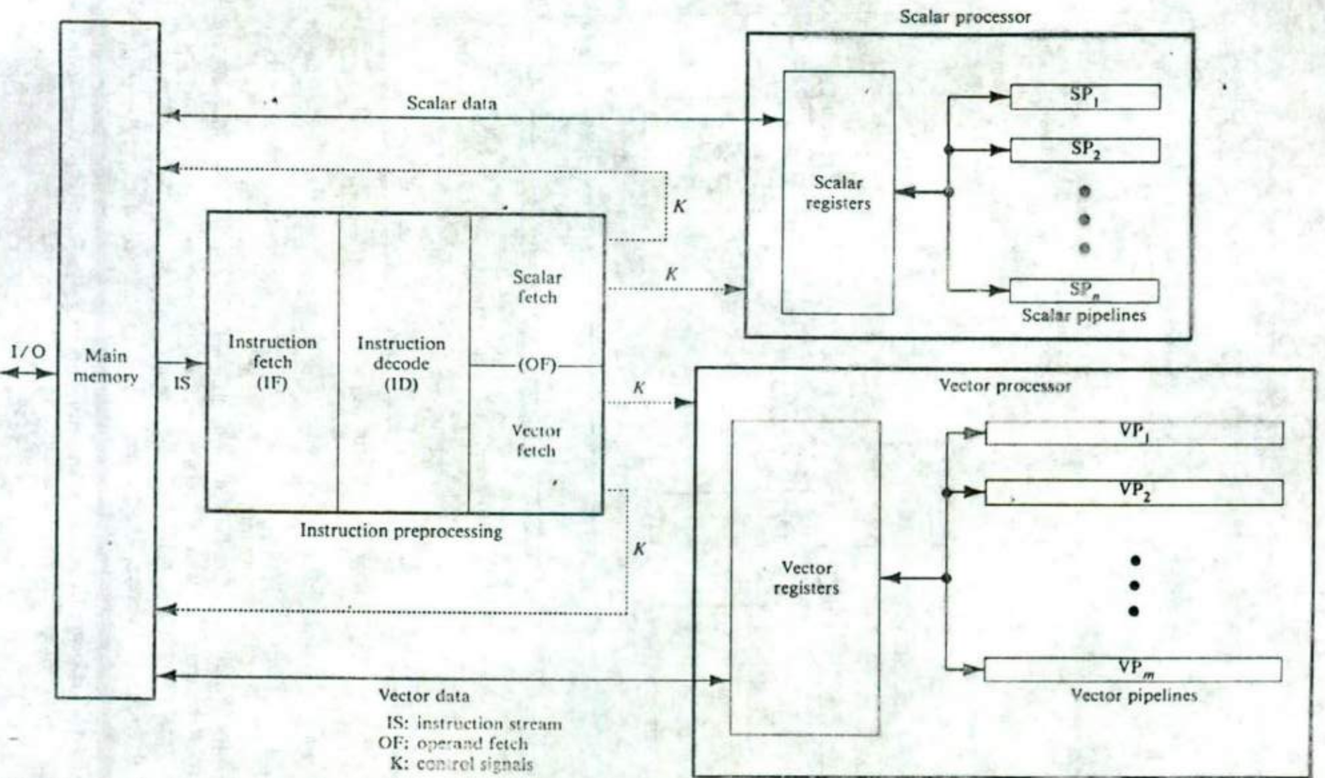



**Figure 1.11** Functional structure of a modern pipeline computer with scalar and vector capabilities.

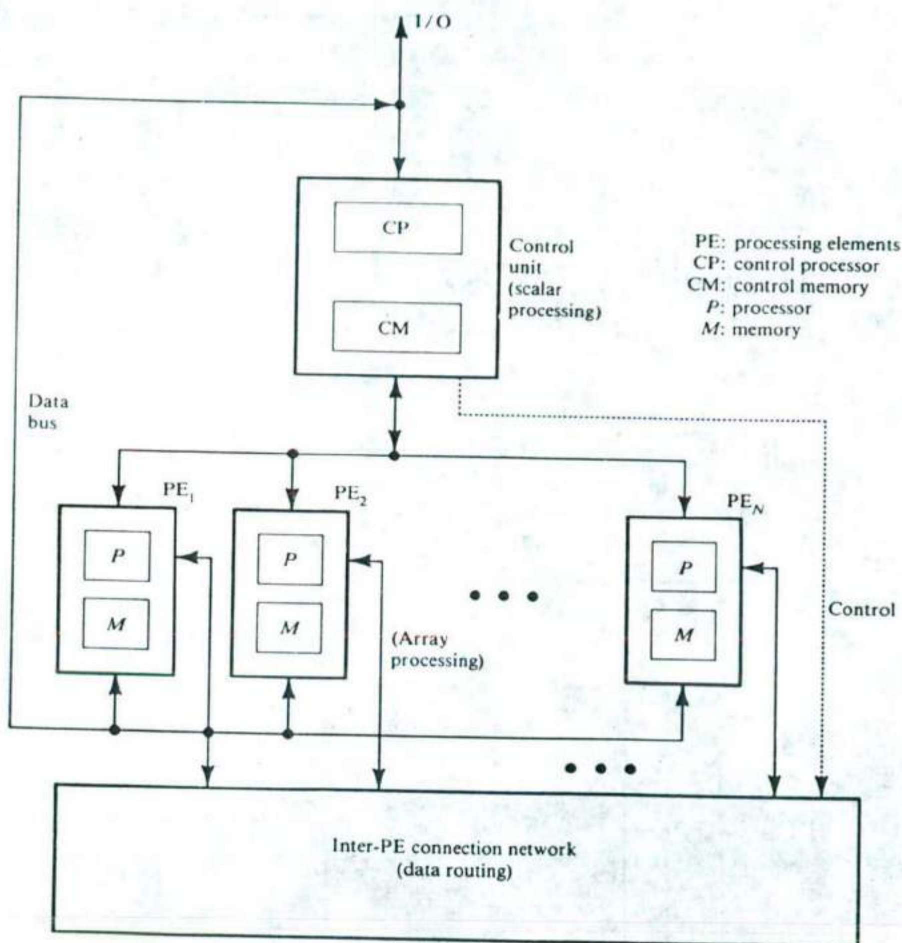



**Figure 1.12 Functional structure of an SIMD array processor with concurrent scalar processing in the control unit.**

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

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

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

### 1.3.3 Multiprocessor Systems

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

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

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

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

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

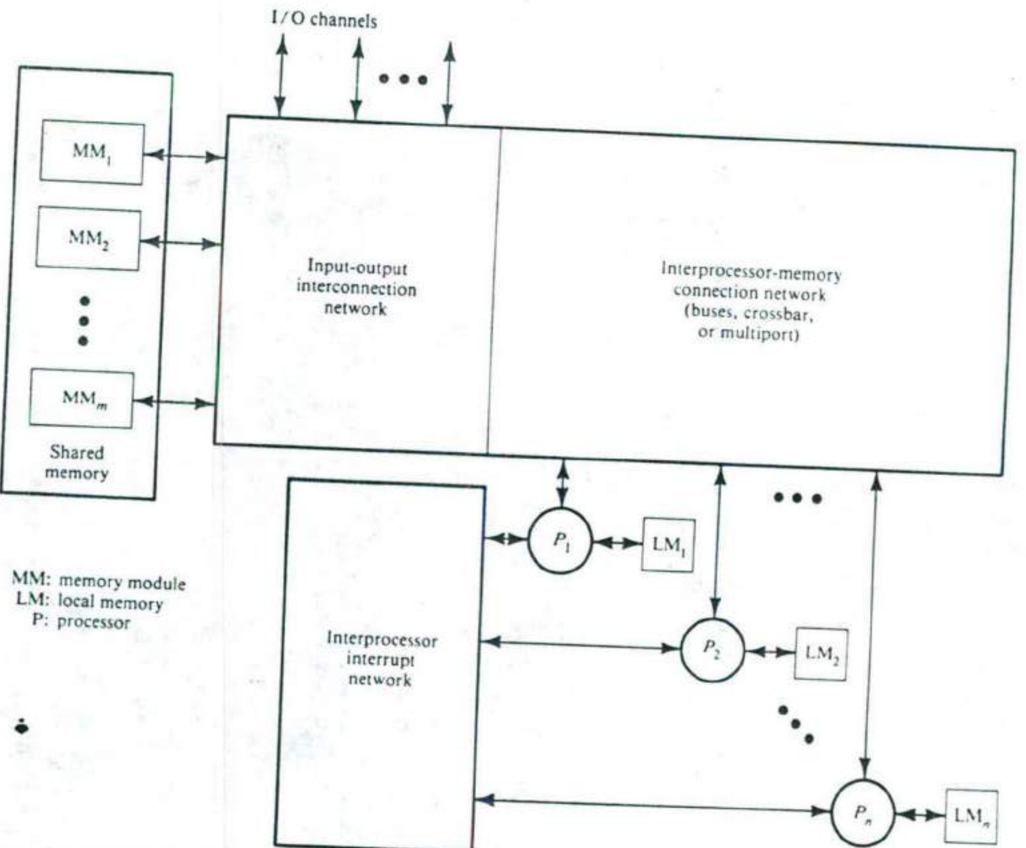



Figure 1.13 Functional design of an MIMD multiprocessor system.

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

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

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

### 1.3.4 Performance of Parallel Computers

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

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

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

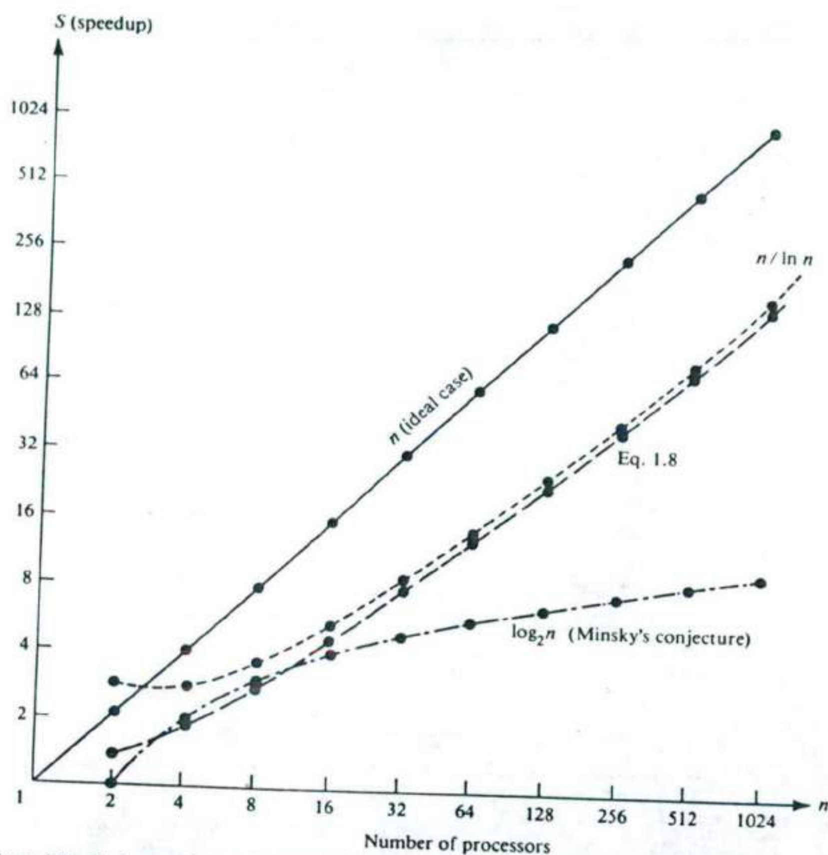



**Figure 1.14 Various estimates of the speedup of an $n$-processor system over a single processor.**

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

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

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

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

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

in Eq. 1.8 can be approximated by $n/\ln n$ for large $n$. For example, $S = 1000/\ln 1000 = 144.72$ for a system with $n = 1000$ processors. We have plotted the upper bound, the lower bound, and the speedup using Eq. 1.8 in Figure 1.14.

The above analysis explains the reason why a typical commercial multiprocessor system consists of only two to four processors. Dr. John Worlton of the United States Los Alamos Scientific Laboratory said once: "The designers of supercomputers will do better at exploiting concurrency in the computing problems if they use a small number of fast processors instead of a large number of slower processors." This conclusion coincides with the analytical prediction given in Eq. 1.8.

To measure the real performance of a computer system, one cannot ignore the computation cost and the ease in programming. Comparing multiprocessor systems with other computer structures, we conclude the following: Pipelined uniprocessor systems are still dominating the commercial market in both business and scientific applications. Pipelined computers cost less and their operating systems are well developed to achieve better resource utilization and higher performance. Array processors are mostly custom designed. For specific applications, they might be effective. The performance/cost ratio of such special-purpose machines might be low. Programming on an array processor is much more difficult due to the rigid architecture. Multiprocessor systems are more flexible in general-purpose applications. Pipelined multiprocessor systems represent state-of-the-art design in parallel processing computers. Many of the computer manufacturers are taking this route in upgrading their existing systems.

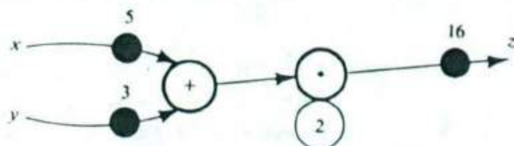
### 1.3.5 Data Flow and New Concepts

New approaches to parallel processing are briefly outlined in this section. Details of these approaches will be treated in Chapter 10.

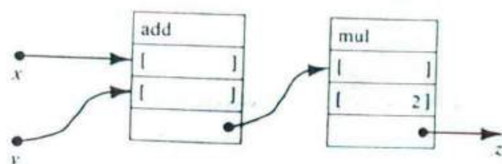
**Data flow computers** The conventional von Neumann machines are called *control flow computers* because instructions are executed sequentially as controlled by a program counter. Sequential program execution is inherently slow. To exploit maximal parallelism in a program, *data flow computers* were suggested in recent years. The basic concept is to enable the execution of an instruction whenever its required operands become available. Thus no program counters are needed in data-driven computations. Instruction initiation depends on data availability, independent of the physical location of an instruction in the program. In other words, instructions in a program are not ordered. The execution follows the data dependency constraints. Theoretically, maximal concurrency can be exploited in such a data flow machine, constrained only by the hardware resource availability.

Programs for data-driven computations can be represented by *data flow graphs*. An example data flow graph is given in Figure 1.15 for the calculation of the following expression:

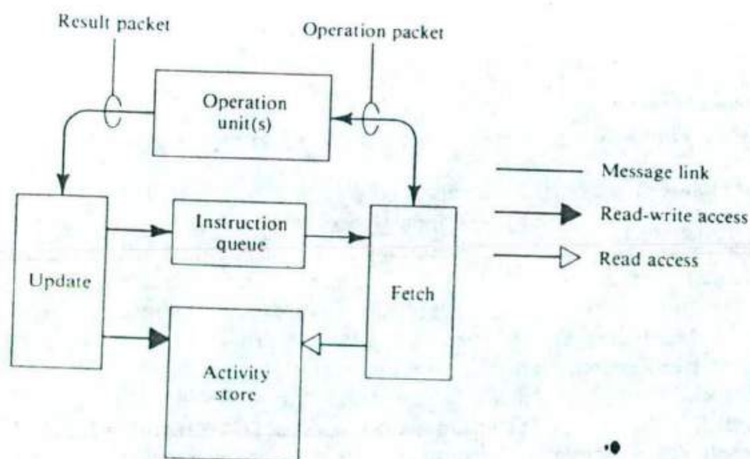
$$z = (x + y) * 2 \tag{1.9}$$




(a) Data flow program graph




(b) Template implementation




(c) Basic data flow mechanism

**Figure 1.15 Data flow graph, language, and architectural concept (Courtesy of Dennis *IEEE Computer*, 1980).**

Each instruction in a data flow computer is implemented as a *template*, which consists of the operator, operand receivers, and result destinations. Operands are marked on the incoming arcs and results on the outgoing arcs. The template implementation of the program graph in part *a* is shown in part *b* of Figure 1.15. The firing rule of an instruction requires that all receivers be filled with operand values.

The basic mechanism for the execution of a data flow program is conceptually illustrated in Figure 1.15*c*. Activity templates are stored in the *activity store*. Each activity template has a unique address which is entered in the *instruction queue* when the instruction is ready for execution. Instruction fetch and data access are handled by the *fetch* and *update* units. The *operation* unit performs the specified operation and generates the result to be delivered to each destination field in the template. This basic structure can be extended to a data flow multiprocessor, to be described in Chapter 10.

**VLSI computing structures** The rapid advent of very-large-scale integrated (VLSI) technology has created a new architectural horizon in implementing parallel algorithms directly in hardware. The new high-resolution lithographic technique has made possible the fabrication of $10^5$ transistors in an NMOS chip. It has been projected that by the late eighties it will be possible to fabricate VLSI chips which contain more than $10^7$ individual transistors. One such chip may contain more functions than one of today's large minicomputers. The VLSI development phases and definitions are summarized in Table 1.2.

The use of VLSI technology in designing high-performance multiprocessors and pipelined computing devices is currently under intensive investigation in both industrial and university environments. The multiprocessors are expected to be regularly interconnected. Pipelining makes it possible to overlap I/O with internal computations. Pipelined multiprocessing is a distinct feature of most of the VLSI computing structures that have been proposed in the literature. Most proposed VLSI arithmetic devices are for vector and matrix type computations. Both globally structured arrays and modular computing networks have been suggested for signal and image processing. We will study VLSI computing algorithms and architectures in Chapter 10.

**Table 1.2 VLSI domain and definition phases**

| Phase \ Domain | Gate equivalent count ($>10^3$ G/C)† | Line width ($<2.5\ \mu$M) | Storage density ($>30$ KB/CM$^2$) | Circuit complexity ($>16$ KD) |
|---|---|---|---|---|
| VLSI-1 | $10^3$–$10^4$ G/C | 2–4 $\mu$M | 30–100 KB/CM$^2$ | 16–64 KD |
| VLSI-2 | $10^4$–$10^5$ G/C | 1–2 $\mu$M | 100–300 KB/CM$^2$ | 64–256 KD |
| VLSI-3 | $10^5$–$10^6$ G/C | 0.5–1 $\mu$M | 300–1000 KB/CM$^2$ | 256–1024 KD |
| VLSI-4 | $>10^6$ G/C | $<0.5\ \mu$M | $>1000$ KB/CM$^2$ | $>1024$ KD |

† G/C = gates/chip, $\mu$M = micron ($10^{-6}$ meter), KB = 1024 bits, and KD = 1024 devices (transistors or diodes).

## 1.4 ARCHITECTURAL CLASSIFICATION SCHEMES

Three computer architectural classification schemes are presented in this section. *Flynn's classification* (1966) is based on the multiplicity of instruction streams and data streams in a computer system. *Feng's scheme* (1972) is based on serial versus parallel processing. *Händler's classification* (1977) is determined by the degree of parallelism and pipelining in various subsystem levels.

### 1.4.1 Multiplicity of Instruction-Data Streams

In general, digital computers may be classified into four categories, according to the multiplicity of instruction and data streams. This scheme for classifying computer organizations was introduced by Michael J. Flynn. The essential computing process is the execution of a sequence of instructions on a set of data. The term *stream* is used here to denote a sequence of items (instructions or data) as executed or operated upon by a single processor. *Instructions* or *data* are defined with respect to a referenced machine. An *instruction stream* is a sequence of instructions as executed by the machine; a *data stream* is a sequence of data including input, partial, or temporary results, called for by the instruction stream.

Computer organizations are characterized by the multiplicity of the hardware provided to service the instruction and data streams. Listed below are Flynn's four machine organizations:

- Single instruction stream-single data stream (SISD)
- Single instruction stream-multiple data stream (SIMD)
- Multiple instruction stream-single data stream (MISD)
- Multiple instruction stream-multiple data stream (MIMD)

These organizational classes are illustrated by the block diagrams in Figure 1.16. The categorization depends on the multiplicity of simultaneous events in the system components. Conceptually, only three types of system components are needed in the illustration. Both instructions and data are fetched from the *memory modules*. Instructions are decoded by the *control unit*, which sends the decoded instruction stream to the *processor units* for execution. Data streams flow between the processors and the memory bidirectionally. Multiple memory modules may be used in the shared memory subsystem. Each instruction stream is generated by an independent control unit. Multiple data streams originate from the subsystem of shared memory modules. I/O facilities are not shown in these simplified block diagrams.

**SISD computer organization** This organization, shown in Figure 1.16*a*, represents most serial computers available today. Instructions are executed sequentially but may be overlapped in their execution stages (pipelining). Most SISD uniprocessor systems are pipelined. An SISD computer may have more than one functional unit in it. All the functional units are under the supervision of one control unit.

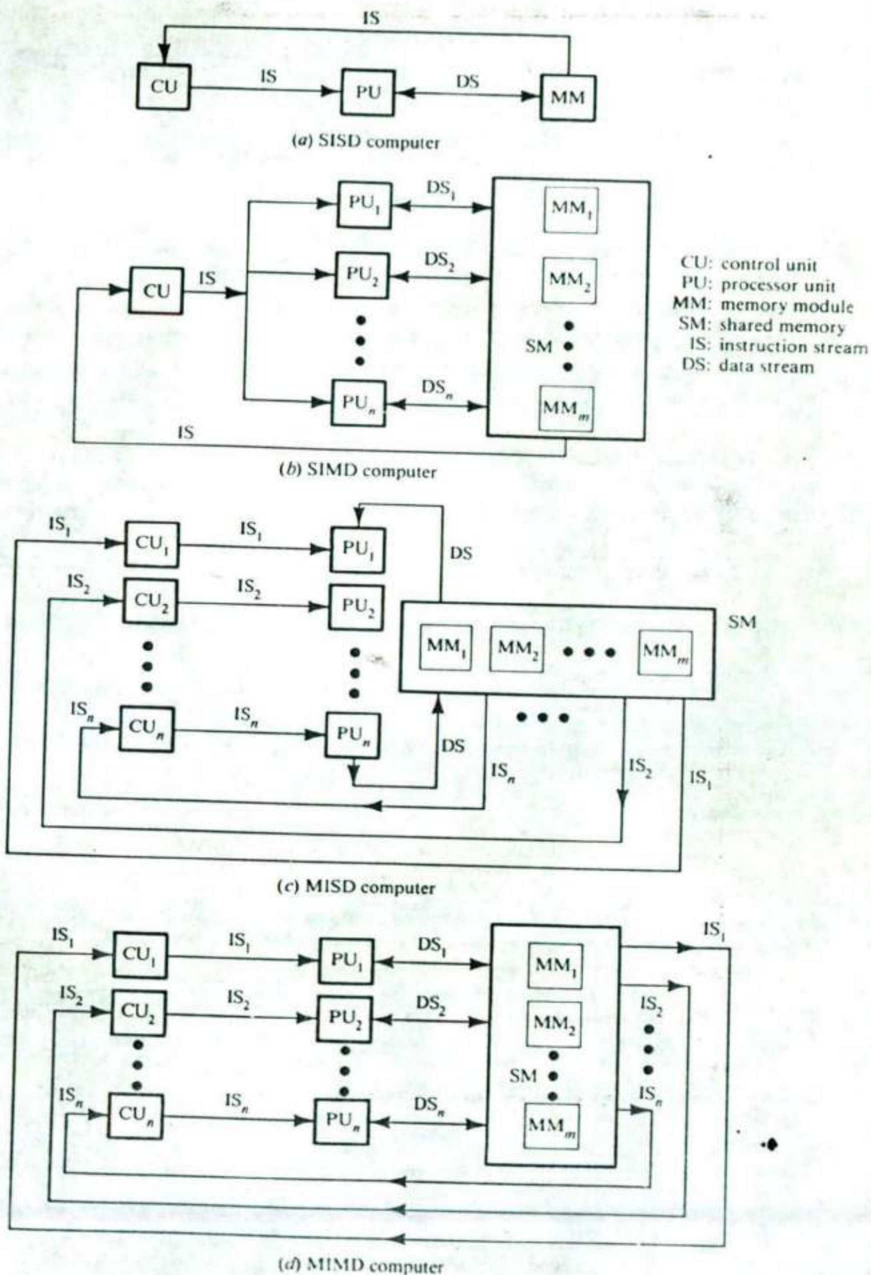



Figure 1.16 Flynn's classification of various computer organizations.

**SIMD computer organization** This class corresponds to array processors, introduced in Section 1.3.2. As illustrated in Figure 1.16*b*, there are multiple processing elements supervised by the same control unit. All PEs receive the same instruction broadcast from the control unit but operate on different data sets from distinct data streams. The shared memory subsystem may contain multiple modules. We further divide SIMD machines into *word-slice* versus *bit-slice* modes, to be described in Section 1.4.2.

**MISD computer organization** This organization is conceptually illustrated in Figure 1.16*c*. There are *n* processor units, each receiving distinct instructions operating over the same data stream and its derivatives. The results (output) of one processor become the input (operands) of the next processor in the macropipe. This structure has received much less attention and has been challenged as impractical by some computer architects. No real embodiment of this class exists.

**MIMD computer organization** Most multiprocessor systems and multiple computer systems can be classified in this category (Figure 1.16*d*). An *intrinsic* MIMD computer implies interactions among the *n* processors because all memory streams are derived from the same data space shared by all processors. If the *n* data streams were derived from disjointed subspaces of the shared memories, then we would have the so-called multiple SISD (MSISD) operation, which is nothing but a set of *n* independent SISD uniprocessor systems. An intrinsic MIMD

**Table 1.3 Flynn's computer system classification**

| Computer class | Computer system models (chapters where the system is quoted or described) |
|---|---|
| SISD (uses one functional unit) | IBM 701 (1); IBM 1620 (1); IBM 7090 (1); PDP VAX11/780 (1). |
| SISD (with multiple functional units) | IBM 360/91 (3); IBM 370/168UP (1); CDC 6600 (1); CDC Star-100 (4); TI-ASC (4); FPS AP-120B (4); FPS-164 (4); IBM 3838 (4); Cray-1 (4); CDC Cyber-205 (4); Fujitsu VP-200 (4); CDC-NASF (4); Fujitsu FACOM-230/75 (4). |
| SIMD (word-slice processing) | Illiac-IV (6); PEPE (1); BSP (6) |
| SIMD (bit-slice processing) | STARAN (1); MPP (6); DAP (1). |
| MIMD (loosely coupled) | IBM 370/168 MP (9); Univac 1100/80 (9); Tandem/16 (9); IBM 3081/3084 (9); C.m* (9) |
| MIMD (tightly coupled) | Burroughs D-825 (9); C.mmp (9); Cray-2 (9); S-1 (9); Cray-X MP (9); Denelcor HEP (9) |

computer is *tightly coupled* if the degree of interactions among the processors is high. Otherwise, we consider them *loosely coupled*. Most commercial MIMD computers are loosely coupled.

In Table 1.3, we have listed several system models under each of the three existing computer organizations. Some of these machines will be studied in subsequent chapters. Readers should check the quoted chapters for details or references related to the specific machines.

### 1.4.2 Serial Versus Parallel Processing

Tse-yun Feng has suggested the use of the *degree* of parallelism to classify various computer architectures. The maximum number of binary digits (bits) that can be processed within a unit time by a computer system is called the *maximum parallelism degree P*. Let $P_i$ be the number of bits that can be processed within the $i$th processor cycle (or the $i$th clock period). Consider $T$ processor cycles indexed by $i = 1, 2, \ldots, T$. The average parallelism degree, $P_a$ is defined by

$$P_a = \frac{\sum_{i=1}^{T} P_i}{T} \tag{1.10}$$

In general, $P_i \leq P$. Thus, we define the *utilization rate* $\mu$ of a computer system within $T$ cycles by

$$\mu = \frac{P_a}{P} = \frac{\sum_{i=1}^{T} P_i}{T \cdot P} \tag{1.11}$$

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

Figure 1.17 demonstrates the classification of computers by their maximum parallelism degrees. The horizontal axis shows the word length $n$. The vertical axis corresponds to the bit-slice length $m$. Both length measures are in terms of the number of bits contained in a word or in a bit slice. A *bit slice* is a string of bits, one from each of the words at the same vertical bit position. For example, the TI-ASC has a word length of 64 and four arithmetic pipelines. Each pipe has eight pipeline stages. Thus there are $8 \times 4 = 32$ bits per each bit slice in the four pipes. TI-ASC is represented as (64, 32). The maximum parallelism degree $P(C)$ of a given computer system $C$ is represented by the product of the word length $w$ and the bit-slice length $m$; that is,

$$P(C) = n \cdot m \tag{1.12}$$

The pair $(n, m)$ corresponds to a point in the computer space shown by the coordinate system in Figure 1.17. The $P(C)$ is equal to the area of the rectangle defined by the integers $n$ and $m$.

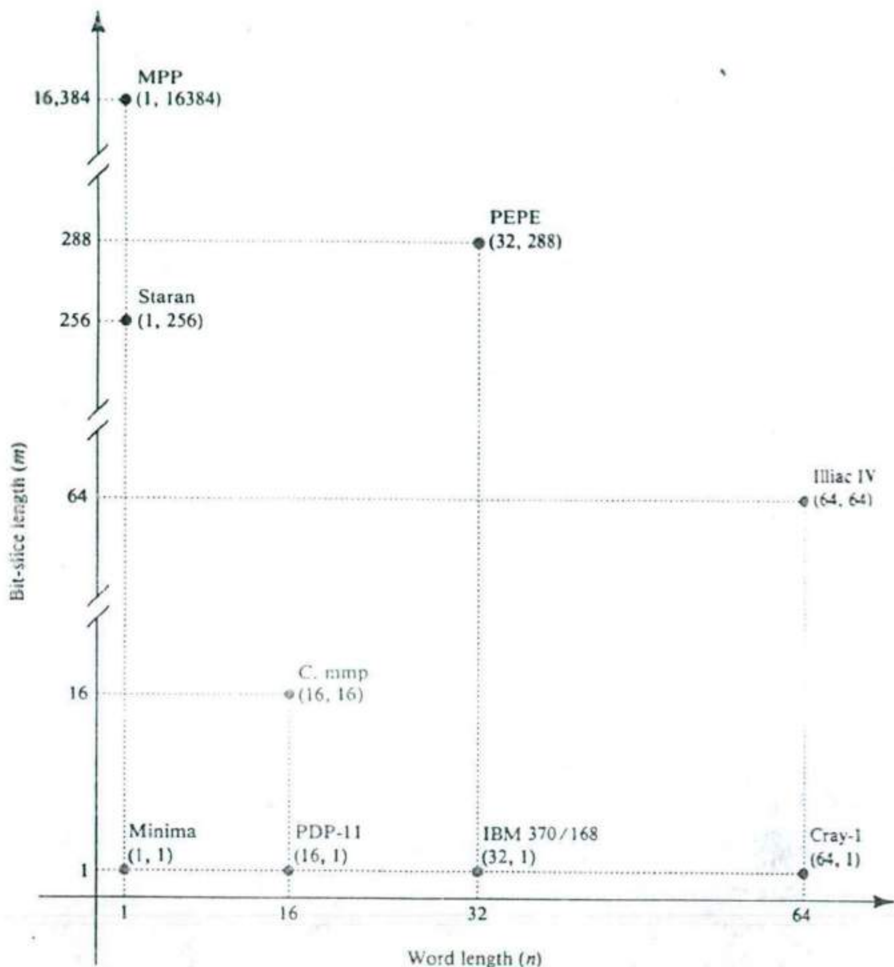



Figure 1.17 Feng's classification of computer systems in terms of parallelism exhibited by word length and bit-slice length.

There are four types of processing methods that can be seen from this diagram:

- Word-serial and bit-serial (WSBS)
- Word-parallel and bit-serial (WPBS)
- Word-serial and bit-parallel (WSBP)
- Word-parallel and bit-parallel (WPBP)

WSBS has been called *bit-serial processing* because one bit ($n = m = 1$) is processed at a time, a rather slow process. This was done only in the first-generation

**Table 1.4 Feng's computer systems classification**

| Mode | Computer model (manufacturer) | Degree of parallelism $(n, m)$ |
| --- | --- | --- |
| WSPS | The "MINIMA" | (1, 1) |
| $n = 1$ | (unknown) | |
| $m = 1$ | | |
| WPBS | STARAN | (1, 256) |
| $n = 1$ | (Goodyear Aerospace) | |
| $m > 1$ | MPP | (1, 16384) |
| (bit-slice processing) | (Goodyear Aerospace) | |
| | DAP | (1, 4096) |
| | (ICL, England) | |
| WSBP | IBM 370/168 UP | (64, 1) |
| $n > 1$ | CDC6600 | (60, 1) |
| $m = 1$ | Burrough 7700 | (48, 1) |
| (word-slice processing) | VAX 11/780 | (16/32, 1) |
| | (DEC) | |
| WPBP | Illiac IV | (64, 64) |
| $n > 1$ | (Burroughs) | |
| $m > 1$ | TI-ASC | (64, 32) |
| (fully parallel processing) | C.mmp | (16, 16) |
| | (CMU) | |
| | S-1 | (36, 16) |
| | (LLNL) | |

computers. WPBS ($n = 1, m > 1$) has been called *bis (bit-slice) processing* because an $m$-bit slice is processed at a time. WSBP ($n > 1$, $m = 1$), as found in most existing computers, has been called *word-slice processing* because one word of $n$ bits is processed at a time. Finally, WPBP ($n > 1, m > 1$) is known as *fully parallel processing* (or simply *parallel processing*, if no confusion exists), in which an array of $n \cdot m$ bits is processed at one time, the fastest processing mode of the four. In Table 1.4, we have listed a number of computer systems under each processing mode. The system parameters $n$, $m$ are also shown for each system. The bit-slice processors, like STARAN, MPP, and DAP, all have long bit slices. Illiac-IV and PEPE are two word-slice array processors. Some of these systems will be described in later chapters.

### 1.4.3 Parallelism Versus Pipelining

Wolfgang Händler has proposed a classification scheme for identifying the parallelism degree and pipelining degree built into the hardware structures of a computer system. He considers parallel-pipeline processing at three subsystem levels:

- Processor control unit (PCU)
- Arithmetic logic unit (ALU)
- Bit-level circuit (BLC)

The functions of PCU and ALU should be clear to us. Each PCU corresponds to one processor or one CPU. The ALU is equivalent to the processing element (PE) we specified for SIMD array processors. The BLC corresponds to the combinational logic circuitry needed to perform 1-bit operations in the ALU.

A computer system $C$ can be characterized by a triple containing six independent entities, as defined below:

$$T(C) = \langle K \times K', D \times D', W \times W' \rangle \tag{1.13}$$

where $K$ = the number of processors (PCUs) within the computer
$D$ = the number of ALUs (or PEs) under the control of one PCU
$W$ = the word length of an ALU or of a PE
$W'$ = the number of pipeline stages in all ALUs or in a PE
$D'$ = the number of ALUs that can be pipelined (pipeline chaining to be described in Chapter 4)
$K'$ = the number of PCUs that can be pipelined (macropipelining to be described in Chapter 3)

Several real computer examples are used to clarify the above parametric descriptions. The Texas Instrument's Advanced Scientific Computer (TI-ASC) has one controller controlling four arithmetic pipelines, each has 64-bit word lengths and eight stages. Thus, we have

$$T(\text{ASC}) = \langle 1 \times 1, 4 \times 1, 64 \times 8 \rangle = \langle 1, 4, 64 \times 8 \rangle \tag{1.14}$$

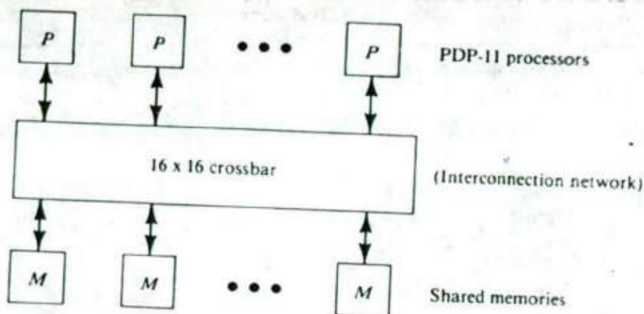
Whenever the second entity, $K'$, $D'$, or $W'$, equals 1, we drop it, since pipelining of one stage or of one unit is meaningless.

Another example is the Control Data 6600, which has a CPU with an ALU that has 10 specialized hardware functions, each of a word length of 60 bits. Up to 10 of these functions can be linked into a longer pipeline. Furthermore, the CDC-6600 has 10 peripheral I/O processors which can operate in parallel. Each I/O processor has one ALU with a word length of 12 bits. Thus, we specify 6600 in two parts, using the operator $\times$ to link them:

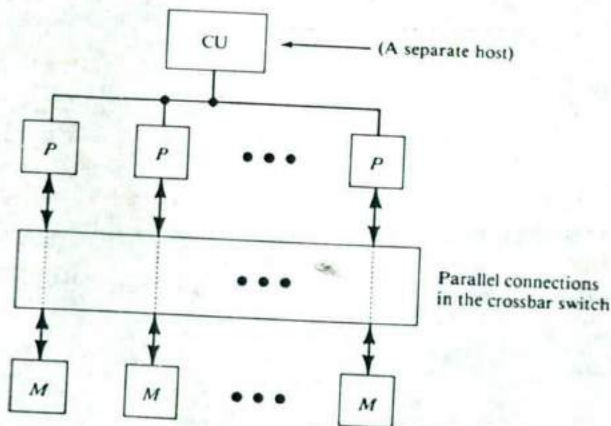
$$\begin{aligned} T(\text{CDC 6600}) &= T(\text{central processor}) \times T(I/O \text{ processors}) \\ &= \langle 1, 1 \times 10, 60 \rangle \times \langle 10, 1, 12 \rangle \end{aligned} \tag{1.15}$$

Another sample system is the C.mmp multiprocessor system developed at Carnegie-Mellon University. This system can be used in a number of ways, as illustrated in Figure 1.18. The system consists of 16 PDP-11 minicomputers of a word length of 16 bits. Normally, it will operate in MIMD mode, as shown in Figure 1.18*a*. Theoretically, it can also operate in SIMD mode, provided all the minicomputers are synchronized by one master controller, as illustrated in Figure 1.18*b*. Finally, the system can be rearranged to operate in MISD mode, as shown in Figure 1.18*c*. Based on these three operating modes, we specify C.mmp in three parts, using the operator + to separate them.

