Multithreading

Unit 3

Introduction to associative memory processor

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

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

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

and the jth bit slice is denoted as:

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

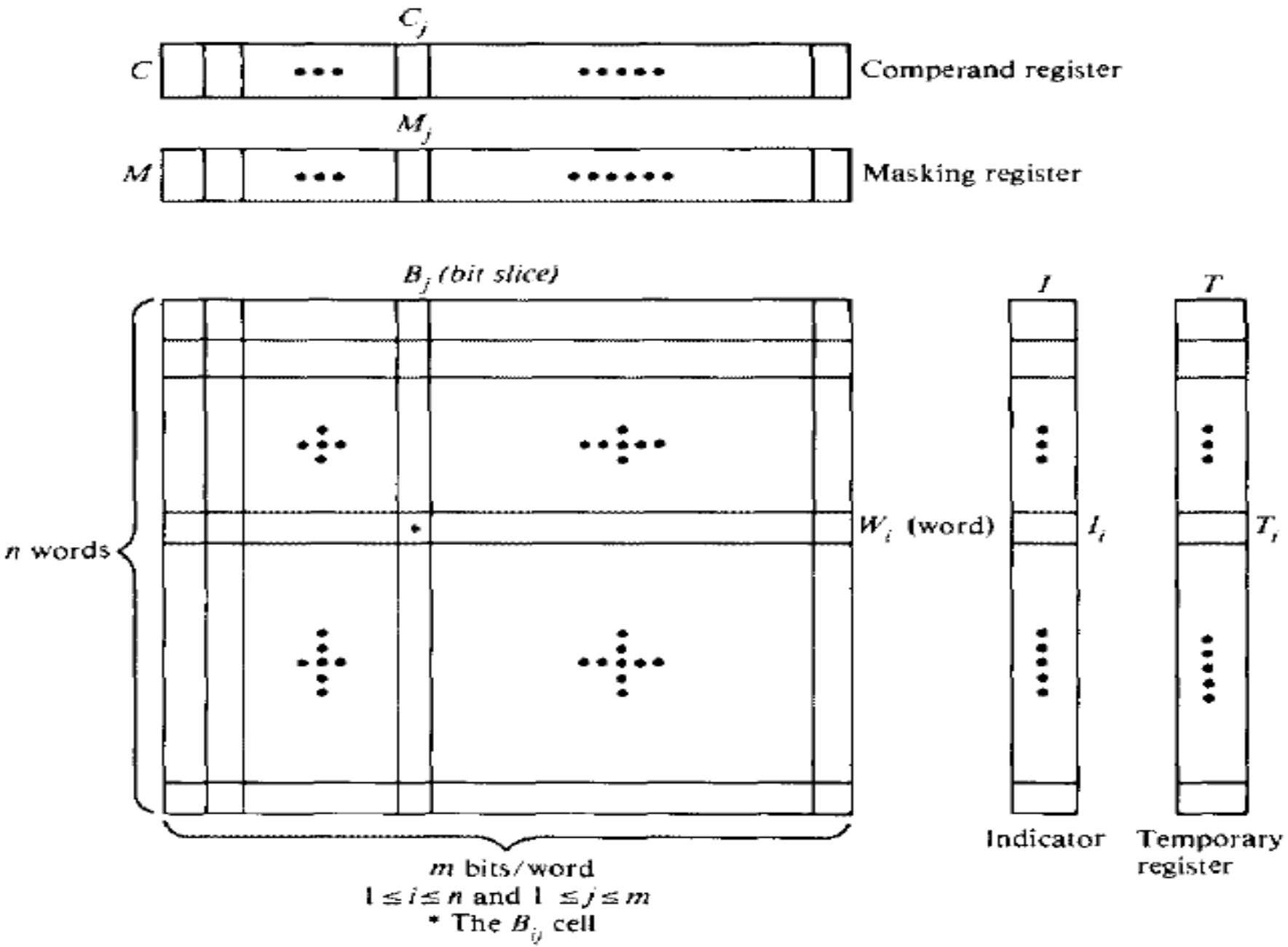


Figure 5.32 An associative memory array and working registers.

