CHAPTER
NINE

EXAMPLE MULTIPROCESSOR SYSTEMS

This chapter studies a number of existing multiprocessor systems. We begin with an introduction of the entire multiprocessor space. Two large multiprocessor projects are reviewed first. These are the early C.mmp system and the S-1 system currently under development. Then we study a number of commercially available multiprocessors, including some models in the IBM 370 series and the 3080 series, the Univac 1100 series, the Tandem Nonstop system, the HEP, and the Cray X-MP. A comprehensive literature guide on multiprocessors is given at the end of the chapter.

9.1 THE SPACE OF MULTIPROCESSOR SYSTEMS

Multiprocessor systems can be divided into two classes: the exploratory research computers and commercial multiprocessors. We consider a system to be exploratory if it is developed mainly for research purposes or for dedicated missions. Commercial multiprocessors are those systems that are available in the computer market. A summary of existing multiprocessors is given below. We leave the details of each system to subsequent sections. Some of the systems were studied in the previous two chapters.

9.1.1 Exploratory Systems

Three exploratory multiprocessors are covered in this book. The C.mmp and Cm* are both research multiprocessors developed at the Carnegie Mellon University. The C.mmp was developed in the early seventies. It consists of 16 PDP-11 minicomputers sharing a common memory via a 16 × 16 crossbar switch. We shall study the C.mmp architecture and its specially developed Hydra operating system in Section 9.2. The hierarchically structured Cm* has already been described in Chapter 7. The Cm* is still being used at CMU as a research vehicle. The C.mmp is no longer in operation now.

Another crossbar-structured multiprocessor is the S-1 system currently under development at the Lawrence Livermore National Laboratory. It is a 16-processor system. However, each uniprocessor in the S-1 is custom designed for a multiprocessing environment. The S-1, once completed, should be a gigaflops machine. We shall study its processor characteristics and software development in Section 9.3.

9.1.2 Commercial Multiprocessors

In Chapter 7, we have already studied several commercial multiprocessors, including the CDC Cyber-170, the Honeywell 60/66, and the DEC System 10. In Sections 9.4 and 9.5, we shall examine more large-scale multiprocessor systems, including the IBM System 370/168 and the IBM 3081 multiprocessors. In particular, we will focus on the details of the 370/168 MP and the 3081; both are dual-processor systems. In the Univac 1100 series, we shall study two multiprocessor systems: the model 1100/8x and the model 1100/9x systems, each of which has up to four processors.

Cray Research recently announced its multiprocessor model the Cray X-MP. This is a dual-processor system highly pipelined for both scalar and vector processing at high speed. Denelcor, Inc. developed the HEP computer, which can be configured to have up to 16 processors sharing multiple memories and peripherals via a packet switching network. We shall study HEP in Section 9.4. Most of the existing commercial multiprocessors operate essentially with multiple SISD operations at the highest program level; that is, single program multiprocessing and loosely coupled multiprogramming. The HEP is an *intrinsic* multiprocessor with real MIMD pipelined operations at the process level. To achieve fault-tolerant multiprocessing, we shall study the Tandem Nonstop multiple processor system. Multitasking techniques developed with the X-MP will be treated in Section 9.6.

In Table 9.1, we summarize the major architectural characteristics of those multiprocessor computers covered in this book. Speed improvement is only one of the concerns in using a multiprocessor computer. Enhanced reliability and availability and the promoting of resource sharing to achieve a high performance/cost ratio are also important factors in developing multiprocessors. The general trend for commercial machines is that high performance is achieved with multiple highly pipelined processors. The goal is to solve large user problems by multiple processors in a cooperative and effective manner. Most existing commercial systems have two to four processors. Only a few systems offer 16 processors like the Tandem and the HEP. Some research multiprocessors have more than 16 processors, like the Cm*, consisting of 50 LSI-11 processors in the system.

Table 9.1 Multiprocessor computer systems

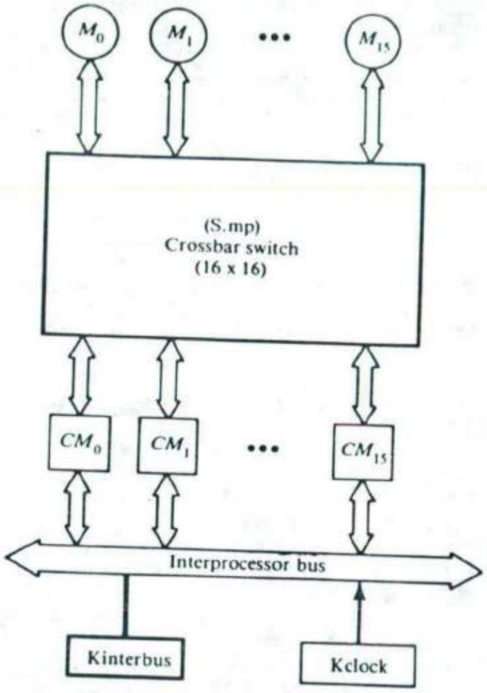
Systems	Architectural features	Remarks
C.mmp	16 PDP-11s, Crossbar, Hydra OS	Section 9.2 (dismantled)
Cm*	50 LSI-11s, Hierarchical processor clusters, Medusa and Star OS	Section 7.1 (exploratory)
S-1	16 Mark IIA processors, Crossbar, Amber OS	Section 9.3 (under development)
IBM 370/168MP	Dual-processor systems, shared memory and separate I/O, Multiple Virtual Storage (MVS) OS	Sections 9.5.1 and 9.5.2
Univac 1100/8x, 1100/9x	Two- or four-processor systems, EXEC OS, shared memory and I/O	Section 9.5.3
Tandem /16	Up to 16 processors, dual-common buses, dual-ported controllers, Nonstop OS	Section 9.5.4
Cray X-MP	Dual processors with shared common memory and dedicated pipelines, COS	Section 9.6
Denelcor HEP	Up to 16 processors, Packet-switched network, MIMD pipelining	Section 9.4
IBM 3081, 3084	Two- or four-processor systems, shared memory and I/O devices, MVS and VM OS	Section 9.5.1
Cyber 170	Two processors with a central memory controller	Section 7.1.1
Honeywell 60/66	A three-processor system with triple redundancy	Section 7.1.1
PDP-10	Two-processor system with a master-slave or a symmetric configuration	Section 7.1.1

9.2 THE C.mmp MULTIPROCESSOR SYSTEM

This section reviews the architectural features of the C.mmp system and the kernel of its Hydra operating system. Reported performance of the C.mmp will be also examined in Section 9.2.3.

9.2.1 The C.mmp System Architecture

The C.mmp is composed of slightly modified Digital Equipment PDP-11/40E processors and built out of early 1970s technology. The average time to execute an instruction on a PDP-11/40 is approximately 2.5 μs. The architecture of the C.mmp is shown in Figure 9.1 for a given configuration that consists of 16 computer modules connected to 16 shared memory modules via a 16×16 crossbar switch (S.mp). The functional structure of a typical computer module in the C.mmp is shown in Figure 9.2. The shared memory provides a physical address space of 32 megabytes. The basic modifications made to the processors were to make user execution of certain privileged instructions illegal. Examples of these



Kclock: common master clock

Kinterbus: interprocessor bus control

Figure 9.1 Architecture of the C.mmp. (Courtesy of Fuller et al., *IEEE Compcon*, 1973.)

instructions are HALT, RESET, WAIT, RTI (return from interrupt), and RTT (return from trap).

Further modifications were made to permit address-bounds checking on the stack pointer register R6. These modifications were required for software protection. The operating system is required to deposit some context information on the stack over protected procedure calls. RTI and RTT were modified since they modify the processor status, which must be protected because it is used to control the memory protection scheme. The PDP-11/40E processors were modified further to allow an extended writable control store.

Each processor has an 8K-byte local memory that is used primarily for operating system functions. The principal secondary memories of the C.mmp consist of four drives of 40M-byte disk controllers, three drives of 130M-byte disk controllers, and fixed head disks with zero latency controllers that are used for paging space. The peripheral devices are assigned to the Unibus of specific processors, as shown in Figure 9.2 for one processor. Hence there is no physical sharing of peripherals. A processor cannot initiate an I/O operation on a peripheral that

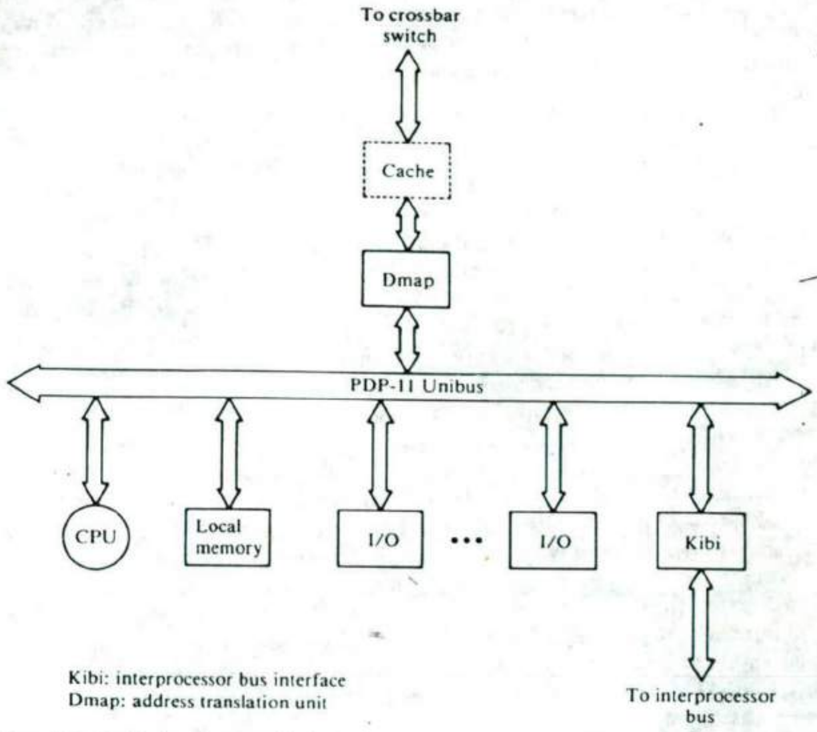


Figure 9.2 A typical computer module in the C.mmp.

is not on its Unibus. Fortunately, the operating system hides many of the asymmetries of the I/O subsystem from the user.

An interprocessor bus which connects the entire set of processors is used to perform the general function of interprocess communication. The bus provides a *common clock* (*Kclock*) as well as an *interprocessor control* (*Kinterbus*). These two logically and functionally separate features travel separate data paths, although they share a common control. Each processor has an *interbus interface* (*Kibi*) that defines the processor's bus address and makes available the bus functions to the software. The bus provides three basic functions, as described below.

The first function is to continuously broadcast a 60-bit 250-kHz nonrepeating clock (Kclock). This is done by multiplexing the clock value onto a 16-bit wide data path in four time periods, with low-order bits first. Any Kibi requesting a clock read waits for the initial time period and then buffers the four transmissions in four local holding registers available to the software. Clock values are often used for unique name generation in the operating system. The otherwise unused high-order four bits of the fourth local register are set to the processor number (bus address) to insure uniqueness when any number of Kibi's read the bus simultaneously. A countdown register is also maintained in each Kibi for interval timing. It may be initialized by a nonzero value in the program; a one is subtracted every 16 μs (timing supplied by the Kclock) and the process is interrupted when the register reaches zero.

The second and third bus functions are the interprocessor interrupts at three priority levels and the control mechanism. Each processor may interrupt, halt, continue or start any processor, including itself. These functions are used only when a drastic action such as systemwide reinitialization is necessary. The control operations are invoked by setting the bit(s) corresponding to the processor(s) to be controlled in a 16-bit register provided by the Kibi for the desired operation. A second 16-bit wide data path is eight-way time multiplexed. Each control operation is assigned a time period. As the appropriate period arrives, each Kibi ORs its control operation register onto the bus and clears the register.

Synchronization of bus accesses and operation specification are accomplished by the multiplexed time periods. The Kibi also inspects the bus to see if the specified operation is being invoked on its processor; if so, the action is performed. Setting the *i*th bit of the Kibi register to one associated with one of the functions will evoke that function on the *i*th processor. Thus, for example, moving a mask of all 1s into the halt register in each Kibi will stop the entire system. Although eight time periods are available, only six are used: three priority levels of interprocessor interrupt, halt, continue and start; the remaining two are ignored.

Probably the greatest limiting factor in building a large computing system from minicomputers is their small address space. In most cases, it is required to be able to address several million bytes of primary memory from the processors. The basic PDP-11 architecture is only capable of generating 16-bit addresses. Although the processor may generate only a 16-bit address, the Unibus supports an 18-bit address, and the shared memory uses a 25-bit address. An address relocation hardware *Dmap* associated with each processor performs the memory address translation. Its relationship with other bus components is shown in Figure 9.2. The processor-generated addresses are divided into eight pages, where each page is an 8K byte unit. Unibus addresses are divided into 32 pages, and the shared memory is divided into 4096 pages.

As shown in Figure 9.3, the two extra bits of the Unibus address are obtained from the program status register (PS) in the processor. These bits may not be altered by any user program. The user programs are actually bound to operate within the eight pages described by a subset of relocation registers. Such a subset is called a *space* and is named by the two bits $\langle 7:8 \rangle$ in the PS. With these two space bits, four address spaces can be specified as (0, 0), (0, 1), (1, 0) and (1, 1). Therefore, four sets of eight registers are provided in each relocation unit, although the stack page is common to all spaces to allow communication across spaces. One of these eight registers in a given address space can be selected by using the high-order three bits $\langle 13:15 \rangle$ of the 16-bit processor address word.

The four address spaces are the heart of the memory-protection mechanisms discussed later. The address-mapping registers and PS registers are both located

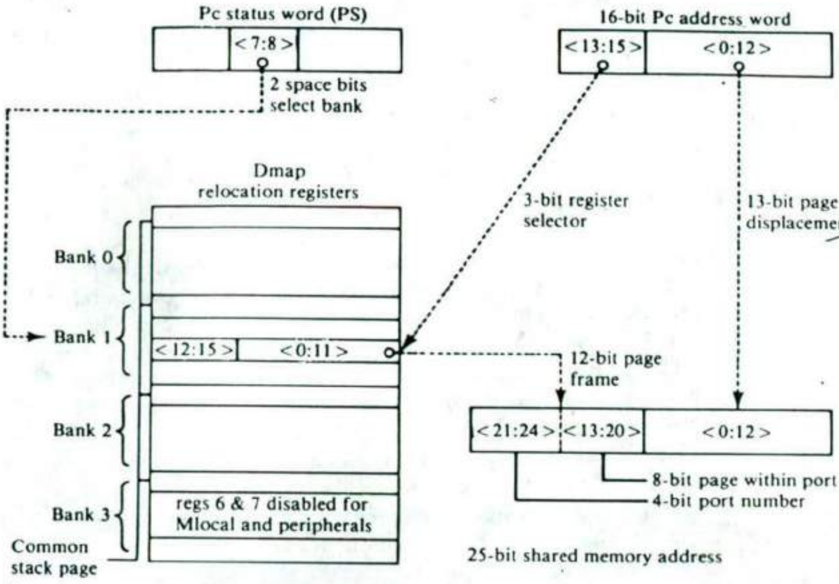


Figure 9.3 Address relocation in the C.mmp.

in the peripheral page, which is addressable via the (1, 1) space bits. The relocation registers in the space described by the (1, 1) space bits are the only ones that are directly addressable and are used exclusively by the kernel of the operating system. Hence, protecting the PS guarantees that no addressability changes may be made without the approval of the operating system. Direct addressability is accomplished by disabling two of the relocation registers in (1, 1) space, one each of Mlocal and the control register bank, for all peripheral devices (including Dmap). With these registers disabled, addresses pass along the unibus unchanged to be received by the addressed register or memory location.

Access to shared memory is performed in two stages: The *relocation* of the 18-bit processor-generated address into a 25-bit address space, and the *resolution* of contention in accessing that memory location. As illustrated in Figure 9.3, the Dmap intercepts the 18-bit unibus addresses (16-bit word plus the two space bits) and translates them as follows: the three high-order bits of the 16-bit word select a register from the bank specified by the space bits. The contents of the register provide a 12-bit page frame number; the remaining 13 bits from the address word are the displacement within that page. The two are concatenated to form the 25-bit mapping shared memory address. This transparency is performed for all memory accesses.

In addition to the 12 page frame bits, there are four bits in each relocation register used for control. They are designated as *no-page-loaded* (nonexistent memory), *write-protected* (read-only), *written-into* (dirty), and *cacheable* bits to control whether values from the page may be stored in a possible per-processor cache. The per-processor cache was planned but not implemented. The cacheable bit would have been used by the operating system to avoid cache consistency. This can be accomplished by indicating pages that are not both shared and writable. Shared and writable pages are never cached.

The shared memory address and (possibly) 16 bits of data, each parity checked, and two bits of access function data are sent to the cross-point switch. The address parity is checked at the switch interface. If the check fails, the request is aborted and the processor interrupted. Data parity is not checked until the data is read from memory. All parity is generated and data parity checked by the relocation unit (Dmap) interface to the bus from the switch.

The switch then routes the request to the memory port specified by the high-order four bits of the address. A port is requested by setting the processor's bit in an initial *request* register. Contention for the port is resolved by periodically gating the request register into a *queue* register, which is left-shifted as the port becomes available. The shifting creates a priority ordered queue: As a bit is shifted out, the corresponding processor is granted access to the port. Processor 15 is assigned the high-order bit; processor 0 the low-order bit, defining the priority. When the queue register is zero, all requests have been satisfied. The request register is again gated into the queue register, cleared, and a new cycle begins. A second request for the same port by a processor must enter via the request register, hence equality of service among the processors is maintained.

This two-level request mechanism also obscures the internal queue's priority ordering to the point that it is of virtually no importance outside the switch, preserving the symmetrical design of the cross-point. The switch's maximum concurrency (16 independent paths) is achieved if all processors request different ports. The cost of address translation, switch overhead (no contention), and round-trip cable overhead is about 1 μs. This is high by today's standards and is more than equal to the access time of the memory.

9.2.2 The Hydra Operating System

The operating system of the C.mmp was based on an experimental kernel called *Hydra*. The *kernel* is the "nucleus" of an operating system. However, we will use the term "kernel" synonymously with an operating system. Hydra was designed as a logically distributed system so that any processor can execute the kernel. Also, more than one processor can execute the kernel concurrently. This parallelism is enhanced by using locks on various data structures and not on the code that references them. There were two basic goals in the design of Hydra. One was to supply primitives that allow most of the facilities generally provided by an operating system to be written as user programs. This will permit the operating system to be effectively tailored to each user's needs. The other goal was to permit any number of user-level definitions of a facility to coexist at the same time. These goals suggest that the kernel be designed as a collection of basic or *kernel mechanisms* of universal user applicability on the C.mmp.

In order to facilitate the design of the kernel mechanisms, the separation of mechanism and policy is required. This principle is called policy-mechanism separation. Such separation contributes to the flexibility of the system because it leaves the complex decisions to the user. Policies are generally encoded in user-level software which is external to, but communicates with, the kernel. Mechanisms are provided in the kernel to implement these policies. The kernel mechanisms are used to provide a protected image of the hardware operation. For example, a user is not permitted to manipulate I/O device control registers, as doing so may allow the user to inadvertently overwrite a protected portion of memory. Although the policy-mechanism separation philosophy is desirable, there are instances in which a kernel mechanism may actually be a parameterized policy. *Parameterized policy* provides the means by which overall long-term policies can be enforced by user-level software and simultaneously can avoid excessive domain-switching mechanisms for decisions which require a fast response. An example of such a policy will be given later.

The testing of Hydra is facilitated by providing language constructs which permit *abstractions* of data. Such abstractions define data types by specifying the storage structures used and a set of functions which operate on it. A property of data abstraction is that the details of representation and manipulation are hidden from the user. This technique is used in the design of the Hydra to enhance the protection mechanism by extending the type definition to specify the representation of virtual resources and the nature of the implementation of various operations on a particular type of resource. Examples of virtual resources are file, directory, semaphore, and page. One of the elegant features of Hydra is that it is an *object*-oriented system. All information is encapsulated in structures called objects which are accessed only through capabilities. Hence, Hydra has capability-based protection mechanisms to support the philosophy discussed.

The set of objects a process can access is its address space. Objects are of variable size and consist of a data part and a capability list (C list). The data part can be expressed as a tuple: representation and type. The *representation* component contains the information and the *type* of an object indicates the nature of the resource. The C list consists of a set of capabilities. The capability is represented by a tuple: unique-name and access rights. The *unique-name* of an object is generated using the 60-bit clock described previously. A process may only perform those operations on an object that are permitted by the *access rights* in the capability through which the process named the object.

The basic unit of a schedulable entity in Hydra is a process. An active process is not bound to a processor. Therefore, a process may migrate from one processor to another during its lifetime. The set of objects defined by the capabilities of a process at a given time defines the execution environment and, hence, the protection environment. This record of the execution environment of a process is called the *local name space* (LNS), an object type.

Another object type often used is the procedure. A *procedure* object contains a list of references to other objects which must be accessed during the execution of the procedure's code. A procedure is thus considered a static entity. There is a unique LNS for each invocation of the procedure. This LNS disappears after the procedure terminates.

Moreover, a procedure object may contain *templates* which characterize the actual parameters expected by the procedure. When the procedure is called, the slots in the LNS which correspond to parameter templates in the procedure object are filled with "normal" capabilities derived from the actual parameters supplied by the caller. This derivation is the heart of the protection-checking mechanism, and the template defines the checking to be performed. If the caller's rights are adequate, a capability is constructed in the new LNS. This LNS references the object passed by the caller and contains rights specified in the template.

System interrupts were provided in the C.mmp architecture to facilitate interprocessor communication. An analogous software mechanism is provided at the user level by the kernel. This mechanism, called *control interrupts*, is used for interprocess communication. When it occurs and is directed to a process, the receiver process transfers control asynchronously to a special address specified by the user. This address is within the addressing domain in which the process is executing at the time. A 16-bit mask is specified by the process that sends a control interrupt. These bits are compared with a mask in the receiver process, and the receiver process is interrupted only if there is a match between one or more bits. Hence adequate protection is maintained. Control interrupts are generally slow; however, they have some properties that make them desirable for error recovery.

The operating system provides an elegant message system for handling communications between processes and thus encourages the use of cooperating processes. The message system uses objects called ports as gateways for processes to send and receive messages between processes. Each port has a set of logical terminals called *channels* which are used to connect between sender and receiver processes. There are basically two types of channels: input and output. Messages are sent from output channels and received in input channels. Two processes can communicate if they each have a capability for the other's port and if a communication path is established between the ports.

To establish a path between the two ports, the output channel of one is connected to the input channel of the other and vice versa. Each port provides a message slot which is a buffer that is used to queue messages. This slot also provides a local mechanism to name messages. The message system can operate in two modes: nonacknowledgement and acknowledgement. In the first mode, the sender sends a message and continues processing without waiting for a reply. Operations in the second require a reply to a send message. A process that attempts to receive a message before its arrival is suspended until the arrival of the message, whereupon the process is unblocked. The message system is also used in I/O communication to provide transparency of the asymmetries of the I/O structure at the user level.

The message system is not very efficient, thus two other synchronization mechanisms, locks and semaphores, are provided. Two types of locks exist in the Hydra. The *kernel lock* makes use of hardware facilities such as interprocessor interrupts which are not available at the user level and therefore can be used only in the implementation of the Hydra. The *spin lock* is available to users as it does not use privileged instructions to implement it. The kernel locks, which are used primarily to provide mutual exclusion for operations on various system queues and tables, pervade the implementation of the kernel.

The kernel lock consists of three components: the lock byte, the sublock byte, and the processor mask word. The lock byte maintains a counter of the number of processes waiting for the lock. A process which wishes to obtain the lock indivisibly increments and tests the lock byte with a single PDP-11 instruction; if the result indicates that the lock is free, it is then locked and the locking process can execute its critical code. Otherwise, the process sets the bit corresponding to its processor in the processor mask word and executes a WAIT instruction with all interrupts except the highest priority IPI (interprocessor interrupt) disabled.

When a process is ready to unlock the lock, it indivisibly decrements and tests the count in the lock byte; if the result indicates that no other processes are waiting, the lock is unlocked and normal execution can continue. If other processes are waiting, the unlocking process sets the sublock byte to one and sends an IPI to every processor with a bit set in the processor mask. These processors resume after their WAIT instructions and indivisibly decrement and test the sublock byte. One random processor will discover the count to be zero, remove itself from the mask of waiting processors, and execute its process's critical code. The other processors go back to waiting.

One disadvantage of locks is that a process which waits during the execution of a kernel or spin lock does not relinquish the processor on which the process is executing. Kernel locks have another disadvantage which involves the overhead of invoking lock and unlock primitives. This overhead is minimized by storing the code for the kernel lock primitives in the processor's local memory. In this case, there is no memory contention and no contention for the lock. However, spin locks have one major disadvantage over kernel locks: When a processor is spinning, it accesses shared memory and thus consumes memory bandwidth which might have been used more constructively.

Generally, when the probability of waiting is low, the lock primitives are very efficient. A study performed on the C.mmp indicates that, although a process may spend over 60 percent of its time executing kernel code, only about 10 percent of accesses to locks cause locking. Moreover, the total fraction of time spent by processors waiting for locks is less than 1 percent. The study also shows that operations performed on data structures while they are locked are small. The overall average time spent in a critical section is about 300 μs.

Situations often arise in which lengthy blocking cannot be avoided in a system. In such cases, using the lock primitive will result in excessive waste of resources. For example, after a process issues an I/O request, a significant amount of time may elapse before its completion. In other cases, relatively large sections of code may require exclusive use of a data structure. In each case, when lengthy blocking is possible, Hydra provides two types of semaphore mechanisms, *kernel semaphore* (K-Sem) and *policy semaphore* (P-Sem). Both are implementations of the generalized or counting semaphores. The main difference between the semaphore and lock mechanisms is that, if a process blocks as a result of executing a **P** operation on a semaphore, some of the resources belonging to the blocked process will be relinquished until the process is able to resume execution. **K-Sems** are designed to be most efficient when no blocking occurs. Under these conditions, they are about as efficient as lock primitives. When blocking occurs, **K-Sems** are considerably more time consuming. Blocking requires the manipulation of the process queue and, in the case of a **P** operation, saving the state of the blocking process. If the average blocking period is large, the context-swap overhead of about 8 ms needed to block and wake up a process is significant. If the blocking period is less than the context-swap time, then the use of a **K-Sem** actually decreases overall system throughput as a result of processor thrashing.

When a suspended process is unblocked, it may have to wait until the resources it requires are released by higher priority processes which are competing for them. A policy of fairness among competing users is enforced by providing a more conservative form of semaphore mechanism. This **P-Sem** has an additional feature. After a predetermined wait time elapses, the primary memory belonging to the blocked process is made eligible for swap-out to a drum and the process is returned to its policy module for reconsideration of long- or medium-term scheduling. The short-term scheduler is called *kernel multiprogramming system* (KMPS). This is a mechanism which one or several policy modules may control via parameterization. Hence, KMPS is an example of a parameterized policy.

For fair multiplexing of all processors, the KMPS ensures the execution of the highest priority feasible process by using a preemptive priority resume policy. Processes within the same priority are scheduled in a round-robin fashion. The scheduling parameters of a process are its priority, a processor mask, and a maximum working set size. The priority is an integer value between 0 and 255, inclusive. The processor mask indicates the subset of processors on which the process can execute. The mask is required because of the earlier heterogeneity of the processors and the asymmetry of the I/O subsystem. The three parameters are set by the policy module. Under certain circumstances, the KMPS will return a process to its policy module. An example of this occurs if the KMPS blocks on a policy module's semaphore.

9.2.3 Performance of the C.mmp

Experience with the C.mmp shows that unavailability of the crossbar switch used did not pose a problem. Hard failures of the switch were rare, perhaps because of the regularity of its structure. The processors, memory modules, and the interprocessor bus were far less reliable by comparison. This demonstrates that the term "reliability" is relative. Although hard switch failures have been very rare, one observed type of transient error demonstrates some of the difficulties in successfully achieving logically distributed responsibility. When a process or device is accessing memory through the switch, the accessed module is unavailable to other processors or devices for the duration of the transaction.

A device controller experiencing an error may sometimes abort a transfer without correctly terminating the protocol between its unibus and the switch, with the result that the accessed memory module is "hung" with all its access paths blocked indefinitely. While the distributed nature of the switch allows access paths to other memories to function normally, any other processor trying to access the hung memory will also wait indefinitely. It is impossible to detect or to recover from this condition with software; only a manual reset can clear the memory and free the waiting processors. This problem uncovers a basic design flaw in the switch: its correct functioning depends on the correct functioning of its attached devices. The problem was minimized by modifications to the controllers. A better solution is for all switch transactions to "time-out" automatically after a predetermined period.

In any system, three basic steps are required to handle faults successfully: detection, diagnosis, and recovery. Fault detection is enhanced by system modularity and consistency checking. All techniques for consistency checking utilize some degree of redundancy in order to recognize inconsistencies. The self-identifying data structure employed for type checking within the Hydra kernel illustrates one set of types of redundancy. Many errors can be detected by associating watchdog timers with important system resources. This is a natural approach in a multiprocessor, where components can monitor each other relatively easily. Implementations of this technique differ, but the basic idea is that the timer will in some way raise an error-condition indicator if it is not reset within some specified time limit.

In the Hydra, a watchdog mechanism of this type has been implemented to detect processors that halt or become trapped in endless loops. The watchdog depends on a word in main memory that is shared by all processors. One bit in this word is assigned to each of the processors; if a processor is correctly executing Hydra, it should frequently execute code, causing it to set its designated bit. Every 4 s each processor also checks whether other processors have set their bits. Thus each processor must set its bit at least once every 4 s or it will be detected as malfunctioning and error recovery will be initiated. Diagnosis in the C.mmp/Hydra is carried out by a mechanism called the *suspect-monitor*, which is invoked whenever a serious error is detected. The processor that detects the error is designated the *suspect*, and the rest of the system is quiesced.

One processor is then chosen at random to act as the *monitor*. The monitor processor tests the suspect by stepping it through a simple diagnostic, following the suspect through each step in the diagnostic by watching a shared word of memory. Any failure in this sequence is grounds for removing the suspect processor from the configuration immediately. If the diagnostic completes successfully, the error is logged on disk and more extensive steps are taken to try to determine its cause. The exact tests depend on the nature of the error but include an extended processor diagnostic and attempts to retry memory fetches that caused parity failures.

Early versions of the suspect-monitor proved ineffective despite the extensive testing performed. This was due to transient errors which occur infrequently except under heavy loads. Thus, the suspect diagnostic rarely caught a processor in the act of failure, and yet another failure often occurred soon after the system was restarted. Such restarts are fast, typically taking less than 2 min; hence relatively high availability was maintained. However, the loss of user jobs represents an extreme inconvenience. The suspect-monitor system's capabilities was improved by providing a system of processor-error counters. These counters record the occurrence of particular errors on each processor, and if they exceed a threshold for total errors or for a particular error class, the offending processor is amputated or removed from the configuration or quiesced. The counters are also periodically right-shifted, causing them to decay over time so that they measure error frequency rather than simply providing an error count. A flaw in this scheme is that error counters are maintained for processors only. The processor that detects an error is charged with causing it, even if no concrete evidence to that effect exists; the error may actually have been caused by another processor, bad memory, or even by software. In practice, the high degree of symmetry among the processors makes it improbable that one processor will detect an error for which it is not responsible with a high enough frequency to exceed its error threshold.

It was found that parity errors are the single most common failure mode. While hard failures occur regularly, most parity failures are transient, suggesting that perhaps error counters should be implemented for memory pages as well as for processors. Although considerable emphasis was placed on error detection and diagnosis in the C.mmp/Hydra, the recovery mechanisms are insufficient to preserve integrity of smaller granules of computation.

To demonstrate the effect of memory contention in an execution environment of the C.mmp, an experiment is described below. This experiment consists of finding the root or zero of a function. The parallel algorithm used to solve this problem was described in Section 8.4.2. There are two implementations of the algorithm. In the first case, the code of the algorithm was stored in a single memory page which was shared by all processes. The second implementation of the algorithm provided separate pages of code for each process. Therefore, the first implementation will encounter more memory conflicts. The experiments were conducted on a C.mmp configuration consisting of Model 20 and 40 PDP-11s. The Model 20 is typically 50 to 60 percent slower than the Model 40. Figure 9.4 illustrates the effect of memory contention on the performance of the two implementations of the root-finder algorithm. Notice that, beyond a certain threshold, an increase in the number of processors will produce a negative effect on the performance for the implementation with shared code page.

This algorithm was also used to study the performance of the various synchronization primitives discussed above. Recall that the key feature of the parallel algorithm is that it is synchronous. The nature of the parallel solution demands the synchronization policy. Figure 9.5 shows the elapsed time required by the root-finder algorithm for varying numbers of slave processes and four different synchronization mechanisms. For these measurements, the function-evaluation time was distributed normally with a mean of 72 ms and a standard deviation of 18 ms. The parameter e refers to the wait-time constant for policy semaphores. The curve labeled PMO corresponds roughly to the case where $e = 0$. While

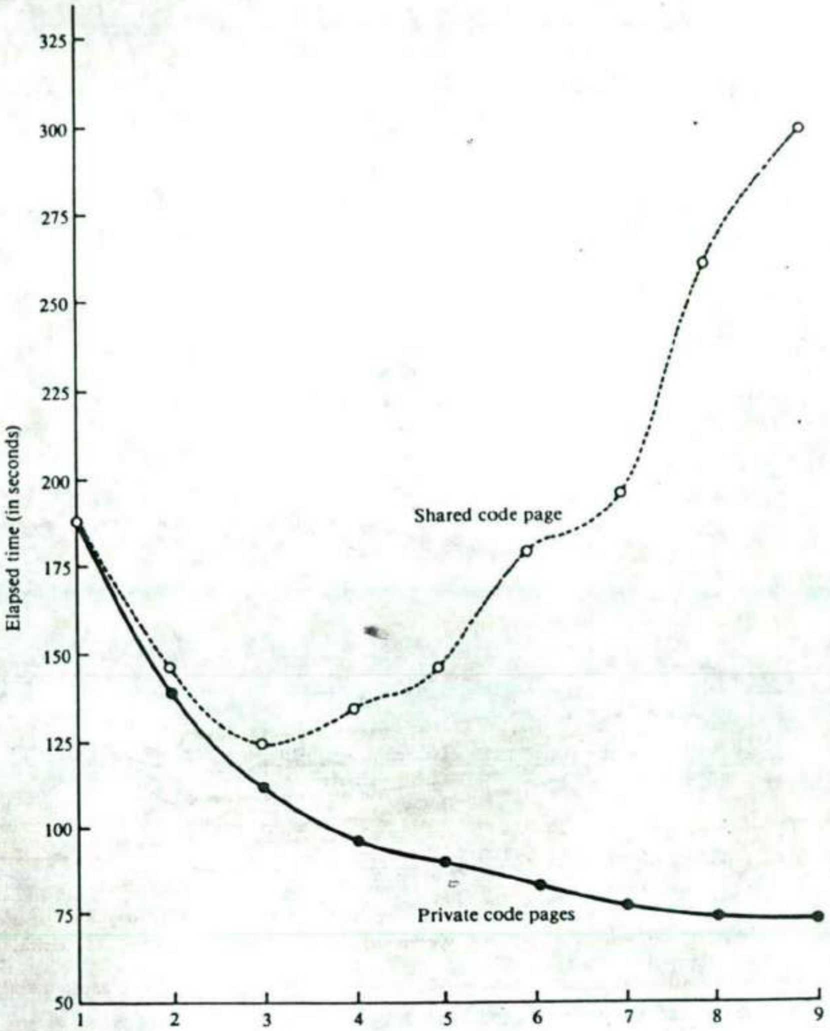


Figure 9.4 Performance degradation due to memory contentions. (Courtesy of Oleinick, Carnegie-Mellon University, 1978.)

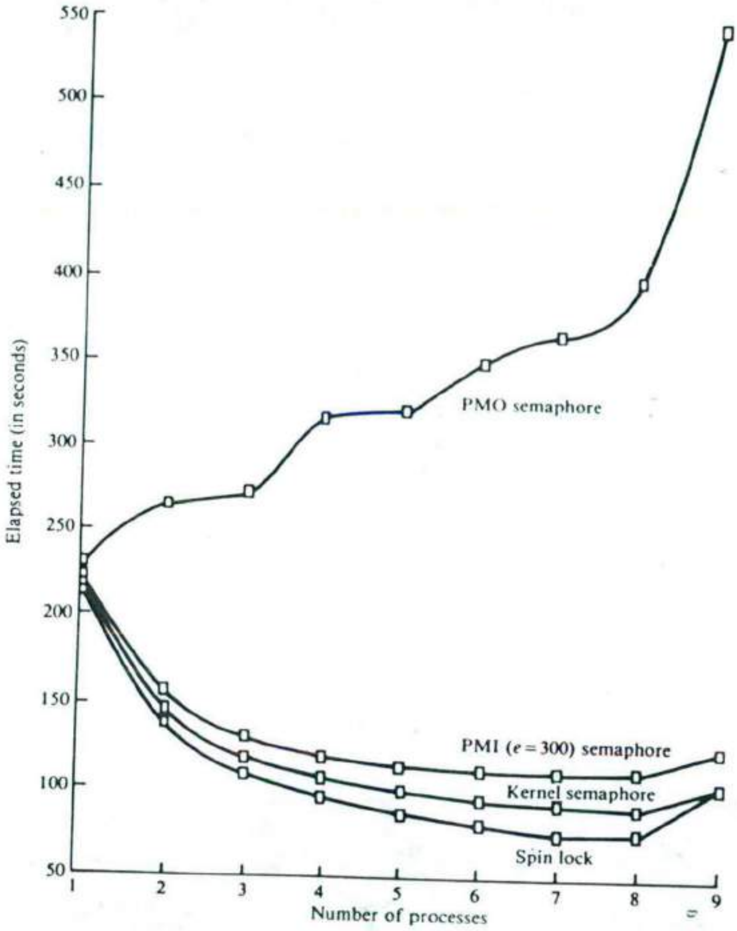


Figure 9.5 Effect on performance of different synchronization mechanisms. (Courtesy of Oleinick, Carnegie-Mellon University, 1978.)

many factors affect the performance of this algorithm, the damaging effect of an inappropriate synchronization mechanism is clearly demonstrated.

9.3 THE S-1 MULTIPROCESSOR SYSTEM

The system architecture of the S-1 multiprocessor system is presented below. The system is being developed under the auspices of the United States Navy. Described below are the basic organization and characteristics of the uniprocessor Mark IIA used in the S-1 construction. This processor is expected to have a performance level comparable to the Cray-1. The S-1 consists of 16 uniprocessors which share 16 memory banks via a crossbar switch. Each processor has a private cache. The software development to be presented includes the operating system for the S-1.

9.3.1 The S-1 System Architecture

The S-1 multiprocessor system is developed to perform computations at an unprecedented aggregate rate on a wide variety of scientific problems. It can be described as a high-speed general-purpose multiprocessor. The S-1 is implemented with the S-1 uniprocessors called Mark IIAs. For a large class of numerical problems, the Mark IIA is expected to achieve a computation rate roughly an order magnitude greater than that of the Cray-1 computer. Figure 9.6 shows the logical structure of a typical S-1 multiprocessor. This structure includes 16 independent Mark IIA uniprocessors which share 16 memory banks through a crossbar switch. Each memory bank can contain up to 2^{30} bytes of semiconductor memory and hence a total physical address space of 16 gigabytes (2^{34}).

Large memories are crucial for the efficient solution of many large problems such as those found in the three-dimensional physical simulations of the Monte Carlo intensive studies. These studies are of great current interest in applications ranging from incompressible fluid flow studies to acoustic ray tracing in highly stratified media. The large memory addressability of the S-1 essentially eliminates the programming cost associated with managing multiple types of computer system storage. Each processor-to-memory bank connection can transfer one word per 50 ns, resulting in a peak data-transfer rate of 320 M words/s.

The crossbar switch is designed to provide access for multiple memory requests. The service discipline for memory requests is such that no processor gets two accesses to a memory bank while another is attempting to access the same bank. The crossbar switch also handles interprocessor communications. The S-1 multiprocessor system has the capability of using dual crossbar switches for reliability and a front-end (diagnostic-maintenance) processor to remove a failing switch and substitute an alternate switch. Although the growth rate of such a "square" crossbar is asymptotically $O(N^2)$, where N is the number of processors or memory banks, the S-1 crossbar is estimated to cost somewhat less than a single S-1 uniprocessor. Less than 25 percent of the switch, or 0.8 percent of the total system cost, exhibits an $O(N^2)$ growth rate. The remainder of the system cost exhibits an $O(N)$ growth rate. This is valid if we assume that half of the total system cost is invested in the memory. This suggests that it is economically feasible to implement crossbar switches for multiprocessor systems with more than 16 processors. However, this suggestion cannot be drawn for a multiprocessor with low-cost processors, where the cost of the crossbar switch may dominate.

Each processor in the S-1 has a private cache which is transparent to the user. As discussed in Section 7.3, the association of a private cache with each processor introduces the problem of cache consistency. To solve the cache coherence problem, the S-1 multiprocessor includes a design closely related to the dynamic model discussed in Section 7.3. In the S-1, a small tag is associated with each line or block

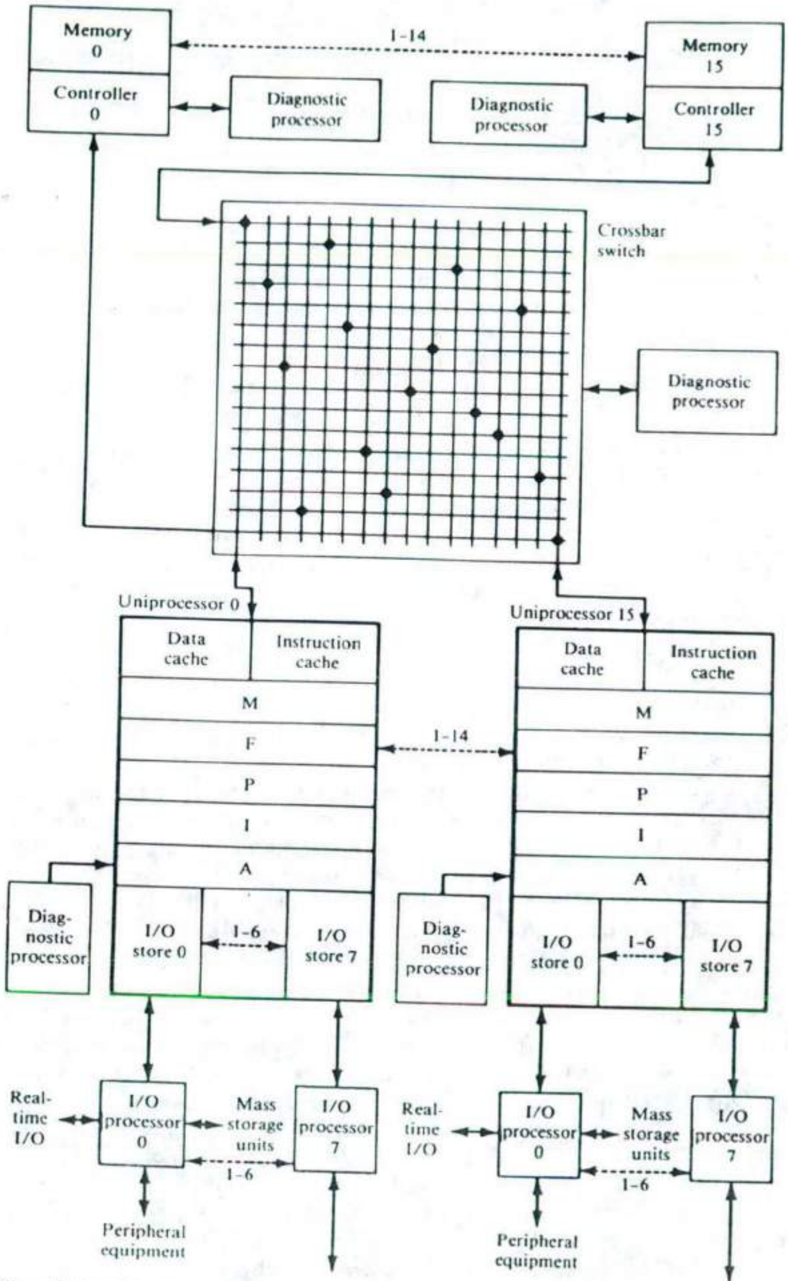


Figure 9.6 Logical structure of the S-1 Mark IIA multiprocessor. (Courtesy of S-1 project at Lawrence Livermore National Laboratory, 1979.)

(a set of 16 words) in the physical memory. This tag identifies the unique member uniprocessor (if any) which has been granted permission to retain (that is, own) the block with write access. It also identifies all processors which own the line with read access.

The memory controller allows multiple processors to own a line with read access. However, it responds with a special error flag when a request is received to grant read or write access for any block which is already owned with write access. The special flag is also set when a request is received to grant write access for any block which is already owned with read access. Any uniprocessor receiving such an access denial is responsible for requesting other uniprocessors to flush or purge the contested block from their private caches. It does this by using send and receive messages via the interprocessor-interrupt mechanism within the crossbar switch. The procedure outlined above thus dynamically maintains cache consistency.

The S-1 design provides a somewhat unconventional I/O subsystem which consists of many microcoded I/O channels. Each channel is managed by an I/O processor. The I/O subsystem also contains I/O buffers or memories which are accessible as part of the S-1 processor's address space. There is a 2K single-word buffer for each channel. These I/O memories are shared between an S-1 processor and an I/O processor. On output, data is placed into the I/O memory and then the I/O processor is signalled to transmit the data to the device. Input is handled similarly. These I/O memories are managed and assigned through the address-space management mechanism of the S-1 processor. Thus processes may perform I/O to devices if they have access to the I/O memory shared with that I/O processor. The S-1 architecture places little constraints on the I/O processor, which may be a commercially available minicomputer or specially designed hardware.

The I/O interconnection structure is designed to be simple and possess some degree of fault tolerance. Each I/O peripheral processor may be connected to input-output ports on at least two uniprocessors, so that the failure of a single uniprocessor does not isolate any input-output device from the multiprocessor system. This fault-tolerance approach is used extensively in the design of the S-1 to achieve high reliability and availability. For reliability, all single-bit errors that occur in memory transactions are automatically corrected, and all double-bit errors are detected regardless of whether the errors occur in the crossbar switch or in the memory system. The crossbar can be configured to keep a backup copy of every datum in memory so that the failure of any memory bank will not entail the loss of crucial data. System maintenance is facilitated by connecting a diagnostic computer to each uniprocessor, each crossbar switch, and each memory bank. This diagnostic computer can probe, report, and change the internal state of all modules that it monitors.

9.3.2 Multiprocessing Uniprocessors

The performance of each Mark IIA is achieved by extensive pipelining due to advances in microcode, hardware structure, and implementation technology.

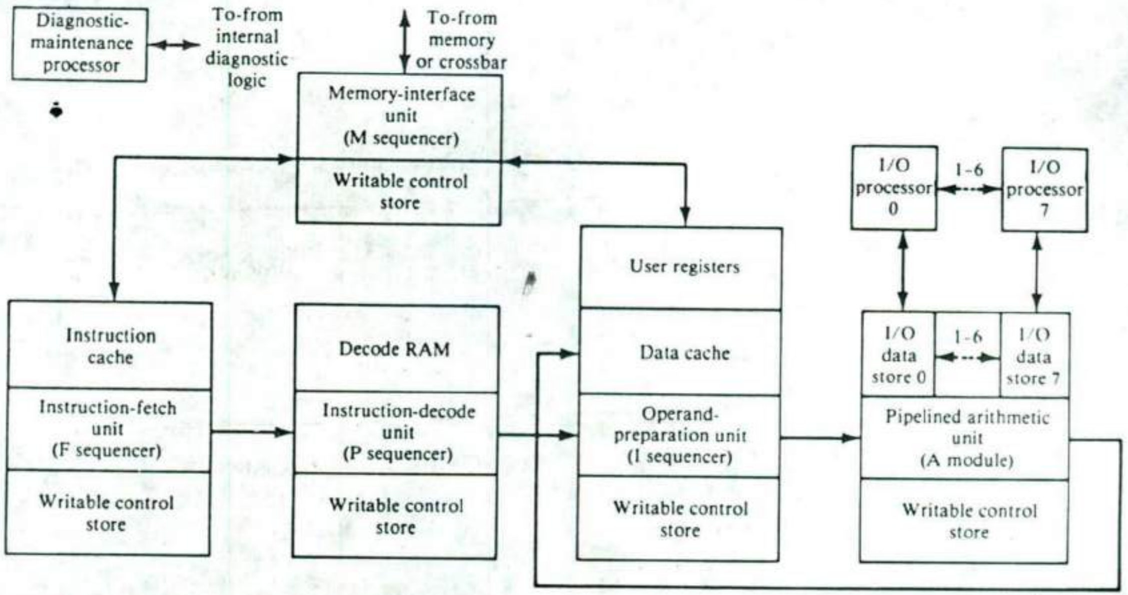
Each uniprocessor has a virtual address space of 2^{29} thirty-six-bit words, uniformly addressable in quarterwords, halfwords, singlewords, and doublewords. The processor has 16 register sets for fast context switching and each register set has 32 general-purpose 36-bit registers. Two registers are used to maintain the processor and user status. The virtual address space is segmented to promote modular sharing and separate access for each user or task. Variable size segments are implemented and bounds checking is performed for reliability. The protection mechanism used in each processor is similar to the ring protection system in the Multics. Separate address spaces are allowed for each of the four rings which provide concentric levels of privilege. Gates at each level provide the necessary protection interface for procedure calls to the kernel.

Facilities are included to perform arithmetic and logical operations on various data types. The data types include boolean, integer, floating-point with a set of rounding modes, complex, vectors and matrices. The instruction set is optimized to contain features for compilers and for operating system efficiency as well as for arithmetic-intensive and real-time applications. In addition, special I/O instructions are provided to manipulate the contents of the I/O buffer. The interrupt architecture consists of vectored interrupts with vector locations which can be changed dynamically. Interrupts can be individually enabled or disabled and can be programmed in eight priority levels. The processor priority is also able to be reset. The uniprocessor has been designed to permit high-speed emulation of general instruction set architectures.

The uniprocessor is designed especially to facilitate pipelined parallelism in the fetching and decoding of instructions, the associated fetching of instruction operands, and the eventual execution of instructions. The preparation and execution of instructions that specify both scalar and vector operations are pipelined. Every instruction proceeds through multiple pipeline stages, including instruction preparation, operand preparation, and execution. Figure 9.7 depicts the internal logical structure of the S-1 Mark IIA uniprocessors. The processor consists of five major sections which are extremely fast, relatively special-purpose programmable controllers that operate in parallel to provide high performance.

Four sections that form the instruction pipeline are for instruction fetch (F sequencer), instruction decode (P sequencer), operand preparation (I sequencer), and arithmetic execution (A module). These sections are internally pipelined to achieve a maximum instruction-issue rate of one instruction per 50 ns, which is equivalent to a maximum data throughput rate of 720 million bytes/s. The maximum computation rate of the pipeline is 400 megaflops. The sequencers and the A module are heavily microcode controlled with a total of 2.5 million control store bits with a total microword width of 996 bits. A microcode is an architecture which defines very low-level program that precisely specifies the operation of every pipeline stage.

Figure 9.8 shows the instruction unit pipeline diagram to consist of 11 major segments. Some stages of the pipeline, particularly those dealing with operand-address arithmetic and instruction execution, necessarily have a wide variety of functions, since the pipeline must process a wide variety of instructions. This



Diagnostic-maintenance processor
To-from internal diagnostic logic
To-from memory or crossbar
Memory-interface unit (M sequencer)
Writable control store
I/O processor 0
1-6
I/O processor 7
User registers
Data cache
Operand-preparation unit (I sequencer)
Writable control store
Instruction cache
Instruction-fetch unit (F sequencer)
Writable control store
Decode RAM
Instruction-decode unit (P sequencer)
Writable control store
I/O data store 0
1-6
I/O data store 7
Pipelined arithmetic unit (A module)
Writable control store

Figure 9.7 The internal logical structure of the S-1 Mark IIA uniprocessor. (Courtesy of S-1 project at Lawrence Livermore National Laboratory, 1979.)

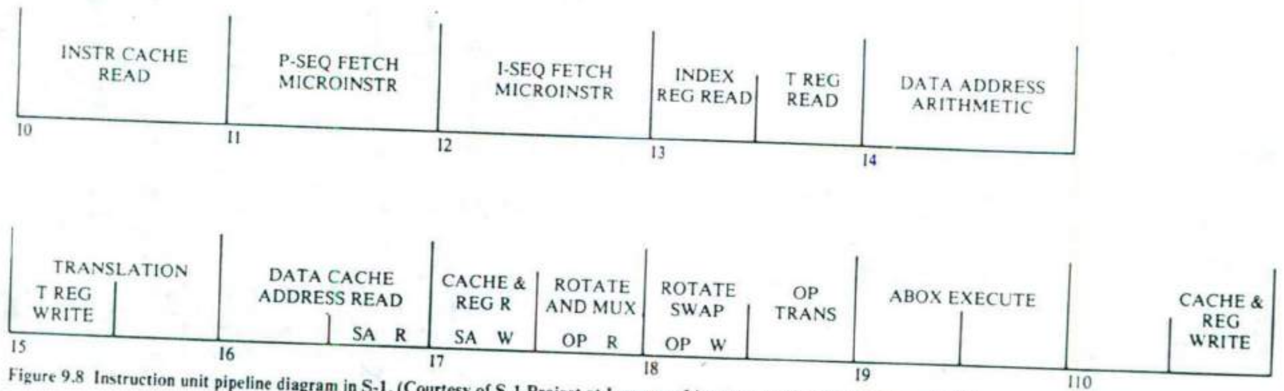


Figure 9.8 Instruction unit pipeline diagram in S-1. (Courtesy of S-1 Project at Lawrence Livermore National Laboratory, 1979.)

variability in operation is effected through the extensive use of microcode. The variability built into the microcode-controlled pipeline also facilitates high performance emulation of other computers. The instruction sequence starts with an instruction cache read. The private cache associated with each processor is partitioned into instruction cache and data cache to achieve a high bandwidth. Each cache is organized as a four-way set-associative memory with 16 words per block. Both caches are vector-structured; that is, in the instruction cache, a cache read can retrieve three consecutive instruction words starting at any word boundary. In the data cache, reads can retrieve eight consecutive halfwords starting at any halfword boundary. The instruction and data caches have capacities of 16K bytes and 64K bytes, respectively. The cache replacement policy is the least recently used algorithm.