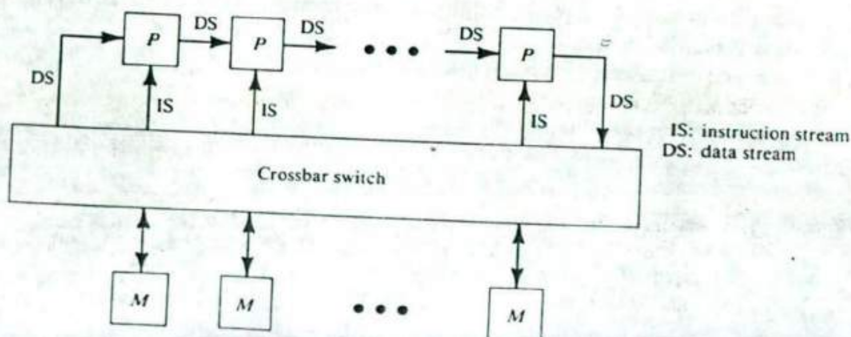
$$T(\text{C.mmp}) = \langle 16, 1, 16 \rangle + \langle 1 \times 16, 1, 16 \rangle + \langle 1, 16, 16 \rangle \tag{1.16}$$



(*a*) $T(16, 1, 16)$ for MIMD mode



(*b*) $T(1, 16, 16)$ for SIMD mode



(*c*) $T(1 \times 16, 1, 16)$ for MISD mode

**Figure 1.18 Operation modes in C.mmp system (all double-arrowed paths are for both IS and DS).**

**Table 1.5 Händler's computer system classification**

| Computer model $T(C)$ | System specification† $\langle K \times K', D \times D', W \times W' \rangle$ |
|---|---|
| $T$(TI-ASC) | $\langle 1, 4, 64 \times 8 \rangle$ |
| $T$(CDC-6600) | $\langle 1, 1 \times 10, 60 \rangle \times \langle 10, 1, 12 \rangle$ <br> central processor; I O processors |
| $T$(Illiac IV) | $\langle 1, 64, 64 \rangle$ |
| $T$(MPP) | $\langle 1, 16384, 1 \rangle$ |
| $T$(C.mmp) | $\langle 16, 1, 16 \rangle + \langle 1 \times 16, 1, 16 \rangle + \langle 1, 16, 16 \rangle$ |
| $T$(PEPC) | $\langle 1 \times 3, 288, 32 \rangle$ |
| $T$(IBM 360/91) | $\langle 1, 3, 64 \times (3 \sim 5) \rangle$ |
| $T$(Prime) | $\langle 5, 1, 16 \rangle$ |
| $T$(Cray-1) | $\langle 1, 12 \times 8‡, 64 \times (1 \sim 14) \rangle$ |
| $T$(AP-120B) | $\langle 1, 2, 38 \times (2 \sim 3) \rangle$ |

† $K'$, $D'$, and $W'$ are omitted when equal to 1.

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

In Table 1.5, we use Händler's classification scheme to specify some computer systems. It should be noted that many computers have variable numbers of stages in different functional units. Under such circumstances, we indicate the range of pipeline stages within parentheses.

## 1.5 PARALLEL PROCESSING APPLICATIONS

Fast and efficient computers are in high demand in many scientific, engineering, energy resource, medical, military, artificial intelligence, and basic research areas. Large-scale computations are often performed in these application areas. Parallel processing computers are needed to meet these demands. In this section, we introduce some representative applications of high-performance computers. Without using superpower computers, many of these challenges to advance human civilization could hardly be realized. To design a cost-effective supercomputer, or to better utilize an existing parallel processing system, one must first identify the computational needs of important applications. With rapidly changing application trends, we introduce only the major computations and leave the readers to identify their own computational needs in solving each specific problem.

Large-scale scientific problem solving involves three interactive disciplines: theories, experiments, and computations, as shown in Figure 1.19. Theoretical scientists develop mathematical models that computer engineers solve numerically; the numerical results may then suggest new theories. Experimental science provides

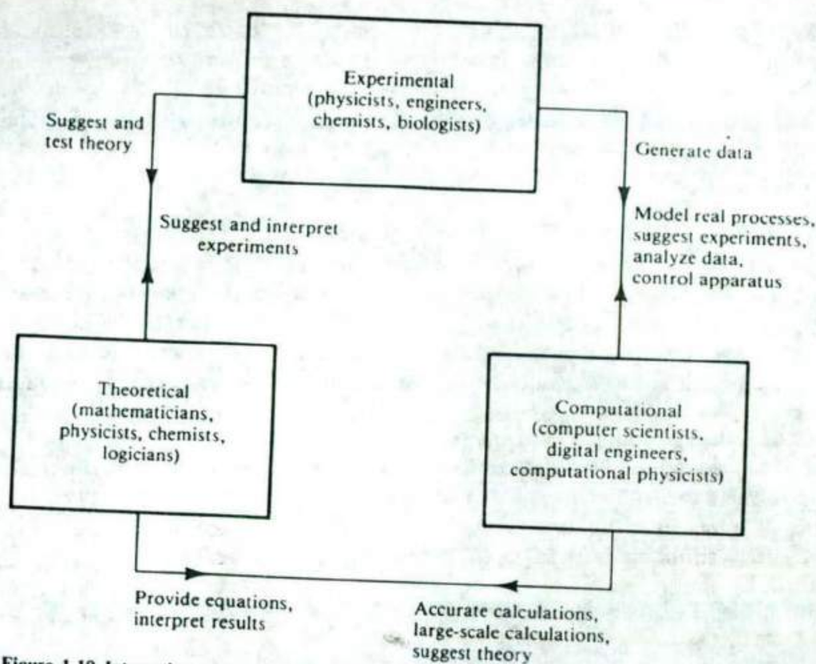



**Figure 1.19 Interaction among experiments, theories, and computations to solve large-scale scientific problems (Courtesy of Rodrique et al., *IEEE Computer*, 1980).**

data for computational science, and the latter can model processes that are hard to approach in the laboratory. Using computer simulations has several advantages:

1. Computer simulations are far cheaper and faster than physical experiments.
2. Computers can solve a much wider range of problems than specific laboratory equipments can.
3. Computational approaches are only limited by computer speed and memory capacity, while physical experiments have many practical constraints.

Theoretical and experimental scientists are users of large program codes provided by the computational scientist. The codes should yield accurate results with minimal user effort. The computer scientists must apply advanced technologies in numerical modeling, hardware engineering, and software development. In what follows, we will review parallel processing applications in four categories, according to their objectives. Within each category, we will then identify several representative application areas that have been challenged by scientists, engineers, and programmers throughout the world.

### 1.5.1 Predictive Modeling and Simulations

Multidimensional modeling of the atmosphere, the earth environment, outer space, and the world economy has become a major concern of world scientists. Predictive modeling is done through extensive computer simulation experiments, which often involve large-scale computations to achieve the desired accuracy and turnaround time. Such numerical modeling requires state-of-the-art computing at speeds approaching 1000 million megaflops or beyond.

**A. Numerical weather forecasting** Weather and climate researchers will never run out of their need for faster computers. Weather modeling is necessary for short-range forecasts and for long-range hazard predictions, such as flood, drought, and environmental pollutions. The weather analyst needs to solve *general circulation model* equations with the computer. The atmospheric state is represented by the surface pressure, the wind field, temperature, and the water vapor mixing ratio. These state variables are governed by the Navier-Stokes fluid dynamics equations in a spherical coordinate system.

The computation is carried out on a three-dimensional grid that partitions the atmosphere vertically into $K$ levels and horizontally into $M$ intervals of longitude and $N$ intervals of latitude (Figure 1.20). A fourth dimension is added as the number $P$ of time steps used in the simulation. Using a grid with 270 miles on a side, a 24-hour forecast would need to perform about 100 billion data operations. This forecast could be done on a 100 megaflops computer in about 100 minutes.

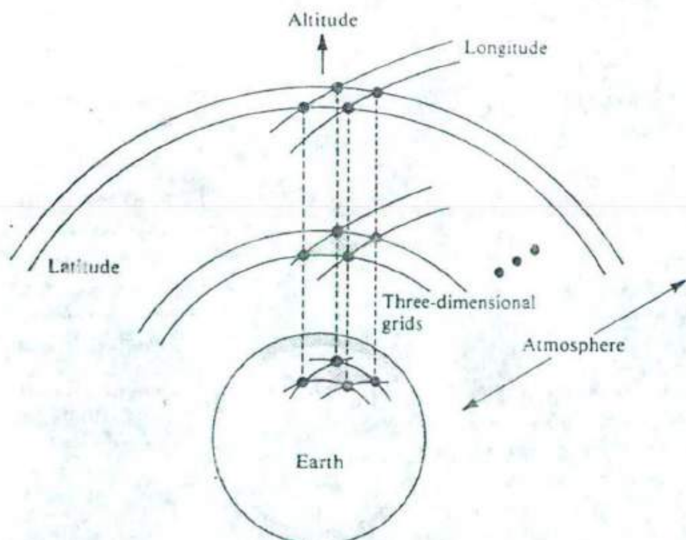



**Figure 1.20** The general circulation model for three-dimensional global atmosphere simulation used in numerical weather forecasting and climate studies.

This 270-mile grid gives the forecast between New York and Washington, D.C., but not for Philadelphia, about halfway between.

Increasing the forecast by halving the grid size in all four dimensions would take the computation at least 16 times longer. The 100 megaflops machine, like a Cray-1, would therefore take 24 hours to complete the 24-hour forecast. In other words, to halve the grid size, giving the Philadelphia weather, requires a computer 16 times more powerful (1.6 gigaflops) to finish the forecast in 100 minutes. Reliable long-range forecasts require an even finer grid for a lot more time steps, and thus demand a much more powerful computer than the 1.6 gigaflops machine.

**B. Oceanography and astrophysics** Since oceans can store and transfer heat and exchange it with the atmosphere, a good understanding of the oceans would help in the following areas:

- Climate predictive analysis
- Fishery management
- Ocean resource exploration
- Coastal dynamics and tides

Oceanographic studies use a grid size on a smaller scale and a time variability on a larger scale than those used for atmospheric studies. To do a complete simulation of the Pacific Ocean with adequate resolution (1° grid) for 50 years would take 1000 hours on a Cyber-205 computer.

The formation of the earth from planetesimals in the solar system can be simulated with a high-speed computer. The dynamic range of astrophysic studies may be from billions of years to milliseconds. Interesting problems include the physics of supernovae and the dynamics of galaxies. Three-dimensional, *n*-body integrations ran in such a study, involving $10^5$ particles moving self-consistently under Newtonian forces. The Illiac-IV array processor was used in this study.

**C. Socioeconomics and government use** Large computers are in great demand in the areas of econometrics, social engineering, government census, crime control, and the modeling of the world economy for the year 2000. Nobel laureate W. W. Leontief (1980) has proposed an input-output model of the world economy which performs large-scale matrix operations on a CDC scientific computer. This United Nations–supported world economic simulation suggests how a system of international economic relations that features a partial disarmament could narrow the gap between the rich and the poor.

In the United States, the FBI uses large computers for crime control; the IRS uses a large number of fast mainframes for tax collection and auditing. There is no doubt about the use of supercomputers for national census and general public opinion polls. It was estimated that 57 percent of the large-scale computers manufactured in the United States have been used by the U.S. government in the past.

### 1.5.2 Engineering Design and Automation

Fast supercomputers have been in high demand for solving many engineering design problems, such as the finite-element analysis needed for structural designs and wind tunnel experiments for aerodynamic studies. Industrial development also demands the use of computers to advance automation, artificial intelligence, and remote sensing of earth resources.

**A. Finite-element analysis** The design of dams, bridges, ships, supersonic jets, high buildings, and space vehicles requires the resolution of a large system of algebraic equations or partial differential equations. Conventional approaches using predeveloped software packages (written in sequential codes) require intolerable turnaround times. Many researchers and engineers have attempted to build more efficient computers to perform finite-element analysis or to seek finite difference solutions. This would imply a fundamental change of engineering design tools and higher productivity in the future.

Computational engineers have developed finite-element code for the dynamic analysis of structures. High-order finite elements are used to describe the spatial behavior. The temporal behavior can be approximated by using a central difference explicit scheme. Vectorization procedures can be used to generate the element stiffness and mass matrices, to decompose the global matrices, and to multiply the global stiffness matrix by a vector. The CDC Star-100 and Cyber-205 have been used to implement these computations for structural analysis.

**B. Computational aerodynamics** Large-scale computers have made significant contributions in providing new technological capabilities and economies in pressing ahead with aircraft and spacecraft lift and turbulence studies. NASA's Ames Research Center is seeking to supplement its Illiac-IV to do three-dimensional simulations of wind tunnel tests at gigaflop speeds. The fundamental limitations of wind tunnels and of numerical flow simulations are compared in Table 1.6. Every wind tunnel is limited by the "scale effects" attributed to the

**Table 1.6 Fundamental limitations of wind tunnel experiment and of numerical flow simulations**

| Wind tunnel experiment | Numerical flow simulation |
|---|---|
| Model size | Processor speed |
| Wind velocity | |
| Density | Memory capacity |
| Temperature | |
| Wall interference | |
| Aeroelastic distortions | |
| Atmosphere | |
| Stream uniformity | |

listed factors. In contrast, computer flow simulations have none of these physical constraints, but have their own: computational speed and memory capacity.

Two gigaflops supercomputers, known as the *Numerical Aerodynamic Simulation Facilities* (NASF), have been proposed by the Burroughs Corporation and by the Control Data Corporation. These are specialized "Navier-Stokes" machines, capable of simulating complete aircraft design for both the U.S. government and commercial aircraft companies. We will study the proposed designs, along with their predecessor vector processors, in Chapters 4 and 6.

**C. Artificial intelligence and automation** Intelligent I/O interfaces are being demanded for future supercomputers that must directly communicate with human beings in images, speech, and natural languages. Listed below are intelligence functions which demand parallel processing:

- Image processing
- Pattern recognition
- Computer vision
- Speech understanding
- Machine inference
- CAD/CAM/CAI/OA
- Intelligent robotics
- Expert computer systems
- Knowledge engineering

Special computer architectures have been developed or proposed for some of the above machine intelligence applications. Recently, Japan launched a national project to develop the fifth-generation computers to be used in the 1990s. The Japanese envision the new generation computers to possess highly intelligent input-output subsystems, capable of most of the above functions. CAD/CAM/CAI stands for *computer-aided design, computer-aided manufacturing, and computer-assisted instruction*, respectively. OA stands for *office automation*.

The projected computing power of the system being developed is 100 mega to 1 giga *logical inferences per second* (LIPS). The time to execute one logical inference equals that of executing 100 to 1000 machine instructions. Therefore, the machine should be able to execute 10,000 to 1 mega *million instructions per second* (MIPS). Such an ultrapower computer is expected to process knowledge-based information and to serve as the multipurpose expert systems demanded by applicationers in the future.

**D. Remote sensing applications** Computer analysis of remotely sensed (via satellite, for example) earth-resource data has many potential applications in agriculture, forestry, geology, and water resources. Explosive amounts of pictorial information need to be processed in this area. For example, a single frame of LANDSAT imagery contains 30 million bytes; it takes 13 such images to cover the

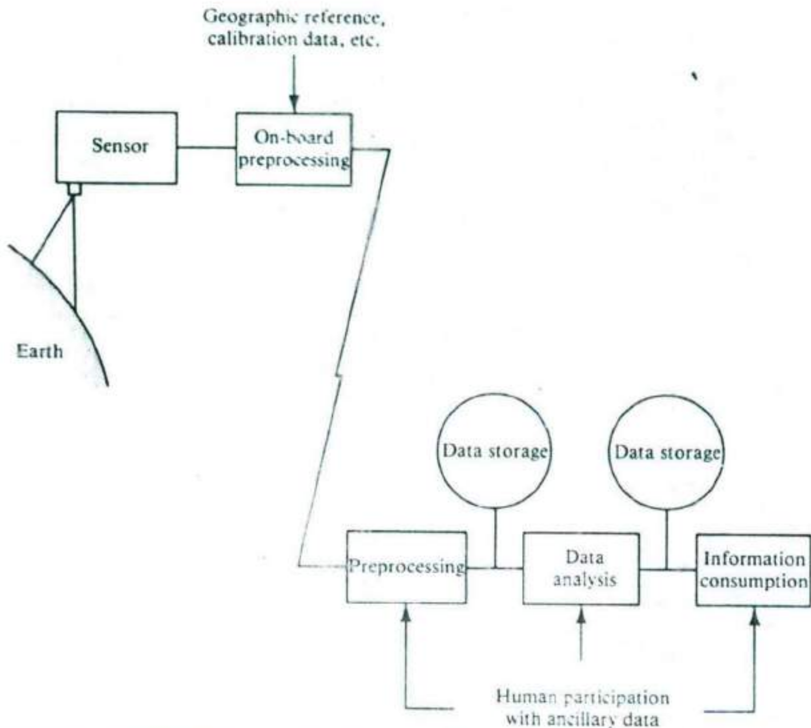



Figure 1.21 Computer analysis of remotely sensed earth resource data (Courtesy of Swain, *McGraw-Hill International*, 1978).

state of Alabama. What is even more demanding is the production of a complete new set of imageries for the entire earth surface every 15 days (Figure 1.21).

NASA has ordered a *massively parallel processor* (MPP) for earth resources satellite image processing. This MPP has a peak computing rate of 6 billion 8-bit integer operations per second. It can almost provide real-time, time-varying scene analysis, for example, where the sensor interacts with the scene. We will study MPP in detail in Chapter 6.

### 1.5.3 Energy Resources Exploration

Energy affects the progress of the entire economy on a global basis. Computers can play an important role in the discovery of oil and gas and the management of their recovery, in the development of workable plasma fusion energy, and in ensuring nuclear reactor safety. Using computers in the energy area results in less production costs and higher safety measures.

**A. Seismic exploration** Many oil companies are investing in the use of attached array processors or vector supercomputers for seismic data processing, which accounts for about 10 percent of the oil finding costs. Seismic exploration sets off a sonic wave by explosive or by jamming a heavy hydraulic ram into the ground and vibrating it in a computer-controlled pattern. A few thousand phones scattered about the spot are used to pick up the echos. The echo data are used to draw two-dimensional cross sections that display the geometrical underground strata. Reconstruction techniques are being used to identify the types of strata that may bear oil. Such seismic exploration may save the drilling of many dry holes.

A typical field record for the response of the earth to one sonic input has 3000 different time values, each at about 48 different locations. This produces about 2 to 5 million floating-point numbers per kilometer along a survey line. In 1979 alone, $10^{15}$ bits of seismic data were processed. One geophysical company in Houston has about 2 million magnetic reels of seismic data in inventory and 300,000 reels awaiting processing. The demand of cost-effective computers for seismic signal processing is increasing sharply.

**B. Reservoir modeling** Supercomputers are being used to perform three-dimensional modeling of oil fields. The reservoir problem is solved by using the finite difference method on the three-dimensional representation of the field. Geologic core samples are examined to project forward into time the field's expected performance. Presently at least 1000 flops needs to be processed per data point in the three-dimensional model of an oil field. This means a superpower computer must be employed to achieve an accurate performance evaluation in a reasonable time period for a large field.

Due to the importance of the Prudhoe Bay oil field, SOHIO Petroleum Company has constructed a numerical simulator for the whole field on a vector computer (Cyber-203). The field is about 168 meters thick and has been subdivided into 12 layers. With an aerial grid no finer than 160 acres, a model of 16,421 active subsurface blocks can cover 1000 oil wells. The finite difference equations for the reservoir model are solved iteratively on the Cyber-203. A simulated year requires 33 minutes of computer time. The success in this large modeling is attributed to both high speed and the large main memory built into a supercomputer.

**C. Plasma fusion power** Nuclear fusion researchers are pushing to use a computer 100 times more powerful than any existing one to model the plasma dynamics in the proposed Tokamak fusion power generator. Magnetic fusion research programs are being aided by vector supercomputers at the Lawrence Livermore National Laboratory and at Princeton's Plasma Physics Laboratory. The potential for magnetic fusion to provide an alternate source of energy has become closer as a result of the cooperative effort of the experimental program with the computational simulation program.

Synthetic nuclear fusion requires the heating of plasma to a temperature of 100 million degrees. This is a very costly effort. The high-temperature plasma, consisting of positively charged ions and negatively charged electrons, must be

magnetically confined. The United States National Magnetic Fusion Energy Computer Center is currently using two Cray-1's and one CDC-7600 to aid the controlled plasma experiments. Supercomputers have become an indispensable tool in magnetic fusion energy exploration.

**D. Nuclear reactor safety** Nuclear reactor design and safety control can both be aided by computer simulation studies. These studies attempt to provide for:

- On-line analysis of reactor conditions
- Automatic control for normal and abnormal operations
- Simulation of operator training
- Quick assessment of potential accident mitigation procedures

The importance lies in the above operations being done in real time. For light reactor safety analysis, a TRAC code has been developed to simulate the non-equilibrium, nonhomogeneous flow of high-temperature water and steam. Another code, Simmer II, has been developed to analyze core melting in a fast breeder reactor. Only supercomputers can make these calculations possible in real time.

### 1.5.4 Medical, Military, and Basic Research

In the medical area, fast computers are needed in computer-assisted tomography, artificial heart design, liver diagnosis, brain damage estimation, and genetic engineering studies. Military defense needs to use supercomputers for weapon design, effects simulation, and other electronic warfare. Almost all basic research areas demand fast computers to advance their studies.

**A. Computer-assisted tomography** The human body can be modeled by *computer-assisted tomography* (CAT) scanning. The Mayo Clinic in Rochester, Minnesota, is developing a research CAT scanner for three-dimensional, stop-action, cross-action viewing of the human heart. At the Courant Institute of Mathematical Sciences, research scientists are seeking an array processor for time-sequence, three-dimensional modeling of blood flow in the heart, with the goal of pursuing the artificial heart. Similar approaches can be applied to reveal the secrets of other human organs in real time.

Cross-sectional CAT images used to take 6 to 10 minutes to generate on a conventional computer. Using a dedicated array processor, the processing time can be reduced to 5 to 20 s. The image reconstruction of human anatomy in present CAT scanners is two-dimensional. It is generated too slowly (5 s) to freeze the motion of organs such as the heart or the lungs. The Mayo Clinic's super CAT scanner is expected to have 2000 to 3000 megaflops speed. It will produce three-dimensional images of the beating heart, within a few seconds, with 60 to 240 thin adjacent cross sections stacked one upon the other. Because of the short processing and exposure time, three-dimensional and stop-motion pictures of a beating heart will be possible for the first time. Dye injection may be used to trace the blood flow.

**B. Genetic engineering** Biological systems can be simulated on supercomputers. Genetic engineering is advancing rapidly in recent years. There is a growing need for large-scale computations to study molecular biology for the synthesis of complex organic molecules, such as proteins. Crystallography also can be aided by computer processing.

A highly pipelined machine, called the Cytocomputer, has been developed at the Michigan Environmental Research Institute for biomedical image processing. It can be used to search for genetic mutations. Sophisticated biomedical and computer techniques are being applied to derive an accurate estimate of the mutation rate for the human species. Gel matching between the father, mother, and child is done in the Cytocomputer using some parallel graph-matching techniques.

**C. Weapon research and defense** So far, military research agencies have used the majority of the existing supercomputers. In fact, the first Cray-1 was installed at the Los Alamos Scientific Laboratories in 1976. By 1981, four upgraded Cray 1's had been acquired by Los Alamos. Listed below are several defense-related military applications of supercomputers.

- Multiwarhead nuclear weapon design (Cray-1)
- Simulation of atomic weapon effects by solving hydrodynamics and radiation problems (Cyber-205)
- Intelligence gathering, such as radar signal processing on the associative processor for the antiballistic missile (ABM) program (PEPE)
- Cartographic data processing for automatic map generation (Staran)
- Sea surveillance for antisubmarine warfare (the S-1 multiprocessor)

**D. Basic Research Problems** Many of the aforementioned application areas are related to basic scientific research. Below are several additional areas that demand the use of supercomputers:

1. Computational chemists solve problems on quantum mechanics, statistical mechanics, polymer chemistry, and crystal growth.
2. Computational physicists analyze particle tracks generated in spark chambers, study fluid dynamics, examine quantum field theory, and investigate molecular dynamics.
3. Electronic engineers solve large-scale circuit equations using the multilevel Newton algorithm, and lay out VLSI connections on semiconductor chips.

## 1.6 BIBLIOGRAPHIC NOTES AND PROBLEMS

Parallel processing computers have been treated in parts of the books by Hayes (1978), Kuck (1978), Stone (1980), and Baer (1980). Enslow (1974) and Satyanarayanan (1980) devoted their books to multiprocessor systems. The book by Hockney and Jesshope (1982) covers only pipeline and array processors

The introductory material on data flow computers is based on Dennis (1980). A recent survey of vector processing computers can be found in Hwang, et al. (1981). Additional material on supercomputer applications can be found in Rodrique, et al. (1980) and Sugarman (1980). Bode and Händler (1980, 1982) have written two computer architecture books in German.

Other surveys on parallel processing and supercomputer systems appeared in Kuhn and Padua (1981), IEEE *Computer Magazine* (Nov. 1981), and *Communications of ACM* (Jan. 1978). The computer architectural classifications are based on Flynn (1966), Feng (1972), and Händler (1977). Interested readers should regularly check the proceedings of the *Annual Symposium on Computer Architecture*, and of the *International Conference on Parallel Processing* for frontier research development. The *Journal of Parallel/Distributed Computing* is a dedicated publication in this area.

## Problems

**1.1** Distinguish among computer terminologies in each of the following groups:

(*a*) Data processing, information processing, knowledge processing, and intelligence processing.

(*b*) Batch processing, multiprogramming, time sharing, and multiprocessing.

(*c*) Parallel processing at the job level, the task level, the interinstruction level, and the intrainstruction level.

(*d*) Uniprocessor systems versus multiprocessor systems.

(*e*) Parallelism versus pipelining.

(*f*) Serial processing versus parallel processing.

(*g*) Control flow computers versus data flow computers.

**1.2** Existing computer systems are classified in Tables 1.3, 1.4, and 1.5, based on the three architectural specification schemes given in Section 1.4. The listing in each table is not complete. Enter the specification of at least two additional computer systems under each architectural category of each of the three tables. Use the same specification format for the existing entries in making the new entries.

**1.3** The speedup of using $n$ processors over the use of one processor in solving a computing problem was analyzed in Section 1.3.4 under various assumptions, such as $f_i = 1/n$ and $d_i = 1/i$ for $i = 1, 2, \ldots, n$.

(*a*) Repeat the performance speedup analysis to derive a new speedup equation (similar to Eq. 1.8), under the following new probability distributions of operating modes.

$$f_i = \frac{i}{\sum_{i=1}^{n} i} \qquad \text{for } i = 1, 2, \ldots, n \tag{1.17}$$

(*b*) Repeat part (*a*) for another probability distribution:

$$f_i = \frac{n - i - 1}{\sum_{i=1}^{n} i} \qquad \text{for } i = 1, 2, \ldots, n \tag{1.18}$$

(*c*) The case in (*a*) favors the assignment of the computing task to a larger number of processors, whereas the case in (*b*) favors the assignment to a smaller number of processors. The case presented in Section 1.3.4 treats all possible task divisions equally. Plot the new speedup curves obtained in case (*a*) and in case (*b*) along with plots given in Figure 1.14. Can you find new upper bounds for the new speedup curves? Derive the upper bound, if it exists.

**1.4** Name three distinct characteristics that exist in the $i$th generation computers for $i = 1, 2, 3$, and 4 but not in the $j$th generation for $j = 0, 1, 2, \ldots, i - 1$, where the 0th generation corresponds to prior electronic computers.

**1.5** Match each of the following computer systems to the phrase that best describes it.

| | |
|---|---|
| ____ Illiac-IV | (1) A cluster of microprocessors |
| ____ TI-ASC | (2) A vector processor made in Japan |
| ____ CDC-7600 | (3) A supermini computer with virtual memory |
| ____ IBM 360/91 | (4) The first MIMD multiprocessor consisting of 16 PDP 11 minicomputers |
| ____ AP-120B | (5) The first IBM computer using the thermal conduction modules |
| ____ Cray-1 | (6) A multiprocessing vector processor by Cray Research |
| ____ B-5500 | (7) A major computer project at IBM in the 1960s |
| ____ PEPE | (8) An array processor with 64 PEs |
| ____ Cyber-205 | (9) A multifunction computer with multiprocessing in I/O subsystem |
| ____ C.mmp | (10) The first operational electronic digital computer |
| ____ BSP | (11) An associate processor with 288 PEs |
| ____ MPP | (12) A commercial multiprocessor with a packet switched interconnection network |
| ____ Cray X-MP | (13) The first IBM scientific processor with multiple functional units |
| ____ HEP | (14) An attached array processor for minicomputers |
| ____ VP-200 | (15) A first-generation pipelined vector processor |
| ____ ENIAC | (16) An array processor with 16384 PEs |
| ____ Stretch | (17) A CDC vector processor enhanced from the STAR-100 |
| ____ C.m* | (18) One of the first stack computers |
| ____ VAX 11/780 | (19) An array processor with 16 PEs and shared memories |
| ____ IBM 3081 | (20) A vector processor with 12 pipes and large register files |

**1.6** You were briefed about 15 important applications of parallel processing computers in Section 1.5. Choose the one of these application areas that interests you most for an indepth study. Dig out more information from the library or request the source information from any application site of supercomputers that you know of. Prepare a study report based on your readings and observations in the chosen area of supercomputer applications.

**1.7** In the following block of computations, $a$ and $b$ are two external inputs and $z$ is the final output. Two intermediate results are labelled $x$ and $y$.

$$x \leftarrow a * a; \quad y \leftarrow b * b; \quad z \leftarrow (x + y)/(x - y)$$

(*a*) Draw a data flow graph for this code block, where $*$, $+$, $-$, and / are arithmetic operators.
(*b*) Show a template implementation of the data flow graph in (*a*).
(*c*) Indicate the events that can be done in parallel in the execution of the above block of codes.

**1.8** Describe at least four characteristics of MIMD multiprocessors that distinguish them from multiple computer systems or computer networks.

**1.9** Prove that a $k$-stage linear pipeline can be at most $k$ times faster than that of a nonpipelined serial processor.

**1.10** Summarize all forms of parallelism that can be exploited at different processing levels of a computer system, including both uniprocessor and multiprocessor approaches. Discuss hardware, firmware, and software supports needed to achieve each form of parallelism. Indicate example computers that have achieved various forms of parallelism.

A-66859

CHAPTER

# TWO

# MEMORY AND INPUT-OUTPUT SUBSYSTEMS

In this chapter, we describe memory organizations and input-output subsystems. material needed to study subsequent chapters. Memories are organized in a hierarchical order of access times. The basic techniques used to create a large virtual address space and the necessary translation mechanisms to the physical space are discussed. Some memory allocation and management schemes are presented for multiprogrammed systems. Various organizations of cache memories are presented. Techniques for estimating the effective bandwidth of such memories are developed. Finally, techniques for exploiting concurrency in input-output subsystems are summarized. Memory and I/O subsystems are needed in uniprocessors, pipeline machines, array processors, and multiprocessors. Special parallel memory structures for each class of machines are treated separately in their respective chapters.

## 2.1 HIERARCHICAL MEMORY STRUCTURE

Memory systems for parallel processor computers are described in this section. We begin with the hierarchical memory structures and the concept of virtual memory. Virtual memory concepts are discussed for paged systems, segmented systems, and systems with paged segments.

### 2.1.1 Memory Hierarchy

The design objectives of hierarchical memory in a parallel processing system and a multiprogrammed uniprocessor system are basically the same. The objectives are to attempt to match the processor speed with the rate of information transfer or the *bandwidth* of the memory at the lowest level and at a reasonable cost. However, one major difference exists in the hierarchical memory structures of the two

systems. This difference is due to the memory reference characteristics of multiprogrammed uniprocessors and parallel processors. In the latter case, the existence of multiple processors necessitates the arrival of concurrent memory requests to memory at the same level of the hierarchy. If two or more of these concurrent requests reference the same section of memory at the same level, a *conflict* is said to occur, which could degrade the performance of the system. Hence, memory for a parallel processing system must be organized to reduce the potential conflicts at each level of the hierarchy. This is usually done by partitioning the memory at a given level into several modules so that some degree of concurrent access can be achieved.

Memories in a hierarchy can be classified on the basis of several attributes. One common attribute is the accessing method, which divides the memories into three basic classes: *random-access memory* (RAM), *sequential-access memory* (SAM), and *direct-access storage devices* (DASDs). In RAM, the access time $t_a$ of a memory word is independent of its location. In SAMs, information is accessed serially or sequentially, as in shift-register memory such as a first-in, first-out (FIFO) buffer, *charged-coupled devices* (CCDs), and *magnetic bubble memories* (MBMs). DASDs are rotational devices made of magnetic materials where any block of information can be accessed directly. The DASDs are accessed via special interfaces called channels, which are discussed in Section 2.5.

Another attribute often used to classify memory is the speed or access time of the memory. In most computer systems, the memory hierarchy is often organized so that the highest level has the fastest memory speed and the lowest level has the slowest speed. On the basis of access time, memory can be further classified into primary memory and secondary memory. Primary memory is made of RAMs and secondary memories are made of DASDs and optional SAMs. In characterizing the access times of memories in the hierarchy we will concentrate on RAMs and DASDs.

The three most common DASDs are drums, fixed-head disks, and moveable-arm disks. For these cases, the time to transfer a block of information is $t_a + t_B$, where $t_a$ is the access time and $t_B$ is the block-transfer time. For drums and fixed-head disks, $t_a$ is the time it takes for the initial word of the desired block to rotate into position. For moveable-arm disks, an additional "seek time" $t_s$ is required to move the arms into track position. Table 2.1 depicts some of the characteristics of the different memories used in a hierarchy.

In general, the memory hierarchy is structured so that memories at level $i$ are "higher" than those at level $i + 1$. If $c_i$, $t_i$, and $s_i$ are respectively the cost per byte, average access time, and the total memory size, at level $i$, the following relationships normally hold between levels $i$ and $i + 1$: $c_i > c_{i+1}$, $t_i < t_{i+1}$, and $s_i < s_{i+1}$, for $i \geq 1$. Figure 2.1 illustrates the typical relative cost-access time relationship of some memory technologies.

Figure 2.2 illustrates an example of a two-processor system with a three-level memory. Memory module $M_{1,j}$ is the *local* or *private* memory of processor $j$ since it is exclusively used by that processor. The local memory is often implemented as a high-speed buffer or *cache* memory using bipolar technology and hence is the

**Table 2.1 Characteristics of memory devices in a memory hierarchy**

| Level $i$ | Memory type | Technology | Typical size $s_i$ | Average access time $t_i$ | Unit of transfer |
|---|---|---|---|---|---|
| 1 | Cache | Bipolar, HMOS, ECL | 2K–128K bytes | 30–100 ns | 1 word |
| 2 | Main or primary memory | MOS core | 4K–16M bytes | 0.25–1 μs 0.5–1 μs | 2–32 words |
| 3 (optional) | Bulk memory (LCS, ECS) | Core | 64K–16M bytes | 5–10 μs | 2–32 words |
| 4 | Fixed head disk or drums | Magnetic | 8M–256M bytes | 5–15 ms | 1K–4K bytes |
| 5 | Moveable arm disk | Magnetic | 8M–500M bytes | 25–75 ms | 4K bytes |
| 6 | Tape | Magnetic | 50M bytes | 1–5 s | 1K–16K bytes |

fastest memory. The cache is used to capture the segments of information which are most frequently referenced by the processor. Information transfer between the processor and the cache is on a word basis. Cache memories will be discussed in detail in Section 2.3. The next lower level of memory consists of modules $M_{2,0}$ to $M_{2,3}$ and constitutes the main memory. The four modules are usually designed with *metal oxide semiconductor* (MOS) or ferromagnetic (core) technology, and the

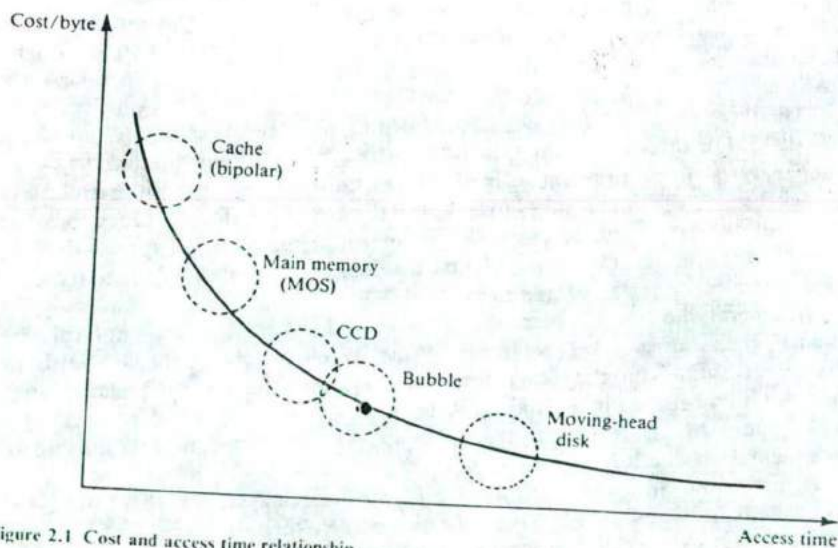



Figure 2.1 Cost and access time relationship.

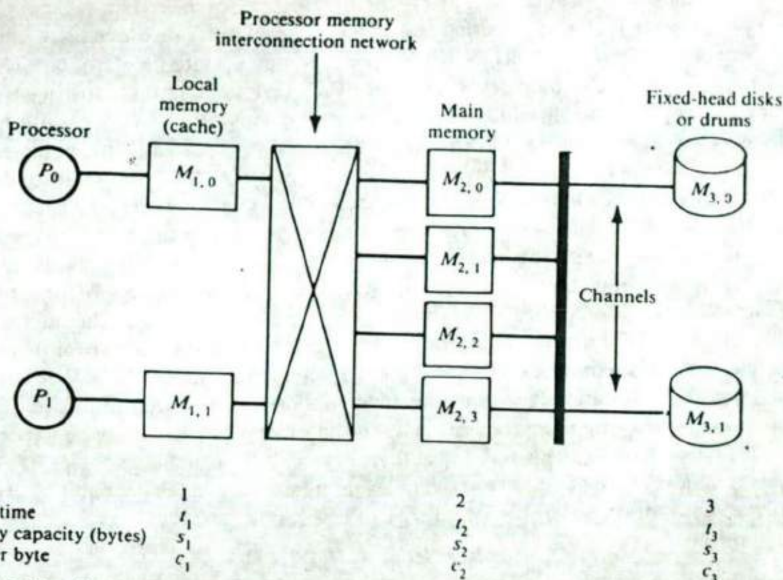



**Figure 2.2 Three-level memory hierarchy.**

unit of information transfer between the main memory and cache is a block of contiguous information (typically 2 to 32 words). The primary memory may be extended either with the so-called *large core storage* (LCS) or with *extended core storage* (ECS), both of which are made of slower core memories. The average access time of the primary memory and its extensions are in the order of 0.5 $\mu s$ and 5 $\mu s$, respectively.

There exists a technological gap between the primary and secondary memory, as evidenced by the access time characteristics shown in Table 2.1. Average access time of secondary memories is 1000 to 10,000 times slower than that of primary memories. Electronic disks, such as CCDs and MBMs, have not proved cost-effective in closing the technological gap and thus have had little impact in the design of memory systems. Hence, as shown in Figure 2.2, the secondary memories most often used are disks and drums.

The processor usually references an item in memory by providing the location or address of that item. A memory hierarchy is usually organized so that the address space in level $i$ is a subset of that in level $i + 1$. This is true only in their relation, however; address $A_k$ in level $i$ is not necessarily address $A_k$ in level $i + 1$, but any information in level $i$ may also exist in level $i + 1$. However, some of the information in level $i$ may be more current than that in level $i + 1$.

