0. ABSTRACT

Since the dawn of computer technology the challenge of closing the computation gap—i.e., the difference between computation power demanded by application areas and the computation power of the computer systems, has introduced some alternatives to the so called traditional von-Neumann concept. These alternatives have proven their effectiveness in practice since the early days of computers. However, in the information explosion era, there is always a demand for higher computation power. For example, several projects currently exist with requirements of 10^9 instructions per second (e.g. 1nsec per instruction) balanced against technologies that are approaching the speed of light transmission limitation (e.g., 30 cm/nsec).

To cope with ever increasing demand to close the computation gap, the design of the computer systems has been advanced in several distinct but interrelated areas

- System software
- Technology and circuit design
- System architecture/organization

The major theme of this unit centers around the contribution of the last approach. However, due to the practical validity and importance of the other directions, the merit of each will also be discussed.

Table 1: The evolution of technology through the computer generation and its impact on computer elements.

	Generation			
Technology	**First**	**Second**	**Third**	**Fourth**
Processor Technology	Vacuum Tube	Transistor	ICS	LSI and VLSI
Processor Structure	Uni-Processor	Multifunctional Units	Multiprocessors Minicomputers MIMD SIMD	Work Stations Local area Networks Extensive SIMD Object Orientation
Main Frame Speed	5 * 10^{4}	5 * 10^{-2}	1	5
Microprocessor Speed	--	--	1	10
Control Unit	Hardwired	Hardwired	Hardwired and Microprogrammed	Hardwired and Microprogrammed
Primary Memory	Vacuum Tubes	Core	Semi Conductor	Semi Conductor 64K-256 K bit chips
Memory size	5 * 10^{-3}	5 * 10^{-2}	1	10
Secondary Memory and I/O Paths	Drum Tape	Channels and Asynchronous I/O Processors	Fixed head and Moveable-arm disks	Extended I/O Paths optical disks
Memory Hierarchy	--	Paging Systems (experiments)	Segmentation & Paging Caches	I/O Caches

1. CLOSING THE COMPUTATION GAP

1.1 System Software

Since the early days of computers, the development of software support for maximizing hardware utility has stimulated much research. Software systems are developed to tailor the embedded hardware features of a system to a specific application. System software includes areas such as data structures, compilers and translators, and operating systems. For example, it has been learned that:

1) By organizing data in proper fashion one can improve the performance. As a classical example we can talk about binary search algorithm over sorted data.

2) A compiler equipped with an optimizer routine improves the performance during the run time by creating an efficient target language program. Extensive research reported in the literature has shown that a vectorizing compiler can enhance the performance by detecting the parallelism in an application program and rearranging the instructions in the object program to allow the simultaneous execution of independent instructions during the run time.

3) Application of memory management routines as part of the operating systems has proven their effectiveness in improving the memory utilization and increasing the system throughput.

1.2 Technology and Circuit Design

During the past 40 years, transition from vacuum tubes to VLSI has increased the processor speed by more than four orders of magnitudes, and has reduced the logic circuit size and memory cell size by factors of 500 and 6400, respectively. The current microelectronics technology has passed the mark of a million transistors per chip, and computer architects are facing the increasing challenge of ULSI—ultra large scale integration—technology. It is anticipated that by the year 2,000, advances in technology will boast the capability of 50-100 million transistors on a chip.

The first electronic computer, designed by Eckert and Mauchly at the University of Pennsylvania, consisted of 18,000 vacuum tubes and 15,000 relays, the U-shaped computer was 100 feet long and nearly 9 feet high and weighed over 30 tons. By comparison, forty years later, vacuum tubes are things of the past replaced by smaller and more reliable units. Today, with the strong emergence of advances in device and storage technology, an entire 32-bit microprocessor or a 1Mbit RAM memory can be incorporated on a single chip. There is no doubt that advances in technology have placed an important role in the organization of computers. Table 1 summarizes the effect of the technology on the main components of the computer system. These advances, however, are not without any drawbacks. For example, recent advances in device technology due to its high design and fabrication costs, and high density (hence the testability issue) at the chip level imposes specific architectural constraints on the designs. A suitable architecture for hardware implementation should reduce communications as well as computation and be based on the replication of a few basic blocks in space or time.