The M sequencer predecodes instructions when loading the instruction cache from memory on a cache miss. Predecoded information includes instructions of a length of one, two or three words, branch offset and branch prediction. The predecoding of instructions allows branches which are relative to the program counter (PC) to be computed in one cycle within 8K bytes of the PC. Branch instructions are predicted to allow the pipeline processing to continue. The success or failure of previous predictions are used to make better predictions. This is facilitated by recording the prediction for each location in the instruction cache with a single bit. For the first execution, an initial prediction is used based on the instruction type. A prediction failure causes an opposite prediction to be tried on the next execution. This technique has been found to correctly predict about 98 percent of all instructions for a typical compilation.

The P sequencer is used to calculate constants and register operands. Each P sequencer instruction calls an I sequencer subroutine. The I sequencer subsequently calculates all operand types and complex branch addresses. An operand queue (not shown) exists between the I sequencer and the pipelined arithmetic A module to buffer up to 16 operands of 1 to 16 bytes each. This buffering smooths out the flow of operands between the I sequencer and the A module. Also, the write queue (not shown) is placed between the A module and the data cache or register to keep track of pending writes from the A module. This queue also detects attempts by the A module to use data-cache locations which have been scheduled to be written by other units (such as other processors). The A module, which runs at twice the rate of other segments, performs the execution part of instruction processing.

The address-translation mechanism ingeniously maps a 31-bit virtual address into a 34-bit physical address, providing both segmentation and paging. It provides four different virtual address spaces, one per ring, which may overlap. A page is 4096 quarterwords long. Because a single address space may contain as many as 2^{19} pages, it is evident that the page mapping tables may themselves be paged. The address translation mechanism has four different steps. Instead of a giant page table of 2^{19} entries, it uses many little page tables each of 16 entries long. Hence, not every page table needs to be in memory at once. The 16 pages pointed to by one page table make up a *segmentito*.

A giant table called a *descriptor segment* contains a pointer to each of the (at most) 2^{15} page tables for each of the four virtual address spaces. Hence, there are at most 2^{17} page tables. If the descriptor segment were placed in memory permanently, an address reference would require two translations: one to find the proper page table and another to find the proper page. However, the descriptor segment itself is composed of pages which are grouped into segmentitos, so that an address reference would first require two translations. The first finds the appropriate point in the descriptor segment, and then two more translations find the target address. Figure 9.9 traces the entire address translation process.

A register called the *descriptor segment pointer* holds the 34-bit physical address of the first word of the *descriptor segmentito table* (DST). Because the descriptor segment points to (at most) four sets of 2^{15} segmentitos and each pointer requires eight quarterwords, the descriptor segment never exceeds 2^{20} quarterwords. That translates into a maximum of 16 segmentitos, which implies that there are at most 16 entries (called *segmentito table entries*) in the DST. The two-bit number of the ring being accessed together with the least two bits of the virtual address select an entry from the 16 in the DST. In turn, that entry points to the physical address of the first word of a *descriptor page table* (DPT), which has an entry (called a *page table entry*) for each of the 16 pages comprising that segmentito. Bits ⟨28:25⟩ of the virtual address select one entry from the 16 in that particular DPT, which points to one page of the descriptor segment itself. The descriptor segment contains pointers to segmentitos that make up the four virtual address spaces. The address translation process can be followed through to obtain the physical address.

It should be noted that the entire mapping structure provided need not be used. A segmentito or page table entry may be full either because the corresponding segmentito or page is absent from memory or because the virtual address space in question is smaller than the maximum allowable size. The address translation process outlined above will be inefficient if every address translation goes through many indirect references. This is alleviated by providing *map cache units* (lookaside buffers) which hold the most recent translations.

In addition, each map contains an 11-bit address-space identification field which allows translations for multiple users to coexist in the map caches. The instruction address translation unit is one-way set-associative with 1024 entries. Two copies are provided to translate addresses of the beginning and end of multiple-word instructions to avoid faulting within an instruction. The operand address translation unit is four-way set-associative with 1024 entries. Two copies are also provided to translate addresses of the beginning and end of multiple-word operands in an instruction. This scheme also avoids faulting within an instruction cycle.

The expected performance of Mark IIA uniprocessors is compared with the CDC 7600 and the Cray-1 on several important benchmark miniprograms in Table 9.2. Notice that the Mark IIA computes these benchmarks at roughly the same speed as the Cray-1 and almost twice the speed of the CDC 7600. The Cray-1 has a performance of two to four times greater than the CDC 7600. The S-1 results assume the use of 36-bit floating-point numbers. However, neither the CDC 7600

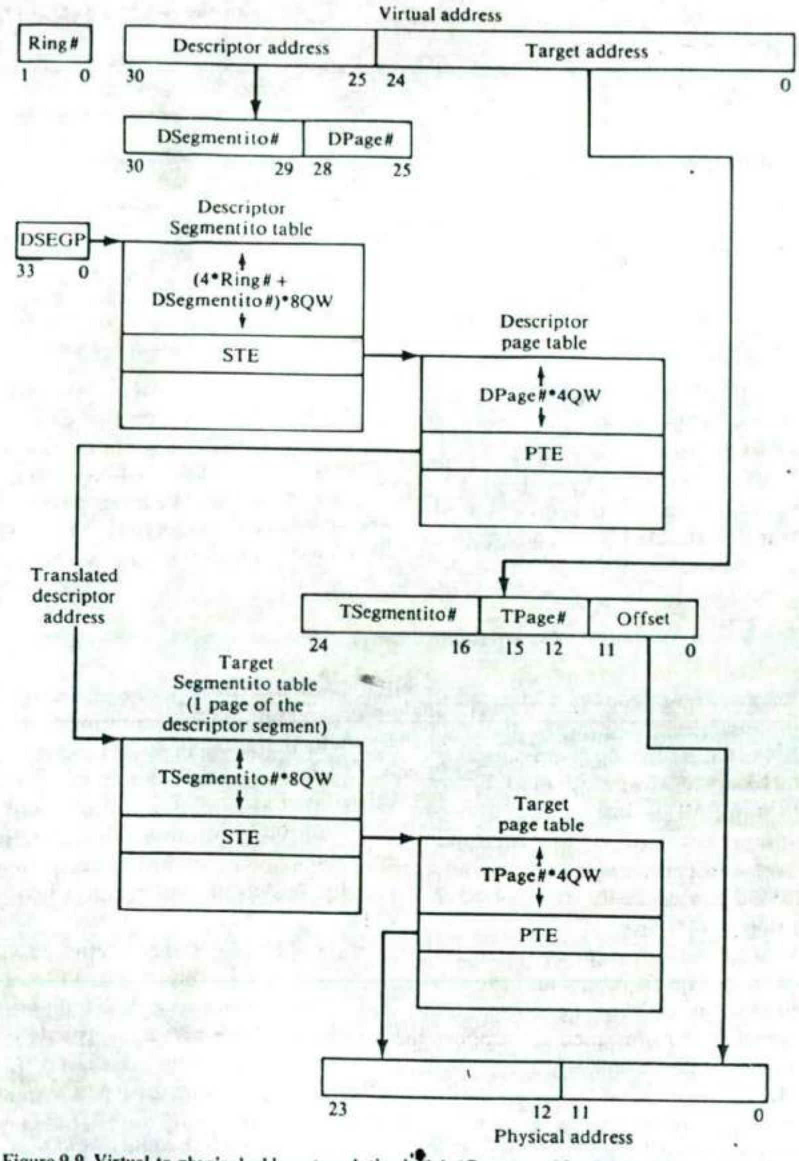


Figure 9.9 Virtual-to-physical address translation in S-1. (Courtesy of S-1 Project at Lawrence Livermore National Laboratory, 1979.)

Table 9.2 Comparison of the expected performances of the S-1 Mark IIA Uniprocessor, the Cray-1, and the CDC 7600.

		Computation rate, megaflops				
		S-1 Mark IIA		Cray-1		
Miniprogram	Miniprogram function	Scalar	Vector	Scalar	Vector	CDC 7600
1	Hydro excerpt	9.1	59	9.3	71	5.3
2	Unrolled inner product	11	74	8.8	47	6.6
3	Inner product	8.0	65	4.4	62	4.6
5	Tridiagonal elimination	7.5	7.5	7.6	7.6	4.0
7	Equation-of-state excerpt	13	46	12.6	80	7.3

nor the Cray-1 provides a low-precision floating-point format. The Cray-1 has only a 64-bit word format. For signal or real-time processing applications, the Mark IIA is expected to perform about four times better than the Cray-1. The S-1 Mark IIA uniprocessor is built out of ECL 100K medium-scale integrated circuits in performance-critical areas and ECL 10K circuits elsewhere. However, the S-1 is still not realized and the expected performance being reported may be rather optimistic.

9.3.3 S-1 Software Development

The major software areas addressed for the S-1 multiprocessor system are the programming language support, single-user and multiuser operating systems. the advanced operating system, and the system design facility. Basic languages supported by S-1 include Pascal, Fortran, C, Ada, New Implementation of LISP (NIL), and FASM—the Mark IIA assembler. The Pascal compiler was enhanced with improved type definition, module definition, exception handling, and additional control constructs. The C language will be based on the Unix implementation and can be easily transported to the single-user, multiuser and advanced operating system.

The *single-user operating system* (OS-0) is a simple stand-alone system which runs a single task at a time and provides only basic I/O functions. The OS-0 was operational on the Mark I processor and used in the hardware system design tool. The OS-0 is also designed to support the testing of processors and provide a minimal base for other operating systems. The *multiuser operating system* (OS-1) to be developed will be based on the Unix operating system because it is a small, relatively powerful system and has demonstrated a suitability for transport. Moreover, it is well known to a large community and has a large body of software available from the user community.

The advanced operating system for the S-1 is the full-functionality *Amber*. The Amber supports a mix of applications which include real-time systems (e.g., signal processing), interactive use (e.g., program development), computation-intensive problems (e.g., physical simulation), and secure environments for data. It also supports full use of the S-1 architectural features by providing multiprocessor support, the management of large, segmented address space, and exploitation of reliability features. The Amber O/S combines functions of the file system and virtual memory. The file directory structure is hierarchical and tree structured. Files are represented as segments. Segmentation facilitates dynamic linking. A demand-paging policy is used to copy pages directly between disk records and main memory. Page replacement works globally on all of main memory and uses the approximate least-recently-used algorithm for eviction of pages. The LRU is not always optimal for some applications such as real-time applications. In such cases, other placement and replacement algorithms are used.

The Amber O/S supports multitasking by the division of problems into cooperating tasks. It also provides low- and high-level scheduling features. The low-level features provide simple mechanisms for real-time applications. Examples are priority scheduling with round-robin queues, dedicated processor assignments, and interrupt processing. The high-level scheduler may implement complex features such as resource allocation and load balancing on multiprocessor configurations. Interprocess communication techniques such as message channels are provided. Other synchronization techniques supported are software interrupts and event notifications. Time-outs on event waits are also implemented. The Amber also possesses features to enhance availability and maintainability. Time-outs on all waits and suspension of processes are performed to prevent deadlock situations. Monitor tasks run concurrently with user and system tasks to detect hardware malfunctions.

9.4 THE HEP MULTIPROCESSOR SYSTEM

The Heterogenous Element Processor (HEP) is a large-scale scientific multiprocessor system which can execute a number of sequential (SISD) or parallel (MIMD) programs simultaneously. The system contains up to 16 process execution modules (PEM) and up to 128 data memory modules (DMM). The PEMs or DMMs are connected with the I/O and control subsystem via a high-speed switching network. The PEM is the computational element of the HEP. In this section, we describe the architecture of the HEP, the organization of the PEM, and extensions made to the programming language to facilitate parallel processing on the HEP.

9.4.1 The HEP System Architecture

The HEP is the first commercially available MIMD multiprocessor system. An example configuration of the HEP with 28 switching nodes is shown in Figure 9.10. This configuration consists of four PEMs, four DMMs, a mass-storage subsystem, an I/O control processor, and node connections to four other devices. We shall describe the mass-storage subsystem and the switch network in this

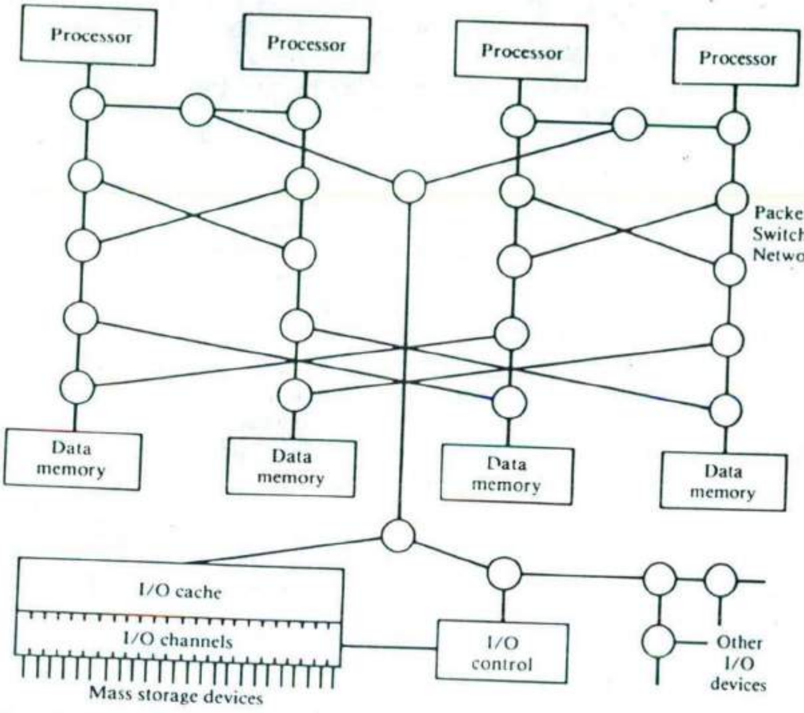


Figure 9.10 The architecture of a typical HEP system with four processors. (Courtesy of Denelcor, Inc., 1982.)

section. All instructions and data words in the HEP are 64 bits wide, although data references within the PEM can access halfword, quarterword, and bytes.

The mass-storage subsystem The mass-storage subsystem consists of three major components. A large MOS buffer memory provides an I/O cache function to mask the seek and rotational delays of disks. Disk storage modules provide storage increments of 600 megabytes. I/O channels couple the disk storage modules to the I/O cache and are controlled by the I/O control processor. Figure 9.11 illustrates the components of the mass-storage subsystem. The cache memory, which is eight-way interleaved with a 400-ns cycle time, may be expanded from the initial 8 megabytes to 128 megabytes in increments of 8 megabytes. Cache accesses are only in full words; at peak rate, a word can be accessed every 50 ns. The system can handle up to 32 I/O channels, with each channel supporting a transfer rate of up to 2.5 megabytes/s. Therefore, the cache memory can accommodate all channels simultaneously, thus yielding a potential transfer rate of 80 megabytes/s.

Figure 9.12 identifies the relationship of the various elements in the mass-storage subsystem (MSS). In this figure, a PDP 11/44 is used as an I/O and control

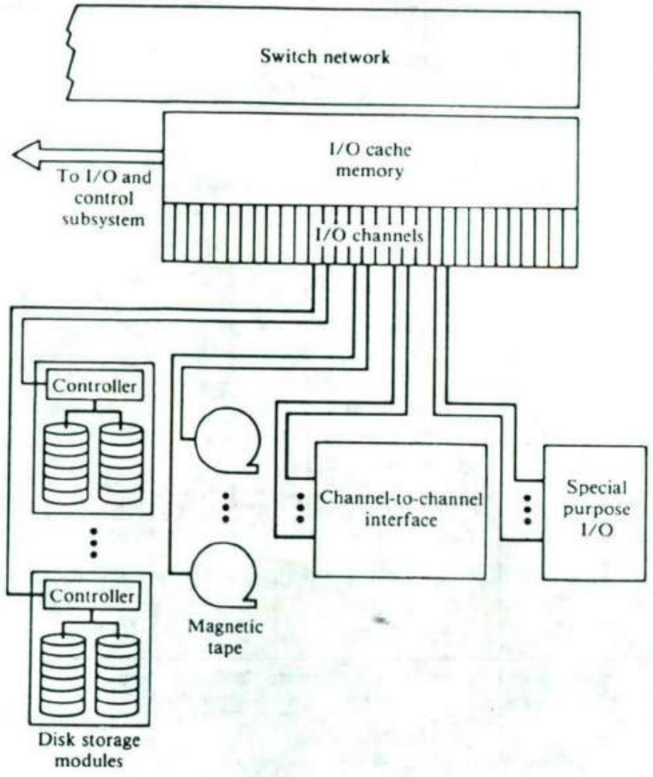


Figure 9.11 The mass storage system (MSS) in the HEP. (Courtesy of Denelcor, Inc., 1982.)

processor (IOCP) to control the communication between the disk controllers and their interfaces to cache memory. This control is accomplished via an IOP command bus. In particular, the IOCP is used by the HEP system software as the scheduler for all I/O activities on the MSS. The IOCP also has direct access to cache memory control for data transmission via an allocated channel. This permits the IOCP to have direct access to data in the cache memory. Each disk storage module consists of two disk drives. One of the first two drives is dual ported in order to permit a direct connection from the drive to a disk controller in the PDP 11/44. This allows initial program load drive-0.

Each of the IOPs (1 through 31) consists of an IOCP bus interface, an I/O controller and a cache interface. The IOP bus interface has access to the PDP 11/44 unibus via the IOP command bus and an IOP controller interface. This extends the PDP 11/44 addressing capability to the IOP bus interface. All the I/O channels are converted to an IOP *snapshot* device in the cache memory control. This device scans the status of all 32 channels and can service a request to transfer

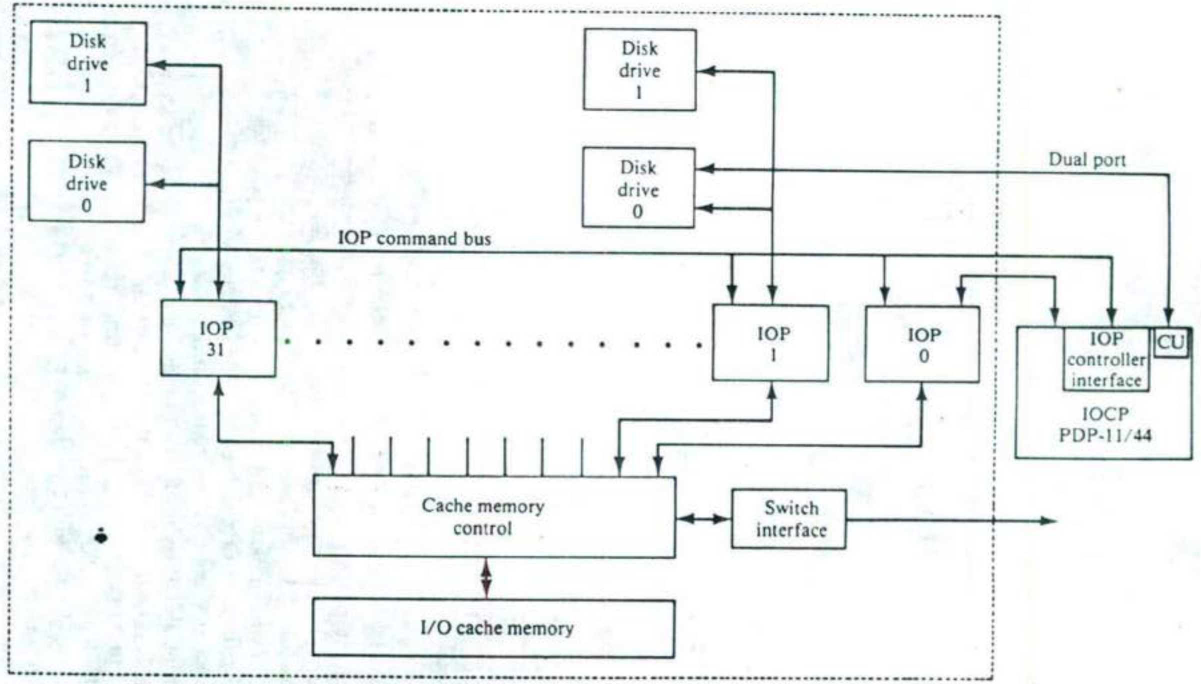
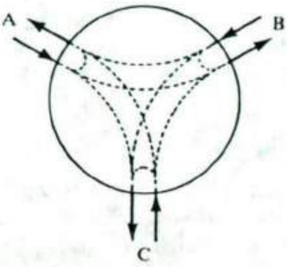


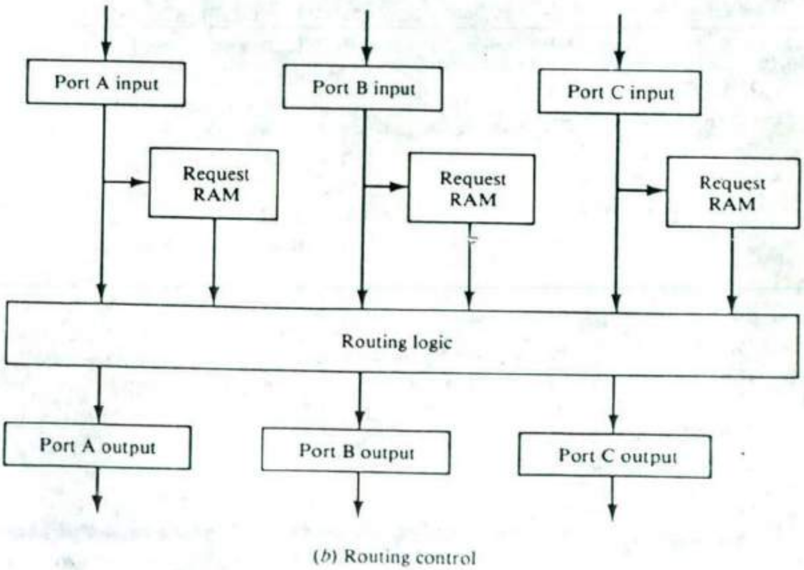
Figure 9.12 Functional components in the MSS of HEP. (Courtesy of Denelcor, Inc., 1982.)

a word for a channel every 100 ns. Cache memory request messages are received from the switch network through the switch interface. This interface is coupled to a switch node and can service a memory request from the switch every 100 ns.

The packet switching network The HEP switch is a synchronous, pipelined, packet-switched network consisting of an arbitrary number of nodes. Each node, which consists of three full duplex ports, is connected to its neighbors. These neighbors may be PEMs, DMMs, subsystems, or other nodes. Figure 9.13 depicts the HEP switching node. Each node receives three message packets on each of its three ports every 100 ns and attempts to route the messages in such a way that



(*a*) Bidirectional three-ported switch node



(*b*) Routing control

Figure 9.13 Switching node in the HEP's interconnection network. (Courtesy of Denelcor, Inc., 1982.)

the distance from each message to its addressed destination is reduced. This is accomplished by incorporating within each node three routing tables (one per port), which are loaded when the system is configured. Thus, each switch node is programmed to know the best output port routing to the final destination. Such programmed routing techniques allow for alternate routing to bypass a faulty component. In practice, the actual routing is determined by the best routing path and by priority in the case of conflicts. The priority is implemented by the use of age counters which increment with each nonoptimal routing.

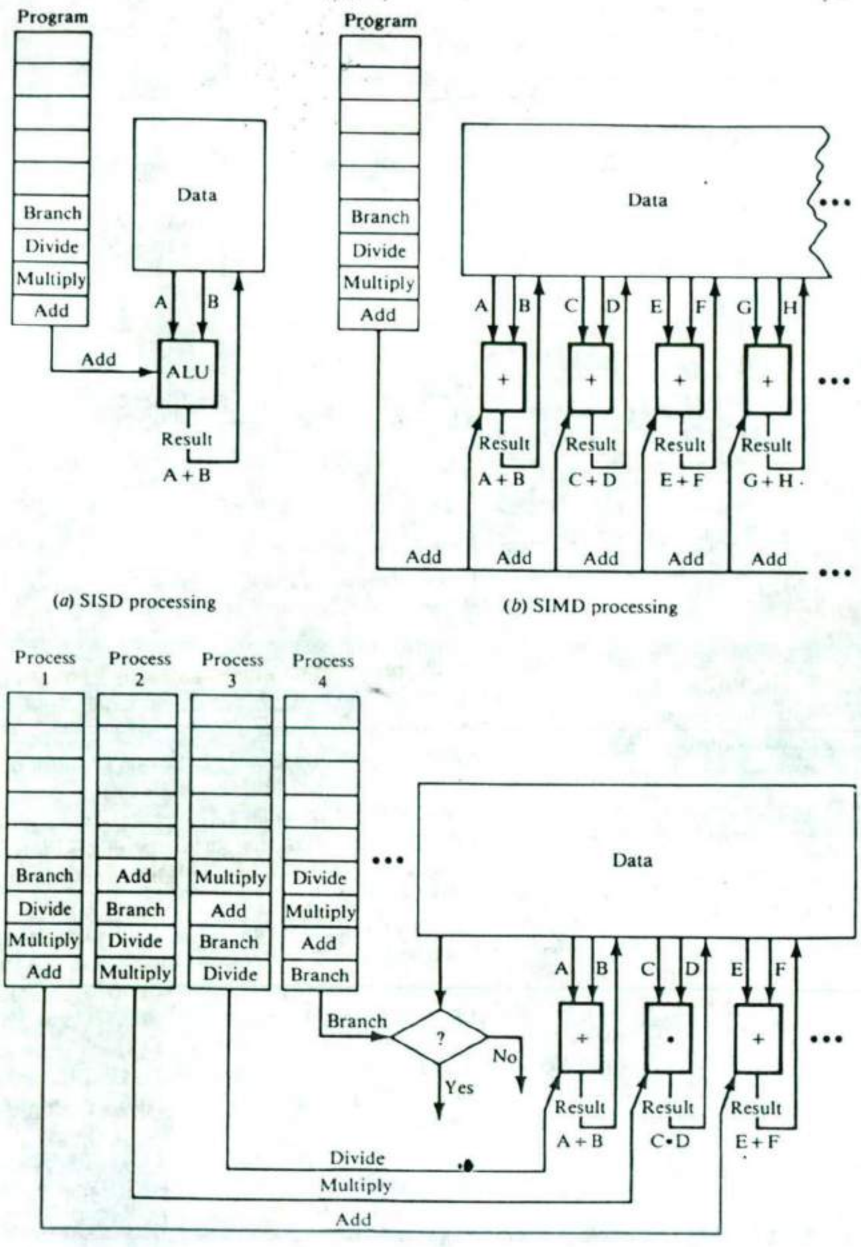
A unique feature of the switching network is that the switch nodes do not enqueue packets. These packets are routed by the switch nodes every 50 ns regardless of port contention. The modularity of the switching network permits field expandability. The increased memory access times that result from the greater physical distances between the PEMs and the DMMs can be compensated for in two ways. Each PEM contains a local memory large enough to buffer most of the program codes. Since each switch node is pipelined, each processor can execute a large number of instruction streams concurrently.

9.4.2 Process Execution Modules

The PEM is designed to execute multiple independent instruction streams on multiple data streams simultaneously. This is accomplished by pipelining each PEM with multiple functional units. Before presenting the organization of the PEM, we illustrate how the concurrency in execution of the MIMD streams is implemented. In the first case, consider a single instruction (SISD) stream which is being executed by a conventional (SISD) processor, as shown in Figure 9.14*a*. Very little overlap in execution can be achieved. Even with instruction-lookahead capability, the dependency constraints resulting from conditional branch instructions limited the concurrency significantly. So also are the SIMD processors in Figure 9.14*b*. These are vector-oriented computations. Because of the occurrence of conditional branch instructions, the performance may be degraded. The machine may have to wait for the total completion of the instruction before proceeding. That is, the conditional branches could not make use of the replicated hardware.

However, by providing multiple independent instruction streams executing multiple data streams in a pipelined execution environment, maximum parallelism can be achieved. For the example shown in Figure 9.14*c*, while an ADD is in progress for one process, a multiply may be executing for another, a divide for a third and a branch for a fourth. Because the multiple instructions executed concurrently by an MIMD machine are independent of each other, the execution of one instruction does not influence the execution of other instructions and full parallelism in processing may be achieved. Note, however, that a single process does not achieve any speedup in such a scheme as was accomplished in the IBM 360/91 system.

Each PEM consists of its own program memory and an instruction processing unit (IPU), as shown in Figure 9.15. The program memory in each PEM has a capacity ranging from 1 to 8 megabytes. Instructions of active processes which are



(*a*) SISD processing

(*b*) SIMD processing

(c) MIMD processing

Figure 9.14 Achieving maximal parallelism with replicated hardware in the HEP. (Courtesy of Denelcor, Inc., 1982.)

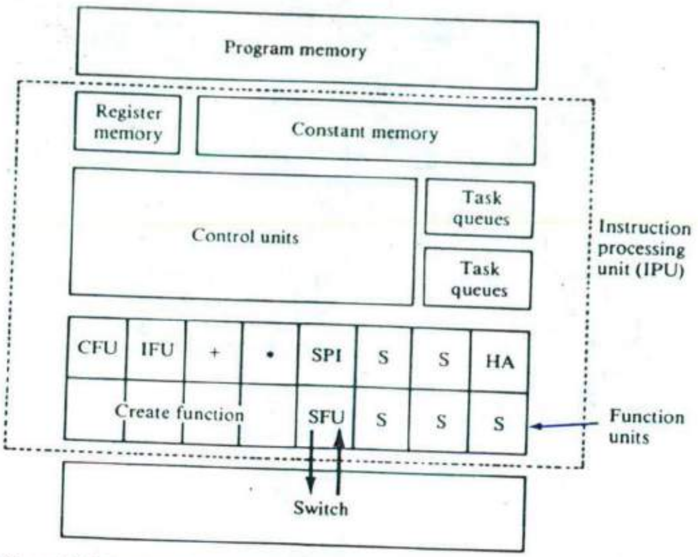


Figure 9.15 Functional description of HEP's process execution module. (Courtesy of Denelcor, Inc., 1982.)

allocated to a PEM are buffered in the program memory. These instructions are fetched from the program memory every 100 ns with concurrent decoding and execution of previously fetched instructions, as depicted in Figure 9.16. Up to 50 instructions may be in various stages of execution operating on one or more data streams simultaneously. However, the instruction fetch unit does not seem to permit simultaneity in instruction fetches and decodes. Also, there is only one instruction fetch unit in the PEM. For these reasons, the performance of the HEP processor may be limited to one instruction cycle per 100 ns. This may subsequently limit the effective utilization of the functional units.

The IPU in each PEM includes 2048 interchangeable general-purpose registers, as well as constant memory and function units. The constant memory is used to store user program constants and is read-only by user programs. The 4096 locations in the constant memory eliminate the need for data memory accesses for program constants. The function units implement the HEP instruction set, which includes extensions used to coordinate MIMD processing. In addition, the IPU has provisions for four expansion function units. These units may be used for custom or special-purpose instructions at the user's option.

In the HEP system, a set of cooperating processes constitute a *task*. Tasks and processes can be of two types: user or supervisor. The execution environment of a task is its *task domain*, which is defined by a 64-bit *task status word* (TSW). The TSW provides protection and relocation information for each task by a specification partition of the program, constant, register and data memories into areas.

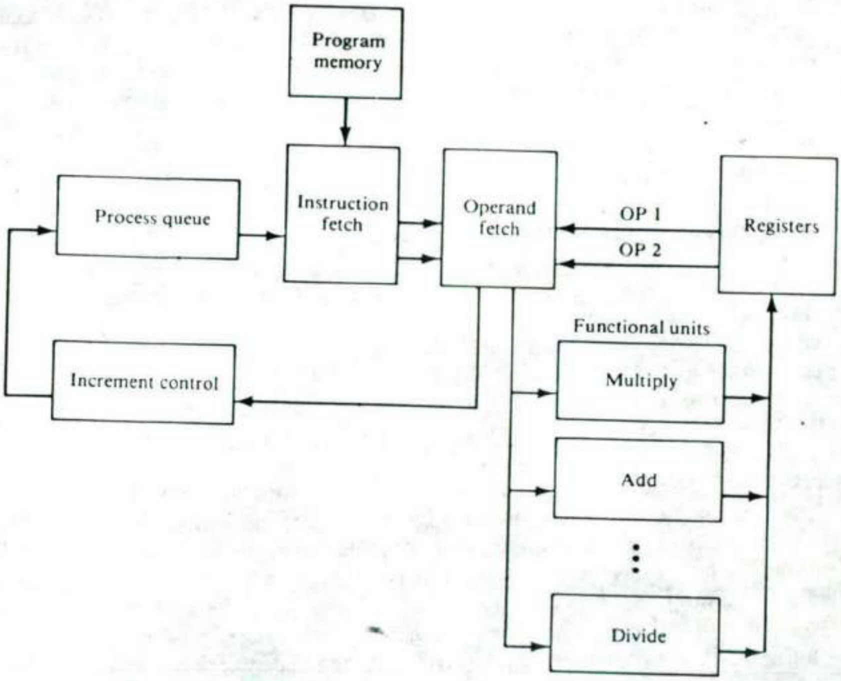


Figure 9.16 Instruction execution and data flow in the HEP.

This information is encoded as the program base, program limit, data base, data limit, constant base, register base, and register limit. All virtual addressing to operands is relative to the base addresses in the memory area in which they are stored. A *task status register* is used to hold the TSW for each task domain. A 16-entry task queue, where each entry contains a unique TSW, is used to implement a simple first-in first-out discipline in deciding which ready-to-run task to schedule. The task queue is equally divided for user and supervisor tasks.

In addition to the TSW, there is a *process status word* (PSW), which contains a 20-bit program counter and other state information for a HEP process. Each PSW points to an instruction that is ready for execution. There is a *process tag* (PT) in the task queue for each PSW that points to an instruction that is ready for execution. When a task is first initiated, it has only one PSW; that is, one process. The software creates additional PSWs as new processes are created to initiate parallel processing within a task. There is a PSW queue which can hold a total of 128 PSWs: 64 for user processes and 64 for supervisor processes.

These PSWs in the process queue circulate in a control loop which includes an incrementer and a pipeline delay. The delay is such that a particular PSW cannot circulate around the control loop any faster than data can circulate around the data loop consisting of general-purpose registers and the function units. As the program counter in a circulating PSW increments to point to successive instructions in program memory, the function units are able to complete each instruction in time to allow the next instruction for that PSW to be influenced by its effects. The control and data loops are pipelined in eight 100-ns segments, so that as long as at least eight PSWs are in the control loop, the processor executes 10 MIPS. However, a particular process cannot execute faster than 1.25 MIPS, and will execute at a lesser rate if more than eight PSWs are in the control loop.

The instruction issuing operation maintains a fair allocation of resources between tasks first and between processes within a task second. The main schedule contains 16 task queues, each containing up to 64 PTs. A secondary queue called the snapshot queue records the head PT in each task queue each time the snapshot queue becomes empty. PTs arriving one at a time from the snapshot queue cause the issuing of an instruction from the corresponding process into the execution pipeline.

A control unit cooperates with the function units to execute instructions in the IPU. The control unit selects an instruction for execution from one of the task queues, fetches the instruction, addresses the operands, and passes the instruction operation code and the operands to one of the function units to perform the specified operation. There are two types of function units: *synchronous* and *asynchronous*. The synchronous function units are pipelined with eight linear segments and a segment time of 100 ns. Thus, instructions are completed in 800 ns. Examples of synchronous function units are the floating-point adder (+), the multiplier function (*), the integer function unit (IFU), the create function unit (CFU), the hardware access (HA) unit, and the system performance instrument (SPI).

The CFU performs all operations affecting the PSWs. This includes activating and terminating processes, incrementing the program counter in a PSW that has had an instruction executed, and executing branch and supervisor call instructions. The HA executes all instructions to read or write program memory and performs bit encode and decode operations. The SPI collects data for performance measurement and monitoring counters and tracks the number of instructions executed by tasks. This allows billing the user for the amount of work done regardless of the time required because of overheads.

Asynchronous function units do not necessarily complete their operations within 800 ns. Examples of such function units are the divider (÷) and the scheduler (SFU). The divide function unit consists of up to eight individual divider modules which asynchronously execute 64-bit floating-point divide instructions. Divide instructions are initiated at a rate of one every 100 ns until all divider modules are busy. Each module can execute a divide instruction every 1700 ns. This is the only function unit in the IPU that is not pipelined. The SFU is both synchronous and asynchronous, and executes all instructions involving data transfers between register memory and data memory. Transfers that pass through the switch are executed asynchronously; all others are executed synchronously. The SFU can accept a new instruction every 100 ns.

When a data-transfer instruction is executed, the SFU sends a switch message packet containing a 32-bit data memory address, a return address identifying both processor and process, and 64 bits of data if a store instruction was executed. The SFU also removes the process that executed the instruction from the control loop and does not reinsert it until a response packet is received from the switch. When that response packet arrives, the SFU writes the data portion of the response in the appropriate register if a load instruction was executed. In order to perform these functions, the SFU is equipped with a queue similar to the queue in the control loop of the processor proper, and a process migrates freely between these two queues as it initiates and completes data-memory reference instructions. An important consequence is that while processes are in the SFU queue they are not present in the control loop queue and thus do not consume the valuable computational cycles while memory operation references are in progress.

The waiting period experienced by a process which encountered a conflicting access to memory or a busy resource does not cause the IPU to remain idle. In fact, the IPU switches context every 100-ns cycle. The rapid context switching is facilitated by its hardware implementation. Recall that the IPU has a program status register and 40 general-purpose registers for each user process. These register sets eliminate the need to save and restore program context when switching between users. Thus, the IPU multiplexes the execution of instructions from up to 128 instruction streams. This is done because the initiation of an instruction execution must often wait for the results from the instruction immediately preceding it. The IPU uses this waiting time to initiate instructions from other active instruction streams. From 8 to 12 processes are usually sufficient for the IPU to use all instruction cycles and thus achieve the 10 MIPS execution rate.

Protection mechanism Protection of one user process from another is accomplished in HEP by various techniques. Nonprivileged processes in separate task domains are prohibited from reading and writing the other's general-purpose registers and data memory by hardware address checking. All addresses are bounds-checked against values in the TSW and access denied or permitted accordingly. Nonprivileged processes are also prohibited from directly initiating I/O and from modifying program memory, constant memory, the task and process queues. A user process cannot change the environments of other user processes unless there is controlled cooperation and their task domains overlap.

Synchronization mechanism Cooperating processes synchronize by means of accesses to shared data. In HEP, this facility is provided by associating an access state with each register memory and data memory location. In data memory, the access states are **full** and **empty**; a load instruction can be made to wait until the addressed location is **full** and indivisibly (i.e., without allowing an intervening reference to the location) set the location **empty**. Similarly, a store instruction can wait for **empty** and then set **full** at any location in data memory. In register memory, an instruction can require that both sources be **full** and the destination **empty**, and then set both sources **empty** and the destination **full**. To ensure the indivisibility of this kind of operation, the third access state **reserved** is implemented in the registers. The destination register is set **reserved** when the source data is sent to the function units, and only when the function unit stores the result is the destination set **full**. No instruction can successfully execute if any of the registers it uses is **reserved**.

A process failing to execute an instruction because of an improper register access state is merely reinserted in the queue with an unincremented program counter so that it will reattempt the instruction on its next turn for execution. A program executing a load or store instruction that fails because of an improper data-memory access state is reinserted in the SFU queue and generates a new switch message on its next attempt.

9.4.3 Parallel Processing on the HEP

Extensions were made to Fortran 77 in HEP to provide language support for parallel processing. A special data type called the *asynchronous variable* was introduced to enable synchronization between cooperating and competing processes. The asynchronous variable type uses the access-state capability of the HEP hardware to support the correct interaction between processes. An asynchronous variable is identified with a $ before the variable name. Such a variable may be written into only when its location is **empty** and may only be fetched when it is **full**. Either operation on an asynchronous variable that does not meet these requirements waits under hardware control until the proper access state is set by a parallel process.

The access state of each asynchronous variable is initially set **empty** or **full** by the program. The HEP Fortran 77 provides two statements and five specially designed intrinsic functions for manipulating and testing access states.

Single statements

```
A = $Q     //Wait for full, read and set empty//
$Q = A     //Wait for empty, write and set full//
```

Intrinsic functions

```
A = VALUE($Q)   //Access the value, regardless of state//
A = SETE($Q)    //Read regardless of state and set empty//

A = WAITF($Q)   //Wait for full, but do not set empty//
L = FULL($Q)    //Test for full access state and return logical result//
L = EMPTY($Q)   //Test for empty access state and return logical result//
```

A PURGE statement is used to unconditionally set the access state to **empty**.

Another class of extensions were made to Fortran 77 to allow parallel process creation (similar to FORK) and termination (JOIN). The first statement, called CREATE, is syntactically similar to a Fortran CALL, but it causes the created subroutine to run in parallel with its creator. Another statement, called RESUME, is syntactically like a RETURN from a subroutine. However, it causes the caller of a subroutine to resume execution in parallel with the subroutine. If a subroutine was CREATEd, a RESUME has no effect. On the other hand, a RETURN causes the termination of the process if it was CREATEd or if it previously executed a RESUME.

HEP Fortran generates fully reentrant code and dynamically allocates registers and local variables in data memory as required by the program. Hence, it is easy to create several processes which simultaneously execute identical programs on different data. This can be accomplished by placing a CREATE statement in a loop so that several parallel processes will execute identical programs on different local data. An example of the implementation of parallel operations in the HEP is given below.

Example 9.1

```
      PURGE $IP, $NP
      $NP = NPROCS
      DO 10 I = 2, NPROCS
      $IP = I - 1
      CREATE S($IP,$NP)
   10 CONTINUE
      $IP = NPROCS
      CALL S($IP,$NP)
C     WAIT FOR ALL PROCESSES TO FINISH
   20 N = $NP
      $NP = N
      :
      IF (N .NE. 0) GO TO 20
      SUBROUTINE S($ IP,$NP)
      MYNUM = $IP
      :
      $NP = $NP - 1
      RETURN
      END
      :
```

In this example, the program creates NPROCS-1 processes all executing subroutine S, and then itself executes the subroutine S by calling it, with the result that NPROCS p.ocesses are ultimately executing S. The parameter $IP is used here to identify each process uniquely. Since parameter addresses rather than values are passed, $IP is asynchronous and is filled by the creating program and emptied within S. This prevents the creating program from changing the value of $IP until S has made a copy of it. The asynchronous variable $NP is used to record the number of processes executing S. When S is finished, $NP is decremented, and when the creating program discovers that $NP has reached zero, all NPROCS processes have completed execution of S (excepting possibly the RETURN statement).

The following example considers converting a serial code into a parallel program in order to speed up the execution on the HEP.

```
      DIMENSION A(270), B(270), C(270), D(270)
      :
      N = 270
      E = 0.0
      DO 100 I = 1, N
      A(I) = A(I)**SIN(B(I))
      IF (SIN(A(I)).GT.COS(C(I))) GO TO 10
      A(I) = A(I) + C(I)
      GO TO 20
 10   A(I) = A(I) - D(I)
 20   E = E + A(I)**2
 100  CONTINUE
      :
```

(*a*) Serial code.

```
      COMMON A(270), B(270), C(270), D(270), $E
      :
      L = 270
      N = 20
      PURGE $E, $IN, $IW
      $E = 0.0
      $IN = 1
      $IW = 0
      DO 100 I = 1, N
      CREATE DOALL ($IN, $IW, L)
 100  CONTINUE
 200  IF (VALUE ($IW).L.T.L) GO TO 200
      :
      SUBROUTINE DOALL ($IN, $IW, L)
      COMMON A(270), B(270), C(270), D(270), $E
 1    I = $IN
      $IN = I + 1
      IF (I.GT.L) GO TO 30
      A(I) = A(I)**SIN(B(I))
      IF (SIN(A(I)).GT.COS(C(I))) GO TO 10
      A(I) = A(I) + C(I)
      GO TO 20
 10   A(I) = A(I) - D(I)
 20   $E = $E + A(I)**2
      $IW = $IW + 1
      GO TO 1
 30   RETURN
```

(*b*) Parallel version.

Figure 9.17 Algorithm restructuring example in using HEP for parallel processing.

Example 9.2 The serial code shown in Figure 9.17*a* manipulates the linear array *A* with 270 elements and accumulates the square of the components of A in a variable E. The asynchronous variables $E, $IN and $IW are introduced in the parallel version of the program shown in Figure 9.17*b* $E is used to mutually exclusively accumulate the square of A(*i*)'s when they are updated. The parallel version creates 20 processes so that the granularity is significant enough to have nontrivial processes. $IN is used to control accesses to unique components of A and $IN is used to terminate the parallelism when all components of A have been updated and $E computed.

Another common technique allows processes to schedule themselves. In the simplest case, a number of totally independent computational steps are to be performed that significantly exceeds the number of processes available; moreover, the execution time of the steps may be widely varying. Self-scheduling allows each process to acquire the next computational step dynamically when it finishes the previous one.

Example 9.3 The following example has a subroutine T which is to be executed 400 times. All iterations are assumed to be independent; i.e., none of the iterations uses the output of any of the others. Four processes are initiated to do the processing. The variable $START is used to keep track of the iteration count and to assure that only 400 iterations are started. $DONE is used to assure that all 400 iterations have completed before the main program continues. Statement 5 in the main program will cause the main program to non-busy wait (i.e., consume no "CPU" cycles) for the 400th iteration to complete and $ALLDONE to be set full by statement 99:

```
       PURGE $START, $DONE, $ALLDONE
       $START = 0
       $DONE = 0
       CREATE T ($START, $DONE, $ALLDONE)
       CREATE T ($START, $DONE, $ALLDONE)
       CREATE T ($START, $DONE, $ALLDONE)
       CALL T ($START, $DONE, $ALLDONE)
    5  DUMMY = $ALLDONE
       :
       END
       SUBROUTINE T ($START, $DONE, $ALLDONE)
   10  1 = 1 + 1
       IF (I.G.E.400) GO TO 99
       :
       $DONE = $DONE + 1
       GO TO 10
   99  IF (VALUE ($DONE).EQ.400) $ALLDONE = 1
       RETURN
       END
```

Self-scheduling is an excellent technique for programs which have steps with widely varying execution times because it balances the work load among available processes. There are many other techniques which have been used to exploit the HEP's parallel architecture.

There are many applications of the HEP machine. It has been suggested to handle the traditional multiprogramming of SISD programs. The application for which HEP was originally designed was the solution of large-scale systems of differential equations, such as those describing flight dynamics problems. Another problem for which the HEP is suitable is the partial differential equations describing continuous meshes. An application area for which the HEP would have a tremendous potential is in the simulation of a discrete event system or process-driven simulation. However, the application areas for HEP are not limited to the above. The architecture is quite flexible for a wide variety of applications.

9.5 MAINFRAME MULTIPROCESSOR SYSTEMS

This section describes some commerical multiprocessor mainframe systems. The systems to be described do not achieve their performance by decomposing a user SISD algorithm into MIMD processes. Instead, the multiprocessor operating systems achieve concurrency through explicit parallelism. The system performance is achieved by mostly concurrent execution of independent and noninteracting user processes. Most commercial multiprocessors offer fairly loose coupling. Of course, the degree of coupling varies from system to system.

9.5.1 IBM 370/168MP, 3033, and 3081

The architecture of the IBM System/370 is extended from the IBM System/360. A series of models were developed in the System/370. Different models in the System/370 represent different performance levels. Most models of the System/370 are SISD machines with a uniprocessor. Multiprocessing is only an added-on feature of the series for the top of line in performance. In 1966, the IBM S/360 Model 67 was introduced as a dual-processor time-sharing system. The S/360 Model 65 MP is a dual-processor version of the standard Model 65. In 1974, the IBM S/370 Models 158 MP and 168 MP were introduced as dual-processor systems with shared real and virtual shortage. In 1976, IBM introduced the S/370 Models 158 AP and 168 AP as asymmetric multiprocessors. The code MP stands for *multiprocessing* and AP for *attached processing*. In this section, we describe the architectural evolution of the IBM 370/168 for both MP and AP. Processor and operating system features for various 370/168 multiprocessor configurations are then reviewed with emphasis on their capabilities. We will also compare the 370/168 MP with the enhanced IBM 3033 and 3081 multiprocessor systems.

A uniprocessor IBM 370/168 configuration is shown in Figure 9.18. The main memory is divided into four logical storage units (LSU), which form a four-way interleaved memory system. The memory access and conflict resolution is con-

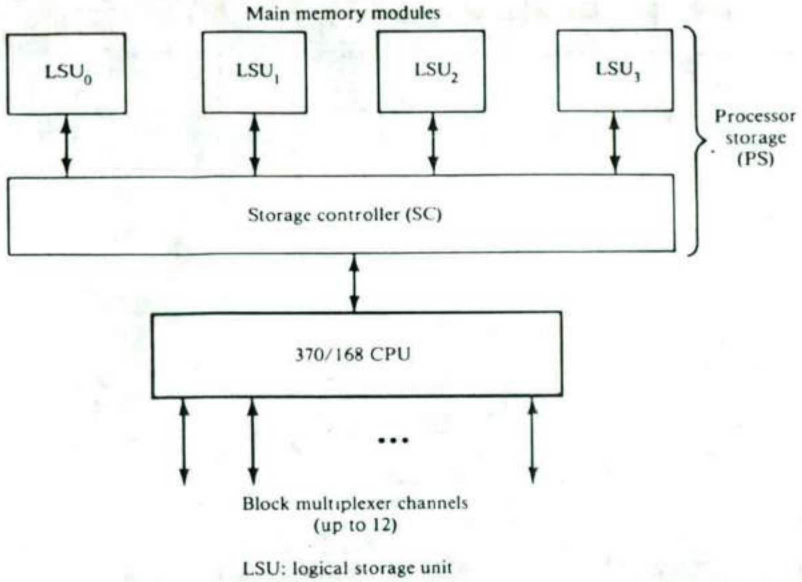


Figure 9.18 An IBM 370/168 uniprocessor configuration. (Courtesy of International Business Machines Corp., 1978.)

trolled by the storage controller. There is only one central processing unit (CPU) which contains the pipelined instruction decode and execution units together with a fast cache. Multiple I/O channels can be connected to the CPU. Each channel is a specialized I/O processor with a simple I/O instruction set and operates asynchronously with the CPU. An AP configuration for the IBM 370/168 is illustrated in Figure 9.19. It is extended from the uniprocessor configuration by attaching an attached processing unit (APU).

The APU is almost identical to the standard 370/168 CPU except no I/O channels can be attached to it. Both the CPU and APU have their own caches. The cache coherence problem is resolved by using the cache invalidate (CI) lines between the CPU and APU. The interprocessor communication (IPC) lines are used for exchanging information or interrupt signals directed between the two processors. The structure is considered asymmetric because the APU is devoted exclusively to computation and the CPU handles both internal computation and I/O communications. The multisystem control unit (MCU) performs the interconnection switching functions between the processors and the shared memory modules.

An MP configuration of the IBM 370/168 is shown in Figure 9.20. Instead of attaching an APU, another CPU is used to form a symmetric dual-processor system. This MP configuration is composed of two 370/168 uniprocessor systems with shared memories. The two CPUs have equal capabilities. Two sets of I/O

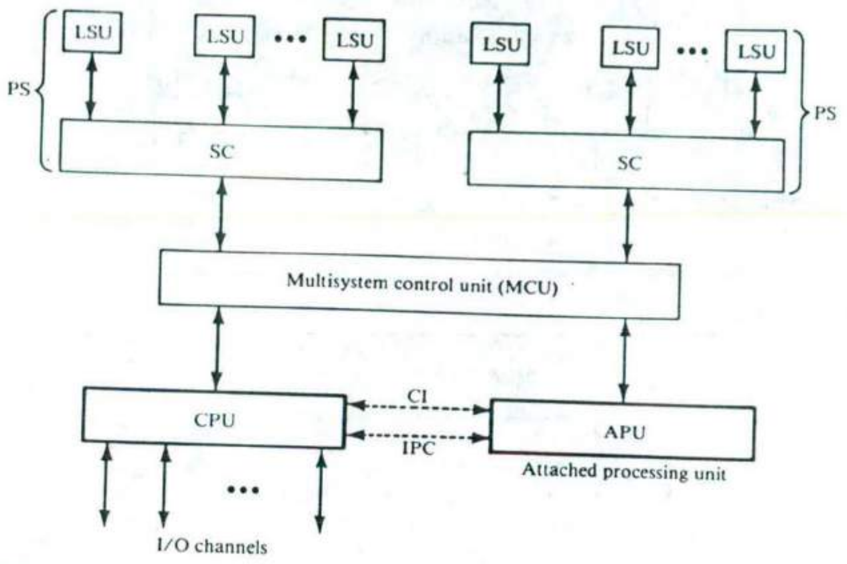


Figure 9.19 An IBM 370/168 AP configuration. (Courtesy of International Business Machines Corp., 1978.)