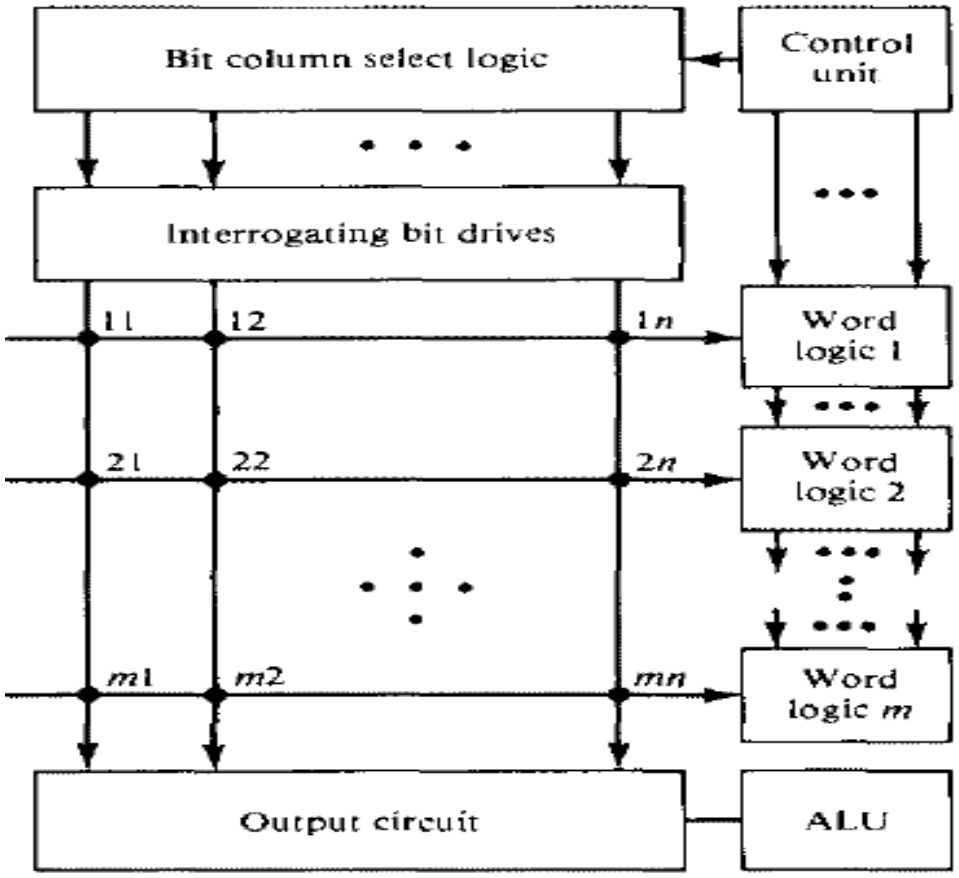
Each bit cell B_{ij} can be written in, read out, or compared with an external interrogating signal. The parallel search operations involve both comparison and masking and are executed according to the organization of the associative memory. There are a number of registers and counters in the associative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

The *indicator* register $I = (I_1, I_2, \ldots, I_n)$ and one or more *temporary* registers $T = (T_1, T_2, \ldots, T_n)$ are used to hold the current and previous match patterns, respectively. Each of these registers can be set, reset, or loaded from an external source with any desired binary patterns. The counters are used to keep track of the i and j index values. There are also some match detection circuits and priority logic.

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

The associative memories are used mainly for search and retrieval of non-numeric information. The bit serial organization requires less hardware but is slower in speed. The bit parallel organization requires additional word-match detection logic but is faster in speed. We present below an example to illustrate the

Associative Memory organization



(b) Bit-serial organization

Figure 5.34 Associative memory organizations.

Associative processor :STARAN

- The bit serial associative processing has been realized in the computer STARAN

consists of up to 32 associative array modules. Each associative array module contains a 256-word 256-bit *multidimensional access* (MDA) memory, 256 processing elements, a flip network, and a selector, as shown in Figure 5.38*a*. Each processing element operates serially bit by bit on the data in all MDA memory

Words

Using the flip network, the data stored in the MDA memory can be accessed through the I/O channels in bit slices, word slices, or a combination of the two. The flip network is used for data shifting or manipulation to enable parallel