This implies localization, modularity, regularity and simplicity of the designs. To fully utilize these advances one has to develop a design which is: i) general enough for mass production, ii) simple, regular and modular at the chip level, and iii) localize to reduce inter-chip communications. In addition, proper mechanism to improve the fabrication process and chip testability should be developed.

Performance improvement is one of the major challenges in the future advances in device technology. Bipolar Junction Transistors (BJT) for faster speed, CMOS for lower power dissipation, Gallium Arsenide Gates (GaAS) for high electron mobility and optical technology for its speed have shown a promising future.

Wafer Scale Integration (WSI) has been seen as an alternative to VLSI. WSI has the advantage of increasing system speed by eliminating off-chip driver delays and increasing reliability by reducing chip interconnections. However, due to their low yield and power limitations, there are questions whether or not logic WSI can be competitive with ICs. On the other hand, main memory WSI is a promising prospect. First, its efficiency is very high. Since all memory modules on the wafer can be identical and hence, global redundancy can be used. With a word-wide module, power is low since all but one module are in stand-by mode, in addition, pin outs are few. Moreover, in WSI many fabrications steps such as scribing, dicing, sorting, lead bonding, IC insertion, soldering, board testing and reworking are eliminated. All these considerations lead to easily produced wafer having inexpensive packaging.

Since the early 1970s, continued demands for high capacity and low cost storage media has allowed the:

1) Steady advances in magnetic recording technology,

2) Introduction of the so called "electronic disks" — i.e., charge coupled devices and magnetic bubble memory, and

3) Application of the optical storage devices in the memory hierarchy.

Unfortunately, the inability to mass produce the electronic disks made it impossible for competitive challenge in the market. However, optical storage devices and parallel disks have shown a very promising future.

Advances in technology have directly affected the execution time of the basic functions. For example, in 1944 the MARK I (a pioneer computer using electromechanical components, i.e. relays) required 333 msec. to complete an addition. In a few years, this was improved by a factor of more than a thousand (i.e. ENIAC which used vacuum tubes). About 10 years later CDC 6600 was able to perform the addition in 300 nsec.

The reduction in the gate switching delay, wire length and miniaturization of circuits has a direct effect on the clock rate and hence, in closing the computation gap. In addition, such a trend has increased the sophistication of the supporting software and the migration of the software functions into hardware.

In the late 1960's Moore predicted that, component density on a chip is quadrupling every three or four years. This is partially due to the development of high resolution lithographic techniques, increases in the size of the silicon wafer, growth of the accumulated circuits and layout design experience, and better understanding of the system level design issues leading to an improved architecture capable to exploiting the technology. However, as improvements in

technology approach the limit (the speed of light), Moore's law is no longer applicable and the emphasis is shifted in the direction of advances in system architecture/organization.

1.3 System Architecture/Organization

Migration of the software functions into the hardware, combined with the advances in the technology's intrinsic speed, has reduced the computation gap. However, there has always been the need for much more computer performance than is feasible with a simple straight forward design. To overcome these limitations, computer designers have long been attracted to techniques that are classified under the general phrase of concurrent operations. In a concurrent system the computer's hardware is simultaneously processing more than one basic operation at each instant of time. Within this general category are several well recognized techniques such as parallelism, pipelining and multiprocessing. Although these techniques have the same origin and are often hard to distinguish, in practice they are different in their general approach. For example, in parallelism concurrency is achieved by replicating the hardware structure many times, while pipelining takes the approach of splitting the function to be performed into smaller pieces and allocating separate hardware to each piece (stage).

This unit is aimed at different proposals for closing the computation gap within the scope of the control flow environment. Section 2 introduces some definitions and background which are used throughout the unit. Section 3, addresses the class of concurrent systems based on the control flow concept. It represents the so called parallel, pipelined, and multiprocessor systems. The programming issue of the control flow computation is the subject of Section 4. Finally, the shortcoming of the control flow environment is discussed in Section 5.

2. DEFINITIONS

2.1 Concurrency

Concurrency is a generic term to define the ability of the computer hardware to simultaneously execute many actions at any instant. In this general sense it implies parallelism, simultaneity, and pipelining. Parallel events may occur in multiple resources during the same time interval; simultaneous events may occur at the same time instant and pipelined events may occur in overlapped time spans. Within this framework, concurrent processing is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Therefore, one can define a concurrent processor as a system that emphasizes concurrent processing.