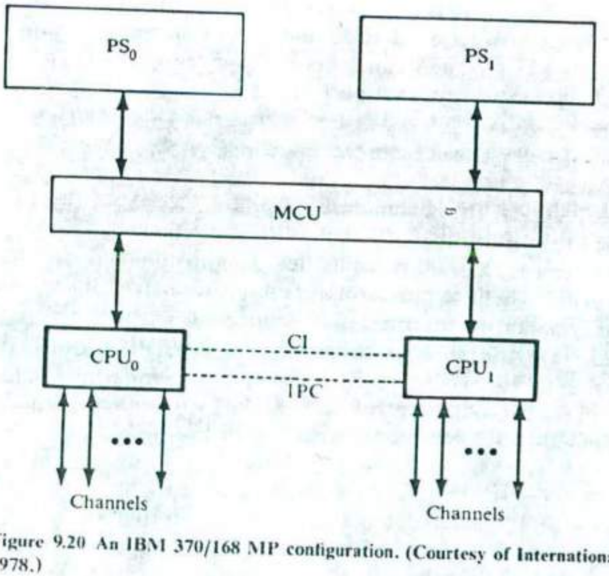


Figure 9.20 An IBM 370/168 MP configuration. (Courtesy of International Business Machines Corp., 1978.)

channels attached to each CPU are mutually exclusive and cannot communicate with each other directly. The MCU provides the necessary interconnection hardware between the two CPUs and shared memories. It also contains a configuration control panel for the purpose of manual systems reconfiguration.

In the IBM 370/158 MP configuration, two processors share from 1 to 8 million bytes of main storage. Each processor has a separate 8K-byte cache with 230-ns access time of 8 bytes. The two processors in the 370/168 MP share from 2 to 16 million bytes of main storage. Each CPU has either an 8K-byte or 16K-byte cache with a reduced 80-ns access time of 8 bytes. The model 158 has 10 block multiplexor channels, while the model 168 can have up to 22 block multiplexor channels. The block multiplexor channels permit concurrent processing of multiple channel programs for various speed peripheral devices, as was illustrated in Section 2.5. The Model 168 is enhanced from the Model 158 mainly in the area of memory and I/O subsystems. Their CPUs essentially have the same capabilities.

The 370/168 MP configuration is considered loosely coupled because two separate copies of operating systems are running in the two CPUs. An IBM 370/168 uniprocessor system can also be reconfigured to a tightly coupled multiprocessing system of dual CPUs with shared memories and shared I/O devices, as shown in Figure 9.21. The two CPUs are tightly coupled by a single copy of the operating system in the shared memory. A tightly coupled CPU pair can also be loosely coupled with another uniprocessor CPU to form a mixture multiprocessor system, as demonstrated in Figure 9.22. This is really a tightly coupled multiprocessor in a loosely coupled configuration. The tightly coupled dual CPU and the uniprocessor share some direct-access devices, such as disk, and some tape units. A channel-to-channel adaptor can be used to link the two CPU modules. Each CPU module still has some private channels connecting to some private I/O and secondary devices, like the 3330 disk storage subsystems.

The IBM 3033 system The 3033 multiprocessor complex consists of two 3033 Model M processors, two 3036 consoles, and the 3038 multiprocessor communications unit (MCU). Figure 9.23 shows a conceptual relationship between the MCU and processor functions. The 3033 attached processor complex consists of a 3033 Model A processor, the 3042 attached processor, two 3036 consoles, and the 3038 MCU.

The MCU for the multiprocessor-attached processor models provides prefixing, interprocessor communication, cache (high-speed buffer) and storage update communication, sharing of processor storage, configuration-partitioning control, synchronization facilities, and communication of changes to the storage protection keys.

The MCU also enables both processors in an MP configuration to access all of processor storage while retaining the overlap capability in storage operations permitted by eight-way interleaving. This means that both processors can have concurrent storage operations in progress with a varying degree of overlap depending upon the particular sequence of LSU accesses. The configuration and partitioning control in an MP system provides a variety of storage configuration options

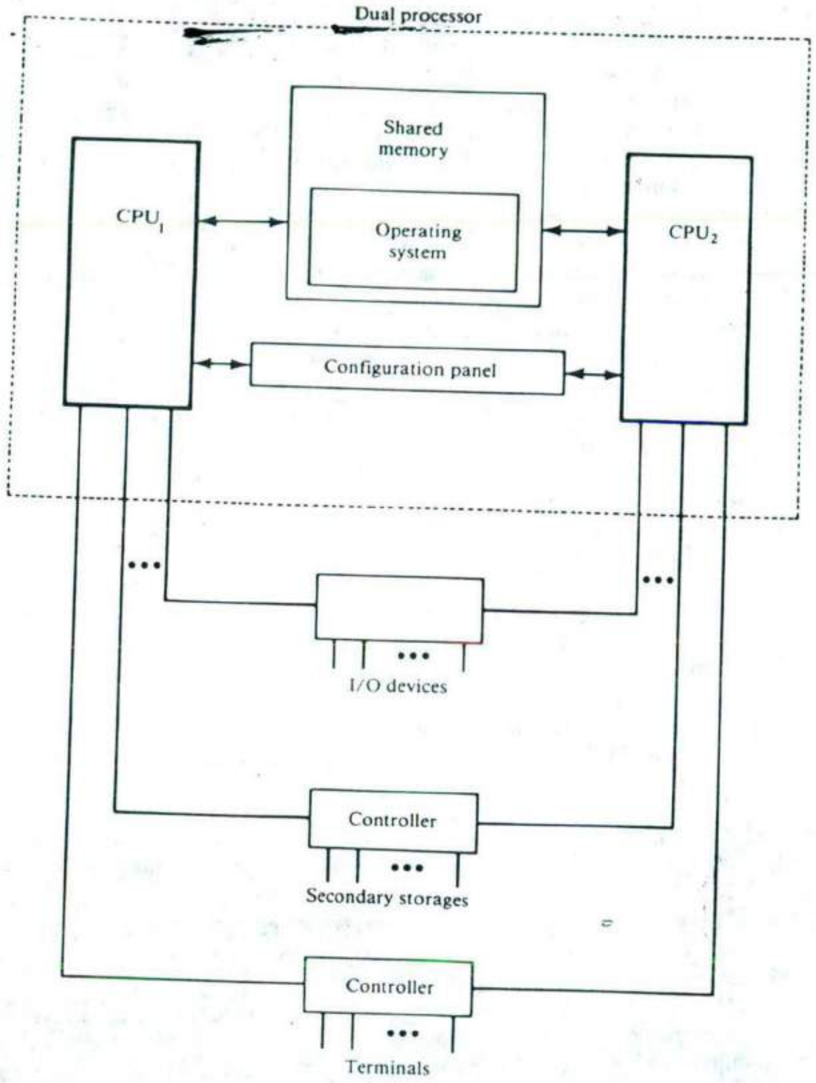


Figure 9.21 A tightly coupled IBM dual processor system with a single copy of operating system residing in the shared memory.

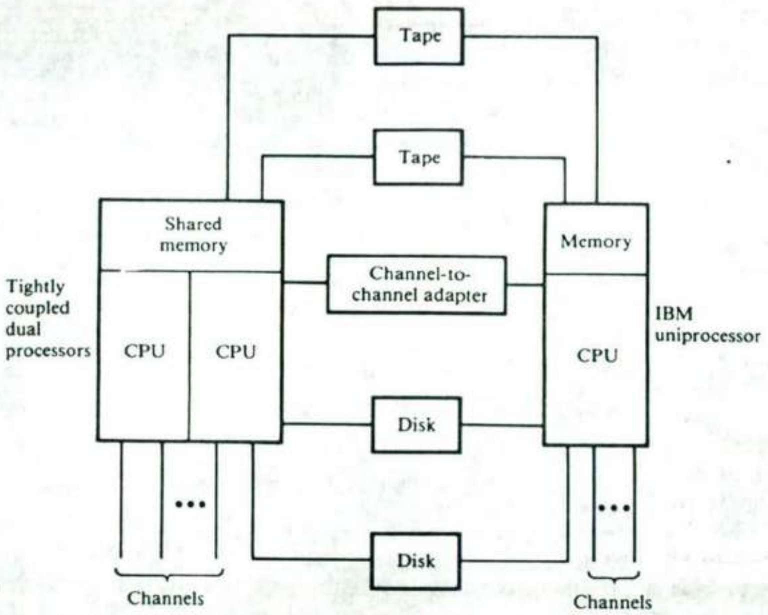


Figure 9.22 A tightly coupled IBM dual-processor system loosely coupled with an IBM uniprocessor.

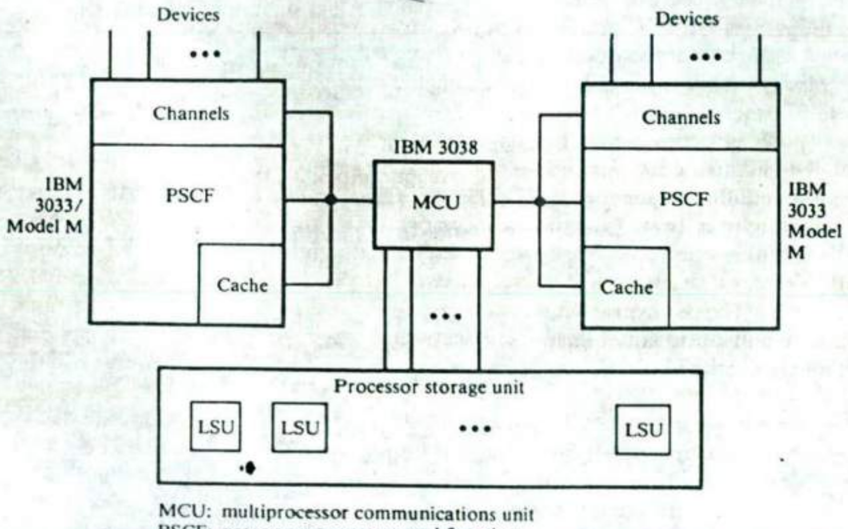


Figure 9.23 The IBM 3033 multiprocessor complex. (Courtesy of International Business Machines Corp., 1978.)

which can apportion the storage either independently to each processor for uniprocessor mode or shared between the two processors for multiprocessor mode. The 3033 MP complex achieves its high performance through higher interprocessor communication, better cache and storage protection and faster access to shared processor storage than the 370/168 MP.

The 3033 MP contains another improvement over the 168 MP and AP, in that the priority bit mechanism has two levels. Low-level bits compete with low-level bits and high-level bits compete with high-level bits but, as might be suspected from the names, a high-level bit can preempt a low-level bit. The 3033 with its enhanced buffering can, at times, sustain an exceptionally long string of storage requests, so this facility prevents a high-priority request on one processor from being unnecessarily delayed by low-priority activity on the other. The 3033 MP performance has been stated to be 1.6 to 1.8 times that of the 3033 uniprocessor system, based on simulation results and running experience.

The IBM 3081 system The IBM 3081 processor unit has a symmetric organization of two central processors, each with a 26-ns machine cycle time, and executes IBM System/370 instructions at approximately twice the rate of the IBM 3033. One of the goals set in the design of the 3081 was better price/performance index over the System/370 and 303x series. The other goal was upward compatibility with those product lines. Furthermore, it was designed to have improved reliability, availability, and serviceability through new technology and partitioning and packaging schemes. One of the major achievements in packaging is the development of a field replaceable unit called a thermal conduction module (TCM), which contains up to 130 IC chips on one substrate. With these TCMs, a large board called a clark board was developed. Each board contains up to either six or nine TCMs, which made it possible to package the entire processor unit on four boards in one frame.

The 3081 processor unit organization shown in Figure 9.24 consists of five subsystems: two central processors, the system controller, the main storage, and the external data controller (EXDC). The 3081 is called a *dyadic processor* since it is configured as two identical processors that share a system controller and EXDC within one frame. Furthermore, they act as a tightly coupled multiprocessor which cannot be decoupled to act as two independent uniprocessors as in the IBM 3033. The configuration is symmetrical because each processor has the same priority and operational characteristics with respect to the central storage and channels. Each processor has access to channels and to central storage via the controller.

The processors share main storage, which could be 16, 24, or 32 megabytes. A segment of main memory called the system area is reserved for microcode. This area also contains unit control words (UCWs) for I/O devices and system tables and directories. Hence, the system area is not accessible to user programs. The main storage is organized as a card-on-board package and is two-way interleaved. Each board, which contains 4M bytes of main memory, is called the basic storage module. This module is configured so that a block (128 bytes) of data can be

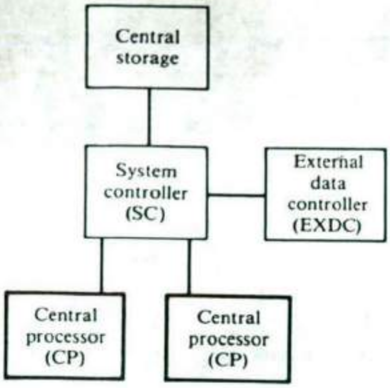


Figure 9.24 The IBM 3081 system components. (Courtesy of International Business Machines Corp., 1980.)

accessed with a single operation to efficiently transfer a block between the processor and memory. The interleaving scheme uses doubleword, which is the basic unit of memory operation, and a 2K-byte address across the 4M-byte modules. The 2K byte segment, which was chosen to minimize the complexity of memory reconfiguration, can thus be independently accessed.

The system controller provides the paths and controls the communications between the main memory and other subsystems. The basic data-bus width of all units connecting to the controller is 8 bytes with a data transfer rate of 8 bytes per machine cycle. The bus is bidirectional. The controller also contains the storage protect keys and time-of-day clock and manages an eight-position queue containing storage requests. The 3081 can support up to 24 channels, which can be of either the byte-multiplexer or block-multiplexer type. Channels are controlled by the EXDC. The EXDC consists of two types of microcode-controlled elements. One of the elements handles the control of I/O instructions and interrupts. The other handles the data control sequencing and provides buffering for each group of eight channels.

Each processor consists of five functional elements, as shown in Figure 9.25, and packaged within a nine-TCM board. The processor is not a pipelined processor as in the IBM System 370/168. However, it has an effective instruction prefetching capability. Each processor has three separate execution elements, a buffer control element, and a control store element. The execution elements are the instruction elements, variable-field element, and execution element. The instruction element controls the instruction sequencing of a processor by initiating requests for instructions and attempting to maintain an instruction buffer of four doublewords locally. It performs the instruction-decode and operand-address generation functions and initiates all requests for operands. It also executes all arithmetic and logical operations.

The variable-field element operates under horizontal microcode control, executes all variable length, storage-to-storage instructions. Within it is a decimal adder and its associated input and output regions. In executing the specified set

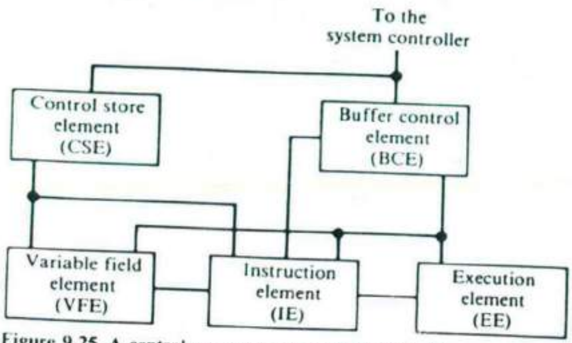

To the
system controller
Control store element (CSE)
Buffer control element (BCE)
Variable field element (VFE)
Instruction element (IE)
Execution element (EE)

Figure 9.25 A central processor of the IBM 3081. (Courtesy of International Business Machine Corp., 1980.)

of instructions, it operates together with the instruction element, which performs operand fetches and stores in parallel. The execution element executes fixed-point multiply and divide instructions. It also executes conversion-type instructions (e.g. binary to decimal) and floating-point instructions. The control store element controls the sequencing of the horizontal microcode of the processor.

The buffer control element contains the 32K- or 64K-byte cache, which uses a write-back main memory update policy. The block size is 128 bytes and the cache is four-way set-associative with a least-recently-used replacement algorithm. The cache has a two-cycle access with the virtual-to-real address translation performed in parallel. The write-back policy used in the 3081 is different from the write-through policy used in the 3033. Simulation studies showed that the write-back policy was better on the 3081 as soon as the memory-access time exceeded about ten machine cycles. Also, the write-back policy facilitated a checkpoint-retry algorithm for the processor. The checkpoint-retry mechanism is a hardware device which can be used to establish a checkpoint at some instruction N. At this instruction, the contents of the processor registers are saved in a backup register set. As program execution continues, any store into the cache will cause the old value of a changed doubleword to be saved in a push-down stack. If an error condition occurs, the processor registers can be restored to their original state at the last checkpoint and the process can be restarted.

Since each processor has a cache, the write-back policy adopted creates a data consistency problem which is managed by the system controller. The controller implements the global table similar to Section 7.3. This table is used to enforce consistency. The dynamic coherence scheme is used in the 3081. The difference is that when a processor requests a fetch of block which is held exclusive and modified in remote cache, the copy of the block is updated in memory. Furthermore, the copy in the remote cache is invalidated instead of changing state to read-only. Thereafter, the local processor fetches the block from memory. An algorithm which predicts the usage of a block of data was developed to minimize the performance overhead of status changes and invalidation. A four-processor system (the IBM 3084) can be configured by a set of two 3081s in either MP or AP mode.

9.5.2 Operating System for IBM Multiprocessors

All IBM 360 and 370 models have a 32-bit word divided into 4 bytes that are byte addressable for business-oriented (character strings) applications. The System/370 instruction set includes fixed-point and floating-point arithmetic operations, character manipulation, binary as well as decimal arithmetic computations plus a number of system control instructions. There are sixteen general-purpose registers, four floating-point registers, and sixteen control registers in the CPU. The control registers are mainly used for executing the system control sequence in the operating system. There is also a 64-bit program status word (PSW) register. This PSW is primarily used by user programs in indicating the status of interrupts, overflow or underflow, and even functions as an extra program counter.

In order to facilitate multiprocessing, the IBM 370/168 has the following special features in addition to those uniprocessor models in the System/370 series. Each CPU use a block of 4K words to hold key status and control sequences. Prefixing capability is provided to logically assign this block to each CPU in a separate physical block in the shared memory. A time-of-day feature is built into the system to synchronize the clocks of two CPUs in MP mode. Interprocessor communication mechanisms are provided to signal a CPU or to respond to the signal sent by another CPU. There are five different interrupt classes in S/370 including program interrupts, I/O interrupts, external interrupts, machine-check interrupts, and service-call interrupts to allow communications among user and system programs. The PSW register is used to facilitate the interrupt services.

Instructions related to I/O operations are treated as privileged instructions. The execution of these privileged instructions is controlled by the bit pattern in the PSW register. Memory protection has been mechanized with the aid of the PSW register. Other interesting functions of the 370/168 processors include dynamic address translation to support the virtual storage operating system. The virtual address space can cover up to 16 million bytes for any physical main memory space from 2 million bytes to 16 million bytes in 1 million-byte increments. Error checking and error correction capabilities are also built into the main memory. Instruction retry and I/O command retry capabilities are also provided to reduce the number of system failures.

The degree of coupling between two processors in a multiprocessor system is determined by the capabilities of the operating system. The IBM 370/168 runs with an Operating System/Virtual Storage 2 (OS/VS2), which is revised from the standard S/370 operating system. This OS/VS2 is a time-sharing multiprocessing system in a virtual storage environment supporting either a tightly coupled system or a loosely coupled configuration. Two important features of multiprocessing in various 370/168 configurations are the separation of architectural configurations

(AP, MP or others) from the implementation of system control mechanisms and the sophistication of the interprocessor communication mechanisms.

For a tightly coupled dual-processor 370/168 MP system (Figure 9.21), five important features have been developed in the OS/VS2 to support the multiprocessing (MP) hardware. These features are listed below:

1. A serialization technique called locking is provided to disable across CPUs. It allows mutually exclusive functions to run in parallel on an MP system.
2. The service management provides a new unit of dispatchability in the system which has less overhead and better performance in encouraging parallelism in system functions.
3. The CPU affinity provides a way for forcing an emulatory job step to a particular CPU having the desired hardware feature.
4. The dispatching supports the changes in MP functions and in multiple address spaces.
5. The alternate CPU recovery (ACR) invokes a special purpose when a CPU is detected malfunctioning.

Various 370/168 MP configurations have been installed with special hardware-software error-recovery mechanisms. When an error is detected on a processor, an interrupt service is generated to recover that processor, if possible. If the interrupt handler realizes that recovery is impossible, an SIGP instruction is issued to specify an emergency alert as the order code. The OS/VS2 uses a linear ordering scheme to avoid system deadlock. The error-recovery software will incrementally execute all tasks holding locks until the deadlock crisis is avoided. Reconfiguration of the 370/168 system is done through operator intervention. An MP system can be divided into two independent uniprocessor systems. The APU in an AP configuration can be disabled without paralyzing the whole system. However, a failure of the CPU in an AP configuration will result in the failure of the entire system.

9.5.3 Univac 1100/80 and 1100/90 Series

Large-scale mainframe computer systems in the Univac 1100 series are described in this section. Beginning with the 1107 in 1962, the 1100 series has progressed through a succession of compatible computer models to the latest 1100/80 and 1100/90. We first review the architectural evolution in the series and then discuss the details of the models 1100/80 and 1100/90 and their operating system and language requirements.

Architectural evolution of Univac 1100 series The 1100 series hardware architecture is based on a 36-bit word, one's complement structure which obtains one operand from storage and one from a high-speed register, or two operands from high-speed registers. The 1100 Operating System is designed to support a symmetrical multiprocessor configuration simultaneously providing multiprogrammed batch,

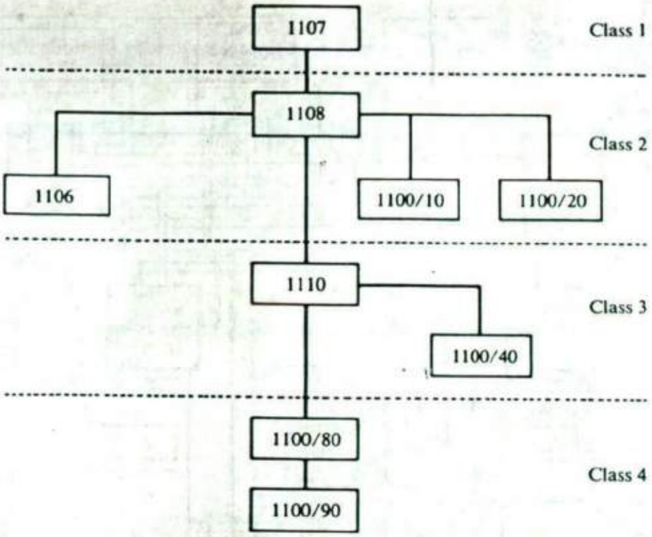


Figure 9.26 Architectural genealogy of the Univac 1100 series.

time-sharing, and transaction environments. The evolution of the Univac 1100 series is depicted in Figure 9.26. Although the basic architecture has remained the same since the 1107, the series has progressed through four architectural classes and 24 different processor-system configurations.

The 1107 was originally designed for batch-oriented scientific and engineering applications. The architectural and the operating system evolutions reflect a continuing push towards more efficient interactive and business-oriented capabilities. One of the strongest features of the 1100 series is its multiprocessor capability. Multiprocessor configurations have been in general use since the introduction of the first 1108 multiprocessor system in 1968. The System 1108 was the first commercially available general-purpose computer to support a completely symmetrical multiprocessor system; i.e., all processors share the same memory, I/O channels, and a single copy of the executive system. The major improvement of the 1108 system over the 1107 was increased throughput and enhanced protection in multiprogramming environments. The primary new feature which provided these improvements was a relative-addressing structure which created a dynamic relocation capability. In addition to faster unit processor operation, multiprocessor configurations were introduced which offered higher performance and greater system availability.

Newer versions of the 1108 system are designed to be tightly coupled symmetric multiprocessors, as shown in Figure 9.27. Each of the main storage units is multiported. The **TEST-AND-SET** instruction was added to facilitate interprocess synchronization. This instruction causes the storage unit to read a semaphore bit

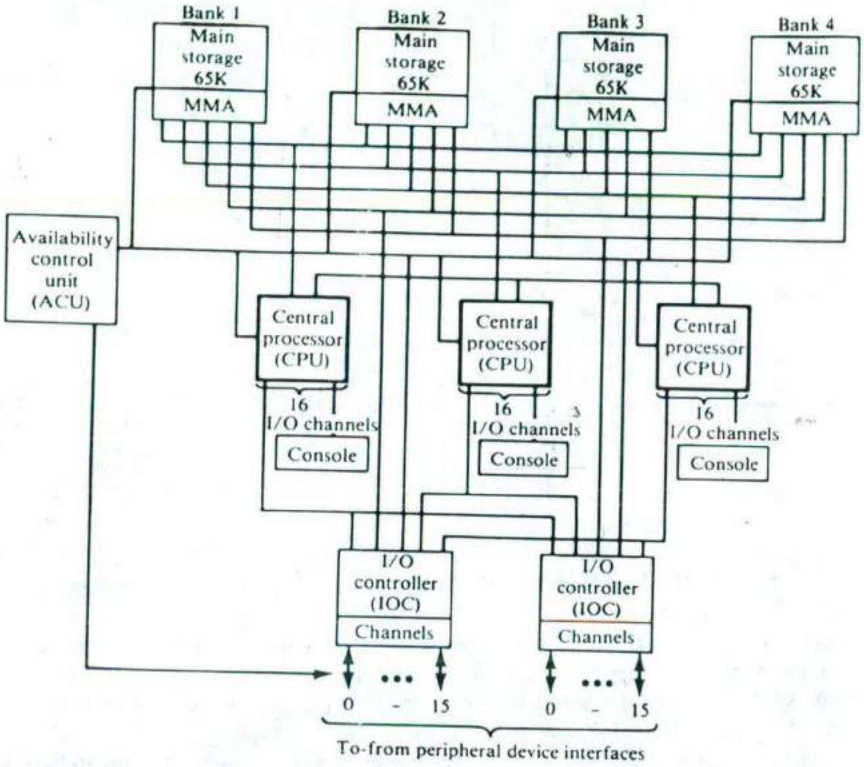


Figure 9.27 The architecture of Univac 1108 multiprocessor. (Courtesy of Sperry Rand Corp., 1965.)

and then, without allowing any other processor to access the same memory word, to set the semaphore bit. If the semaphore was initially set, an interrupt occurs (indicating that the item protected by this semaphore is already used). At this point, the interrupted process is queued until the semaphore is cleared. If the semaphore was initially clear, the next instruction is executed. Execution of the **TEST-AND-SET** instruction must precede the use of any data where erroneous results could be produced by two or more instruction streams operating on these data concurrently.

The introduction of the multiprocessor version of the 1108 led to the development of a new kind of system component called the availability control unit (ACU). This unit allows partitioning of the system into three smaller independent systems for debugging of either hardware or software on one system, while normal operation (at reduced throughput) continues on the remainder of the system. Each processor periodically sends a signal to the ACU indicating that the processor is still functioning and the executive is still in control. If the ACU does not receive all

Table 9.3 The Univac 1100 series models

	Model			
Features	1108	1110	1100/80	1100/90
First delivery	1962	1972	1977	1982
Maximum number processors	2	4	4	4
Maximum number I/O processors	2	4	4	4
Integer add time (ns)	750	300	200	60
Storage structure	One level	Primary/extended	Cache/main	Cache/main Cache/disk
Instruction set	151	206	201	238

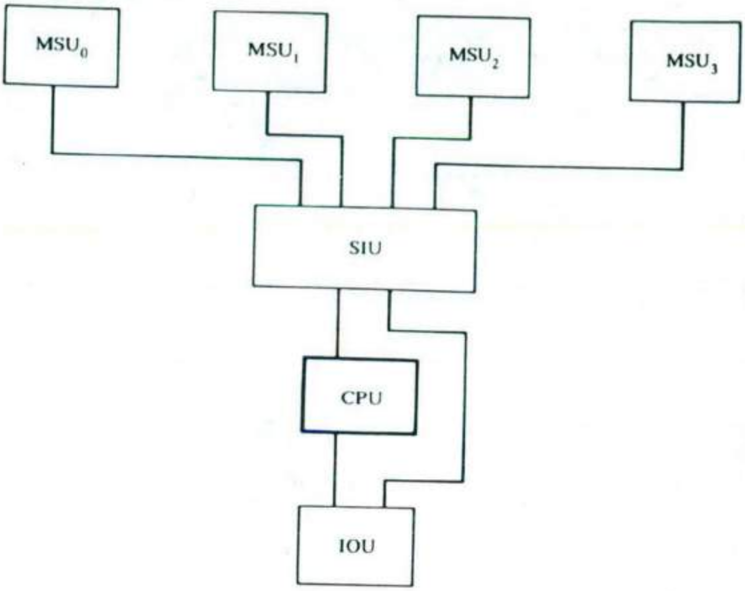
the expected signals within a predetermined time, an automatic recovery sequence is initiated.

The functional characteristic of various 1100 systems are summarized in Table 9.3. Readers can see the evolutional changes in processor features, memory structures, hardware technologies, and instruction sets from 1108 to 1100/90 in 20 years. The 1110 was the first processor in the 1100 series constructed entirely of integrated circuits (mainly with high-speed TTL). Through á chronological development of the 1100/20, 40, and 10 systems, the 1100/80 and 1100/90 systems merged as the latest product in the series.

The Univac 1100/80 systems The 1100/80 performs an add instruction in only 200 ns. Important features of the 1100/80 architecture and design approaches are listed below:

1. Addressable memory is returned to a single level structure.
2. The effective memory speed is increased by using a user-transparent cache.
3. Single-bit-error correction and double-bit error detection are on main memory.
4. The arithmetic-logic unit is microprogrammed.
5. Instructions have been added to accelerate user and executive common functions.
6. This is automatic recovery from system failures.

A uniprocessor 1100/80 system is shown in Figure 9.28. The system is organized with a central processor and an I/O unit attached to main storage units through a storage interface unit. The storage interface unit contains a cache to speed up memory reference. Peripherals are connected to the storage interface unit through channels in the I/O units. This configuration is called a 1 × 1 system, for it consists of one processor and one I/O unit. In general, 1100/80 systems are designated as $M \times N$ configurations, where M is the number of processors and N the number of I/O units. Configurations 1 × 1, 2 × 2, and 4 × 4 are possible. In a 2 × 2 system (Figure 9.29), two processors and two I/O units are connected to a storage



MSU: main storage unit
SIU: storage interface unit
IOU: I/O unit

Figure 9.28 A single-processor Univac 1100/80 configuration. (Courtesy of *Comm. of ACM*, Borgerson et al., 1978.)

interface unit. There is still only one cache, which is common to both processors and located in the storage interface unit.

Figure 9.30 depicts a 4 × 4 system. A second storage interface unit with its own independent cache is now present and connected to the two additional processors and I/O units. The two storage interface units have a cache invalidate interface which ensures that if both caches contain copies of the same data, altering the copy in one cache will cause the corresponding copy in the other to be marked as invalid.

Main memory is a common resource for all processors and I/O units and is accessed by them via the corresponding storage interface units. There can be up to four main storage units, each containing from 512K to 1M words of memory. Each main storage unit is connected to both storage interface units and can be two-way interleaved. Processors are connected to each other by interprocessor interrupt interfaces, which permit a processor to cause an interrupt in any other processor. An I/O unit is electrically connected to only one storage interface unit and to the processors on that storage interface unit. As a result, a processor can handle I/O only on I/O connected to the same storage interface unit as itself.

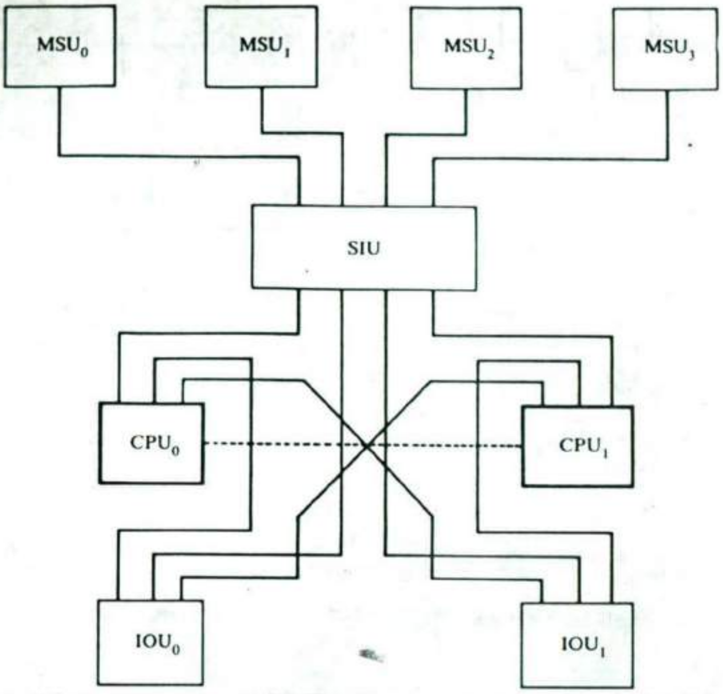


Figure 9.29 A dual-processor Univac 1100/80 configuration. (Courtesy of *Comm. of ACM*, Borgerson et al., 1978.)

The central processor of an 1100/80 has a 36-bit word length and a reasonably rich repertoire of fixed-point, floating-point, data-movement, and character-manipulation instructions. The architecture is essentially register-oriented, with separate index registers and accumulators. Most double-operand instructions have one operand in a register and one in memory. Central to the architecture of this system is a set of 128 words called the general register set (GRS). Programs can address 16 index registers and 16 accumulators.

The 1100/80 is installed with a new group of instructions to accelerate common functions for both users and the executive. These include several context-switching instructions such as save and restore system status and load and store GRS, and user-oriented instructions, including new constant storage and memory increment

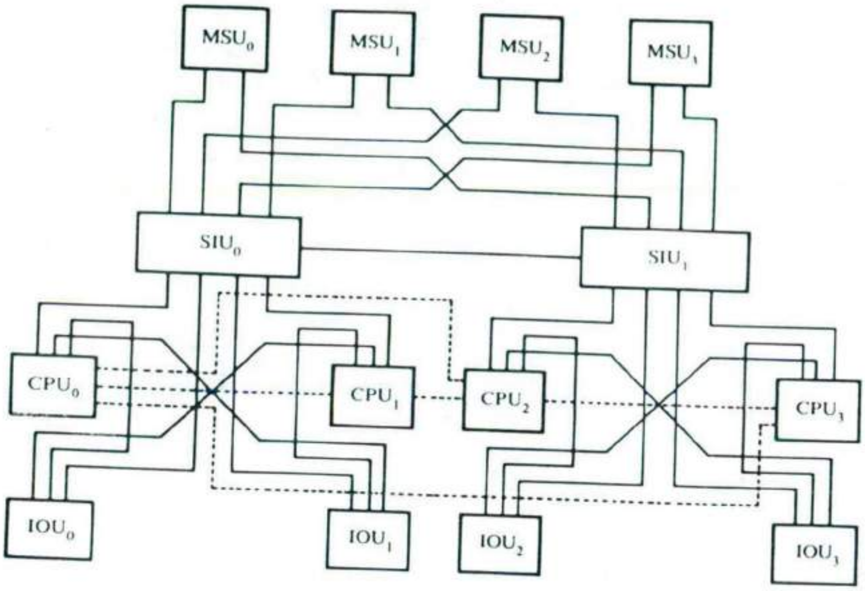


Figure 9.30 A four-processor Univac 1100/80 configuration. (Courtesy of *Comm. of ACM*, Borgerson et al., 1978.)

and decrement instructions. Two new instructions were also added to support the autorecovery feature of the 1100/80. These instructions reset the autorecovery timer and toggle the autorecovery path. When autorecovery is enabled and the system software does not reset the automatic recovery timer within the preset time interval, the system transition unit (similar to the ACU of the 1108) clears, reloads, and reinitiates the system. Two recovery paths are provided. The alternative recovery path is system initiated when an attempted automatic recovery fails. The instructions mentioned above provide for software resetting of the automatic recovery time and for selection of the first automatic recovery path to be used by the next recovery attempt.

The 1100/80 introduced instructions to aid the address-space manipulator. The most significant new instruction transfers a two-word segment descriptor directly from the segment descriptor table to the segment descriptor register, saves the previous contents of the segment descriptor register, and branches. The granularity of segment sizes has been improved on the 1100/80. Segments can be as large as 262K words and can be specified in 64-word granules beginning on any 512-word boundary and ending on any 64-word boundary.

Input-output channels on the 1100/80 are available in two forms. Word channels are available that are compatible with the 1110 system. Additionally, intelligent byte channels are available that allow the direct usage of byte-oriented peripheral equipment. The 1100/80 uses a high-speed cache memory between the processor and main storage. The cache memory is transparent to the user. It is constructed of emitter-coupled logic storage elements and contains up to 16K words; these words are the most recently used contents of main storage. The physical main storage capacity was increased to a maximum of 4M words of MOS memory. Single-bit error correction and double-bit error detection are provided.

The Univac 1100/90 systems The Univac 1100/90 multiprocessors are the most recent systems by Sperry Univac. The systems permit one, two, three, or four central processing units (CPU) as 1100/91, 1100/92, 1100/93, and 1100/94 systems, respectively. The 1100/9*x* is an *x*-by-*x* system containing *x* CPUs and *x* I/O processors, which can be tightly coupled. Figure 9.31 shows an example of a four-by-four system. However, loosely coupled configurations are also possible in which there are two independent systems sharing one mass storage subsystem.

The 1100/94 system configuration, in addition to having four CPUs and four I/O processors, contains four main storage units (MSU) and two system support processors. Each CPU is pipelined with an 8K word instruction cache and an 8K word data cache. A word is 36 bits wide. Each cache is organized into 256 sets with four blocks per set. Each block contains eight words. The CPU uses a virtual addressing scheme with 2^{36} words of address space. The initial address is divided into four portions. A segmentation scheme is used with a maximum of 262,144 segments. A write-through policy is used to update the MSU. On a write to shared data in a local cache, all caches in other CPUs containing a copy of the block are invalidated.

Each MSU contains four independent banks. A block read is a single reference resulting in four doubleword transfers with a 600-ns cycle time. A doubleword read is accomplished in 360 ns. The 1100/90 systems use the same system software as the 1100/80 systems for upward compatibility.

Operating system features in Univac 1100 series The operating system structure for the 1100 series consists of multiple layers of software, as shown in Figure 9.32. The structure of the kernel of the operating system is discussed here. The 1100 series executive system is called EXEC. A user's request to EXEC is made by executing a software interrupt instruction called *executive request interrupt* (ERI). Execution of this instruction in a processor causes a transfer of control to the executive. The EXEC has an input-output control routine concept called a symbiont (spooling routine). These routines overlap read, print, and punch operations with program execution. The EXEC also possesses multiprogramming capabilities designed to operate in both a multiprogram and multiprocessor

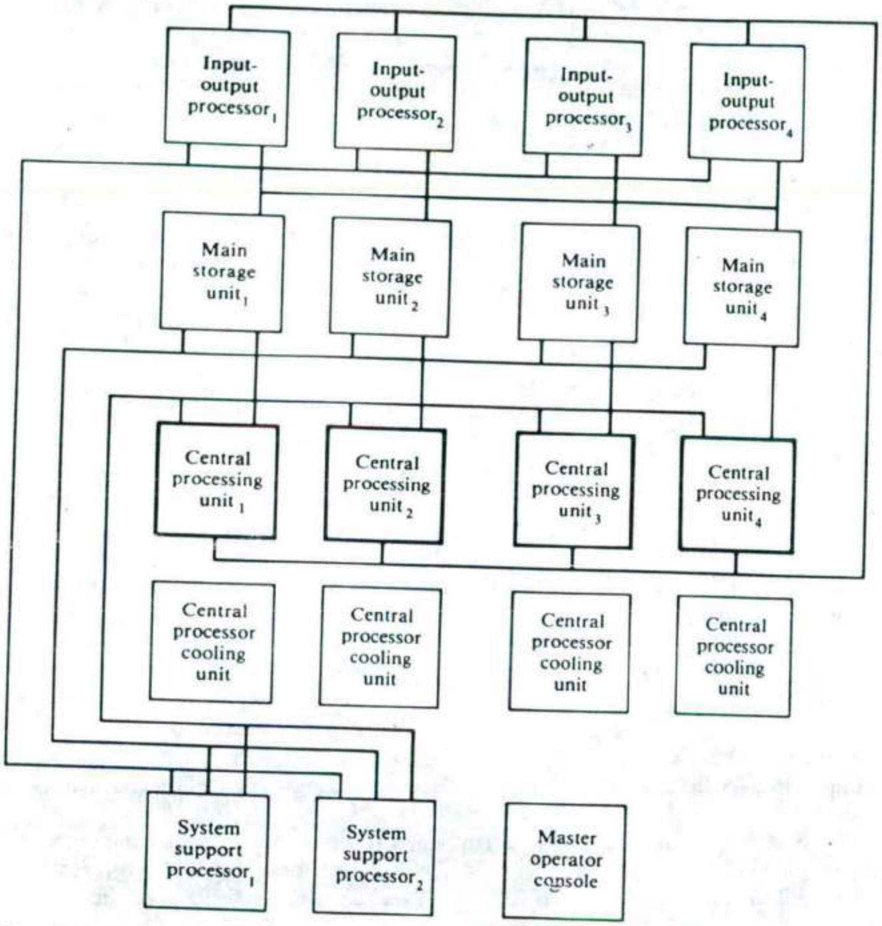

Input-output processor$_1$
Input-output processor$_2$
Input-output processor$_3$
Input-output processor$_4$
Main storage unit$_1$
Main storage unit$_2$
Main storage unit$_3$
Main storage unit$_4$
Central processing unit$_1$
Central processing unit$_2$
Central processing unit$_3$
Central processing unit$_4$
Central processor cooling unit
Central processor cooling unit
Central processor cooling unit
Central processor cooling unit
System support processor$_1$
System support processor$_2$
Master operator console

Figure 9.31 The Univac 1100/94 system with four processors. (Courtesy of Sperry Rand Corp., 1983.)

environment. Moreover, it supports a concurrent mix of batch, remote batch, remote batch, demand and transaction programs.

Particular emphasis is placed on demand mode also known as time sharing. All system facilities available in batch mode are also available in demand mode; the same *run stream* can be used in either batch mode or demand mode without change. A run stream is a collection of user service requests used for batch, remote batch, and demand. Included in the run stream are any data images supporting each service request. The run stream is formulated using a control language consisting of service requests. A task is an individual service request, to assign a tape, for example, or to perform a compilation within a run stream.

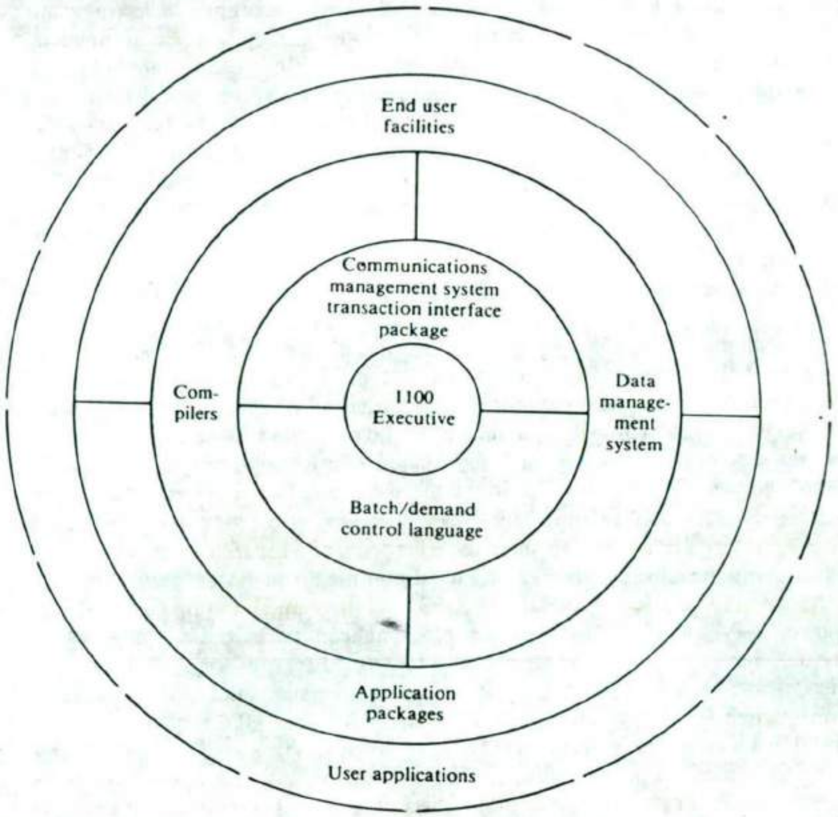


Figure 9.32 The Univac/EXEC operating system.

The origin of a program is in symbolic elements within the run stream. These elements are then compiled to form relocatable elements which are collected (bound) with other relocatable elements to form an absolute element (the program). The term "absolute" refers to the program relative-address solution only; the relative-addressing capability of the hardware allows the program to be loaded (or swapped and reloaded) and executed anywhere within main storage. References to shared segments of both user code and system libraries are resolved during execution.

A batch stream which enters the system is first processed by the symbiont complex. This complex disassociates the run stream from the relatively slow unit-record device speeds and allows tasks to proceed at higher mass-storage speeds. The run stream is scanned for facility allocation and prescheduling. Multiple asynchronous input-output services are allowed. This is particularly important in a multiprogramming-multiprocessing environment.

After staging in a mass-storage schedule queue, the run stream is processed by the coarse scheduler, which is responsible for the scheduling, selecting, and activating of the run stream. The run is assigned a temporary program file in which intermediate results, such as compiler output or text editor output, are held until termination, when the file is released. The results may be retained by directing them instead to a permanent file or copying the contents of the temporary file prior to termination. Demand runs are scheduled for immediate activation; batch runs are scheduled on a user-selected priority basis. A major batch-scheduling consideration at this point is the availability of facilities such as tape devices. When necessary facilities are available, the program is then queued for main storage allocation.

The dynamic allocator, which has the responsibility for the distribution of main storage space among users, removes the program on a priority basis. The main functions of the dynamic allocator include the allocation and release of main storage and the initial load, swap-out, and reload of programs. The decisions of the dynamic allocator are facilitated by relative addressing, which allows the program to be loaded anywhere in memory. The program may be partitioned into multiple segments which need not be loaded contiguously. Segments (usually shared) that were not a part of the initial program may be referenced (and loaded) dynamically. This facility allows for greater program protection and re-entrancy efficiency. It also allows each segment to be loaded into a separate memory module, thus reducing the effective instruction execution time through storage unit overlap.

A switch list is used by the 1100 EXEC to control an executing program. The program may fork into any number of asynchronous execution paths called activities, each of which is independently scheduled (except for synchronization requests). It is given main memory space and a time slice under the control of a unique switch list entry. The executing activity requests 1100 executive services through a set of executive requests. These perform services such as input-output, control statement processing, and activity activation. An I/O operation can be requested by a user program. If the request is not to an I/O unit on the same storage interface unit as the processor that is running the program, the processor will signal another processor on the other storage interface unit. The I/O operation is then initiated by this processor. The signalling is performed via the interprocessor interrupt mechanism. On completion of the I/O transaction for this request, the I/O unit sends a completion interrupt signal. One of the processors on the same interface handles this interrupt for eventual notification of the user program which requested the I/O transaction.

The dispatcher allocates real processor time slices to outstanding activities according to the priority and needs of the respective activities. In a multiprocessor system, any available processor may be assigned to execute the next time slice. Thus, an activity may execute successive time slices on different processors, or two parallel activities within the same program may be executing concurrently on different processors. Upon completion of processing within an activity, a request for normal or abnormal termination is made. After all activities within a task have terminated, the task is terminated and control is returned to the coarse scheduler. We have only sketched the EXEC operations here. Interested readers should check with the Univac EXEC manuals for details.

9.5.4 The Tandem Nonstop System

On-line computing with continuous availability is in high demand in many business applications. Certain applications, such as automatic toll billing for telephone systems, lose money each minute the system is down and the losses are irrecoverable. The Tandem-16 Nonstop system was designed to offer better availability than most existing multiple processor computers. The system is organized around three types of components: the processors, device controllers, and system buses (Figure 9.33). The processors are interconnected by the *dynabus*, which consists of two buses. The device controllers are each connected to two independent I/O channels, one from each processor on its side. The system is designed to continue operation through any single failure and to allow on-line repairing without affecting the rest of the system. The on-line maintenance was a key factor in using dual power supplies in the system.

Each processor includes a 16-bit CPU, a main memory, a bus control, and an I/O channel (Figure 9.34). Up to 2M bytes of main memory are available. The processor module is viewed by the user as a stack-oriented computer with a virtual memory system capable of supporting multiprogramming. The CPU is a microprogrammed processor consisting of eight registers which can be used as general-purpose registers, an ALU, and several miscellaneous flags and counters. The instruction set includes arithmetic operations, logical operations, procedure calls and exits, interprocessor SENDs, and I/O operations. The system has 16 interrupt levels which include bus data received, I/O transfer completion, memory error, interval time, page fault, privileged instruction violation, etc. Packets are the primitive data transferred over the dynabus.

Main memory is organized in physical pages of 1K words. Up to one megaword of memory may be attached to a processor. In a semiconductor memory system, there are six check bits per word to provide single error correction and double error detection. The semiconductor memory with error correction is more reliable than core. Battery backup provides short-term nonvolatility of the semiconductor memory system.

Memory is logically divided into four address spaces. These are the virtual address spaces of the machine; both the system and the user have separate code spaces and data spaces. The code space is unmodifiable and the data space can be viewed either as a stack or a random-access memory, depending on the adressing mode used. Each virtual address space has 46K words. Four maps are provided, one for each logical address space. The map access time and the delay by error correction are included in the 500-ns cycle time of the semiconductor memory system. The high-level language provided on the Tandem 16 system is the TAL, a block-structured, ALGOL-like language.

Multiple processors in Tandem-16 communicate with each other over the X bus and Y bus shown in Figure 9.35. Bus access is determined by two independent

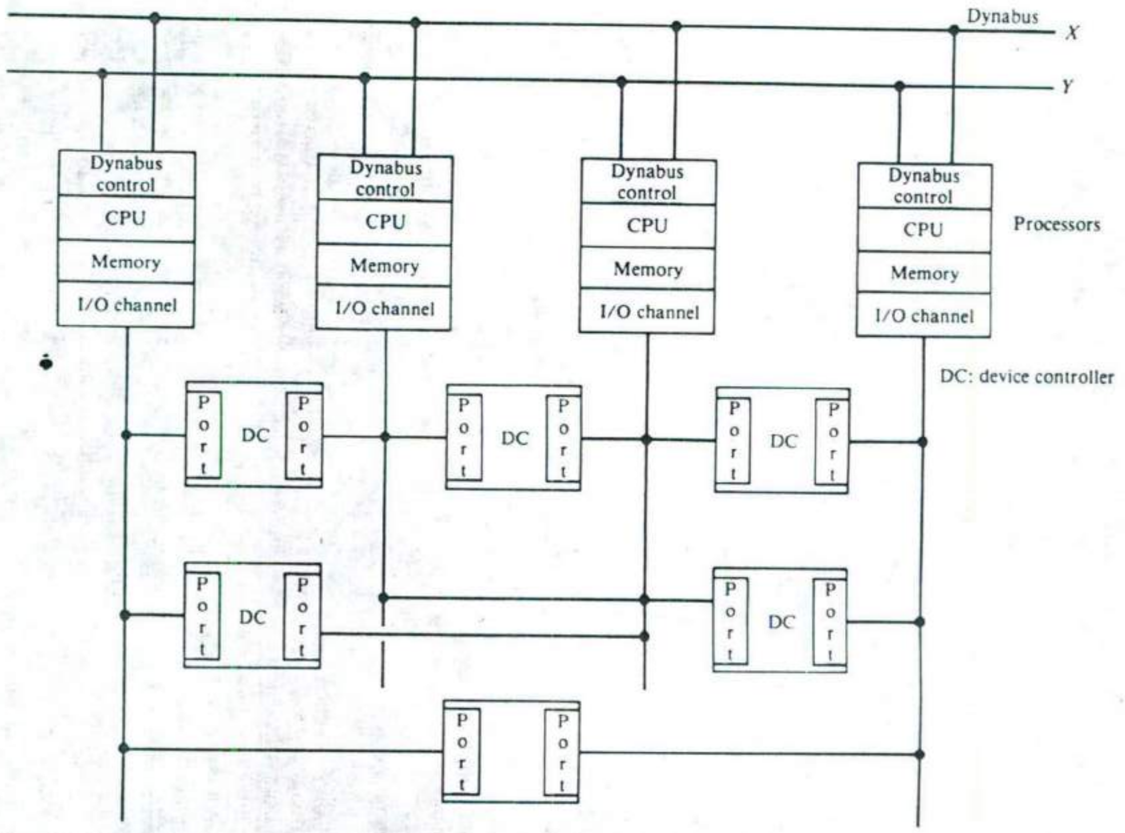


Figure 9.33 A typical Tandem system with four processors. (Courtesy of Tandem Computer, Inc., 1978.)

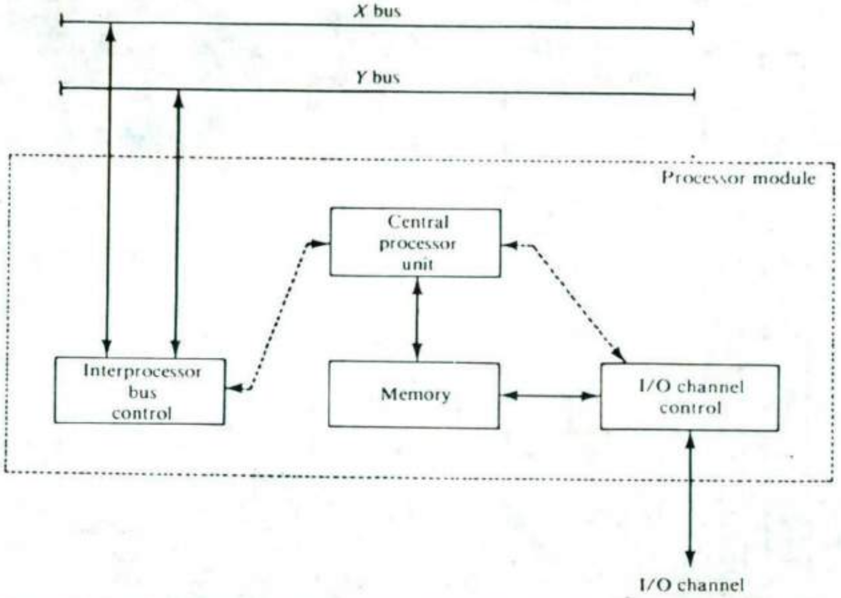


Figure 9.34 System components in one processor of the Tandem/16. (Courtesy of Tandem Computer, Inc., 1978.)

interprocessor bus controllers. There are two sets of radial connections from each bus controller to each processor module. They distribute clocks for synchronous transmission over the bus. No failed processor can dominate the dynabus utilization. Each bus has a clock associated with it, running independently of the processor clocks.