This creates a *data consistency* or *coherence* problem between adjacent levels because they have different copies of the same information. Usually level $i + 1$ is eventually updated with the modified information from level $i$. The data consistency

problem may also exist between the local memories or caches when two cooperating processes, which are executing concurrently or on separate processors, interact via one or more shared variables. One process may update the copy of a shared variable in its local memory while the other process continues to access the previous copy of the variable in its local memory. This situation may result in the incorrect execution of the cooperating processes. In general, a memory hierarchy encounters such a coherence problem as soon as one of its levels is split into several independent units which are not equally accessible from faster levels or processors. Solutions to data consistency problems are discussed in Sections 2.3, 2.5, and 7.3.

In modeling the performance of a hierarchical memory, it is often assumed that the memory management policy is characterized by a *success function* or *hit ratio* $H$, which is the probability of finding the requested information in the memory of a given level. In general, $H$ depends on the granularity of information transfer, the capacity of memory at that level, the management strategy, and other factors. However, for some classes of management policies, it has been found that $H$ is most sensitive to the memory size $s$. Hence the success function may be written as $H(s)$. The *miss ratio* or probability is then $F(s) = 1 - H(s)$. Since copies of information in level $i$ are assumed to exist in levels greater than $i$, the probability of a hit at level $i$ and of misses at higher levels 1 to $i - 1$, is:

$$h_i = H(s_i) - H(s_{i-1}) \tag{2.1}$$

where $h_i$ is the access frequency at level $i$ and indicates the relative number of successful accesses to level $i$. The missing-item fault frequency at level $i$ is then $f_i = 1 - h_i$.

### 2.1.2 Optimization of Memory Hierarchy

The goal in designing an $n$-level memory hierarchy is to achieve a performance close to that of the fastest memory $M_{1,j}$ and a cost per bit close to that of the cheapest memory $M_{n,j}$. The performance of the hierarchy may be indicated by the effective hierarchy access time per each memory reference. However, it should be noted that the performance depends on a variety of interrelated factors. These include the program behavior with respect to memory references, the access time and memory size of each level, the granularity of information transfer (block size), and the management policies. One other important factor that also affects the effective access time is the design of the processor-memory interconnection network, which is discussed in Sections 5.2 and 7.2.

The interrelation among some of these factors can be used to derive a criterion for optimizing the performance of the memory hierarchy. One performance measure is the effective memory access time. Another measure may include the utilization of the processor. The effective access time $T_i$ from the processor to the $i$th level of the memory hierarchy is the sum of the individual average access times $t_k$ of each level from $k = 1$ to $i$:

$$T_i = \sum_{k=1}^{i} t_k \tag{2.2}$$

In general, $t_k$ includes the wait time due to memory conflicts at level $k$ and the delay in the switching network between levels $k - 1$ and $k$. The degree of conflicts is usually a function of the number of processors, the number of memory modules, and the interconnection network between the processors and memory modules. In most systems, a request for a word which is not in memory level $i$ causes the block of information which contains the requested word to be transferred from level $i + 1$ to level $i$. When the block transfer to level 1 has been completed, the requested word is accessed in the local memory.

The effective access time for each memory reference in the $n$-level memory hierarchy is

$$T = \sum_{i=1}^{n} h_i T_i \tag{2.3}$$

Substituting $h_i$ and $T_i$ into Eq. 2.3,

$$T = \sum_{i=1}^{n} [H(s_n) - H(s_{i-1})]t_i \tag{2.4}$$

Assuming that there is a copy of all requested information in the lowest level $n$, $H(s_n) = 1$. In the derivation of Eq. 2.4, it is convenient to define $H(s_0) = 0$, hence $F(s_0) = 1$. Rewriting Eq. 2.4:

$$T = \sum_{i=1}^{n} [1 - H(s_{i-1})]t_i$$

Since $1 - H(s_{i-1}) = F(s_{i-1})$, we obtain

$$T = \sum_{i=1}^{n} F(s_i - 1)t_i \tag{2.5}$$

If $c(t_i)$ is the cost per byte of memory at level $i$ which is expressed as a function of its average access time, the total cost of the memory system is

$$C = \sum_{i=1}^{n} c(t_i)s_i \tag{2.6}$$

A typical memory-hierarchy design problem involves an optimization which minimizes the effective hierarchy access time $T$, subject to a given memory system cost $C_0$ and size constraints. That is, minimize $T = \sum_{i=1}^{n} F(s_{i-1})t_i$, subject to the constraints $C = \sum_{i=1}^{n} c(t_i)s_i \leq C_0$, where $s_i > 0$ and $t_i > 0$, for $i = 1, 2, \ldots, n$. In practice, the cost constraints should include the cost of the processor-memory interconnection network.

In the memory types we have discussed so far, the contents of a memory location is accessed by specifying the memory location or address of the item. In another type of memory, *associative memory*, the data stored in the memory can be accessed by specifying the contents or part of the contents. In this sense, associative memory has also been known as *content-addressable memory* and *parallel search memory*. The major advantage of associative memory over the RAM is its capability of

performing parallel search and comparison operations, which are needed in many important applications, such as table lookup, information storage and retrieval of rapidly changing databases, radar-signal tracking and processing, image processing, and real-time artificial intelligence computations. The major disadvantage of associative memory is its much increased hardware cost. Currently, associative memories are much more expensive than RAMs, even though both are built with integrated circuitry. However, with the rapid advent of VLSI technology, the price gap between these types of memories may be reduced in the future. Associative memories and associative processors will be treated in Section 5.4.

### 2.1.3 Addressing Schemes for Main Memory

In a parallel processing environment, main memory is a prime system resource which is normally shared by all the processors or independent units of a pipelined processor. Care must be taken in the organization of the memory system to avoid severe performance degradation because of memory interference caused by two or more processors simultaneously attempting to access the same modules of the memory system. It would be undesirable to have one monolithic unit of memory to be shared among several processors, as this would result in serious memory interference. Hence, the main memory is partitioned into several independent memory modules and the addresses distributed across these modules. This scheme, called *interleaving*, resolves some of the interference by allowing concurrent accesses to more than one module. The interleaving of addresses among $M$ modules is called *$M$-way* interleaving.

There are two basic methods of distributing the addresses among the memory modules. Assume that there are a total of $N = 2^n$ words in main memory. Then the physical address for a word in memory consists of $n$ bits, $a_{n-1}a_{n-2} \cdots a_1a_0$. One method, *high-order* interleaving, distributes the addresses in $M = 2^m$ modules so that each module $i$, for $0 \leq i \leq M - 1$, contains consecutive addresses $i2^{n-m}$ to $(i + 1)2^{n-m} - 1$, inclusive. The high-order $m$ bits are used to select the module while the remaining $n - m$ bits select the address within the module, as depicted in Figure 2.3.

The second method, *low-order* interleaving, distributes the addresses so that consecutive addresses are located within consecutive modules. The low-order $m$ bits of the address select the module, while the remaining $n - m$ bits select the address within the module, as shown in Figure 2.4. Hence, an address $A$ is located in module $A$ mod $M$.

The two schemes depicted in Figures 2.3 and 2.4 represent extremes in the choice of the address decoding. The first scheme permits easy memory expansion by the addition of one or more memory modules as needed to a maximum of $M - 1$. However, the placement of contiguous memory addresses within a module may cause considerable memory conflicts in the case of pipelined, vector, or array (SIMD) processors. The sequentiality of instructions in programs and the sequentiality of data in vector processors cause consecutive instructions or data to be in the same module. Since memory cycle time is much greater than the pipeline clock

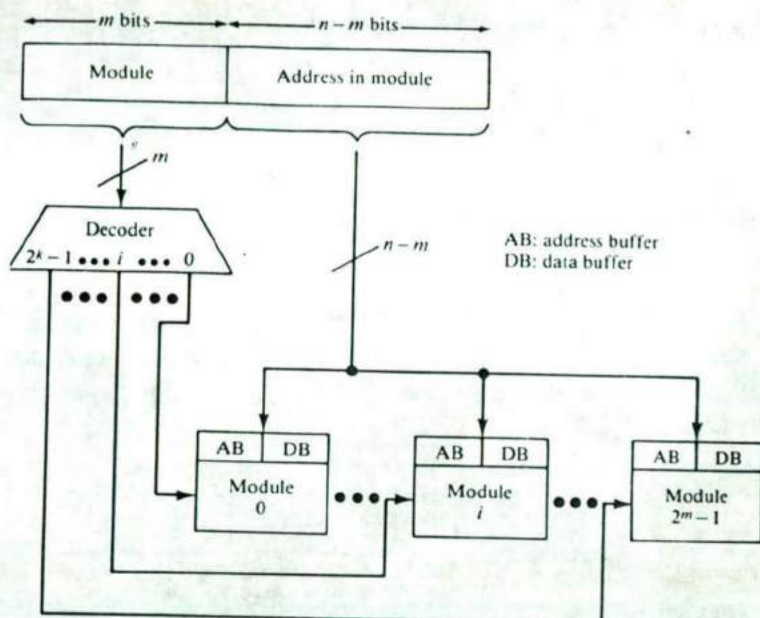



**Figure 2.3 Parallel memory system with consecutive words in a module.**

time, a previous memory request would not have completed its access before the arrival of the next request, thereby resulting in a delay.

In array processors, if the data elements of a vector reside in the same module, there will be insignificant parallelism in computation because the elements cannot be fetched simultaneously by all processors for the "lock-step" manipulation. The high-order interleaving can be used without conflict problems in multiprocessors if the modules are partitioned according to disjoint or noninteracting processes. In practice, however, processes interact and share instructions and data in multiprocessor systems and will thereby encounter considerable conflicts in a high-order interleaved memory subsystem. For the above reasons, low-order interleaving is frequently used to reduce memory interference.

An advantage of high-order interleaving is that it provides better system reliability, since a failed module affects only a localized area of the address space and therefore provides graceful degradation in performance. The failed module can be logically isolated from the system and the memory manager can be informed so that no process address space is mapped into the failed module. A failure of any single module in the second scheme will almost certainly be catastrophic to the whole system. The second scheme, however, seems preferable if memory interference is the only basis of choice.

A compromise interleaving technique is to partition the module address field into the two sections $S_{m-r}$ and $S_r$ so that section $S_r$ is the least significant

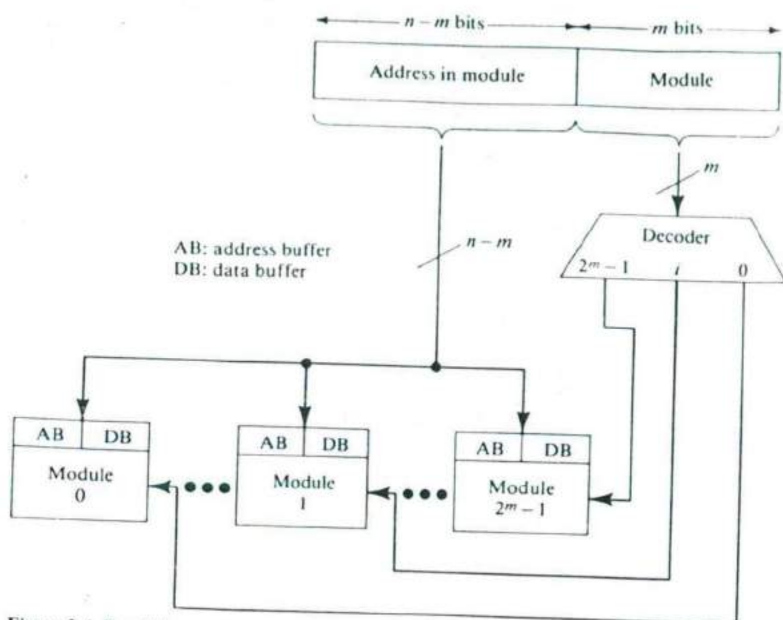



Figure 2.4 Parallel memory system with consecutive words in consecutive modules.

$r$ bits of the memory address and section $S_{m-r}$ is the high-order $m$-$r$ bits of the address. Notice that the module address is formed by the concatenation of section $S_{m-r}$ and $S_r$. In this scheme, the addresses are interleaved among groups of $2^r$ memory modules. This tends to reduce memory interference to a segment of shared data. The memory system is expandable in blocks of $2^r$ modules; however, a single module failure disables an entire block of $2^r$ modules. This scheme is appealing for systems with a large number of memory modules if $r$ is chosen to be very small.

## 2.2 VIRTUAL MEMORY SYSTEM

In many computer systems, programmers often realize that some of their large programs cannot fit in main memory for execution. Even if there is enough main memory for one program, the main memory may be shared with other users, causing any one program to occupy some fraction of memory which may not be sufficient for the program to execute. The usual solution is to introduce management schemes that intelligently allocate portions of memory to users as necessary for the efficient running of their programs. The use of virtual memory to achieve this goal is described in this section.

### 2.2.1 The Concept of Virtual Memory

Memory management is distributed over several overlapping phases. It begins with the program structure and design, the naming function performed by the compiler in translating the program modules from programming language into modules of machine code or *unique identifiers*. A linker then combines these modules of unique identifiers and the composite is translated by a loader into main memory locations. The set of unique identifiers defines the *virtual space* or the *name space* and the set of main memory locations allocated to the program defines the *physical memory space*. The last phase of memory management is the dynamic memory management required during the execution of the program.

In earlier computers, when the entire program would not fit into memory space at one time, a technique called *overlay* was used. Phases of the program were brought into memory when needed, overlaying those that were no longer needed.

Memory requirements of some programs are difficult to predict, a factor that influences memory management during executions. Another and perhaps the strongest influence is that high-performance computer systems are often operated in multiprogramming mode. The result of these influences is that the fraction of main memory which is assigned to any one program is unpredictable outside the execution environment. Only at execution time are physical addresses assigned, for only then are the total memory size, the currently unused memory space, and the sizes of the various routines from called libraries known.

Virtual memory gives programmers the illusion that there is a very large memory at their disposal, whereas the actual (physical) memory available may be small. This illusion can be accomplished by allowing the programmer to operate in the name space while the architecture provides a mechanism for translating the program-generated (virtual) addresses (during execution) into the memory-location addresses. In multiple processor systems with virtual memory, this mechanism must be provided for each processor. Assume that the name space $V_j$ generated by the $j$th program running on a processor consists of a set of $n$ unique identifiers. Hence

$$V_j = \{0, 1, \ldots, n - 1\}$$

Assume that the memory space allocated to the program in execution has $m$ locations. This space can be represented as a sequence of addresses:

$$M = \{0, 1, \ldots, m - 1\}$$

since main memory can be regarded as a linear array of locations, where each location is identified by a unique memory address. Also, since the allocated memory space may vary with program execution, $m$ is a function of time.

At any time $t$ and for each referenced name $x \in V_j$, there is an *address map*

$$f_j(t): V_j \rightarrow M \cup \{\phi\}$$

which identifies a mapping between names and memory addresses at instant $t$ so as to *bind* them. The function $f_j(t)$ is defined by

$$f_j[x, t] = \begin{cases} y & \text{if at time } t \text{ item } x \text{ is in } M \text{ at location } y \\ \phi & \text{if at time } t \text{ item } x \text{ is missing from } M \end{cases}$$

When $f_j[x, t] = \phi$, an *addressing exception* or *missing item* fault is said to occur, which causes a fault handler to bring in the required item from the next lower level of memory. The fault handler also updates the $f_j$ map to reflect the new binding of names to memory addresses. In a general hierarchy, the missing item is retrieved by sending a memory request for the item to successive lower levels until it is found in a level, say $k$. Three basic policies define the control of the transfer of the missing item from a lower level to the desired level. A *placement* policy selects a location in memory where the fetched item will be placed. Where the memory is full, a *replacement* policy chooses which item or items to remove in order to create space for the fetched item. A *fetch policy* decides when an item is to be fetched from lower level memory. These policies and their impact on memory management will be discussed fully in Section 2.3.

**Program locality** The sequence of references made by the $j$th program in execution can be represented by a *reference string* $R_j(T) = r_j(1)r_j(2) \ldots r_j(T)$, where $r_j(t) \in V_j$ is the $t$th virtual address generated by process $j$. It is common knowledge that the virtual addresses generated are nonrandom but behave in a somewhat predictable manner. Such characteristics of programs are due to looping, sequential and block-formatted control structures inherent in the grouping of instructions, and data in programs. These properties, referred to as the *locality of reference*, describe the fact that over an interval of virtual time, the virtual addresses generated by a typical program tend to be restricted to small sets of its name space, as shown in Figure 2.5. For example, if one considers the interval $\Delta$ in Figure 2.5, the subset of pages referenced in that interval is less than the set of pages addressable.

There are three components of the locality of reference, which coexist in an active process. These are *temporal*, *spatial*, and *sequentiality* localities. In temporal locality, there is a tendency for a process to reference in the near future the elements of the reference string referenced in the recent past. Program constructs which lead to this concept are loops, temporary variables, or process stacks. In spatial locality there is a tendency for a process to make references to a portion of the virtual address space in the neighborhood of the last reference. The principle of sequentiality states that if the last reference was $r_j(t)$, then there is a likelihood that the next reference is to the immediate successor of element $r_j(t)$. Traversals of a sequential set of instructions and arrays of data enforce spatial and sequentiality localities. It should be noted that each process exhibits an individual characteristic with respect to the three types of localities.

Each type of locality aids or influences the characterization of an efficient memory hierarchy. The principle of spatial locality permits us to determine the size of the block to be transferred between levels. The principle of temporal

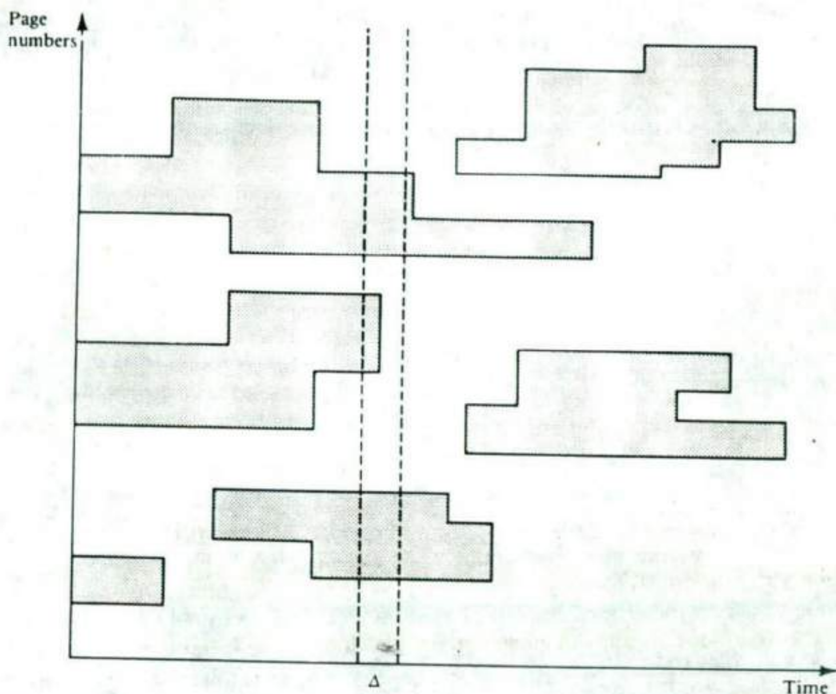



**Figure 2.5 An example of a page reference map for a process.**

locality aids in identifying the number of blocks to be contained at each level. Sequentiality locality permits the distribution of the unique identifiers to concurrently operating devices at certain levels of the hierarchy for concurrent accesses.

From Figure 2.5, if the reader considers a hypothetical interval time window $\Delta$ which moves across the virtual time axis, it can be seen that only a subset of the virtual address space is needed during the time interval of the history of the process. The subset of virtual space referenced during the interval $t$, $t + \Delta$ is called the *working set* $W(t, \Delta)$. During the execution of a process, the working set quickly accumulates in the highest level of the memory hierarchy to reduce the effective memory access time of a reference. In general, the time window $\Delta$ is a critical parameter which may be chosen to optimize the working set of the process over its lifetime.

**Program relocation** During the execution of a program, the processor generates logical addresses which are mapped into the physical address space in the main memory. The address mapping, considered as the function $f_j\colon V_j \rightarrow M$, is performed both when the program is initially loaded and during the execution of the

program. The former case is called *static relocation*; the latter is called *dynamic relocation*. Static relocation makes it difficult for processes to share information which is modifiable during execution. Furthermore, if a program is displaced from main memory by mapping, it must be reloaded into the same set of memory locations, thereby fixing or binding the physical address space of the program for the duration of the execution. This constraint causes inefficient memory management policies. Multiprogramming systems do not generally use static relocation because of these and other disadvantages. In order to effectively utilize memory resources, dynamic relocation is often used, in which the function $f_i$ is varied during the execution of the programs.

One technique in performing dynamic relocation is to use a set of base or relocation registers in which the content of a relocation register is added to the virtual address at each memory access. In this case, the programs may be initially loaded into memory using static relocation, after which they may be displaced within memory and the contents of the relocation register adjusted to reflect the displacement. Two or more processes may share the programs by using different relocation registers.

**Address map implementation** The address map $f$ can be implemented in several ways. The simplest implementation, *direct mapping*, is a table with $n$ entries so that the $x$th entry contains $y$ whenever $f(x) = y$ and is null ($\phi$) otherwise. The time to access the element identified by $x$ involves an additional memory access; the time to look up the $x$th entry in the table. If the table is implemented in main memory, the effective access time of the element may be intolerable. Fast registers may be used at a great expense. Since the virtual space size $n$ may be much greater than the physical space size $m$ in practice, the table would contain $n - m$ null entries. Even if we created a table with only $m$ entries, the execution time variation of $m$ may present some management problems, as we shall see later. Another implementation, *associative mapping*, uses an *associative memory* (AM) that contains those pairs $(x, y)$ for which $f(x) = y$ and the search is by content. Since the search time in an AM increases with an increase in the number of entries, a small high-speed buffer is often used. This buffer, often called the *translation lookaside buffer* (TLB), maintains the mapping between recently used virtual and physical memory addresses.

The implementation discussed above for the address map is still impractical because the virtual memory size $n$ is usually too large, so that even the locality set of the program cannot be stored in a practical AM. In the following three sections, we examine methods that result in considerable reduction in the amount of mapping information that must be stored. Each method groups information into nonoverlapping blocks, so that the entries in the address may refer to blocks instead of individual addresses in the address space. The first method organizes the address space into blocks of fixed size, called a *page*. The second method organizes the name space into blocks of arbitrary size, called a *segment*. In the third method, we combine paging and segmentation.

**Virtual memory in perspective** The principle of locality of reference has proved virtual memory to be effective for the given access times and costs. That is, users have been willing to accept the overhead and burden of a page management system in order to have the benefits of an apparently large memory. If the cost of a large memory is so inexpensive that the user is willing to buy it in the first place regardless of the inefficiency in its use because the references are local, then virtual memory may be unimportant. The microcomputers used in offices and small businesses probably fit in this category today and will certainly be in this category when 256K and possibly 1M random-access memory chips are in high production.

At this point, it may be more effective to have large real memory for a small computing system than virtual memory in a two-level system, although paging hardware for automatic relocation will still be useful. Whereas virtual memory was present on the majority of interactive, time-shared systems in the 1970s, it may disappear from use on small personal systems. However, virtual memory will continue to be used in many large systems, such as large database systems or computing facilities, where the program-size requirements are extremely large and do not fit into real memory at an economical cost.

### 2.2.2 Paged Memory System

In this scheme, the virtual space is partitioned into pages, which can be resident in matching size blocks (called *page frames*) in memory. Each virtual address that is generated by a program in execution consists of two fields: a virtual page number $i_p$, which is the mapped field, and the displacement $i_w$ of the word within the page, which is the unmapped field. The address map consists of a *page table* (PT), from which is read the corresponding base address of the page frame if the page exists in the main memory. The simplest page table may contain one entry for each possible virtual page.

There is one page table for each process, and the page table is created in main memory at the initiation of the process. A page table base register (PTBR) in each processor contains the base address of the page table of the process that is currently running on that processor. The page table entry may be accessed by indexing into the page table array. Figure 2.6 shows how the page table is used by direct mapping to implement the mapping of a virtual address to a physical address. Each *page table entry* (PTE) consists of a *valid bit* ($F$), a *permissible access code* (RWX), a *memory-disk bit* ($M$) and a *page-frame address* (PFA).

The valid bit, if set, indicates that the page *exists*, or is *nonnull*. A page which is null (valid bit cleared) would have to be created when referenced. A page is said to be active with respect to a process if it is resident in main memory. The memory bit ($M$) flag is set in the page table entry of that process and the PFA field of the PTE contains the address of the page in memory. In contrast, a nonnull page is inactive with respect to a process if the memory bit ($M$) is cleared. Then the PFA field of the PTE contains the disk address of the page.

The page table mapping mechanism is rather inefficient since it requires two memory accesses for each data accessed. This may be improved by using a fast

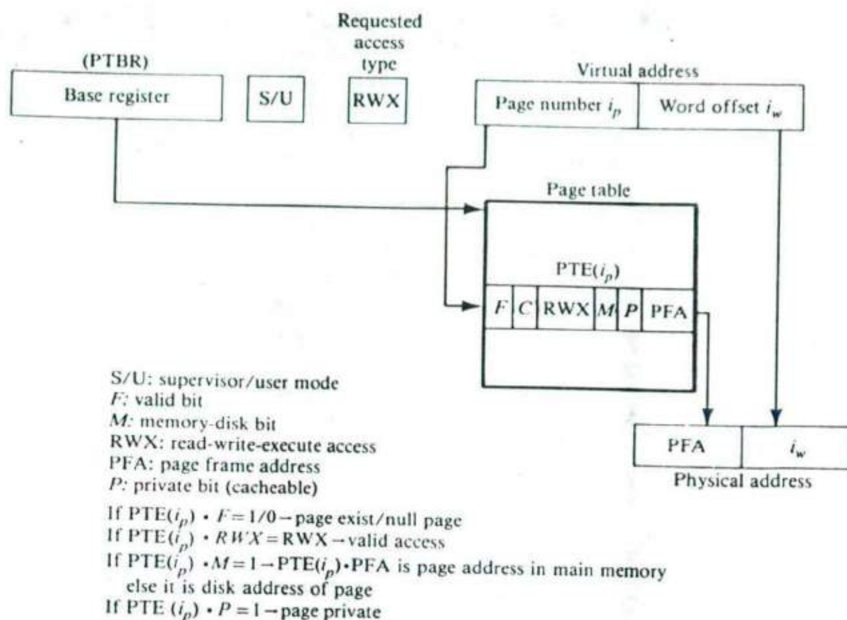



**Figure 2.6 Virtual to real page address translation.**

random-access memory or register set to store the page table. For example, the Xerox Sigma 7 processor has a 256 register-set of nine bits each and a page size of 512 words. This corresponds to a virtual memory of $2^{17}$ words (128 K). A better solution is to exploit the locality of reference property of programs and use an associative map which consists of an $N$-entry translation lookaside buffer (TLB). Hence the TLB may contain the $N$ most recently accessed virtual page numbers and their corresponding page-frame addresses.

In a system with a single virtual address space, all users reside in the same virtual memory. Another method is to partition the virtual space into several independent areas, allocating one to each active process. This can be accomplished by using the high-order bits of the virtual page number as a process identification. These bits with the PTBR can be used to select the page table of a process.

Yet another technique of maintaining multiple virtual address spaces is to fix the virtual space and concatenate a system-generated process identification with the virtual address. This is illustrated in Figure 2.7. For a multiprogrammed processor, a page map entry typically consists of six fields: a virtual page number $i_p$, a process identification, the RWX, a modified bit ($C$), and the PFA in shared memory. The process identification of the currently running process is in the current process register (CPR) of the processor.

When a virtual address is generated by a running process, the virtual-to-real address translation involves the associative comparison of the virtual page number

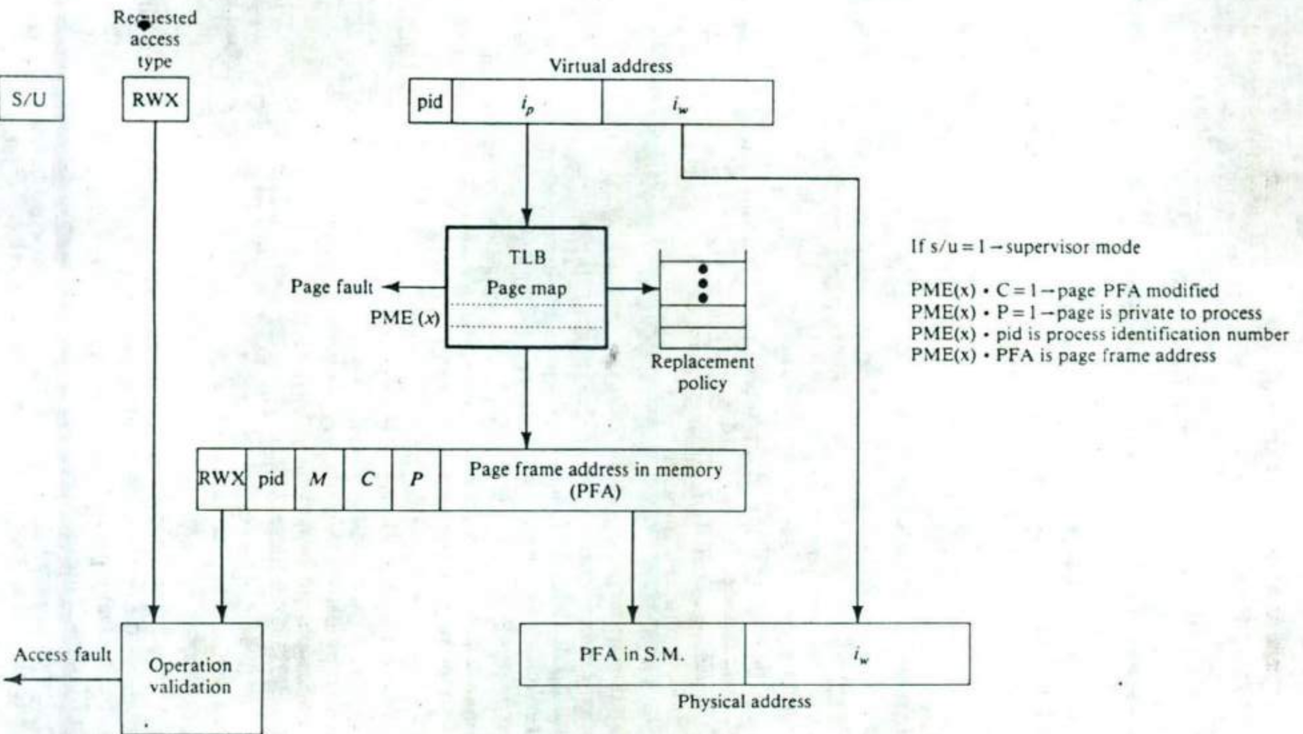



Figure 2.7 Virtual to real address translation using page map.

$i_p$ with all the *page map entries* (PME) that contain the same process identification as the current running process. If there is a match, the page-frame number is retrieved and the physical address formed by concatenating the displacement with the PFA. If there is no match, a page fault interrupt occurs, which is serviced to locate the page. Moreover, if the page-access key presented by the virtual address does not match the RWX field of the PME with a corresponding virtual page number and PID, an access violation is trapped. When a referenced page is modified, the modified bit C of the corresponding PME is set in the page map. This bit may be used by the replacement and memory update policies.

When a page fault occurs because the virtual page number $i_p$ was not found in the TLB, a *dynamic address translation* is requested, using the page table which is resident in main memory. The virtual page number $i_p$, used as an index, is added to the page table address in the PTBR and the resulting address is used to access the PTE as described earlier. If the PTE indicates that the page is not in main memory, the running process is *blocked* or suspended. A *context switch* is then made to another ready-to-run process while the page is transferred from drum or disk to memory and the PTE entry updated. The page address on disk or drum may be found in the address field of the PTE. The context or task switch involves the saving of the state of the faulting process and restoring the state of the runnable process in the processor.

The TLB is invalidated or its contents are saved in memory as part of the faulting process. The task switch is made because the page-transfer operation is slow compared to the processor speed. If the page is in memory, the TLB is updated with the virtual page number and the page's page-frame address pair before the process resumes execution. Updating the TLB involves replacing one of its entries if it is full. The entry chosen for replacement is usually the least recently used entry. Additional control bits, such as a set of usage bits, are associated with each page map entry. The usage bits determine which entry is overwritten during the replacement policy. Sometimes a private bit $P$ is associated with each page to indicate that the page is private to a process or shared by a set of processes.

Pure paged memory systems can become very inefficient if the virtual space is large. The size of a page table can become unreasonably large. For example, consider a system with a 32-bit virtual address and a 1024 (1 K)-byte page size. The page address field is thus 22 bits, assuming byte addressability. Hence, we have $2^{22}$ page table entries! Assuming that we have an 8M-byte main memory, there are $2^{23}/2^{10} = 2^{13}$ page frames. Therefore, in the PTE we have a 13-bit page-frame field, or approximately 4 bytes per PTE. The total space consumed by a page table is thus $2^{24}$ bytes! In such cases, the page table may have to be paged also.

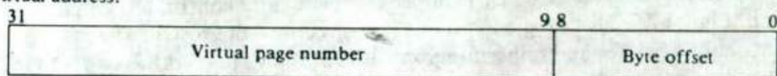
There are other disadvantages of a pure paged system. There are no mechanisms for a reasonable implementation of sharing. The size of a program space is not always an integral number of pages hence, oftentimes, *internal fragmentation* occurs in memory because the last part of the last page is wasted. In addition, there is another type of storage fragmentation called *table fragmentation*, which occurs because some of the physical memory are occupied by the page tables and so are

unavailable for assignment to virtual pages. The VAX 11/780 virtual memory system is described below as an example of a paged memory system.

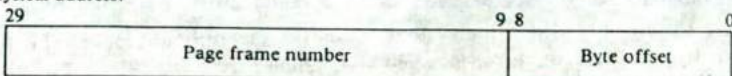
The virtual address of the VAX 11/780 is 32 bits wide and the page size is $2^9 = 512$ bytes. For each reference, this address is translated, via a page map, to a physical address that is 30 bits wide. The entry format of the page table is shown in Figure 2.8. Bit 31 of the PTE represents the valid bit which, when set, indicates that the referenced page is in main memory. Therefore, bits ⟨20:0⟩ of the PTE contain the physical page-frame number of the page. If the valid bit is reset, bits ⟨20:0⟩ of the PTE contain the invalid memory address of the referenced page. Thus a page fault occurs and bits ⟨25:0⟩ of the PTE are used to determine the location of the page on disk.

The modified bit (bit 26) of the PTE, if set, indicates that the page was modified. Hence, the disk copy of the page must be updated when the page frame is deallocated. The modified bit is set on the first reference to the page. Bits ⟨30:27⟩ of the PTE contain the protection mask or access privileges permitted on that page. The protection mask is defined for four process types: kernel, executive, supervisor, and user processes. In a memory reference, the requested access type for the process is compared to the allowable accesses, if any. Access is denied if an unpermitted access was requested.

Virtual address:




Physical address:




Page table entries:

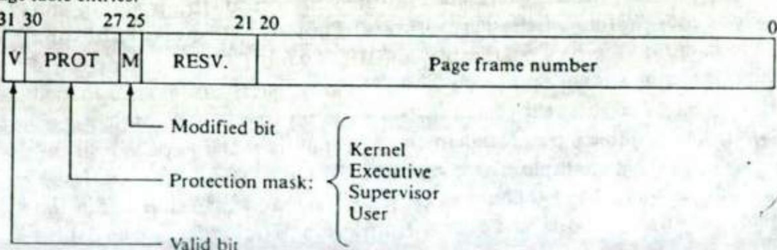



**Figure 2.8 Address and page table entry formats of VAX-11/780 virtual memory (Courtesy of Digital Equipment Corp.).**

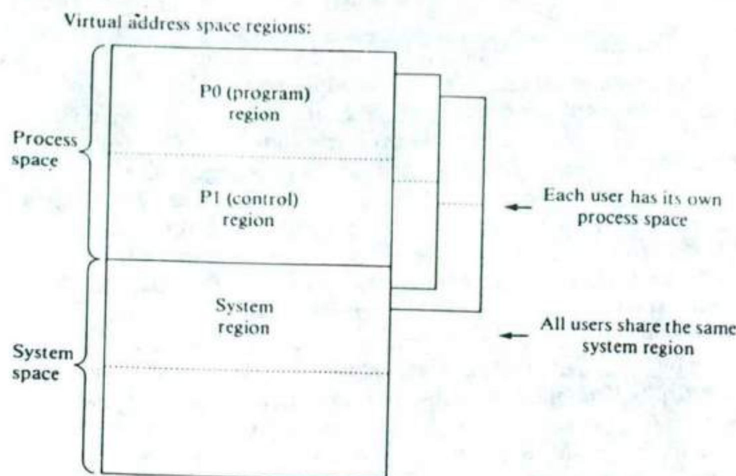


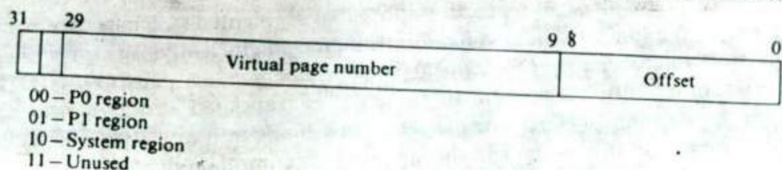

**Figure 2.9 Partitions of virtual address space (Courtesy of Digital Equipment Corp.).**

For further memory protection, the virtual address is partitioned into two spaces, process and system. Each of the two spaces are further partitioned into two regions. The process space consists of program (P0) and control (P1) regions, as shown in Figure 2.9. These regions permit two directions of growth. The system space consists of a system and unused regions. Bits of the virtual address ⟨31:30⟩ are used to specify the addressed region. A page table is established for each region. Each user process is assigned its own process space and, therefore, page tables for its private program and control regions. However, all user processes share the same system space.

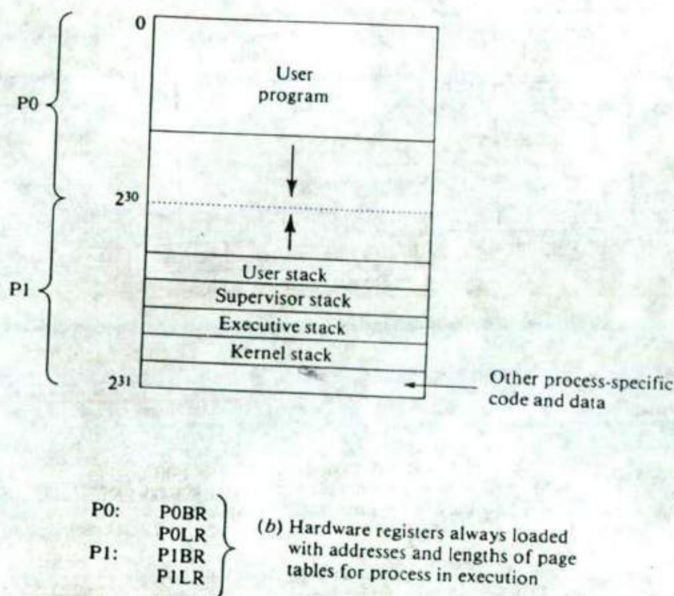
Since all users share the same system space, there is only one page table for the system space. This page table is called the *system page table* (SPT). The SPT is described by two hardware registers: the system base register (SBR) and the system length register (SLR). The SBR contains the starting physical address of the SPT, which must be contiguous and cannot be paged.