2.2 System Utilization

For any computer there is a maximum number of bits or bit pairs that can be processed concurrently whether it is under single-instruction or multiple-instruction control. This maximum degree of concurrency, or **maximum concurrency**, C_m is an indication of the computer processing capability. The actual utilization of this capability is indicated by the **average concurrency** defined to be

$$C_a = \frac{\sum C_i \, \Delta t_i}{\sum \Delta t_i} \qquad (1)$$

where C_i is the concurrency at Δt_i. If Δt_i is set to one time unit, then the average concurrency over a period of T time units is

$$C_a = \frac{\sum_{i=1}^{T} C_i}{T} \tag{2}$$

The **average hardware utilization** is then

$$m = \frac{C_a}{C_m} = \frac{\sum C_i \times \Delta t_i}{C_m \times \sum \Delta t_i} = \frac{\sum_{i=1}^{T} C_i}{C_m \times T} = \frac{1}{T}\sum_{i=1}^{T} \boldsymbol{d}_i \tag{3}$$

where ?$_i$ is the hardware utilization at time i. While C_m is determined by the hardware design, C_a or μ is highly dependent on the software and applications. A general-purpose computer should achieve a high μ for as many applications as possible, while a special-purpose computer would yield a high μ for at least the intended applications. In either case, maximizing the value of μ for a computer design is important. Equation (3) can also be used to evaluate the relative effectiveness of machine designs.

For a parallel processor the degree of concurrency is called the degree of parallelism. A similar discussion can be used to define the average hardware utilization of a parallel processor. The **maximum parallelism** is then P_m and the **average parallelism** is:

$$P_a = \frac{\sum P_i \times \Delta t_i}{\sum \Delta t_i} = \frac{\sum_{i=1}^{T} P_i}{T} \tag{4}$$

for T time units. The **average hardware utilization** of a parallel processor becomes

$$u = \frac{P_a}{P_m} = \frac{\sum P_i \times \Delta t_i}{P_m \times \sum \Delta t_i} = \frac{\sum_{i=1}^{T} P_i}{P_m \times T} = \frac{1}{T}\sum_{i=1}^{T} \boldsymbol{r}_i \tag{5}$$

where ρ_i is the hardware utilization for the parallel processor at time i. With appropriate instrumentation, the average hardware utilization of a system can be determined.

In practice, however, it is not always true that every bit or bit pair that is being processed results in productive information. Some of the bits produced contain only repetitious (superfluous) or even meaningless results. This happens more often and more severely in a parallel processor than in a word-sequential processor. Consider, for example, performing a maximum search operation in a mesh connected parallel processor (such as ILLIAC IV). For N operands it takes $(N/2)\log_2 N$ comparisons ($N/2$ comparisons for each $\log_2 N$ iterations) instead of the usual N-1 comparisons in word-sequential machines. Thus, in effect there are

$$(N/2)\log_2 N - (N-1) = (N/2)(\log_2 N - 2) + 1 \tag{6}$$

comparisons which are non-productive. If we let $\tilde{p}_a$ be the effective parallelism over a period of T time units, and $\tilde{n}$, $\tilde{p}_i$ and $\tilde{r}_i$ be the corresponding effective values, the effective hardware utilization is then

$$\tilde{u} = \frac{\tilde{P}_a}{P_m} = \frac{\sum \tilde{P}_i \times \Delta t_i}{P_m \times \sum \Delta t_i} = \frac{\sum_{i=1}^{T} \tilde{P}_i}{P_m \times T} = \frac{1}{T} \sum_{i=1}^{T} \tilde{r}_i \qquad (7)$$

A successful parallel processor design should yield a high $\tilde{n}$, as well as the required throughput for, at least, the intended applications. This involves not only a proper hardware and software design, but also the development of efficient parallel algorithms for these applications.

Suppose T_u is the execution time of an application program using a conventional von-Neumann machine, and T_c is the execution time of the same program using a concurrent system, then the **speed up** ratio is defined as:

$$S = T_u/T_c \qquad (8)$$

Naturally for a specific concurrent organization the speed up ratio determines how well an application program can utilize the hardware resources. Supporting software has a direct effect on the speed up ratio.