The dynabus interface controller consists of three high-speed caches, two of which are associated with the two buses and one is an output queue that can be switched between the two buses. Each caches has 16 words and all bus transfers are made from cache to cache. All components attaching to the buses are kept physically distinct, so that no single component failure can contaminate both buses simultaneously. Also, the controller has clock synchronization and interlock circuitry. All processors communicate in a point-to-point manner using this shared bus configuration.

For any interprocessor data transfer, one processor is the sender and the other is the receiver. Before a processor can receive data over a bus, the operating system must configure an entry in a table known as the *bus receive table* (BRT). Each BRT entry contains the address where the incoming data is to be stored, the sequence number of the next packet, the send processor number, the receiver processor number, and the number of words to be transfered. A SEND instruction is executed in the sending processor, which specifies the bus to be used, the intended

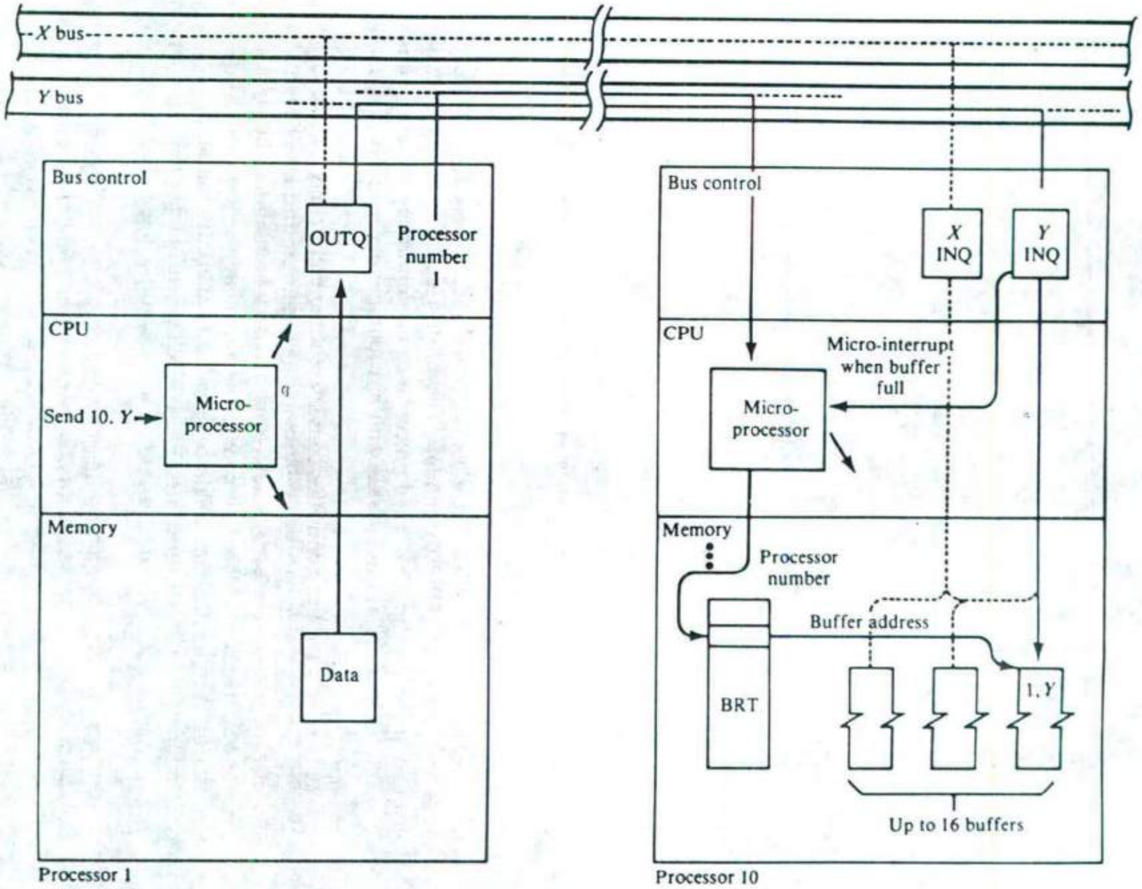


Figure 9.35 Tandem/16 dynabus interface for interprocessor data transfer. (Courtesy of Tandem Computer, Inc., 1978.)

receiver, and the number of words to be sent. The sending processor's CPU stays in the SEND instruction until the data transfer is completed. Up to 65,535 words can be sent in a single SEND instruction. While the sending processor is executing the SEND instruction, the dynabus control logic in the receiving processor is storing the data away according to the appropriate BRT entry. In the receiving processor, this occurs simultaneously with its program execution.

A message is divided into packets of 16 words. The sending processor fills its outgoing queue with these packets, requests a bus transfer, and transmits upon grant of the bus by the bus controller. The receiving processor fills the incoming queue associated with the bus and issues a microinterrupt to its own CPU. The CPU checks the BRT entry accordingly. The BRT entries are four words that include a transfer buffer address, a sequence number, and the sender and receiver processor numbers. Error recovery action is to be taken in case the transfer is not completed within a time-out interval. These parameters are placed on a register stack and are dynamically updated so that the SEND instruction is interruptible on packet boundaries.

All I/O is done on a direct memory access basis through a microprogrammed, multiplexed channel with a block size determined by the individual controller. All the controllers are buffered so that all transfers over the I/O channel are at memory speed (4M bytes/s) and never wait for mechanical motion since the transfers always come from a buffer in the controller rather than from the actual I/O device. There exists the I/O Control Table (IOC) in the system data space of each processor that contains a two-word entry for each of the 256 possible I/O devices attached to the I/O channel. These entries contain a byte count and virtual address in the system data space for I/O data transfers. The I/O channel moves the IOC entry to active registers during the connection of an I/O controller and restores the updated values to the IOC upon disconnection. The channel transfers data in parallel with program execution. The memory system priority always permits I/O accesses to be handled before CPU or dynabus accesses.

The dual-ported I/O device controllers provide the interface between the I/O channel and a variety of peripheral devices. Each controller contains two independent I/O channel ports so that it can never simultaneously cause failure of both ports. Each port attached to an I/O channel must be assigned a controller number and a priority distinct from other ports attached to the same I/O channel. Logically only one of the two ports of an I/O controller is active; the other port is utilized only in the event of a path failure to the primary port. If a processor determines that a given controller is malfunctioning on its I/O channel, it can issue a command that logically disconnects the port from the controller. This does not affect the ownership status. If the problem is within the port, an alternate path can be used.

Each disk drive in the system may be dual-ported. Each port of a disk drive is connected to an independent disk controller. Each of the disk controllers are also dual-ported and connected between two processors. A string of up to each drives (four mirrored pairs) can be supported by a pair of controllers in this manner. The disk controller is buffered and absolutely immune to overruns. All data transferred over the bus is parity checked in both directions, and errors are reported via the interrupt system. A watchdog timer in the I/O channel detects if a nonexistent I/O controller has been addressed, or if a controller stops responding during an I/O sequence. In case of channel failure, the path switching between devices and controller is demonstrated in Figure 9.36, where an alternate path is chosen to provide the access of the disk.

The operating system of Tandem is called the Guardian. It is a "nonstop" operating system designed to achieve the following capabilities:

1. It should be able to remain operational after any single detected module or bus failure.
2. It should allow any module or bus to be repaired on-line and then reintegrated into the system.
3. It is to be implemented with high reliability provided by the hardware but not negated by software problems.
4. It should support all possible hardware configurations, ranging from a two-processor, diskless system through a sixteen-processor system with billions of bytes of disk storage.
5. It should hide the physical configuration as much as possible so that applications could be written to run on a great variety of system configurations.

The Guardian resides in each processor but is aware of all other processors. In fact, the operating systems in different processors constantly monitor each other's performance. The instant one processor's operating system fails to respond correctly, other processors assume that it is failing and take over its work load. Obviously, this requires a great deal of communication among the processors. This requires a process to be able to address the system resources by a logical name rather than by a physical address. The Guardian operating system is designed in a top down manner with three levels of well-defined interfaces, as shown in Figure 9.37. It is based on the concept of processes sending messages to other processes.

All resources in the system are considered to be files, and each resource has a logical file name. Communication between an application process and any resource (disk, tape, another process, etc.) is via the file system. The file system knows only the logical name of the intended recipient of a message. It passes the message to the message system, which then determines the physical location of the recipient. The message system is a software analog of the dynabus. It handles automatic path retries in case of path errors. Because application programs deal only with logical file names, the system offers total geographic independence of resources. The programmer views this multiprocessor system as a single processor with resources available through file system calls.

The processes and messages are further elaborated with abstraction. Each processor module has one or more processes residing in it. A process is initially created in a specific processor and may not execute in another processor. Each process has an execution priority assigned to it. Processor time is allocated to the

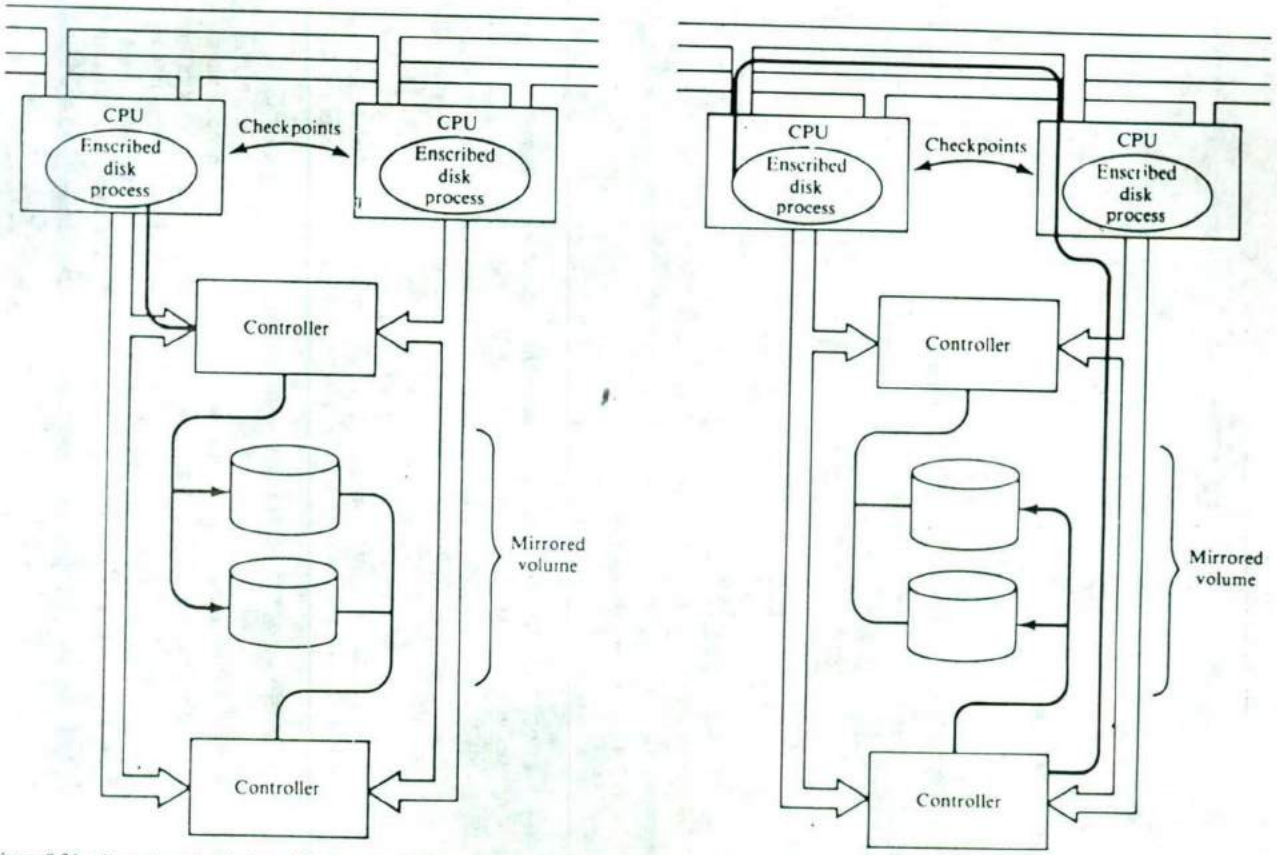


Figure 9.36 Alternate path switch on I/0 controller failure. (Courtesy of Tandem Computer, Inc., 1978.)

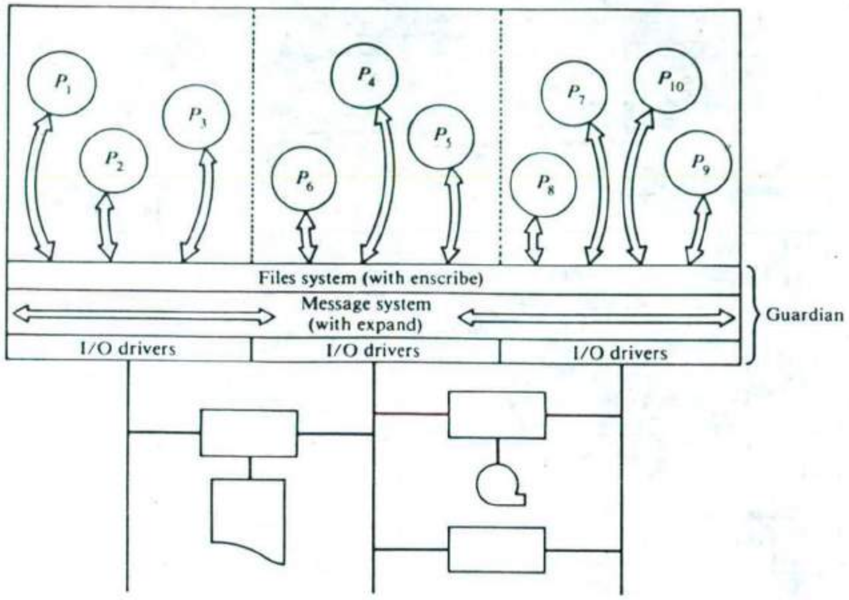


Figure 9.37 The Guardian operating system.

highest priority process. Process-synchronization primitives include counting semaphores and process local event flags. Semaphore operations are performed via the functions **PSEM** and **VSEM**, similar to Dijkstra's **P** and **V** operators. Semaphores may only be used for synchronization between processes within the same processor. They are typically used to control access to resources such as resident memory buffers, message control blocks, and I/O controllers.

The message system provides five primitive operations, which are illustrated in Figure 9.38 in the context of a process making an inquire to the process. A process sends a message to the appropriate server process via a procedure LINK. The message will consist of parameters denoting the type of inquires and data needed. The message will be queued in the server process, setting an event flag, and then the requester process may continue executing. The server process calls a procedure to return the first message queued. It will then obtain a copy of the requestor's data by calling the procedure READLINK. Next, the server process will process the request. The status of the operation and the result will then be returned by the WRITELINK procedure, which will signal the requester process via another event flag. Finally, the requester process will complete its end of the transaction by calling the BREAKLINK procedure.

The message system is designed to obtain resources needed for message transmission (control blocks) at the start of a message-transfer request. Once the

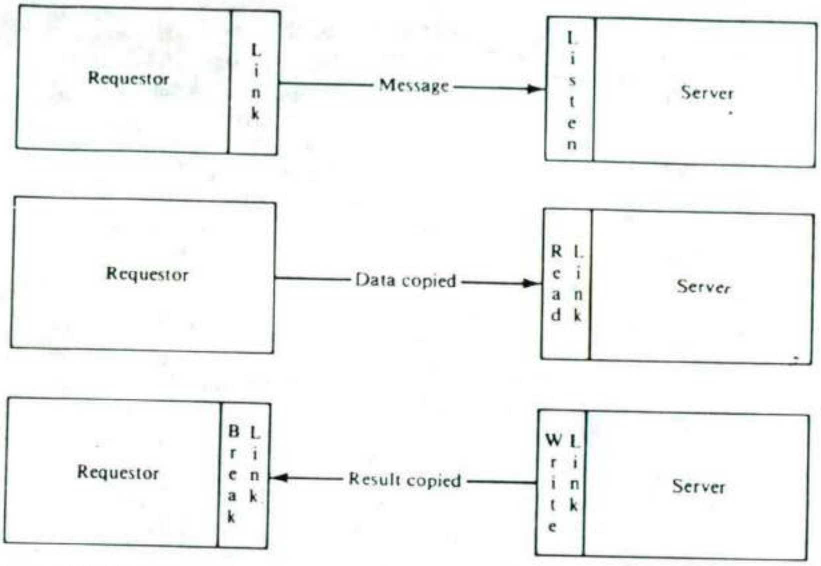


Figure 9.38 The message system primitive developed in Tandem/Guardian O.S.

LINK has been successfully completed, both processes are assured that sufficient resources are in hand to complete the message transfer. Furthermore, a process may reserve some control blocks to guarantee that it will always be able to send messages to respond to a request from its message queue. Such a resource control assures that deadlocks are prevented by complex producer-consumer interactions.

The Guardian is constructed of processes which communicate with messages. Fault tolerance is provided by duplication of both hardware and software components. Access to I/O devices is provided by process pairs consisting of a primary process and a backup process. The primary process must check out state information to the backup process so that the backup may take over on a failure. Requests to these devices are routed using the logical device name or number so that the request is always routed to the primary process. The result is a set of primitives and protocols which allow recovery and continued processing in spite of single failures in bus, processor, or I/O device.

A "network" system can link up to 255 Tandem-16 systems. The Guardian-Expand Network system can extend the dynabus into a long-range network. To a user at a terminal, the entire network appears to be a single Tandem-16 system. The network maintains the geographic independence of resources. Any resource in the network can be addressed by its logical file name without regard for its physical location. However, a configuration option allows users to reserve processors for local processing requirements, thereby excluding those processors from the network.

9.6 THE CRAY X-MP AND CRAY-2

The Cray X-MP is a multiprocessor extension of the Cray-1. It contains two Cray-1-like processors with shared memory and I/O subsystems. The first X-MP was installed in later 1983. Cray Research is currently also developing a four-processor Cray-2. In this section, we describe the system architecture of X-MP and examine its vector processing and multitasking capabilities. We will examine the multiprocessing performance of X-MP for both compute-bound and I/O-bound applications. The details of Cray-2 were not available at the time this book was published. However, some target specifications of Cray-2 will be indicated simply to show the trend of development.

9.6.1 Cray X-MP System Architecture

The system organization of the Cray X-MP is shown in Figure 9.39. The mainframe features two identical CPUs and a multiport memory shared by both

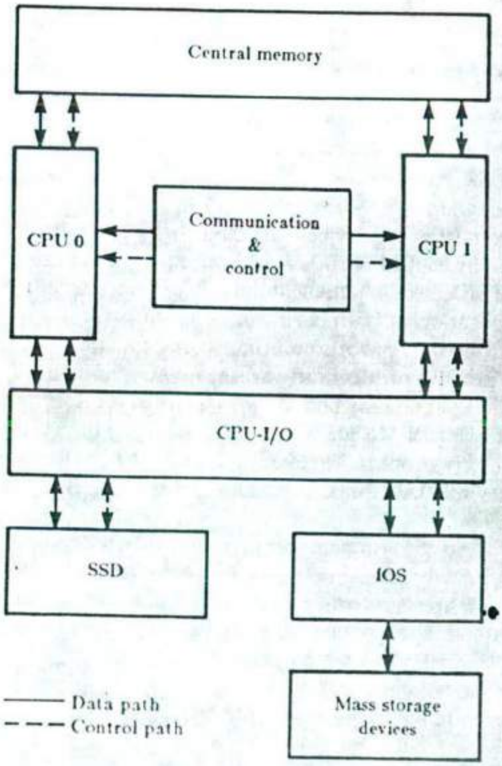


Figure 9.39 Cray X-MP overall system organization. (Courtesy of Cray Research, Inc., 1983.)

processors. Each CPU has an internal structure very similar to Cray-1. However, there are three ports per CPU (instead of two ports in Cray-1). The extra port is added to allow communication between the two CPUs via a communication and control unit.

Shared central memory The two processors share a central bipolar memory with 4M 64-bit words. This shared memory is organized in 32-way interleaved memory banks (twice that of Cray-1). All banks can be accessed independently and in parallel during each machine clock period. Each processor has four parallel memory ports (four times that of Cray-1) connected to this central memory, two for *vector fetches*, one for *vector store*, and one for independent I/O operations. The multiport memory has built-in conflict resolution hardware to minimize the delay and maintain the integrity of all memory references to the same bank at the same time, from all processor's ports. The interleaved multiport memory design coupled with shorter memory cycle time provides a high-performance and balanced memory organization with sufficient bandwidth (eight times that of Cray-1) to support simultaneous high-speed CPU and I/O operations.

The CPU of X-MP Throughout the two CPUs, 16-gate array integrated circuits are used. These circuits, which are faster and denser than the circuitry used in the Cray-1, contributed to a clock cycle time of 9.5 ns and a memory bank cycle time of 38 ns. Proven cooling and packaging techniques have also been used on the Cray X-MP to ensure high system reliability.

Each CPU is basically a Cray-1 processor with additional features to permit multiprocessing. Within each CPU are four instruction buffers, each with 128 16-bit instruction parcels, twice the capacity of the Cray-1 instruction buffers. The instruction buffers of each CPU are loaded from memory at the burst rate of 8 words per clock period. The contents of the exchange package have been augmented to include cluster number and processor number. Increased protection of data is also made possible through a separate memory field for user programs and data. Exchange sequences occur at the rate of 2 words per clock period on the X-MP.

Operational registers and functional pipelines are among the features providing compatibility with the Cray-1. There are 13 functional pipes and A, B, S, T, and V registers as in Cray-1. With a basic machine cycle of 9.5 ns, the X-MP is capable of an overall instruction issue rate of over 200 MIPS. Computation rates of over 400 megaflops are possible, and combined arithmetic/logical operations can exceed 1000 million operations per second.

CPU intercommunication The CPU intercommunication section comprises three clusters of shared registers for interprocessor communication and synchronization. Each cluster of shared registers consists of eight 24-bit shared address (SB) registers, eight 64-bit shared scalar (ST) registers, and thirty-two 1-bit synchronization (SM) registers. Under operating system control, a cluster may be allocated to both, either, or none of the processors. The cluster may be accessed by any processor to which it is allocated in either user or system mode. A 64-bit real-time clock is shared by the processors.

Solid-state storage device (SSD) A new and large CPU-driven *solid-state storage device* (SSD) is designed as an integral part of the mainframe with very high block transfer rate. This can be used as a fast-access device for large prestaged or intermediate files generated and manipulated repetitively by user programs, or used by the system for job "swapping" space and temporary storage of system programs. The SSD design with its large size (32 megawords), typical rate of 1000 megabytes/s (250 times faster than disk), and much shorter access time (less than 0.5 ms, 100 times faster than disk), coupled with the high-performance multiprocessor design, will enable the user to explore new application algorithms for solving bigger and more sophisticated problems in science and engineering. The concept of SSD is illustrated in Figure 9.40. It performs much better than that of the disk due to its shorter access time and faster transfer rate.

I/O subsystem (IOS) The I/O subsystem, which is an integral part of the X-MP system, also contributes to the system's overall performance. The I/O subsystem (compatible with Cray-1/2) offers parallel streaming of disk drives, I/O buffering (8 megawords maximum size) for disk-resident and buffer memory-resident datasets, high-performance on-line tape handling, and common device for front-end system communication, networking, or specialized data acquisition. The IOP design enables faster and more efficient asynchronous I/O operations for data access and deposition of initial and final outputs through high-speed channels (each channel has a maximum rate of 850 megabytes/s, and a typical rate of 40 megabytes/s, 10 times faster than disk), while relieving the CPUs to perform computation-intensive operations.

Interfaces to front-end computers The Cray X-MP is interfaced to front-end computer systems through the I/O subsystem. Up to three front-end interfaces per I/O subsystem, identical to those used in the Cray-1, can be accommodated. Front-end

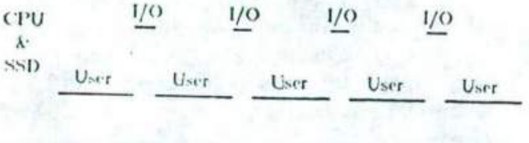


Figure 9.40 The concept of SSD in Cray X-MP as compared with the use of disks.

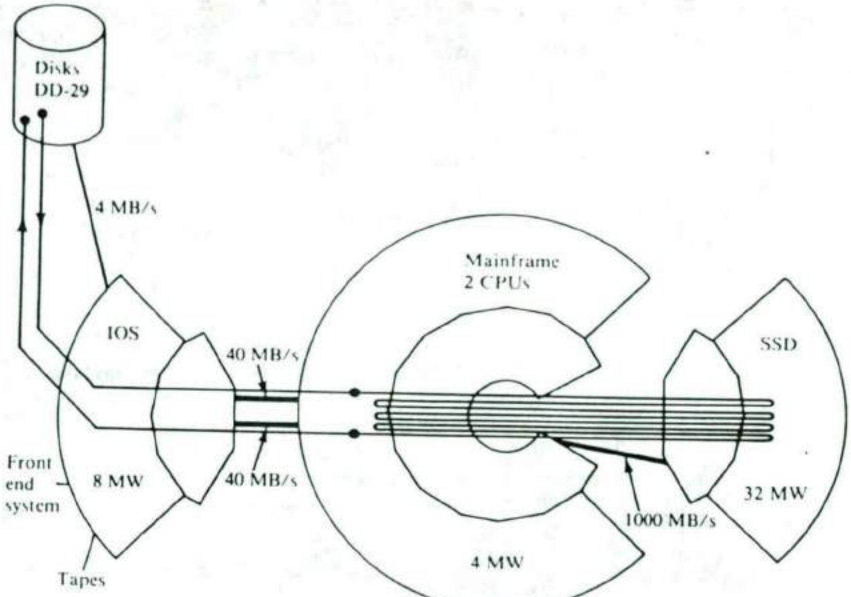


Figure 9.41 The data flow and transfer rates in Cray X-MP. (Courtesy of Cray Research, Inc., 1983.)

interfaces compensate for differences in channel widths, word size, logic levels, and control protocols, and are available for a variety of front-end systems. The X-MP can be connected to front-end machines like IBM/MVS, CDC NOS, and NOS/BE, systems.

The data flow patterns and data transfer rates among the mainframe (two CPUs and main memory of 4 megawords), the SSD, the IOS, the external disks and tapes, and the front-end system are illustrated in Figure 9.41. The high-speed transfer rates between the mainframe and SSD (1000 megabytes/s) and between the mainframe and IOS (80 megawords/s) make the system suitable for solving large-scale scientific problems, which are both computation-intensive and I/O demanding.

9.6.2 Multitasking on Cray X-MP

The X-MP is designed to be a general purpose multiprocessor system for multitasking applications. It can run independent tasks of different jobs on two processors. Program compatibility with Cray-1 is maintained for all tasks. It can also run related tasks of a single job on two processors. The two processors are tightly coupled through shared memory and shared registers. The system has low overhead of task initiation for multitasking. Typically O(1 μs) to O(1 ms) times is needed, depending on granularity of the tasks and software implementation techniques.

The two processors can assume various flexible architectural clustering patterns. Faster exchange for switching machine state between tasks is provided. Hardware supports the separation of memory segments for each user's data and program to facilitate concurrent programming.

Let $p = 2$ be the number of physical processors in the system. Listed below are various processor clustering patterns that are challenged in the design of X-MP. The Cray Operating System (COS) has been designed to control the allocation of the clusters of shared registers to the CPU in either user or supervisor mode.

1. All processors are identical and symmetric in their programming functions, i.e., there is no permanent master-slave relation existing between all processors.
2. A cluster of k processors ($0 \leq k \leq p$) can be assigned to perform a single task.
3. Up to $p + 1$ processor clusters can be assigned by the operating system.
4. Each cluster contains a unique set of shared data and synchronization registers for the intercommunication of all processors in a cluster.
5. Each processor in a cluster can run in either monitor or user mode controlled by the operating system.
6. Each processor in a cluster can asynchronously perform either scalar or vector operations dictated by user programs.
7. Any processor running in monitor mode can interrupt any other processor and cause it to switch from user mode to monitor mode.
8. Detection of system deadlock is provided within the cluster.

The vector performance of each processor is improved through faster machine clock, parallel memory ports, and hardware automatic "flexible chaining" features. These new features allow simultaneous memory fetches, arithmetic, and memory store operations in a series of related vector operations (this contrasts to the "fixed chaining" and unit-directional vector fetch/store in Cray-1). As a result, the processor design provides higher speed and more balanced vector processing capabilities for both long and short vectors, characterized by heavy register-to-register or heavy memory-to-memory vector operations.

Example 9.4 Vector computations on X-MP are illustrated in Figure 9.42 for the following computation:

$$\mathbf{A} = \mathbf{B} + \mathrm{S} * \mathbf{D}$$

where boldface letters denote vector quantities. Using three memory ports per processor, the hardware automatically "chains" through all five vector operations such that one result per clock period can be delivered.

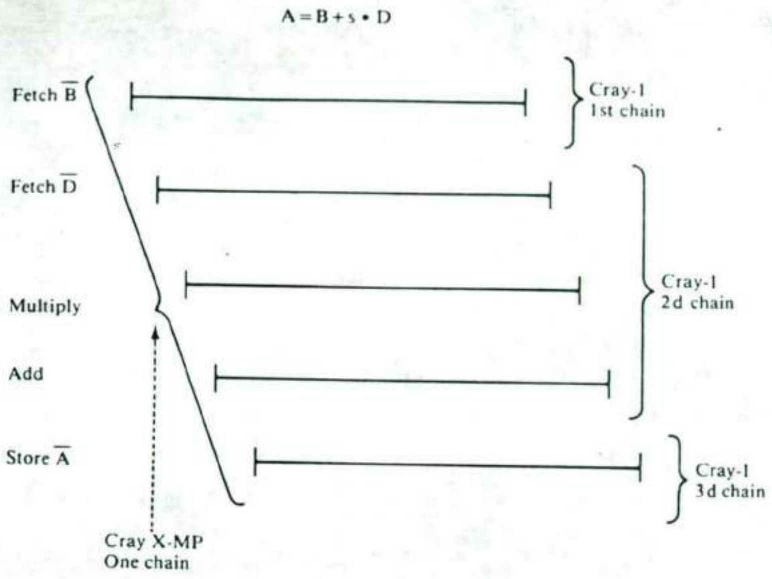
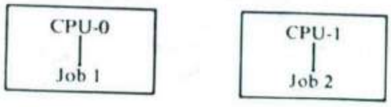


Figure 9.42 Pipeline chaining in Example 9.4 for Cray X-MP.

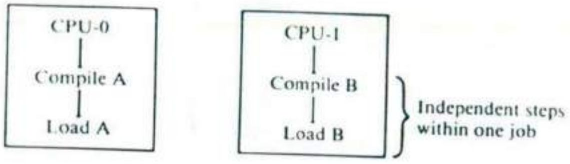
Multitasking exploits another dimension of parallelism beyond that provided by vectorization. So far, *vectorization* has been focused on the low-level parallelism, primarily among independent and statement-oriented operations from single-job programs. The *multitasking* offers a high-level parallelism among independent algorithms, which are job/program/loop-oriented to achieve both single and multijob performance. Combining both vectorization and multitasking, one can explore new and faster parallel algorithms at several levels, as listed below:

1. Multitasking at the job level (Figure 9.43*a*)
2. Multitasking at the job-step level (Figure 9.43*b*)
3. Multitasking at the program level (Figure 9.44)
4. Multitasking at the loop level (Figure 9.45)

When this book was published, software support for multitasking at the job level (Figure 9.43*a*), the program level (Figure 9.44), and the loop level (Figure 9.45) were available from Cray Research. However, the feasibility of implementing multitasking at the job-step level (Figure 9.43*b*) is still under further study. In what follows, we show three examples to illustrate the concept of multitasking, in particular, for the X-MP multiprocessor system.



(*a*) Multitasking at the job level



(*b*) Multitasking at the job-step level

Figure 9.43 Multitasking at job levels for Cray X-MP.

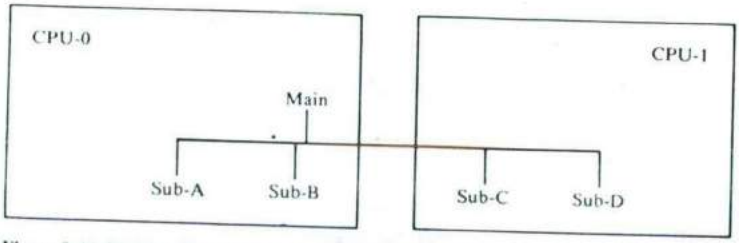


Figure 9.44 Multitasking at the program level for Cray X-MP.

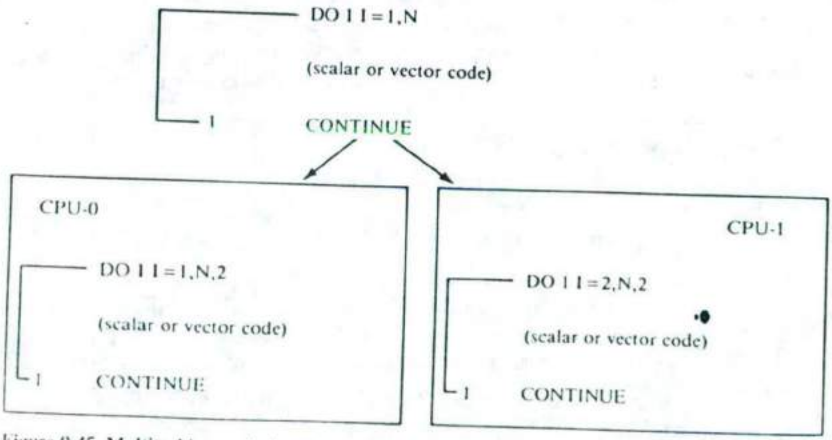


Figure 9.45 Multitasking at the loop level for Cray X-MP.

Example 9.5 Multitasking of vector code and scalar code is illustrated in Figure 9.46 for the following two loop computations, respectively.

```
      DO  2 I = 1,M
            :
          DO 1 J = 1,N                 }
                                       }  Vector
1           A(I,J) = B(I,J) + C(I,J)   }  code
            :
2         CONTINUE
```

```
      DO  2 I = 1,M
            :
          DO 1 J = 1,N                 }
                                       }  Scalar
1           A(I,J) = A(I,J-1)*A(I,J)   }  code
            :
2         CONTINUE
```

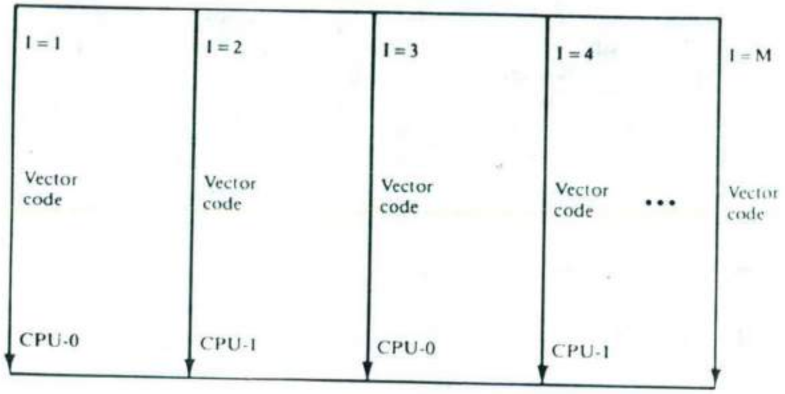
Example 9.6 Multitasking by processor pipelining (macropipelining introduced in Chapter 1) is illustrated in Figure 9.47 for the following loop computations, where S1 and S2 stand for the two vector computations involved.

```
   DO 1 I = 2,N
   A(I) = A(I - 1) + B(I)   S1
   D(I) = A(I) + C(I)       S2
1  CONTINUE
```

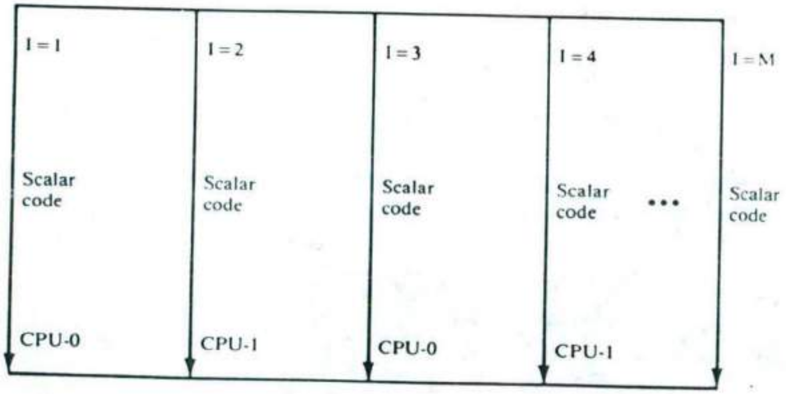
9.6.3 Performance of Cray X-MP

In this section, the performance of X-MP is evaluated with theoretical predictions and benchmark timings as reported by the designers. The X-MP processor design is well-balanced for processing both scalar and vector codes. At the end, we sketch the Cray-2, which is still under development.

The overall effective performance of each processor in execution of typical user programs with interspersing scalar and vector codes (usually short vectors) is ensured through fast data flow between scalar and vector functional units, short memory access time for vector and scalar references, as well as small start-up time for scalar and vector operations. As a result of this design characteristic, the machine can perform very well in real programming environments using standard compiler, without resorting to an enormous amount of hand-coding or even restructuring of the original application algorithms. Certainly, as the code is more



(*a*) For vector loop



(*b*) For Scalar loop

Figure 9.46 Multitasking in Example 9.5 among two CPUs in Cray X-MP.

vectorized, and the vector length is becoming longer, an even better performance can be achieved.

With a 9.5-ns clock rate, the peak speed of one CPU in X-MP is 210 megaflops and that of using two CPUs (dedicated to multitasking of a single large job) is 420 megaflops. We assume one *unit* to be based on compiler-generated code running on Cray 1/S. This means that the "minimum" 1-CPU rate is 1 and that of 2-CPU is 2. Let "typical" be the cases of small-to-medium size vectors encountered in typical programs. Let "maximum" refer to the cases of very long vectors. The performance of various types of programs relative to that of Cray 1/S is summarized in Table 9.4. The unit for I/O rate is based on measured time per sector.

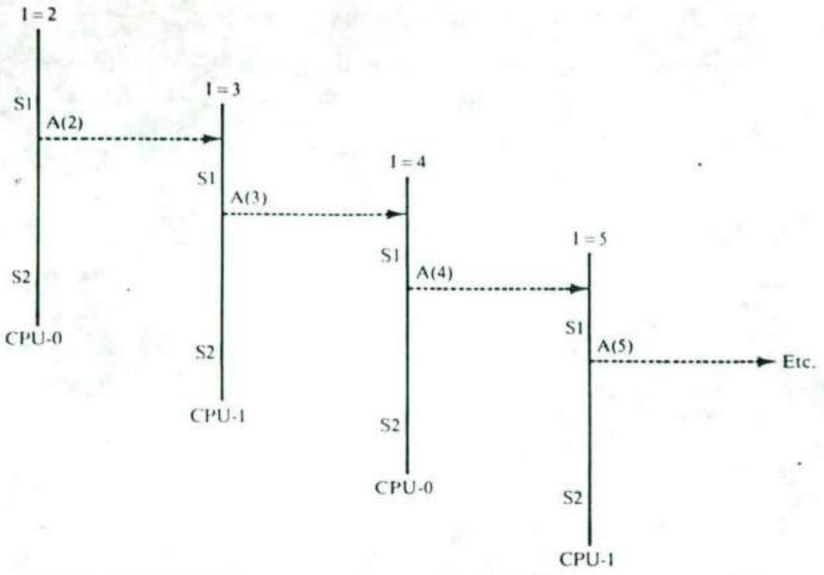


Figure 9.47 Multitasking by processor pipelining in Example 9.6.

Some benchmark timings on X-MP are reported by the Cray Research designers. Table 9.5 shows the timing on various vector loop families with respect to three representative vector lengths. The improvement of X-MP over Cray 1/S on typical vector loops ranges from 1.5 to 3.0 with respect to increasing vector lengths. It has been also reported that for general linear algebra (Fortran) benchmark programs, the speedup factor of X-MP over Cray 1/S ranges from 2.77 to 3.27.

Table 9.4 Overall performance of the Cray X-MP relative to Cray 1/S and disk

1-CPU rate	(Peak 210 megaflops)	
Minimum:	1 (Cray 1/S)	
Typical:	Scalar-dominated 1.5	
	Vector-dominated 2.0	
Maximum:	4	
2-CPU rate	(Peak 420 megaflops)	
Minimum:	2	
Typical:	Scalar-dominated 3.0	
	Vector-dominated 4.9	
Maximum:	4	
I/O rate	Access time	Transfer rate
Disk	1	1
SSD	0.01	250

Table 9.5 Vector loop families benchmark timing on X-MP over Cray 1/S

Vector loop families	Short vector (VL = 8)	Medium vector (VL = 128)	Long vector (VL = 1024)
$A = B$	1.1	1.8	2.1
$A = B + C$	1.2	2.2	2.7
$A = B \bullet C$	1.5	2.6	3.3
$A = B/C$	1.5	1.9	2.0
$A = B + C + D$	1.5	2.7	3.2
$A = B + C \bullet D$	1.4	2.9	3.6
$A = B + s \bullet D$	1.3	3.0	4.0
$A = B + C + D + E$	1.3	2.3	2.7
$A = B + C + D \bullet E$	1.6	2.5	2.9
$A = B \bullet C + D \bullet E$	1.3	2.5	3.1
$A = B + C \bullet D + E \bullet F$	1.5	2.1	2.2
Typical	1.5	2.5	3.0

Theoretically, we can roughly analyze the total speedup of X-MP over a scalar processor as follows: Vectorization offers a speedup of 10 to 20 over scalar processing, depending on actual code and vector length. Multitasking offers an additional speedup of $S_m \leq 2$, depending on task size and relative multitasking overhead. The total speedup over scalar processing is thus equal to $S = S_m \times (10$ to $20)$. In the benchmark, SPECTRAL, for short-term weather forecasting, the actual 2-CPU speedup has been measured as $S_m = 1.89$ over 1-CPU. Therefore, we obtain $S = 18$ to 38 under the assumption that $S_m = 1.8$ to 1.9.

We describe below a model developed by Ingrid Bucher (1983) to evaluate the performance of Cray X-MP in environments with different work loads. Work loads on the X-MP can be characterized by three types of execution requirements: *scalar* mode, *vector* mode, and *concurrent* mode. The scalar mode is characterized by a process code being executed sequentially either for reasons of logic or because it is too costly to vectorize. In the vector mode, a process code is vectorizable and executed in the vector section of the processor. In this mode the process granularity is small. In the concurrent mode, the process code is decomposed into cooperating processes and can be executed on more than one processor. The process granularity is large enough to overcome the communication and process creation overheads.

Let S_s, S_v, and S_c represent the rates at which a machine can execute scalar, vector, and concurrent codes, respectively. Also, let F_s, F_v, and F_c be the fractions of the work load that can be executed only in scalar, vector, and concurrent modes, respectively. Therefore the time required to execute this work load is proportional to

$$T = \frac{F_s}{S_s} + \frac{F_v}{S_v} + \frac{F_c}{S_c} \equiv \frac{1}{S_{eff}} \tag{9.1}$$

where S_{eff} is the work load–dependent effective speed of the machine. Equation 9.1 implies that the slowest of the execution speeds will critically influence the effective speed unless the fractional work load F associated with it is negligibly small.

The weight factors F_s, F_v, and F_c have to be determined empirically from the work load and adjusted by projections of how the work load will evolve in the future. In general, the choice of machine configurations may influence the characteristics of the work load. For example, a machine with high concurrent speed may encourage more Monte Carlo simulations. A satisfactory measure that is independent of machine architecture and compiler optimization and characterizes the amount of computational work done is desirable. A generally accepted metric for the execution speed of supercomputers is the megaflops. However, this metric does not include much of the work done. Examples of such work are the logical operations, integer arithmetic, and table hookups. Because of the highly parallel architecture of each processing unit, these operations may often be performed in parallel with the floating-point work. Nevertheless, we adopt the megaflops measure.

In the vector mode, the time required to perform operations on vectors of length N is a linear function of N given by

$$T = T_{\text{start}} + N\Delta \tag{9.2}$$

where T_{start} is the startup time for the vector operation and Δ is the time per result elements. For example, the Cray X-MP has shorter startup and element times for its vector operations than the Cray-1S and therefore clearly has the superior vector processor. The vector execution speed can be estimated by

$$S_v = \frac{N}{T} = \frac{1}{\Delta + T_{\text{start}}/N} \tag{9.3}$$

Note, however, that the average time to process a vector length N has a more complicated relationship than Eq. 9.2, because vectors of length $N > 64$ are stripmined in sections of length 64.

Not all vector operations follow the simple relationship shown in Eq. 9.2. For the Cray-1S, vectors must have a constant stride (distance between memory locations), but the stride need not be 1. However, for many repetitive operations, data are not stored in memory in a regular pattern. Hence we can identify these types of vector operations:

1. Vector operations for vectors stored in continuous locations
2. Operations for vectors stored with constant stride
3. Operations for vector operands stored in irregular locations in memory

Operands stored in irregular locations must be gathered, and the results may have to be scattered back into memory. The Cray-1S performs, gathers, and scatters in scalar mode only. Therefore, we can modify the execution rate in the vector mode by

$$\frac{1}{S_v} = \frac{F_1}{S_{v1}} + \frac{F_2}{S_{v2}} + \frac{F_3}{S_{v3}} \tag{9.4}$$

where S_{v1}, S_{v2}, and S_{v3} are the vector speeds for operands stored in continuous memory location, locations with constant stride > 1, and random locations, respectively. F_1, F_2, and F_3 are the corresponding fractions of the vectorizable work load.

The number of *loads* and *store* per floating-point operation greatly influences the vector speed. For some typical vector codes, measurements indicate that 0.6 to 1.0 *loads* and 0.2 to 0.5 *stores* were observed per floating-point operation. Therefore, the floating Fortran statement

$$\text{V1(I)} = \text{S1} * \text{V2(I)} + \text{S2} * \text{V3(I)} \tag{9.5}$$

produces the typical code. Further, two such statements are contained in a typical DO loop, reducing the startup time for the other loop per floating-point operation by a factor of 2 for the X-MP.

Table 9.6 contains values for vector speeds for a work load similar to that at the Los Alamos Laboratory. Table 9.7 indicates the values for a more ideal work load. Note the effective speeds are degraded by only small amounts due to the slow components. Measurements of the scalar speed S_s is performed by running a bench program. For example, the scalar speeds for Cray-1S and one of two processors of the Cray X-MP are 4.2 and 5.4, respectively.

For the asynchronous concurrent mode, there are communication overheads and portions of the parallel algorithm that must be executed sequentially. Let F_c be the fraction of code that can be executed in parallel mode on an arbitrary number of P processors, with the remainder F_{seq} of it to be executed sequentially. Then the execution time of the concurrent algorithm is proportional to

$$\frac{1}{S_c} = \frac{F_c}{P * S_s} + \frac{F_{seq}}{S_s} + P * T_{comm} \tag{9.6}$$

Table 9.6 Characteristic vector speeds (megaflops) for vector length = 100 and work load $F_s = 0.78$, $F_v = 0.20$, $F_c = 0.02$. (Courtesy of *ACM Sigmetrics*, Bucher 1983)

Machine	S_{v1} Vector speed for continuous vectors	S_{v2} Vector speed for constant stride	S_{v3} Vector speed for random access	S_v Effective vector speed
Cray-1S	58	56	5	46
Cray X-MP*	107	101	6	79

* One of two processors.

Table 9.7 Characteristic vector speeds (megaflops) for vector length = 500 and ideal work load $F_s = 0.85, F_v = 0.15, F_c = 0.00$. (Courtesy of *ACM Sigmetrics*, Bucher 1983)

Machine model	S_1 Vector speed for continuous vectors	S_2 Vector speed for constant stride	S_3 Vector speed for random access	S_v Effective vector speed
Cray-1S	66	65	5	66
Cray X-MP*	133	132	6	133

* One of two processors.

where S_s is the speed for executing the sequential portion of code and T_{comm} is the communication overhead. Using Eq. 9.1 and results described earlier, we can compile characteristic speeds of Cray-1S, Cray X-MP, and some hypothetical machines for two work loads with differing characteristics. In Tables 9.8 and 9.9, the machines in quotes are hypothetical machines and the numbers in parentheses are postulated numbers for hypothetical machines.

From these tables, it is obvious that the effective speed of a supercomputer is strongly work load–dependent. The slowest characteristic speed will affect the effective speed critically unless the fraction of the work load associated with that speed is negligibly small. It acts like a bottleneck. The most effective way to speed up a machine is to increase this speed or to decrease the fraction of work associated with it. The results also show that speeding up the fastest characteristic speed of a supercomputer will markedly improve its effective speed, only if the fraction of the work load running at that speed is close to 1. If this is not the case, the installation of additional vector pipelines on a vector computer will not be effective.

In summary, the Cray X-MP has 8 times Cray-1 memory bandwidth with guaranteed chaining of linked vector operations. Compared to Cray-1, the X-MP offers 1.25 to 3.75 speedup for single job and 2.5 to 5 times throughput on CPU

Table 9.8 Characteristic speeds (megaflops) for Cray-1S, Cray X-MP, and a hypothetical machine for vector length = 100, with work load $F_s = 0.20$, $F_v = 0.60$, and $F_c = 0.20$. (Courtesy of *ACM Sigmetrics*, Bucher 1983)

Machine	S_s	S_v	S_c	S_{eff}
Cray-1S	4.2	46	4.2	9.2
Cray X-MP (1 processor)	5.4	79	5.4	12.2
Cray X-MP (2 processors)	5.4	(158)	(10.8)	(16.8)
"Cray X-MP" (4 processors)	5.4	(316)	(21.6)	(20.7)

Table 9.9 Characteristic speeds (megaflops) for Cray-1S, Cray X-MP, and a hypothetical machine for vector length = 500, with work load $F_s = 0.10, F_v = 0.80$ and $F_c = 0.10$. (Courtesy of *ACM Sigmetrics*, Bucher 1983)

Machine	S_s	S_v	S_c	S_{eff}
Cray-1S	4.2	46	4.2	16.7
Cray X-MP (1 processor)	5.4	133	5.4	23.2
Cray X-MP (2 processors)	5.4	(266)	(10.8)	(32.5)
"Cray X-MP" (4 processors)	5.4	(532)	(21.6)	(40.6)

dominated job mix. The improvement in speed is due to two processors scheduled by the COS, shorter clock period, higher memory bandwidth, and guaranteed chaining. Like the Cray-1, the X-MP is good for both short and long vector processing. A general guideline to explore the computing power in X-MP is to partition tasks at the highest level to apply multitasking and then to vectorize tasks at the lower levels as much as possible.

The Cray-2 Cray Research, Inc. is currently developing the Cray-2, which is expected to have 6 times speedup in scalar and 12 times speedup in vector operations over the Cray-1. Cray-2 is planned to have four processors using 32 M words of main memory. The CPU cycle time is targeted to be 4 ns. The I/O will be improved 20 times from current Cray-1 capability. It has been suggested that 16-gate ECL chips will be used in Cray-2. Highly densed logic and memory modules will be cooled by immersion in inert fluorocarbon liquid. The longest wire length is confined to 16 in. The system is planned to be housed in a circular frame 38 inches in diameter and 26 inches in height, a rather condensed size for a supercomputer. The Cray-2 is expected to become available in the late 1980s.

9.7 BIBLIOGRAPHIC NOTES AND PROBLEMS

The original design of C.mmp architecture is described in Wulf et al. (1972). The final C.mmp architecture is described in Fuller and Harbison (1978). The experience of the C.mmp among other multiprocessors is presented in Jones and Schwartz (1980). Reliability of C.mmp has been studied in Siewiorek et al. (1978). The Hydra operating system is described in Wulf et al. (1981). Oleinick (1978) studied parallel algorithms for the C.mmp. Marathe and Fuller (1977) evaluated the C.mmp architecture and the Hydra kernel.

The S-1 multiprocessor has been reported in Widdoes (1980). Lawrence Livermore National Laboratory has published a series of reports on the S-1 project at the Lawrence Livermore National Laboratory (1981). IBM System/370 architecture was assessed in Case and Pedegs (1978). IBM 3033 and 3081 are described in IBM (1978, 1983). Programming and operating system design considerations of tightly coupled IBM multiprocessor system are given in Arnold et al. (1974) which includes an overview of the OS/VS2 MVS. Functional characteristics of 370/168 can be found in the IBM manual IBM (1979). System description of the Denelcor HEP computer is extracted from technical notes by Denelcor, Inc. (1983) and a report by Smith and Fink (1980). Jordan (1983) has studied the performance of the HEP. The material on the Cray X-MP and Cray-2 is based on Chen (1983) and some technical presentations by Cray Research, Inc. Bucher (1983) developed the performance models for Cray X-MP.