Similarly, two hardware registers are allocated to each of the program and control regions' page tables of the user process. These registers are P0BR and P0LR for the program region's page table, and P1BR and P1LR for the control region's page table, as shown in Figure 2.10. These registers are always loaded with the address and length of the page tables for the process in execution.

The process page tables are stored in the contiguous system space's virtual memory, therefore, the page table's base registers contain system space addresses so that the process-space page tables can be paged. An address reference in the process space requires a two-level address translation. The address translation process is illustrated in Figure 2.11. To speed up the translation process, an associative page map (address translation buffer) is provided. It has 128 entries divided




(a) Virtual address format




| | | |
|---|---|---|
| P0: | P0BR<br>P0LR | (*b*) Hardware registers always loaded with addresses and lengths of page tables for process in execution |
| P1: | P1BR<br>P1LR | |

**Figure 2.10 Region addressing scheme (Courtesy of Digital Equipment Corp.).**

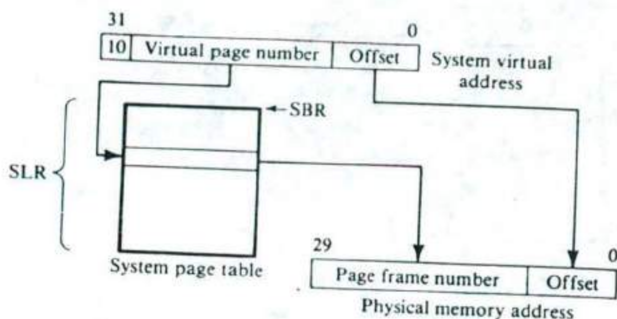
into two 64-entry groups for process and system spaces. On a context switch, only the process space entries are purged.

### 2.2.3 Segmented Memory System

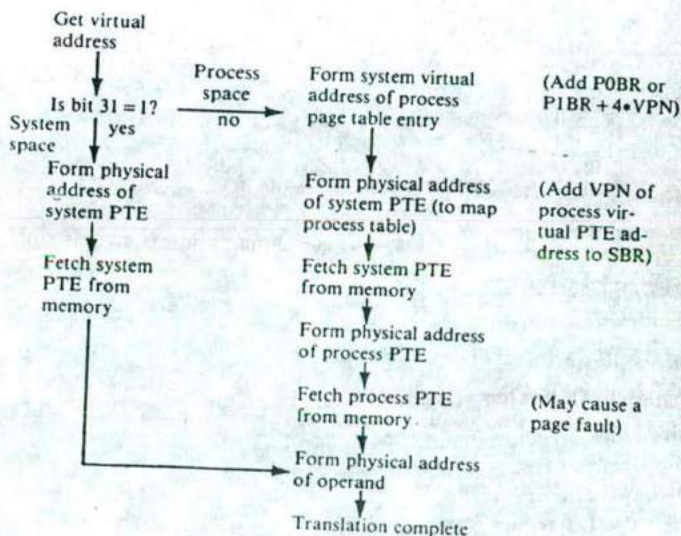
Programs which are block-structured and written in languages such as Pascal, C, and Algol yield a high degree of modularity. These modules may include procedures or subroutines which call other procedures. The modules are compiled to produce machine codes in a logical space which may be loaded, linked, and executed. The set of logically related contiguous data elements which are produced is commonly called a *segment*, which is given a segment name. Segments are allowed to grow and shrink almost arbitrarily, unlike pages, which have fixed sizes. Segmentation is a

technique for managing virtual space allocation, whereas paging is a concept used to manage the physical space allocation. In a segmented system, a user can define a very large logical space, which can be managed efficiently. An element in a segment is referenced by the segment name–element name pair (⟨s⟩, [i]). During program execution, the segment name ⟨s⟩ is translated into a segment address by the operating system. The element name maps into a relative address or displacement within the segment during program compilation.

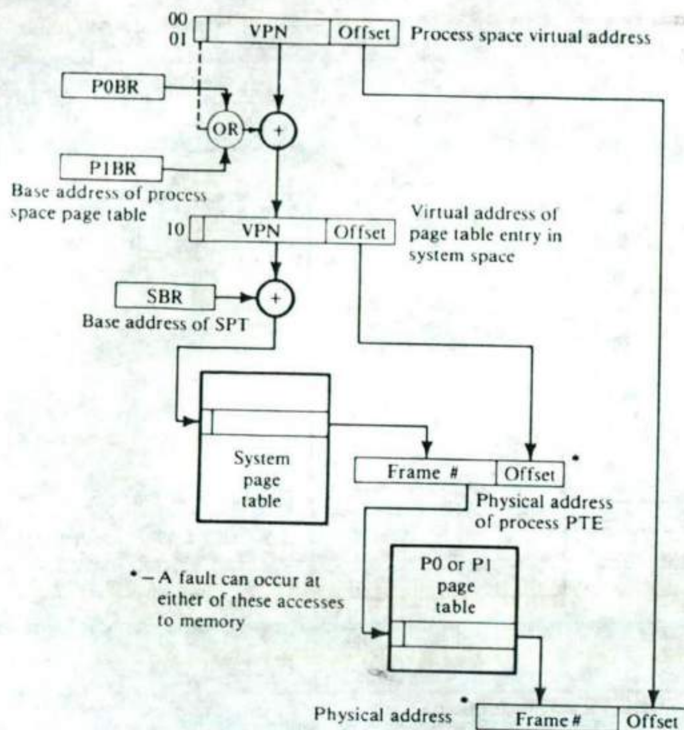
A program consists of a set of linked segments where the links are created as a result of procedure segment calls within the program segment. The method of




(a) System address translation




(b) The address translation algorithm




(*c*) Process virtual address translation

**Figure 2.11 Address translation mechanisms in VAX-11/780 virtual memory (Courtesy of Digital Equipment Corp.).**

linking the segments is an implementation problem. There are other implementation problems in the segmentation concept, including the determination of the number of segments to be allowed in the system and the maximum size of each segment. The method for sharing and protecting the segments and for mapping the virtual address into a physical address is also a design factor.

Segmentation was used in the Burroughs B5500. Each process in the system has a *segment table* (ST), pointed to by a segment table base register (STBR), in the processor when the process is active. The STBR permits the relocation of the ST, a segment itself, which is stored in main memory when the process is active. The ST consists of *segment table entries* (STE), each of which has a node structure shown in Figure 2.12.

The address field contains the absolute base address of the segment in memory if the segment is present, as indicated by the missing segment flag *F*. If the segment is missing from memory, the address field may point to the location of the segment in disk. L and RWX fields contain the length and access rights (read, write, execute)

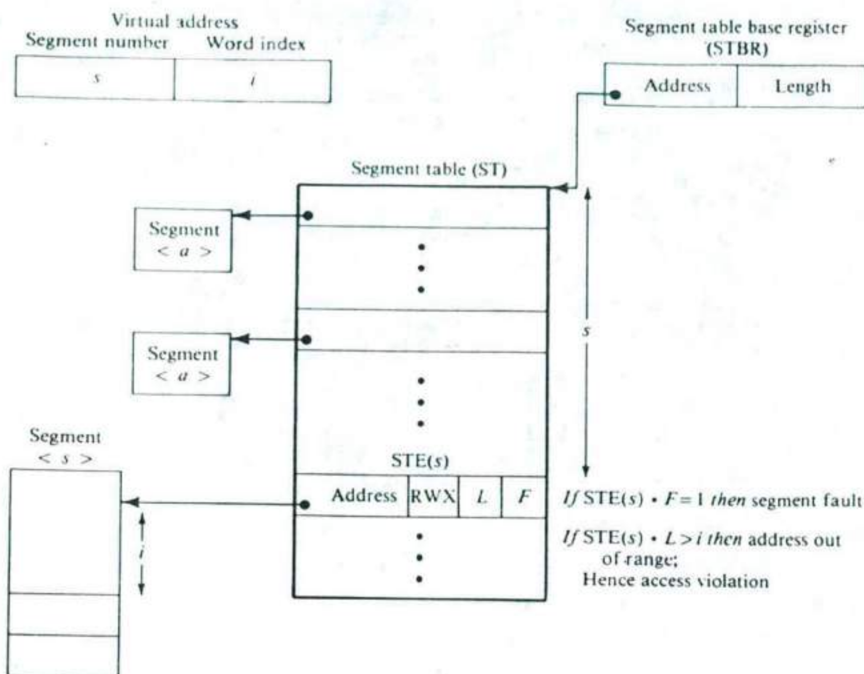



**Figure 2.12 Address mapping in a segmented system.**

attributes of the segment, respectively. Figure 2.12 illustrates how the physical address is determined from the segmented virtual address, which consists of the segment number $s$ and the index $i$ of the word within the segment. Segments may be shared by several processes, as shown in Figure 2.13 for two processes in separate processors. Notice that the relative positions of a shared segment need not be the same in different segment tables.

When a segment $\langle s \rangle$ is initially referenced in a process, its segment number $s$ is not established. The segment must therefore be *made known* to the process by providing a corresponding segment number as an entry in the ST to be used in subsequent references. Using the segment name (directory pathname) $\langle s \rangle$ as a key, a global table, called the *active segment table* (AST), which is shared by all processes, is searched to determine whether the segment is active in memory. If it is, the absolute base address of the segment and its attributes are returned and an entry is made in the AST to indicate that the process is using this segment. If $\langle s \rangle$ does not exist in AST, a file directory search is initiated to retrieve the segment and its attributes. The returned absolute base address of the segment and its attributes are entered into the AST and a newly created node of the ST. A segment number $s$, which is the displacement of the node, is assigned by the operating system from the set of unused segment numbers for that process.

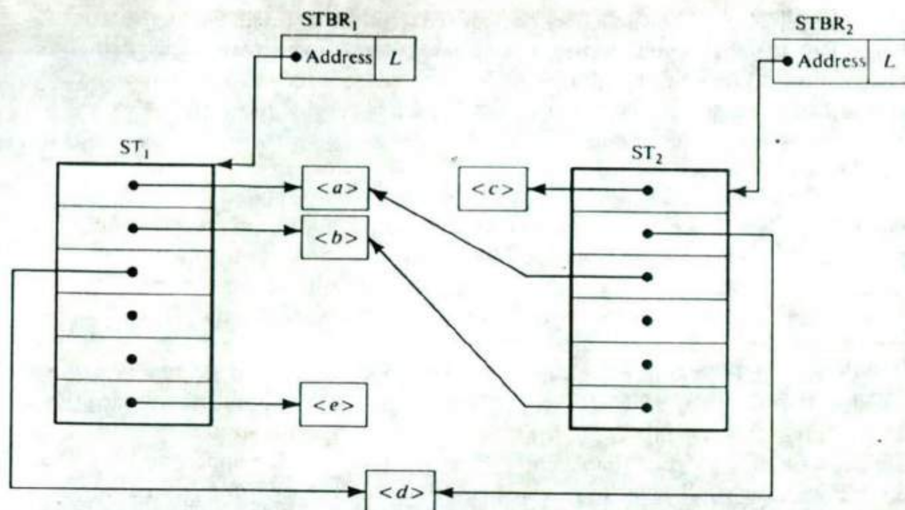



**Figure 2.13 Sharing of segments by two active processes.**

Associated with each process is a *known segment table* (KST), which contains entries on a set of segments known to the process. Each entry in the table contains a segment name–segment number pair. This is used to obtain the segment number when subsequent references are made to the segment name in the process. The address mapping mechanism shown for the segmented system involves a method of indirection to access each word that is referenced. This inefficiency may be resolved by the use of associative mapping techniques, as discussed in the paged system.

When a segment is copied from disk to memory, it is moved in its entirety. This is also true when the segment is relocated in memory. An appropriate size of contiguous data area must be found and allocated to that segment before the transfer operation is initiated. It is not often that a contiguous block of memory is found to fit the segment. In many cases, there are unused fragments of space, called *holes*, each of which may not always fit the segment to be placed. Various placement algorithms have been proposed. We present four of them.

Let $s_1, s_2, \ldots, s_n$ be the sizes of the $n$ holes available in memory, and let $s$ be the size of the segment to be placed. If the holes are listed in order of increasing size, $s_1 \leq s_2 \leq \cdots \leq s_n$, then the *best fit* algorithm finds the smallest $i$, such that $s \leq s_i$. Similarly, the *worst fit* algorithm can be defined if the holes are listed in order of decreasing size. This algorithm places the segment in the first hole and links the hole formed by the remaining space into the appropriate position in the list. In a third algorithm, called the *first fit*, the hole table lists holes in order of increasing initial address. The hole with the smallest $i$, such that $s \leq s_i$, is selected.

The fourth algorithm is the *buddy system*. In this case we assume that the segment size is $s = 2^k$ for some $k \leq n$. This policy maintains $n$ hole lists, one for each

size hole, $2^1, 2^2, \ldots, 2^n$. A hole may be removed from the $(i + 1)$th list by splitting it into half, thereby creating a pair of "buddies" of size $2^i$, which are entered in the $i$ list. Conversely, a pair of buddies may be removed from the $i$ list, coalesced, and the new hole entered in the $(i + 1)$th list. With this scheme, we can develop an algorithm to find a hole of size $2^k$.

The best fit algorithm appears to minimize the wastage in each hole it selects, since it selects the smallest hole that will fit the segment to be placed. However, the worst fit algorithm is based on the philosophy that the allocation of a larger hole will probably leave a hole large enough to be useful in the near future. It also assumes that making an allocation from a small hole will leave an even smaller hole, which will probably be useless without coalescing with other holes. The first fit and the buddy system are the most efficient algorithms.

In most cases, a time-consuming *memory compaction* is used to collect fragments of unused space into one contiguous block for the appropriate segment size. Moreover, since in the process of compaction segments in use are moved, the corresponding segment table entries must be modified. The unoccupied holes of various sizes which tend to appear between successive segments give rise to a phenomenon called *external fragmentation*. This causes memory management inefficiencies. Moreover, a whole segment may be brought into memory when only a small fraction of its address space will be referenced during the lifetime of the process, resulting in *superfluity*. These problems can be alleviated by combining segmentation with paging. It should be noted that table fragmentation also occurs in segmented systems.

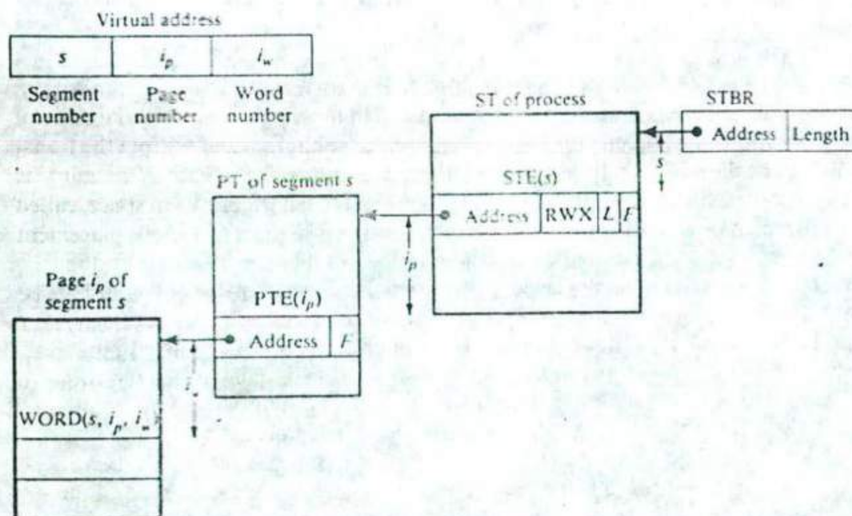



Figure 2.14 Address mapping in a system with paged segments.

### 2.2.4 Memory with Paged Segments

In this case, paging and segmentation are combined to gain the advantages of both. There are two types of paged segment schemes. One uses *linear segmentation*, in which the paging characteristics dominate, and the other is *segmented name space*, in which the segmentation characteristics dominate. In both cases, each segment is divided into pages and is referenced by the processor via a page table for that segment. An entry in the PT of a segment $s$ contains an address field and a page presence bit, $F$. If the page is present in main memory, the address field contains the absolute base address of the page. A referenced page which is absent in memory causes a *page-fault* interrupt, which invokes the page-fault handler to retrieve the page from disk to the memory. Figure 2.14 illustrates the mapping of the initial address triple $(s, i_p, i_w)$ to the physical address. The mapping of a virtual address to a physical address requires two levels of indirection, which is inefficient. Again, the mapping operation can be improved dramatically by using the associative mapping technique in each processor as illustrated in Figure 2.15. However, the improvement may be at a considerable cost. The Multics system, IBM 370/168, and the Amdahl 470 V/6 are examples of systems with segmentation and paging.

During the mapping of a virtual address of a known segment to the physical address, an access fault may occur in the TLB because of the absence of the segment or page number in the associative memory. Using the segment number, the information about the page address could be obtained from the page table, which is stored in main memory, and possibly from a local table memory (LTM) for a nonconflicting access. However, storing the page table in LTM may not facilitate the sharing of the segment. It may also create table consistency problems. An entry

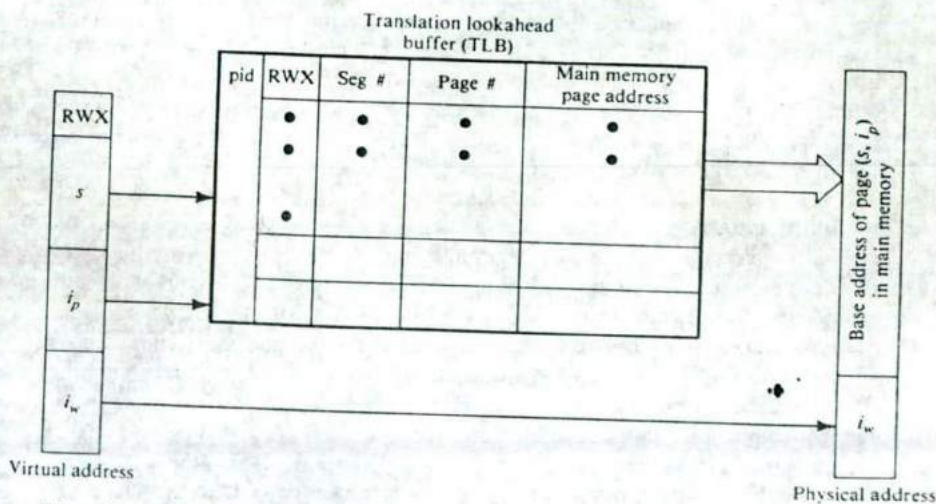



Figure 2.15 Associative map for paged segments.

in the TLB is chosen for replacement with the new segment number, page number, and page address triple.

If the segment number is not known when using segmented name space, a segment fault occurs, which invokes a procedure to make the segment known and causes the processor to perform a context switch to another process. The segment is made known by searching for the segment pathname in the AST. If it exists, the segment is in memory and is being used by an active process. Hence, its PT location in shared memory is known and is obtained from the AST entry. An unused segment number for the faulting process is obtained and an entry is made in the process control block of the process to prepare the process for subsequent execution. However, if the segment is not known to any active process, a directory search is performed to find the location of the segment in the file memory. A free PT and an unused AST entry are obtained. These segment attributes are copied to the PT and a pointer is established in the AST entry to point to the PT. A page of the segment is then copied to the memory and the appropriate entries are made in the ST of the process, as described previously.

In a virtual memory system, a *page-fault* interrupt typically violates the assumption made about interrupts on a processor. While interrupts occur asynchronously, they are constrained to be serviced at the end of an instruction cycle. However, a page fault interrupt which occurs within an instruction cycle must be serviced before the instruction cycle can be completed. This problem occurs, for example, when an instruction encoding crosses a page boundary or when a reference is made to an operand which is outside the page. Hence, at the point of interrupt, the page-fault handler must determine how far the instruction has progressed and what it must do to restart or continue the instruction cycle.

In some systems, many instructions can be restarted simply by backing up the program counter and reexecuting the instruction from the beginning. However, the partial execution of other instructions may have already made irrevocable changes to the registers and memory states. Such instructions must be restarted from the point of interruption. In general, this requires the saving of many "atomic" processor states, such as machine cycles, or the prohibition of any instructions which cannot be "backed out" of.

There are certain problems involved with using an associative map (TLB) in a multiprogrammed processor. The size of a TLB is fixed and hence can contain only a limited number of entries. If a segment number–page number pair $\langle s, i_p \rangle$ does not exist in the TLB at the time of reference, it is accessed from the PT of the segment in memory and is used to replace an entry in the TLB. When a page fault or segment fault occurs because the page or segment is not in memory, the processor suspends the faulting process and switches to a ready-to-run process.

The new process creates its own address space, which is different from the suspended process. Therefore, all the entries in the TLB map become invalid. The mapping mechanism must ensure that no old TLB entry is used in the new address space, as this may result in incorrect access to physical words of the suspended process and thereby create a hole in the protection mechanism. This problem can be solved by the context switch mechanism, which can invalidate all entries of the

TLB by the use of a special instruction, as was done in the original GE-645 Multics system. This technique degrades the performance of the system since the new process goes through initial slow indirect accesses to retrieve the $\langle s, i_p \rangle$ entries from the STs and PTs of the process.

Further problems exist when all TLB entries are invalidated. As the new process slowly fills up the TLB map with the valid entries, it may be interrupted or page faulted again, which will cause the TLB entries to be invalidated once more. Processes may continuously undergo the TLB reload cost and severely degrade the system performance. This problem can be solved by introducing in each TLB entry a process identification field which contains a short encoding of the process identification number. This technique, as implemented in the IBM 370/168, permits the associative map to contain more than one process entry (address space). However, only the entry that matches the currently running process is used. A process may therefore be restarted with part of its mapping entries in the TLB, thereby reducing the reload cost.

**Choice of page size** In purely segmented memory systems, we found that external fragmentation is a potential cause of memory under-utilization. The external fragmentation can theoretically be avoided by paging. However, paged segments reduce the utilization of memory by using additional storage space for segment and page tables (table fragmentation) and by rounding up the memory requirements for a segment to an integral number of pages (internal fragmentation). If $z$ words is the size of a page and $s$ is the segment size in words, the number of pages allocated to the segment is

$$n(s, z) = \left\lceil \frac{s}{z} \right\rceil$$

Hence, the amount of space

$$I(s, z) = n(s, z)z - s$$

usually called the internal fragmentation is wasted in the last page allocated to the segment because of the rounding off effect, assuming that $z > 1$. The page table for the segment occupies the following number of words:

$$T(s, z) = cn(s, z)$$

where $c$ is a constant.

The fraction $w$ of memory wasted because of paging in a segmented system is

$$w = \frac{n(s, z)z - s + cn(s, z)}{s} \tag{2.7}$$

The expectation of the numerator of $w$ is $(c + z)E[n(s, z)] - E[s]$ and that of the denominator is $E[s]$. If we denote the ratio of the expectations by $\bar{w}$, we have

$$\bar{w} = \frac{(c + z)E[n(s, z)] - E[s]}{E[s]}$$

If we let $\bar{s} = E[s]$, it can be shown that $E[n(s, z)] = \bar{s}/z + \frac{1}{2}$. Hence

$$\bar{w} = \frac{(c + z)(s/z + \frac{1}{2})}{\bar{s}} - 1 \tag{2.8}$$

By setting $d\bar{w}/dz = 0$, we find that the optimum page size $z_0$ and the minimum fractional wasted space $\bar{w}_0$ are $z_0 = \sqrt{2c\bar{s}}$ and $\bar{w}_0 = \sqrt{2c/\bar{s}} + c/2\bar{s}$. In general, the fractional wasted space decreases when the segments (and pages) increase in size. This is in contrast to the requirements for contiguous segments, which should be small in size to reduce external fragmentation.

The choice of the page size $z$ is a critical parameter which affects the performance of a virtual memory system. Assuming that $\bar{s} = 8192$ bytes and $c = 4$, $z_0 = 256$ bytes. This seems rather small when it is known that typical values of $z$ are 256 to 2048 bytes. In practice, the choice of $z$ depends mostly on the efficiency of the paging device.

## 2.3 MEMORY ALLOCATION AND MANAGEMENT

In this section, we discuss the various models and classification of memory management schemes. Basically, two policies, fixed and variable partitioning, are identified to manage the allocation of memory pages to active processes. In the fixed allocation scheme, the partition of memory allocated to an active process is fixed during the lifetime of the process. The variable allocation scheme permits the partition to vary dynamically during the lifetime of the process and according to the memory requirements of the active process. Various paging algorithms are discussed for both the fixed and variable partitioning policies.

### 2.3.1 Classification of Memory Policies

In general, the page-fault rate $f$ is not a value entirely intrinsic to the process. It is a critical parameter which depends on the memory management policy, which in turn determines: (1) how many pages of main memory are allocated to the process, and (2) what policy is chosen to decide which of the process's pages reside in main memory. A memory policy's *control parameter* can be used to trade paging load against resident set size.

A memory management policy change which improves the page-fault rate without changing the system load or other system parameters is expected to improve processor utilization, increase the system throughput, and decrease the response time. To show whether a change in the memory policy improves processing efficiency, it is usually sufficient to show that the change does not increase a process-paging rate. We will now discuss the various memory management policies.

Two classes of memory management policies are often used in multiprogramming systems, *fixed partitioning* and *variable partitioning*. These have been treated comprehensively by Denning and Graham and the terminologies used here are

borrowed from their work. Let us denote by $A = \{P_1, P_2, \ldots, P_d\}$ the set of active processes during the interval in which the level of multiprogramming is fixed $[d = d(t)]$. To each $P_i$ at time $t$ is associated its *resident set* $Z_i(t)$ (which is the set of the page frames of the process present in memory) containing $z_i(t) \geq 1$ pages. In general, the resident sets $Z_i(t)s'$ overlap because of the sharing that takes place among active processes. The management configuration is represented by a *partition vector* $Z(t) = [Z_1(t), \ldots, Z_d(t)]$. Hence, the *size vector* $z(t) = [z(t), \ldots, z_d(t)]$.

The total set of page frames used by the $d$ processes is

$$Z(t) = Z_1(t) \cup Z_2(t) \cup \cdots \cup Z_d(t) = \bigcup_{i=1}^{d} Z_i(t) \tag{2.9}$$

Let $z_{ij}(t)$ represent the number of pages shared by processes $P_i$ and $P_j$ such that $P_i \neq P_j$ and let $z_{ijk}(t)$ represent the number of pages shared by processes $i$, $j$, and $k$, at $t$. That is, ignoring the $t$'s, we write

$$z_i = n(Z_i),\ z_{ij} = n(Z_i \cap Z_j),\ z_{ijk} = n(Z_i \cap Z_j \cap Z_k).$$

where $n(z)$ is the number of pages in a set $z$. The sum of all $z(t)$'s with $r$ subscripts represents the total number of pages shared by $r$ processes at time $t$. We will denote this by $N_r(t)$. Hence, $N_1(t) = \sum z_i(t)$, $N_2(t) = \sum z_{ij}(t)$, $N_3(t) = \sum z_{ijk}(t)$, where $1 \leq i < j < k < \cdots \leq d$. Note that $N_r(t)$ has $\binom{d}{r}$ terms and the last sum, $N_d(t)$, reduces to a single term that indicates the number of pages shared by all the $d$ processes. If $M$ is the total number of page frames available for allocation in memory, then Feller (1970) found

$$\sum_{r=1}^{d} (-1)^{r+1} N_r(t) \leq M$$

at every time instant $t$. The pages of main memory which are unused by any active process is called the *resource memory* and is denoted by

$$R(t) = M - \sum_{r=1}^{d} (-1)^{r+1} N_r(t) \tag{2.10}$$

Analytical modeling of the sharing concept is very difficult. Most results obtained to date assume that there is no sharing and $N_r = 0$ for $r > 1$. This simplifies the problem greatly and the reserve memory becomes $R(t) = M - N_1(t)$. A memory management policy includes a method of estimating programs' locality sets. The estimates obtained are used to specify the content (and size, if adjustable) of each process resident set.

If the resident set size $z_i(t)$ is a fixed constant $z_i$ for all $t$ during which process $P_i$ is active, then the size vector $Z(t)$ is constant during any interval in which the set of active processes is fixed; this is known as the fixed-partition approach. In variable partitioning, the partition vector $Z(t)$ varies with time. The important advantage of fixed partitioning is the apparent low overhead of implementation,

since partition changes occur as infrequently as possible; that is, when the set of active processes changes. This advantage can be very easily offset (even if the memory requirements of each process can be predicted prior to processing) when one accounts for the changing locality in a process. Consider the behavior of a fixed partition when each process of the set of active processes $(P_1, \ldots, P_d)$ has a large variance in locality set size as the time varies.

Since the partition is fixed, there is no way to reallocate page frames from $Z_i$ to $Z_j$ at a time when $P_i$'s locality is smaller than $z_i$ and $P_j$'s locality is larger than $z_j$, even though such a reallocation would not degrade the performance of $P_i$ but would improve the performance of $P_j$. This effect has been analyzed by comparing fixed versus variable memory-partitioning strategies in terms of the probability that the memory space a process demands exceeds the allocated space. A study suggests that the variable-partitioning strategy is much better than the fixed-partitioning strategy because there is a severe loss of memory utilization for processes that exhibit a wide variance of locality size.

In addition to the fixed- and variable-partitioning strategies, a memory policy can either be *global* or *local*. A local policy involves only the resident set of the faulting process; the global policy considers the history of the resident sets of all active processes in making a decision.

We describe the behavior of programs being executed in terms of certain parameters which define various memory management policies for fixed- and variable-partitioning strategies. Recall that a program in execution generates a sequence of references (known as an *address trace*) to information in its virtual address space. The $i$th process's behavior is described in terms of its reference string, which is a sequence:

$$R_i(T) = r_i(1)r_i(2) \cdots r_i(T)$$

in which $r_i(k)$ is the number of the page containing the virtual address references of the process $P_i$ at time $k$, where $k = 1, 2, \ldots, T$ measures the execution time or virtual time. The set of pages that $P_i$ has in main memory just before the $k$th reference is denoted by $Z_i(k-1)$, and its size (in pages) by $z_i(k-1)$. A page fault occurs at virtual time $k$ if $r_i(k)$ is not in $Z_i(k-1)$.

There are basically two memory-fetching policies used in fetching the pages of a process when a page fault occurs, *demand prefetching* and *demand fetching*. In demand prefetching, a number of pages (including the faulting page) of the process are fetched in anticipation of the process's future page requirements. In general, prefetching can, if properly designed, improve performance by permitting an overlap between the execution and the fetching of the same program. Prefetching techniques will be discussed later. In demand fetching, only the page referenced is fetched on a miss. Demand fetching can result in an increase in superfluity. Under the assumption of demand fetching, $Z_i(k)$ is the same as $Z_i(k-1)$ plus $r_i(k)$, less any pages $\{y_i\}$ of $Z_i(k-1)$ replaced by the memory policy. Hence, using set notations,

$$\begin{aligned} Z_i(k) &= Z_i(k-1) + \{r_i(k)\} - \{y_i\} \\ z_i(k) &\leq z_i(k-1) + 1 \end{aligned} \tag{2.11}$$

The memory policy, or paging algorithm $A$, is a mechanism for processing the reference string $R_i(T)$ and for determining the sequence of resident sets $Z_i(1)Z_i(2)\cdots Z_i(T)$ and, hence, the paging rate experienced by process $P_i$. We should note that although the behavior of a process is formulated with respect to its virtual time, the behavior of the system is formulated with respect to real time.

The concept of program locality usually applies to *phases* of the program execution. Although there is a strong correlation between adjacent phases of the execution of the program, there are *transitions* between phases which do not always satisfy the concept of locality. The transitions between phases are usually characterized by fairly disruptive changes in the set of favored pages, which cannot be predicted from the past behavior. Although intraphase behavior covers the majority of the virtual time, it is the interphase behavior that produces the majority of the misses or faults. This enforces the reason for some type of anticipatory fetch policy.

A number of models for program locality have been developed. Two examples are the *independent reference model* (IRM) and the *least recently used stack model* (LRUSM). The IRM regards the reference string as a sequence of independent random variables with a common stationary reference distribution. Hence the probability that the $r_i(k)$ reference is in page $j$ is written as:

$$Pr[r_i(k) = j] = a_j \qquad \text{for all } t$$

This model predicts a geometric interreference distribution:

$$I_j(k) = (1 - a_j)^{k-1}a_j \qquad \text{for } k = 1, 2 \ldots$$

The optimal memory policy for IRM replaces the page with the smallest value of $a_j$ among the pages present in the resident set. The IRM is the simplest way of accounting for the nonlinearities observed in the swapping curves of real programs. Note that an assumption of completely random references would imply linear swapping curves. The IRM is not a good model of overall program behavior.

It has been shown that the LRUSM is a result of the LRU memory policy. This model uses an "LRU stack," which is a vector that orders the pages by decreasing recency of reference. Just after referencing $r(t)$, the first position will contain $r(t)$. A stack distance $g(t)$ is associated with the reference $r(t)$. $g(t)$ is the position of $r(t)$ in the stack just after $r(t-1)$. The LRU stack has the property that (*a*) the LRU policy's resident set of capacity $s$ pages always contains the first $s$ elements of the stack, and (*b*) the missing-page rate is the frequency of occurrences of the event $g(t) > s$. The LRUSM assumes that the distances are independent random variables with a common stationary distribution. Thus the probability of referencing a page in stack at distance $j$ is

$$Pr[g(t) = j] = b_j \qquad \text{for all } t$$

If $b_1 \geq b_2 \geq \cdots \geq b_j \geq \cdots \geq b_s$, then the LRU policy is optimal both in fixed-space and variable-space strategies. The LRUSM is slightly better than the IRM.

The above models do not adequately capture the essence of program behavior, which demands the changing need for memory from one phase to another. A realistic model must account for multiple program phases over locality sets of significantly different sizes and must not rule out strong correlations between distant phases. Some phase-transition models of program behavior have been developed which are more realistic than the last two models. Briefly, the program model consists of a *macromodel* and a *micromodel*. The macromodel is a semi-Markov chain whose "states" are mutually disjoint locality sets and whose "holding times" are phases. The macromodel is used to generate a sequence of locality-set-holding-time pairs $(S, T)$. The micromodel is used to generate a reference substring of length $T$ over the pages of locality set $S$. For the micromodel, the IRM or LRUSM may be used.

The *page-fault rate function* for process $P_i$, denoted by $f_i(A, s)$, is the expected number of page faults generated per unit of virtual time when a given reference string $R_i$ is processed by memory policy $A$, subject to the memory space constraints $s$. The page-fault rate function is one of the most important parameters in the study of memory management. Most studies performed indicate that this function is relatively independent of $R_i$. For the fixed-memory allocation, the space constraints are interpreted to mean that the resident set sizes must satisfy $z_i(k) \leq s$ for all virtual times $k$. For the case of variable-space allocation, the space constraint $s$ is interpreted as the *average resident set size* of process $P_i$, that is

$$s = \frac{1}{d} \sum_{k=1}^{d} z_i(k) \tag{2.12}$$

for a system with $d$ active processes. In both allocation schemes, the *page-fault rate* for the total page is

$$f = \sum_{i=1}^{d} f_i(A, s) \tag{2.13}$$

which is the figure of merit to be minimized, subject to the allocation constraints

$$\sum_{r=1}^{d} (-1)^{r+1} N_r \leq M$$

where $N_r$ is the number of pages shared by $r$ processes and $M$ is the total number of page frames in the main memory.

Another measure of the page-fault rate is the *lifetime function* $e_i(z_i)$, which gives the mean execution interval (in virtual time) between successive page faults for process $P_i$ when it has $z_i$ of its pages in shared memory. The derivation of this function assumes a given memory policy. A *knee* of a lifetime curve is a point at which $e_i(z_i)/z_i$ is locally maximum. The primary knee is the global maximum of $e_i(z_i)/z_i$. A typical lifetime curve is shown in Figure 2.16.

Several empirical models of the lifetime curve have been proposed. One is the *Belady model*:

$$e_i(z_i) = a \cdot z_i^k \tag{2.14}$$

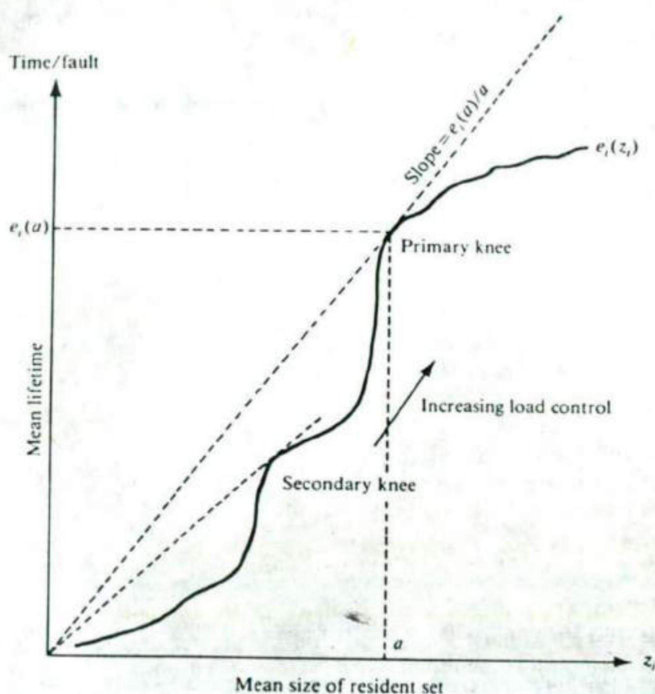



Figure 2.16 A lifetime curve.

where $z_i$ is the mean resident set size, $a$ is a constant, and $k$ is normally between 1.5 and 3. In general, $a$ and $k$ depend on the program characteristics. This model is often a reasonable approximation of the portion of the lifetime curve below the primary knee, but it is otherwise poor.

A second model is the *Chamberlin model*:

$$e_i = \frac{2b_i}{1 + (c_i/z_i)^2} \tag{2.15}$$

This model was derived empirically as a result of many measurements performed on computer systems. It was observed that the lifetime curve $e_i(z_i)$ is concave for small values of $z_i$ and becomes convex as $z_i$ increases, as shown in Figure 2.16. Two parameters characterize the behavior of a process $P_i$ in the concave-convex model: $c_i$, the number of pages for which $e_i = b_i$, that is, for which the mean execution interval is half of the longest interval for $P_i$; and $b_i$, the mean execution interval between page faults when the process $P_i$ is allocated a memory space of $c_i$ page frames. The parameter $c_i$ gives a relative measure of the memory space needed to enable the process to be executed efficiently. It can be seen from Eq. 2.15 that the

transition from the concave to the convex region occurs at $z_i = c_i/\sqrt{3}$, and that in $z_i = c_i$ the curvature in the convex region is maximum. Therefore, $e_i$ could be considered a reasonable approximation to the memory demand of $P_i$.