In case of pipeline organization, the literature has used other parameters to discuss the performance issues. In a pipeline system the term **Latency** is used as a performance measure. Latency (L) is defined as the number of time units separating two successive initiations of events. Naturally, the lower the latency the higher the performance. Latency could be any integer value including zero. Then the **average latency** is defined as the average number of time unit between two initiations. Based on the value of the then one can define the initiation rate (I). It is the average number of the initiations per clock unit:

$$I = \frac{1}{(\tilde{L})} \qquad (9)$$

For stage S_i, **stage utilization** ($^U S_i$) indicates on the average how often S_i has been used:

$$^U S_i = I * n_i \qquad (10)$$

where n_i represents the number of time S_i is used in one initiation. For a linear pipe, if δ_i denotes the execution time of stage S_i then:

$$(\tilde{L}) = MAX(\delta_i) \qquad 1 \le i \le k$$

$$I = \frac{1}{MAX(\delta_i)}$$

$$U_{S_i} = \frac{1}{MAX(\delta_i}$$

2.3 Classifications of Concurrent Systems

System designers have long recognized the intrinsic limitation and vulnerability of the classical von-Neumann design, in which all system resources are clustered around a single central processing unit. In this classical model, the arithmetic logic unit (ALU) can perform operations only on a bit or a bit pair (serial ALU). Therefore, an operation on an M-bit operand or operand pair must be repeated bit serially *M* times. In order to speed up the processing, a parallel ALU is usually used, so that all bits of an operand or operand pair can be operated on simultaneously. This discussion can be extended to the cases in which either: i) all the ith bits of *n* operands or operand pairs may be operated on simultaneously (i.e. bit slice-Bis), or ii) the operation is performed on *n* M-bit operands or operand pairs. Points *A*, *B*, *C* and *D* in Figure 1 illustrate these four approaches, respectively. This discussion represents Feng's classification, where the concurrent space is identified as a two dimensional space based on the bit and word multiplicities. Figure 1 shows the allocation of some of the computer architectures in the Feng's concurrent space.

Since the early days of computer technology researchers have attempted to classify various proposed/designed computer architectures. These efforts were directed to: (i) generalize and identify the characteristics of different designs, (ii) formulate a systematic mechanism by which different designs can be analyzed and compared against each other, and (iii) define a systematic mechanism to transform the solutions from one design to other designs.

Flynn has classified the concurrent space according to the multiplicity of instruction and data streams:

I = {Single Instruction Stream (SI), Multiple Instruction Stream (MI)}
D = {Single Data Stream (SD), Multiple Data Stream (MD)} (11)

The cartesian product of these two sets will define four different classes:

I*D = {SISD, SIMD, MISD, MIMD} (12)

SISD This class represents the classical von-Neumann architecture (with serial or parallel ALU)

SIMD This class represents the multiple ALU type architectures (e.g. array processor)

MISD This class is not found to be as practical as the other classes. A database machine—search processor—represents a model in this class.

MIMD This class represents the multiprocessor system (loosely or tightly coupled).

Flynn's classification suffers from the fact that it does not uniquely identify a specific organization. In addition, it does not address the interactions among the processing modules and the methods by which processing modules in a concurrent system are controlled. In general a classification scheme should:

i) categorize all existing as well as foreseeable computer designs,
ii) differentiate essential processing elements, and
iii) assign an architecture to a unique class.

Handler has extended Feng's concurrent space by a third dimension, namely the number of control units. Handler's space is defined as t = (k, d, w) in which:

k is the number of control units (CUs) interpreting a program,

d is the number of arithmetic and logic units (ALUs) controlled by a control unit, and

w is the word length or number of bits handled in one of the ALUs.