STARAN handles parallel processing task

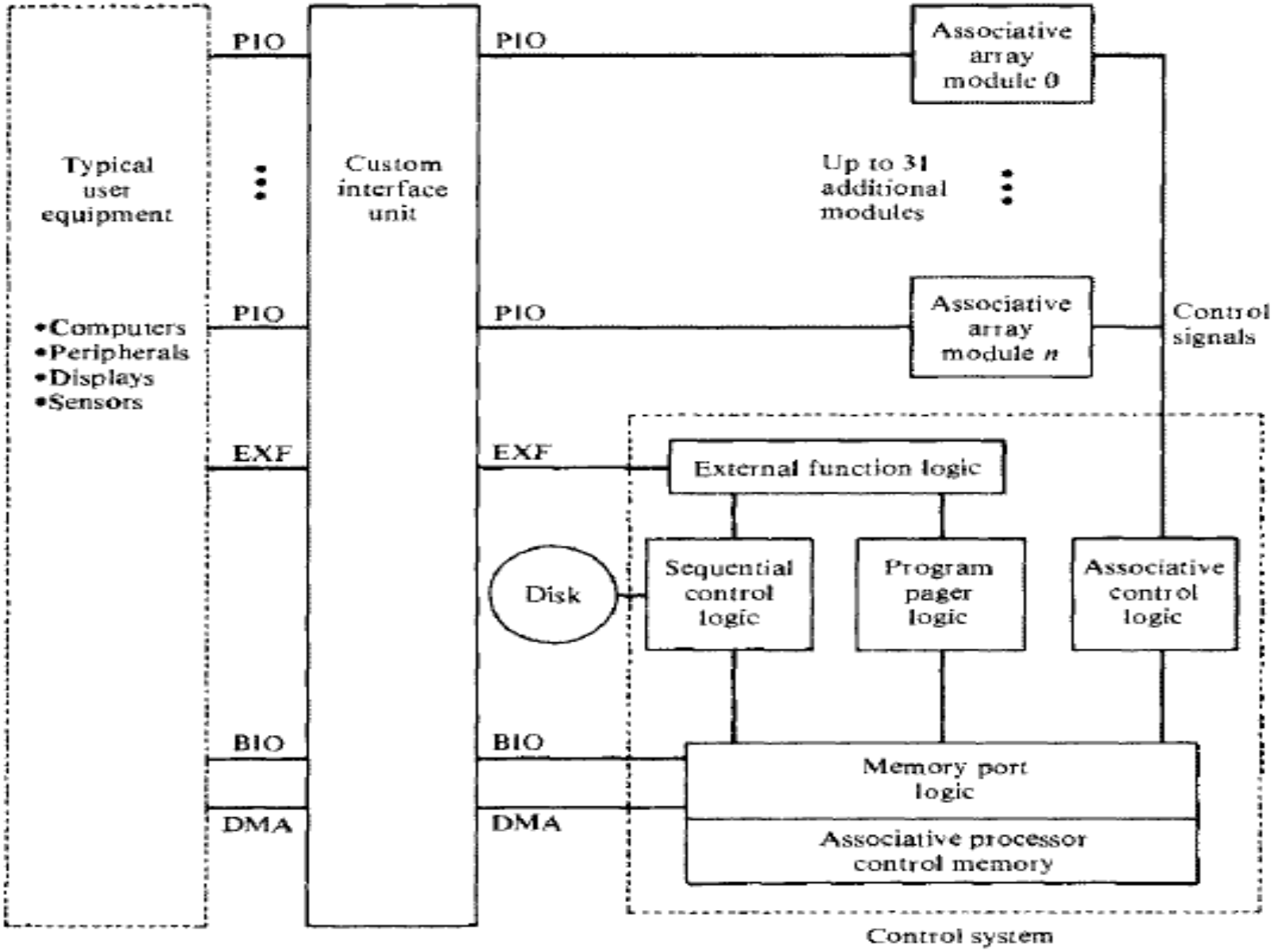


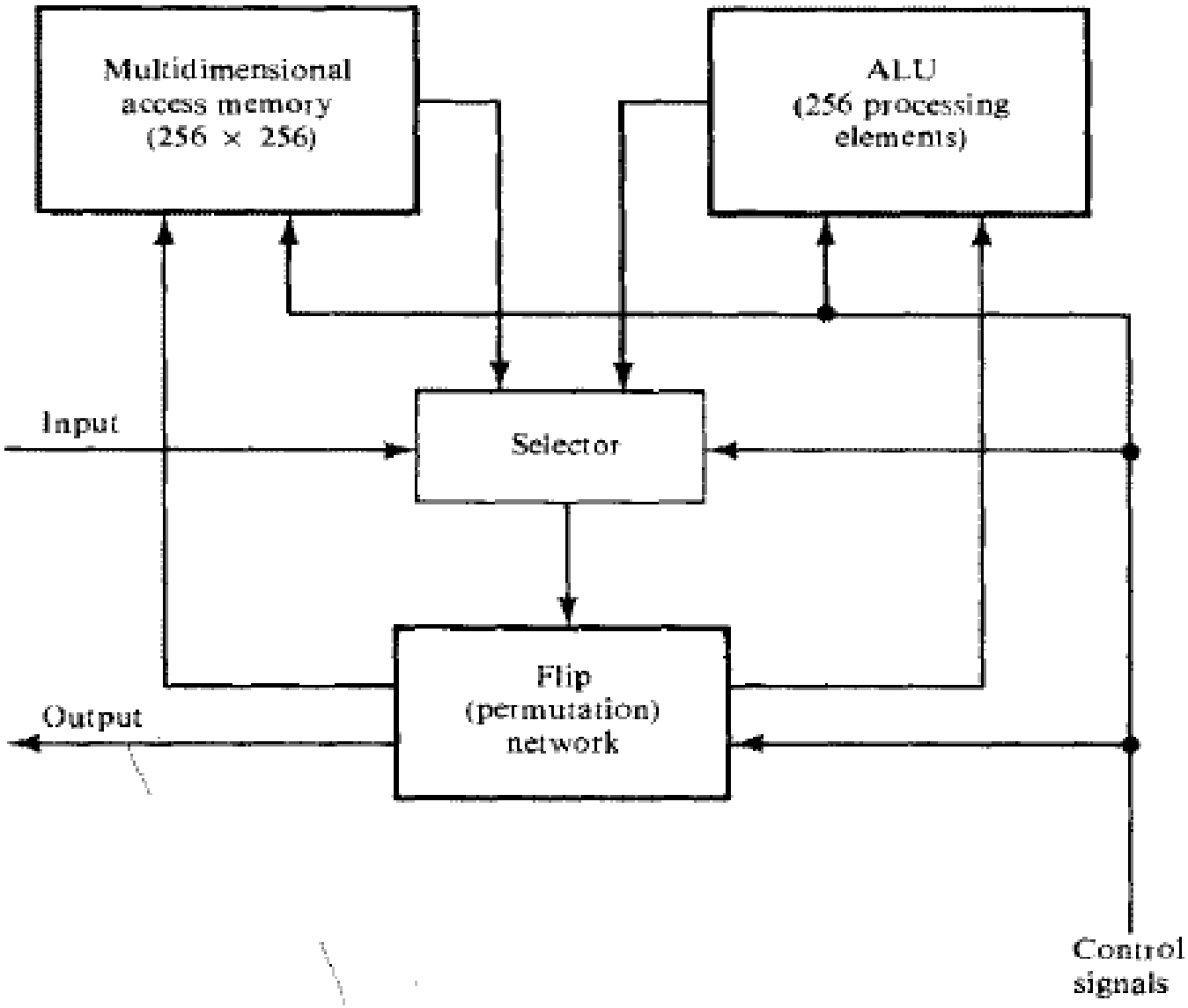
Figure 5.37 The STARAN system architecture. (Courtesy of *Proc. of National Computer Conference*, AFIPS Press, Batcher 1974.)