Although this model has a knee, it is not a very good match for real programs. It is generally quite easy to measure lifetime curves from real data and such measurements are generally more reliable than estimates from models. If the page transfer time is $S$ then the *page-fault rate* for process $P_i$ is

$$f_i = \frac{1}{e_i + S} \tag{2.16}$$

This equation can be used to derive an optimization problem, which can then be solved to obtain optimum memory space allocation in a multiprogramming system.

Another measure that is often used is the *space-time product* of an active process. This product is the integral of a program's resident set size over the time $T$ it is running or waiting for a missing page to be swapped into shared memory. Let $z(t)$ be the size of the resident set at time $t$, $t_i$ be the time of the $i$th page fault ($i = 1, \ldots, K$), and $D$ be the mean swapping delay. The *space-time product* is

$$ST = \sum_{t=1}^{T} z(t) + D \sum_{i=1}^{K} z(t_i) \tag{2.17}$$

If $s$ is the mean resident set size, we can approximate $ST$ by noting that the first sum becomes $sT$. If we approximate the second sum by $sK$ and note that $sK = s(K/T)T = sf(s)T$, where $f(s)$ is the missing page rate, the space-time product is approximated by

$$ST = Ts[1 + Df(s)] \tag{2.18}$$

Although Eq. 2.18 is simple to compute, the approximation is not very reliable. Note that $sf(s) = s/e_i(s)$ is minimum at the primary knee of the lifetime curve. If $D$ is large, choosing $s$ at this knee will approximately minimize the space-time product.

### 2.3.2 Optimal Load Control

Main memory is considered a prime system resource which is used dynamically by the active processes in a multiprogramming environment. The number of active processes (degree of multiprogramming) in a parallel processor system is usually greater than the number of available processors so that when one of the running processes is suspended the processor may switch to another active process. This capability requires the memory be able to hold the pages of the active processes in order to reduce the context switching time. In general, multiprogramming improves concurrency in the use of all system resources, but the degree of multiprogramming should be varied dynamically to maintain both a low overhead on the system and a high degree of concurrency.

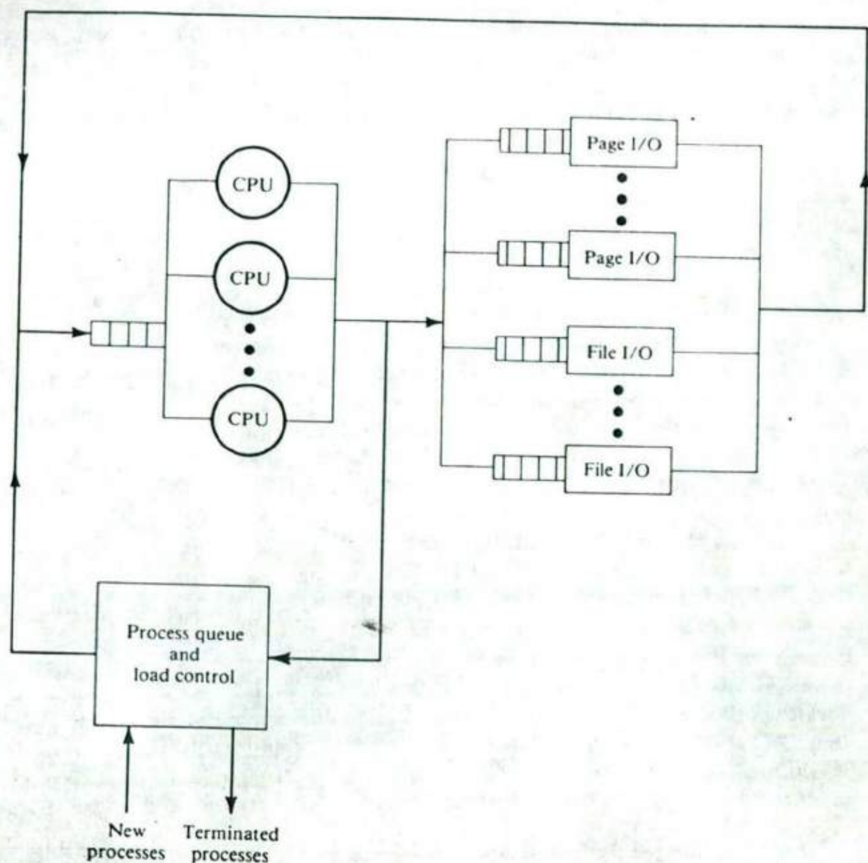



**Figure 2.17 A multiprogrammed multiprocessing virtual memory system model.**

Usually these are two conflicting requirements. Increasing the degree of multiprogramming may overcommit the memory to holding sets of only a few pages or segments of the active processes. In such a case, a context switch to a process with too small a working set may almost immediately encounter another page fault, which would necessitate another context switch to get another process with a small working set. If these activities occur continuously, the system is said to be *thrashing*, whereupon its performance is degraded considerably as it spends most of its time page-faulting and context-switching. The number of active processes or the *degree of multiprogramming* (DOM) will be denoted by a time-dependent variable $d(t)$.

Figure 2.17 depicts the model of the multiprogrammed multiprocessor system. This network consists of two main portions: the *active network* which contains

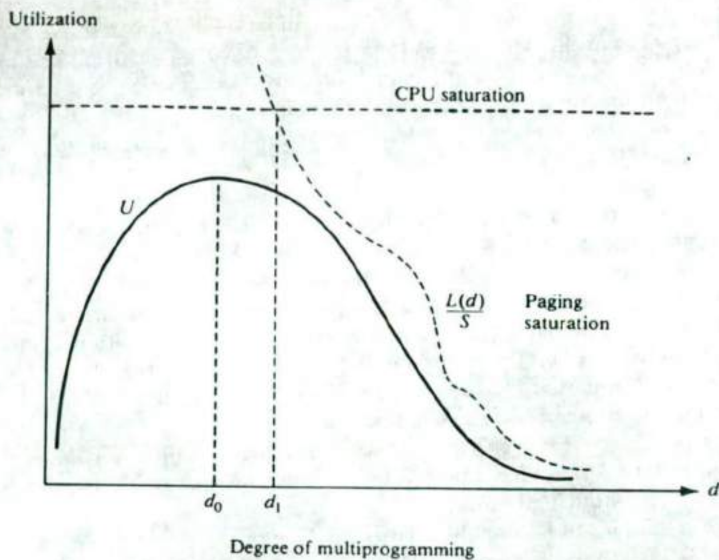
the processors, memory and the file memory, and the *passive network* which contains a process queue and the policies for admiting new processes to active status. A process is considered active if it is in the active network, where it is eligible to receive processing and have pages in main memory. Each active process is waiting or in service at one of the three classes of resources in the active network. It waits at the file I/O class whenever it requires a segment to be transferred between main memory and the disk memory. An active process waits at the paging device modules whenever it requires a page to be transferred between main memory and a paging device, such as a drum or fixed-head disk. Otherwise it is in the CPU station.

The box labeled "Process queue" contains a set of enabled (passive) processes, a decision policy for activating them, and a load-control mechanism for controlling $d(t)$. Notice that each CPU node is usually considered to have a cache whose action is transparent, i.e., a cache miss does not necessitate a context switch. When a process either issues a file I/O request or creates a page fault, it will release its processor to another ready process and wait for the completion of the I/O transaction. Such a model, as depicted in the Figure 2.17, is called a closed queueing network model with $d$ processes, where $d$ is the steady-state degree of multiprogramming.

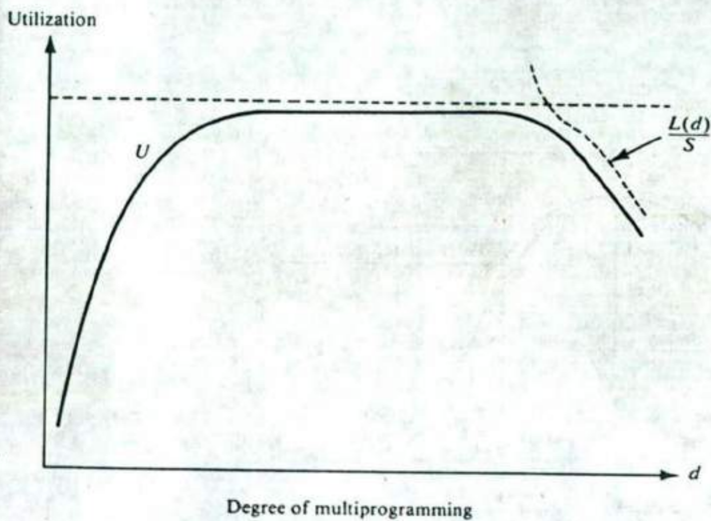
In addition to the DOM, another parameter used in the memory management model is the average total time used to service each process which requested paging device $i$. This time, which is denoted $D_i$, is the demand per process for the $i$th device. For each device, $D_i$ is the product of the mean number of requests per process for that device and the mean time to service one request. For the paging device, the demand per process grows with $d$ because higher DOMs imply smaller resident sets and higher rates of paging. For devices such as CPU and I/O, the demand per process does not depend on $d$. The demand for each CPU is the mean execution time $E$ of a process. The average number of page faults per process is $E/L(d)$, where $L(d)$ denotes the lifetime or mean time between faults for a DOM of $d$.

The demand for the paging device is $D_i = ES/L(d)$, where $S$ is the mean time to service one page transfer (exclusive of queueing delays). If the function $L(d)$ is not available from a direct measurement of the system, it can be estimated from the lifetime curve of a typical program. One method to estimate $L(d)$ is to set $L(d) = e_i(M/d)$, where $e_i(x)$ is the mean time between page faults for a typical program when the given memory policy produces a mean resident set size of $x$ pages and $M$ is the number of available pages of main memory.

The queueing network model of Figure 2.17 can be used to estimate the system's throughput $X_0$, which is the number of processes completed per second. The throughput is proportional to the average utilization of the CPUs, $U$, and is given by $X_0E$. Figure 2.18 illustrates typical CPU utilization curves as a function of the DOM. The curve rises toward CPU saturation as the degree of multiprogramming $d$ increases, but is eventually depressed by the ratio $L(d)/S$, the utilization of the saturated paging device. As suggested in Figure 2.18$a$, the DOM $d_1$, at which $L(d) = S$, is slightly better than the optimum $d_0$. Note that beyond $d_0$, the system begins to thrash.




(*a*) Small main memory




(*b*) Large main memory

**Figure 2.18 Optimum degrees of multiprogramming.**

This is known as the $L = S$ *criterion*, which can be used as an adaptive load control. It keeps the averaged lifetime at least as great as the page-transfer time for a page fault. The size of the main memory certainly affects the $L = S$ criterion. A very large main memory offsets the instabilities in memory policies and overheads created when the resident sets attempt to overflow the available memory space. The degree of multiprogramming can increase considerably in a system with a very large main memory without significant overhead because of swapping, as illustrated in Figure 2.18*b*. Once the main memory is large enough to allow the CPU utilization to be near one for some $d$, further increases of memory cannot increase the system throughput or decrease the response time.

An approximation of the optimum DOM, as characterized by the relation $L = aS$ for some constant, is not quite adequate. This approximation fails when the system is I/O bound or when the maximum lifetime $L$ does not exceed the page-swapping time. The optimum DOM is actually achieved by running each process at its minimum space-time product, which is more difficult to achieve than the $L = aS$ criterion. Recall that if the total delay (queueing time plus swap time $S$) per page fault is large, the space-time product will be minimized approximately at the primary knee of the lifetime curve.

This *knee criterion* can be used as a basis for load control. It is more robust than the $L = S$ criterion. The knee criterion is a memory allocation strategy which achieves the maximum ratio of the lifetime to the memory allotment for process in a multiprogramming set. To limit the drop of CPU utilization, a maximum limit $d_{max}$ is set on the DOM. The function of the load controller is to attempt to set $d_{max}$ near the current optimum. If the number of submitted processes at a

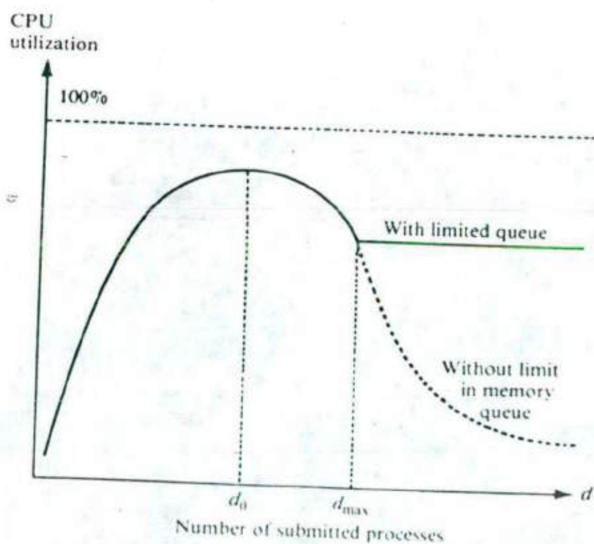



**Figure 2.19 Effect of the load control on CPU utilization.**

given time does not exceed $d_{max}$, all are active; otherwise, the excess processes are held inactive in a memory queue. The limit effect of the memory queue is illustrated in Figure 2.19. In practice, the optimum DOM varies with the work load, therefore an adaptive control is required to adjust $d_{max}$.

The load control is accomplished by a component of a dispatcher, which is part of the operating system. The purpose of the dispatcher is to control the scheduling of processes and allocation of main memory so that the throughput for each work load is maximum. The dispatcher consists of three components: the *scheduler*, the *memory policy* and the *load controller*. The scheduler determines the composition of the active set of processes. It does this by activating processes from the passive process queue into the active set. The memory policy determines a resident set for each active process and, as we have seen, the load controller adjusts the limit $d_{max}$ on the degree of multiprogramming. All memory policies manage a pool of unused space in main memory. The pool contains the pages of resident sets of recently deactivated processes. Under a fixed-space policy, the pool also contains pages which have recently left the resident sets of active processes. By comparing the measured memory demand of a process with the pool's size, the scheduler avoids activating a process if the activation would overload the system.

### 2.3.3 Memory Management Policies

The following definitions will be used in describing various paging algorithms. Given a reference string $R(t) = r(1)r(2)\cdots r(t)$, the *forward distance* $d_t(x)$ at time $t$ for page $x$ is the distance of the first reference to $x$ after time $t$. That is,

$$d_t(x) = \begin{cases} k & \text{if } r(t+k) \text{ is the first occurrence of } x \text{ in } R(\infty) - R(t) \\ \infty & \text{if } x \text{ does not appear in } R(\infty) - R(t) \end{cases} \tag{2.19}$$

Similarly, we define the *backward distance* $b_t(x)$ as the distance to the most recent reference of $x$ in $R(t)$. Hence

$$b_t(x) = \begin{cases} k & \text{if } r(t-k) \text{ is the last occurrence of } x \text{ in } R(t) \\ \infty & \text{if } x \text{ never appeared in } R(t) \end{cases} \tag{2.20}$$

Let $Q(Z)$ be the page replaced from resident set $Z$ when a page fault occurs. Below we list examples of commonly used demand-paging page-replacement memory policies for fixed-space and local-policy allocation schemes:

1. *Least recently used* $(LRU)$—At page fault replaces the page in $Z(t)$ with the largest backward distance:

$$Q(Z(t)) = y \qquad \text{if and only if } b_t(y) = \max_{x \in Z(t)} [b_t(x)] \tag{2.21}$$

2. *Belady's optimal algorithm* $(MIN)$—At page fault replaces the page in $Z(t)$ with the largest forward distance:

$$Q(Z(t)) = y \qquad \text{if and only if } d_t(y) = \max_{x \in Z(t)} [d_t(x)] \tag{2.22}$$

This algorithm minimizes the number of page faults.

3. *Least frequently used (LFU)*—Replaces the page in $Z(t)$ that has been referenced the least number of times.
4. *First-in, first-out (FIFO)*—Replaces the page in $Z(t)$ that has been in memory for the longest time.
5. *Clock algorithm (CLOCK)*—A disconcerting feature with the FIFO algorithm is that it may end up replacing a frequently referenced page because it has been in memory for the longest time. This problem is alleviated by associating a *usage bit* with each entry in the FIFO queue, which is made circular, and establishing a pointer for the circular queue. The usage bit for an entry in the queue is set upon initial reference. On a page fault, the pointer resumes a cyclic scan through the entries of the circular queue, skipping used page frames and resetting their usage bits. The page frame in the first unused entry is selected for replacement. This algorithm attempts to approximate LRU within the simple implementation of FIFO.
6. *Last-in, first-out (LIFO)*—Replaces the page in $Z(t)$ that has been in memory for the shortest time.
7. *Random (RAND)*—Chooses a page in $Z(t)$ at random for replacement.

Since the LRU policy is one of the most popular algorithms, we will describe its implementation. Associated with this policy is a dynamic list known as the *LRU stack*, which arranges the referenced pages from top to bottom by decreasing order of recency of reference. At a page replacement time, the LRU policy chooses the lowest-ranked page in the stack, therefore, the contents of an $s$-page resident set must always be the pages occupying the first $s$ stack positions. When a page is referenced, the stack is updated by moving the referenced page to the top and pushing down the intervening pages by one place. The position at which the referenced page was found before being promoted to the top is called the *stack distance*. A page fault occurs in an $s$-page resident set at a given reference if and only if the stack distance of that reference exceeds $s$. In the fixed partitioning strategy, each active process has its own LRU stack.

Algorithms such as LRU, LFU, LIFO, FIFO, and RAND, which are called *nonlookahead* algorithms, are realizable. MIN is a lookahead page-replacement algorithm and is not realizable, but provides a benchmark on which we can measure the relative performance of the realizable algorithms. Figure 2.20 illustrates the typical relative page-fault rates for various paging algorithms. The page-fault rate $f(A, s)$ for a given algorithm $A$ and resident size constraint $s$ can be computed from the reference string $R$.

Let $S_j(A, s, R)$ represent the set of pages in the resident set-size constraint $s$ at time instant $j$ when processing a reference string $R$. A natural expectation is that if the size constraint $s$ increases, the following *inclusion property* would hold:

$$S_j(A, s, R) \subseteq S_j(A, s+1, R) \tag{2.23}$$

However, the FIFO algorithm has the disadvantage of exhibiting erratic and undesirable behaviors under certain circumstances and does not always satisfy

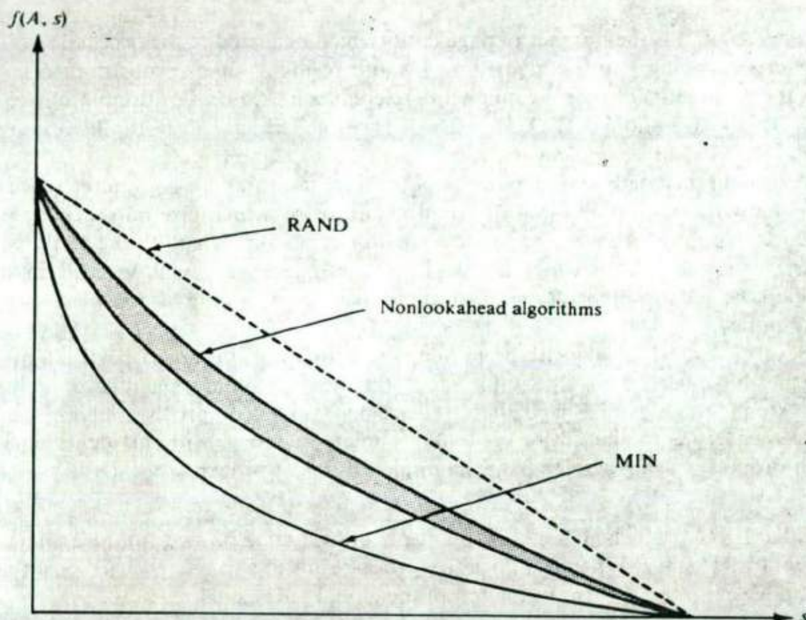



**Figure 2.20** Page fault rates of realizable and nonrealizable algorithms for various resident set sizes.

the inclusion property. For example, consider the processing of the reference string $R = 12314$, using the FIFO algorithm, when the address space of the process is the set $M = \{1, 2, 3, 4\}$ for two resident size constraints, $s = 2$ and $s = 3$. Below we show the sequence of $S_j$ states generated as a result of the processing of the string $R$. In this illustration, an asterisk (*) after a reference indicates that no page fault occurred, otherwise, a page fault did occur.

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | |
|---|---|---|---|---|---|---|
| $R =$ | 1 | 2 | 3 | 1 | 4 | |
| | 1 | 1 | 2 | 3 | 1 | $s = 2$ |
| | | 2 | 3 | 1 | 4 | |
| | 1 | 1 | 1 | 1* | 2 | $s = 3$ |
| | | 2 | 2 | 2 | 3 | |
| | | | 3 | 3 | 4 | |

Notice that $S_5(\text{FIFO}, 2, R) \not\subset S_5(\text{FIFO}, 3, R)$ and, hence, does not satisfy the inclusion property. The *normalized page-fault rate* can be obtained from the expression

$$f(A, s) = \frac{N(A, s, R)}{|R|} \tag{2.24}$$

where $N(A, s, R)$ is the number of page faults which occurred in the processing of the reference string $R$ using algorithm $A$ and a resident set-size constraint of $s$. $|R|$ is the cardinality of $R$ or the number of references in $R$. For the example above, $f(\text{FIFO}, 2) = 1.0$ and $f(\text{FIFO}, 3) = 0.8$. Algorithms which satisfy the inclusion property are called *stack algorithms*.

Although this method of derivation of the page-fault rate for a given reference is adequate, it does not account for the mechanisms by which programs generate reference strings. Moreover, the procedures do not readily extend to the analysis of variable-space policies which use the locality of reference model. We will now consider the paging algorithms for variable-space partitioning strategy using a global policy.

Several important algorithms for implementing variable-space partitioning strategies have been used. One approach to the memory management commonly used extends the idea of a fixed-space replacement policy simply by applying the replacement rule to the entire contents of main memory, without identifying which process is using a given page. Examples of this approach are:

1. *Global LRU*—which arranges all the pages of the active processes into a single global LRU stack. Whenever an active process runs, it will reference its locality set pages and move them to the top of the global LRU stack.
2. *Global FIFO*—which arranges all the pages of the active programs into a single global FIFO list.

A load control is necessary for the successful implementation of the global LRU policy, for if there are too many active processes, pages will be taken from the resident set of the least recently run process (whose pages will tend to occupy the lowest stack positions), whereupon that process, when run, will soon experience a page fault. This type of policy has been found highly susceptible to thrashing and may not perform better than fixed-space partition policies.

There is a variation of the global LRU policy which uses a *usage bit* $u$ and a *changed bit* $c$, which are associated with every resident page. The bit $u$ is set to 1 by the addressing hardware on any reference to the given page and is cleared to 0 by the memory management routine. The bit $c$ is set to 1 by the addressing hardware on the first write reference to the given page and is cleared when the page is loaded or when the disk copy is updated. At intervals, the memory management process scans all resident set pages and maintains them in four lists according to the possible values of the bits $(u, c)$. At a page fault, the first page of the first non-empty list in the order $(u, c) = [(0, 0), (0, 1), (1, 0), (1, 1)]$ is selected for replacement. This policy, which approximates LRU, is subject to the same problems as LRU when used for multiprogramming.

Another variation of the global LRU and FIFO combines elements of both policies. It is called *global FINUFO* (first-in, not-used, first-out). In this policy, all the pages of the active processes are linked in a circular list with a pointer designating the current position. Each page has a usage bit which is set by the

hardware to 1 when the page is referenced. Whenever a page fault occurs, the memory policy advances the current-position pointer around the list clearing set usage bits and stopping at the first page whose usage bit is already cleared to 0: this page is selected for replacement. This paging algorithm was used in the Multics system. The above variable-space allocation policies do not attempt to identify locality sets and protect them from preemption.

Another example of the variable memory partitioning is the *working-set* (WS) algorithm, which takes into account the varying memory requirements during the execution of a process. Denning (1968) introduced the concept of working set to describe program behavior in virtual memory environments. The working set $W(t, \theta)$ is used to denote an estimator of a locality set. $W(t, \theta)$ of a process at time $t$ is defined as the set of distinct pages which are referenced during the execution of the process over the interval $(t - \theta, t)$, where $\theta$ is the window size. The working-set size $w(t, \theta)$ is the number of pages (cardinality) of the set $W(t, \theta)$.

This algorithm retains in memory exactly those pages of each process that have been referenced in the preceding $\theta$ seconds of process (virtual) time. If an insufficient number of page frames are available, then a process is deactivated in order to provide additional page frames. Notice that the working-set policy is very similar to the LRU policy in that the working-set algorithm specifies the removal of the LRU page when that page has not been used for the preceding $\theta$ time units, whereas the LRU algorithm specifies the removal of the $s$th least recently used page when a page fault occurs in a memory of capacity $s$.

The success of the working-set algorithm is based on the observed fact that a process executes in a succession of localities; that is, for some period of time the process uses only a subset of its pages and with this set of pages in memory, the program will execute efficiently. This is because, at various times, the number of pages used in the preceding $\theta$ seconds (for some appropriate $\theta$) is considered to be a better predictor than simply the set of $K$ (for some $K$) pages most recently used. Thus for example, a compiler may need only 25 pages to execute efficiently during parsing, but may need 50 during code generation. A working set with the correct choice of the parameter $\theta$ would adapt well to this situation, whereas a constant $K$ over both phases of the compiler would either use excess space in the syntax phase or insufficient space in the code-generation phase. The working-set paging algorithm, although efficient, is difficult to implement, however.

Yet another variable-partitioning strategy which can use local or global policy is the *page-fault frequency* (PFF) replacement algorithm. The PFF also attempts to follow variations in localities when allocating memory space to processes. This policy is implemented using hardware usage bits and an interval timer and is invoked only at the time of a page fault. Let $t'$ and $t$ for $t > t'$ denote two successive (virtual) times at which page faults occur in a given process. Also, let $R(t, \theta)$ denote the PFF resident set just after time $t$, given that the control parameter of PFF has the value $\theta$. Then

$$R(t, \theta) = \begin{cases} W(t, t - t') & \text{if } t - t' > \theta \\ R(t', \theta) + r(t) & \text{otherwise} \end{cases} \tag{2.25}$$

where $r(t)$ is the page referenced at time $t$ (and found missing from the resident set). The interfault interval $t - t'$ is used as a working-set window and the parameter $\theta$ acts as a threshold to guard against underestimating the working set in case of a short interfault interval. Hence, if the interval is too short, the resident set is augmented by adding the fault page $r(t)$. The usage bits, which are reset at each page fault, are used to determine the resident set if the timer reveals that the interfault interval exceeds the threshold. Note that $1/\theta$ can be interpreted as the maximum tolerable frequency of page faults.

Various experimental studies have shown that WS and PFF, when properly "tuned" by good choices of their control parameters, perform nearly the same as each other and considerably better than LRU. The PFF may display anomalies for certain programs since it does not satisfy the inclusion property. Since global memory policies make no distinctions among programs, their load controls have no dynamically adjustable parameters. However, these controls cannot ensure that each active program is allocated a space-time minimizing resident set. Local memory policies such as WS and PFF offer a much finer level of control and are capable of much better performance than global policies. These policies, however, present the problem of selecting a proper value of the control parameter $\theta$ for each program.

Finally, we present an ideal variable-space memory policy which could be local or global. This is the *optimal variable replacement* algorithm called VMIN. The VMIN generates the least possible fault rate for each value of mean resident-set size. At each reference $r(t) = i$, VMIN looks ahead: if the next reference to page $i$ occurs in the interval $(t, t + \theta)$, VMIN keeps $i$ in the resident set until that reference; otherwise, VMIN removes $i$ immediately after the current reference. Page $i$ can be reclaimed later when needed by a fault. In this case, $\theta$ serves as a window for lookahead, analogous to its use by WS as a window for lookbehind. VMIN anticipates a transition into a new phase by removing each old page from residence after its last reference prior to the transition. This results in a behavior depicted in Figure 2.21. In contrast, WS retains each segment for as long as $\theta$ time units after the transition. VMIN and WS generate exactly the same sequence of page faults for a given reference string. The suboptimality of WS results from resident set "overshoot" at interphase transitions, as shown in Figure 2.21. However, since VMIN is a lookahead algorithm, it is not practical.

**Prefetching techniques** Prefetching is a technique to reduce the paging traffic during locality phase transitions. Recall that there are two aspects of phase transition behavior. The first aspect is the removal of the pages of the old locality set; the second aspect is the fetching of the pages of the new locality set. A prefetch policy must dictate three main issues:

1. When do you initiate a prefetch?
2. Which block or blocks do you prefetch?
3. What replacement status do you give the prefetched block?

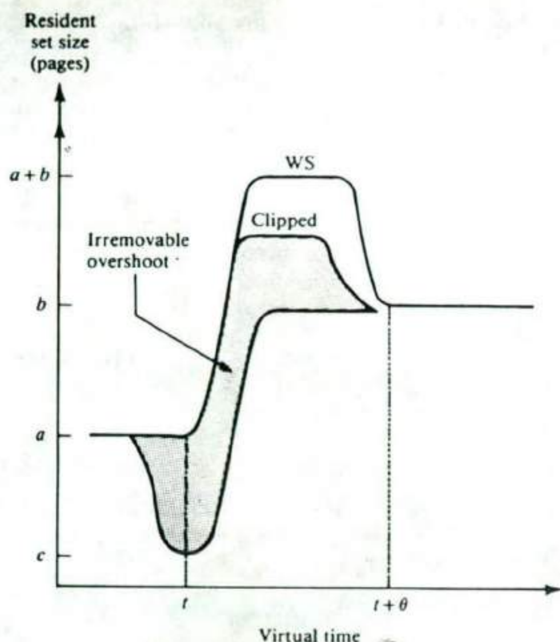



**Figure 2.21 Behavior of policies near a transition between phases.**

One prefetch technique is the well-known *swapping*, commonly used in multiprogrammed or time-sharing systems. If we do not use swapping, in such systems a context switch occurring because of the end of a time slice is followed by a slow purge of the process's blocks from memory. Similarly, later reactivation of the process is followed by a slow and tedious recognition of the locality set. This behaves as a transition of localities. A compromise between demand fetching and swapping is to demand fetch during a process's active intervals and use swapping at the end and beginning of the time slice to save and restore, respectively, the working set of the process.

Another type of prefetching technique is based on the frequently observed principles of spatial locality and sequentiality of references in programs. An example of a prefetching algorithm exploiting this property is the so-called one block lookahead (OBL) algorithm. We illustrate this algorithm by considering a stack of $s$ pages, where $s$ is the page allotment to the process. The demanded page is placed at the top of the stack in the usual manner. When a page fault occurs, the page that is the sequential successor of the demanded page in the virtual address space is prefetched, provided it is not resident in main memory. When such a prefetching occurs, an additional replacement is done and the prefetched page is placed at the bottom of the stack. Thus, if another page fault occurs before the

prefetched page is referenced, then the prefetched page is replaced. In all other respects, this stack is maintained like an LRU stack.

This algorithm, when applied to database systems and vector operations, performs adequately where a high degree of sequentiality is present. It has been observed that common programs in execution do not possess adequate spatial locality unless the page size is rather small. As we shall see in Section 2.4, systems with caches employ a small block size so the OBL algorithm may be used to advantage in them. Also, since the units of information transfer from memory to the processor are quite small, and the instruction stream tends to exhibit a high degree of sequentiality, sequential prefetching of instructions into an instructor buffer is commonplace. Sequentiality may be induced in the data streams for vector instructions. Prefetching algorithms must be designed carefully so as not to nullify the potential gain in the reduction of page faults or misses by a disproportionate increase in the number of fetches.

## 2.4 CACHE MEMORIES AND MANAGEMENT

Cache memories are high-speed buffers which are inserted between the processors and main memory to capture those portions of the contents of main memory which are currently in use. They can also be inserted between main memory and mass storage. Since cache memories are typically five to 10 times faster than main memory, they can reduce the effective memory access time if carefully designed and implemented. This section discusses the characteristics of cache memories and the various cache management strategies. Four cache organizations—direct, fully associative, set associative, and sector mappings—are discussed. Cache replacement policies are used to decide what cache block to replace when a new block is to be brought into the cache.

### 2.4.1 Characteristics of Cache Memories

The success of cache memories can be attributed to its property of locality of references. The effectiveness of the cache in capturing localities is measured by the asymptotic fraction of program references found in the cache, called the *hit ratio* $h$.

The design of a cache memory for a concurrent computer system usually involves the minimization of a number of parameters, such as the miss ratio $(1 - h)$, the access time, the delay due to a miss, and the penalty for updating main memory. It also involves maintaining data consistency between the cache and main memory and, in the case of a multicache system, maintaining data consistency between the multiple caches. These and other aspects are discussed below. First, we describe the functional operation of a cache.

**Operation of a cache** The cache memory generally consists of two parts, the *cache directory* (CD) and the *random-access memory* (RAM). The memory portion is partitioned into a number of equal-sized blocks called *block frames*. The directory,

which is usually implemented as some form of associative memory, consists of block address tags and some control bits such as a "dirty" bit, a "valid" bit, and protection bits. The address tags contain the block addresses of the blocks that are currently in the cache memory. The control bits are used for cache management and access control. Hence, the cache contains a set of address-data pairs, each of which consists of the main memory block address and a copy of the contents of the main memory block corresponding to that address.

The cache directory can be implemented as either an *implicit* or *explicit* lookup table. In the explicit directory, the referenced data is fetched from the memory portion of the cache only after the corresponding address tag has been searched. The implicit lookup table permits the simultaneous searching of the address tags and the fetching of the corresponding data. However, the presence of the desired block and its location in the cache are only detected at the end of the cache cycle.

The operation of the cache is simple in concept, as illustrated in Figure 2.22, for a fetch operation. The processor generates is a virtual address which is mapped into a physical memory address via a *translation lookaside buffer* (TLB). If there is a TLB hit, the corresponding physical page address is retrieved to form the physical address and the replacement status of the TLB entries is updated. If the TLB does not contain the ⟨virtual, physical⟩ address pair required for translation, the virtual address is passed along to the address translator to determine the physical address. This translation is performed, as discussed in Section 2.2, by using the high-order bits of the virtual address as an entry into the segment and page tables. The address pair is returned to the TLB (possibly replacing an existing TLB entry).

From the cache's viewpoint, the physical address formed consists of two components: the block-frame address and the byte within the block. The block-frame address is used to search the cache directory. If there is a *match* (cache hit), the block (or part of it) containing the target locations is copied from the RAM portion of the cache into a shift register. Concurrently, the replacement status of the cache entries is updated. The shift register is shifted to select the target bytes which are transmitted to the CPU. If a miss occurs during the fetch operation, the block is fetched from main memory by using the physical address. The fetched block is stored in the cache and also passed to the shift register for selection of the target bytes.

Recall that, in a paging system, a *miss* in main memory (page fault) necessitates a context switch to another runnable process because the time to service a page fault is usually much greater than the context switch overhead. In a cache, however, the time to service a miss is comparatively smaller than the context switch time and, because of the small size of the cache, misses are more frequent than page faults. Also, context switching will invariably cause the new process to encounter initial cache misses in an attempt to restore its "locality set" in the cache. For these reasons, the process does not context switch on a cache miss.

**Design aspects** In general, the design of a cache is subject to different constraints and trade-offs than that of main memory. One of the important parameters in the

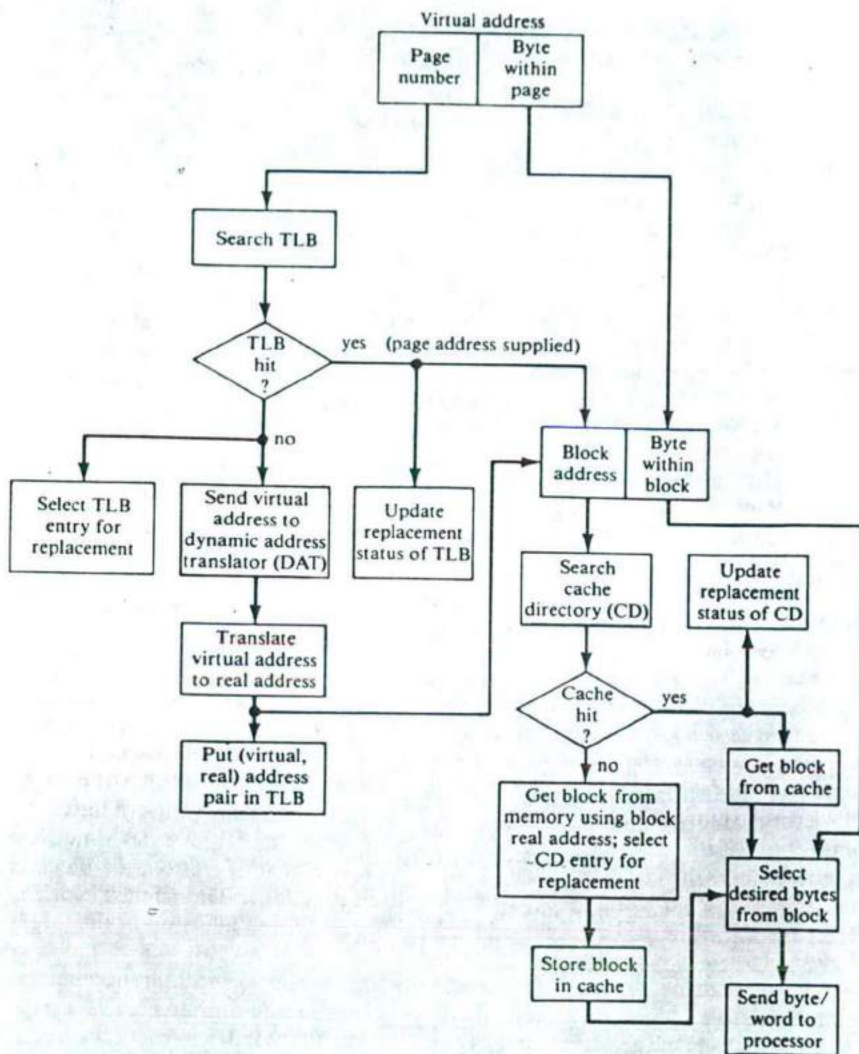



Figure 2.22 Simplified flowchart of cache operation for a fetch.

design of a cache memory is the *placement policy*, which establishes the correspondence between the main memory block and those in the cache. Other organizational parameters are the *fetch policy*, the *replacement policy*, the *main memory update policy*, *homogeneity*, the *addressing scheme*, cache and block sizes, and the *cache bandwidth*. The main memory update policy decides the time the information in memory is to be updated once the processor has requested a modification of the information. The fetch policy denotes how, when, and what information is to be fetched into the cache. The cache could be partitioned into several independent caches to segregate various types of references. An unpartitioned cache is said to be homogeneous. The cache could be multiported so that two or more requests can be made to the cache concurrently. In this case, a priority algorithm must exist to select one of the arbitrating requests. Furthermore, the cache accesses can be pipelined as in many mainframes, so that more than one cache access can be in progress concurrently. For example, four cache requests can each be in a unique phase of completion if the cache cycle is partitioned into four segments as follows: priority selection, TLB access, cache access, and replacement status update.