The evolution of the Sperry Univac 1100 series is reported in Borgerson et al. (1978). The detailed description of Univac 1100/80 systems can be found in several Sperry Univac manuals on the processor and storage, hardware system, programming, and executive systems. An introductory treatment of commercial multiprocessor systems, including the IBM 370/168, CDC Cyber-170, Honeywell Series 60 Level 66, Univac 1100/80, Burroughs B7700, and DEC System 10 Model KL-10, C.mmp, and C.m*, is given in Satyanarayanan (1980) plus an annotated bibliography up to 1979. Surveys of earlier multiprocessors can be found in Enslow (1974, 1977). A recent tutorial text on supercomputers by Hwang and Kuhn (1984) covers recent vector processors and multiprocessor systems. The Tandem Nonstop system is described in Katzman (1978). Recently, there are a number of research multiprocessors reported by Gajski et al. (1983), Gottlieb et al. (1983), and Fritsch et al. (1983).

Problems

9.1 Determine the evaluation time of the arithmetic expression

$$S = A[1]B[1] + A[2]B[2] + A[3]B[3] + A[4]B[4]$$

in each of the following computer systems:

(*a*) An SISD system with a general-purpose PE
(*b*) A multifunction SISD system with one adder and one multiplier
(*c*) An SIMD system with four PEs
(*d*) An MIMD system with four processors

The addition and multiplication require two and four time units, respectively. Memory-access time due to instruction and data fetch are ignored. The data-transfer time from one PE to another PE is assumed to be one time unit in SIMD and MIMD systems, whereas it is ignored in SISD and multifunction systems. In an SIMD system, the interconnection of PEs is in a linear circular fashion; i.e., each PE is connected to two neighboring PEs. In an MIMD, each PE can directly communicate with other PEs.

9.2 Three types of switch boxes are used to design a multistage interconnection network for an MIMD system. A square switch has two inputs and two outputs. An arbitration switch has two inputs and one output. A distribution switch has one input and two outputs. You are asked to design an 8×4 interconnection network using these switch modules.

(*a*) Use the minimum number of switch modules to construct the 8×4 network with a unique path between any source and any destination.

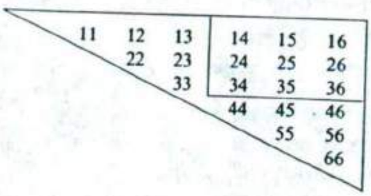
(*b*) Repeat (*a*) for a 4×8 network.

(*c*) There are many ways to construct a $2^m \times 2^n$ network, if $m > n$. Comment on the better choices among all the possible configurations in terms of hardware requirements and expected network performance.

9.3 Consider the storage of a symmetric n-by-n matrix $\mathbf{A} = (a_{ij})$ in a memory system with n parallel modules, where n is a perfect square. Devise storage schemes to satisfy each of the following requirements. It is assumed that only $[n/2]$ rows of the matrix are to be stored if n is odd, and $[n/2] + 1$ rows are needed if n is even. Each memory module has a separate index register to keep track of the allocation.

(*a*) It is required to access any row or any column in one memory cycle. For example, we want to access the elements a_{13}, a_{23}, and a_{33} in one cycle, or the elements a_{43}, a_{53}, a_{63}, a_{73}, a_{83}, and a_{93} in another cycle.

(*b*) It is required to access any row or any nonoverlapping square of blocks as in the example below:



(*c*) Illustrate your memory allocation schemes for (*a*) and (*b*) with $N = 9$ and $N = 16$, respectively. Is it possible to achieve (*a*) and (*b*) with the same scheme?

9.4 Answer the following questions associated with the C.mmp system:

(*a*) Explain the special system instructions HALT, RESET, WAIT, RTI, and RTT developed for the PDP-11 processors in the C.mmp.

(*b*) Explain the function of the *Dmap* and of the *interprocessor bus* installed in the C.mmp.

(*c*) What are the special features built into the *Hydra* operating system?

9.5 Answer the following questions on the S-1 multiprocessor project:

(*a*) Explain the use of separate data cache and instruction cache in the pipelined design of the S-1 Mark IIA uniprocessors.

(*b*) Explain the virtual-to-physical address translation scheme used in the S-1 system.

(*c*) What are the special features in the *Amber* operating system that facilitates multiprocessing?

9.6 Answer the following questions for the HEP multiprocessor:

(*a*) Distinguish between the conventional SISD pipelining and the MIMD pipelining introduced in the HEP.

(*b*) Explain the design and priority operations of the packet-switching interconnection network developed in the HEP.

(*c*) Explain the synchronization and protection mechanisms developed in the HEP.

9.7 Answer the following questions for the IBM multiprocessors:

(*a*) Distinguish the Attached Processing (AP) mode from the Multiprocessing (MP) mode in various IBM multiprocessors.

(*b*) What are the improvements of the IBM 3081 over the IBM 370/168MP and IBM 3033 in both technologies and designs?

(*c*) What are the multiprocessing features in the IBM OS/VS2 operating system?

9.8 Answer the following questions for the Univac 1100 series:

(*a*) Describe the increase of multiprocessing capability in various $M \times N$ configurations of the Univac 1100/80.

(*b*) What are the improvements made in Univac 1100/90 multiprocessors over the Univac 1100/80 models?

(*c*) Explain the functional structure of the kernel of the EXEC operating system.

9.9 Answer the following questions for the Tandem nonstop system:

(*a*) Why can the Tandem multiprocessor tolerate all single failures in the system?

(*b*) Explain the alternate path switching between devices and controllers in the Tandem/16.

(*c*) Describe the message system developed in the Guardian operating system.

9.10 Answer the following questions for the Cray X-MP system:

(*a*) Explain the inter-CPU communication structure in the Cray X-MP.

(*b*) Explain the functions of the Solid-State Storage Device (SSD) and of the I/O Subsystem (IOS) in the Cray X-MP.

(*c*) Explain the improvements made in the Cray X-MP over its predecessor, the Cray-1.

9.11 A computing system has 10 tape drives available. All jobs that run on the system require a maximum of four tape drives to complete, but we know that they start by running for a long period with only three; they request the one remaining drive for a short period when needed near the end of their operation. (There is an endless supply of these jobs.)

(*a*) If the job scheduler operates with the policy that it will not start a job unless there are four unassigned drives, and it assigns those four drives to a job for its entire duration, what is the maximum number of jobs that can be in progress at once? What are the minimum and maximum number of drives that may actually be idle as a result of this policy?

(*b*) Figure out a better scheduling policy to improve the drive utilization rate and at the same time to avoid a system deadlock. What is the maximum number of jobs that can be in progress in your new policy? What are the minimum and maximum numbers of drives that may be idle as a result of this policy?

9.12 *Chained vector time* (*chime*) is a useful term for discussing vector operation timing. Cray X-MP can combine several chimes implementable on a Cray-1 into a single long chime. Give an example vector computation sequence (more involved than that shown in Example 9.4) to show the advantages of using X-MP over the use of Cray-1.

CHAPTER

TEN

DATA FLOW COMPUTERS AND VLSI COMPUTATIONS

Computer architects have been constantly searching for new approaches to designing high-performance machines. Data flow and VLSI offer two mutually supportive approaches towards the design of future supercomputers. In this chapter, we study the requirements of data-driven computations, functional programming languages, and various data flow system architectures that have been challenged in recent years. In the VLSI computing area, we introduce topological structures of multiprocessor arrays for large-scale numeric computations and for symbolic manipulations. Techniques for directly mapping parallel algorithms into hardware structures will be studied. VLSI architectures for designing large-scale matrix arithmetic solvers are presented based on matrix partitioning and algorithmic decomposition. Potential applications of some of these VLSI computing structures are demonstrated for real-time image processing.

10.1 DATA-DRIVEN COMPUTING AND LANGUAGES

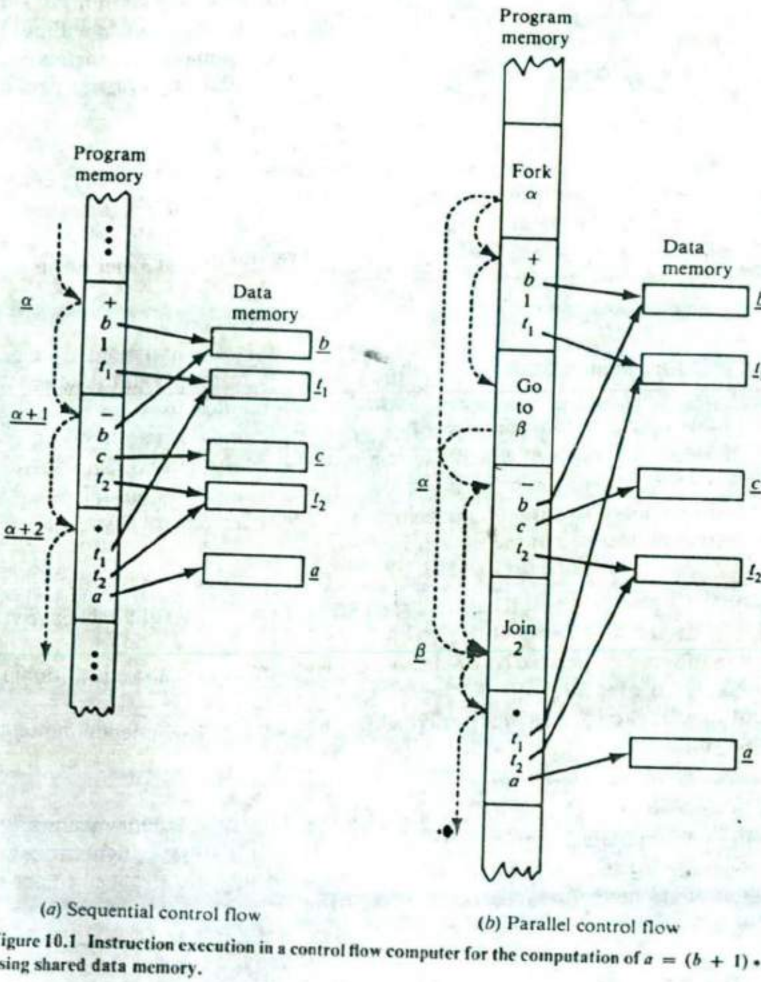
Data flow computers are based on the concept of *data-driven* computation, which is drastically different from the operation of a conventional von Neumann machine. The fundamental difference is that instruction execution in a conventional computer is under program-flow control, whereas that in a data flow computer is driven by the data (operand) availability. We characterize below these two types of computers. The basic structures of data flow computers and their advantages and shortcomings will be discussed in subsequent sections.

Jack Dennis (1979) of MIT has identified three basic issues towards the development of an ideal architecture for future computers. The first is to achieve a high performance/cost ratio; the second is to match the ratio with technological progress; and the third is to offer better programmability in application areas.

The data flow model offers an approach to meet these demands. The recent progress in the VLSI microelectronic area has provided the technological basis for developing data flow computers.

10.1.1 Control Flow vs. Data Flow Computers

The concepts of *control flow* and *data flow* computing are distinguished by the control of computation sequences in two distinct program representations. In the control flow program representations shown in Figure 10.1, the statement



(*a*) Sequential control flow (*b*) Parallel control flow

Figure 10.1 Instruction execution in a control flow computer for the computation of $a = (b + 1) * (b - c)$ using shared data memory.

$a = (b + 1) * (b - c)$ is specified by a series of instructions with an explicit flow of control. Shared memory cells are the means by which data is passed between instructions. Data (operands) are referenced by their memory addresses (variables). In Figure 10.1, solid arcs show the access to stored data, while dotted arcs indicate the flow of control.

In the traditional *sequential control flow* model (von Neumann), there is a single thread of control, as shown in Figure 10.1*a*, which is passed from instruction to instruction. Explicit control transfers are caused by using operators like GO TO. In the *parallel control flow* model (Figure 10.1*b*), special parallel control operators such as FORK and JOIN are used to explicitly specify parallelism. These operators allow more than one thread of control to be active at an instant and provide means for synchronizing these threads, as demonstrated in Figure 10.1*b*. All underlined variables refer to addresses of operands and instructions. Special features are identified below for either the sequential or parallel control flow model:

- Data is passed between instructions via references to shared memory cells.
- Flow of control is implicitly sequential, but special control operators can be used explicitly for parallelism.
- Program counters are used to sequence the execution of instruction in a centralized control environment.

In a data flow computing environment, instructions are activated by the availability of data tokens as indicated by the ( ) in Figure 10.2. Data flow programs are represented by directed graphs, which show the flow of data between instructions. Each instruction consists of an operator, one or two operands, and one or more destinations to which the result (data token) will be sent. Three snapshots of the data flow graph for $a = (b + 1) * (b - c)$ are shown in Figure 10.3. The dots correspond to data tokens being passed between instructions. Listed below are interesting features in the data flow model:

- Intermediate or final results are passed directly as data token between instructions.
- There is no concept of shared data storage as embodied in the traditional notion of a variable.
- Program sequencing is constrained only by data dependency among instructions.

Control flow computers have a *control-driven* organization. This means that the program has complete control over instruction sequencing. Synchronous computations are performed in control flow computers using centralized control. Data flow computers have a *data-driven* organization that is characterized by a passive examine stage. Instructions are examined to reveal the operand availability, upon which they are executed immediately if the functional units are available.

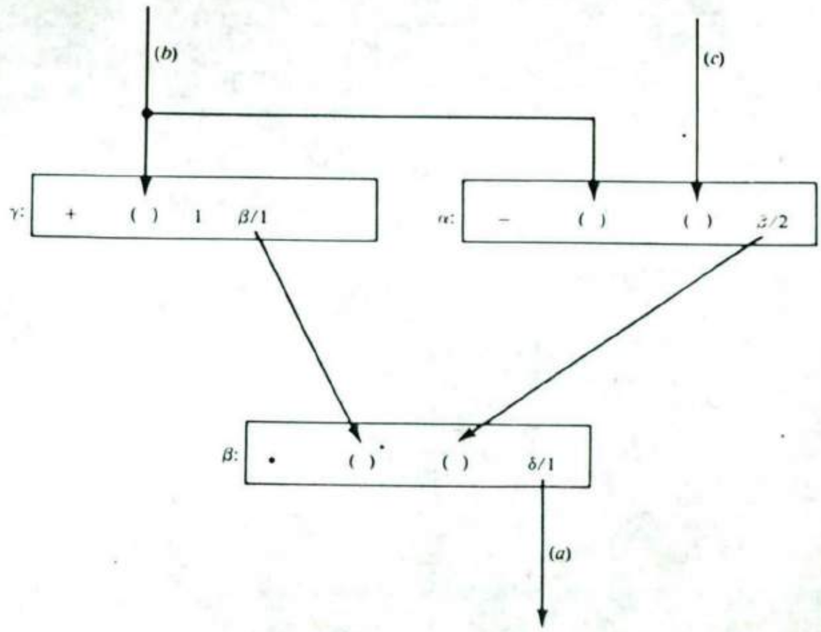


Figure 10.2 Instruction execution in a dataflow computer for the computation of $a = (b + 1) \cdot (b - c)$ by direct data forwarding.

This data-driven concept means *asynchrony*, which means that many instructions can be executed simultaneously and asynchronously. A high degree of implicit parallelism is expected in a data flow computer. Because there is no use of shared memory cells, data flow programs are free from side effects. In other words, a data flow operation is purely functional and produces no side effects such as the changes of a memory word. Operands are directly passed as "tokens" of values instead of as "address" variables. Data flow computations have no far-reaching effects. This locality of effect plus asynchrony and functionality make them suitable for distributed implementation.

Information items in a data flow computer appear as *operation packets* and *data tokens*. An operation packet is composed of the opcode, operands, and destinations of its successor instructions, as shown in Figure 10.2. A data token is formed with a result value and its destinations. Many of these packets or tokens are passed among various resource sections in a data flow machine. Therefore, the machine can assume a packet communication architecture, which is a type of distributed multiprocessor organization.

Data flow machine architectures Depending on the way of handling data tokens, data flow computers are divided into the *static model* and the *dynamic model*, as introduced in Figure 10.4 and Figure 10.5, respectively. In a static data flow

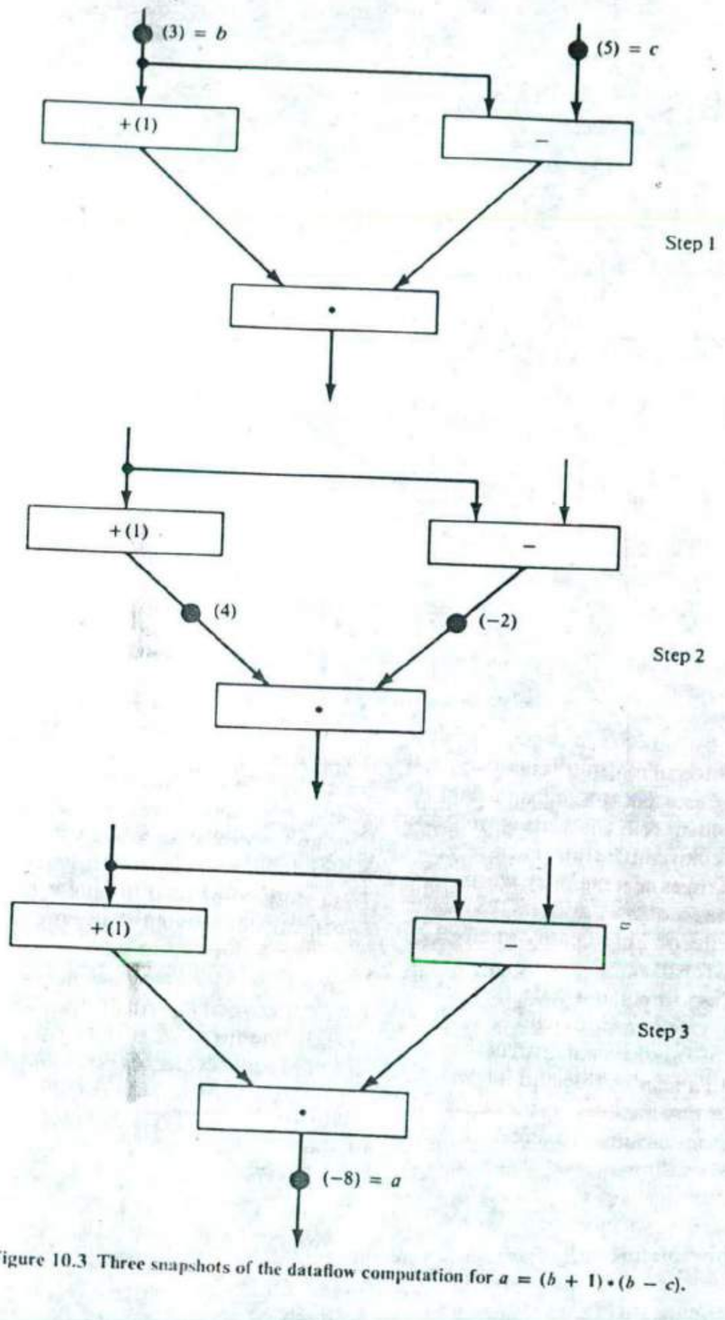


Figure 10.3 Three snapshots of the dataflow computation for $a = (b + 1) \cdot (b - c)$.

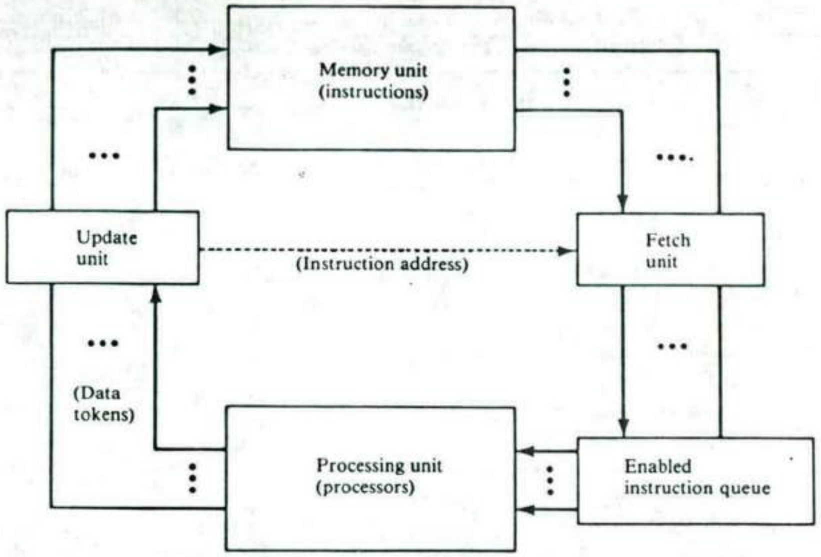


Figure 10.4 A static dataflow computer organization.

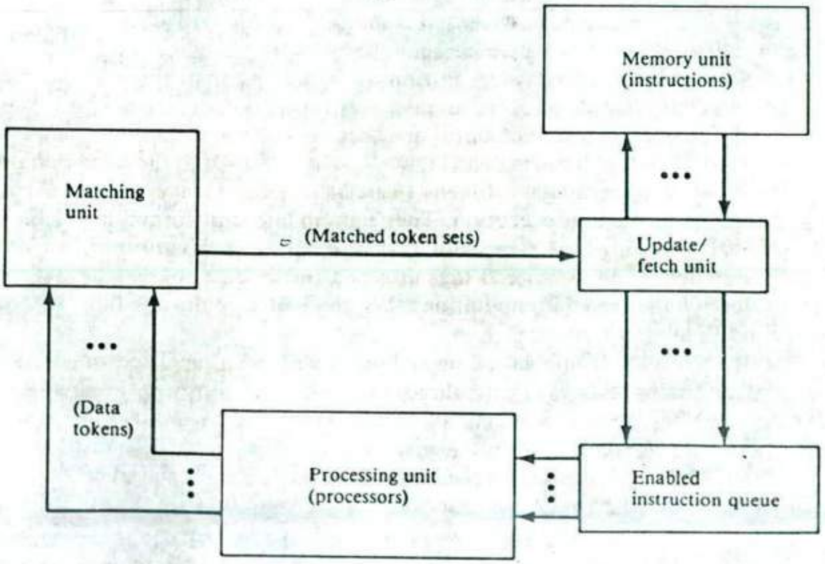


Figure 10.5 A dynamic dataflow computer organization.

machine, data tokens are assumed to move along the arcs of the data flow program graph to the operator nodes. The nodal operation gets executed when all its operand data are present at the input arcs. Only one token is allowed to exist on any arc at any given time, otherwise the successive sets of tokens cannot be distinguished. This architecture is considered static because tokens are not labeled and *control tokens* must be used to acknowledge the proper timing in transferring data tokens from node to node. Jack Dennis and his research group at the MIT Laboratory for Computer Science is currently developing a static data flow computer.

A dynamic data flow machine uses *tagged tokens*, so that more than one token can exist in an arc. The tagging is achieved by attaching a label with each token which uniquely identifies the context of that particular token. This dynamically tagged data flow model suggests that maximum parallelism can be exploited from a program graph. If the graph is cyclic, the tagging allows dynamically unfolding of the iterative computations. Dynamic data flow computers include the Manchester machine developed by Watson and Gurd at the University of Manchester, England, and the Arvinds machine under development at MIT, which is evolved from an earlier data flow project at the University of California at Irvine.

The two packet-communication organizations are based on two different schemes for synchronizing instruction execution. A point of commonality in the two organizations is that multiple processing elements can independently and asynchronously evaluate the executable instruction packets. In Figure 10.4, the data tokens are in the input pool of the update unit. This update unit passes data tokens to their destination instructions in the memory unit. When an instruction receives all its required operand tokens, it is enabled and forwarded to the enabled queue. The fetch unit fetches these instructions when they become enabled.

In Figure 10.5, the system synchronization is based on a matching mechanism. Data tokens form the input pool of the matching unit. This matching unit arranges data token into pairs or sets and temporarily stores each token until all operands are compared, whereupon the matched token sets are released to the fetch-update unit. Each set of matched data tokens (usually two for binary operations) is needed for one instruction execution. The fetch-update unit forms the enabled instructions by merging the token sets with copies sent to their consumer instructions. The matching of the special tags attached to the data tokens can unfold iterative loops for parallel computations. We shall further discuss this tagged-token concept in later sections.

Both static and dynamic data flow architectures have a pipelined ring structure. If we include the I/O, a generalized architecture is shown in Figure 10.6. The ring contains four resource sections: the *memories*, the *processors*, the *routing network*, and the *input-output unit*. The memories are used to hold the instruction packets. The processing units form the task force for parallel execution of enabled instructions. The routing network is used to pass the result tokens to their destined instructions. The input-output unit serves as an interface between the data flow computer and the outside world. For dynamic machines, the token matching is performed by the I/O section.

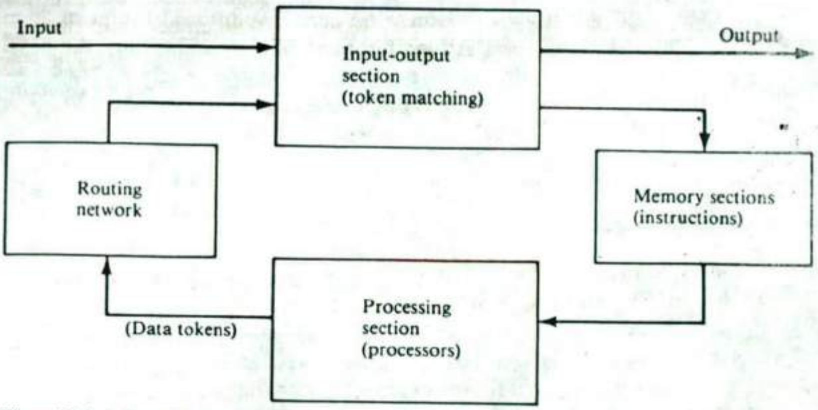

Input
Output
Input-output section (token matching)
Routing network
Memory sections (instructions)
Processing section (processors)
(Data tokens)

Figure 10.6 A ring-structured dataflow computer organization including the I/O functions.

Most existing data flow machine prototypes are built as an attached processor to a host computer, which handles the code translation and I/O functions. Eventually, computer architects wish to build stand-alone data flow computers. The basic ring structure can be extended to many improved architectural configurations for data flow systems. For example, one can build a data flow system with multiple rings of resources. The routing network can be divided into several functionally specialized packet-switched networks. The memory section can be subdivided into cell blocks. We shall describe variants of data flow computer architecture in Section 10.2.

Major design issues Toward the practical realization of a data flow computer, we identify below a number of important technical problems that remain to be solved:

1. The development of efficient data flow languages which are easy to use and to be interpreted by machine hardware
2. The decomposition of programs and the assignment of program modules to data flow processors
3. Controlling and supporting large amounts of interprocessor communication with cost-effective packet-switched networks
4. Developing intelligent data-driven mechanisms for either static or dynamic data flow machines
5. Efficient handling of complex data structures, such as arrays, in a data flow environment
6. Developing a memory hierarchy and memory allocation schemes for supporting data flow computations
7. A large need for user acquaintance of functional data flow languages, software supports, data flow compiling, and new programming methodologies
8. Performance evaluation of data flow hardware in a large variety of application domains, especially in the scientific areas

Approaches to attack the above issues and partial solutions to some of them will be presented in subsequent sections. We need first to understand the basic properties of data flow languages. After all, the data flow computers are language-oriented machines. In fact, research on data flow machines started with data flow languages. It is the rapid progress in VLSI that has pushed the construction of several hardware data flow prototypes in recent years.

10.1.2 Data Flow Graphs and Languages

There is a need to provide a high-level language for data flow computers. The primary goal is to take advantage of implicit parallelism. Data flow computing is compatible with the use of dependence graphs in program analysis for compiling in a conventional computer. An efficient data flow language should be able to express parallelism in a program more naturally, to promote programming productivity, and to facilitate close interactions between algorithm constructs and hardware structures. Examples of data flow languages include the Irvine Dataflow (ID) language and the Value Algorithmic Language (VAL) among several *single assignment* and *functional programming* languages that have been proposed by computer researchers

In a maximum parallel program, the sequencing of instructions should be constrained only by data dependencies and nothing else. Listed below are useful properties in a data flow language. We shall describe their implications and usages separately.

- Freedom from side effects based on functional programming
- Locality of effect without far-reaching data dependencies
- Equivalence of instruction-sequencing constraints with data dependencies
- Satisfying single-assignment rule with aliasing
- Unfolding of iterative computations into parallelism
- Lack of "history sensitivity" in procedures calls

Data flow graphs In a conventional computer, program analysis is often performed at compile time to yield better resource utilization and code optimization and at run time to reveal concurrent arithmetic logic activities for higher system throughput. For an example, we analyze the following Fortran program:

Example 10.1

1. $P = X + Y$	must wait for inputs X and Y
2. $Q = P \div Y$	must wait for instruction 1 to complete
3. $R = X \times P$	must wait for instruction 1 to complete
4. $S = R - Q$	must wait for instructions 2 and 3 to complete
5. $T = R \times P$	must wait for instruction 3 to complete
6. $U = S \div T$	must wait for instruction 4 and 5 to complete

Permissible computation sequences of the above program on a serial computer include the following five:

(1, 2, 3, 4, 5, 6)
(1, 3, 2, 5, 4, 6)
(1, 3, 5, 2, 4, 6)
(1, 2, 3, 5, 4, 6)
(1, 3, 2, 4, 5, 6)

On a parallel computer, it is possible to perform these six operations (1, [2 and 3 simultaneously], [4 and 5 simultaneously], 6) in three steps instead of six steps.

The above program analysis can be represented by the data flow graph shown in Figure 10.7. A *data flow graph* is a directed graph whose nodes correspond to operators and arcs are pointers for forwarding data tokens. The graph demonstrates sequencing constraints (consistent with data dependencies) among instructions. In a data flow computer, the machine level program is represented by data flow graphs. The firing rule of instructions is based on the data availability.

Two types of links on data flow graphs are depicted in Figure 10.8. The purpose is to distinguish those for numerical data from those for boolean variables.

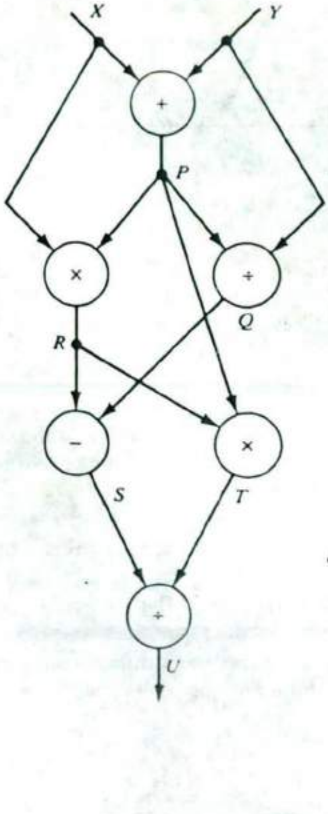


Figure 10.7 A dataflow graph for the computation of $U = f(X, Y) = (X \times (X + Y) - (X + Y) \div Y) \div (X \times (X + Y) \times (X + Y))$.

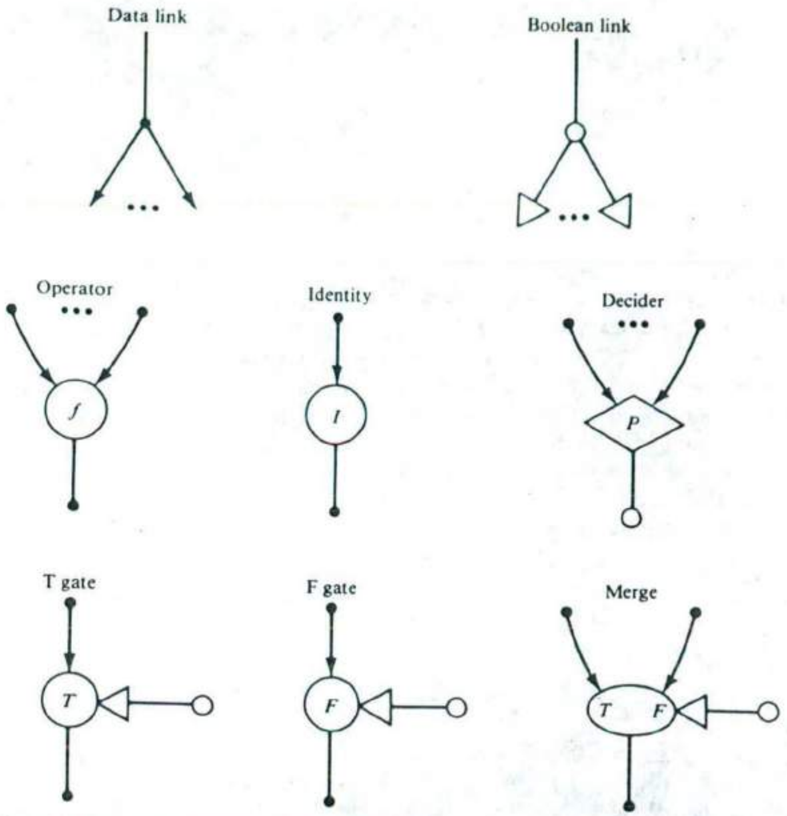


Figure 10.8 Operators (nodes) and links (arcs) for the construction of dataflow graphs.

Numerical data links transmit integer, real, or complex numbers and boolean links carry only boolean values for control purposes. Figure 10.8 presents various operator types for constructing data flow graphs. An *identity* operator is a special operator that has one input arc and transmits its input value unchanged. *Deciders*, *gates*, and *merge* operators are used to represent conditional or iterative computation in data flow graphs. A decider requires a value from each input arc and produces the truth value resulting from applying the predicate P to the values received.

Control tokens bearing boolean values control the flow of data tokens by means of the T gates, the F gates, and the merge operators. A T gate passes a data token from its data input arc to its output arc when it receives the true value on its control input arc. It will absorb a data token from its data input arc and place nothing on its output arc if it receives a false value. An F gate has similar behavior, except the sense of the control value is reversed. A merge operator has T and F data input arcs and a truth-value control arc. When a truth value is received, the merge actor places the token from the true input arc on its output arc. The token on the other unused input arc is discarded. Similarly, the false input is passed to the output, when the control arc is false.

Example 10.2 The following iterative computation is represented by the data flow graph in Figure 10.9, using some of the operators and graph links specified in Figure 10.8. The integer power $z = x^n$ of an input number x is desired:

```
input x,n
  y=1;i=n
  while i>0 do
    begin y=y*x;i=i-1 end                    (10.1)
  z=y
output z
```

The successive values assumed by the loop variables y and i pass through the links labeled in the program graph. The decider emits a token carrying the true value each time execution of the loop body is required. When the firing of the decider yields a false, the value of y is routed to the output link z. Note the presence of tokens carrying false values on the input arcs of the merge operators. These tokens allow the merge operator to initiate execution of the loop by passing initial values for the loop variables. The initial values of the control token are marked as *false* in Figure 10.9.

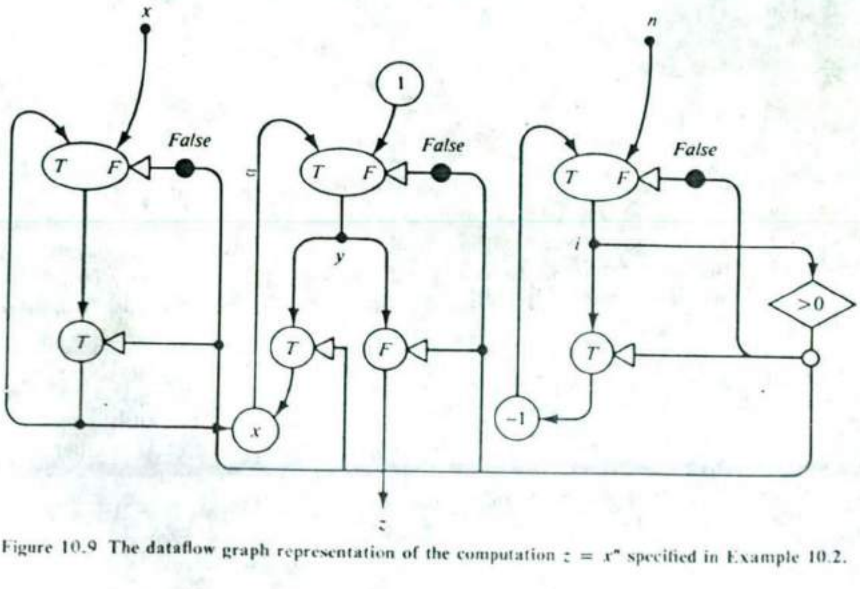


Figure 10.9 The dataflow graph representation of the computation $z = x^n$ specified in Example 10.2.

Data flow graphs form the basis of data flow languages. We briefly describe below some attractive properties of data flow languages.

Locality of effect This property can be achieved if instructions have no unnecessary far-reaching data dependencies. In Algol or Pascal, blocks and procedures provide some locality by making assignment only to local variables. This means that global assignments and common variables should be avoided. Data flow languages generally exhibit considerable locality. Assignment to a formal parameter should be within a definite range. Therefore, block structures are highly welcome in a data flow language.

Freedom from side effects This property is necessary to ensure that data dependencies are consistent with sequencing constraints. Side effects come in many forms, such as in procedures that modify variables in the calling program. The absence of global or common variables and careful control of the scopes of variables make it possible to avoid side effects. Another problem comes from the aliasing of parameters. Data flow languages provide "call by value" instead of the "call by reference." This essentially solves the aliasing problem. Instead of having a procedure modify its arguments, a "call by value" procedure copies its arguments. Thus, it can never modify the arguments passed from the calling program. In other words, inputs and outputs are totally isolated to avoid unnecessary side effects.

Single assignment rule This offers a method to promote parallelism in a program. The rule is to forbid the use of the same variable name more than once on the left-hand side of any statement. In other words, a new name is chosen for any redefined variable, and all subsequent references are changed to the new name. This concept is shown below:

$$\begin{array}{lcl} X := P - Q & & X := P - Q \\ X := X \times Y & \rightarrow & X1 := X \times Y \\ W := X - Y & & W := X1 - Y \end{array} \qquad (10.2)$$

The statements on the right are made to satisfy the single assignment rule. It greatly facilitates the detection of parallelism in a program. Single assignment rule offers clarity and ease of verification, which generally outweighs the convenience of reusing the same name. Single assignment rule was first proposed by Tesler and Enea in 1968. It has been applied in developing the French data flow computer LAU and the Manchester data flow machine in England.

Unfolding iterations A programming language cannot be considered as effective if it cannot be used to implement iterative computations efficiently. Iterative computations are represented by "cyclic" data flow graphs, which are inherently sequential. In order to achieve parallel processing, iterative computations must be unfolded. Techniques have been suggested to use tagged tokens to unfold activities embedded in a cyclic data flow graph. In a conventional machine, the same operation must be sequentially performed in successive iterations.

We define each distinct evaluation of an operator as an *activity*. Unique names can be created by tagging repeated evaluations of the same operator in different iterations. A tagged token is formed with two parts: ⟨data, destination activity name⟩. The destination activity is a unique name which identifies the context in which a code block is invoked, the code block name, the instruction number within the code block, and the iteration number associated with the looping index value. This tagged information will unfold iterative computations in a recursive manner because the context tag may be itself an activity name.

Arvind and Gostelow (1978) have proposed a *U-interpreter*, which is part of the data flow language ID. The U-interpreter is used to unfold iterations. This notion of labeling computational activities, coupled with some rules for manipulating the activity names, should help to enhance the parallelism in programs written in any functional or data flow language. Special hardware structures are needed to exploit the practical benefits in designing a data flow architecture with tagged tokens. Such a dynamic architecture (Figure 10.5) needs to match the tags of many data tokens before they can be paired to enable parallel computations asynchronously.

10.1.3 Advantages and Potential Problems

Pros and cons of data flow computers are discussed in this section. Due to their strong appeal to parallelism, data flow techniques have attracted a great deal of attention in recent years. We assess first the advantages and then examine the opposition opinions. The development of data flow computers and languages is still in its infancy stage. The research community is currently divided in opinions. It is too early to draw a final conclusion because the success of data flow computing depends heavily on high technology and its matching with applications. However, it is commonly recognized that more research and development challenges should continue in this area. The idea of data-driven computation is rather old, but only in recent years have architectural models with anticipated performance been developed.

Most advantages are claimed by researchers in this area. The claimed advantages were only partially supported by performance analysis and simulation experiments. Operational statistics are not available from existing prototype data flow machines. Therefore, some of the claimed advantages are still subject to further verification. Data flow computers are advantageous in many respects over the traditional von Neumann machines. These aspects are elaborated below in terms of projected performance, matching technology, and programming productivity.

Highly concurrent operations Parallelism can be easily exposed in a data flow program graph. The data flow approach offers a possible solution to the problem of efficiently exploiting concurrency of computation on a large scale. It benefits not only regularly structured but also arbitrary parallelism in programs. The direct use of values instead of names of value containers (addresses) enables purely functional programming without side effects. Asynchronous parallelism can be exploited at the instruction level or at the procedure level. Inherently sequential computations can be unfolded to enable parallelism. The data flow approach has applied pipelining, array processing, and multiprocessing techniques discussed in previous chapters for control flow computers.

The data flow language does not introduce instruction-sequencing constraints other than the ones imposed by data dependencies in the algorithm. In theory, maximum parallelism can be achieved if sufficient resources are provided. This approach extends naturally to an arbitrary number of processors in the system. The speedup should be linearly proportional to the increase of processor number. The high concurrency in a data flow computer is supported by easier program verification, better modularity and extendability of hardware, reduced protection problems, and superior confinement of software errors.

Matching with VLSI technology Recall the basic architecture of a data flow computer (Figure 10.6). The memory section contains instruction cells which can be uniformly structured in large-scale memory arrays. The pool of processing units and the network of packet switches can be each also regularly structured with modular cells. All this homogeneity and modularity in cellular structures contributes to the suitability of VLSI implementation of major components in a data flow computer. As introduced in Chapter 1, the impressive progress in microelectronics technology has made it possible to challenge the fabrication of large arrays of processors, memories, and switches on VLSI chips.

The interconnection between chips can be built into highly densed packaging systems. It is fair to say that data flow machine architecture matches nicely with the technological supports that we anticipate to have. The potential of VLSI and VHSIC technologies can be fully exploited in the development of data flow machines. The operations in a data flow computer may be asynchronous. However, the hardware components can be designed with synchronous functional pipes and clocked memory and switch arrays. With more lessons to be learned and the data flow hardware properly evaluated, it would be appropriate to consider VLSI implementation of some large-scale data flow systems.

Programming productivity In a control flow vector processor or a multiprocessor system, the percentage of code that can be vectorized ranges from 10 to 90 percent across a broad range of scientific applications. The nonvectorizable code (scalar operations) tends to become a bottleneck. Automatic vectorization requires sophisticated data flow analysis, which is difficult in Fortran because of the side effects caused by the global scope and aliasing of variables. A well-designed data flow computer should be able to overcome these difficulties and to remove the bottleneck caused by assorted scalar operations.

It has been claimed by many computer researchers that functional programming languages will increase the software productivity as compared to the imperative languages like Fortran and Pascal. This is especially true when the computing environment demands a high degree of parallel processing to achieve a prespecified level of performance. Intuitively, this assertion is valid for certain algorithm constructs and work-load distributions. However, more empirical results are needed to prove its validity for general scientific computations.

Shortcomings of data flow computing Critics of the data flow approach have pointed out quite a number of potential problems in the development and application of data flow computers at the instruction level. It is instructional to learn from these reserved positions and to explore other alternatives to achieve high performance. In the conventional computer with centralized control hardware, an imperative language such as Fortran is used and an intelligent compiler is needed to normalize the program and to generate the dependence graph, which guides the vectorization and optimization processes we have studied in previous chapters. High-level use of the dependence graph is practiced here primarily at compile time. The major advantages of this high-level approach are summarized below:

- There is no need to use a new functional programming language, which ordinary programmers may be reluctant to learn.
- Existing software assets for vectorizing, compiling, and dedicated application software can continue to be utilized.
- Data flow analysis at the higher level of procedures and loops will result in less overhead when averaged over all instructions to be executed.

In the data flow approach, special functional programming languages must be used which can be easily compiled into a dependence graph. The object code is generated to efficiently map the dependence graph onto the data flow machine with distributed control hardware. Some advantages of high-level data flow machines become potential shortcomings in the data flow computer, which exploits parallelism at the lowest level of instruction execution. Apparent disadvantages of instructional-level data flow computers are summarized below:

1. The data driven at instruction level causes excessive pipeline overhead per instruction, which may destroy the benefits of parallelism. The long pipeline filling problem is attributed to queueing all enabled instructions at the input ports of every subsystem in the data flow ring. The queue lengths absorb some of the parallelism in a program, thus, performance becomes weak for improper buffering and traffic congestion.
2. Data flow programs tend to waste memory space for the increased code length due to the single assignment rule and the excessive copying of data arrays. The damaging effects of the memory access conflict problem are so far not well addressed by data flow researchers.
3. When a data flow computer becomes large with high numbers of instruction cells and processing elements, the packet-switched network used becomes cost-prohibitive and a bottleneck to the entire system.

4. Some critics feel that data flow has a good deal of potential in small-scale or very large-scale parallel computer systems, with a raised level of control. For medium-scale parallel systems, data flow competes less favorably with the existing pipeline, array, and multiprocessor computers. We shall further discuss this assessment in Section 10.2.3.

10.2 DATA FLOW COMPUTER ARCHITECTURES

Several interesting data flow computer architectures are studied in this section. The intent is to identify related architectural concepts rather than to describe the implementation details. We shall start with static data flow computers represented by the Dennis machine at MIT. Then we describe several ring-structured, dynamic data flow computers, including the original Irvine machine and its successor machine at MIT, the EDDY system in Japan, and the Manchester machine in England. Several specially designed data flow systems will be briefly examined, including the Utah machine, the French LAU system, and the Newcastle data-control flow computer. Design alternatives of data flow computers will also be discussed to inspire future development.

10.2.1 Static Data Flow Computers

Jack Dennis and his associates at MIT have pioneered the area of data flow research. They have developed a static data flow computing model and the associated language supports. There are two interesting data flow projects at MIT. For identification purpose, we distinguish them by calling them the Dennis machine and the Arvind machine. The Dennis machine has a static architecture, whereas the Arvind machine is dynamic, using tagged tokens and colored activities.

Data flow graphs used in the Dennis machine must follow a static execution rule that only one data token can occupy an arc at an instant. This leads to a static firing rule that an instruction is enabled if a data token is present on each of its input arcs and no token is present on any of its output arcs. Thus, the program graph contains control tokens as well as data tokens, both contributing to the enabling of an instruction. These control tokens act as acknowledge signals when data tokens are removed from output arcs.

The Dennis machine is designed to exploit the concurrency in programs represented by static data flow graphs. The structure of this data flow computer is shown in Figure 10.10; it consists of five major sections connected by channels through which information is sent in the form of discrete tokens (packets):

- *Memory* section consists of instruction cells which hold instructions and their operands.
- *Processing* section consists of processing units that perform functional operations on data tokens.

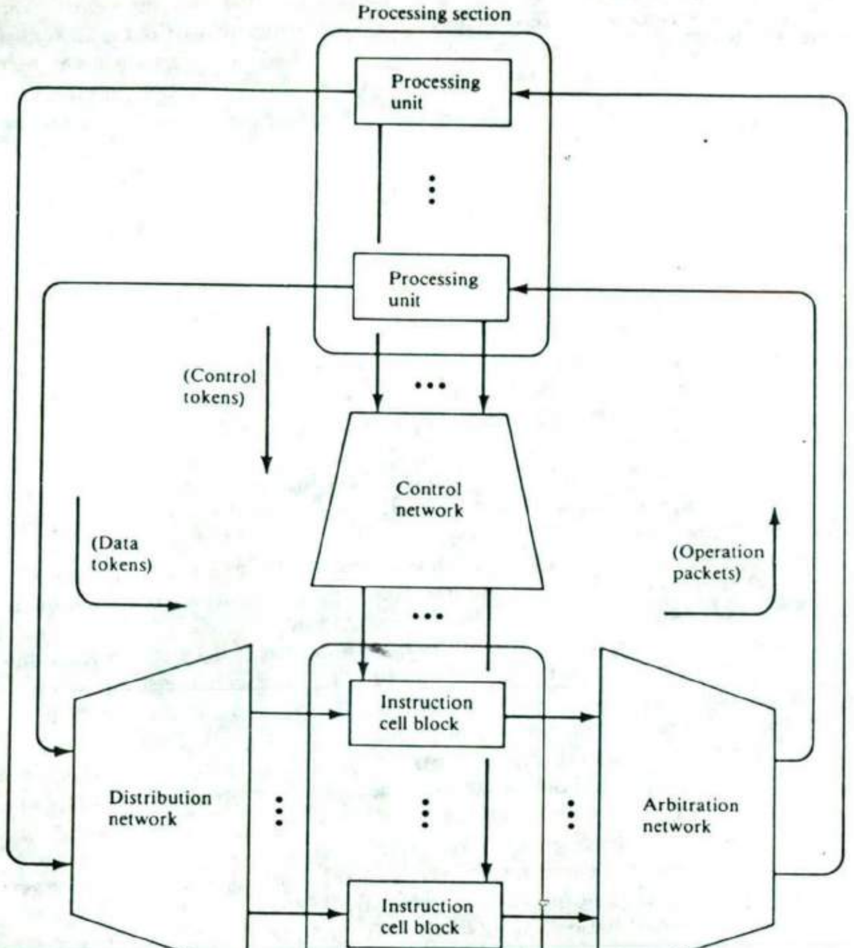


Figure 10.10 The static dataflow computer architecture proposed at MIT. (Courtesy of Dennis et al., 1979.)

- *Arbitration* network delivers operation packets from the memory section to the processing section.
- *Control* network delivers a control token from the processing section to the memory section.
- *Distribution* network delivers data tokens from the processing section to the memory section.

Instructions held in the memory section are enabled for execution by the arrival of their operands in data tokens from the distribution network and control tokens from the control network. Enabled instructions, together with their operands, are sent as operation packets to the processing section through the arbitration network. The results of instruction execution are sent through the distribution network and the control network to the memory section, where they become operands of other instructions. Each instruction cell has a unique address, the cell identifier. An occupied cell holds an instruction consisting of an operation code and several destinations. Each destination contains a destination address, which is a cell identifier, and additional control information used by processing units to generate result tokens. An instruction represents one or more operators of the program graph, together with its output links. Instructions are linked together through destination addresses stored in their destination fields.

Each instruction cell contains receivers which await the arrival of token values for use as operands by the instruction. Once an instruction cell has received the necessary operand tokens and acknowledge signals, the cell becomes enabled and sends an operation packet consisting of the instruction and the operand values to the appropriate processing unit through the arbitration network. Note that the acknowledge signals are used to correctly implement the firing rule for program graphs.

The arbitration network provides a path from each instruction cell to each processing unit and sorts the operation packets among its output ports according to the operation codes of the instructions they contain. For each operation packet received, a processing unit performs the operation specified by the instruction using the operand values in the packet and produces one or more result tokens, which are sent to instruction cells through the control network and distribution network. Each result token consists of a result value and a destination address derived from the instruction being processed by the processing unit. There are control tokens containing boolean values or acknowledge signals, which are sent through the control network, and data packets containing integer or complex values, which are sent through the distribution network.

The two networks deliver result tokens to receivers of instruction cells as specified by their destination address fields; that is, data packets are routed according to their destination address. The arrival of a result token at an instruction cell either provides one of the receivers of the cell with an operand value or delivers an acknowledge signal; if all result tokens required by the instruction in the cell have been received, the instruction cell becomes enabled and dispatches its contents to the arbitration network as a new operation packet.

The functions performed by the processing unit are distributed among several sections of the data flow processor. The operations specified by instructions are carried out in the processing section, but control of instruction sequencing is a function of the control network, and the decoding of operation codes is partially done within the arbitration network. The address fields (destination addresses) of instructions specify where the results should be sent instead of addressing a shared memory cell. Instead of instructions fetching their operands, the operand values are sent to the instructions.

All communication between subsystems in the Dennis machine is by packet transmission over the channels. The transmission of packets over each channel uses an asynchronous protocol so that the five sections of the computer can operate independently without using central timing signals. Systems organized to operate in this manner are said to have the packet communication architecture.

The instruction cells are assumed to be physically independent, so at any time many of them may be enabled. The arbitration network should be designed to allow many instruction packets to flow through it concurrently. Similarly, the control network and the distribution network should be designed to distribute dense streams of control and data packets back to the instruction cells. In this way, both the appetites of pipelining and parallelism are satisfied. The arbitration, distribution, and control networks of the data flow processor are examples of packet-switched routing networks that perform the function of directing packets to many functional units of the processor. If the parallelism represented in the data flow graph is to be fully exploited, routing networks must have a high bandwidth.

When the number of instruction cells becomes large, the three networks shown in Figure 10.10 may become exceedingly large and thus cost prohibitive. One approach that has been suggested to overcome this difficulty is to use the concept of *cell blocks*. A cell block is a collection of instruction cells which share the same set of input and output ports from the distribution, control, and arbitration networks. The cell-block implementation and its use in the machine architecture are demonstrated in Figure 10.11. By using shared I/O ports, the arbitration networks can be partitioned into subnetworks of significantly smaller sizes; so can the other networks in the system.

In Figure 10.12, we show an example design of the cell block-structured data flow multiprocessor system. The system consists of four processors and 32 cell blocks. Three building blocks can be used to construct the 32×4 arbitration network and the 4×32 distribution network. Illustrated in Figure 10.13 are the 2×1 arbiter, the 1×2 distributor, the 2×2 switch, and the 3×2 switch used in the network constructions in Figure 10.12 and Figure 10.14, respectively. The distributors are blocking free. The arbiter can pass only one input to its output. The 2×2 switch has nine possible states, two of which may cause blocking. The 3×2 switch has 27 states, 14 of which will cause blocking. Whenever a blocking takes place, only one of the conflicting requests can get through the switch.