search, arithmetic or logical operations among words of the MDA memory. The MDA was implemented by Goodyear Aerospace using random-access memory chips with additional XOR logic circuits. The first STARAN was installed for digital image processing in 1975. Since then, Goodyear Aerospace has announced some enhanced STARAN models. The size of the MDA memory has increased to

To locate a particular data item, STARAN initiates a search by calling for a match specified by the associative control logic. In one instruction execution, the data in all the selected memories of all the modules is processed simultaneously by the simple processing element at each word. The interface unit shown in Figure 5.37 involves interface with sensors, conventional computers, signal processors, interactive displays, and mass-storage devices. A variety of I/O options are

Each associative array module can have up to 256 inputs and 256 outputs into the custom-interface unit. They can be used to increase the speed of inter-array data communication to allow STARAN to communicate with a high-bandwidth I/O device and to allow any device to communicate directly with the associative array modules. In many applications, such as matrix computation, air-

s



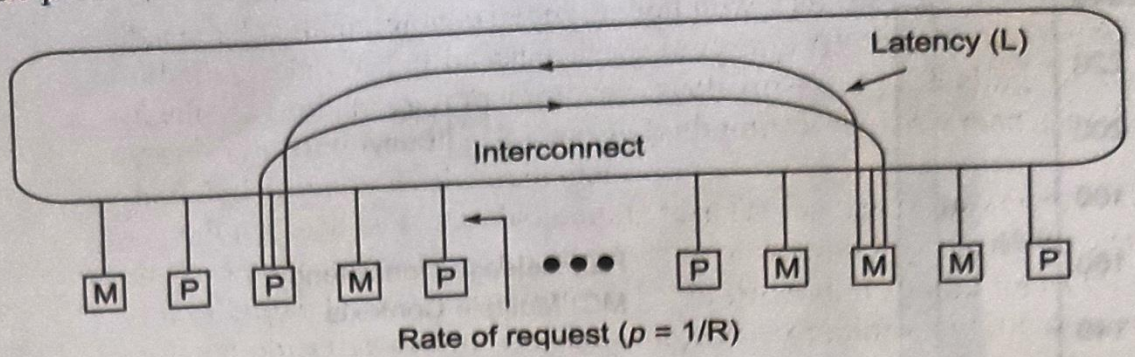
(*a*) STARAN associative array module

Principles of multithreading

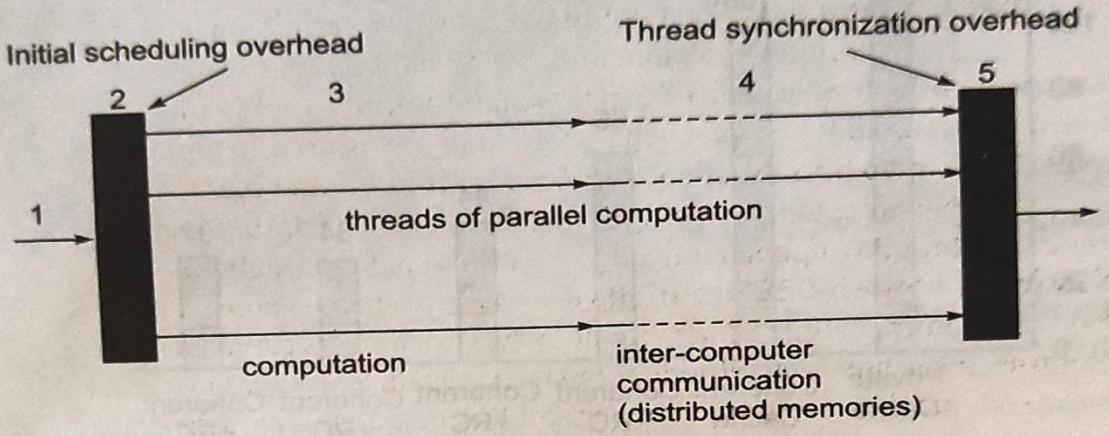
- Multithreading demands that the processor be designed to handle multiple contexts simultaneously on a context switching basis.