**Cache bandwidth** The cache bandwidth is the rate at which information can be transferred from or to the cache. The bandwidth must be sufficient to support the rate of instruction execution and I/O. The bandwidth can be improved by increasing the data-path width, interleaving the cache for concurrency, and decreasing the access time. The cache bus width affects the cost, reliability, and throughput of the system. An increase in bus width increases the access time because of packaging problems and additional gate delays because of line drivers and receivers. It also diminishes the signal-switching noise immunity. However, the wider the bus, the faster the data transfer. The number of fetches to main memory required to load a block of a given size depends on the bus width. Interleaving the cache can keep the bus width low while maintaining the bandwidth.

**Effects of multiprogramming** Most cache-based concurrent computers are, in fact, multiprogrammed. In most cases, each process gets to use the processor for a time slice or quantum in a round-robin fashion until the process terminates. Because of the alternate use of the processor, a significant fraction of the cache misses is due to the loading of the data and instructions for the new process which is assigned to the processor at the end of an intertask interval. This assumes that there is only one context in the cache in a given time slice and cache is purged at the end of the time slice. However, if the cache contains multiple contexts, a context switch may still increase the "cold-start" miss ratio because some of the new process's context may have been displaced in the cache. Note that the contexts can be distinguished in the cache by augmenting the address space of each context with a unique address space identification or process identification.

The problem of high cold-start miss ratios can be alleviated in a number of ways. A large cache can be used so that several processes' contexts will exist in it simultaneously. The scheduling policy can be modified to give priority to a task most likely to have its context in the cache. The time slice can be increased so that

the frequency of task switches is reduced and a task, once assigned to the processor, will get a chance to reach a steady-state ("warm-start") miss ratio before the next context switch. Another solution is to save the process's context in main memory on a context switch and reload it *en masse* the next time it is assigned to the processor.

**Data consistency** The problem of having several different copies of the same block in a system is referred to as the *cache coherence* or data consistency problem. This problem exists in a uniprocessor with cache when the processor can be active after modifying a word in the cache and before the copy in memory has been updated. The effect of the main memory update policy on data consistency will be discussed in Section 2.4.3. If the processor is the only unit to access memory, then the coherence problem is a mere theoretical observation without practical bearing on the correctness of the program execution. However, practical systems contain I/O units which require access to the memory. The method in which the I/O unit accesses the memory in a system with cache may create consistency problems, as will be seen in Section 2.5. In a multiple processor system with caches, the data consistency problems may also exist between caches. Solutions to such coherence problems will be discussed in detail in Chapter 7.

### 2.4.2 Cache Memory Organizations

The cache is usually designed to be user-transparent. Therefore, in order to locate an item in the cache, it is necessary to have some function which maps the main memory address into a cache location. For uniformity of reference, both cache and main memory (MM) are divided into equal-sized units, called blocks in the memory and block frames in the cache. The placement policy determines the mapping function from the main memory address to the cache location.

**Placement policies** There are basically four placement policies: *direct*, *fully associative*, *set associative*, and *sector* mappings. In discussing the mapping functions, we will consider a specific running example in which each processor's cache is of size 2K (2048) words with 16 words per block. Thus the cache has 128 block frames. Let the main memory have a capacity of 256K words or 16,384 blocks. The physical address is representable in 18 bits.

***Direct mapping*** This is the simplest of all organizations. In this scheme, block $i$ of the memory maps into the block frame $i$ modulo 128 of the cache. The memory address consists of three fields: the tag, block, and word fields, as depicted in Figure 2.23. Each block frame has its own specific tag associated with it. When a block of memory exists in a block frame in the cache, the tag associated with that frame contains the high-order 7 bits of the MM address of that block. When a physical memory address is generated for a memory reference the 7-bit block address field is used to address the corresponding block frame. The 7-bit tag address field is compared with the tag in the cache block frame. If there is a match, the information in the block frame is accessed by using the 4-bit word address field.

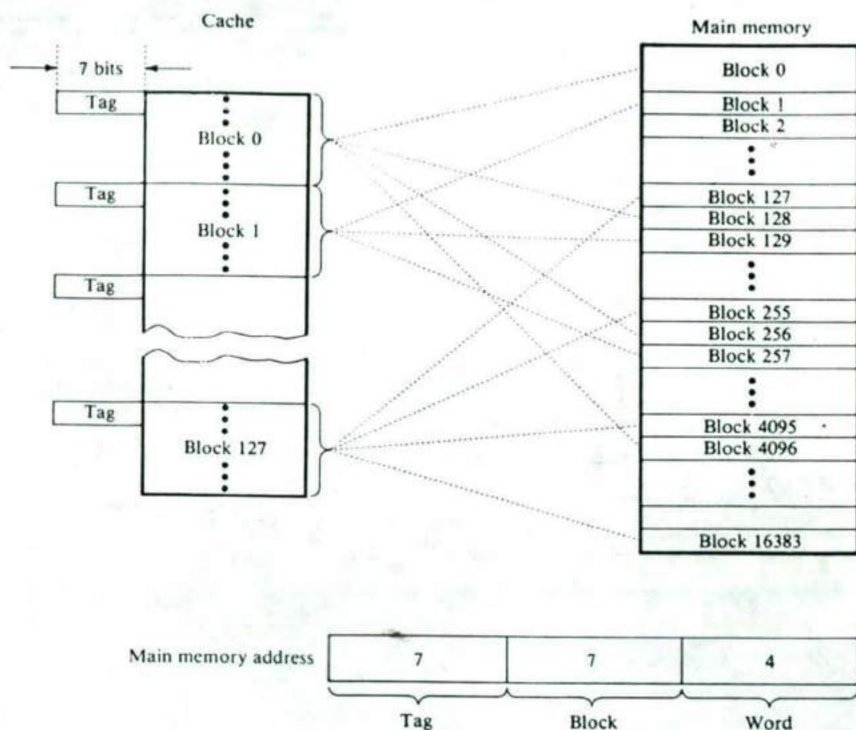



**Figure 2.23 Direct mapping cache organization.**

This scheme has the advantage of permitting simultaneous access to the desired data and tag. If there is no tag match, the output data is suppressed. No associative comparison is needed and, hence, the cost is reduced. The direct mapping cache also has the advantage of a trivial replacement algorithm by avoiding the overhead of record keeping associated with the replacement rule. Of all the blocks that map into a block frame, only one can actually be in the cache at a time. Hence, if a block caused a miss, we would simply determine the block frame this block maps onto and replace the block in that block frame. This occurs even when the cache is not full.

A disadvantage of direct-mapping cache when associated with a processor is that the cache hit ratio drops sharply if two or more blocks, used alternately, happen to map onto the same block frame in the cache. The possibility of this contention may be small in a uniprocessor system if such blocks are relatively far apart in the processor-generated address space. The possibility of this contention in a multiple-stream shared cache system may be much higher than that in a uniprocessor because many concurrently active streams are sharing the cache. The instruction cache in the IBM System/370 Model 158 uses direct mapping.

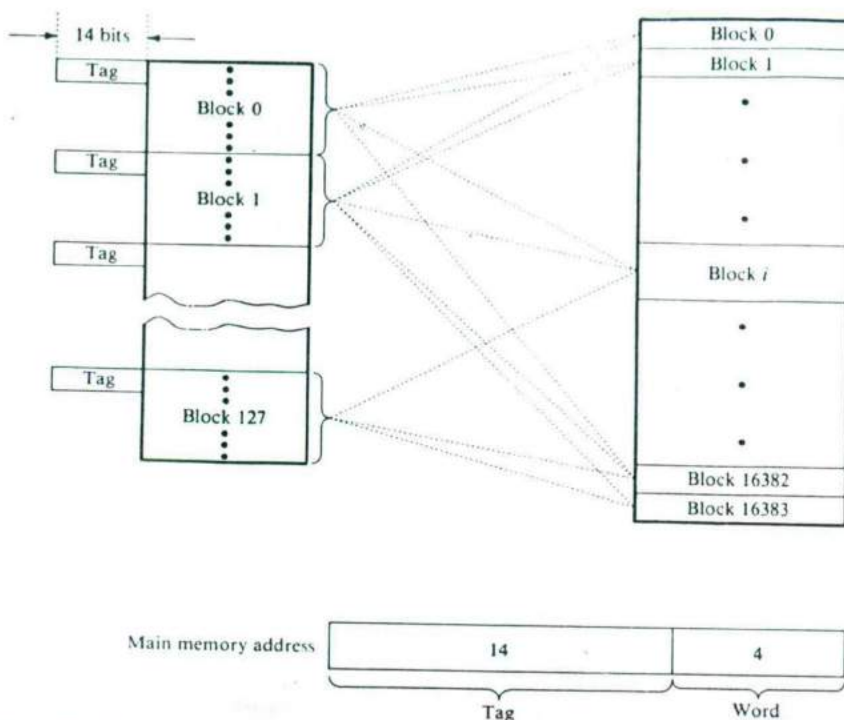



**Figure 2.24 Fully associative cache organization.**

***Fully associative*** In terms of performance, this is the best and most expensive cache organization. The mapping is such that any block in memory can be in any block frame. When a request for a block is presented to the cache, all the map entries are compared simultaneously (associatively) with the request to determine if the request is present in the cache. In the running example, 14 tag bits are required to identify the memory block when it is present in the cache. Figure 2.24 illustrates the fully associative buffer. The mapping flexibility permits the development of a wide variety of replacement algorithms, some of which may be impractical. Although the fully associative cache eliminates the high block contention, it encounters longer access time because of the the associative search.

***Set associative*** This represents a compromise between direct- and associative-mapping organization. In this scheme, the cache is divided into $S$ sets with $E = M/S$ block frames per set, where $M$ is the total number of block frames in the cache. A block $i$ in memory can be in any block frame belonging to the set $i$ modulo $S$, as shown in Figure 2.25 for the running example, where $M = 128$ and $S = 64$.

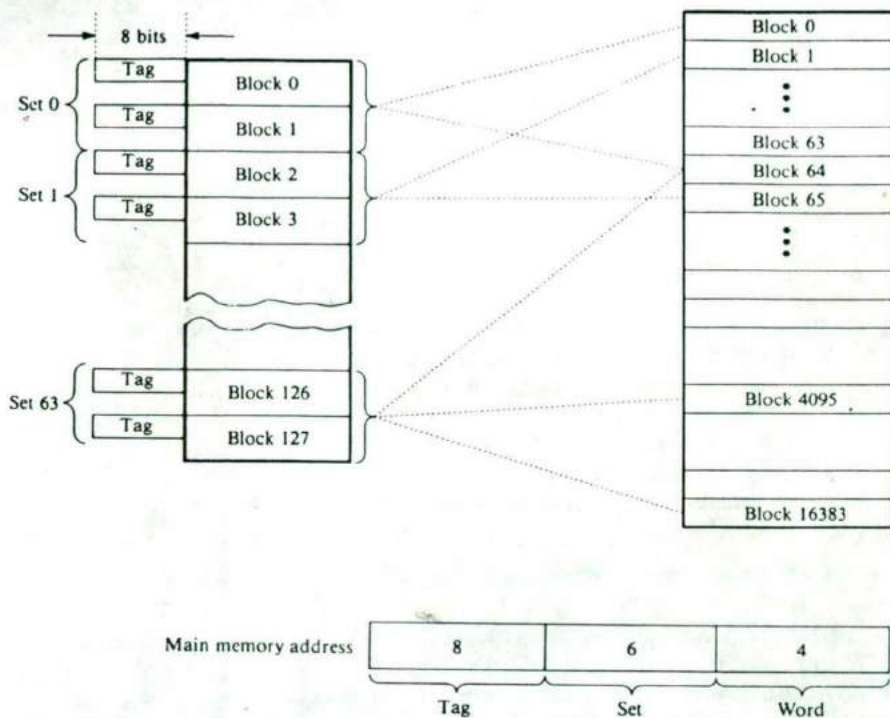



**Figure 2.25 Set associative cache organization.**

Several possible schemes are used for mapping a physical address into a set number. The simplest and most common is the *bit-selection* algorithm. In this case, the number of sets $S$ is a power of 2 (say $2^k$). If there are $2^j$ words per block, the first $j$ bits, $0, \ldots, j-1$, select the word within the block, and bits $j, \ldots, j+k-1$ select the set via a decoder. Hence, for the example, the 6-bit set field of the memory address defines the set of the cache which might contain the desired block, as in the direct-mapping scheme. The 8-bit tag field of the memory address is then associatively compared to the tags in that set. If a match occurs, the block is present. The cost of the associative search in a fully associative cache depends on the number of tags (blocks) to be simultaneously searched and the tag field length. The set-associative cache attempts to cut this cost down and yet provide a performance close to that of the fully associative cache. For this reason, it is the most commonly used placement policy for cache memories.

The main consideration in choosing the values for $S$ and $E$ depends on directory lookup time, cost, miss ratio, and addressing. Note that $S$ and $E$ are inversely related, assuming a constant $M = SE = 2^m$. The set size defines the degree of

associative search and thus the cost of the search. The addressing scheme used can indicate whether an overlap in cache lookup and the translation operation (via TLB) is possible in order to reduce the cache access time. Recall that the only address bits of a virtual address that get mapped in a virtual memory system are the ones that specify the page address. In order to illustrate how an overlap may occur, assume that there are $2^j$ bytes per block and $2^k$ sets in the cache. Let the page size be $2^p$ bytes. Assuming bit selection mapping, $p - j$ bits are immediately available to choose the set, since the low-order $p$ bits, which specify the byte within the page, are invariant with respect to the mapping. It is quite advantageous to make $p - j \geq k$ so that the set can be selected immediately, in parallel with the translation process. This overlap is shown in Figure 2.26. However, if $p - j < k$, then the search for the cache block can only be narrowed down to a small number, $2^{(k-p+j)}$, sets.

The effect $S$ and $E$ have on the miss ratio can only be measured by trace-driven simulation on a typical work load. However, Smith (1978) derived a relationship between the miss ratio for fully associative and set associative cache organizations. Assuming an LRU stack-programming model with coefficients drawn from the linear paging model, it was shown that the ratio of the miss ratios between the set associative and the fully associative cache is

$$R(E, S) = \frac{E - 1/S}{E - 1} \qquad \text{for } E \geq 3 \tag{2.26}$$

Figure 2.27 shows an example on the effect of $S$ and $E$ on the miss ratio. Experimental results have shown for uniprocessors that a set size in the range of 2 to 16 performs almost as well as fully associative mapping at little cost increase over direct mapping. Notice that when $E = m$, it is the fully associative mapping and when $E = 1$, it is the direct-mapping scheme. Table 2.2 shows some examples of systems that use set-associative cache and the choices of $S$ and $E$.

***Sector mapping*** In this scheme, the memory is partitioned into a number of sectors, each composed of a number of blocks. Similarly, the cache also consists of sector frames, each composed of a set of block frames. The memory requests are for blocks, and if a request is made for a block not in cache, the sector to which this block belongs is brought into the buffer with the following constraints: A sector from memory can be in any sector in the buffer, but the mapping of blocks in a sector is congruent. Also, only the block that caused the fault is brought into the cache, and the remaining block frames in this sector frame are marked invalid. A valid bit is associated with each block frame to indicate the blocks in a sector that have been referenced and hence retrieved from memory. Figure 2.28 illustrates the sector-cache organization for the running example with 16 blocks per sector and, hence, eight sector frames in the cache. This cache also attempts to reduce the cost of the map since it requires relatively few tags, which permits simultaneous comparisons with all tags. The IBM System/360 Model 85 has a sector-cache organization with 16 sectors and 16 blocks per sector.

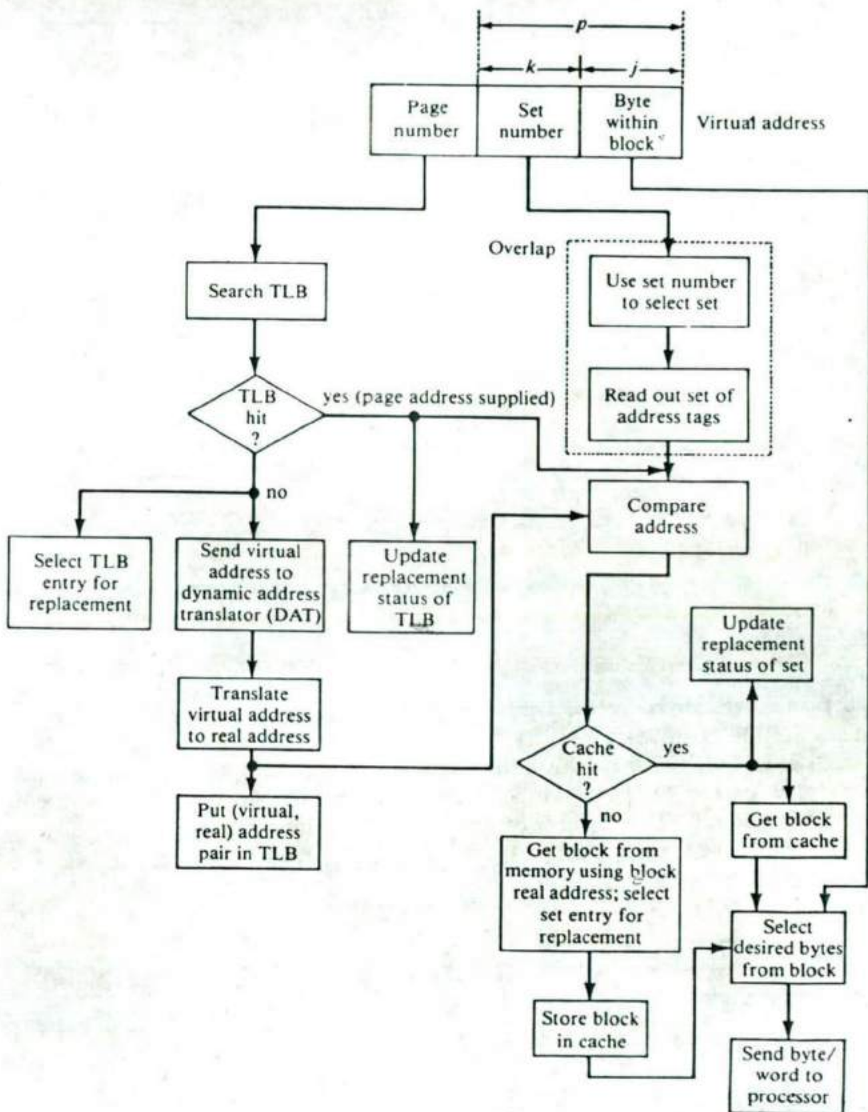



Figure 2.26 Set associative cache operations.

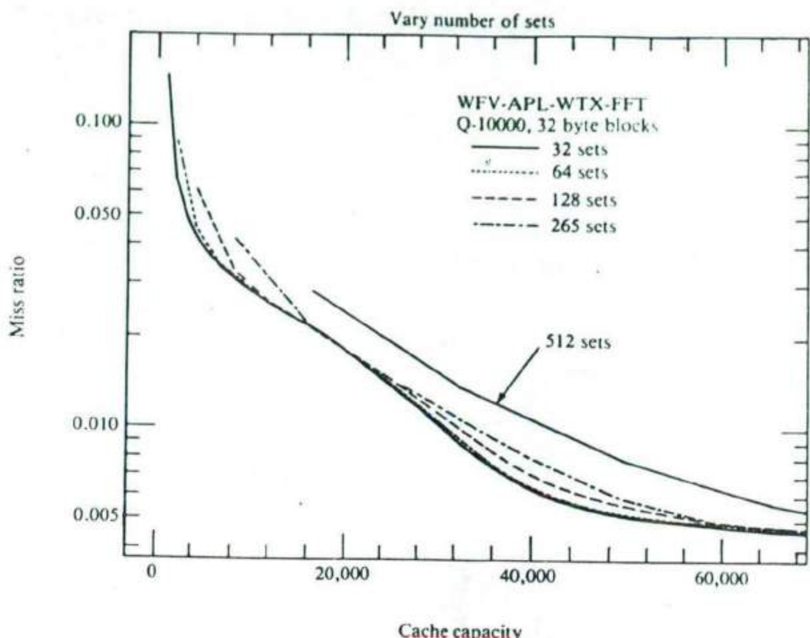



Figure 2.27 Miss ratios as a function of the number of sets and cache capacity (Smith, *ACM Surveys*, 1982).

**Block and cache size selection** In caches, a block is so small that spatial locality effects are the main consideration in the choice of the block size. The effect of cache size and block size on the hit ratio relates to spatial and temporal localities. For a given cache size, the miss ratio improves as the block size increases, because an increase in the block size captures more of the spatial locality. This improvement is achieved to the detriment of the temporal locality, because the total number of

**Table 2.2 Typical values of $S$ and $E$ in example systems**

| System | $(S, E)$ |
|---|---|
| Amdahl 470v/6 | (2, 256) |
| Amdahl 470v/7 | (8, 128) |
| Amdahl 470v/8 | (4, 512) |
| IBM 370/168-3 | (8, 128) |
| IBM 3033 | (16, 64) |
| DEC VAX 11/780 | (2, 512) |
| Honeywell 66/60 | (4, 128) |

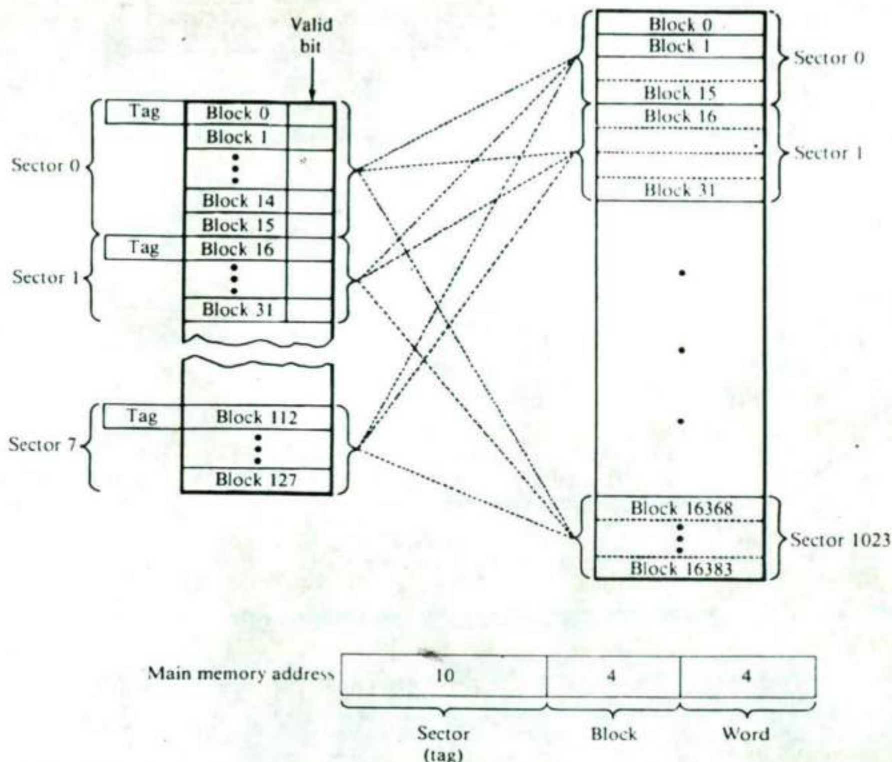
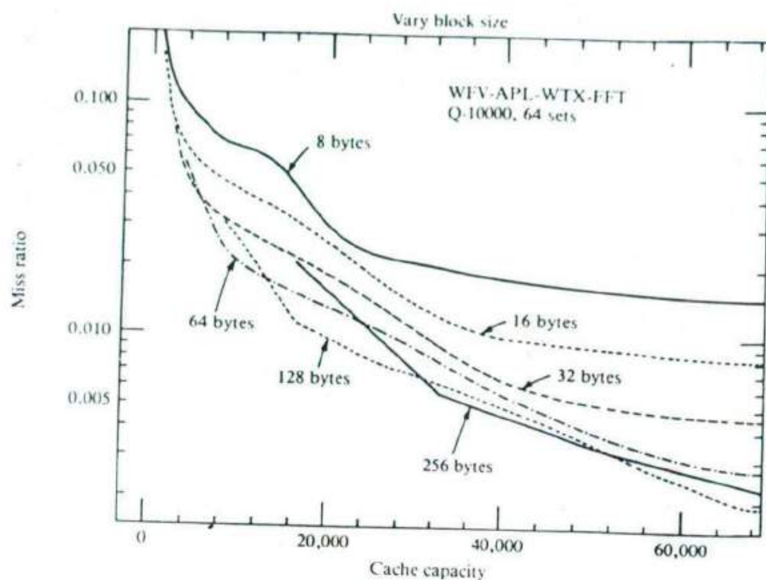


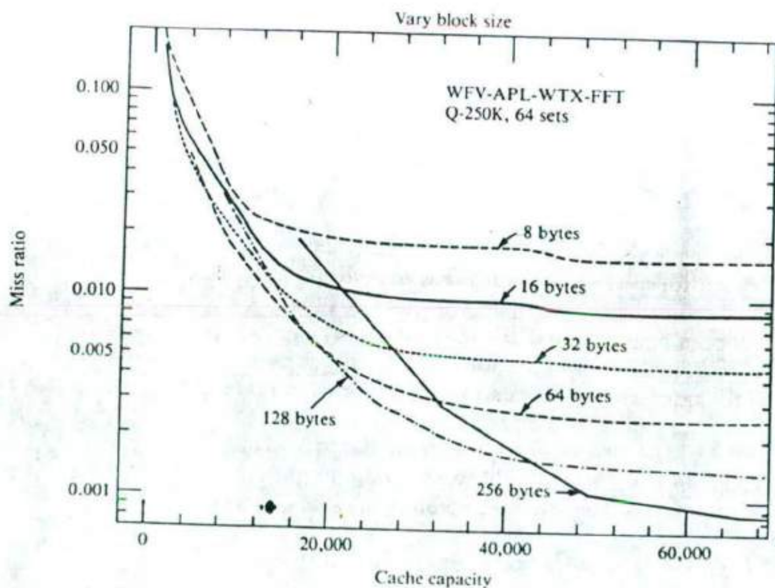

**Figure 2.28 Sector mapping cache.**

blocks in the cache is diminished proportionally. At some point, the miss ratio curve levels off for a block size such that the effects of a block-size increase on both localities compensate each other. Beyond this block size, most of the spatial locality has been captured and the miss ratio curve inflects, as the cache is not capable of holding the temporal locality of the program.

When the number of blocks in the cache is so small that blocks are swapped constantly between the cache and the memory, the efficiency of the cache goes to zero. The block size corresponding to the minimum miss ratio depends not only on the cache and its organization but also on the program behavior or work load. Because properties of programs vary widely, the choice of block and cache sizes must result from extensive simulations based on traces of programs constituting a representative work load. In Figure 2.29, we show two example miss ratio curves for a given work load in which the time slice $Q$ is varied and represented in a number of memory references. It can be seen that $Q$ also affects the selection of the block and cache sizes.




(*a*) Quantum size, $Q = 10{,}000$ references




(*b*) Quantum size, $Q = 250\text{K}$ references

**Figure 2.29 Miss ratios as a function of block size and cache capacity (Smith, *ACM Surveys*, 1982).**

For a given block size, a cache size increase is accompanied by an improvement in the miss ratio as more of the temporal locality is contained in the cache. Finally, the size and behavior of supervisor programs are a major factor in the selection of the cache size, as the supervisor typically uses the processor 25 to 60 percent of the time.

**Practical translation lookaside buffers** The TLB is typically designed as a small set-associative memory. However, the design is somewhat different from a set-associative cache. Unlike the set-associative cache, the input to the TLB is the virtual page number with no offset. However, if bit-selection algorithms were used, the low-order TLB entries would not be used efficiently. Therefore, the virtual page number is first randomized by hashing and the result applied as input to the TLB. Hashing can be performed rapidly by a set of exclusive-or logic on some bits of the virtual page number. However, the overhead due to hashing can be avoided, as in the VAX 11/750, which has a TLB with 256 sets and a set size of two. In this case, the set is selected with the high-order address bits and five low-order bits of the page number. Other examples of systems with TLB are the IBM 3033 (with 64 sets and two elements each), Amdahl 470V/6 (with 128 sets and two elements each), and the Amdahl 470V/7 and V/8 (with 256 sets and two elements each).

During a context switch, the entries of the TLB become invalid if only one context's *address-space identifications* reside in the TLB. Hence, the TLB is purged or invalidated at context switch. The purge operation, usually initiated by a privileged instruction, may create and extra overhead. For example, in the IBM 3033, it takes 16 machine cycles to purge the TLB.

**Virtual versus physical addressing** We have seen that the cache is addressed using the physical address. Although in the set-associative caches the translation of the virtual address can be overlapped with the lookup in the cache, the lookup cannot be completed until the physical address is available. The cache access time could be significantly reduced if the translation step could be eliminated. In this case, the cache would have to be addressed directly using the virtual address. The major problem with using virtual addresses to access the cache is that these names are defined within a process. Two processes may know the same physical word under different virtual names. Conversely, the same virtual name may designate different blocks for different processes. If the processor is multiprogrammed, a context switch is accompanied by a cache sweep or purge. Otherwise, the new process may issue a reference with a virtual name which will hit on a block that had the same name for the previously running process. This problem can be avoided by augmenting the virtual address with an address-space identification, which makes it unique.

However, there is still a possibility that several copies of the same physical block may exist in the cache under different names. This is called the *synonym problem* and causes coherence problems within the cache. If a block is shared among several active processes, several of its copies can be present in the cache under

different names. The solution is to avoid multiple copies of the same block in the same cache by detecting the synonym when it occurs and enforcing consistency. Synonyms can be detected by mapping the virtual address into a physical address via a TLB and determining if there exist other virtual addresses in the cache that have the same physical address. This can be accomplished by a mapping device that is inverse to the TLB and is called an *inverse translation buffer* (ITB). The ITB is accessed on a physical address and indicates all the virtual addresses associated with that physical address that is in the cache.

To reference memory, the virtual address is applied to the TLB at the same time as the cache. If a miss occurs in the cache, the physical address obtained from the TLB is used to request a fetch of the block from main memory. Simultaneously, the physical address is also used to search the ITB to determine if that block is already in the cache under a different name (virtual address). If it is, the virtual address is renamed and moved to its new location to avoid multiple copies of the same block for consistency reasons. Also, the block-fetch request to memory is discarded. Otherwise, the block fetched from memory is used and the ITB and cache updated accordingly. The addressing of the cache by virtual addresses may decrease the cache access time on a hit at the cost of increased hardware complexity.

**Partitioned cache** Another issue in the design of a cache is the partitioning of the cache into several independent caches in order to segregate various types of references. Usually, segregation is limited to reference types that are hardware-detectable: for example, instructions versus data, or references in user mode versus references in supervisor mode. It could be extended to compiler-imposed segregation, in which references could be tagged at compile time. This extension, however, violates the principle of cache transparency, which simplifies the compiler. Splitting the cache generally improves the cache bandwidth and access time.

In a pipelined system, the processor is usually physically partitioned into two units, the I unit and the E unit. The I unit performs instruction fetch and decode and forwards the decoded instruction to the E unit, which executes it. In the execution phase, the E unit may fetch and store operands. By splitting the cache into data (D) and instruction (I) caches, the I cache (D cache) can be placed next to the I unit (E unit) to permit simultaneous access and reduce the access time. While one instruction is being fetched from the I cache, another instruction in the E unit can be accessing its operands from the D cache.

It is generally known that a significant fraction of misses is due to task switching for the execution of supervisor tasks. In order to reduce these miss transients, the cache can be split between a user cache and a system cache. Depending on the mode of execution, one cache or the other is referenced. Note, however, that the supervisor cache may still have a high-miss ratio because of its large working set.

The most obvious problem with split-cache organization is the consistency problem, because two copies of an information may now exist in separate caches. For example, in a pipelined processor, instructions being modified by the E unit must be stored in the I cache before they can be fetched. However, the E unit can access the D cache. Even if we assume that programs are not self-modifying, a

cache block may contain instructions and data. Presumably, this effect can be minimized by designing compilers to insure that instructions and data are in separate blocks. Another problem with split cache results in possible inefficient use of cache memory. Locality properties of instructions and data are not homogeneous in this case. The miss ratio may increase as a result of splitting the cache. However, this depends on the work load. Examples of systems with split cache are the S-1 and the Amdahl 580.

### 2.4.3 Fetch and Main Memory Update Policies

As discussed earlier, this policy is used to decide when and what information to fetch into the cache. There are three basic types of fetch policies which are applied to cache: *demand*, *anticipatory*, and *selective* fetches. Demand and anticipatory fetch techniques used for paging systems can be applied to caches. In selective fetch, some information, such as shared writeable data, may be designated as unfetchable; further, there may be no fetch-on-write when a miss occurs, as discussed below.

Prefetching can be successfully used to prefetch the needed blocks ahead of time so that the cache miss ratio can be reduced. The major factor in determining the usefulness of prefetching in a cache is the block size. It has been found that a block size of less than 512 bytes results in useful prefetching. Only the OBL prefetch algorithm is usually considered because of its ease of implementation at cache speeds. Several possibilities exist for deciding when to initiate a prefetch. For example, for all $i$, prefetch block $i + 1$ if a reference is made to block $i$ for the first time.

This technique, termed *always prefetch*, while good in reducing the miss ratio, creates more traffic to the cache and main memory. In multiprocessor systems, this may be detrimental. A refined technique is to prefetch block $i + 1$ only on a miss to block $i$. Yet another technique is *tagged prefetch* which, in addition to prefetching on a miss, also prefetches block $i + 1$ if a reference to a previously prefetched block $i$ is made for the first time. Prefetching has been found to be very effective in pipelined systems such as the Amdahl 470 V/8, which uses *prefetch on a miss*.

One technique used to reduce the wait time of the processor during the fetching of a missed block is to forward the requested word directly to the processor first and then complete the fetching of the block in a wraparound fashion. This technique is called *load-through* or *read-through*.

The time when a word in memory is updated after a write depends on the write policy. One possibility, *write-through* (WT), is to update directly the memory copy of the data word. In this case, the copy of a block in the cache is never different from its copy in memory. Two variations of write-through are possible. The first is the *write-through-with-write-allocate* (WTWA) policy, in which a block is loaded into the cache on a write-miss. In WTWA, both read and write references contribute to the hit ratio. The second possibility is the *write-through-with-no-write-allocate* (WTNWA) policy, in which blocks are loaded into the cache on

read-misses but not on write-misses. In WT, the effectiveness of the cache is limited by the fact that 5 to 30 percent of all memory references are write operations. When no buffer is provided at the memory, the processor is blocked during the write-through. In general, one can consider that the memory address and data registers form a buffer of size one. If another write-through or a miss occurs when a previous write has not been completed, the processor is blocked.

In order to estimate the effect of the write policies on the average memory access time, let $\omega_t$ be the fraction of writes in the system and assume a nonread-through fetch policy. Also, let $t_c$, $t_m$, and $t_B$ be the cache cycle time, the memory cycle time ($t_c < t_m$), and the block transfer time, respectively. Assuming a WTWA policy, the *average time* to complete a reference when no buffer is present is

$$t_c + (1 - h)t_B + \omega_t(t_m - t_c) \tag{2.27}$$

Note this assumes that the writes to cache and main memory are performed simultaneously in the case of a hit and the miss ratio is $1 - h$. For WTNWA policy, the *average cycle time* is

$$t_c + (1 - h)(1 - \omega_t)t_B + \omega_t(t_m - t_c) \tag{2.28}$$

Note that the hit ratios in Eqs. 2.27 and 2.28 are not equal because of the difference in block-frame allocation. However, both equations have the same lower bound $t_c + \omega_t(t_m - t_c)$, which occurs when $h = 1$. This lower bound limits write-through policies to low-performance caches.

To improve the effectiveness of WT, increased buffering must be provided at the memory. The processor stores the write request in a FIFO buffer and then proceeds. To take full advantage of the buffering capability, block transfers due to cache misses should have a higher priority in accessing memory than write requests. Additional hardware checks the write buffer to ensure that a block requested by the processor does not contain any word waiting in the FIFO buffer. If the buffer contains such a word, the block is updated accordingly before it reaches the cache. At the limit, if the buffer has infinite size, the processor never waits for the completion of a write request, and thus the cache with WT policy can potentially achieve an *average reference time* of

$$t_c + (1 - h)t_B \tag{2.29}$$

The alternate policy is *write-back* (WB). The WB always allocates a cache block frame on a miss. When a write-hit occurs, only the block in the cache is modified. The memory update takes place when a block is replaced and swapped back to memory. Note that since the block-frame allocations in WB and WTWA are the same, the hit ratios are equal for both policies. The policy in which all replaced blocks are written back to memory is called *simple write-back* (SWB). Of course, the SWB results in many redundant swaps, during which the processor waits. To improve performance, a replaced block is written back only if it has been modified. A "dirty" bit is included in the directory with the tag of each cache frame. The dirty bit is reset when the block is loaded in the cache and is set when any word of the block is modified. This strategy is called *flagged write-back* (FWB).

and it increases performance at low cost by reducing the average time the processor waits on a cache miss.

To further improve the performance, the write-backs on misses have to be buffered. In a technique called *flagged register write-back* (FRWB), the modified block selected for replacement is first written in a fast register to avoid interfering with the fetch. The new block is then brought into the freed cache block frame. The block write-back to the memory is activated later and is completed "in the background." In an extension to this policy, the blocks to write back could be buffered, as the modified words are in WT. The fetching of blocks from memory to cache on misses is given higher priority, and special hardware checks for the possible presence of a requested block in the write-back queue. The cost effectiveness of such an extension depends on the relative improvement achieveable beyond FRWB.

As for WT, we can estimate the efficiency of write-back strategies for some special cases. For SWB, the *average reference time* is

$$t_c + 2(1 - h)t_B \tag{2.30}$$

For FWB, it is

$$t_c + (1 - h)(t_B + w_b t_B) = t_c + (1 - h)(1 + w_b)t_B \tag{2.31}$$

where $w_b$ is the probability that a replaced block has been modified.

The comparison between WT and WB is rather complex and depends on the program behavior. However, three major factors influence the effectiveness of WT or WB in a given system: the extent of memory traffic, data consistency, and reliability. WT usually results in more memory traffic, which can be very detrimental to the performance of a system with multiple processors. However, when the WT is used, main memory is always consistent with the cache, since the memory always has the updated copy of the data. Thus the failure of a processor and its cache permits recoverability.

### 2.4.4 Block Replacement Policies

When a miss occurs in a cache and a new block has to be brought in, a decision must be made as to which of the old blocks is to be overwritten if the cache is full. Various *replacement algorithms* have been proposed to select the block to be displaced. The property of locality of reference in programs gives a clue as to the types of algorithms that may result in a reasonable strategy. We would expect that a good replacement rule would appropriately treat a program's pages, depending on their reference probabilities. Since the cache has a small size it is generally overcommitted; that is, it is generally impossible to keep the working set of even one program in the cache at a time, except for some systems, such as Amdahl 470 V/8 and IBM 3033, with a 32K or more byte cache. Also, because of the constrained mapping mechanisms, only fixed-space replacement algorithms are generally considered. For example, in the set-associative cache, the block to be replaced is within a set and thus the replacement algorithm is invoked for block frames within that set.