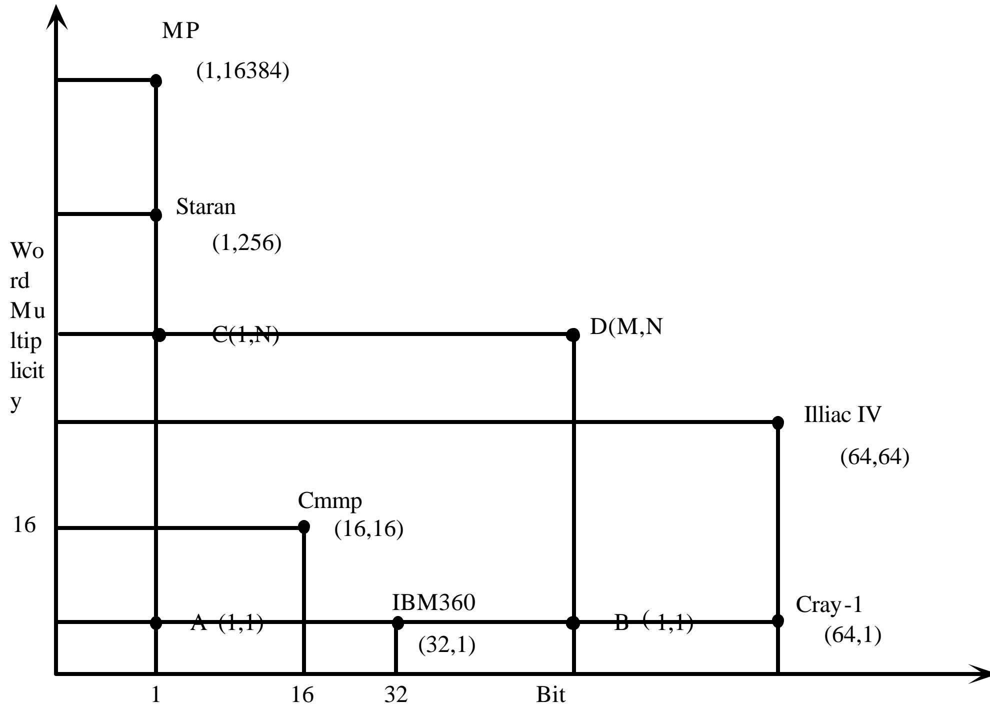


Figure 1. Feng's Classification.

According to this classification a von-Neumann machine with serial/parallel ALUs is represented as (1, 1, 1), (1, 1, M), respectively. Figure 2 depicts the position of some of the computer systems in the Handler space. To represent pipelining at different levels (e.g. macro pipeline, instruction pipeline and arithmetic pipeline), and illustrate the diversity, sequentiality and flexibility/adaptability of an organization, the above triplet has been extended by 3 variables (e.g. k', d', w') and 3 operators (e.g. +, *, v) where:

- **k'** represents the macro pipeline - the number of control units interpreting the tasks of a specific program, where the data flow through them is sequential.
- **d'** represents instruction pipeline - the number of functional units managed by one control unit and working on one data stream.
- **w'** represents arithmetic pipe - the number of stages.
- **+** represents diversity - existence of more than one structure
- ***** represents sequentiality - for sequentially ordered structures
- **v** represents flexibility/adaptability - for reconfigurable organization

According to this extension to Handler's notation, CDC 7600 and DAP are represented as:

(15*1, 1*1, 12*1)*(1*1, 1*9, 60*1) and

(1*1, 1*1, 32*1)*[(1*1, 128*1, 32*1) v (1*1, 4096*1, 1*1)], respectively.

These classifications suffer from the fact that either, they do not uniquely identify a specific organization, or, they can not thoroughly determine the interrelationships among different modules in an organization. For example, Flynn's classification does not address the interactions among the processing modules and the methods in which processing modules in a concurrent system are controlled. As a result, one can classify a pipeline computer and a uni-processor computer as SISD machines, since both instructions and data are provided sequentially. And,

according to the Feng's classification a word organized array processor falls in the same region as a multiprocessor system. In the following, we classify the conventional concurrent systems into three groups - namely, **parallel SIMD, pipeline** and **multiprocessors**. Our distinction is according to the exploitation of concurrency and the interrelationships among the control unit, processing elements and memory modules in each of the aforementioned groups. As will be discussed later, each group is further divided into subsections. Table 2 shows this taxonomy. By a close observation one can realize the progression trend in the development of the parallel and multiprocessor systems. Practically, both techniques achieve concurrency as the result of hardware replication (e.g., redundancy). However, in a multiprocessor system, the degree of freedom associated with the processors is much higher than the one in the parallel systems. As a result, processors are more independent with respect to each other and the central control unit. This independence naturally will introduce a degree of complexity on the dynamic communication capability among the processing elements. In addition, this complexity will be reflected in the control structure and software supports which is needed for each approach in order to map application programs into the hardware features. This discussion can be traced in the evolution the distributed systems, where the processing units are more independent from each other than the processing units in a multiprocessor system.