Multithreaded architecture

Architecture Environment One possible multithreaded MPP system is modeled by a network of processor (P) and memory (M) nodes as depicted in Fig. 9.11a. The distributed memories form a global address space. Four machine parameters are defined below to analyze the performance of this network:



(a) The architecture environment. (Courtesy of Rafael Saavedra, 1992)



(b) Multithreaded computation model. (Courtesy of Gordon Bell, *Commun. ACM*, August 1992)

Fig. 9.11 Multithreaded architecture and its computation model for a massively parallel processing system

(1) *The latency* (L): This is the communication latency on a remote memory access. The value of L includes the network delays, cache-miss penalty, and delays caused by contentions in split transactions.

(2) *The number of threads* (N): This is the number of threads that can be interleaved in each processor. A *thread* is represented by a *context* consisting of a program counter, a register set, and the required context status words.

(3) *The context-switching overhead* (C): This refers to the cycles lost in performing context switching in a processor. This time depends on the switch mechanism and the amount of processor states devoted to maintaining active threads.

(4) *The interval between switches* (R): This refers to the cycles between switches triggered by remote reference. The inverse $p = 1/R$ is called the *rate of requests* for remote accesses. This reflects a combination of program behavior and memory system design.

In order to increase efficiency, one approach is to reduce the rate of requests by using distributed coherent caches. Another is to eliminate processor waiting through multithreading. The basic concept of multithreading is described below.

Multithreaded Computations Bell (1992) has described the structure of the multithreaded parallel computations model shown in Fig. 9.11b. The computation starts with a sequential thread (1), followed

s

by supervisory scheduling (2) where the processors begin threads of computation (3), by intercomputer messages that update variables among the nodes when the computer has a distributed memory (4), and finally by synchronization prior to beginning the next unit of parallel work (5).

The communication overhead period (4) inherent in distributed memory structures is usually distributed throughout the computation and is possibly completely overlapped. Message-passing overhead (send and receive calls) in multicomputers can be reduced by specialized hardware operating in parallel with computation.