Examples of commonly used fixed-space policies are *least recently used* (LRU) which, at a cache miss, replaces the least recently-referenced block of the resident set; *first-in, first-out* (FIFO) which, at a miss, replaces the longest resident block; and *random* (RAND) which, at a miss, replaces a randomly chosen block from the set in the cache. It has been found that, on the average, LRU performs better than FIFO or RAND and is therefore preferred.

We will discuss the implementation of the LRU policy, which is used very often as a cache-replacement policy. Associated with an instance in this policy is a dynamic list, called the *LRU stack*, in which is arranged the referenced block-frame numbers from top to bottom by decreasing recency of reference. At a block replacement time, the LRU policy chooses the lowest-ranked block-frame number in the stack. Each time a block frame of the cache is referenced, the stack is updated by moving the referenced block-frame number $x$ to the top and pushing down the intervening blocks by one position, thereby giving $x$ a new lease on its life in the cache. The position at which the referenced block frame $x$ was found in the stack before being promoted to the top is called the *stack distance* of $x$.

For small set sizes, the LRU policy may be implemented efficiently in hardware so as to operate at cache speeds. Three implementation schemes are discussed. The first scheme employs a set of fast counters, called *age registers*. Each age register is associated with every block in a set. As an example, consider the LRU stack implementation of a four-block set. Associate a 2-bit counter with each block, which can therefore count from 0 to 3. Each time a reference results in a hit and a block frame with count $j$ is referenced, its counter is reset to 0 and all other counters with a value less that $j$ are incremented by 1. The other counters are unmodified. If a reference results in a miss and the set is full, the block with counter value $j = 3$ is overwritten with the new block and its counter is reset to 0. The counters of the other three blocks are incremented by 1. The block with a counter value of 3 can be obtained by an associative search of the counters. If the set is not full when the miss occurs, the counter associated with the new block loaded from main memory is reset to 0 and all others incremented by 1. A little thought will show that the counter values of occupied blocks are always distinct.

A second implementation employs a set of $D$ flip-flops to maintain the status of the blocks which currently reside in the cache. A few logic gates can achieve the updating function. Since there is plenty of time to update the LRU stacks for cache misses, only the updating for hit requests is considered in the example. For a set containing $E$ blocks, that is, for a set of size $E$, $\log_2 E$ bits will be sufficient to address any given block in the set and a total of $E \log_2 E$ bits are enough to keep all the necessary information for LRU replacement operation. Figure 2.30 shows one example of an LRU stack with set size equal to 4. In this example, the four words of the stack are denoted as $X$, $Y$, $Z$, and $W$. Register $X$ corresponds to the top of the stack and register $W$ the bottom of the stack. Register $X$ contains the block number $X_1X_0$, register $Y$ contains the block number $Y_1Y_0$, and so on.

The number of the block just accessed is available on lines $I_1$ and $I_0$, and the number of the least-recently-used block is available as $W_1W_0$. Three control signals, $NX$, $NY$, and $NZ$, are provided, each of which controls its corresponding

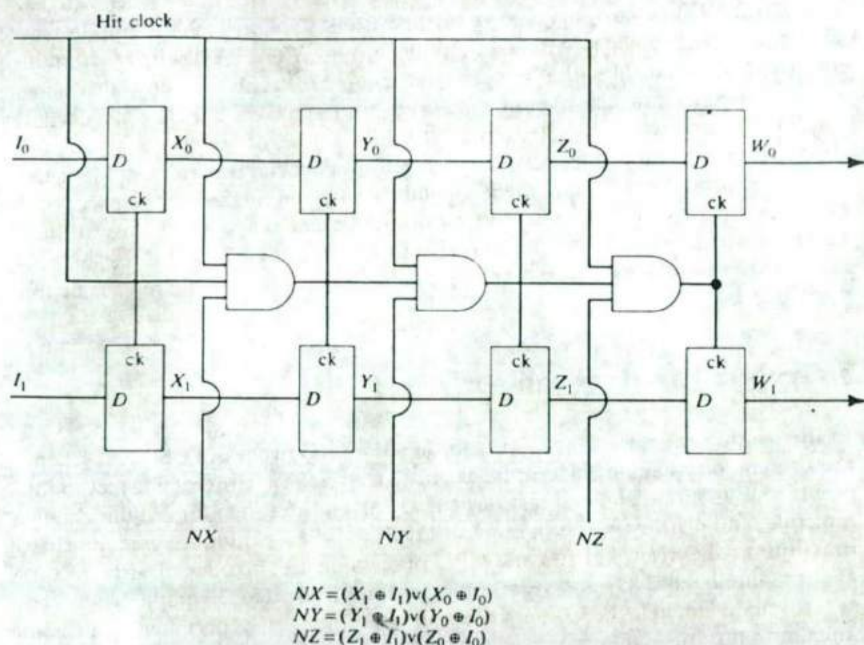



**Figure 2.30 An implementation of the LRU algorithm.**

block in the LRU stack. $NX$ is 1 if the block that has just been accessed is not block number $X$; otherwise, $NX$ is 0. The values of $NY$ and $NZ$ can be obtained similarly. Whenever a request results in a hit, a hit clock is generated immediately to control the updating process. Each of these three control signals, together with the hit clock, determine if the corresponding block should be shifted to the right in the LRU stack. The number of the block that has just been accessed is loaded into the leftmost pair of $D$ flip-flops every time a hit in this set occurs. The contents of the other pairs are shifted to the right until the previous position of the just-accessed block is reached. The rightmost pair of the $D$ flip-flops always indicates the number of the least recently used block in the set associated with this LRU stack.

A third implementation uses $E(E\text{-}1)$ active bits of status for a set with $E$ elements. These $E(E - 1)$ active bits are derived from an $E$-by-$E$ binary matrix in which the diagonal elements are passive and always zero. When the block in the $j$th block frame is referenced, the $j$th row of the binary matrix is first set to all 1's and then the $j$th column is set to all 0's. It is easy to show that, using such a scheme, the most recently used block is always the block in the block frame that has the largest number of 1's in its row. Similarly, the least recently used block is in the block frame with the smallest number of 1's in its row.

The three implementation schemes discussed above require a number of status bits that increase with the square of the set size. For a small set size (4 or 6),

it is acceptable. However, for machines with a large set size (8 for IBM 370/168-3; and 16 for IBM 3033), it may be too expensive and slow. In such systems, the set of elements is partitioned into nonoverlapping groups. The LRU group is determined and the LRU element within the group is selected for replacement. If this scheme were applied to a set size of 8, in which the groups consist of 2 elements each, the implementation would use 20 active status bits instead of 56.

It has been shown that, in general, the effect of cache replacement algorithms on the performance of the cache is secondary when compared to the effect of the mapping on performance. The fully associative cache is most sensitive to the replacement algorithm (and least sensitive to mapping), while the direct-mapping cache is the most sensitive to mapping (and least sensitive to the replacement algorithm).

## 2.5 INPUT-OUTPUT SUBSYSTEMS

In this section, we review techniques for handling I/O processing. Several schemes are presented to handle different types of I/O transactions. Interfacing methodologies for slow, moderate, and fast devices are given. Methods for handling single, multiple, and priority interrupt requests are discussed. Techniques used to achieve maximum concurrency of I/O and CPU processing are introduced. Architectures of some intelligent I/O subsystem controllers are presented. Example I/O processors discussed include the IBM channels, the CDC integrated peripheral processing units, and the Intel I/O processor.

### 2.5.1 Characteristics of I/O Subsystems

The performance of a computer system can be limited by compute-bound jobs or input-output (I/O) bound jobs. The emphasis in the following discussion is on the I/O problem and various techniques which can be used to manage I/O data transfer. An example I/O subsystem for a dual processor system is shown in Figure 2.31. The subsystem consists of I/O interfaces and peripheral devices. Sometimes the distinction between the device and its associated interface is fuzzy. The I/O interface controls the operation of the peripheral device attached to it. The control operations are initiated by commands from the CPU. The set of commands used to accomplish an I/O transaction is called the *device driver* or *software*. The functions of the interface are to buffer and perform data conversion into the required format. It also detects transmission errors and requests regeneration of an I/O transaction in case of error. Moreover, the interface can interrogate, start, and stop the device according to commands issued by the CPU. In some cases, the interface can also interrogate the CPU if an urgent attention is requested by the device. Not all interfaces possess these capabilities and many design options are available depending on the device characteristics. Below, we outline a few devices and their speed characteristics.

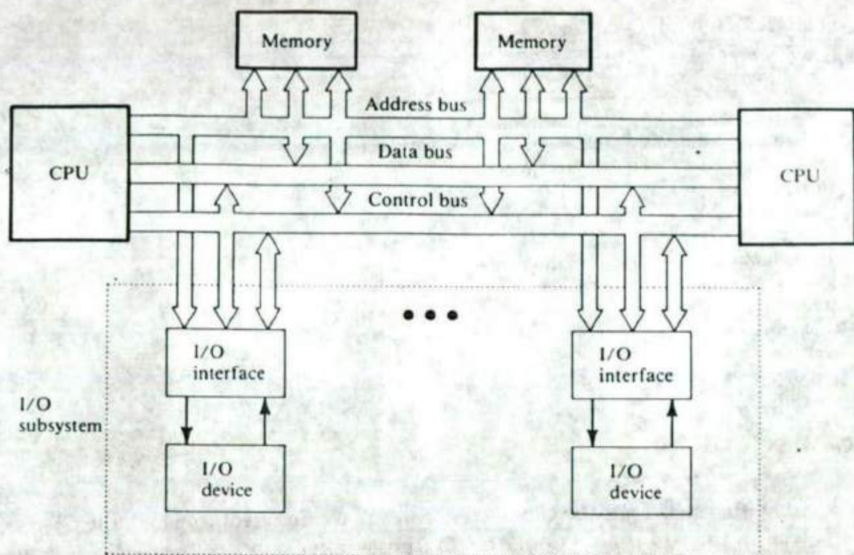



**Figure 2.31 I/O subsystem in a dual processor system.**

There are many different types of peripheral devices. Most of them are electromechanical devices and hence transfer data at a rate often limited by the speed of the electromechanical components. Table 2.3 shows some typical peripheral devices. Bubble memories, disk drums, and tape devices are mass storage devices which store data cheaply for later retrieval. Typical capacities of mass storage devices are: fixed-head and moving-head disks 512M bytes; floppy disks, 1M bytes; 9-track tape, 46M bytes; and cassette tape, from 64K to 512K bytes. Display terminals are input-output devices which consist of keyboards and cathode ray tubes (CRT). The keyboard acts as input while the CRT is the output display. In some cases where the CRT is replaced by a printer, the terminals are called teletypes.

Since terminals are often used interactively and are relatively slow devices, a reliable technique for transmitting characters between the processor and the terminal is *serial data transmission*. This method is cheaper than parallel transmission of characters because only one signal path is required. Data communication over long distances is usually done serially. For this reason, remote communication can be done over telephone lines by using a *modem* (*mo*dulator-*dem*odulator) interface. The modem is used at each end of the transmission line. There are a variety of character codes used in the transmission of data. However, one of the standard codes often used is the American Standards Committee on Information Interchange (ASCII), which uses seven-bit characters.

**Table 2.3 Some I/O devices**

| I/O device | Function | Data rate |
|---|---|---|
| Bubble memories | Mass storage | 300K, 4M cps |
| Charged-coupled devices | Mass storage | 500K, 4M cps |
| Disk | Mass storage | |
| Fixed head | | 300K, 2M cps |
| Moving head | | 30K, 1M cps |
| Floppy | | 25K cps |
| Display terminal | Input-output | 10-300 cps |
| Line printer | Output | |
| Impact | | 100-3000 lpm |
| Electrostatic | | 300-40,000 lpm |
| Ink jet | | 100-3000 lpm |
| Tape drive | Mass storage | |
| Reel to reel (7, 9 tracks) | | 15-300K cps |
| Cassette | | 10-400 cps |

I/O subsystems may be classified according to the extent to which the CPU is involved in the I/O transaction. An I/O transaction can be the transfer of a single bit, byte, word, or block of bytes of information between the I/O device and the CPU, or between the I/O device and the main memory. The simplest I/O architecture is one in which all processing is performed sequentially. In such systems, the CPU executes programs that initiate, test the status of the device, perform the data transfer, and terminate I/O operations. In this case, the I/O transaction is performed using *program-driven* I/O. Most computers provide this option, as it requires minimal hardware. However, as the action of the program-driven I/O is illustrated in Figure 2.32, the CPU can spend a significant amount of time testing the status of the device. This *busy-wait* feature of the program-driven I/O scheme has the disadvantage that the time required to transfer a unit of information between main memory and an I/O device is typically several orders of magnitude greater than the average instruction cycle. Therefore, even a moderate I/O transfer rate will significantly degrade the useful cycles of the CPU in performing actual computations. Hence, the system performance may be degraded significantly.

A solution to this possible degradation is to permit concurrent CPU and I/O processing. This can be achieved by a modest increase in the hardware complexity of the interface. As the degree of concurrency is increased, the complexity of the hardware will have to be increased to match the data transfer requirements. One scheme uses a "pseudo" program-driven I/O method. In this scheme, the CPU initiates the I/O transaction and resumes its regular computation. When the device is ready with the data, in an input operation, the device controller notifies the CPU of the presence of the data in the controller's buffer. The CPU can then service the device to retrieve the data. A similar description can be made regarding an output operation. The notification signal is referred to as an *interrupt request*. An interrupt capability relieves the CPU from the task of periodically testing the I/O device status.

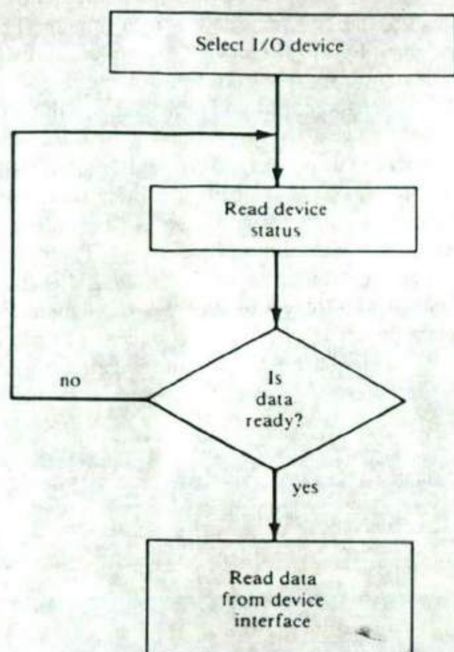



Figure 2.32 Programmed-driven I/O.

Although an interrupt request may arrive asynchronously during an instruction cycle, most processors permit the current instruction cycle in progress to be completed before the interrupt request is serviced. When an interrupt request is issued by a device to the CPU, the CPU may not be willing to accept the interrupt. It indicates its willingness (or unwillingness) to receive interrupts by setting (or resetting) an interrupt-enable flag in the CPU by executing an enable (or disable) interrupt instruction. This flag informs the device of the CPU's interruptibility status. When the CPU receives an interrupt, it acknowledges the interrupt by issuing an interrupt acknowledge signal to the device controller. At the same time, it saves the status of the interrupted process. The CPU then transfers control to a specified location in memory where the service routine of the device resides. The transfer of control is similar to a procedure call. The device is serviced and the status of the interrupted process is restored before its execution resumes. More details on interrupts will be given later.

The ultimate degree of concurrency in I/O processing can be achieved if the device controller is intelligent enough to perform the I/O transaction between the device and the main memory without the intervention of the CPU. This parallelism is very effective when a block of data is to be transferred. This requires the device controller to be capable of generating a sequence of memory addresses. However, the CPU is still responsible for initiating the block transfer. As an example, we illustrate a typical sequence of operations required to transfer a block

of data from a device to main memory. The CPU initializes a buffer in main memory which will receive the block of data after the I/O transaction is complete. The address of the buffer and its size are transmitted to the device controller, and the address of the required block of data in the device, is also given to the controller.

The CPU then executes a special "start I/O" command which causes the I/O subsystem to initiate the transfer. While the transfer is in progress, the CPU will be free to perform basic computations, thereby improving overall system performance. When the block transfer is complete, the CPU is notified. Notice that since the CPU and the controller share the main memory, the device will periodically "steal" memory cycles from the CPU to deposit the data in memory. The cycle-stealing is very effective since the devices are often slower than the CPU. When the CPU and the device controller conflict in accessing the bus or a memory module, the device is given priority over the CPU in the access since it is a more time-critical component. This type of I/O data transfer scheme is called *direct*

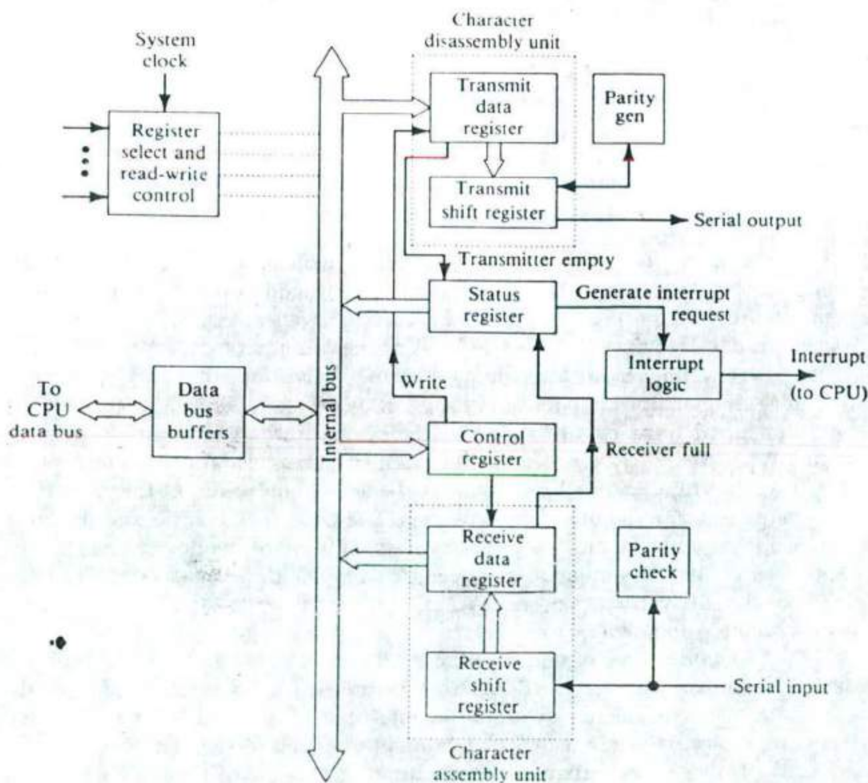



Figure 2.33 Simplified I/O interface for serial data.

*memory access* (DMA). The I/O controller often used for DMA operations is called an *I/O data channel*.

Notice that the DMA facility does not yield total control of the I/O transaction to the I/O subsystem. The I/O subsystem can assume complete control of the I/O transactions if a special unit, called an *I/O processor* (IOP) is used. The IOP has a direct access to main memory and contains a number of independent data channels. It can execute I/O programs and can perform several independent I/O transactions between main memory and devices or between two devices without the intervention of the CPU.

### 2.5.2 Interrupt Mechanisms and Special Hardware

An example of an interface used for slow I/O devices is the *universal asynchronous receiver-transmitter* (UART), often used in a microcomputer system. Its architecture is depicted in Figure 2.33. Its function is to buffer and translate between the parallel word format used by the CPU and the asynchronous serial format used by most slow-speed devices. The interface consists of addressable I/O registers or ports. The formats of the status and control registers are shown in Figure 2.34. The control register is a write-only register which is used to program the command specification. The status register contains the current state of the device and the outcome of the I/O transaction. Of importance is the device's busy-ready flag, which indicates whether the device is busy servicing an I/O transaction or is ready to receive the next transaction. This is the flag used when performing I/O transactions in busy-wait mode.

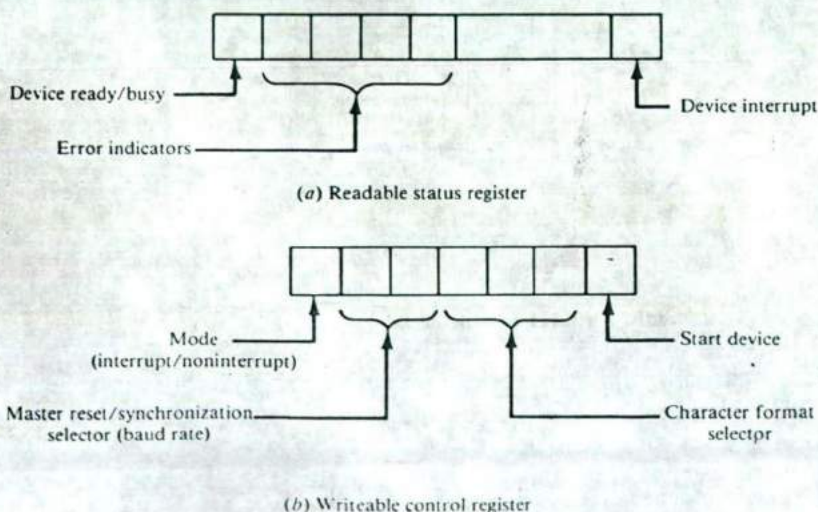



**Figure 2.34 Format of control and status registers in interface of Figure 2.33.**

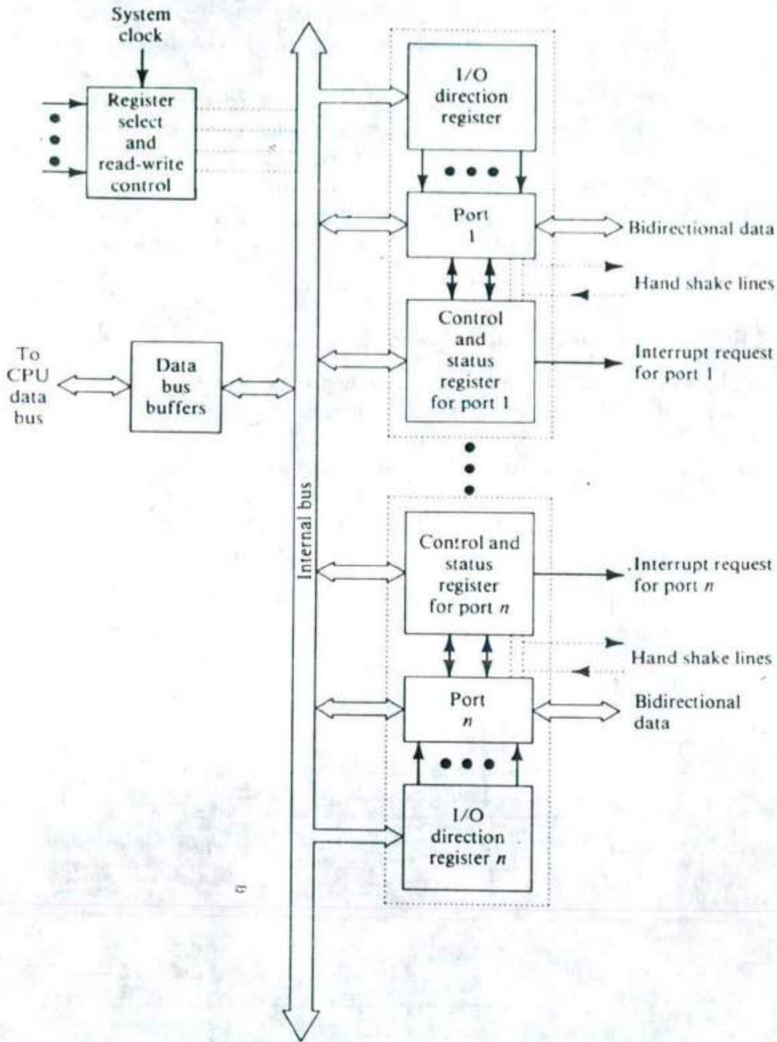



Figure 2.35 Simplified I/O interface for parallel data.

Figure 2.35 shows an example of a simplified parallel interface which contains *n* programmable data ports. Each data port is controlled by two associated control-status and I/O direction registers. By loading the appropriate command into the control-status register, the CPU can define the characteristics of the data port. Some definable characteristics are *I/O direction*, *I/O mode*, and *interrupt mode*. For the I/O direction, each bit, group of bits of the data port can be individually programmed as either input or output. For the I/O mode, each port can be programmed as *direct* I/O, *strobed* I/O, or *bidirectional* I/O. Moreover, each port can be programmed as interrupting or noninterrupting. In direct I/O, the device acts as a passive unit. Strobed and bidirectional I/O modes are used with active devices which must have established I/O communication protocols. A procedure, called *handshaking*, of mutual communication and cooperation between the CPU and a device is established so that the CPU knows when the input data is ready or when the output port is vacant.

In a handshake interface, the circuitry that receives data must first indicate its willingness to do so with a "ready" signal. For the example in Figure 2.36, this willingness is indicated by the INPUT LATCH EMPTY signal. Hence, if the input latch is empty, the sender can strobe in the data into the interface and indicate the presence of the data to the receiver (the CPU in this case) by generating an interrupt request. Note that the interrupt request is generated only if the CPU is willing to be interrupted (by the indication of the interrupt-enable signal from the CPU).

There are basically three classes of interrupts in a computer system: *internal*, *external*, and *software* interrupts. Internal interrupts, often called *traps*, are generated

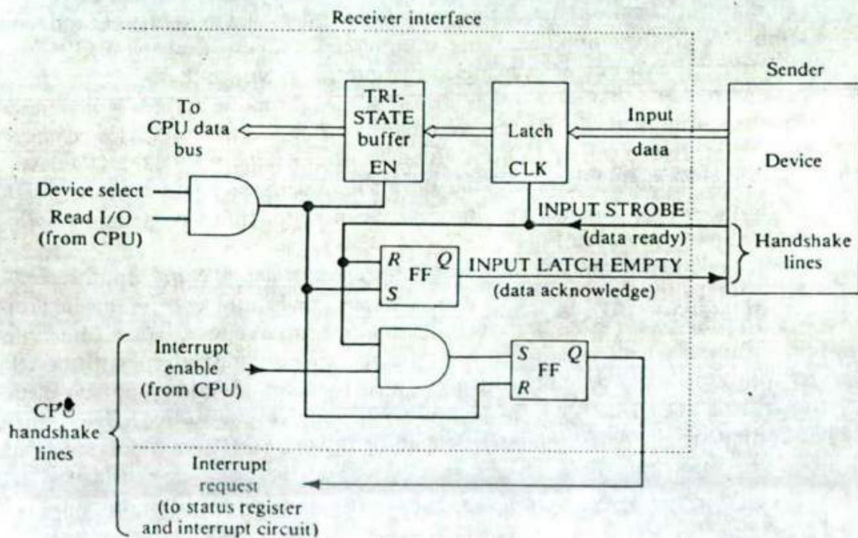



Figure 2.36 Input interface with handshaking.

within the CPU as a result of certain internal processor events. Traps may occur because of arithmetic-exception conditions, such as overflow and underflow (divide by zero) operations. It may also occur as a result of program faults, such as page faults, protection violation or the execution of an illegal instruction. Hardware faults, such as memory-parity errors and power failures, can also generate a trap. On the occurrence of a trap, the processor saves the state of the current process and transfers control to a *trap vector* location in memory, where the trap event is handled. Different trap vectors are often provided for different conditions or set of conditions.

Software interrupts, or system calls, affect the processor state in much the same way as a hardware interrupt. An example of a software interrupt occurs in the execution of the SVC system call instruction provided in IBM System 370. System call instructions are often used as a convenient and efficient method of calling operating system utilities.

External interrupts can be further classified as *maskable interrupts* (MI) and *nonmaskable interrupts* (NMI). Nonmaskable interrupts are considered the highest priority interrupts because they cannot be ignored, even if the CPU interrupt system is not enabled. NMI is particularly useful in monitoring a watchdog timer. It is also used in handling power failures. Maskable interrupts are accomplished through the use of an interrupt-enable flag associated with each device or set of devices. When this flag is set by the CPU, the flag permits the interrupt issued by the corresponding device to be received by the CPU. Otherwise, the device interrupt request is masked and does not reach the CPU until the interrupt-enable flag is set. The interrupt-enable flag is often incorporated in the device interface, as shown in the example of Figure 2.36.

In many applications, more than one device operating in interrupt mode may be connected to the computer. When an interrupt request reaches the CPU, it is known that at least one device caused the interrupt. Notice that since interrupts are asynchronous, there is a possibility that two or more devices will generate interrupts simultaneously. For the moment, assume that only one device caused the interrupt. In the simple I/O bus configuration of Figure 2.31, the I/O devices are identified by their addresses. The address lines can also be used to identify the interrupting device. The interrupting device can be identified by a simple *polling* arrangement, as shown in Figure 2.37.

In this scheme, called the *polled interrupt method*, the interrupt received by the CPU causes it to transfer control to a specified location, where the interrupt service routine is stored. The interrupt service routine consists of an interrupt polling routine which polls the devices in order to establish the identity of the interrupting device. The polling is performed by testing the interrupt bit of the status register of each device controller. When the interrupting device is determined, a call is made to the particular device handling procedure. If more than one device caused the interrupt, these devices are serviced in the order established by the polling direction. Hence all the devices that caused interrupt within the unpolled subcycle are serviced.

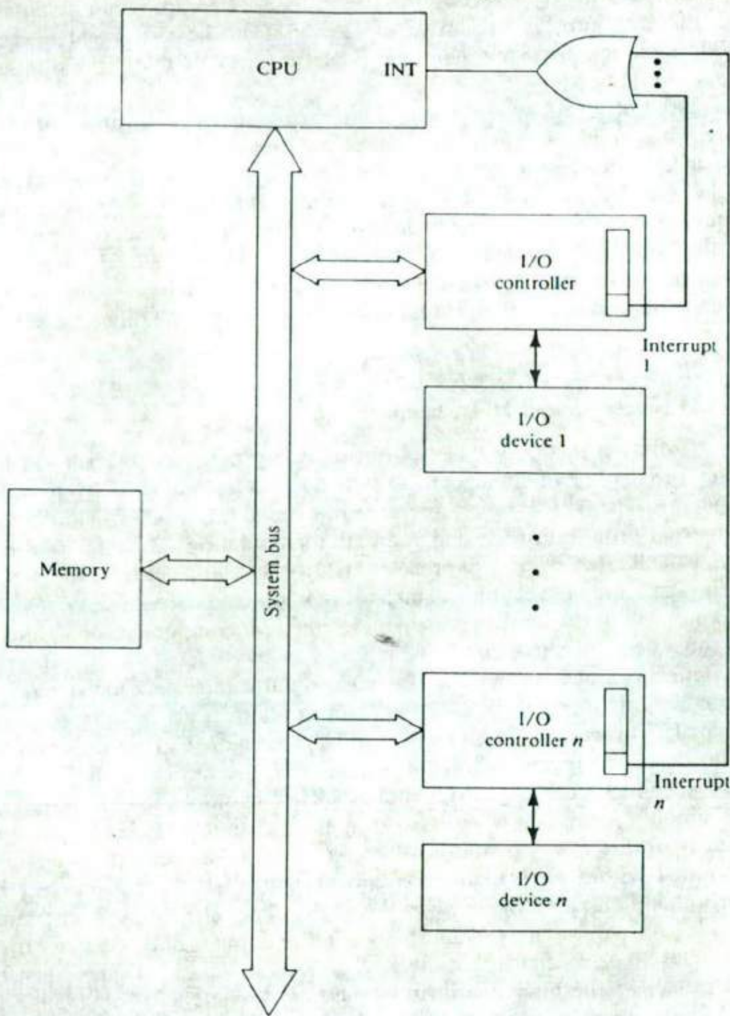



**Figure 2.37 Polled interrupt method.**

This method is effective for slow- to moderate-speed devices. However, the order of polling may have inherently established some form of fixed priority levels. The various methods used for establishing bus-control priority will be discussed in Section 7.2: that is, rotating daisy chaining, fixed-time slice, dynamic priority, and independently using a built-in hardware that automatically selects the highest priority device from the set of interrupting devices and also supplies the unique

starting address or *interrupt vector* of the device. This interrupt vector permits the CPU to transfer control to the device service routine at the corresponding vector location in main memory. A system that possesses this capability is said to have *vectored interrupts*.

A vectored-interrupt system requires a priority scheme to be provided in the hardware. This priority scheme could be fixed, rotating, or dynamic priority. When the CPU accepts an interrupt request, it sends an acknowledgement to the vectored-interrupt controller. The controller, upon receipt of this acknowledgement, sends the unique interrupt vector of the highest priority device of the set of unmasked-interrupting devices. This action is illustrated in Figure 2.38. The interrupt-acknowledge signal can in turn be transmitted to the highest priority device controller, which caused the interrupt in order to reset the interrupt request from that device.

### 2.5.3 I/O Processors and I/O Channels

The logical solution to the problem of obtaining maximum concurrency in I/O processing is to deploy an intelligent I/O system which isolates the CPU from the I/O peripherals. The CPU is therefore free to proceed at full speed with its primary task of internal program processing and data manipulation. The intelligent I/O subsystem is facilitated by an I/O processor (IOP). Basically, an I/O processor is one which is capable of executing a small set of commands to service the I/O request. Figure 2.39 illustrates the principal architectural components of an intelligent I/O subsystem. As shown in the figure, the I/O processor is attached directly to the system bus and is responsible for selecting and retrieving individual I/O commands from main memory. IOP generally contains a processor specifically designed for I/O processing and a number of I/O channels. The channels provide a communication path from the I/O processor to the device controllers and devices. I/O channels can also exist without the IOP, as shown in the figure.

In its simplest form, and when it exists alone, a channel may be a small processor that performs DMA operations for a small set of devices. If the channel is incorporated within an IOP, it is essentially a passive component with no logical processing capacity of its own. When the channel possesses processing capability, it is often used as an IOP. Notice that a number of devices and their controllers can be connected to an active channel. Hence, the channel must be capable of selecting the highest priority requesting device and also servicing it. The stand-alone channels in the I/O subsystem are used in various mainframes such as the IBM 370. IOPs are used in such systems as the CDC 6600 and the 8- and 16-bit Intel microcomputers.

**Channel architecture** There are basically two types of channels: *selector* and *multiplexor*, as used in the IBM 370 systems. A selector channel is an IOP designed to handle one I/O transaction at a time. Once the device is selected, the set of I/O operations for a given transaction runs to completion before the next transaction is initiated. The selector channel is thus normally used to control high-speed I/O

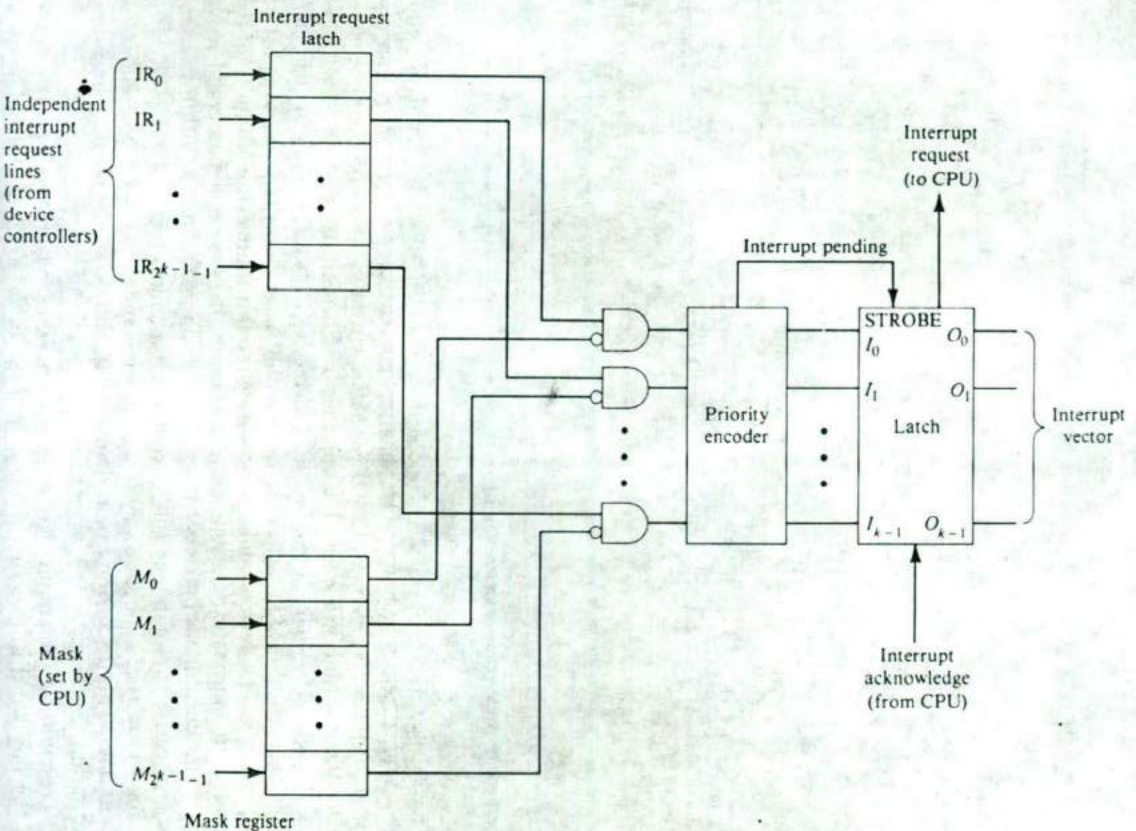



**Figure 2.38 Vectored interrupt system.**

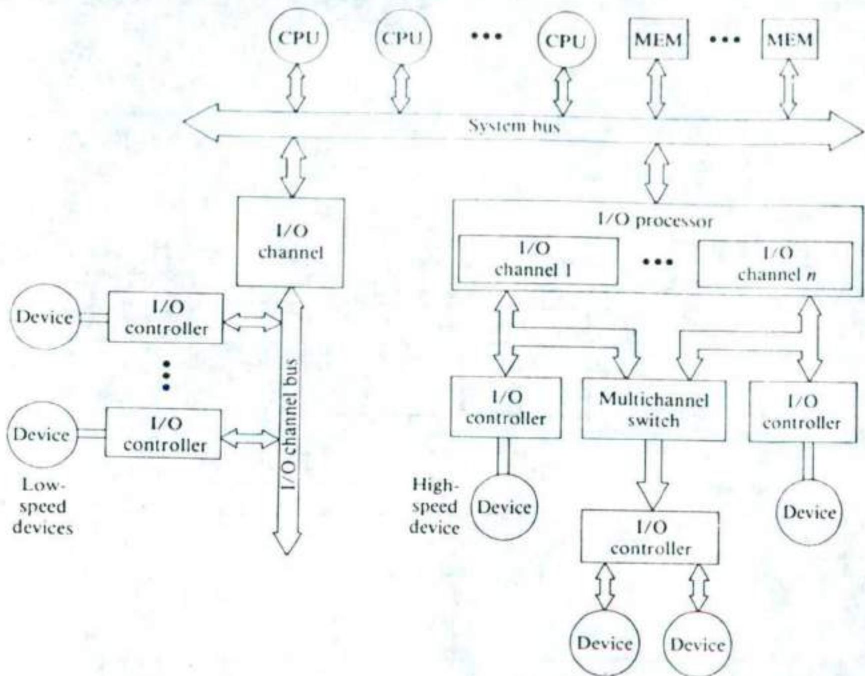
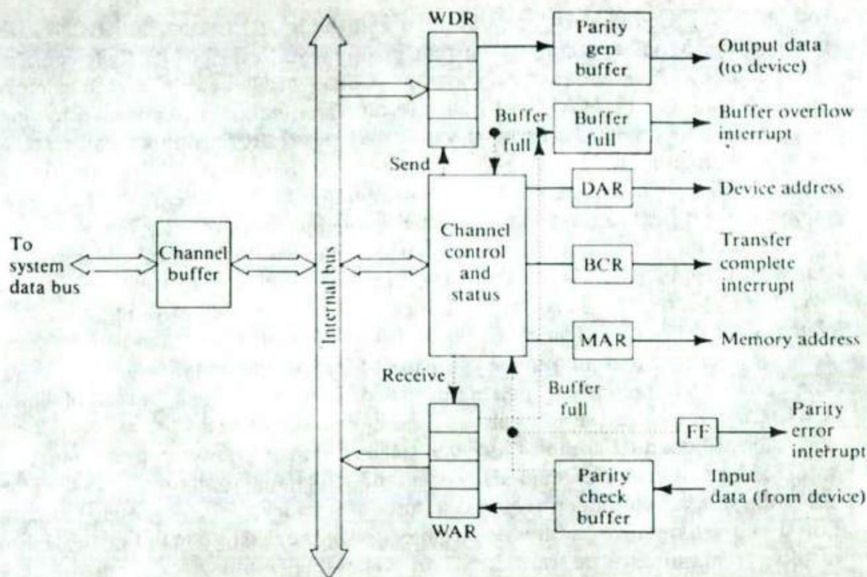



**Figure 2.39 Architecture of an intelligent I/O subsystem.**

devices such as fixed-head disks and drums. Figure 2.40 shows the architecture of a typical selector channel. The channel consists of word assembly and disassembly registers (WAR and WDR), which store the current word being received from the external device and the current word being transferred to the device, respectively. The channel can be made to receive or transmit data in character, halfword, or fullword mode. Thus, the assembly-disassembly registers can operate accordingly. The devices could be operating too fast (in the case of input) for the channel to handle the data reliably. For example, the next character could arrive before the current one in the WAR has been transmitted to the CPU. If this occurs, an overrun error or buffer full interrupt is generated to the CPU, which can request retransmission. One way to alleviate this problem is to double-buffer the input data. This discussion can also apply to the WDR when operating on a slow output device.