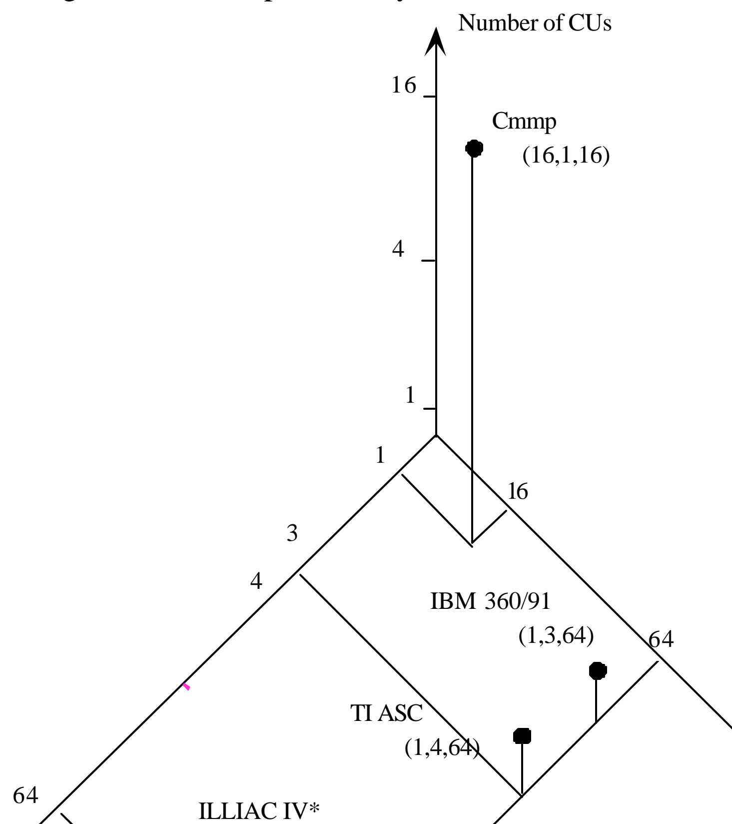
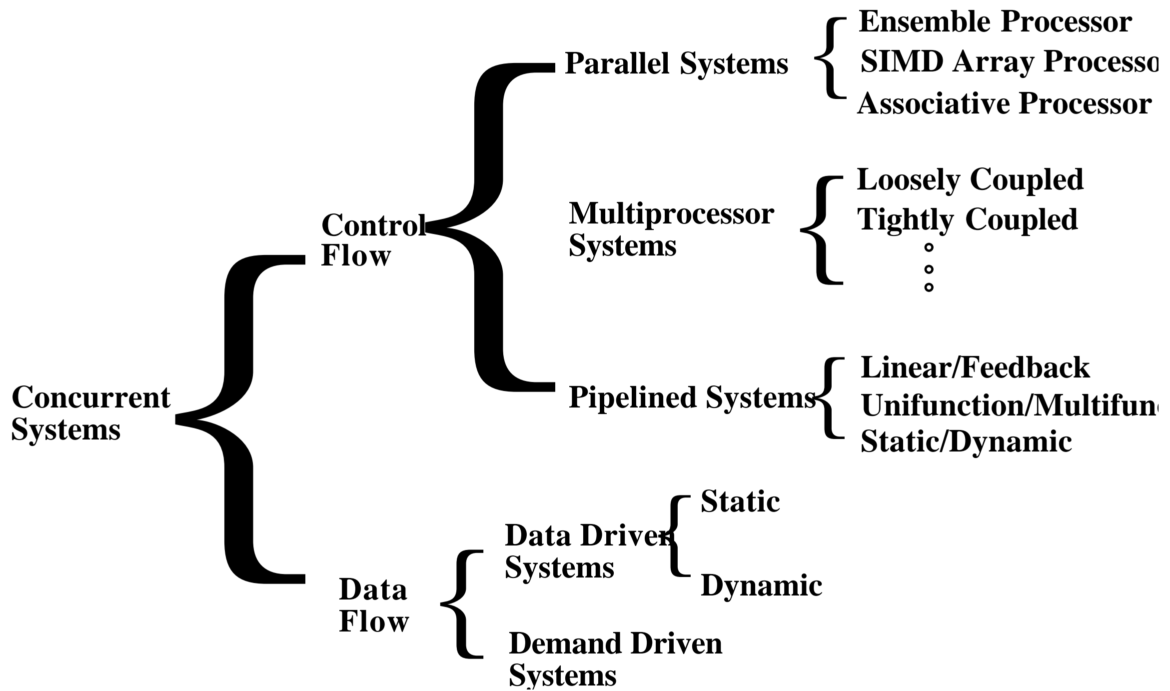


Figure 2. Handler's Classification.