Scalable and Multithreaded architecture

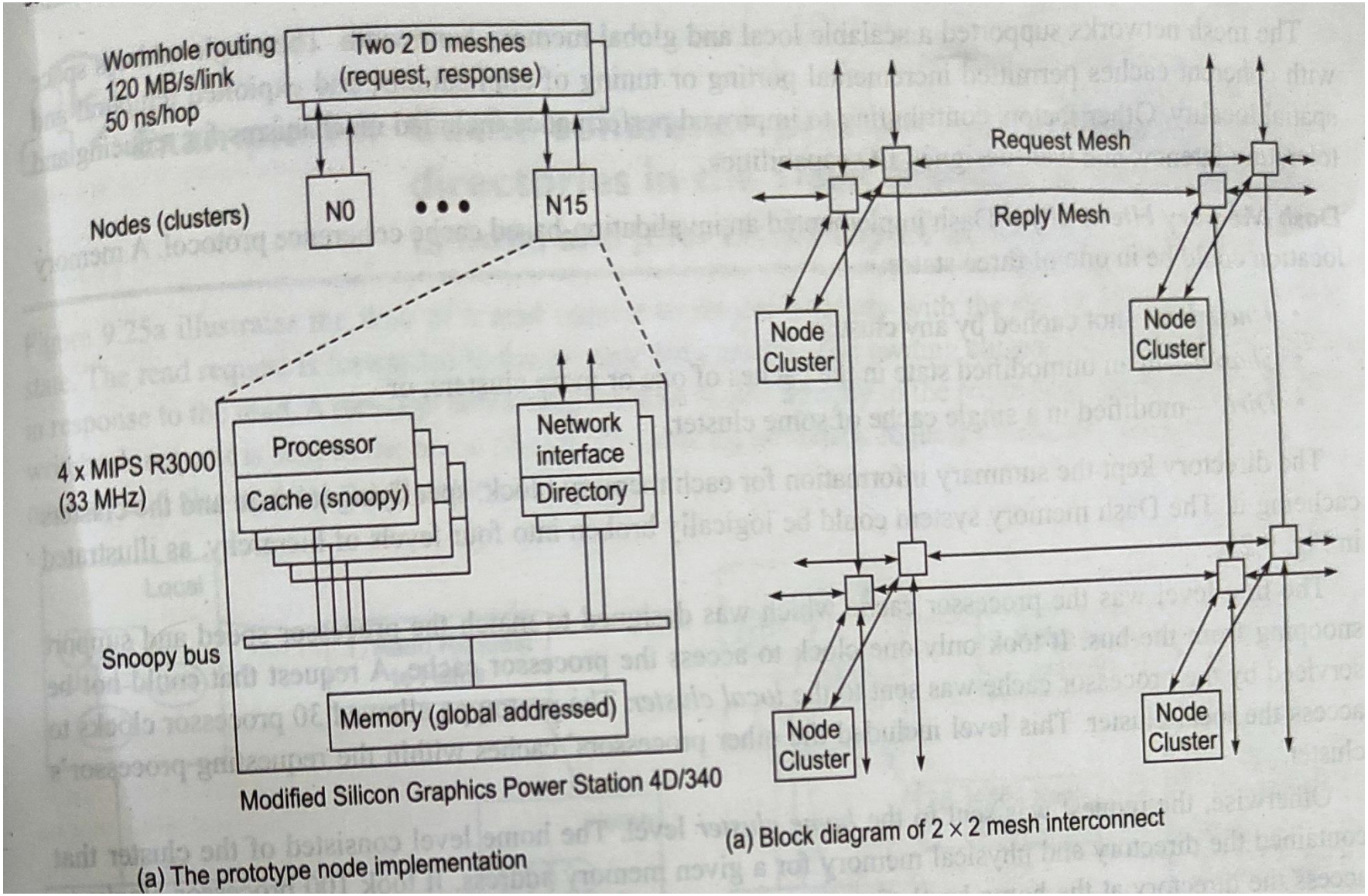
9.4.1 The Stanford Dash Multiprocessor

This was an experimental multiprocessor system developed by John Hennessy and coworkers at Stanford University beginning in 1988. The name Dash is an abbreviation for *Directory Architecture for Shared Memory*. The fundamental premise behind Dash was that it is possible to build a scalable high-performance machine with a single address space, coherent caches, and distributed memories. The directory-based coherence gave Dash the ease of use of shared-memory architectures, while maintaining the scalability of message-passing machines.

The Prototype Architecture A high-level organization of the Dash architecture was illustrated in Fig. 9.1 when we studied the various latency-hiding techniques. The Dash prototype is illustrated in Fig. 9.24. It incorporated up to 64 MIPS R3000/R3010 microprocessors with 16 clusters of 4 PEs each. The cluster hardware was modified from Silicon Graphics 4D/340 nodes with new directory and reply controller boards as depicted in Fig. 9.24a.

The interconnection network among the 16 multiprocessor clusters was a pair of wormhole-routed mesh networks. The channel width was 16 bits with a 50-ns fall-through time and a 35-ns cycle time. One mesh network was used to *request* remote memory, and the other was a *reply* mesh as depicted in Fig. 9.24b, where the small squares at mesh intersections are the 5×5 mesh routers.

The stanford Dash prototype system



s

The Dash designers claimed scalability for the Dash approach. Although the prototype was limited to at most 16 clusters (a 4 × 4 mesh), due to the limited physical memory addressability (256 Mbytes) of the 4D/340 system, the system was scalable to support hundreds to thousands of processors.

To use the 4D/340 in the Dash, the Stanford team made minor modifications to the existing system boards and designed a pair of new boards to support the directory memory and intercluster interface. The main modification to the existing boards was to add a bus retry signal, to be used when a request required service from a remote cluster.

The central bus arbiter was modified to accept a mask from the directory. The mask held off a processor's retry until the remote request was serviced. This effectively created a split-transaction bus protocol for requests requiring remote service.

Latency hiding Techniques

1]prefetching techniques

2]Distributed coherent caches

3]Scalable coherence Interface

1] Prefetching techniques

Prefetching Techniques Prefetching uses knowledge about the expected misses in a program to move the corresponding data close to the processor before it is actually needed. Prefetching can be classified based on whether it is *binding* or *nonbinding*, and whether it is controlled by *hardware* or *software*.

With binding prefetching, the value of a later reference (e.g. a register load) is bound at the time when the prefetch completes. This places restrictions on when a binding prefetch can be issued, since the value will become stale if another processor modifies the same location during the interval between prefetch and reference. Binding prefetching may result in a significant loss in performance due to such limitations.

In contrast, nonbinding prefetching also brings the data close to the processor, but the data remains visible to the cache coherence protocol and is thus kept consistent until the processor actually reads the value.

Hardware-controlled prefetching

1] Prefetching techniques

consistent until the processor actually reads the value.

Hardware-controlled prefetching includes schemes such as long cache lines and instruction lookahead. The effectiveness of long cache lines is limited by the reduced spatial locality in multiprocessor applications, while instruction lookahead is limited by branches and the finite lookahead buffer size.

With software-controlled prefetching, explicit prefetch instructions are issued. Software control allows the prefetching to be done selectively (thus reducing bandwidth requirements) and extends the possible interval between prefetch issue and actual reference, which is very important when latencies are large.