The initialization of the selector channel requires the definition of the location of the first word in memory, the length of the block to be transferred, and the device address. The initialization program is stored in memory and can be executed by the channel in order to initialize its internal registers. The registers used in this case are the device address register (DAR), the block count register (BCR),




BCR: block count register
DAR: device address register
MAR: memory address register
WAR: word assembly register
WDR: word disassembly register

**Figure 2.40 Selector channel architecture.**

and the memory address register (MAR). In order to perform an I/O transaction, the CPU transmits a START signal and the device address to the channel on which the selected device is attached. The channel then fetches the *channel address word* (CAW) from a prespecified location in memory. This word, which was stored prior to initiation of the I/O transaction, contains the starting address of the I/O program (called *channel program*) to be executed by the channel.

The channel program consists of *channel command words* (CCW) or control words or instructions. In most sophisticated channels, the channel programs may include commands for positioning the read-write heads of disk drives, rewinding tapes, and selecting or testing the status of a device. In addition, the set of CCWs may contain instructions which permit looping and branching. The concept of the single-channel program can be extended to the CPU preparing an arbitrary number of I/O transactions to be executed by the I/O subsystem as a sequence of I/O transactions. This feature is known as *command chaining*.

If the addressed device is available, the channel executes the channel program to perform the I/O transaction; otherwise, the request may be queued or the CPU notified of the unavailability of the device. If the channel program is executed,

the DAR, BCR, and MAR are initialized and the block transfer initiated. Subsequently, the MAR contains the current memory address and the BCR contains the remaining block length. After the transfer of a word between the main memory and the channel, the MAR and BCR are incremented and decremented by one, respectively, to reflect the updated values. When the BCR counts down to zero, a "transfer-complete" interrupt is generated and sent to the CPU. In case of errors (parity or lost character), an error interrupt is also generated. The typical maximum data rate of a selector channel is on the order of 1 to 3 megabytes/s.

A multiplexor channel is an IOP which can control several different I/O transactions concurrently. In this case, the data transfers are time-multiplexed over the I/O interface. This type of channel can be further divided into *block* and *character multiplexors.* The character multiplexors are used to handle low-speed devices, whereas block multiplexors are used for medium- and high-speed devices. The block or character multiplexor consists of a set of *subchannels*, each of which can act as a low-speed selector channel, as shown in Figure 2.41.

Each subchannel contains a buffer, device address register, request flag, and some control and status flags. However, the subchannels share global channel control. Each subchannel is required to have a memory address register (to maintain the current memory address) and a block count register (to maintain the length of block remaining to be transferred). In a character multiplexor channel with a large number of subchannels, as in the IBM 370 system with 256 subchannels, it is cost prohibitive to maintain these pairs of registers in the subchannels. Hence, these registers are maintained in main memory and are accessed by the channel control, as shown in Figure 2.41. The channel controller can select a subchannel for a burst mode or multiplex mode. In the multiplex mode, the scan control cyclically polls the request flag of each subchannel. If the flag is set, the subchannel is selected for a character or block transfer. The subchannel mode control is checked to determine the direction of the transfer operation. When the character or block is transferred, the next subchannel is polled. The block multiplexor interleaves by blocks instead of characters as in a character multiplexor.

For example, suppose that three successive I/O transactions $X$, $Y$, and $Z$ are requested. Assume that each transaction is required to transfer a string of $n$ characters. $X$, $Y$, and $Z$ are sequences of characters $X_0, X_1, \ldots, X_{n-1}$, $Y_0$, $Y_1, \ldots, Y_{n-1}$, and $Z_0, Z_1, \ldots, Z_{n-1}$, respectively. If these transactions are initiated on a selector channel, then the selector channel transmission appears as $X_0 X_1 \ldots X_{n-1} Y_0 Y_1 \ldots Y_{n-1} Z_0 Z_1 \ldots Z_{n-1}$. On a character multiplexor with at least three subchannels, they may appear as $X_0 Y_0 Z_0 X_1 Y_1 Z_1 \ldots X_{n-1} Y_{n-1} Z_{n-1}$. On a block multiplexor programmed for $k$ characters per blocks (assuming that $k < n$), the sequence may appear as $X_0 X_1 \ldots X_{k-1} Y_0 Y_1 \ldots Y_{k-1} Z_0 Z_1 \ldots Z_{k-1}$ $X_k X_{k-1} \ldots X_{2k-1} Y_k Y_{k+1} \ldots Y_{2k-1} Z_k Z_{k+1} \ldots, Z_{2k-1} \ldots$ and so on. The frequent switching and the associated overhead degrade the performance of the character multiplexor. The maximum data rate for the character multiplexor is typically on the order of 100K to 200K bytes p/s. The maximum data rate of the block multiplex or channel approaches that of the selector channel as the block size $k$ approaches the string length $n$.

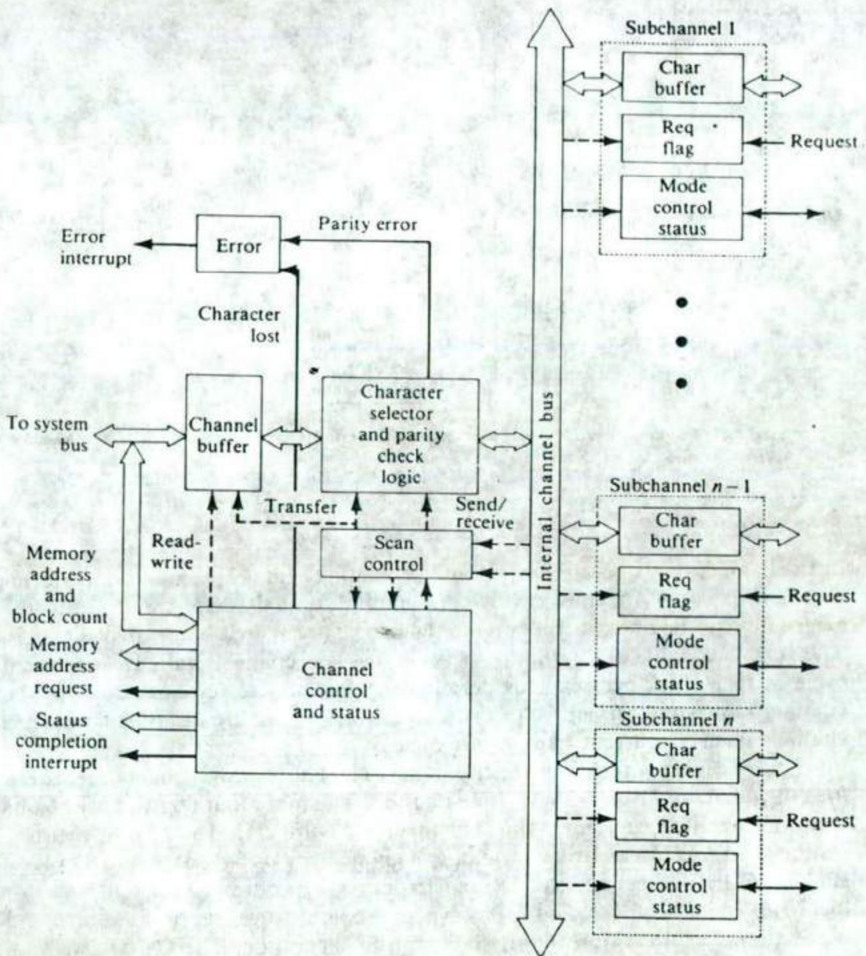



Figure 2.41 Architecture of character-multiplexor channel.

With current technology, an I/O processor can be implemented on a single chip for microcomputer and minicomputer systems. An example of an IOP on a chip is the INTEL 8089 integrated IOP, which is capable of being interfaced to 8-bit and 16-bit systems. This IOP contains two independent I/O channels and a processor on the same chip, as shown in Figure 2.42. It also contains a bus interface, an assembly-disassembly register file and an instruction fetch unit. In order to enable autonomous operation of the I/O channels, each channel maintains its own register set, control and status registers, and a flexible channel controller.

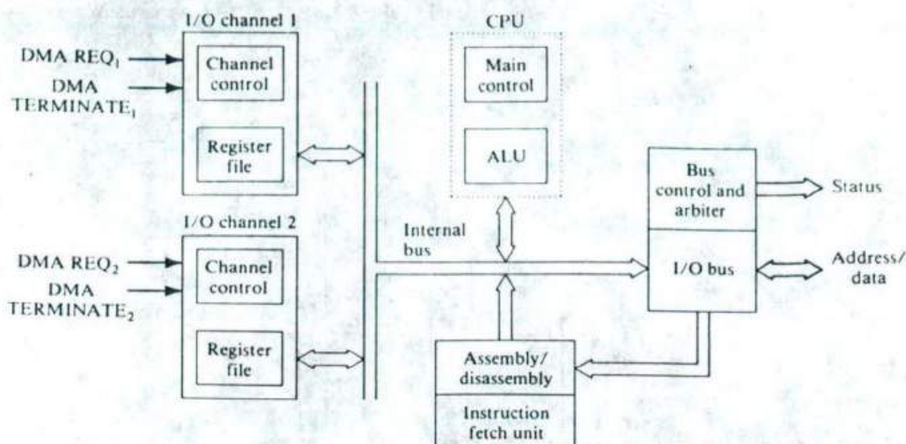
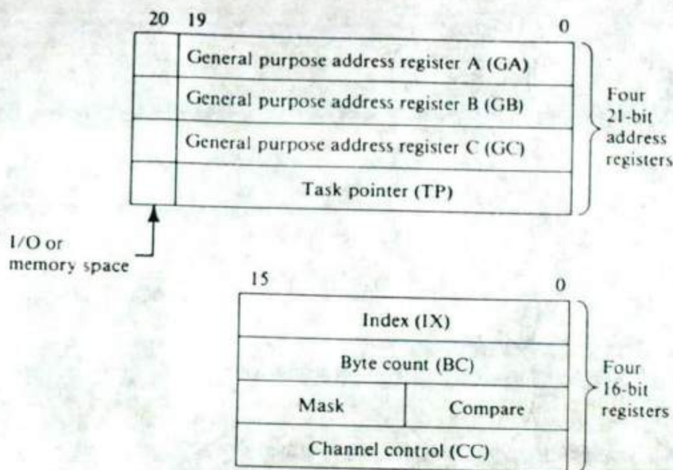



**Figure 2.42 The Intel-8089 I/O processor with two separate I/O channels (El-Ayat, *IEEE Computer*, 1979).**

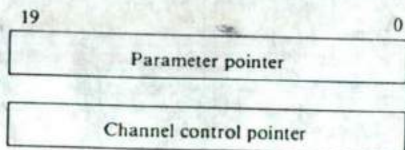
Both channels may operate concurrently, executing channel programs or performing high-speed DMA transfers by time multiplexing the access and use of the external bus. The bus control and interface logic are shared by the two channels. The IOP is capable of alternating between the two channels with every internal cycle (4 to 8 clock cycles). This permits very fast service response times to the channel requesting service. A priority algorithm is used by the IOP to select a channel when concurrent requests are made.

Each register set of a channel contains eight user-programmable registers, four of which are 21-bit-wide address registers. The other four registers are 16 bits wide. This is illustrated for a single channel in Figure 2.43. The address registers can be used to address 1 megabyte of system memory or 64K bytes of I/O space. Bit 20 of the address register is used to select the address space as system or local I/O space. The GA and GB registers are used to reference the source and destination locations during any data transfer operation. The GC register can also be used as a general register pointer by the channel program. The task pointer (TP) serves as the channel program counter, which is initialized whenever the channel is started. Using the TP, the instruction unit in the IOP can fetch the next CCW. The TP can also be manipulated by the channel program.

The byte register (BC) contains the number of bytes to be transferred during DMA operation. BC can also be set up to terminate the DMA transfer if this mode is selected. The index register (IX) is used as an index in the indexed addressing mode. The mask-compare register is used to perform masked-byte comparisons during channel program execution and DMA operations. During program execution, the comparisons are used for conditional branching, and in the DMA mode,




(*a*) User programmable registers

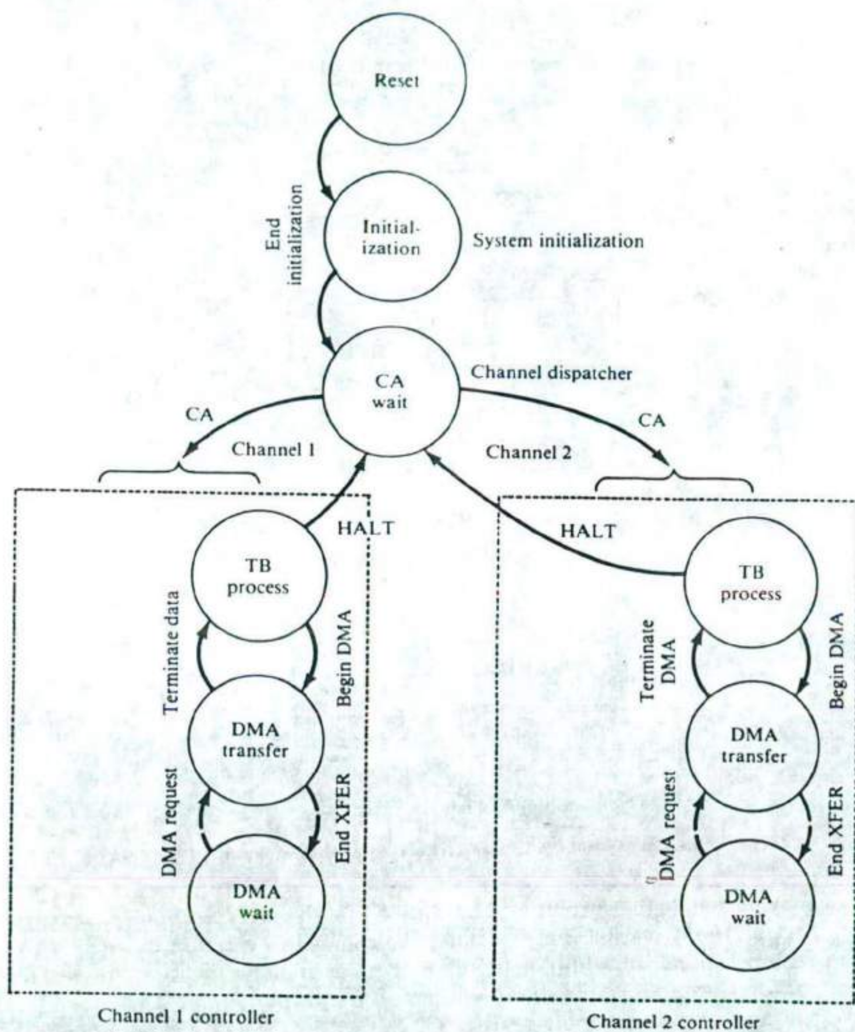



(*b*) Nonuser programmable registers

**Figure 2.43 Register set of a channel in Intel IOP (Courtesy of *IEEE Computer*, 1979, El-Ayat).**

they may terminate the current DMA transfer. The channel control register (CC) is a special 16-bit register which defines the channel's operation during DMA transfer operations. In addition to the user-programmable registers, there are two non-user programmable 20-bit registers, also shown in Figure 2.43.

The assembly-disassembly register file is used in the DMA transfer mode. For example, when data is transferred during a DMA operation from an 8-bit bus to a 16-bit bus, the IOP assembles 2 bytes in its assembly register file before transferring a word to the destination. A simplified computational model of the INTEL IOP is given in Figure 2.44. After reset, a channel attention (CA) input pulse forces an internal initialization sequence. Then the processor is ready to dispatch an I/O transaction request to either of the two channels to perform the desired I/O task. The I/O channel normally begins its operation in the task block (TB) state with the execution of the I/O program and enters the DMA state under IOP program control. In this state, the channel proceeds with high-speed data




CA: channel attention

**Figure 2.44 Simplified computational model of the Intel I/O processor (Courtesy of *IEEE Computer*, 1979, El-Ayat).**

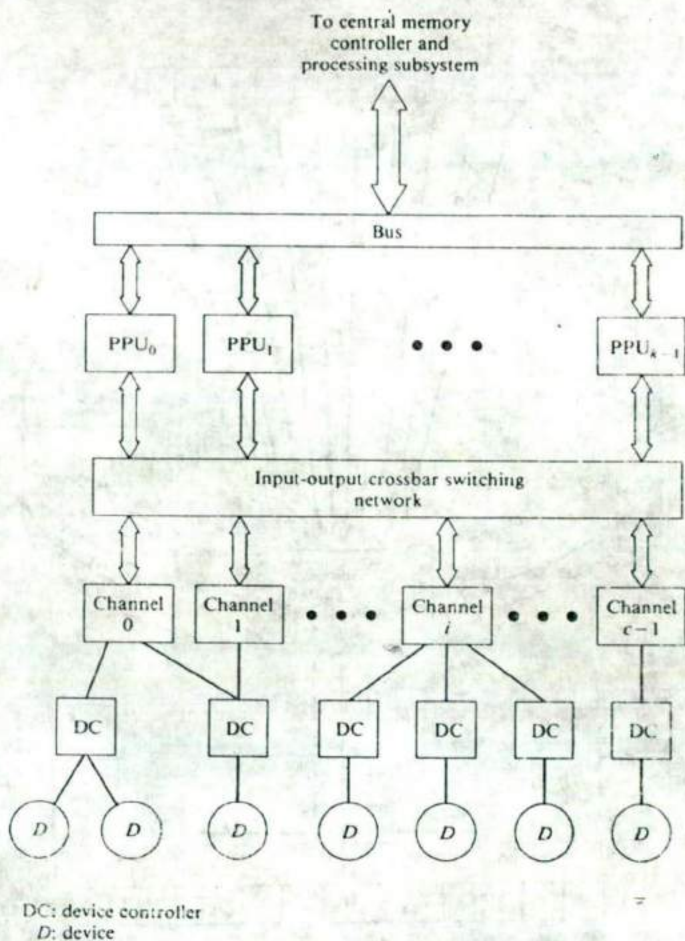



**Figure 2.45 Logical representation of peripheral processing subsystem for CDC-6600 and Cyber 170 (Courtesy of Control Data Corp.).**

transfers in either burst or request-synchronized mode until the occurrence of a valid termination condition, which returns the channel to the TB state. HALT commands force the channel into the idle state until further dispatching occurs.

Another example of an integrated IOP is in the CDC 6600 I/O subsystem. The integrated IOP is also used as the peripheral processing subsystem (PPS) in the Cyber 170 multiprocessor system. It consists of a set of 10 peripheral processing units (PPU) which share a set of channels to which devices and their controllers are connected. A logical representation of such an I/O subsystem is shown in Figure 2.45.

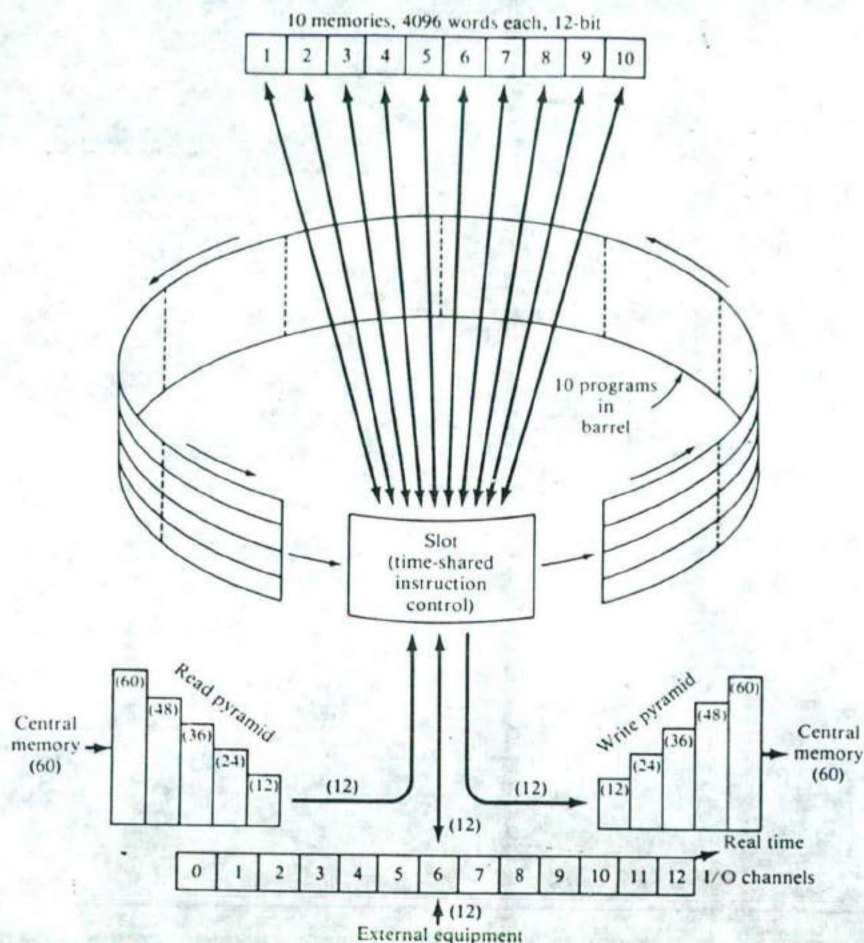



**Figure 2.46 Barrel processing of I/O transactions in CDC integrated peripheral processing units (Courtesy of Control Data Corp.).**

The CDC 6600 integrated peripheral processor uses a so-called barrel design to share logical units within the IOP. It uses a set of registers to share a common arithmetic logic unit and a data distribution system in a synchronous fashion. The barrel contains 10 peripheral processing units (PPUs) and a PPU is 12-bits wide. A PPU instruction requires a number of steps for its execution. The execution in each step is performed in a distinct "slot" which logically represents a PPU. Hence, the PPU instruction is executed as in a cyclic pipeline process, as shown in Figure 2.46. This execution sequence is possible because each instruction

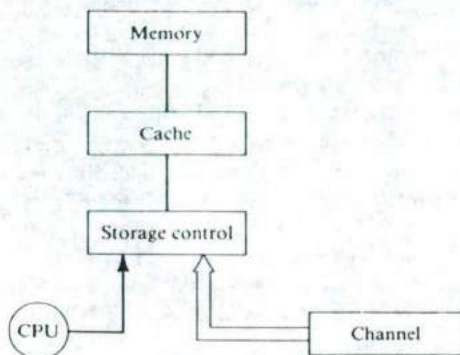
cycle is an integral number (up to 10) of minor cycles. A minor cycle is 100 ns and a major cycle is 1000 ns; hence, the choice of 10 PPUs.

In each minor cycle, all information in the barrel is moved one position (synchronously) after each step is executed in its current slot. The information in each PPU is moved through the shared slot position once every major cycle. Since each PPU operates once per major cycle, the maximum data rate is 12 bits $\div$ 1000 ns $=$ $12 \times 10^6$ bits/s. Therefore, the 10 PPUs are time-shared by the slot hardware without significant degradation in performance. However, since the CDC 6600 is a 60-bit computer, five PPU transfers are required to form a 60-bit word. Also, since the I/O processing is synchronized in the CDC system, no handshaking is necessary as in the IBM channels.

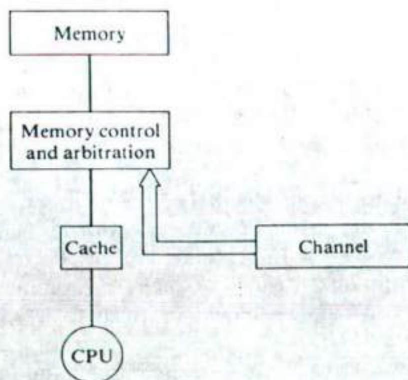
**I/O configuration in systems with cache** There are two basic methods of connecting an I/O subsystem to the processor-memory complex in a system with a cache. In the first configuration, the I/O channel can be attached to the cache so that the cache is shared by the processor and channel, as shown in Figure 2.47*a*. The channel competes with the processor for access to the cache. An I/O channel is often slower than the processor. Thus connecting the channel to the cache does not significantly improve the performance of I/O transfers. I/O transfers have little locality and they increase the traffic between the cache and memory. This increase is caused by three main effects: main memory update of memory-bound I/O data; misses caused by channel fetches from memory; and channel programs (and I/O data) occupying cache, reducing the effective cache aggregate miss ratios seen by processor-bound jobs. The configuration of Figure 2.47*a* may also encounter cache *data-overrun*, in which the data transfer occurs at a rate higher than the cache controller can sustain.

An alternate configuration is to connect the channel to the memory directly, as shown in Figure 2.47*b*. In this case, the channel competes with the cache controller for access to the memory. However, the I/O channel and processor executions conflict at miss times only, assuming a write-back memory update policy. Also, the cache is not encumbered with the data blocks destined to I/O. It has, however, one major drawback: data consistency or coherence problems. To illustrate, consider a cache which uses write-back main memory update policy. Assume that the processor has modified a copy of a data element X in the cache so that the value of the copy in the cache is NEWX and the memory has not been updated.

Let OLDX be the value of X in memory. Before the memory is updated, the I/O channel requests a fetch from location X in the memory, which delivers OLDX instead of NEWX. A coherence problem has occurred. One solution is to keep a dynamic table in the memory controller which, at any time, indicates the set of blocks in the cache and their status (whether modified or unmodified). Let the modified status be denoted by RW. When the I/O channel makes a reference to a memory block which is also in the cache, the status is checked by the memory controller. If it is RW and the channel requests a read, the data is fetched from the cache. However, if the channel requests a write to the block, the corresponding




(*a*) The I/O processor accesses the cache




(*b*) The I/O processor accesses the memory

**Figure 2.47 Two I/O configurations for a uniprocessor with cache.**

cache block frame is invalidated before the memory block is modified by the channel. A similar description can be given for a processor reference.

Note that for a system with buffered write-through update policy, the coherence problem is automatically corrected in the second configuration if the write queue is maintained within the memory controller. However, this configuration may also encounter the data-overrun problem. More cache coherence studies will be given in Chapter 8.

## 2.6 BIBLIOGRAPHIC NOTES AND PROBLEMS

A model of memory hierarchies in which the memory management strategy is characterized by the hit ratio was given by Chow (1974). Storage systems are covered in detail by Matick (1977, 1980). The Vax 11/780 paging system was discussed in DEC (1979) and Levy and Eckhouse (1980). Cache memories for multiprocessors was studied by Dubois and Briggs (1982a). A discussion of virtual memory and the concept of locality of reference are given in Denning (1970), Denning and Graham (1975) and Baer (1980). A treatment of paged, segmented memory systems and systems with paged segments can be found in Watson (1970). These concepts and their implementations were also described thoroughly in Bensoussan et al. (1972).

Characteristics of cache memories and their organizations have been studied by various authors as in Conti (1969), Mead (1970), Bell et al. (1974). An introduction of the characteristics of cache memories is given in Kaplan and Winder (1974). Recently, Smith (1982) presented a comprehensive survey paper on cache memories. The LRU hardware diagram and its description were given in Yeh (1981). The relationship between the miss ratios for set-associative and fully associative cache was derived in Smith (1978) assuming a linear paging model of program behavior which was studied in Saltzer (1974).

The effect of sharing in the resident set of pages was developed by using the results on union of events in Feller (1970). The demonstration of the flexibility and efficiency of the variable partitioning strategy was given in Coffman and Ryan (1972). Details of some fixed allocation strategies and stack algorithms are given in Coffman and Denning (1973). The two-parameter fit for the lifetime function was proposed in Chamberlin, Fuller, and Lin (1973). The variable-partitioning strategies are presented in Denning and Graham (1975). The working-set model was presented in Denning (1968) and its properties in Denning and Schwartz (1972). The reader is encouraged to read Denning (1980) and Baer (1980) for a complete study of memory management policies.

A general overview of I/O architecture was presented in Buzen (1975). There are good treatments of I/O subsystems and their organizations in Baer (1980), Hayes (1978), and Kuck (1979). Details of I/O subsystems in IBM System/370 and the CDC 6600 integrated peripheral processing subsystem can be found in IBM (1974) and Thorton (1970), respectively. Another overview on I/O channel architectures can be found in Lane (1980). A description of the architecture of Intel IOP is given in El-Ayat (1979). The reader is encouraged to read this reference for typical applications and programming example of the I/O processor.

### Problems

**2.1** Consider a two-level memory hierarchy $(M_1, M_2)$ for a computer system, as depicted in the following diagram. Let $C_1$ and $C_2$ be the costs per bit, $S_1$ and $S_2$ be the storage capacities, and $t_1$ and $t_2$ be the access times of the memories $M_1$ and $M_2$, respectively. The hit ratio $H$ is defined as the

probability that a logical address generated by the CPU refers to information stored in $M_1$. Answer the following questions associated with this virtual memory system.

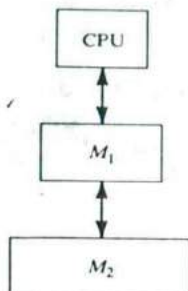
(*a*) What is the average cost $C$ per bit of the entire memory hierarchy?

(*b*) Under what condition will the average cost per bit $C$ approach $C_2$?

(*c*) What is the average access time $t_a$ for the CPU to access a word from the memory system?

(*d*) Let $r = t_2/t_1$ be the speed ratio of the two memories. Let $E = t_1/t_2$ be the access efficiency of the virtual memory system. Express $E$ in terms of $r$ and $H$. Also plot $E$ against $H$ for $r = 1, 2, 10$, and 100 respectively on a grid-graph paper.

(*e*) Suppose that $r = 100$, what is the required minimum value of the hit ratio to make $E > 0.90$?




**2.2** A page trace is a sequence of page numbers $P = r_1, r_2, r_3, \ldots, r_{k-1}, r_k, r_{k+1}, \ldots$, where $r_k$ is the page number of the $k$th address in a sequence of addresses

| Page trace | *a* | *a* | *b* | *b* | *c* | *c* | *a* | *b* |
|---|---|---|---|---|---|---|---|---|
| Page faults | • | | • | | • | | • | • |
| Mp contents | *a* | *a* | *a* | *a* | *c* | *c* | *c* | *b* |
| | – | – | *b* | *b* | *b* | *b* | *a* | *a* |

The fault rate $F$ is the number of page faults divided by the number of page addresses (length of the page trace). For this example, $F = \frac{5}{8} = 0.625$. The hit ratio $H$ is $1 - F$. For the remainder of this problem, let $P = abacabdbacd$.

(*a*) Produce a table similar to the above table for page trace $P$ under a FIFO replacement algorithm with memory size, $|Mp| = 2$ page frames. What is the hit ratio?

(*b*) Do the same for an LRU replacement algorithm.

(*c*) Repeat (*a*) and (*b*) for $|Mp| = 3$ page frames.

(*d*) Intuitively, both FIFO and LRU would seem to be "good" algorithms. A *most recently used* (MRU) algorithm intuitively sounds like a bad algorithm. Repeat (*a*) with an MRU replacement algorithm. Compare this with the results obtained in (*a*) through (*c*). What does this say about the particular page trace $P$ and about the generality of results obtained by comparing replacement algorithms based on a single page trace?

**2.3** In a uniprocessor with cache, the processor issues its memory access requests to the cache controller (CC). In the case of a miss or a write-through, the CC interacts with the memory controller (MC). Draw the flowcharts describing the operations of a CC for a read and a write operation. Consider the write-back-write-allocate with flagged swap and the write-through-write-allocate strategies. Assume that no read-through is implemented. Indicate how to modify the flowcharts for (*a*) a write-back-write-allocate with simple WB and with flagged register WB and for (*b*) write-through without write-allocation.

**2.4** Consider the following search algorithm:

```
begin
ifound→N+1;i→0;
while (ifound≠i) do
  begin
    i←i+1;
    if (template=data [i])
       then ifound←i;
  end
end
```

In this program, *data*[*i*] is an array of $N(=2^n)$ floating-point numbers; *template* is a floating-point number; $N, i$, and *ifound* are integers. A floating-point number occupies two memory words, while an integer occupies one memory word only. Assume that the program code as well as the variables $N, i$, *ifound*, and *template* fit on the same memory page. *Data* [•] is stored in a set of consecutive pages, starting at the beginning of a page. A page is $P = 2^r$ words long. The memory is $M = 2^m$ words long ($M \ll N$). Assume that there is one and only one element equal to *template* in the array *data*. The algorithm is run on a uniprocessor with a paged virtual memory system. The replacement policy is LRU.

(*a*) If Probability[*ifound* = *i*] = $(1/N)$ $(1 \le i \le N)$, determine the mean number of page faults in the cases where the memory does not contain any of the process pages at the beginning of the process, and where the memory is preloaded to capacity with the program page and the first $2^{m-p} - 1$ data pages of the process.

(*b*) Repeat (*a*) if Probability[*ifound* = *i*] = $G(N)q(1-q)^{i-1}$ for $1 \le i \le N$, $0 < q < 1$, where $G(N) = 1/[1-(1-q)^N]$.

**2.5** A computer architect is considering the adoption of write-through-with-write-allocate (WTWA) or write-back (WB) cache management strategy. Assuming no read-through, each block consists of $b$ words, which can be transferred between main memory (MM) and cache in $b + c - 1$ time units, where $c$ is the MM cycle time. The cache is independent of the strategy and is given by $h$. The probability that a memory reference is a write is $w_t$ and the probability that the block being replaced in the cache was modified (in WB strategy) is $w_b$. Usually $w_b > w_t$.

(*a*) Using each strategy, give a formula for the expected time to process a reference in terms of the above variables.

(*b*) Assuming $w_t = 0.16$ and $w_b = 0.56$, what is the performance of the WB strategy in comparison to WTWA strategy when (1) $h \to 1$ and (2) $h \to 0$.

(*c*) Give a general expression describing when WTWA is better than WB as a function of $h$ and $b$. Assume that $w_t = 0.16$, $w_b = 0.56$, and $c = 10$.

(*d*) Does $w_t$ depend on $h$? Give intuitive reasons.

**2.6** A certain uniprocessor computer system has a paged segmentation virtual memory system and also a cache. The virtual address is a triple $(s, p, d)$ where $s$ is the segment number, $p$ is the page within $s$, and $d$ is the displacement within $p$. A translation lookaside buffer (TLB) is used to perform the address translation when the virtual address is in the TLB. If there is a miss in the TLB, the translation is performed by accessing the segment table and then a page table, either or both of which may be in the cache or in main memory (MM).

Address translation via the TLB requires one clock cycle. A fetch from the cache requires two clock cycles (one clock cycle to determine if the requested address is in the cache plus one clock cycle to read the data). A read from MM requires eight clock cycles. There is no overlap between TLB translation and cache access. Once the address translation is complete, the read of the desired data may be from either the cache or MM. This means that the fastest possible data access requires three clock cycles:

one for TLB address translation and two to read the data from the cache. There are nine other ways in which a read can proceed, all requiring more than three clock cycles.

(*a*) Assuming a TLB hit ratio of 0.9 and a cache hit ratio of $h$, enumerate all 10 possible read patterns, the time taken for each, and the probability of occurrence for each pattern. What is the average read time in the system? (Assume that when a word is fetched from memory, a read-through policy is used.)

(*b*) The above discussion assumes that the cache is always given a physical memory address. Suppose that the cache is presented with the virtual address of the data being requested rather than its physical address in memory. In this case, the TLB translation and cache search can be done concurrently. This means that whenever the requested data is in the cache, no address translation is necessary and only two clock cycles are required for the fetch. If the data is not in the cache, either a TLB translation segment table–page table access is needed to generate the physical address of the data. When data is written into the cache, it is tagged with its virtual address. Find the average read time for a system organized in this fashion. Assume that only one clock cycle is required to establish that an item is not in the cache.

(*c*) What are the disadvantages of a cache using virtual addresses?

**2.7** In the LRU stack model, assume that the stack distances are independently and identically drawn from a distribution $\{g(j)\}, j = 1, 2, \ldots, n$, for a stack of size $n_o$. Since each set in the cache constitutes a separate associative memory, it can be managed with LRU replacement. Show that the probability $p(i, S)$ of referencing the $i$th most recently referenced block in a set, given $S$ sets, is

$$p(i, S) = \sum_{j=i}^{\infty} g(j) \cdot \left(\frac{1}{S}\right)^{i-1} \cdot \left(\frac{S-1}{S}\right)^{j-i} \cdot \binom{j-1}{i-1}$$

**2.8** Consider three interleaved memory organizations for a main memory system containing 8 memory modules, $M_0, M_1, \ldots, M_7$. Each module has a capacity of 2K words. In total, the memory capacity is 16K words. The maximum memory bandwidth is 8 words/cycle. In each of the following organizations, first specify the memory address format (14 bits), then show the address assignment patterns in each memory module, and finally indicate the maximum bandwidth when one of the 8 modules fails to function. Comment on the relative merits of the three interleaved memory organizations.

(*a*) Eight-way interleaved memory organization (one group).

(*b*) Grouped four-way interleaved organization (two groups).

(*c*) Grouped two-way interleaved organization (four groups).