The blocked requests will be discarded in an unbuffered network and the requests will be resubmitted. For a packet-switched network with buffers at the inputs of the switches or arbiters, the blocked requests will be held waiting to be passed to the output ports in a later time. The buffered Delta networks discussed in Chapter 7 can be modified to be used in a data flow environment. An example buffered arbitration network of size 27×8 is shown in Figure 10.14. Each input

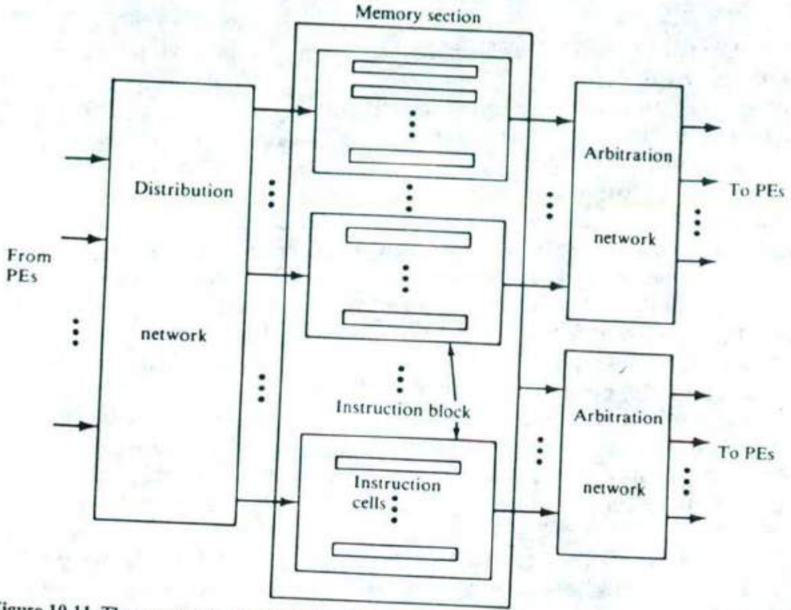


Figure 10.11 The concept of grouping instruction cells into cell blocks.

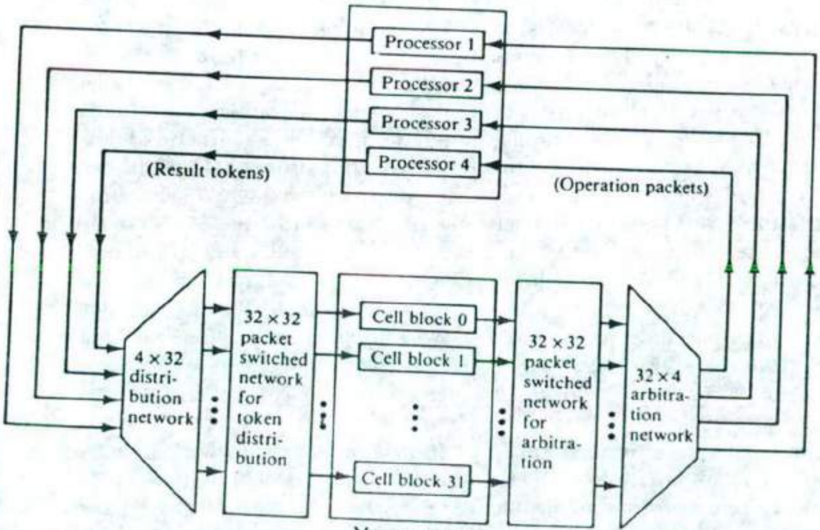
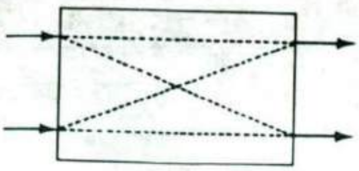
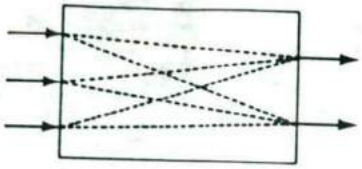
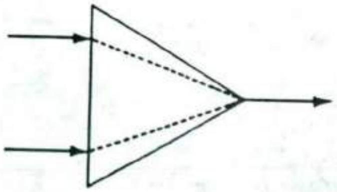

Processor 1
Processor 2
Processor 3
Processor 4
(Result tokens)
(Operation packets)
4 × 32 distribution network
32 × 32 packet switched network for token distribution
Cell block 0
Cell block 1
Cell block 31
32 × 32 packet switched network for arbitration
32 × 4 arbitration network
Memory store

Figure 10.12 A 4-processor dataflow computer with 32 cell blocks interconnected by a 32 × 4 arbitration network and a 4 × 32 distribution network.


(a) A 2-by-2 switch


(b) A 3-by-2 switch


(c) A 2-by-1 arbiter

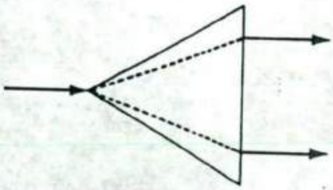

(d) A 1-by-2 distributor

Figure 10.13 Switches, arbiters, and distributers used in network construction (dash lines are all the possible request paths).

port of the 3 × 2 switch has a buffer. A round-robin scheme can be used to resolve conflicts among multiple requests destined to the same output ports.

Since the arbitration network has many inputs, a serial format is appropriate for packet transfer between instruction cells (or cell blocks) and the arbitration network to reduce the number of connections needed. However, to achieve a high rate of packet flow at the output ports, a parallel format is required. For this

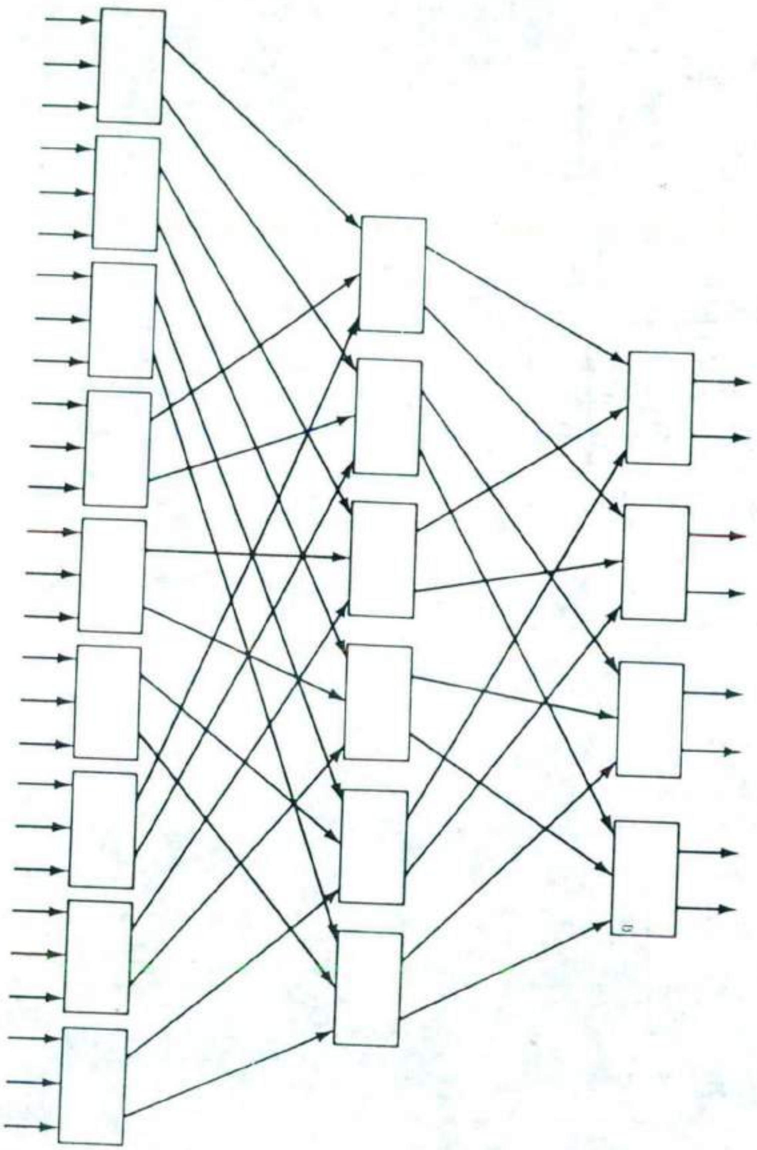


Figure 10.14 A 27-by-8 buffered Delta network for resource arbitration in a Delta network.

reason, serial-to-parallel conversion is done within the buffers as a packet travels through the arbitration network. Parallel-to-serial conversion is performed in the distribution network for similar reasons. The control network is usually unbuffered with direct circuit-switching paths.

The data flow project led by Dennis at MIT is undergoing four stages of development:

1. The construction of a four PE prototype machine to support language concepts of scalar variables, conditionals, and iterations used in signal processing
2. The extension of the scalar data flow machine to a vector/scalar processor to support data structures for scientific computations and to develop a user programming language
3. The overcoming of the program size limitation of stages one and two by adding cache memories and backup storage areas for inactive programs
4. The building of a general-purpose data flow computer which stands alone, compiles its own program, and runs a time-sharing service to multiple users

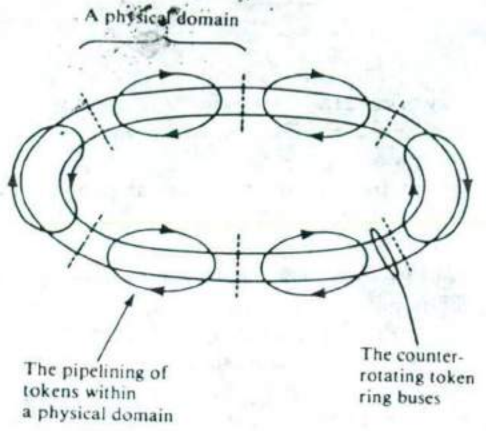
Currently, the prototype hardware is under construction and a compiler is being written for the VAL programming language. A number of supportive projects on fault tolerance, data flow hardware description languages, etc., are also in progress at MIT's Laboratory for Computer Science.

10.2.2 Dynamic Data Flow Computers

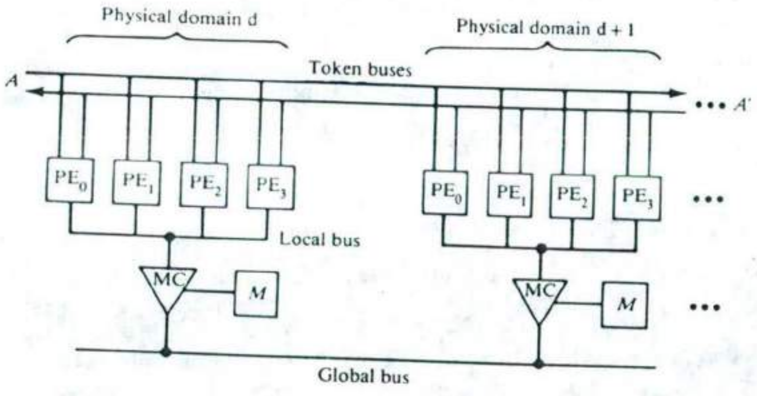
Three dynamic data flow projects are introduced below. In dynamic machines, data tokens are *tagged* (labelled or colored) to allow multiple tokens to appear simultaneously on any input arc of an operator node. No control tokens are needed to acknowledge the transfer of data tokens among instructions. Instead, the matching of token tags (labels or colors) is performed to merge them for instructions requiring more than one operand token. Therefore, additional hardware is needed to attach tags onto data tokens and to perform tag matching. We shall first present the Arvind machine, followed by the EDDY system and the Manchester data flow machine.

The development of the Irvine data flow machine was motivated by the desire to exploit the potential of VLSI and to provide a high-level, highly concurrent program organization. This project originated at the University of California at Irvine and now continues at the Massachusetts Institute of Technology by Arvind and his associates. The architecture of the original Irvine machine is conceptually shown in Figure 10.15. The ID programming language was developed for this machine. This machine has not been built; but extensive simulation studies have been performed on its projected performance.

The Irvine machine was proposed to consist of multiple PE clusters. All PE clusters (physical domains) can operate concurrently. The physical domains are interconnected by two system buses. The token bus is a pair of bidirectional


A physical domain
The pipelining of tokens within a physical domain
The counter-rotating token ring buses

(*a*) Token rings and local pipelining



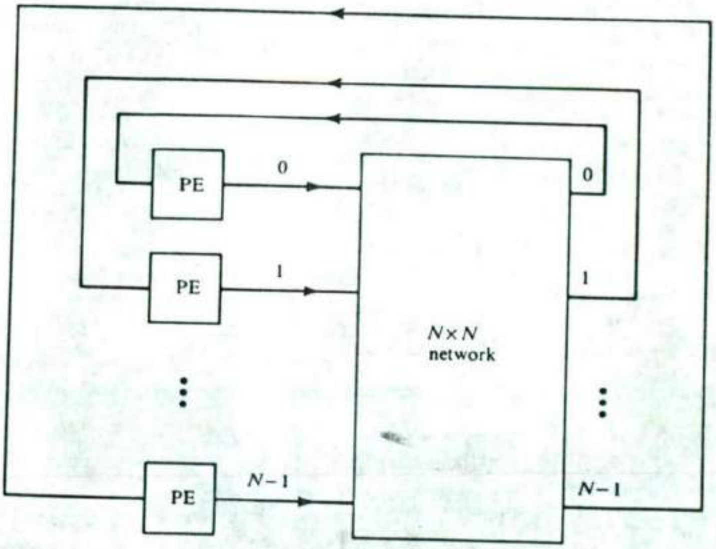
(*b*) Processor clusters and interconnection buses

Figure 10.15 The Irvine dataflow computer. (Courtesy of *IEEE Trans. Computers*, Gostelow and Thomas, October 1980.)

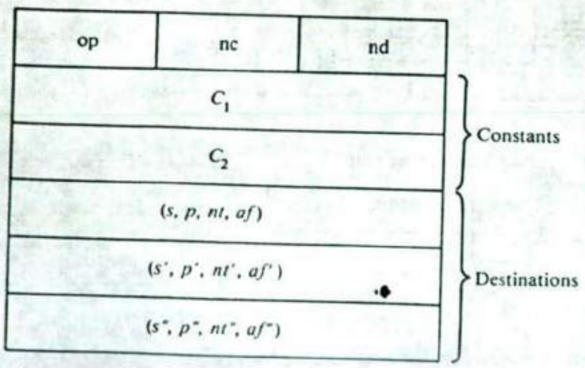
shift-register rings. Each ring is partitioned into as many slots as there are PEs and each slot is either empty or holds one data token. Obviously, the token rings are used to transfer tagged tokens among the PEs.

Each cluster of PEs (four PEs per cluster, as shown in Figure 10.15) shares a local memory through a local bus and a memory controller. A global bus is used to transfer data structures among the local memories. Each PE must accept all tokens that are sent to it and sort those tokens into groups by activity name. When all input tokens for an activity have arrived (through tag matching), the PE must execute that activity. The U-interpreter can help implement iterative or procedure computations by mapping the loop or procedure instances into the PE clusters for parallel executions.

The Arvind machine at MIT is modified from the Irvine machine, but still based on the ID language. Instead of using token rings, the Arvind machine has chosen to use an $N \times N$ packet switch network for inter-PE communications as demonstrated in Figure 10.16*a*. The machine consists of N PEs, where



(*a*) The machine architecture



(*b*) A typical instruction

Figure 10.16 Arvind's dataflow machine organization and instruction format. (Courtesy of Arvind et al., Sept. 1980.)

each PE is a complete computer with an instruction set, a memory, tag-matching hardware, etc. Activities are divided among the PEs according to a mapping from tags to PE numbers. Each PE uses a statistically chosen assignment function to determine the destination PE number.

A general format for instructions is shown in Figure 10.16*b*, where *op* is the opcode, *nc* is the number of constants (maximum of two) stored in the instruction, and *nd* is the number of destinations for the result token. Each destination is identified by four fields (*s*, *p*, *nt*, *af*); where *s* is the destination address, *p* indicates the input port at the destination instruction, *nt* indicates the number of tokens needed to enable the destination instruction, and *af* indicates the assignment function to be used in selecting the PE for the execution of the destination instruction.

The functional structure of each PE is shown in Figure 10.17. The input section has a register which if empty can accept a token either from the communication system or from the output section of the same PE. Each activity requires

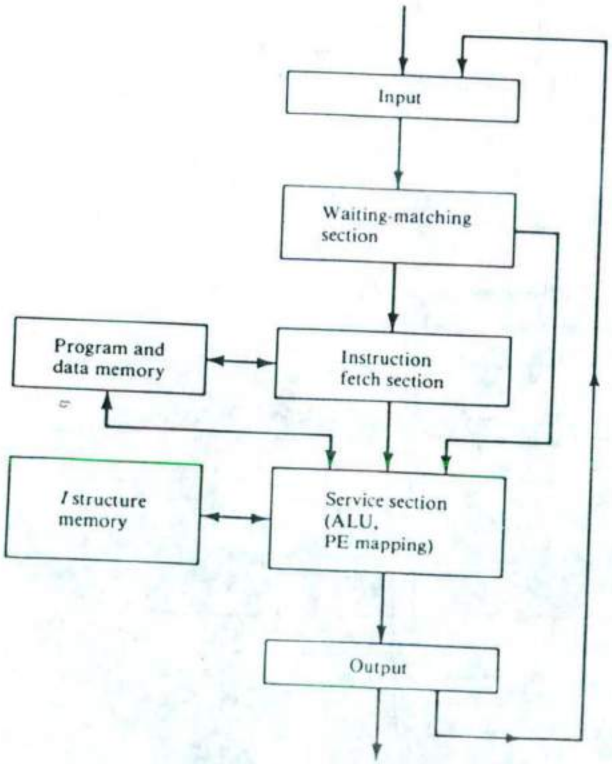


Figure 10.17 The processing element (PE) in Arvind's dataflow machine at MIT.

either one or two tokens, as indicated by the *nt* field of a token. If the activity corresponding to the input token requires another token, the waiting-matching section is informed. The latter has a buffer to hold those tokens for which another token with a matching tag has not yet arrived. Whenever the tags of two tokens match, both tokens are moved to the instruction-fetch buffer. Based on the statement number part of the tag, an instruction from the local program memory is fetched.

If a match for the tag of the input token is not found and the waiting-matching buffer is full, a refusal to accept the token will cause a deadlock. Therefore, if the buffer-full condition exists, the token has to be stored somewhere else and retrieved at a later time. After the instruction has been fetched, an operation packet containing the operation code, the operands, and the destinations is formed and sent to the service section. The service section contains a floating-point ALU and the hardware to calculate new activity names and destination PE numbers.

The *I structure* is a special tagged memory for storing arraylike data structures with constraints on their creation and access. Essentially, an element of an I structure can be defined only once. A presence bit is associated with every element of an I structure. An attempt to read an element whose presence bit is not set causes the read to be deferred. The use of I structures can avoid excessive array copying. The service section also processes the memory operations except I structure reads. After the ALU or the memory produces the result and the new tags and destination PE numbers have been computed, the result tokens are sent to the output section. Since it is possible to encounter delays in transmitting a token through the communication network, some buffer space is provided in the output section.

A separate section to hold deferred reads (i.e., requests to read an element of an I structure before it has been produced) is needed to avoid the blocking of the service section. Every unsuccessful read request will be marked and set aside. An unusual feature of the PE is that it has no program counter. Instead, it maintains a list of enabled activities in the service section and can execute them in any order. A PE will have 100 percent ALU utilization as long as there is at least one enabled activity in the service queue at any given time instant.

A group of PEs known as a *physical domain* is allocated whenever a procedure or a loop is invoked. All activities of the invoked procedure (or loop) take place within the physical domain except those activities which are caused by an operator that changes the context part. The activities of a procedure (loop) can be distributed within a physical domain on the basis of the instruction number of the iteration number of an activity name. Tag matching may contribute to additional overhead, which is potentially a performance bottleneck for dynamic data flow computers.

In order to include the possibility of activating several code blocks (not necessarily distinct from each other) within a physical domain, one can assign a different *color* to each activation. However, only a finite number of colors are allowed within a physical domain and, if all colors of a physical domain are in use, no new loop or procedure activation can be scheduled on it. Colors are released when a loop or procedure terminates. Sharing of code blocks within a physical domain is feasible because all invocations carry different colors. Several color registers are used in each PE. This will help distribute logical activities to PEs on a resource-sharing basis. Coloring will increase resource utilization and thus total system throughput.

In Japan, the development of a scientific data flow machine called EDDY (Experimental system for Data Driven processor arraY) is in progress. The hardware consists of 4 × 4 PEs and two broadcast control units (BCUs), as shown in Figure 10.18. Each PE is composed of two microprocessors (Z8001) and is connected directly with eight neighboring PEs. The broadcast control can load or unload programs and data to or from all PEs, in column or row, at the same time.

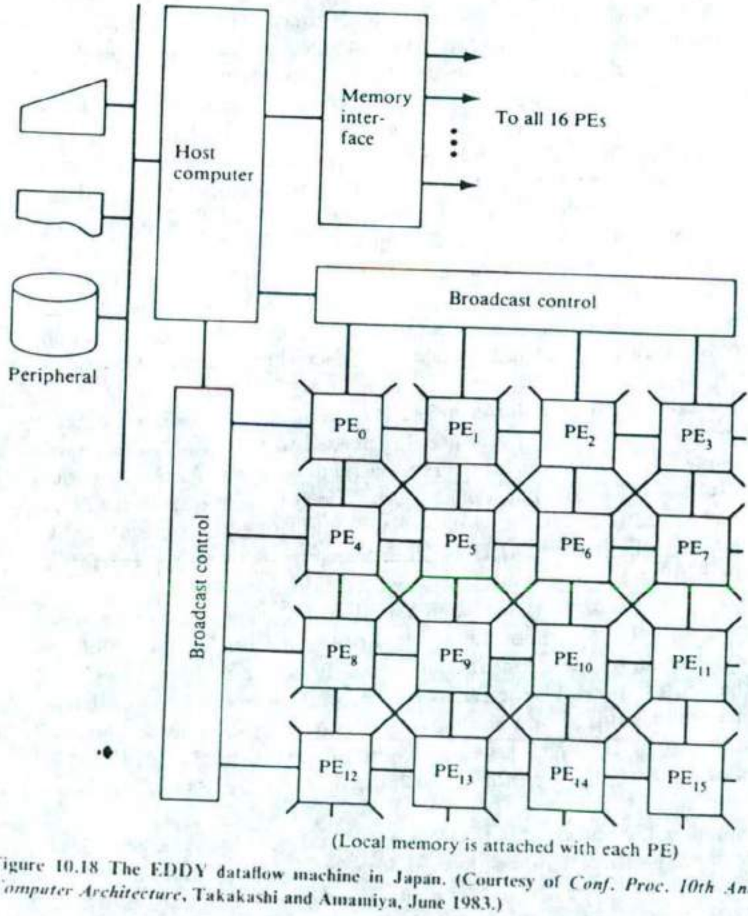


Figure 10.18 The EDDY dataflow machine in Japan. (Courtesy of *Conf. Proc. 10th Annual Symp. Computer Architecture*, Takakashi and Amamiya, June 1983.)

The software system implemented on each Z8001 "simulates" the circular pipeline data flow control of the PE in detail, using logical simulation clocks. It generates statistical data, such as operation rates of the function units and average queue length. The simulation results will be used to help develop the custom-designed PE hardware. The functional language to be used in EDDY is called VALID.

The custom-designed PE is a circular pipeline consisting of an instruction memory section, operand memory section, operation section and communication section, as shown in Figure 10.19. On the arrival of each operand token, the instruction memory fetches its operation node and sends both the fetched instruction and operand data to the operand memory section. If the arrived data is an operand for a two-operand operation, the operand memory searches for its paired partner associatively. When the paired operand is found, an operation packet is constructed and sent to the operation section. When no paired operand is found, the arrived data token is stored in the operand memory with a key attached.

All data tokens are tagged, which represents their execution environment so as to allow more than one token to be travelling on an arc. In order to realize highly distributed control, it is important to decentralize the tagging control. It is also necessary to mechanize the tagging control so as to extract maximal parallelism. The EDDY system uses a mixed strategy with both *static* tagging and *dynamic* tagging. In static tagging, execution environments are predefined and a

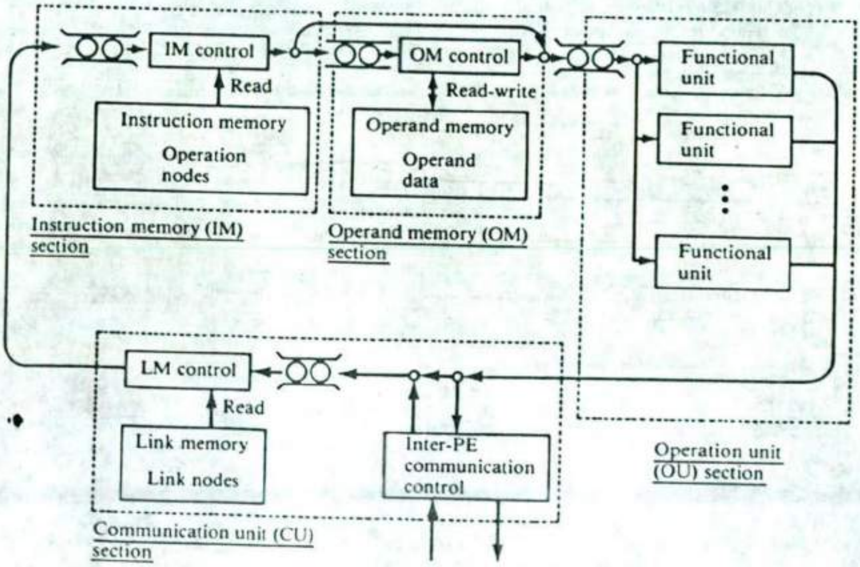


Figure 10.19 The functional design of each PE in the EDDY system.

unique tag is assigned to each of them. In dynamic tagging, tags are assigned to each environment dynamically. An execution environment is represented by a (tag) name.

The environment name and opcode are used as a key for the associative search. The operation section consists of several functional units including some number crunchers. The communication section consists of final link memory and an inter-PE communication controller. This controller sends result packets to link memory or to other PEs and also receives result packets from other PEs and transmits them to its own link memory.

The data flow project at Manchester University has included the design of the high-level, single-assignment programming language Lapse; the implementation of translators for Lapse and a subset of Pascal; and the production of a detailed stimulator for the Manchester compute architecture. Currently the group is completing a 20-processor data flow computer prototype.

The Manchester machine also assumes a ring structure, as demonstrated in Figure 10.20. Five functional blocks communicate in clockwise direction around a ring. A token package is the main unit of information and comprises a data-value, label, and destination node pointer. The *matching unit* groups tokens. When sufficient tokens arrive to fire a node, an appropriate group package finds a destination node description in the *node store*. An executable package [containing operator, operands, label, and pointer(s) to further destination node(s)] is sent to the processing unit for execution. The *switch* handles external input output. The *token queue* saves excess tokens generated at about the same time.

Parallel data-driven rings have been suggested to extend the Manchester machine (Figure 10.21). Very high processing rates may be achieved by connecting multiple numbers of pipelined rings in this approach. A unidirectional pipelined exchange switch is modularly extensible and of relatively simple form as compared

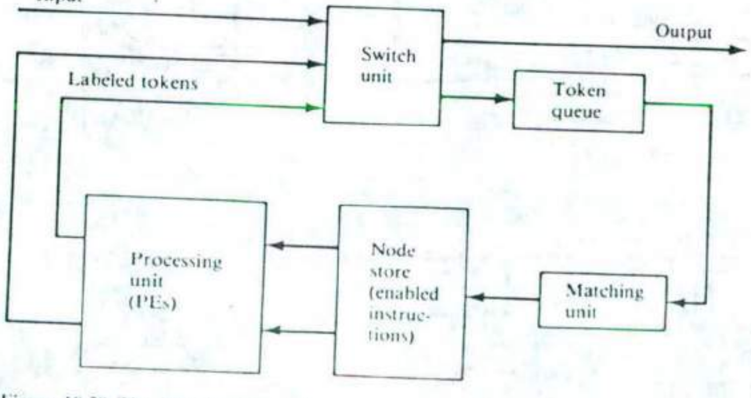


Figure 10.20 The Manchester dataflow computer organization. (Courtesy of AFIPS *Proc. of NCC*, Watson and Gurd, June 1979.)

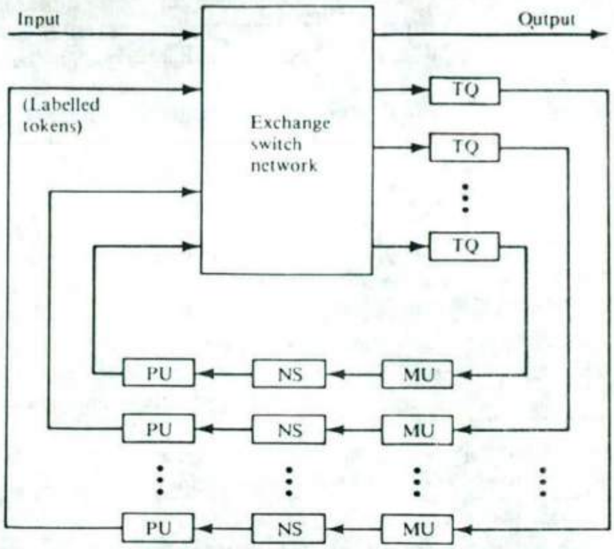


Figure 10.21 Multiple ring architecture proposed for the Manchester machine (TQ: Token Queue; MU: Matching Unit; NS: Node Store; PU: Processing Unit). (Courtesy of *Computer Design*, Gurd and Watson, July 1980.)

to a crossbar switch, for example. In such a very large system, the major problem is to distribute the work load evenly among multiple data flow rings demonstrated in the figure.

10.2.3 Data Flow Design Alternatives

There are several data flow projects that have special architectural approaches different from the static or dynamic machines described in previous sections. In fact, the first operational data flow machine in the USA is the Data-Driven Machine (DDM-1) designed by A. Davis and his colleagues in 1976. The program and machine organization are based on the concept of recursion, markedly different from the previous data flow systems we have examined. The computer is composed of a hierarchy of computing elements (processor-memory pairs), where each element is logically recursive and consists of further offspring elements.

Logically the DDM architecture is tree structured, with each computing element being connected to a parent element (above) and up to eight offspring elements (below), which it supervises. The DDM project is currently located at the University of Utah. It is operational and communicates with a DEC-20/40, which is used for software support of compilers, simulators, and performance measurement programs.

In Toulouse, France, a data flow system called LAU has been constructed with 32 bit-slice microprocessors interconnected by multiple buses. The LAU programming language is based on single assignment rule, but the computer's program organization is based on control flow concepts. In the computer, data are passed via sharable memory cells that are accessed through addresses embedded in instructions. Seperate control signals are used to enable instructions.

In Newcastle, England, another data flow system is under development. This system uses both data tokens and control tokens to enable instruction execution. The Newcastle system is a combination of the data flow and control flow mechanisms in one integrated system approach.

Up to 1983, only the DDM at Utah, the EDDY in Japan, the Manchester machine, and the French LAU system are operational data flow computers. There are many other research projects that are devoted to various aspects of data flow computing. Most data flow projects emphasize run-time simultaneity at the instruction level. Unless the program being executed is embedded with a high degree of parallelism, the performance of such instruction-level data-driven computers could be very poor because of high system overhead in detecting the parallelism and in scheduling the available resources. Two design alternatives to the data flow approach are discussed below. These modified approaches offer higher machine compatibility and better utilization of the existing software assets.

Dependence-driven approach This approach was independently proposed by Gajski et al. (1982), and by Motooka et al. (1981). The idea is to raise the level of parallelism to *compound-function* (procedure) level at run time. A compound function is a collection of computational tasks that are suitable for parallel processing by multiprocessors. Listed below are six compound functions investigated by the research group at the University of Illinois:

- Array (vector matrix) operations
- Linear recurrence
- For-all loops
- Pipeline loops
- Blocks of assignment statements
- Compound conditional statements

A program is a dependence graph connecting the compound-function nodes. *Dependence-driven* refers to the application of data flow principles over multiple compound-function nodes. In a sense, it is procedure driven. Instead of using data flow languages, traditional high-level language programs can be used in this dependence-driven approach (Figure 10.22). Additional program transformation packages should be developed to convert ordinary language programs to dependence graphs and to generate codes from dependence graphs. A hardware organization needed for dependence-driven computations is illustrated in Figure 10.23. Such a dependence-driven machine has a global controller for multiple processor clusters and shared memory and other resources, instead of decentralized control as emphasized in a data-driven machine.

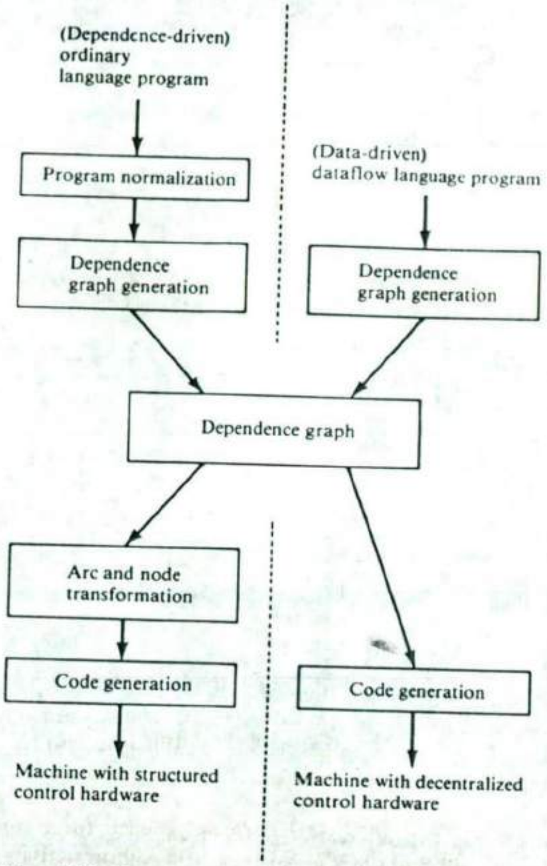

(Dependence-driven) ordinary language program
Program normalization
Dependence graph generation
(Data-driven) dataflow language program
Dependence graph generation
Dependence graph
Arc and node transformation
Code generation
Code generation
Machine with structured control hardware
Machine with decentralized control hardware

Figure 10.22 Comparison between data-driven and dependence-driven computing models. (Courtesy of IEEE *Computer*, Gajski et al., February 1982.)

A second opinion has been expressed on data flow machines and languages by the dependence-driven researchers. Two questions were raised: First, are data flow languages marketable? To date, the high-speed computer market has been dominated by conservatism and software compatibility. Can data flow languages, as currently proposed, overcome this conservatism? Second, will data flow languages enhance programmer productivity? Although data flow researchers have made some claims to this effect, they remain unsubstantiated.

Dependence-driven researchers felt that, in small-scale parallel systems, data flow principles have been successfully demonstrated. When simultaneity is low, irregular, and run time–dependent, data flow might be the architecture of choice. In very large-scale parallel systems, data flow principles still show some potential

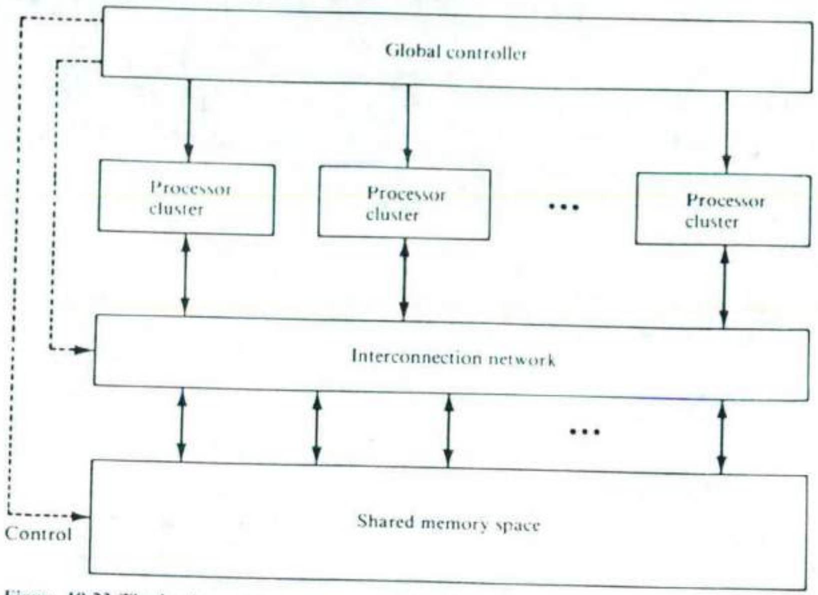


Figure 10.23 The hardware structure suggested for high-level data flow computing, for either the dependence-driven model or the event-driven model.

for high-level control. It is in medium-scale parallel systems that data flow has little chance of success. Pipelined, parallel, and multiprocessor systems are all effective in this range. For data flow processing to become established here, its inherent inefficiencies must be overcome.

Multilevel event-driven approach The dependence-driven was generalized to an event-driven approach by Hwang and Su (1983c). An *event* is a logical activity which can be defined at the job (program) level, the procedure level, the task level, or at the instruction level after proper *abstraction* or *engrossment*, as illustrated in Figure 10.24. Hierarchical scheduling is needed in this event-driven approach. A mechanism for program abstraction needs to be developed. Such a mechanism must not require high system overhead. It could be implemented partially at compile time and partially at run time. The choice depends on the performance criteria to be used in promoting parallel processing at various levels. Hierarchical scheduling of resources is the most challenging part of research in this approach.

Heuristic algorithms are needed for scheduling multiple events to the available resources in an event-driven computer. Instead of using the first-in, first-out (FIFO) scheduling policy on all enabled activities, as in a data-driven computer, this approach considers the use of *priority queues* in the event scheduling. The priority is determined by pre-run-time estimating of the time-space complexities of all enabled events. The optimal mapping of logical events to physical resources has

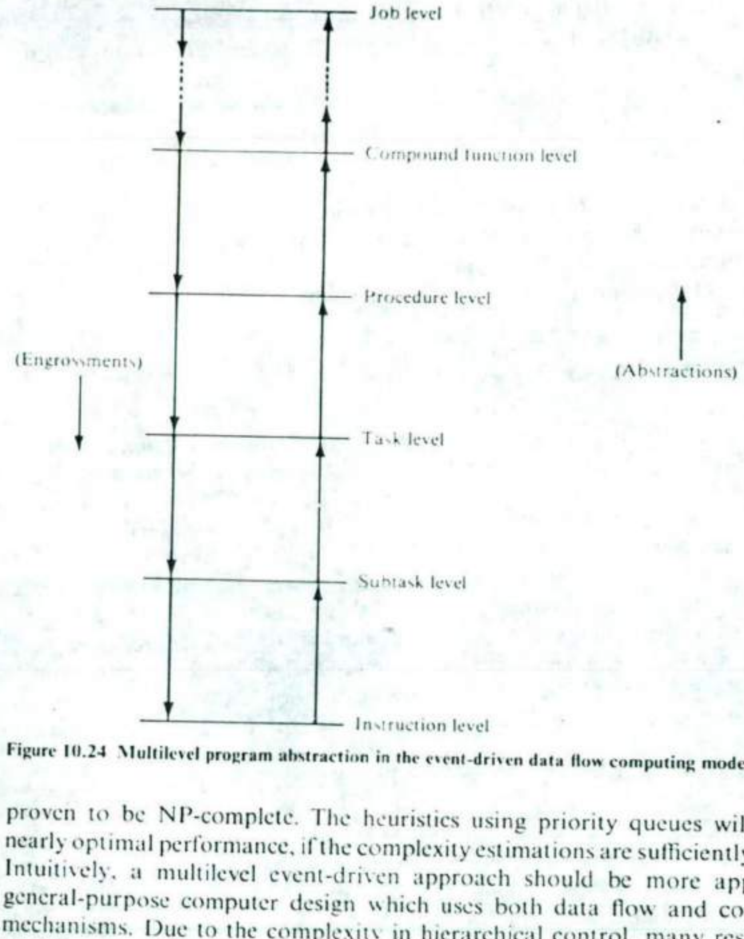


Figure 10.24 Multilevel program abstraction in the event-driven data flow computing model.

proven to be NP-complete. The heuristics using priority queues will result in nearly optimal performance, if the complexity estimations are sufficiently accurate. Intuitively, a multilevel event-driven approach should be more appealing to general-purpose computer design which uses both data flow and control flow mechanisms. Due to the complexity in hierarchical control, many research and development efforts are still needed to make this approach workable and cost effective.

For research-oriented readers, we identify below a number of important issues that demand further efforts towards the development of workable data flow multiprocessor systems.

1. The design of a machine instruction set and high-level data flow languages
2. The design of the packet communication networks for resource arbitration and for token distribution
3. The development of data flow processing elements and the structured memory
4. The development of activity control mechanisms and data flow operating system functions

5. Overhead estimations of data flow computers, including development overhead, execution overhead, and application overhead
6. Performance analysis of data flow machines for irregular parallelism with intermix of scalar and vector computations
7. Relative performance of asynchronous data flow machines as compared with synchronous SIMD, MIMD, or pipeline computers
8. The development of program debugging tools and tuning mechanisms

Data flow offers a viable approach to improve today's computer performance. The development of data flow computers is in its infancy stage. With the push of VLSI computing structures, we can anticipate an important role of data flow mechanisms and their variations in future computers.

10.3 VLSI COMPUTING STRUCTURES

Highly parallel computing structures promise to be a major application area for the million-transistor chips that will be possible in just a few years. Such computing systems have structural properties that are suitable for VLSI implementation. Almost by definition, parallel structures imply a basic computational element repeated perhaps hundreds or thousands of times. This architectural style immediately reduces the design problem by similar orders of magnitude. In this section, we examine some VLSI computing structures that have been suggested by computer researchers. We begin with a characterization of the systolic architecture. Then we describe methodologies for mapping parallel algorithms into processor arrays. Finally, we present the reconfigurable processor arrays for designing algorithmically specialized machines. Only globally structured, cellular array structures are presented below. Modularly structured VLSI computing structures will be presented in Section 10.4. Described below are key attributes of VLSI computing structures.

Simplicity and regularity Cost effectiveness has always been a major concern in designing special-purpose VLSI systems; their cost must be low enough to justify their limited applicability. Special-purpose design costs can be reduced by the use of appropriate architectures. If a structure can truly be decomposed into a few types of building blocks which are used repetitively with simple interfaces, great savings can be achieved. This is especially true for VLSI designs where a single chip comprises hundreds of thousands of identical components. To cope with that complexity, simple and regular designs are essential. VLSI systems based on simple, regular layout are likely to be modular and adjustable to various performance levels.

Concurrency and communication Since the technological trend clearly indicates a diminishing growth rate for component speed, any major improvement in computation speed must come from the concurrent use of many processing elements.

The degree of concurrency in a VLSI computing structure is largely determined by the underlying algorithm. Massive parallelism can be achieved if the algorithm is designed to introduce high degrees of pipelining and multiprocessing. When a large number of processing elements work simultaneously, coordination and communication become significant—especially with VLSI technology where routing costs dominate the power, time, and area required to implement a computation. The issue here is to design algorithms that support high degrees of concurrency, and in the meantime to employ only simple, regular communication and control to allow efficient implementation. The locality of interprocessor communications is a desired feature to have in any processor arrays.

Computation intensive VLSI processing structures are suitable for implementing *compute-bound* algorithms rather than *I/O-bound* computations. In a compute-bound algorithm, the number of computing operations is larger than the total number of input and output elements. Otherwise, the problem is I/O bound. For example, the matrix-matrix multiplication algorithm represents a compute-bound task, which has $O(n^3)$ multiply-add steps, but only $O(n^2)$ I/O elements. On the other hand, adding two matrices is I/O bound, since there are n^2 adds and $3n^2$ I/O operations for the two input matrices and one output matrix. The I/O-bound problems are not suitable for VLSI because VLSI packaging must be constrained with limited I/O pins. A VLSI device must balance its computation with the I/O bandwidth. Knowing the I/O-imposed performance limit helps prevent overkill in the design of a special-purpose VLSI device.

10.3.1 The Systolic Array Architecture

The choice of an appropriate architecture for any electronic system is very closely related to the implementation technology. This is especially true in VLSI. The constraints of power dissipation, I/O pin count, relatively long communication delays, difficulty in design and layout, etc., all important problems in VLSI, are much less critical in other technologies. As a compensation, however, VLSI offers very fast and inexpensive computational elements with some unique and exciting properties. For example, bidirectional transmission gates (pass transistors) enable a full barrel shifter to be configured in a very compact NMOS array.

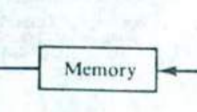
Properly designed parallel structures that need to communicate only with their nearest neighbors will gain the most from very-large-scale integration. Precious time is lost when modules that are far apart must communicate. For example, the delay in crossing a chip on polysilicon, one of the three primary interconnect layers on an NMOS chip, can be 10 to 50 times the delay of an individual gate. The architect must keep this communication bottleneck uppermost in his mind when evaluating possible structures and architectures for implementation in VLSI.

The *systolic* architectural concept was developed by Kung and associates at Carnegie-Mellon University, and many versions of systolic processors are being designed by universities and industrial organizations. This subsection reviews the basic principle of systolic architectures and explains why they should result in cost-effective, high-performance, special-purpose systems for a wide range of potential applications.

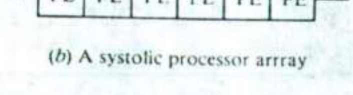
A systolic system consists of a set of interconnected cells, each capable of performing some simple operation. Because simple, regular communication and control structures have substantial advantages over complicated ones in design and implementation, cells in a systolic system are typically interconnected to form a systolic array or a systolic tree. Information in a systolic system flows between cells in a pipelined fashion, and communication with the outside world occurs only at the "boundary" cells. For example, in a systolic array, only those cells on the array boundaries may be I/O ports for the system.

The basic principle of a systolic array is illustrated in Figure 10.25. By replacing a single processing element with an array of PEs, a higher computation throughput can be achieved without increasing memory bandwidth. The function of the memory in the diagram is analogous to that of the heart; it "pulses" data through the array of PEs. The crux of this approach is to ensure that once a data item is brought out from the memory it can be used effectively at each cell it passes. This is possible for a wide class of compute-bound computations where multiple operations are performed on each data item in a repetitive manner.

Suppose each PE in Figure 10.25 operates with a clock period of 100 ns. The conventional memory-processor organization in Figure 10.25*a* has at most a performance of 5 million operations per second. With the same clock rate, the systolic array will result in 30 MOPS performance. This gain in processing speed can also be justified with the fact that the number of pipeline stages has been increased six times in Figure 10.25*b*. Being able to use each input data item a number of times is just one of the many advantages of the systolic approach. Other advantages include modular expansionability, simple and regular data and



(*a*) The conventional processor



(*b*) A systolic processor arrray

Figure 10.25 The concept of systolic processor array. (Courtesy of IEEE *Computer*, Kung, January 1981.)

control flows, use of simple and uniform cells, elimination of global broadcasting, limited fan-in and fast response time.

Basic processing cells used in the construction of systolic arithmetic arrays are the *additive multiply* cells specified in Figure 3.29. This cell has the three inputs a,b,c, and the three outputs $a = a$, $b = b$, and $d = c + a * b$. One can assume six interface registers are attached at the I/O ports of a processing cell. All registers are clocked for synchronous transfer of data among adjacent cells. The additive-multiply operation is needed in performing the inner product of two vectors, matrix-matrix multiplication, matrix inversion, and L-U decomposition of a dense matrix.

Illustrated below is the construction of a systolic array for the multiplication of two banded matrices. An example of band matrix multiplication is shown in Figure 10.26*a*. Matrix **A** has a bandwidth $(3 + 2) - 1 = 4$ and matrix **B** has a bandwidth $(2 + 3) - 1 = 4$ along their principal diagonals. The product matrix $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ then has a bandwidth $(4 + 4) - 1 = 7$ along its principal diagonal. Note that all three matrices have dimension $n \times n$, as shown by the dotted entries. The matrix of bandwidth w may have ω diagonals that are not all zeros. The entries outside the diagonal band are all zeros.

It requires $w_1 \times w_2$ processing cells to form a systolic array for the multiplication of two sparse matrices of bandwidths w_1 and w_2, respectively. The resulting product matrix has a bandwidth of $w_1 + w_2 - 1$. For this example, $w_1 \times w_2 = 4 \times 4 = 16$ multiply cells are needed to construct the systolic array shown in Figure 10.26*b*. It should be noted that the size of the array is determined by the bandwidths w_1 and w_2, independent of the dimension $n \times n$ of the matrices. Data flows in this diamond-shaped systolic array are indicated by the arrows among the processing cells.

The elements of $\mathbf{A} = (a_{ij})$ and $\mathbf{B} = (b_{ij})$ matrices enter the array along the two diagonal data streams. The initial values of $\mathbf{C} = (c_{ij})$ entries are zeros. The outputs at the top of the vertical data stream give the product matrix. Three data streams flow through the array in a pipelined fashion. Let the time delay of each processing cell be one unit time. This systolic array can finish the band matrix multiplication in T time units, where

$$T = 3n + \min(w_1, w_2) \qquad (10.3)$$

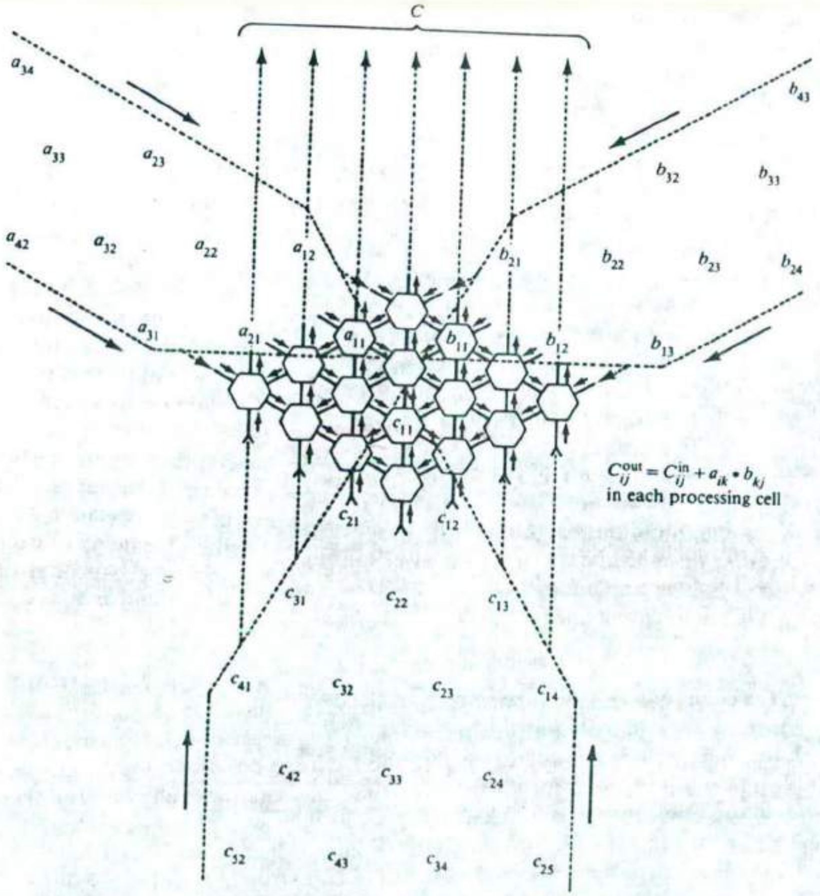
Therefore, the computation time is linearly proportional to the dimension n of the matrix. When the matrix bandwidths increase to $w_1 = w_2 = n$ (for dense matrices **A** and **B**), the time becomes $O(4n)$, neglecting the I/O time delays. If one used a single additive-multiply processor to perform the same matrix multiplication, $O(n^3)$ computation time would be needed. The systolic multiplier thus has a speed gain of $O(n^2)$. For large n, this improvement in speed is rather impressive.

VLSI systolic arrays can assume many different structures for different compute-bound algorithms. Figure 10.27 shows various systolic array configurations and their potential usage in performing those computations is listed in Table 10.1. These computations form the basis of signal and image processing, matrix arithmetic, combinatorial, database algorithms. Due to their simplicity and

$$\begin{bmatrix} a_{11} & a_{12} & & & 0 \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & & \ddots & \\ 0 & & & & \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} & & 0 \\ b_{21} & b_{22} & b_{23} & b_{24} & \\ & b_{32} & b_{33} & b_{34} & b_{35} \\ & & b_{43} & \ddots & \\ 0 & & & & \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & 0 \\ c_{21} & c_{22} & c_{23} & c_{24} & \\ c_{31} & c_{32} & c_{33} & c_{34} & \\ c_{41} & c_{42} & & \ddots & \\ 0 & & & & \end{bmatrix}$$

$A \qquad\qquad B \qquad\qquad C$

(*a*) Band matrix multiplication



(*b*) The systolic array

Figure 10.26 A systolic array for band matrix multiplication. (Courtesy of ***Proc. of the Symposium on Sparse Matrix Computing and Their Applications***, Kung and Leiserson, November 1978.)