The disadvantages of software control include the extra instruction overhead required to generate the prefetches, as well as the need for sophisticated software intervention. In our study, we concentrate on *non-binding software controlled prefetching*.

s

Benefits of Prefetching The benefits of prefetching come from several sources. The most obvious benefit occurs when a prefetch is issued early enough in the code so that the line is already in the cache by the time it is referenced. However, prefetching can improve performance even when this is not possible (e.g. when the address of a data structure cannot be determined until immediately before it is referenced). If multiple prefetches are issued back to back to fetch the data structure, the latency of all but the first prefetched reference can be hidden due to the pipelining of the memory accesses.

Prefetching offers another benefit in multipro

2]Distributed coherent caches

9.1.3 Distributed Coherent Caches

While the coherence problem is easily solved for small bus-based multiprocessors through the use of snoopy cache coherence protocols, the problem is much more complicated for large-scale multiprocessors that use general interconnection networks. As a result, some large-scale multiprocessors did not provide caches (e.g. BBN Butterfly), others provided caches that must be kept coherent by software (e.g. IBM RP3), and still others provided full hardware support for coherent caches (e.g. Stanford Dash).

Dash Experience We evaluate the benefits when both private and shared read-write data are cacheable, as allowed by the Dash hardware coherent caches, versus the case where only private data are cacheable. Figure 9.4 presents a breakdown of the normalized execution times with and without cacheing of shared data for each of the applications. Private data are cached in both caches.

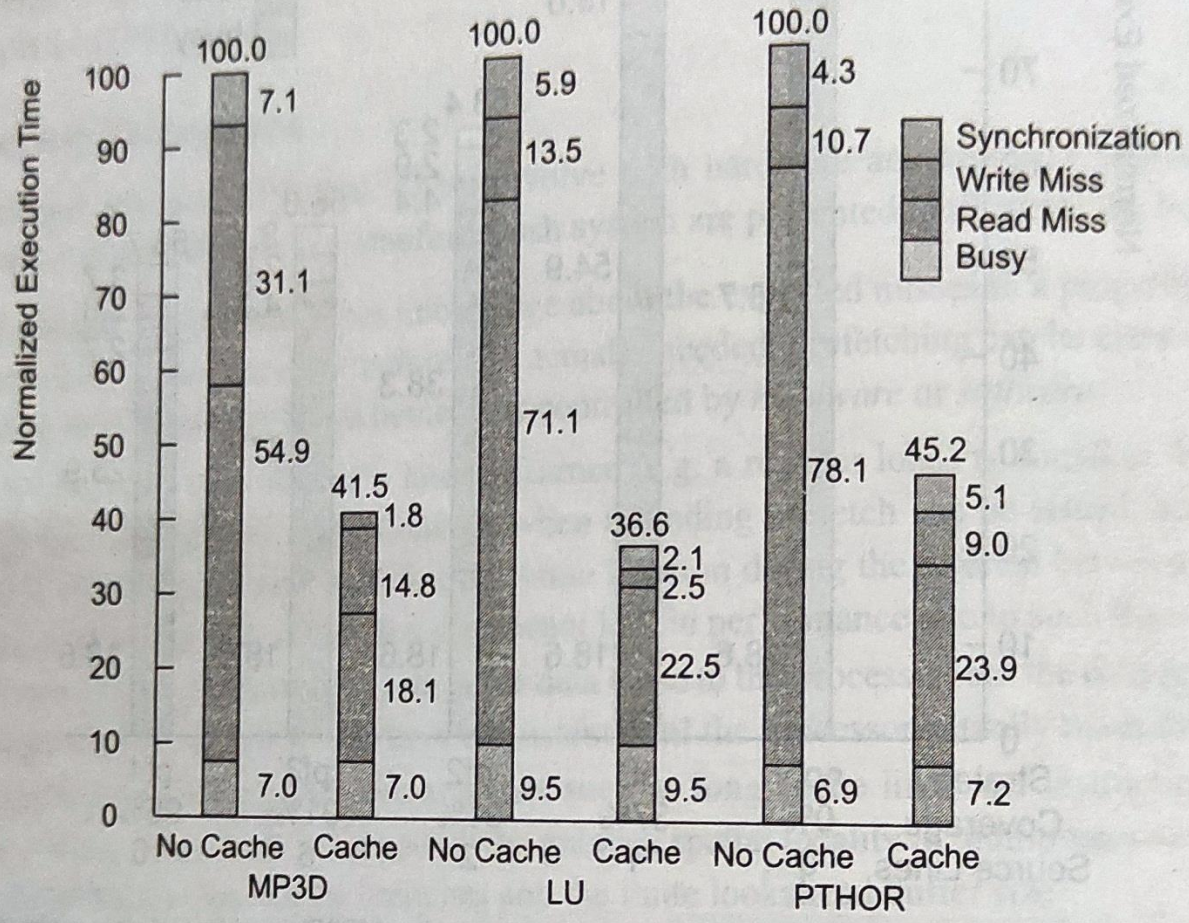


Fig. 9.4 Effect of cacheing shared data in simulated Dash benchmark experiments (Courtesy of Gupta et al, *Proc. Int. Symp. Comput. Archit.*, Toronto, Canada, May 1991)

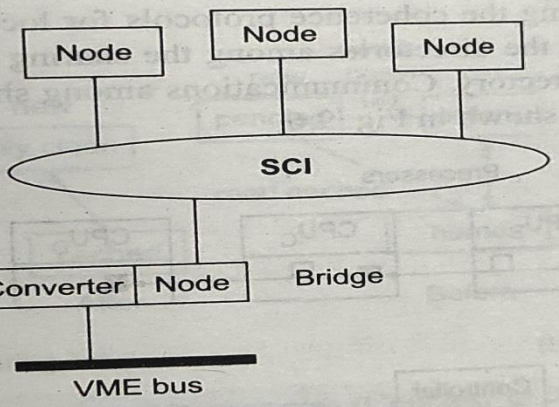
s

The execution time of each application is normalized to the execution time of the case where shared data is not cached. The bottom section of each bar represents the busy time or useful cycles executed by the processor. The section above it represents the time that the processor is stalled waiting for reads. The section above that is the amount of time the processor is stalled waiting for writes to be completed. The top section, labeled "synchronization," accounts for the time processor is stalled due to locks and barriers.

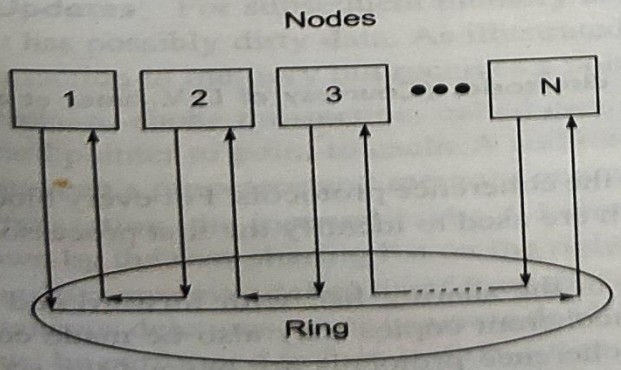
Benefits of Cacheing As expected, the cacheing of shared read-write data provided substantial gains in performance, with benefits ranging from 2.2- to 2.7-fold improvement for the three Stanford benchmark programs. The largest benefit came from a reduction in

3]Scalable coherence Interface

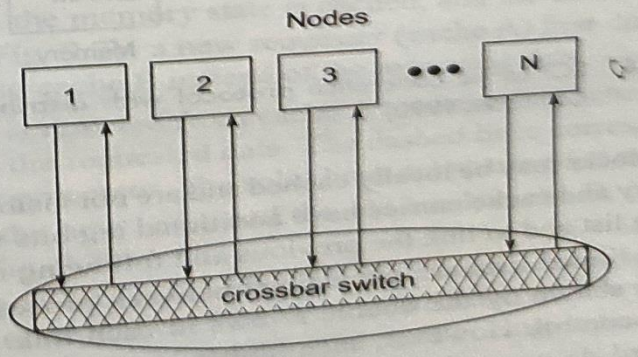
SCI Interconnect Models SCI defines the interface between nodes and the external interconnect, using 16-bit links with a bandwidth of up to 1 Gbyte/s per link. As a result, backplane buses have been replaced by unidirectional point-to-point links. A typical SCI configuration is shown in Fig. 9.5a. Each SCI node can be a processor with attached memory and I/O devices. The SCI interconnect can assume a ring structure or a crossbar switch as depicted in Figs. 9.5b and 9.5c, respectively, among other configurations.



(a) Typical SCI configuration with bridge to other bus



(b) A ring for point-to-point transactions



(b) A crossbar multiprocessor

Fig. 9.5 SCI interconnection configurations (Reprinted with permission from the IEEE Standard 1596-1992, copyright © 1992 by IEEE, Inc.)

s

Each node has an input link and an output link which are connected from or to the SCI ring or crossbar. The bandwidth of SCI links depends on the physical standard chosen to implement the links and interfaces.

In such an environment, the concept of broadcast bus-based transactions is abandoned. Coherence protocols are based on point-to-point transactions initiated by a requester and completed by a responder. A ring interconnect provides the simplest feedback connections among the nodes.

The converter in Fig. 9.5a is used to bridge the SCI ring to the VME bus as shown. A mesh of rings can also be considered using some bridging modules. The bandwidth, arbitration, and addressing mechanisms of an SCI ring significantly outperform backplane buses. By eliminating the snoopy cache controllers, the SCI is also less expensive per node, but the main advantage lies in its low latency and scalability.

Although SCI is scalable, the amount of memory used in the cache directories also scales up well. The performance of the SCI protocol does not scale, since when the sharing list is long, invalidations take