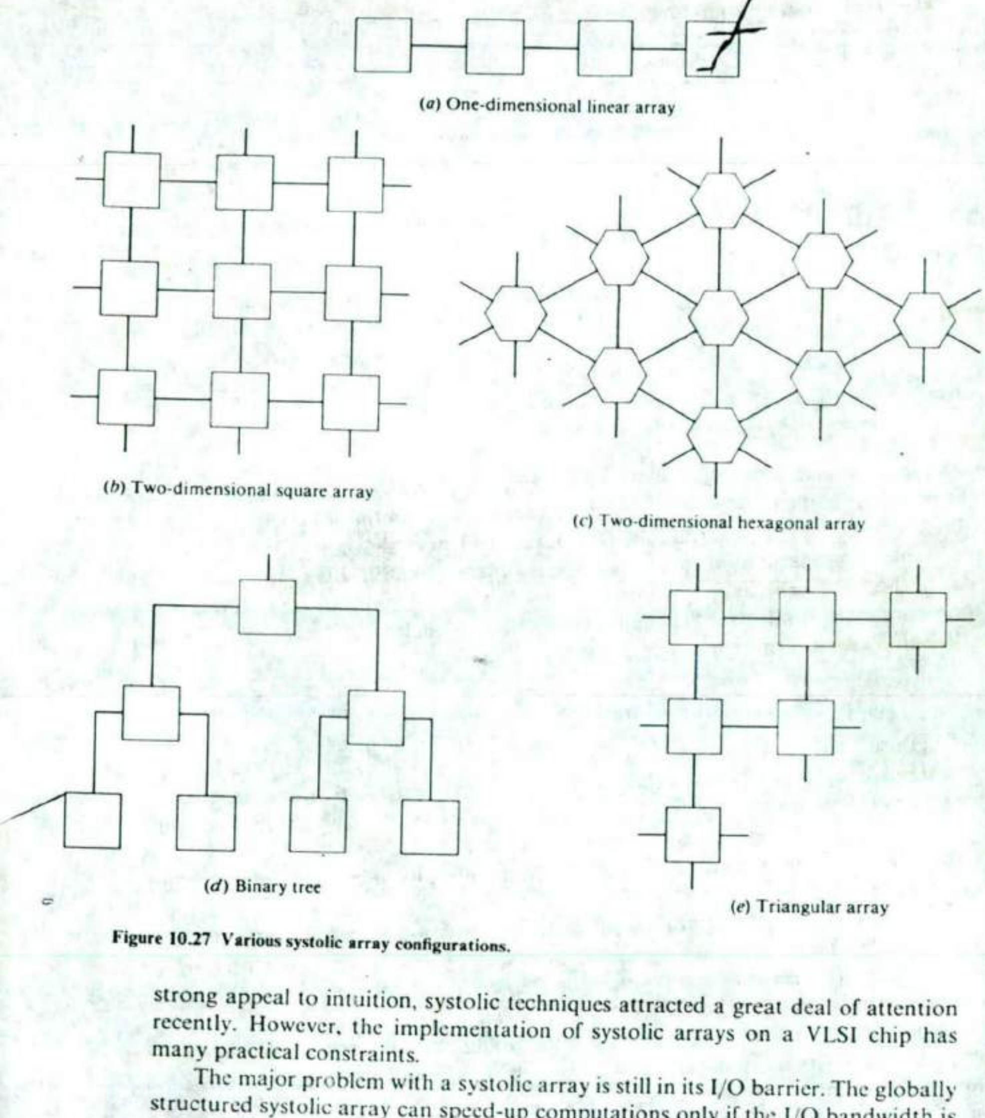

(a) One-dimensional linear array
(b) Two-dimensional square array
(c) Two-dimensional hexagonal array
(d) Binary tree
(e) Triangular array

Figure 10.27 **Various systolic array configurations.**

strong appeal to intuition, systolic techniques attracted a great deal of attention recently. However, the implementation of systolic arrays on a VLSI chip has many practical constraints.

The major problem with a systolic array is still in its I/O barrier. The globally structured systolic array can speed-up computations only if the I/O bandwidth is high. With current IC packaging technology, only a small number of I/O pins can be used for a VLSI chip. For example, a systolic array of n^2 step processors can perform the L-U decomposition in $4n$ time units. However, such a systolic array in a single chip may require $4n \times w$ I/O terminals, where w is the word length.

Table 10.1 Computation functions and desired VLSI structures

Processor array structure	Computation functions
1-D linear arrays	Fir-filter, convolution, discrete Fourier transform (DFT), solution of triangular linear systems, carry pipelining, cartesian product, odd-even transportation sort, real-time priority queue, pipeline arithmetic units
2-D square arrays	Dynamic programming for optimal parenthesization, graph algorithms involving adjacency matrices.
2-D hexagonal arrays	Matrix arithmetic (matrix multiplication, L-U decomposition by Gaussian elimination without pivoting, QR-factorization), transitive closure, pattern match, DFT, relational database operations.
Trees	Searching algorithms (queries on nearest neighbor, rank, etc., systolic search tree), parallel function evaluation, recurrence evaluation.
Triangular arrays	Inversion of triangular matrix, formal language recognition.

For large n (say $n \geq 1000$) with typical operand width $w = 32$ bits, it is rather impractical to fabricate an $n \times n$ systolic array on a monolithic chip with over $4n \times w = 12{,}000$ I/O terminals. Of course, I/O port sharing and time-division multiplexing can be used to alleviate the problem. But still, I/O is the bottleneck. Until the I/O problem can be satisfactorily solved, the systolic arrays can be only constructed in small sizes. The modular VLSI approach to be described in Section 10.4 offers an alternative to overcome this difficulty.

10.3.2 Mapping Algorithms into VLSI Arrays

Procedures to map cyclic loop algorithms into special-purpose VLSI arrays are described below. The method is based on mathematical transformation of the index sets and the data-dependence vectors associated with a given algorithm. After the algorithmic transformation, one can devise a more efficient array structure that can better exploit parallelism and pipelining by removing unnecessary data dependencies.

The exploitation of parallelism is often necessary because computational problems are larger than a single VLSI device can process at a time. If a parallel algorithm is structured as a network of smaller computational modules, then these modules can be assigned to different VLSI devices. The communications between these modules and their operation control dictates the structure of the VLSI system and its performances. In Figure 10.28, a simplistic organization of a computer system is shown consisting of several VLSI devices shared by two processors through a resource arbitration network.

The I/O bottleneck problem in a VLSI system presents a serious restriction imposed on the algorithm design. The challenge is to design parallel algorithms which can be partitioned such that the amount of communication between modules is as small as possible. Moreover, data entering the VLSI device should be utilized exhaustively before passing again through the I/O ports. A global model of the VLSI processor array can be formally described by a 3-tuple (G,F,T); where G is

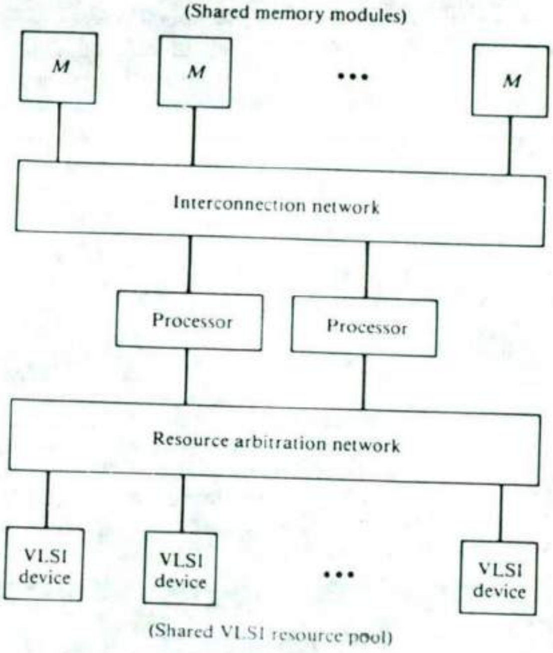


Figure 10.28 A dual-processor system with shared memories and shared VLSI resource pool.

the network geometry, F is the cell function, and T is the network timing. These features are described below separately.

The *network geometry* G refers to the geometrical layout of the network. The position of each processing cell in the plane is described by its Cartesian coordinates. Then, the interconnection between cells can easily be described by the position of the terminal cells. These interconnections support the flow of data through the network; a link can be dedicated only to one data stream of variables or it can be used for the transport of several data streams at different time instances. A simple and regular geometry is desired to uphold local communications.

The *functions* F associated to each processing cell represent the totality of arithmetic and logic expressions that a cell is capable of performing. We assume that each cell consists of a small number of registers, an ALU, and control logic. Several different types of processing cells may coexist in the same network; however, one design goal should be the reduction of the number of cell types used.

The *network timing* T specifies for each cell the time when the processing of functions F occurs and when the data communications take place. A correct timing assures that the right data reaches the right place at the right time. The speed of the data streams through the network is given by the ratio between the distance of the communication link over the communication time. Networks with constant data speeds are preferable because they require a simpler control logic.

The basic structural features of an algorithm are dictated by the data and control dependencies. These dependencies refer to precedence relations of computations which need to be satisfied in order to compute correctly. The absence of dependencies indicates the possibility of simultaneous computations. These dependencies can be studied at several distinct levels: blocks of computations level, statement (or expression) level, variable level, and even bit level. Since we concentrate on algorithms for VLSI systolic arrays, we will focus only on data dependencies at the variable level.

Consider a Fortran loop structure of the form:

```
DO  10    I^1 = l^1, u^1
DO  10    I^2 = l^2, u^2
 :
DO  10    I^n = l^n, u^n
  S_1(I)
  S_2(I)                                    (10.4)
   :
  S_N(I)
10 CONTINUE
```

where l^j and u^j are integer-valued linear expressions involving $I^1, \ldots, I^{j-1}$ and $\mathbf{I} = (I^1, I^2, \ldots, I^n)$ is an index vector. $S_1, S_2, \ldots, S_N$ are assignment statements of the form $x = \mathrm{E}$ where x is a variable and E is an expression of some input variables.

The index set of the loop in Eq. 10.4 is defined by:

$$S^n(\mathbf{I}) = \{(I^1, \ldots, I^n): l^1 \leq I^1 \leq u^1, \ldots, l^n \leq I^n \leq u^n\} \tag{10.5}$$

Consider two statements $S(\mathbf{I}_1)$ and $S(\mathbf{I}_2)$ which perform the functions $f(\mathbf{I}_1)$ and $g(\mathbf{I}_2)$, respectively. Let $V_1(f(\mathbf{I}_1))$ and $V_2(g(\mathbf{I}_2))$ be the output variables of the two statements respectively.

Variable $V_2(g(\mathbf{I}_2))$ is said to be *dependent* on variable $V_1(f(\mathbf{I}_1))$ and denote $V_1(f(\mathbf{I}_1)) \rightarrow V_2(g(\mathbf{I}_2))$, if (i) $\mathbf{I}_1 > \mathbf{I}_2$ (less than in the lexicographical sense); (ii) $f(\mathbf{I}_1) = g(\mathbf{I}_2)$; and (iii) $V_1(f(\mathbf{I}_1))$ is an input variable in statement $S(\mathbf{I}_2)$. The difference of their index vectors $\mathbf{d} = \mathbf{I}_1 - \mathbf{I}_2$ is called the *data-dependence vector*. In general, an algorithm is characterized by a number of data-dependence vectors, which are functions of elements of the index set defined in Eq. 10.5. There is a large class of algorithms which have fixed or constant data dependence vectors.

The transformation of the index sets described above is the key towards an efficient mapping of the algorithm into special-purpose VLSI arrays. The following procedure is suggested to map loop algorithms into VLSI computing structures.

Mapping procedure

1. Pipeline all variables in the algorithm.
2. Find the set of data-dependence vectors.
3. Identify a valid transformation for the data-dependence vectors and the index set.
4. Map the algorithm into hardware structure.
5. Prove correctness and analyze performance.

We consider an example algorithm to illustrate the above procedure: the L-U decomposition of a matrix $\mathbf{A}$ into lower- and upper-triangular matrices by

Gaussian elimination without pivoting. It is shown that better interconnection architectures can be formally derived by using appropriate algorithm transformations.

Example 10.3 The L-U decomposition algorithm is expressed by the following program:

```
for k ← 0 until n−1 do
  begin
    $u_{kk} \leftarrow 1/a_{kk}$
    for j ← k+1 until n−1 do
      $u_{kj} \leftarrow a_{kj}$
    for i ← k+1 until n−1 do
      $l_{ik} \leftarrow a_{ik} u_{kk}$                               (10.6)
    for i ← k+1 until n−1 do
      for j   k+1 until n−1 do
        $a_{ij} \leftarrow a_{ij} + l_{ik} u_{kj}$
  end.
```

This program can be rewritten into the following equivalent form in which all the variables have been pipelined and all the data broadcasts have been eliminated:

```
    for k ← 0 until n−1 do
      begin
1:      i ← k;
        j ← k;
        $u^i_{kj} \leftarrow 1/a^k_{ij}$
        for j ← k+1 until n−1 do
2:      begin
          i ← k;
          $u^i_{kj} \leftarrow a^k_{ij}$
        end
      for i ← k+1 until n−1 do
3:      begin
          j ← k;
          $u^i_{kj} \leftarrow u^{i-1}_{kj}$;                        (10.7)
          $l^i_{ik} \leftarrow a^k_{ij} \cdot u^i_{kj}$
        end
      for i ← k+1 until n−1 do
        for j ← k+1 until n−1 do
4:        begin
            $l^i_{ik} \leftarrow l^{i-1}_{ik}$;
            $u^i_{kj} \leftarrow u^{i-1}_{kj}$;
            $a^k_{ij} \leftarrow a^{k-1}_{ij} - l^{i-1}_{ik} u^{i-1}_{kj}$
          end
    end
```

The data dependencies for this three-loop algorithm have the nice property that

$$\begin{aligned} \mathbf{d}_1 &= (1, 0, 0)^T \\ \mathbf{d}_2 &= (0, 1, 0)^T \\ \mathbf{d}_3 &= (0, 0, 1)^T \end{aligned} \tag{10.8}$$

We write the above in matrix form $\mathbf{D} = [\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3] = \mathbf{I}$. There are several other algorithms which lead to these simple data dependencies, and they were among the first to be considered for the VLSI implementation.

The next step is to identify a linear transformation **T** to modify the data dependencies to be $\mathbf{T} \cdot \mathbf{D} = \Delta$, where $\Delta = [\delta_1, \delta_2, \delta_3]$ represents the modified data dependencies in the new index space, which is selected a priori. This transformation **T** must offer the maximum concurrency by minimizing data dependencies and **T** is a bijection. A large number of choices exist, each leading to a different array geometry. We choose the following one:

$$\mathbf{T} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{such that} \quad \begin{bmatrix} \hat{k} \\ \hat{i} \\ \hat{j} \end{bmatrix} = \mathbf{T} \cdot \begin{bmatrix} k \\ i \\ j \end{bmatrix} \tag{10.9}$$

The original indices k, i, j are being transformed by **T** into $\hat{k}$, $\hat{i}$, $\hat{j}$. The organization of the VLSI array for $n = 5$ generated by this T transformation is shown in Figure 10.29.

In this architecture, variables a_{ij}^k do not travel in space, but are updated in time. Variables l_{ij}^k move along the direction $\hat{j}$ (east with a speed of one grid per

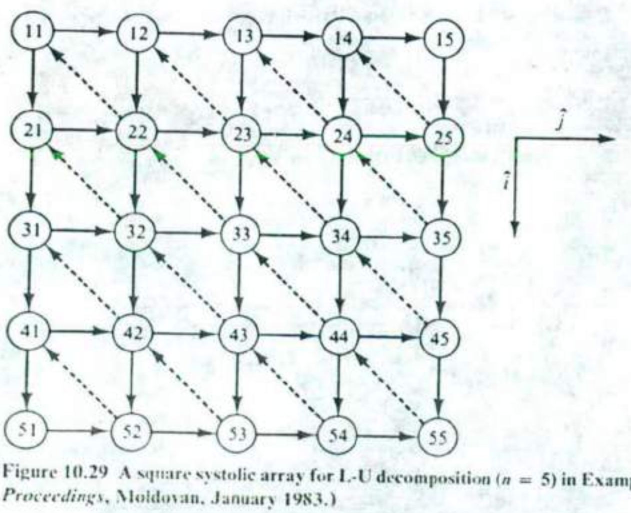


Figure 10.29 A square systolic array for L-U decomposition ($n = 5$) in Example 10.3. (Courtesy of *IEEE Proceedings*, Moldovan, January 1983.)

time unit), and variables u_{ij}^{k} move along the direction $\vec{i}$ (south) with the same speed. The network is loaded initially with the coefficients of **A**, and at the end the cells below the diagonal contain **L** and the cells above the diagonal contain **U**.

The processing time of this square array is $3n - 5$. All the cells have the same architecture. However, their functions at one given moment may differ. It can be seen from the program in statement (10.7) that some cells may execute loop four, while others execute loops two or three. If we wish to assign the same loops only to specific cells, then the mapping must be changed accordingly.

For example, the following transformation:

$$\mathbf{T}' = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

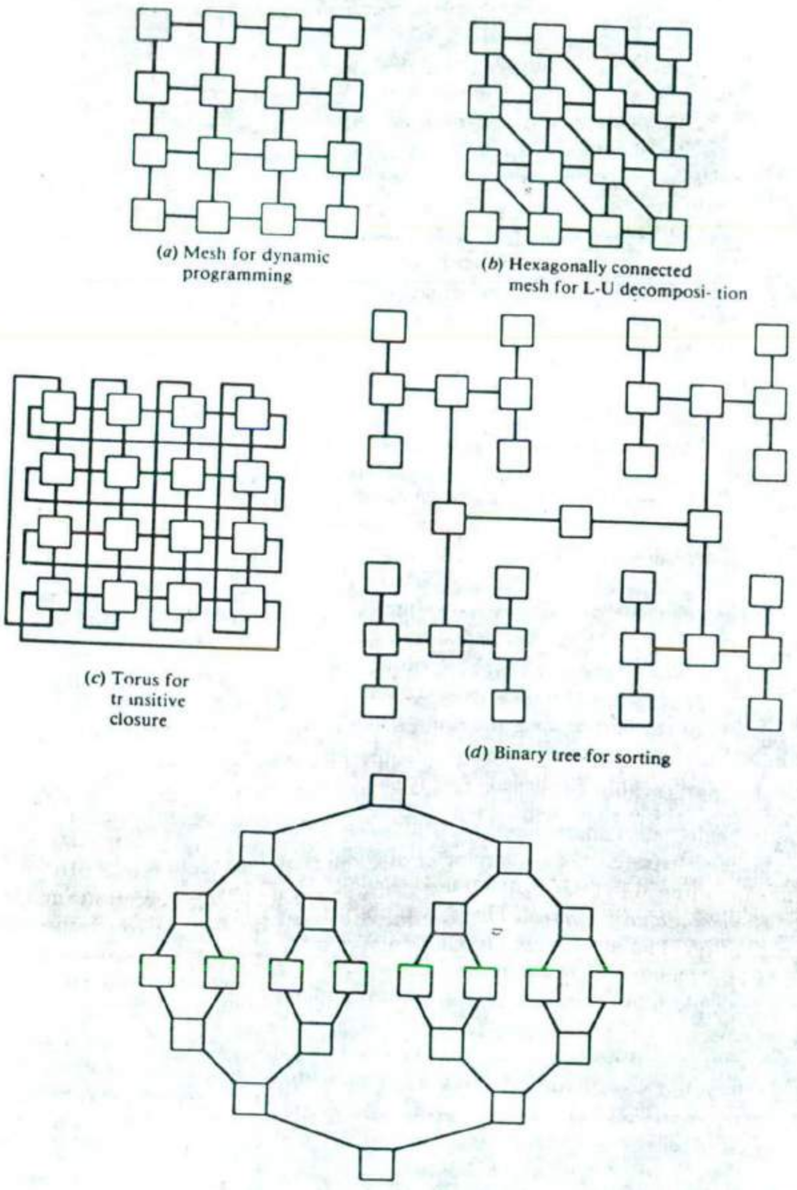
introduces a new data communication link between cells toward north-west. These new links will support the movement of variables a_{ij}^{k}. According to this new transformation, the cells of the first row always compute loop two, the cells of the first column compute loop three, and the rest compute loop four. The reader can now easily identify some other valid transformations which will lead to different array organizations.

The design of algorithmically specialized VLSI devices is at its beginning. The development of specialized devices to replace mathematical software is feasible but still very costly. Several important technical issues remain unresolved and deserve further investigation. Some of these are: I/O communication in VLSI technology, partitioning of algorithms to maintain their numerical stability, and minimization of the communication among computational blocks.

10.3.3 Reconfigurable Processor Array

Algorithmically specialized processors often use different interconnection structures. As demonstrated in Figure 10.30, five array structures have been suggested for implementing different algorithms. The *mesh* is used for dynamic programming. The *hexagonally connected mesh* was shown in the previous section for L-U decomposition. The *torus* is used for transitive closure. The *binary tree* is used for sorting. The *double-rooted tree* is used for searching. The matching of the structure to the right algorithm has a fundamental influence on performance and cost effectiveness.

For example, if we have an $n \times n$ mesh-connected microprocessor structure and want to find the maximum of n^2 elements stored one per processor, $2n - 1$ steps are necessary and sufficient to solve the problem. But a faster algorithmically specialized processor for this problem uses a tree machine to find the solution in $2 \log n$ steps. For large n, this is a benefit worth pursuing. Again, a bus can be introduced to link several differently structured multiprocessors, including mesh- and tree-connected multiprocessors. But the bus bottleneck is quite serious. What we need is a more polymorphic multiprocessor that does not compromise the benefits of VLSI technology.


(*a*) Mesh for dynamic programming

(*b*) Hexagonally connected mesh for L-U decomposi- tion

(*c*) Torus for tr ınsitive closure

(*d*) Binary tree for sorting

(*e*) Doubly rooted tree for searching

Figure 10.30 Algorithmically specialized processor array configurations. (Courtesy of IEEE *Computer*, Snyder, January 1982.)

A family of reconfigurable processor arrays is introduced in this section. This configurable array concept was first proposed in 1982 by Lawrence Snyder at Purdue University. Each configurable VLSI array is constructed with three types of components: a collection of processing elements, a switch lattice, and an array controller. The *switch lattice* is the most important component and the main source of differences among family members. It is a regular structure formed from programmable switches connected by data paths. The PEs are not directly connected to each other, but rather are connected at regular intervals to the switch lattice. Figure 10.31 shows three examples of switch lattices. Generally, the layout will be square, although other geometries are possible. The perimeter switches are connected to external storage devices. With current technology, only a few PEs and switches can be placed on a single chip. As improvements in fabrication technology permit higher device densities, a single chip will be able to hold a larger region of the switch lattice.

Each switch in the lattice contains local memory capable of storing several configuration settings. A configuration setting enables the switch to establish a direct static connection between two or more of its incident data paths. For example, we achieve a mesh interconnection pattern of the PEs for the lattice in Figure 10.31*a* by assigning north-south configuration settings to alternate switches in odd-numbered rows and east-west settings to switches in the odd-numbered columns. Figure 10.32*a* illustrates the configuration; Figure 10.32*b* gives the configuration settings of a binary tree.

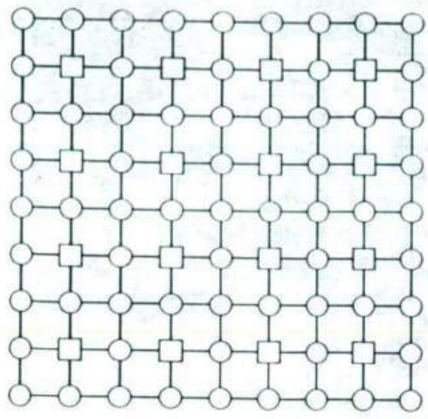
The controller is responsible for loading the switch memory. The switch memory is loaded preparatory to processing and is performed in parallel with the PE program memory loading. Typically, program and switch settings for several phases can be loaded together. The major requirement is that the local configuration settings for each phase's interconnection pattern be assigned to the same memory location in all switches.

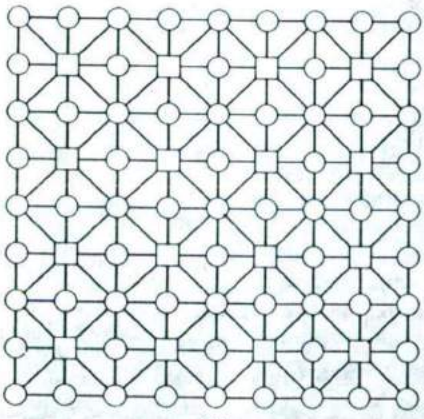
Switch lattices It is convenient to think of the switches as being defined by several characteristic parameters:

- m—the number of wires entering a switch on one data path (path width)
- d—the degree of incident data paths to a switch
- c—the number of configuration settings that can be stored in a switch
- g—the number of distinct data-path groups that a switch can connect simultaneously.

The value of m reflects the balance struck between parallel and serial data transmission. This balance will be influenced by several considerations, one of which is the limited number of pins on the package. Specifically, if a chip hosts a square region of the lattice containing n PEs, then the number of pins required is proportional to $m\sqrt{n}$.

The value of d will usually be four, as in Figure 10.31*a*, or eight, as in Figure 10.31*c*. Figure 10.31*b* shows a mixed strategy that exploits the tendency of switches


(*a*)


(*b*)

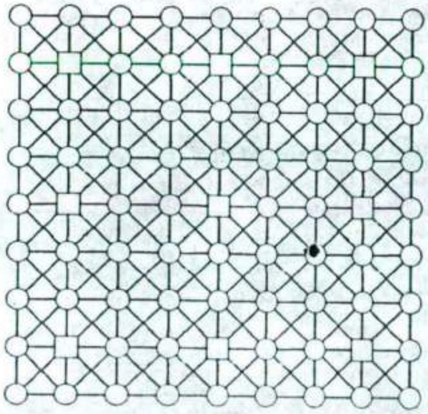
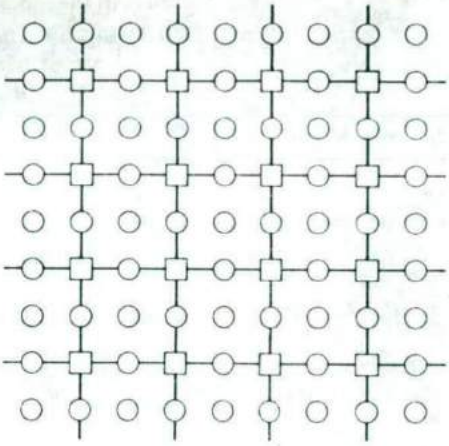
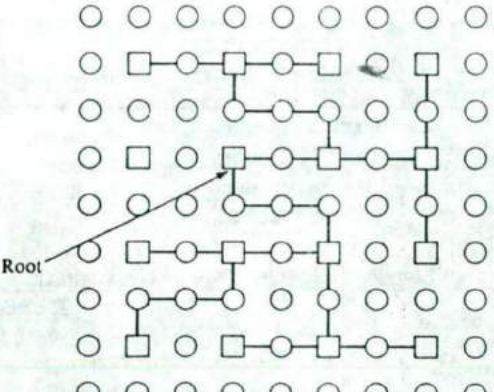

(*c*)

Figure 10.31 Three switch lattice structures (circles represent switches and squares represent PEs). (Courtesy of IEEE *Computer*, Snyder, January 1982.)



(*a*) The switch lattice of Figure 10.31*a* configured into a mesh pattern



(*b*) The switch lattice of Figure 10.31*a* configured into a binary tree

Figure 10.32 Switch lattice configuration settings of the structure in Figure 10.31*a*. (Courtesy of IEEE *Computer*, Snyder, January 1982.)

to be used in two different roles. Switches at the intersection of the vertical and horizontal switch corridors tend to perform most of the routing, while those interposed between two adjacent PEs act more like extended PE ports for selecting data paths from the "corridor buses." The value of *c* is influenced by the number of configurations that may be needed for a multiphase computation and the number of bits required per setting.

The *crossover capability* is a property of switches and refers to the number of distinct data-path groups that a switch can simultaneously connect. Crossover capability is specified by an integer g in the range 1 to $d/2$. Thus, 1 indicates no crossover and $d/2$ is the maximum number of distinct paths intersecting at a degree d switch.

It is clear that lattices can differ in several ways. The PE *degree*, like the switch degree, is the number of incident data paths. Most algorithms of interest use PEs of degree eight or less. Larger degrees are probably not necessary since they can be achieved either by multiplexing data paths or by logically coupling processing elements, e. g., two degree-four PEs could be coupled to form a degree-six PE where one PE serves only as a buffer.

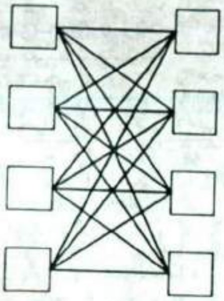
The number of switches that separate two adjacent PEs is called the *corridor width*, w. (See Figure 10.31*c* for a $w = 2$ lattice.) This is perhaps the most significant parameter of a lattice since it influences the efficiency of PE utilization, the convenience of interconnection pattern embeddings, and the overhead required for the polymorphism.

Pattern embedding A given interconnection pattern can be embedded in a programmable switch lattice. We say that the switch lattice "hosts" the given pattern. Figure 10.33 shows the embedding of the complete bipartite graph in the lattice of Figure 10.31*c* where the center column of PEs is unused. Increasing the corridor width improves processor utilization when the complex interconnection patterns must be embedded because it provides more data paths per unit area.

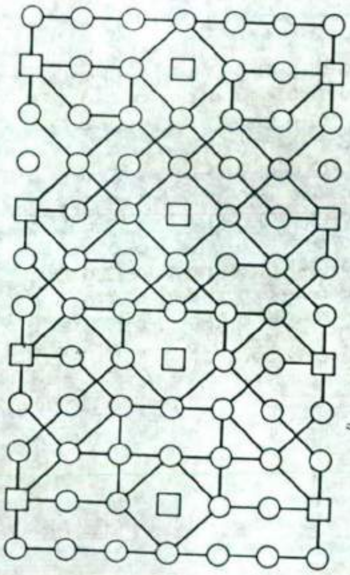
For most of the algorithmically specialized processors arrays, a corridor width of two suffices to achieve optimal or near optimal PE utilization. However, to be sure of hosting all planar interconnection patterns of n nodes with reasonably complete processor utilization, a width proportional to $\log n$ may be necessary. It is possible, using basis elements of 15-node trees embedded in 4×4 square regions of the lattice, to achieve a completely planar embedding of a 255-node complete binary tree (Figure 10.34) into the lattice of Figure 10.31*a*. There is only one unused PE in this planar embedding, as marked by the darkened square at the lower right corner.

By integrating programmable switches with the processing elements, the computer achieves a polymorphism of interconnection structure that also preserves locality. This enables us to compose algorithms that exploit different interconnection patterns. In addition to responding to problems of different sizes and characteristics, the flexibility of integrated switches provides substantial fault tolerance. The above reconfigurable processor array embedded in the switch lattice is a good candidate for *waver-scale integration* (WSI). WSI has been previously attempted by discretionary wiring. Due to the additional masking steps required, this has not proved to be practical. Other researchers are currently investigating laser restructuring and fuse-blowing approaches to implementing WSI.

The concept of WSI implementation of the reconfigurable switch lattice is illustrated in Figure 10.35. A 9×9 grid of building blocks is patterned on a 4-inch wafer. Multiple 2×2 virtual lattices are mapped into a 4×3 building block,



(*a*) Bipartite graph



(*b*) Embedding of the bipartite graph

Figure 10.33 A bipartite graph embedded into the lattice of Figure 10.31*c* using a switch with a crossover value $g = 2$. (Courtesy of IEEE *Computer*, Snyder, January 1982.)

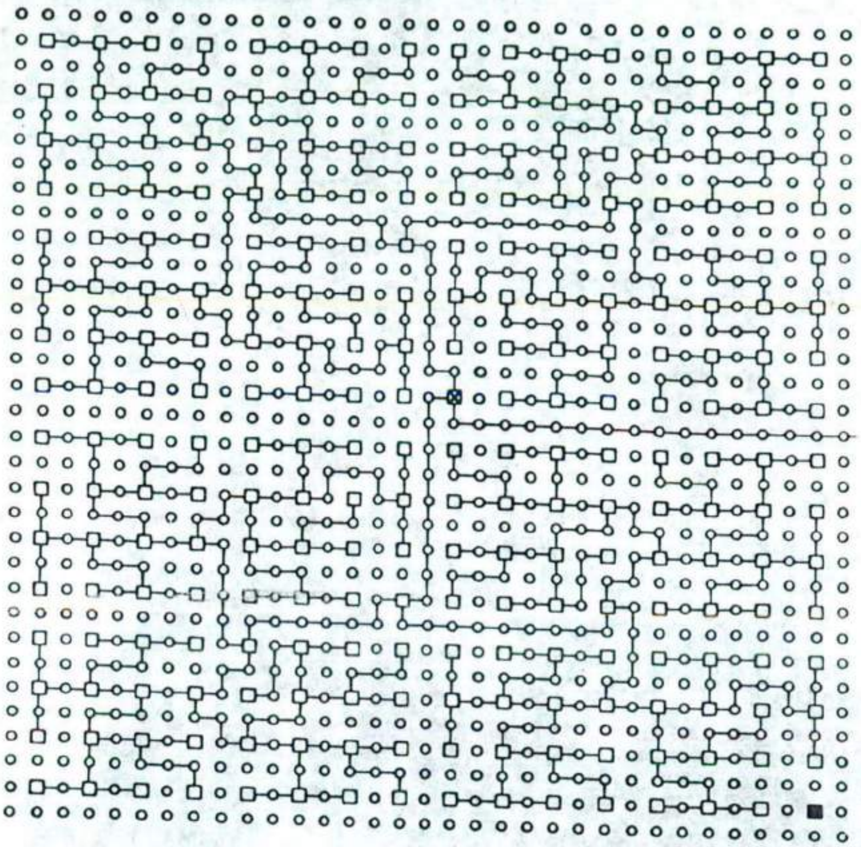


Figure 10.34 Planar embedding of a 255-node binary tree into the switch lattice of Figure 10.31*a* (the root of the tree is at the center of the lattice). (Courtesy of IEEE *Computer*, Snyder, January 1982.)

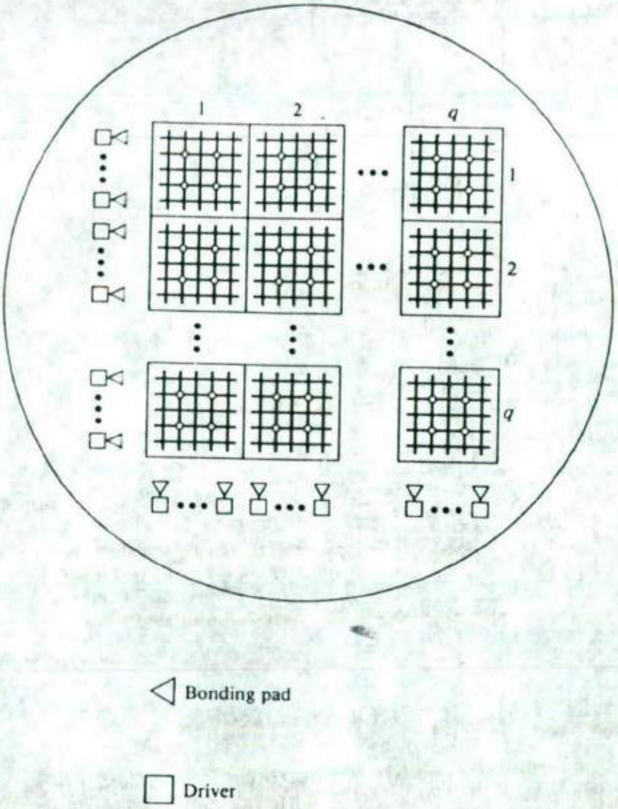


Figure 10.35 Layout of a Wafer Scale Integrated (WSI) processor array. (Courtesy of K. S. Hedlun, Ph.D. Thesis, Purdue University, June 1982).

as illustrated in Figure 10.36. WSI implementation of highly parallel computing structures, either static function or programmable, demands high yield on the wafer. To implement a wafer scale system, all PEs on a wafer are tested, and then the good PEs are connected together. The wafer is structured so that the presence of faulty PEs is masked off and only functional PEs are used.

With current technology, machines with over 300 processors per wafer can be fabricated. These wafer-scale machines will be cheaper, faster, and more reliable than their counterparts implemented with single chip components. However, many practical problems of testing, routing around a faulty PE, power consumption, synchronization, and packaging remain to be solved. Both VLSI and WSI computing structures are being vigorously sought by parallel computing specialists. Three dimensional VLSI computing structures were also recently proposed, in which multi-layer devices are demanded.

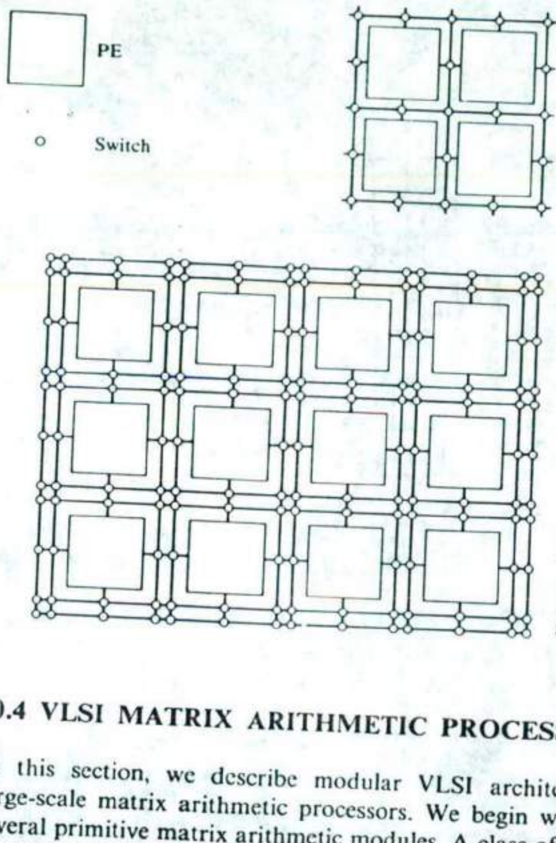


Figure 10.36 A 2 × 2 virtual lattice and a 4 × 3 building block for WSI implementation. (Courtesy of *Proc. of Int'l Conf. on Parallel Processing*, Hedlund and Snyder, August 1982.)

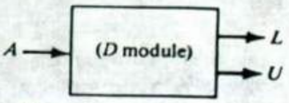
10.4 VLSI MATRIX ARITHMETIC PROCESSORS

In this section, we describe modular VLSI architectures for implementing large-scale matrix arithmetic processors. We begin with the cellular design of several primitive matrix arithmetic modules. A class of partitioned matrix algorithms and their pipelined network implementations are then presented. Performance analysis is given for these VLSI matrix arithmetic solvers. Finally, we show how these matrix solvers can be used for real-time image processing.

10.4.1 VLSI Arithmetic Modules

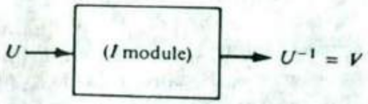
Four primitive VLSI arithmetic modules are functionally introduced in Figure 10.37. These VLSI devices are building blocks for implementing the partitioned matrix algorithms to be studied shortly. These modules are used to perform $m \times m$ submatrix or m-element subvector computations. Each module is constructed with a cellular array of multipliers, dividers, and interface latches for pipelined operations. The schematic logic designs of these primitive VLSI chips are described below.

The D module is for *L-U decomposition* of an intermediate $m \times m$ submatrix $\hat{A}_{rr} = L_{rr} \cdot U_{rr}$ along the principal diagonal of a given matrix A ($\hat{A}_{rr}$ will be defined



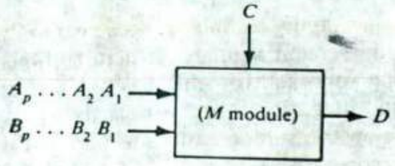
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

(*a*) Submatrix decomposition module



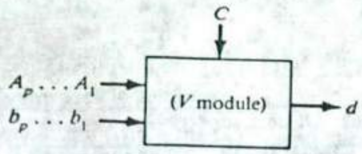
$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}^{-1} = \begin{bmatrix} v_{11} & v_{12} & v_{13} \\ 0 & v_{22} & v_{23} \\ 0 & 0 & v_{33} \end{bmatrix}$$

(*b*) Submatrix inverter



$D = C + \sum_{i=1}^{p} A_i \cdot B_i$ where C, D, and $\{A_i$ and B_i for $i = 1,\ldots,p\}$ are $m \times m$ matrices.

(*c*) Matrix multiplier



$d = c + \sum_{i=1}^{p} A_i \cdot b_i$ where c, d, $\{b_i$ for $i = 1,\ldots,p\}$ are $m \times 1$ column vectors, and $\{A_i$ for $i = 1,\ldots,p\}$ are $m \times m$ matrices.

(*d*) Matrix-vector multiplier

Figure 10.37 VLSI arithmetic modules for matrix computations. (Courtesy of *IEEE Trans. Computers*, Hwang and Cheng, December 1982.)

shortly). The I module is for the *inversion* of a triangular $m \times m$ submatrix. The input-output arithmetic specifications of these VLSI modules are given in the drawing. Both D and I modules have a fixed delay of 2m time units. One time unit equals the time required to perform one multiply-add operation $a \times b + c = d$, or one divide operation $a/b = c$, by one step processor in the cellular arrays shown in Figure 10.38 and Figure 10.39.

The M module is the predominant device to be used in the construction of various matrix solvers. *Accumulative chain matrix multiplications* are performed by an M module as specified in Figure 10.40. The number p of pairs of $m \times m$ matrices to be multiplied and added is determined by the external input sequence. Therefore, the time delay of an M module is equal to $p \cdot m + 1$. The V module is modified from the M module for *accumulative submatrix-vector multiplications*. The delay of a V module is also measured as $p \cdot m + 1$. Because each D, I, or M module contains an array of about m^2 step processors, their interior chip complexity is $O(m^2)$. Each V module contains a pipeline of m step processors and thus has an interior chip complexity of $O(m)$. The time delays of each D or I module has an order $O(m)$ and those for M and V modules are $O(m \cdot p)$, depending on the number of input pairs to be processed.

10.4.2 Partitioned Matrix Algorithms

Based on the state-of-the-art electronic and packaging technologies, we can only expect VLSI arithmetic devices designed to implement regularly structured functions with limited I/O terminals. A modular approach to achieve VLSI matrix arithmetic is amenable from the viewpoints of feasibility and applicability. A matrix partitioning approach is presented below to overcome those technological constraints in constructing large-scale matrix processors. Four partitioned algorithms are described below for modular VLSI implementation:

- L-U decomposition by a variant of Gaussian elimination
- Normal inversion of a nonsingular triangular matrix
- Multiplication of two compatible matrices
- Solving a triangular system of equations by back substitution

L-U decomposition by partitioning A partitioned approach is proposed to circumvent the I/O problem of systolic arrays by using $m \times m$ VLSI modules, where m is smaller than n in at least two orders of magnitude. Of course, I/O port sharing and time-division multiplexing are often used to satisfy the IC packaging constraints, even for small m. The partitioned triangular decomposition is illustrated in Figure 10.41. The given matrix $\mathbf{A} = (a_{ij})$ is partitioned into k^2 submatrices of the order $m \times m$ each. The submatrix computation sequence is identified in steps. This method is equivalent to Crout's method when $m = 1$. However, we consider nontrivial cases where $m = 2$.

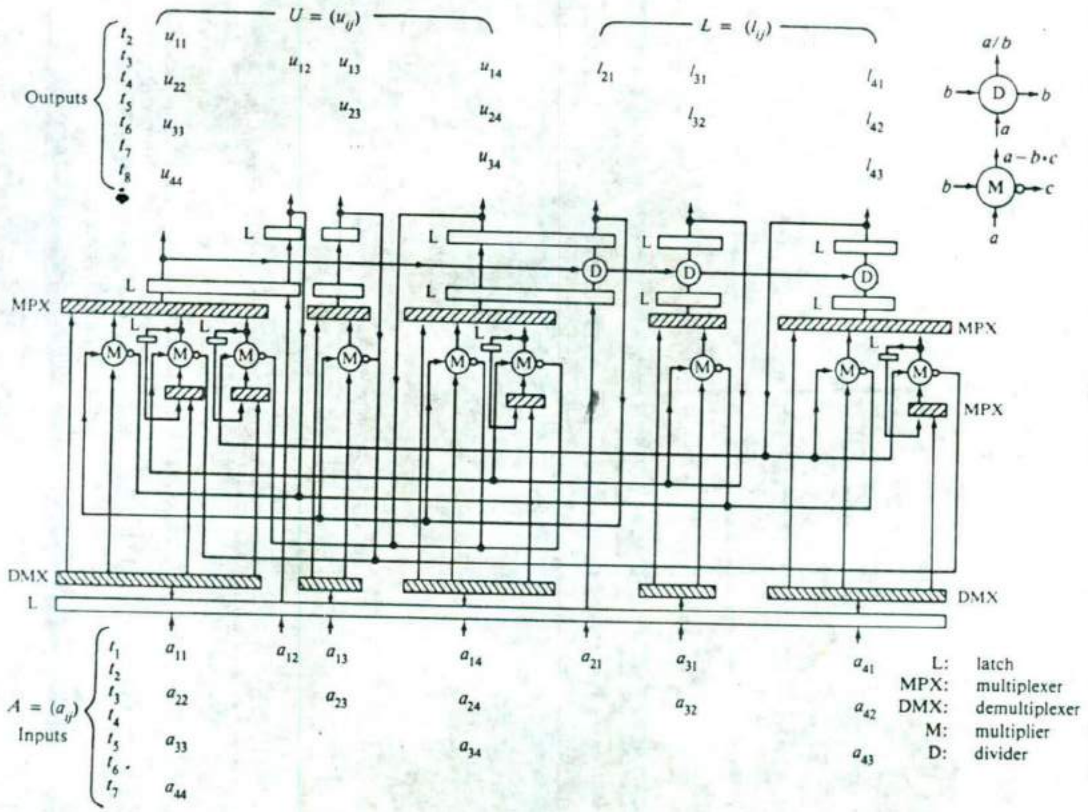
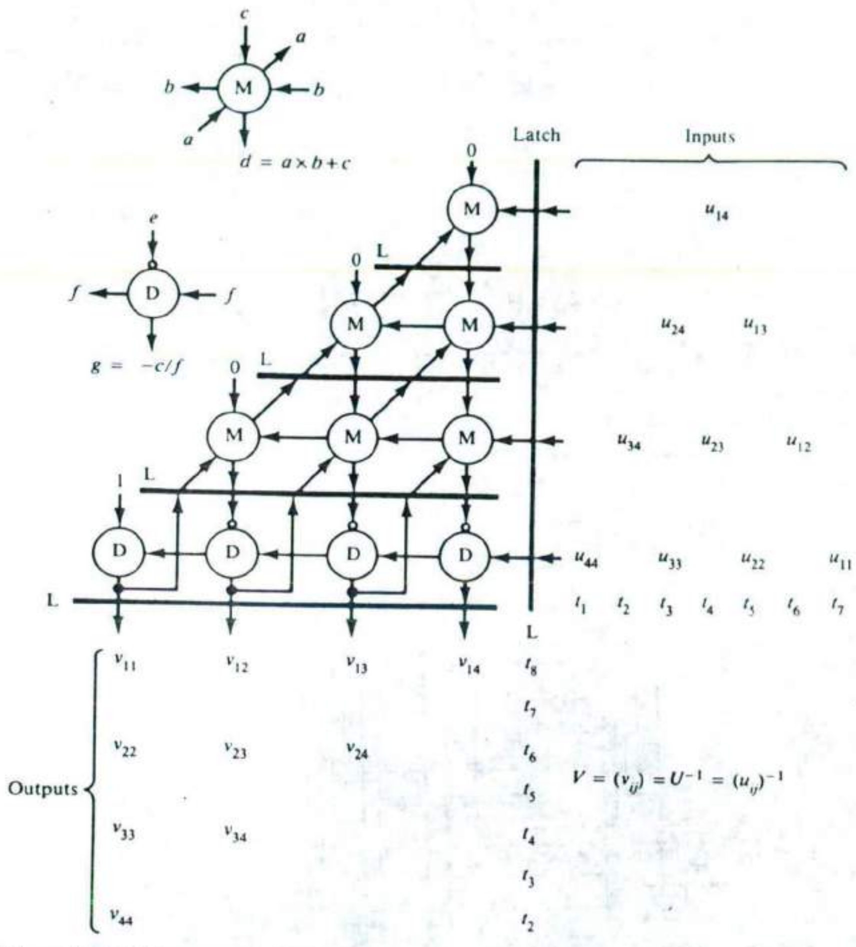


Figure 10.38 VLSI computing module for local L-U decomposition. (Courtesy of IEEE *Proc. of 5th Symp. of Computer Arithmetic*, Hwang and Cheng, May 1981.)

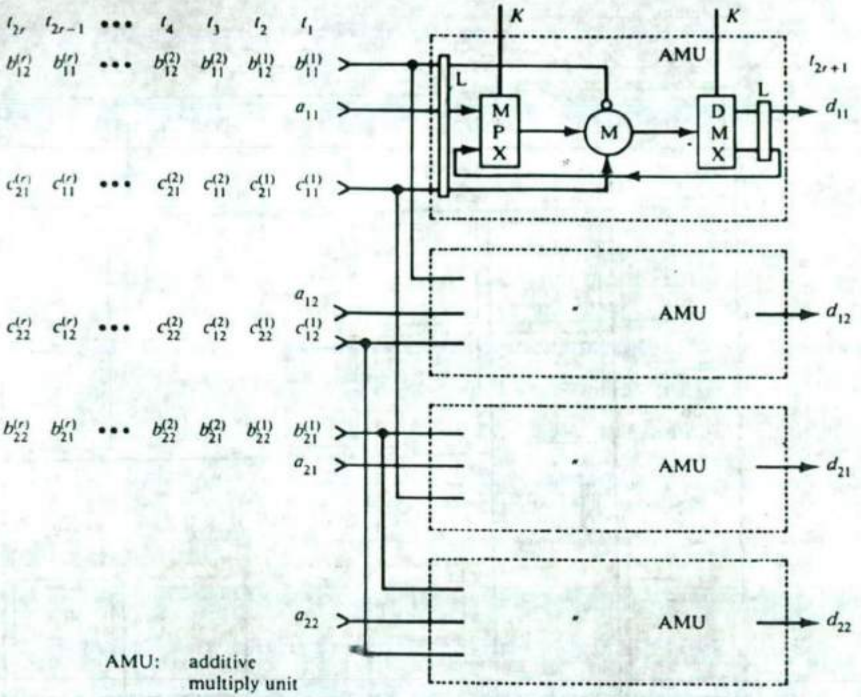


c
a
b
M
b
a
$d = a \times b + c$
e
f
D
f
$g = -c/f$
Latch
Inputs
0
L
I
u_{14}
u_{24}
u_{13}
u_{34}
u_{23}
u_{12}
u_{44}
u_{33}
u_{22}
u_{11}
t_1 t_2 t_3 t_4 t_5 t_6 t_7
Outputs
v_{11} v_{12} v_{13} v_{14}
v_{22} v_{23} v_{24}
v_{33} v_{34}
v_{44}
t_8 t_7 t_6 t_5 t_4 t_3 t_2
$V = (v_{ij}) = U^{-1} = (u_{ij})^{-1}$

Figure 10.39 VLSI computing module for the inversion of a triangular matrix. (Courtesy of *IEEE Proc. of 5th Symp. on Computer Arithmetic*, Hwang and Cheng, May 1981.)

At step one, L-U decomposition of submatrix A_{11} is performed using a D module to generate the two triangular submatrices L_{11} and U_{11} such that $A_{11} = L_{11} \cdot U_{11}$. Two I modules are used to compute the inverse submatrices L_{11}^{-1} and U_{11}^{-1} at step two. The following matrix multiplications are then performed by $2(k - 1)$ M modules in parallel at step 2.

$$L_p = A_{p1} \cdot U_{11}^{-1} \qquad \text{for } p = 2, 3, \ldots, k$$
$$U_{1q} = L_{11}^{-1} \cdot A_{1q} \qquad \text{for } q = 2, 3, \ldots, k \tag{10.11}$$


t_{2r} t_{2r-1} ••• t_4 t_3 t_2 t_1
$b_{12}^{(r)}$ $b_{11}^{(r)}$ ••• $b_{12}^{(2)}$ $b_{11}^{(2)}$ $b_{12}^{(1)}$ $b_{11}^{(1)}$
a_{11}
$c_{21}^{(r)}$ $c_{11}^{(r)}$ ••• $c_{21}^{(2)}$ $c_{11}^{(2)}$ $c_{21}^{(1)}$ $c_{11}^{(1)}$
a_{12}
$c_{22}^{(r)}$ $c_{12}^{(r)}$ ••• $c_{22}^{(2)}$ $c_{12}^{(2)}$ $c_{22}^{(1)}$ $c_{12}^{(1)}$
$b_{22}^{(r)}$ $b_{21}^{(r)}$ ••• $b_{22}^{(2)}$ $b_{21}^{(2)}$ $b_{22}^{(1)}$ $b_{21}^{(1)}$
a_{21}
a_{22}
K
K
L
MPX
M
DMX
L
AMU
t_{2r+1}
d_{11}
AMU
d_{12}
AMU
d_{21}
AMU
d_{22}

AMU: additive multiply unit

$$\begin{bmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} - \sum_{i=1}^{r} \begin{bmatrix} b_{11}^{(i)} & b_{12}^{(i)} \\ b_{21}^{(i)} & b_{22}^{(i)} \end{bmatrix} \cdot \begin{bmatrix} c_{11}^{(i)} & c_{12}^{(i)} \\ c_{21}^{(i)} & c_{22}^{(i)} \end{bmatrix}$$

$$d_{11} = a_{11} - \sum_{i=1}^{r} b_{11}^{(i)} \cdot c_{11}^{(i)} + b_{12}^{(i)} \cdot c_{21}^{(i)}$$
$$d_{12} = a_{12} - \sum_{i=1}^{r} b_{11}^{(i)} \cdot c_{12}^{(i)} + b_{12}^{(i)} \cdot c_{22}^{(i)}$$
$$d_{21} = a_{21} - \sum_{i=1}^{r} b_{21}^{(i)} \cdot c_{11}^{(i)} + b_{22}^{(i)} \cdot c_{21}^{(i)}$$
$$d_{22} = a_{22} - \sum_{i=1}^{r} b_{21}^{(i)} \cdot c_{12}^{(i)} + b_{22}^{(i)} \cdot c_{22}^{(i)}$$

K:	control
L:	latch
MPX:	multiplexer
DMX:	demultiplexer

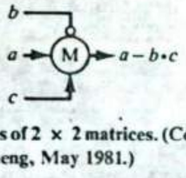
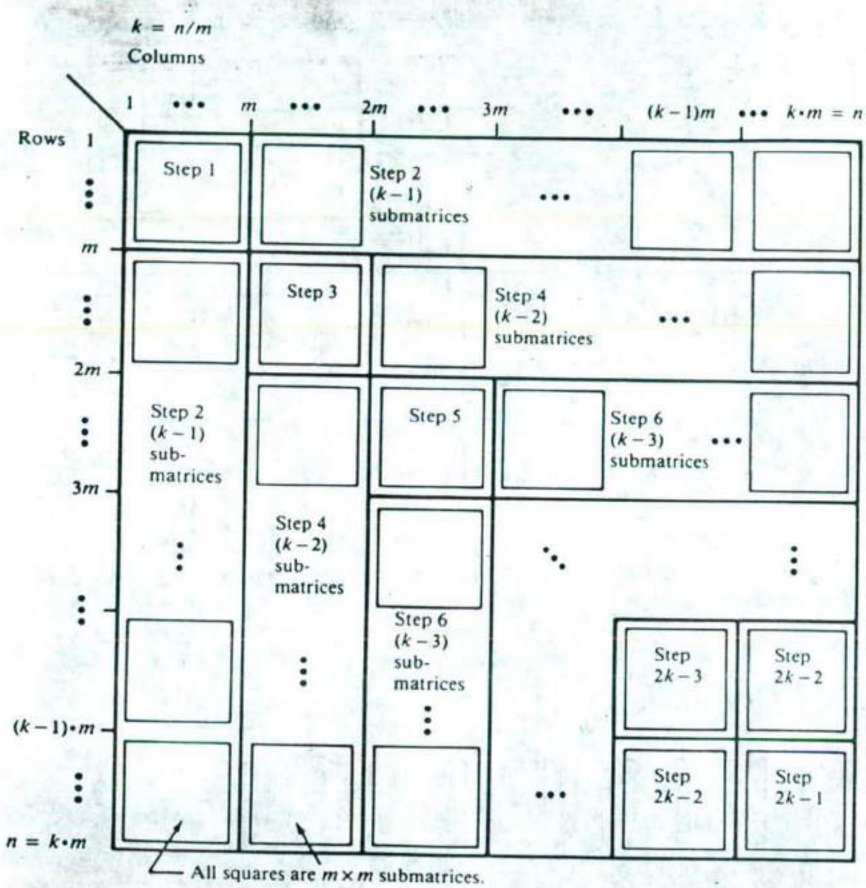


Figure 10.40 VLSI arithmetic module for the multiplication of two sequences of 2 × 2 matrices. (Courtesy of IEEE *Proc. of 5th Symposium of Computer Arithmetic*, Hwang and Cheng, May 1981.)



k = n/m
Columns
1 ••• m ••• 2m ••• 3m ••• (k − 1)m ••• k•m = n
Rows 1
m
2m
3m
(k − 1)•m
n = k•m
Step 1
Step 2 (k − 1) submatrices
Step 3
Step 4 (k − 2) submatrices
Step 5
Step 6 (k − 3) submatrices
Step 2 (k − 1) sub-matrices
Step 4 (k − 2) sub-matrices
Step 6 (k − 3) sub-matrices
Step 2k − 3
Step 2k − 2
Step 2k − 2
Step 2k − 1
All squares are m × m submatrices.

Figure 10.41 Computation sequence of the partitioned L-U decomposition algorithm by Hwang and Cheng (1982).

In subsequent steps, we generate the following intermediate submatrices using the M modules:

$$\hat{A}_{pq} = A_{pq} - \sum_{s=1}^{r-1} L_{ps} \cdot U_{sq} \qquad \text{for } p, q = 2, 3, \ldots, k \tag{10.12}$$

Local L-U decompositions are then performed on $\hat{A}_{rr}$ at successively odd-numbered steps along the principal diagonal as shown in Figure 10.41:

$$L_{rr} \cdot U_{rr} = \hat{A}_{rr} \qquad \text{for } r = 2, 3, \ldots, k \tag{10.13}$$

The remaining off-diagonal submatrices L_{pr} and U_{rq} are computed by inverting the diagonal submatrices U_{rr} and L_{rr} and then multiplying them by intermediate submatrices $\hat{A}_{pr}$ and $\hat{A}_{rq}$ at successively even-numbered steps. For $r = 2, 3, \ldots, k$, we compute

$$\begin{cases} L_{pr} = \hat{A}_{pr} \cdot U_{rr}^{-1} & \text{for } p = r + 1, \ldots, k \\ U_{rq} = L_{rr}^{-1} \cdot \hat{A}_{rq} & \text{for } q = r + 1, \ldots, k \end{cases} \tag{10.14}$$

The partitioned L-U decomposition is exemplified below. Each capital entry in the matrix **A** represents an $m \times m$ submatrix. The given matrix has an order n. We assume $n = m \cdot k$ for some integer k. The given matrix **A** is thus partitioned into $k^2 = n^2/m^2$ submatrices. The L-U decomposition of matrix **A** can be done in $2k - 1$ submatrix steps. All the diagonal elements of **L** are equal to 1. All diagonal submatrices of **U** are upper triangular matrices. The following example corresponds to the case of $k = n/m = 3$:

Example 10.4

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

$$= \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \cdot \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix} = \mathbf{L} \cdot \mathbf{U}$$

1. $A_{11} = L_{11} \cdot U_{11}$ (D module)
2. $L_{21} = A_{21} \cdot U_{11}^{-1}; L_{31} = A_{31} \cdot U_{11}^{-1}$

   $U_{12} = L_{11}^{-1} \cdot A_{12}; U_{13} = L_{11}^{-1} \cdot A_{13}$ (I modules and M modules)
3. $\hat{A}_{22} = A_{22} - L_{21} \cdot U_{12} = L_{22} \cdot U_{22}$

   $\hat{A}_{23} = A_{23} - L_{21} \cdot U_{13}; \hat{A}_{32} = A_{32} - L_{31} \cdot U_{12}$ (M modules and D modules)
4. $U_{23} = L_{22}^{-1} \cdot \hat{A}_{23}; L_{32} = \hat{A}_{32} \cdot U_{22}^{-1}$ (I modules and M modules)
5. $\hat{A}_{33} = A_{33} - (L_{31} \cdot U_{13} + L_{32} \cdot U_{23}) = L_{33} \cdot U_{33}$ (M modules and D modules)

The above interactive procedures are summarized in Figure 10.42 for partitioned L-U decomposition of any groups of any nonsingular dense matrix **A** of

```
ALGORITHM (Partitioned L-U Decomposition)
Inputs:
    An $n \times n$ dense matrix $\mathbf{A} = (a_{ij})$ partitioned into $k^2 m \times m$ submatrices $A_{ij}$ for $i, j = 1, 2, \ldots, k$,
    where $n \equiv k \cdot m$.
Outputs:
    $k \cdot (k + 1)$ submatrices $L_{pq}$ for $q \leq p \equiv 1, 2, \ldots, k$ and $U_{rs}$ for $s \geq r \equiv 1, 2, \ldots, k$, each of order
    $m \times m$.
Procedures:
    (1) Decompose $A_{11}$ into $L_{11}$ and $U_{11}$ such that $L_{11} \cdot U_{11} = A_{11}$.
    (2) Compute inverse matrices $L_{11}^{-1}$ and $U_{11}^{-1}$;
        Compute $L_{p1} = A_{p1} \cdot U_{11}^{-1}$; $U_{1p} = L_{11}^{-1} \cdot A_{1p}$ for $p = 2, 3, \ldots, k$.
    (3) For $q \leftarrow 2$ to $(k - 1)$ step 1 do
            Compute $\hat{A}_{qq} = A_{qq} - \sum_{s=1}^{q-1} L_{qs} \cdot U_{sq}$;
            Decompose $\hat{A}_{qq} = L_{qq} \cdot U_{qq}$;
            Compute the matrices $L_{qq}^{-1}$ and $U_{qq}^{-1}$.
        For $p \leftarrow (q + 1)$ to $k$ step 1 do
            Compute $\hat{A}_{pq} = A_{pq} - \sum_{s=1}^{r-1} L_{ps} \cdot U_{sq}$;
            and $\hat{A}_{pq} = A_{qp} - \sum_{s=1}^{r-1} L_{qs} \cdot U_{sp}$ for $r = \min(p, q)$;
            Compute $L_{pq} = \hat{A}_{pq} \cdot U_{qq}^{-1}$; $U_{qp} = L_{qq}^{-1} \cdot \hat{A}_{qp}$.
        Repeat
    Repeat
    (4) Compute $\hat{A}_{kk} = A_{kk} - \sum_{s=1}^{k-i} L_{ks} \cdot U_{sk}$
        Decompose $\hat{A}_{kk} = L_{kk} \cdot U_{kk}$
```

Figure 10.42 Partitioned algorithm for L-U decomposition. (Courtesy of *IEEE Trans. Computers*, Hwang and Cheng, 1982.)

order n. Submatrix computations are specified in groups. Each group can be computed in parallel by multiple VLSI modules. Submatrix computations can also be computed in sequential order, if only a limited number of VLSI chips are available.

Matrix inversion and multiplication A partitioned algorithm is described for iterative inversion of an $n \times n$ nonsingular triangular matrix using I and M modules. For clarity, we demonstrate the partitioning method by finding the inverse of an example $4m \times 4m$ upper triangular matrix $U = (u_{ij})$ with $m \times m$ modules. The inverse matrix $V = (v_{ij}) = U^{-1}$ is partitioned into $k^2 = (n/m)^2 = (4m/n)^2 = 4^2 = 16$ submatrices, as exemplified below:

Example 10.5

$$\mathbf{U}^{-1} = \begin{pmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{23} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{pmatrix}^{-1} .$$

$$= \begin{pmatrix} V_{11} & V_{12} & V_{13} & V_{14} \\ 0 & V_{22} & V_{23} & V_{24} \\ 0 & 0 & V_{33} & V_{34} \\ 0 & 0 & 0 & V_{44} \end{pmatrix} = \mathbf{V}$$

1. $V_{11} = U_{11}^{-1}$; $V_{22} = U_{22}^{-1}$; $V_{33} = U_{33}^{-1}$; $V_{44} = U_{44}^{-1}$ (I modules)
2. $V_{12} = -V_{11} \cdot (U_{12} \cdot V_{22})$; $V_{23} = -V_{22} \cdot (U_{23} \cdot V_{33})$;
   $V_{34} = -V_{33} \cdot (U_{34} \cdot V_{44})$ (M modules)
3. $V_{13} = -V_{11} \cdot (U_{12} \cdot V_{23} + U_{13} \cdot V_{33})$
   $V_{24} = -V_{22} \cdot (U_{23} \cdot V_{34} + U_{24} \cdot V_{44})$ (M modules)
4. $V_{14} = -V_{11} \cdot (U_{12} \cdot V_{24} + U_{13} \cdot V_{34} + U_{14} \cdot V_{44})$ (M modules)

Partitioned multiplication of two large $n \times n$ matrices, say $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$, is rather straightforward. We include it here for completeness. Basically, each $m \times m$ submatrix C_{pq} of matrix $\mathbf{C}$ is obtained by performing $C_{pq} = \sum_{r=1}^{k} A_{pr} \cdot B_{rq}$, by one M module for each $p, q = 1, 2, \ldots, k$, where $km = n$ and A_{pr} and B_{rq} are $m \times m$ submatrices of the input matrices $\mathbf{A}$ and $\mathbf{B}$ respectively.

Triangular linear system solver The VLSI solution of a triangular linear system of equations can be done in k back-substitution steps. In the following example, $\mathbf{x}_i$ and $\mathbf{b}_i$ for $i = 1, 2, \ldots, k$ are $m \times 1$ subvectors and U_{ij} are $m \times m$ submatrices:

Example 10.6

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \mathbf{x}_4 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \\ \mathbf{b}_4 \end{bmatrix}$$

1. $\mathbf{x}_4 = U_{44}^{-1} \cdot \mathbf{b}_4$ (I modules and V modules)
2. $\mathbf{x}_3 = U_{33}^{-1} \cdot (\mathbf{b}_3 - U_{34} \cdot \mathbf{x}_4)$ (I modules and V modules)
3. $\mathbf{x}_2 = U_{22}^{-1} [\mathbf{b}_2 - (U_{23} \cdot \mathbf{x}_3 + U_{24} \cdot \mathbf{x}_4)]$ (I modules and V modules)
4. $\mathbf{x}_1 = U_{11}^{-1} \cdot [\mathbf{b}_1 - (U_{12} \cdot \mathbf{x}_2 + U_{13} \cdot \mathbf{x}_3 + U_{14} \cdot \mathbf{x}_4)]$ (I modules and V modules)

In general, $k = n/m$ steps are needed in the back substitution. Matrix **U** is partitioned into $k \times (k + 1)/2$ submatrices of order $m \times m$ each. The solution vector **x** is divided into k subvectors.

10.4.3 Matrix Arithmetic Pipelines

Several matrix arithmetic processors are described below using those VLSI arithmetic modules as building blocks. These matrix networks will be used in Section 10.4.4 for the implementation of a feature extractor and a pattern classifier. The first network is for L-U decomposition of an $n \times n$ matrix **A**. The second one is for the inversion of a triangular matrix of order n. The third one is for partitioned matrix multiplication, and the fourth one is for solving a triangular system of equations. With a pipelined architecture, only linear number $O(n/m)$ of VLSI chips is needed. The projected speedup of $O(n^3)/O(n^2/m)$ or $O(n^2)/O(n)$ is rather impressive for $n \gg m$ in real-time computations.

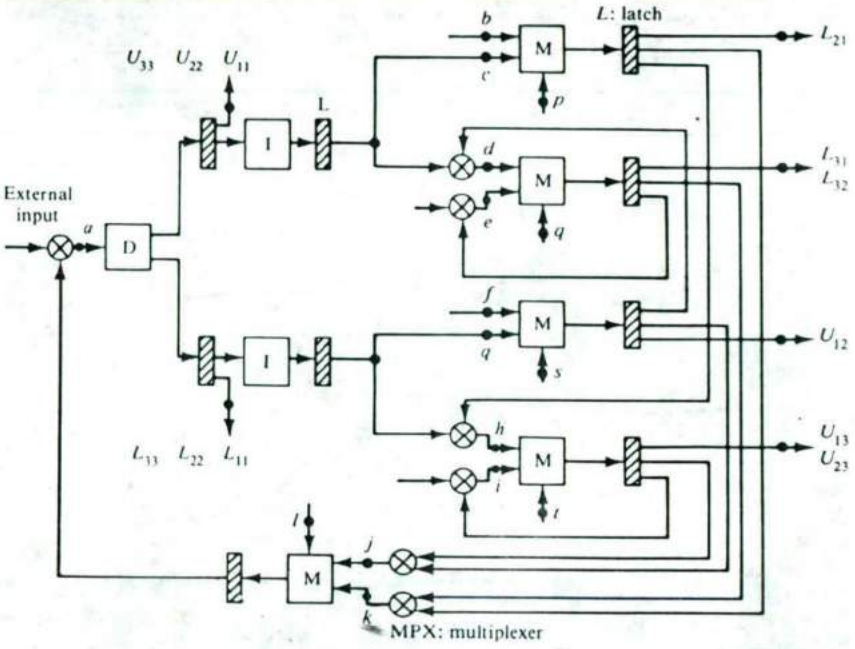
Matrix arithmetic solvers An L-U decomposition network is designed in Figure 10.43 for the computations in Example 10.4. In general, such a VLSI network needs to use one D module, two I modules, and $2k - 1$ M modules. These VLSI arithmetic modules are interfaced with high-speed latch memories to yield pipelining operations with feedback connections. Multiplexers are used to select the appropriate input to the functional modules at different steps. Note that the steps are divided according to submatrix operations. Each step may require a different number of pipeline cycles to complete the operation. To decompose an $n \times n$ matrix **A** into the two triangular matrices **L** and **U** such that $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$, the network requires $O(n^2/m)$ time with a total module count of $O(n/m)$. It requires $O(n^3)$ time delay for implementing the same algorithm on a uniprocessor computer.

Parallel M modules can be used to perform the partitioned matrix multiplications, as demonstrated in Figure 10.44 for the case of $k = n/m = 3$. $O(n^2/m)$ time is required with the use of $O(n/m)$ M modules. Moreover, one can achieve $O(n)$ time at the expense of using $O(n^2/m^2)$ M modules for partitioned matrix multiplication.

The matrix inversion algorithm in Example 10.5 is realized by the pipelined matrix network in Figure 10.45. In general, inverting a triangular matrix of order n requires the use of k I modules and $2(k - 1)$ M modules. Thus, the total module count equals $k + 2(k - 1) = 3k - 2 = O(k) = O(n/m)$ for $n \gg m$. The input assignments and data flows at intermediate and output terminals are also specified in the drawing. The total time delay in using this network to generate the inverse matrix $\mathbf{V} = \mathbf{U}^{-1}$ is equal to $O(n^2/m)$ for $n \gg m$.

Figure 10.46 shows the pipelined network for solving a triangular system of equations, as demonstrated in Example 10.6. In general, the network requires one I module and $[(n + m + 2)/(2m + 2)]$ V modules. The time delay of this network is $O(n)$ with the use of $O(n/m)$ V modules.

Performance analysis VLSI module requirements and the speed complexity of the above matrix algorithms are analyzed below. We consider two architectural



Terminals	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*p*	*q*	*s*	*t*
Step 1	A_{11}															
Step 2		A_{21}	U_{11}^{-1}	U_{11}^{-1}	A_{31}	A_{12}	L_{11}^{-1}	L_{11}^{-1}	A_{13}				0	0	0	0
Step 3	$\overline{A}_{22}$			U_{12}	L_{31}			L_{21}	U_{13}	U_{12}	L_{21}	A_{22}		A_{32}		A_{23}
Step 4				U_{22}^{-1}	$\overline{A}_{32}$			L_{22}^{-1}	$\overline{A}_{23}$					0		0
Step 5	$\overline{A}_{33}$									U_{13} U_{23}	L_{31} L_{32}	A_{33}				

Figure 10.43 Arithmetic pipeline for partitioned L-U decomposition (Example 10.4). (Courtesy of *Computer Vision, Graphics, and Image Processing*, Hwang and Su, 1983a.)

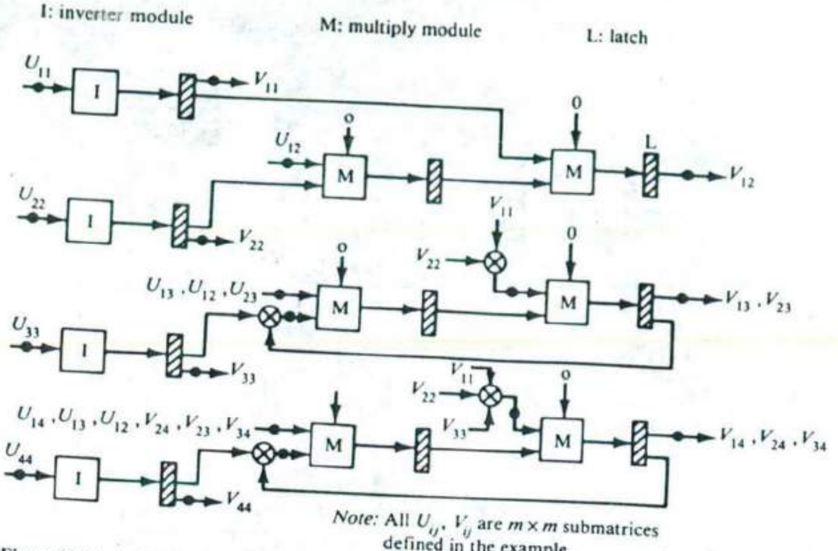


Figure 10.44 A pipelined VLSI matrix inverter (Example 10.5). (Courtesy of *Computer Vision, Graphics and Image Processing*, Hwang and Su, 1983a.)

configurations for the partitioned matrix arithmetic algorithms. In a *strictly parallel* configuration, all submatrix operations at each step are performed in parallel by multiple VLSI modules. This implies minimum time delay in each step. The total time delay among all steps is also minimized by overlapping some step operations. In a *serial-parallel* configuration, the number of available VLSI modules in each step is upper bounded. Thus, some parallel-executable operations

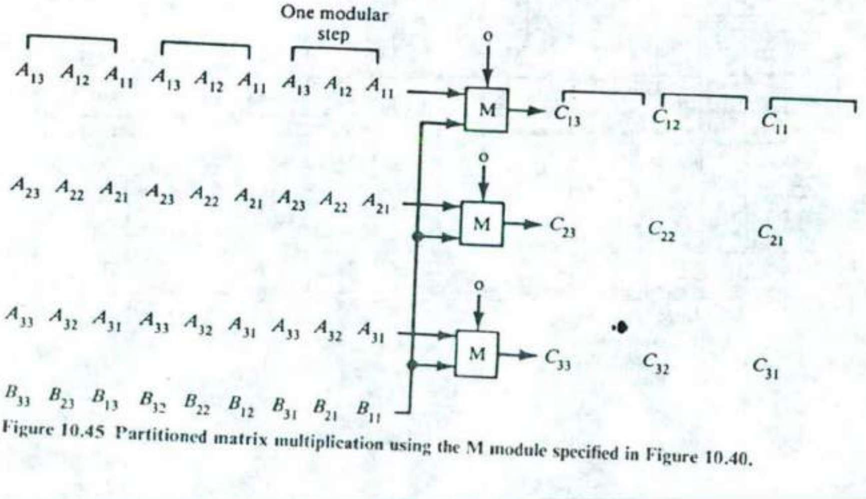
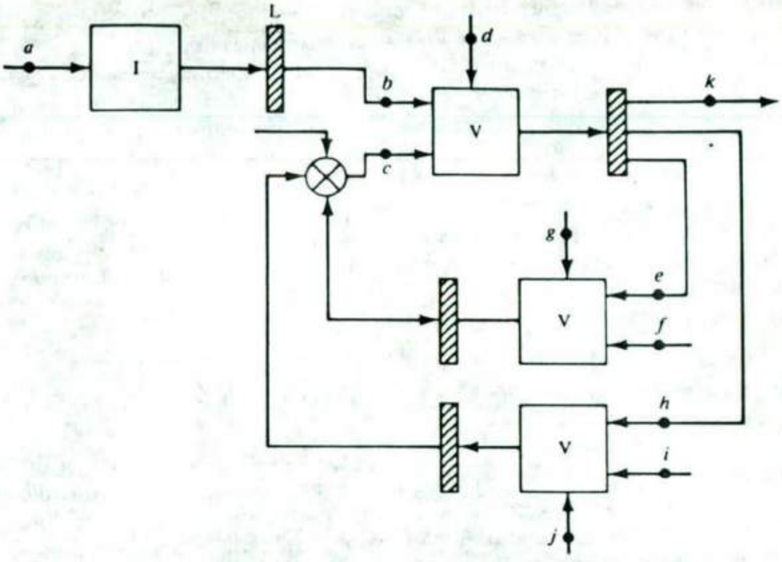

One modular step
A_{13} A_{12} A_{11} A_{13} A_{12} A_{11} A_{13} A_{12} A_{11}
A_{23} A_{22} A_{21} A_{23} A_{22} A_{21} A_{23} A_{22} A_{21}
A_{33} A_{32} A_{31} A_{33} A_{32} A_{31} A_{33} A_{32} A_{31}
B_{33} B_{23} B_{13} B_{32} B_{22} B_{12} B_{31} B_{21} B_{11}
M
C_{13} C_{12} C_{11}
C_{23} C_{22} C_{21}
C_{33} C_{32} C_{31}

Figure 10.45 Partitioned matrix multiplication using the M module specified in Figure 10.40.



	Input terminals										Outputs
Steps	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*
Step 1	U_{44}	U_{44}^{-1}	b_4	0							x_4
Step 2	U_{33}	U_{33}^{-1}	b_3	0	x_4	U_{34}	b_3				x_3
Step 3	U_{22}	U_{22}^{-1}	b_2	0				x_3	U_{23}	b_2	x_2
								x_4	U_{24}		
Step 4	U_{11}	U_{11}^{-1}	b_1	0	x_2	U_{12}	b_1				x_1
					x_3	U_{13}					
					x_4	U_{14}					

Figure 10.46 A VLSI triangular system solver (Example 10.6). (Courtesy of *Computer Vision, Graphics, and Image Processing*, Hwang and Su, 1983a.)

may have to be executed sequentially. Of course, serial-parallel operations will result in longer time delays because of limited hardware modules.

To implement the partitioned L-U decomposition algorithm, we need to use one D module, two I modules, and large number of M modules. The number of needed M modules depends on the chosen architectural configuration. The partitioned L-U decomposition can be realized in $O(n)$ time with $O(n^2/m^2)$ VLSI modules each with interior chip complexity $O(m^2)$. Using a uniprocessor, $O(n^3)$ time steps are needed to perform the L-U decomposition. It is interesting to note that, with the partitioned approach, the triple product of the *chip count* $O(n^2/m^2)$, the *compute time* $O(n)$, and the *chip size* $O(m^2)$ yields the uniprocessor compute time $O(n^3)$; that is:

$$O(n^2/m^2) \cdot O(n) \cdot O(m^2) = O(n^3) \tag{10.15}$$

This property is called *conservation law* between available hardware chips and achievable speed.

The chip count $O(n^2/m^2)$ is too high to be of practical value because of the fact that $n \gg m$. Therefore, we have to bound the chip count with a linear order $O(n/m)$ in a serial-parallel implementation of the partitioned matrix algorithm. One can use $2n/m - 1$ M modules to implement a serial-parallel architecture for L-U decomposition as shown in Figure 10.42. Using $O(n/m)$ modules yields a prolonged time delay $O(n^2/m)$ for $n \gg m \gg 1$.

The conservation law is again preserved in serial-parallel architecture; that is:

$$O(n/m) \cdot O(n^2/m) \cdot O(m^2) = O(n^3) \tag{10.16}$$

Similar analysis can be made to estimate the chip counts and time delays for partitioned matrix inversion and multiplication. In Table 10.2, we show that the first three matrix algorithms can each be implemented by $O(n^2/m^2)$ VLSI modules with $O(n)$ time delays for the strictly parallel architecture, and by $O(n/m)$ modules with $O(n^2/m)$ time for the serial-parallel configuration. Multiplication needs to use M modules exclusively. To solve a triangular LSE, the total time delay is $O(n)$ and $O(n/m)$ VLSI modules are used. The strictly parallel architecture is suggested for constructing a VLSI triangular system solver. Again the conservation law holds as:

$$O(n/m) \cdot O(n) \cdot O(m) = O(n^2) \tag{10.17}$$

where $O(n^2)$ is the compute time of using a uniprocessor to solve triangular system of algebraic equations.

It is obvious that trade-offs exist between the module counts and time delays of all partitioned matrix algorithms. By presetting a speed requirement, one can decide the minimum number of VLSI modules needed to achieve the desired speed performance. On the other hand, one can predict the speed performance under prespecified hardware allowance. This trade-off study is necessary for cost-effective design of large-scale matrix system solvers.

Table 10.2 Hardware requirements and speed performances of partitioned VLSI matrix algorithms

	VLSI architecture and complexity			
	Strictly parallel architectures with minimum time delays		Serial-parallel architecture with bounded number of VLSI modules	
Matrix algorithm	VLSI module count and module types	Total compute time	VLSI module count and module types	Total compute time
L-U decomposition	$O(n^2/m^2)$ D, I, M*	$O(n)$	$O(n/m)$ D, I, M*	$O(n^2/m)$
Inversion of triangular matrix	$O(n^2/m)$ I, M*	$O(n)$	$O(n/m)$ I, M*	$O(n^2/m)$
Matrix multiplication	$O(n^2/m^2)$ M*	$O(n)$	$O(n/m)$ M*	$O(n^2/m)$
Solving triangular linear system of equations	$O(n/m)$ I, V*	$O(n)$		

Note: All measures are based on the assumption $n \gg m \gg 1$, where n is the matrix order and m is the VLSI module size.

* Predominating VLSI modules to be used.

10.4.4 Real-Time Image Processing

A computational model for a statistical image analysis system is illustrated in Figure 10.47. All pattern vectors **x** (the raw patterns to be recognized) form the input space $\mathbf{V}_n$. To design a feature extractor, one has to produce a set of m transformation vectors $\{\mathbf{d}_i | i = 1, 2, \ldots, m\}$ from a set S of training samples with known classes. Each $\mathbf{d}_i$ is an n-dimensional column vector. We denote the jth sample of class s as $\mathbf{x}_j^{(s)}$. The output of the extractor is the feature vector **y**, which is related to the input **x** by a linear transformation $\mathbf{y} = \mathbf{D} \cdot \mathbf{x}$, where $\mathbf{D} = [\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m]^T$ is an $m \times n$ transformation matrix.

VLSI feature extraction Foley and Sammon (1975) have introduced a discriminating method to generate an optimal set of transformation vectors based on maximum separability instead of best picture fitting. The Foley-Sammon algorithm is modified below to allow modular implementation of a feature extractor by the proposed VLSI hardware.

Let n_s be the number of training samples and $\mathbf{m}_s$ be the sample mean for class s. The sample offset $\mathbf{z}_j^{(s)}$ is a column vector formed by the following vector subtraction:

$$\mathbf{z}_j^{(s)} = \mathbf{x}_j^{(s)} - \mathbf{m}_s \qquad \text{for } j = 1, 2, \ldots, n_s$$

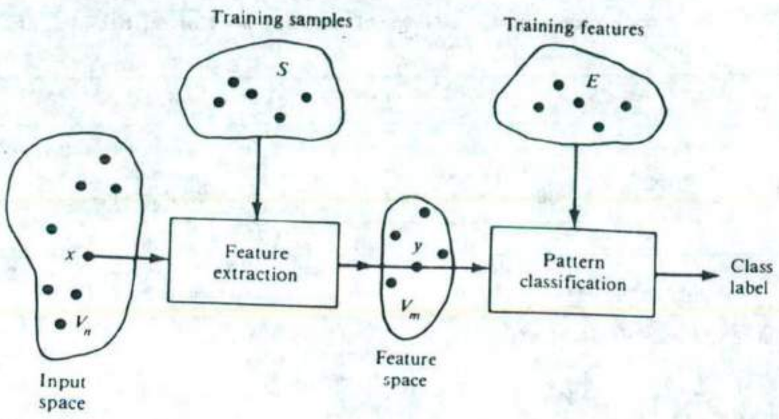


Figure 10.47 A statistical pattern recognition system.

The sample offset matrix $\mathbf{Z}_s$ is an $n \times n_s$ matrix formed by $\mathbf{Z}_s = [\mathbf{z}_1^{(s)}, \mathbf{z}_2^{(s)}, \ldots, \mathbf{z}_n^{(s)}]$. A within-class scatter matrix $\mathbf{S}_s$ is obtained by performing an orthogonal matrix multiplication, where $\mathbf{S}_s$ is an $n \times n$ matrix:

$$\mathbf{S}_s = \mathbf{Z}_s \cdot \mathbf{Z}_s^T \tag{10.18}$$

A weighted scatter matrix **A** is defined below for classes s and t. The fraction C, where $0 \leq C \leq 1$, is determined by a generalized Fisher criterion:

$$\mathbf{A} = C \cdot \mathbf{S}_s + (1 - C) \cdot \mathbf{S}_t \tag{10.19}$$

Let $\mathbf{m} = \mathbf{m}_s - \mathbf{m}_t$ be the mean difference. We define an $h \times h$ matrix $\mathbf{B}_h = (b_{ij})$ for $h = 1, 2, \ldots, m - 1$, where $b_{ij} = \mathbf{d}_i^T \cdot \mathbf{A}^{-1} \cdot \mathbf{d}_j$ for $1 \leq i, j \leq h$. Foley-Sammon algorithm is summarized below:

Foley-Sammon feature extraction algorithm

1. Initialize $i = 1$ and $\mathbf{B}_1^{-1} = b_{11}^{-1}$. Compute $\mathbf{d}_1$ by

$$\mathbf{d}_1 = \alpha_1 \cdot \mathbf{A}^{-1} \cdot \mathbf{m} \tag{10.20}$$

where α_1 is a scalar constant satisfying $\mathbf{d}_1 \cdot \mathbf{d}_1^T = 1$ and

$$\alpha_1^2 = (\mathbf{m}^T \cdot [\mathbf{A}^{-1}]^2 \cdot \mathbf{m})^{-1}$$

2. Increment $i \leftarrow i + 1$. Halt, if $i > m$.
3. Compute the ith transformation vector $\mathbf{d}_i$ by

$$\mathbf{d}_i = \alpha_i \cdot \mathbf{A}^{-1} \cdot (\mathbf{m} - [\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_{i-1}] \cdot \mathbf{B}_{i-1}^{-1} \cdot \mathbf{w}) \tag{10.21}$$

where $\mathbf{w} = [1/\alpha_1, 0, 0, \ldots, 0]^T$ is a column vector with $(i - 1)$ elements, and α_i satisfies $\mathbf{d}_i^T \cdot \mathbf{d}_i = 1$. Go to step 2.

The above computations involve the inversions of **A** and $\mathbf{B}_i = 1, 2, \ldots, m$, which are very lengthy. Instead of performing recursive matrix inversion, a block-partitioning method is used to generate the inverse matrices $\mathbf{A}^{-1}$ and $\mathbf{B}_i^{-1}$. In a feature extraction process, matrix computations include $\mathbf{Z}_s \cdot \mathbf{Z}_s^T, \mathbf{Z}_t \cdot \mathbf{Z}_t^T$, and $[\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_{i-1}] \cdot \mathbf{B}_{i-1}^{-1}$, $\mathbf{A}^{-1}$ and $\mathbf{B}_i^{-1}$ for all $i = 1, 2, \ldots, m - 1$. Figure 10.48 shows the functional design of a VLSI feature extractor. This extractor is constructed with three subsystems: scatter matrix generator, matrix inverter, and feature generator, as shown by dash-line boxes. The vector subtractor is implemented with modified V modules for generating $\mathbf{Z}_s$, $\mathbf{Z}_s^T$, $\mathbf{Z}_t$, $\mathbf{Z}_t^T$, and the mean difference **m** used in the above algorithm. Two matrix multiply networks (Figure 10.44) are used to perform the orthogonal matrix multiplications. Each network contains n/m M modules. The weighted matrix adder can be implemented by n/m M modules with some special constant inputs.

The inversion of the scatter matrix **A** is done by employing an L-U decomposition network (Figure 10.43), two triangular matrix inverters (Figure 10.45), and one multiply network to yield the computation $\mathbf{A}^{-1} = (\mathbf{L} \cdot \mathbf{U})^{-1} = \mathbf{U}^{-1} \cdot \mathbf{L}^{-1}$. The inverse matrices B_i^{-1} for $i = 2, 3, \ldots, m - 1$ can be similarly generated by this matrix inverter. Let $\mathbf{D}_i = [\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_i]$. The matrix multiplications $\mathbf{D}_i \cdot \mathbf{B}_i^{-1}$ are performed by the same matrix multiply network generating the scatter matrix S′. If i/r is not an integer, the matrices $\mathbf{D}_i$ and $\mathbf{B}_i^{-1}$ can also be augmented with zeros and the identity matrix in order to use standard VLSI modules. The above computations are recursively performed by the transformation vector generator. This generator can be implemented by V modules with modified constant inputs. The matrix-vector multiplier is implemented with V modules.

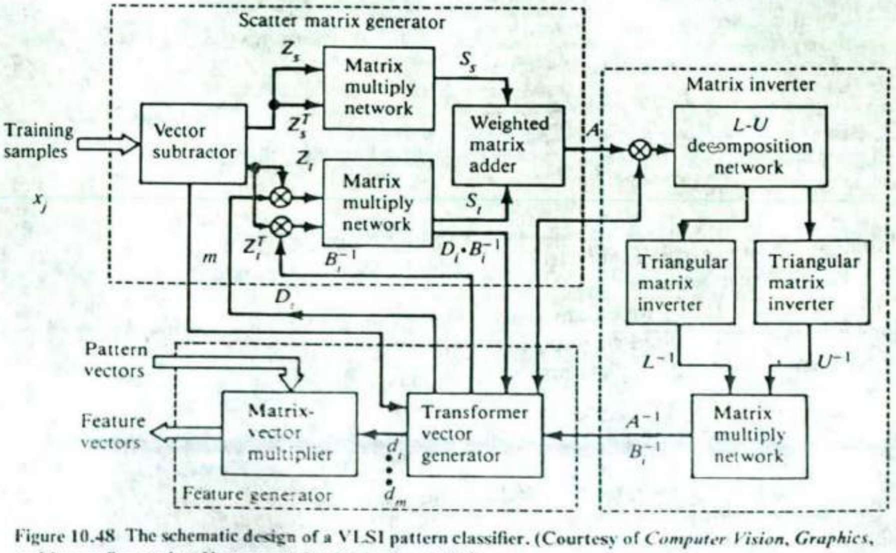


Figure 10.48 The schematic design of a VLSI pattern classifier. (Courtesy of *Computer Vision, Graphics, and Image Processing*, Hwang and Su, November 1983.)

VLSI pattern classification Linear discriminant functions can be used for pattern classification. To distinguish among p pattern classes, $p(p-1)/2$ pairwise discriminant functions are needed. A linear discriminant function between two distinct classes is defined by:

$$F(\mathbf{y}) = \mathbf{v}^T \cdot \mathbf{y} + \alpha \tag{10.22}$$

where $\mathbf{y} = (y_1, y_2, \ldots, y_m)^T$ is the feature vector, $\mathbf{v} = (v_1, v_2, \ldots, v_m)^T$ is called the discriminant vector and α is a scalar threshold constant. The discriminant vector $\mathbf{v}$ and threshold constant α are determined with the aid of a set E of training features with known class labels. The feature pattern $\mathbf{y}$ is classified into class s if $F(\mathbf{y}) \geq 0$, and into class t, if otherwise.

Fisher's method is used to generate the discriminant vector $\mathbf{v}$ from training features. The threshold constant α can be set to "zero" with an appropriate choice of the coordinate system. Let $\mathbf{y}_j^{(s)}$ be the jth training feature vector of class s, for $j = 1, 2, \ldots, m_s$. The feature mean difference is $\mathbf{f} = \mathbf{f}_s - \mathbf{f}_t$ and feature offset vector is $\mathbf{w}_j^{(s)} = \mathbf{y}_j^{(s)} - \mathbf{f}_s$. An $m \times m_s$ feature offset matrix $\mathbf{W}_s = [\mathbf{w}_1^{(s)}, \mathbf{w}_2^{(s)}, \ldots, \mathbf{w}_{m_s}^{(s)}]$ is defined for each class. The covariance matrix for class s is computed by:

$$\sum\nolimits_s = \frac{1}{m_s - 1} \cdot \mathbf{W}_s \cdot \mathbf{W}_s^T \tag{10.23}$$

where $\sum_s$ has dimension $m \times m$. The following computation steps are needed to generate the discriminant vectors needed for pattern classification.

A linear pattern classification algorithm

1. Compute $\mathbf{f} = \mathbf{f}_s - \mathbf{f}_t$ and the feature offset matrices $\mathbf{W}_s$ by subtracting the mean $\mathbf{f}_s$ from each training feature $\mathbf{y}_j^{(s)}$ for $j = 1, 2, \ldots, m_s$.
2. Generate matrices $\sum_s$ and $\sum_t$ using Eq. 10.23.
3. Solve the following linear system of equations to determine the discriminant vector $\mathbf{v}$:

$$(\textstyle\sum_s + \sum_t) \cdot \mathbf{c} = \mathbf{f} \tag{10.24}$$

Functional design of a VLSI pattern classifier based on the above algorithm is sketched in Figure 10.49. The schematic design of the covariance matrix generator is similar to the scatter matrix generator in Figure 10.48. The linear system solver is composed of an L-U decomposition network (Figure 10.43) and a triangular system solver (Figure 10.46). This matrix solver is needed to solve the dense system specified in Eq. 10.24. The Fisher classifier is essentially a threshold decision unit (Eq. 10.22) which can be easily implemented by some combinational logic circuits and the modified V modules. The VLSI approaches to both feature extraction and pattern classification can result in significant speedups of

$$\frac{O(n^3)}{O(n^2/m)} \quad \text{or} \quad \frac{O(n^3)}{O(n)} \tag{10.18}$$

This will make it possible to achieve real-time image processing.

Feature extraction and pattern classification are initial candidates for possible VLSI implementation. VLSI computing structures have been proposed for

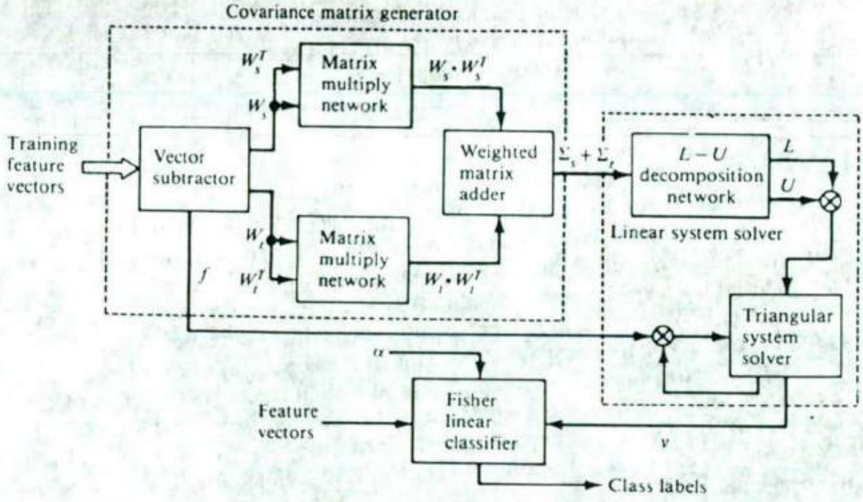


Figure 10.49 The schematic of a VLSI feature extractor. (Courtesy of *Computer Vision, Graphics, and Image Processing*, Hwang and Su, November 1983.)

general signal/image processing applications. Other methods, such as the eigen-vector approaches to feature selection and Bayes quadratic discriminant functions, should also be realizable with VLSI hardware. It is highly desirable to develop VLSI computing structures also for smoothing, image registration, edge detection, image segmentation, texture analysis, multistage feature selection, syntactic pattern recognition, pictorial query processing, and image database management. The potential merit lies not only in speed gains, but also in reliability and cost effectiveness.

10.5 BIBLIOGRAPHIC NOTES AND PROBLEMS

The data flow approach to designing high-performance computers was pioneered by Dennis (1974), among many other researchers. A good distinction between control flow computers and data flow computers can be found in Treleaven et al. (1982). A special issue on data flow system appeared in the IEEE *Computer Magazine* in February 1982. This issue contains several introductory articles on data flow languages, machine architectures, and some critics' opinions. Static data flow machines are described in Dennis et al. (1979) and Dennis (1980). Data flow languages have been studied in Tesler and Enea (1968), Backus (1978), Ackerman and Dennis (1979), Arvind et al. (1978), and Arvind and Gostelow (1982). Potential problems of the data flow approach are assessed by Gajski et al. (1982).

The Irvine data flow machine with tagged tokens is reported in Arvind and Gostelow (1977), Gostelow and Thomas (1980), and Thomas (1981). The

Manchester machine is reported in Gurd and Watson (1980). The MIT dynamic data flow project has been described in Arvind et al. (1980) and Arvind (1983). The EDDY system is described in Takahashi and Amamiya (1983). The French LAU system is described in Syre et al. (1977, 1980). The Utah machine is described in Davis (1978). The Newcastle control data flow machine is described in Treleaven (1978). The dependence-driven approach was proposed by Gajski et al. (1981). The event-driven approach using priority queues was suggested by Hwang and Su (1983c). Packet switching networks for dataflow multiprocessors were treated in Dias and Jump (1981) and in Chin and Hwang (1983).

Systolic array for VLSI computation was suggested by Kung and Leiserson (1978). A review of the systolic architecture can be found in Kung (1982). An assessment of VLSI for highly parallel computing is given by Fairburn (1982). The material on mapping algorithms into VLSI arrays is based on the work by Moldovan (1983). Reconfigurable processor-switch lattices are based on the work of Snyder (1982). Water scale integration of the switch lattice is studied in Hedlund's Thesis (1982). Partitioned matrix algorithms and VLSI image processing structures are based on the work by Hwang and Cheng (1982) and by Hwang and Su (1983a). A wavefront approach to designing cellular processor arrays can be found in S. Y. Kung et al. (1982). The 3-dimensional VLSI architecture was treated in Grinberg et al (1984).

Problems

10.1 Describe the following terms associated with data flow computers and languages:

(*a*) Control flow computers
(*b*) Data-driven computations
(*c*) Static data flow computers
(*d*) Dynamic data flow computers
(*e*) Data flow graphs and languages
(*f*) Single-assignment rule
(*g*) Unfolding of iterative computations
(*h*) Freedom from side effects
(*i*) Dependence-driven computation
(*j*) Coloring technique
(*k*) Event-driven computation

10.2 Draw data flow graphs to represent the following computations:

```
(a) If (a=b) and (c<d)   then c←c−a
                         else c←c+a
(b) For i←1 until m do
    begin
      C(i)←0
      For j←1 until n do
         C[i]←C[i]+a[i,h]*b[j]
    end
(c) Z = N! = Nx(N − 1) × (N − 2)x . . . x 2 × 1
```

You are allowed to use the *merge operator*, the *true gate*, the *false gate*, the *multiply operator*, the *add* or *subtract operator*, the *logical and* the *compare operator* in your graph construction.

10.3 It is desired to construct a packet-switched arbitration network for a static data flow computer similar to the Dennis machine at MIT. Use 5×2 switch boxes as building blocks to construct the network. A unique addressing path is demanded in the network.

(*a*) Design the 5×2 switch box with multiplexers, demultiplexers, and buffers. Switch control mechanism should be explained.

(*b*) Construct a $5^3 \times 2^3$ buffered Delta network with the 5×2 switch boxes to be used for arbitration purpose. Show all the interconnections between stages.

(*c*) There are 243 possible states of a typical 5×2 switch box. Each input port sends one request to be connected to one of the two output ports or none. Assuming that all states are equally probable, derive the blocking probability of a typical 5×2 switch box. When the switch is in a nonblocking state, all requests from input ports are connected to distinct output ports without conflicts. Whenever two or more input requests are destined to the same output port, the switch is entering a blocking state. The blocking probability indicates the chance that a switch may be blocked.

(*d*) Repeat the same question in part (*b*) for a 2×2 switch box. Compare the blocking probabilities between 5×2 and 2×2 switch boxes. Which one has higher blocking probability?

(*e*) Based on the blocking probability found in part (*c*), derive an expression to represent the network blocking probability, if requests from all 125 input ports are equally probable and their destination distribution is uniform among the eight output ports.

10.4 (*a*) Use 8×8 switch boxes to construct a 64×64 routing network for the Arvind machine with 64 PEs. Label all the input-output ports and show all the interconnections among the 8×8 boxes.

(*b*) Show how to join two 64×64 networks to form a network of size 112×112. Some outputs of one network can be connected to some inputs of the network in the joining process.

(*c*) Suppose that each 8×8 switch box has a delay of D. Analyze the delays of the 64×64 network and of the 112×128 network separately, under no blocking conditions.

10.5 Given a sequence of weights $\{\omega_1, \omega_2, \ldots, \omega_k\}$ and an input sequence of signals $\{x_1, x_2, \ldots, \omega_k\}$, design two linear systolic arrays with k processing cells to solve the convolution problem

$$y_i = \omega_1 x_1 + \omega_2 x_2 + \cdots + \omega_k x_k \tag{10.27}$$

(*a*) In the first design, you are given the unidirectional cells which compute $y_{out} \leftarrow y_{in} + \omega \cdot x_{in}$ as shown in Figure 10.50*a*. Explain your design, the distribution of the inputs, and the systolic flow of the partial results y_i's from left to right.

(*b*) In the second design, you are given the bidirectional cells which compute $y_{out} \leftarrow y_{in} + \omega \cdot x_{in}$ and $x_{out} \leftarrow x_{in}$, as shown in Figure 10.50*b*. Explain the design and operation of this systolic convolution array.

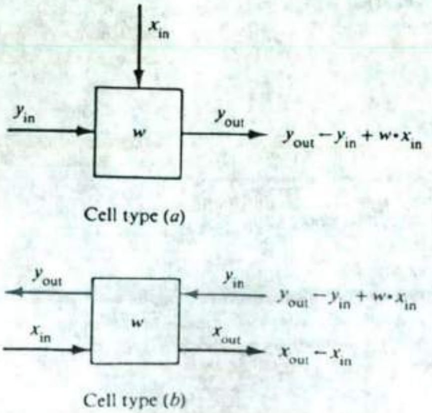


Figure 10.50 Processing cell types for the construction of a linear systolic array for one-dimensional convolution (Problem 10.5).

(*c*) Comment on the advantages and drawbacks in each design:

Note that in both designs (*a*) and (*b*), weights are preloaded to the cells, one at each cell, and stay at cells throughout the computation. In design (*b*), the x_i's and the y_i's move systolically in opposite directions.

10.6 Consider the implementation of the partitioned L-U decomposition algorithm (Figure 10.42) with the VLSI matrix arithmetic modules shown in Figure 10.37. We will analyze the hardware and time complexities: the module count M and the compute time T for chip complexity $C = O(m^2)$ in each of the following architectural configurations:

(*a*) In serial-parallel implementation, prove that $M = 0(n/m)$ and $T = 0(n^2/m)$.

(*b*) For strictly parallel implementation, prove that $M = 0(n^2/m^2)$ and $T = 0(n)$.

In the above proofs, n is the given matrix size and m is the VLSI module size. It is assumed that n is much greater than m.

10.7 Consider the VLSI implementation of an optimal parenthesization algorithm based on integer programming. A string of n matrices is multiplied:

$$M = M_1 \times M_2 \times \cdots \times M_n \tag{10.28}$$

Let $r_0, r_1, \ldots, r_n$ be the dimensions of the n matrices with r_{i-1} and r_i dimensions of M_i. Denote by m_{ij} the minimum cost of computing the product $M_i \cdot M_{i+1} \cdots M_j$. The algorithm which produces m_{in} is given below:

```
for i←1 to n do m_ii←0
for 1←1 to n−1 do
    for i←1 to n−1 do
        begin
            j←i+1
            m_ij←MIN (m_ik+m_k+1,j+r_i−1 r_k r_j)
                  i<k<j
        end
```

Following the mapping procedure in Section 10.3.2, transform the above algorithm into a suitable form which can be implemented by the triangular array shown in Figure 10.50. All the processing cells perform the same functions to be defined in your transformed algorithm.
[*Hint*: One possible transformation of the indices **T**: $(l, i, k) \rightarrow (\hat{l}, \hat{i}, \hat{k})$ is given by $\hat{l} = \max(2l + i - k, l - i + k + 1)$: $\hat{i} = l + k$; and $\hat{k} = -k$.].

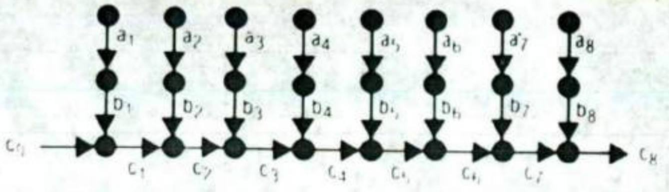
10.8 Develop a cellular array processor for implementing a tridiagonal linear system solver. Specify the cell functions, the VLSI array structure, and the data flow patterns in the cellular array. You have freedom in choosing either a global systolic approach or a modular pipelined approach based on "block" partitioning and back substitution. Comment on the speed and hardware complexities in your design.

10.9 Consider the program graph shown in Figure 10.51. The critical path is $a, b, c_1, c_2, \ldots, c_8$, which results in a lower bound on execution of 13 time units, assuming that division takes 3 time units, multiplication 2, and addition 1. A hypothetical data flow computer has four processing units, each capable of executing any function. We idealize the machine by assuming that memory and interconnection delays are zero. In each of the following machine utilizations, show the schedule (a time-space diagram similar to that in Figure 3.46) of the 24 computing events (8 *divisions*, 8 *multiplications*, and 8 *additions*) and indicate the total execution time and utilization rate of processors.

(*a*) Use only one processor to perform sequential execution, one operation at a time.

(*b*) Use three processors to perform static data flow computations with a one-token-per-arch policy. Note that the three processors can be pipelined to execute a block of statements inside the loop.

(*c*) Use four processors to perform dynamic data flow computations such that tokens are labeled and logical events are colored to share the available resources.



```
input d,e,f
    $c_0 = 0$
for i from 1 to 8 do
  begin
    $a_i := d_i \div e_i$
    $b_i := a_i * f_i$
    $c_i := b_i + c_{i-1}$
  end
output a,b,c
```

Figure 10.51 A program graph for Prob. 10.9. (Courtesy of IEEE *Computer*, Gajski, et al., Feb. 1982.)

(*d*) Apply a random scheduling policy on four processors.

(*e*) Using a control flow vector machine with a "perfect" vectorizing compiler, which could detect the recurrence and achieve full vectorization. Note that intermediate variables can be introduced to completely balance the computing load among the four processors.

Hint: Parts (*c*) and (*e*) should result in the same minimum execution time. However, processor utilization rates would be different.

10.10 Consider a square matrix $B = A + C$, where A, B, and C are all $n \times n$ matrices, the inverse A^{-1} of matrix A is known and the rank of matrix C equals k for some $k \leq n$. You are asked to design a cellular array of processors to find the inverse, B^{-1} of matrix B, given the above conditions.

(*a*) Construct the VLSI processor array to find B^{-1} given A^{-1} and rank $(C) = k = 1$.

(*b*) Modify the design in part (*a*) to find B^{-1} given A^{-1} and rank $(C) = k > 1$.