Table 2. Classification of the Concurrent Systems.



2.4 Self Test Problems #1

1) a) Define the term **Technology Driven Architecture**.

b) Name and discuss two classes of the so called technology driven architectures (be sure to address the architectural features of each class and the characteristics of the underlying technology).

2) Different researchers have attempted to classify computer organizations/ architectures:

a) What are the motivations behind the classification of computer systems?

b) What should be the goals of a classification scheme?

3) Hardware utilization for a parallel system is defined as:

$$\gamma = \frac{P_a}{P_m} \quad \text{where } P_a \text{ (average parallelism)} = \frac{\Sigma P_i \Delta t_i}{\Sigma \Delta t_i} = \frac{\sum_{i=1}^{T} P_i}{T}$$

P_i is the parallelism at time slice i and P_m is the maximum parallelism. For a mesh connected parallel processor — i.e., ILLIAC IV type organization — calculate the hardware utilization when performing a maximum search operation.

3. CONTROL FLOW ORGANIZATION

As mentioned before, the classifications cited in Section 2 suffer from the fact that either they do not uniquely identify a specific organization or they can not determine the interrelationships among different modules in an organization. In this Section, we classify the conventional (e.g. control flow) concurrent systems into three groups - namely, **parallel**, **pipeline** and **multiprocessors**. This distinction is due to the exploitation of concurrency and the interrelationships among the control unit, processing elements and memory modules in each group.

Despite our distinction, system designers in the 1960's witnessed the emergence of a new class of computer organizations known as multifunctional unit systems. Although such a computational model is classified as an SISD organization in Flynn's classification, this concept was clearly made in response to closing the computation gap. The CDC-6600, IBM 360/91, CDC 7600, CRAY, NEC SX-2, Fijitsu VP-200, and Hitachi S-810 are classical examples of this organization.

A block diagram of a multifunctional system is shown in Figure 3. The system consists of a single control unit and a processor. However, the processor unit is composed of several functional units. Each functional unit has a set of local registers and all functional units share a set of global registers, which hold intra-functional operands and act as a buffer for memory units. The global registers share a common bus and a switching network which together allow fast data transmission from point to point. The primary memory should support a high processor bandwidth. The control unit is responsible for the resolution of register and functional unit conflicts and scheduling of their operations. Functional units work independently in asynchronous mode. With equal probability of using all the functional units, the maximum speed up ratio of K (K the number of functional units) can be achieved. However, it is very unlikely that all the K functional units will be used equally well. The limitation of multifunctional systems is due to the complexity of the control unit which must detect the parallel execution of several operation during the execution time. For programs with a lot of sequential data dependence, the efficiency is much diminished.

3.1 Parallel Systems

Parallel systems are the natural extension of parallel ALU systems. Point D in Feng's concurrent space (Figure 1) represents a parallel system. In this organization concurrency is exploited through a collection of identical and independent processing elements controlled by the same control unit. Thus, at any given moment all of the processors perform the same operation on different pieces of data. In general, this approach is very scalable and offers a good degree of fault tolerance. In this study, we distinguish three groups of parallel systems: Ensemble processors, Array processors and Associative processors.

3.1.1 Ensemble Processors

Ensemble system is an extension of a conventional uni-processor system. It is a collection of N processing elements (a processing element consists of an ALU, a set of local registers and very limited local control capability) and N memory modules, under the control of a single control unit. Such a simple organization does not provide any direct communication paths among processing elements. Moreover, it does not allow flexible interconnections among

processing elements and memory modules. Such communications are done through the control unit.

As one can conclude, the organization is capable of executing up to *N* identical and independent jobs simultaneously. However, due to the lack of direct inter-processor communications, this organization has very limited applications.

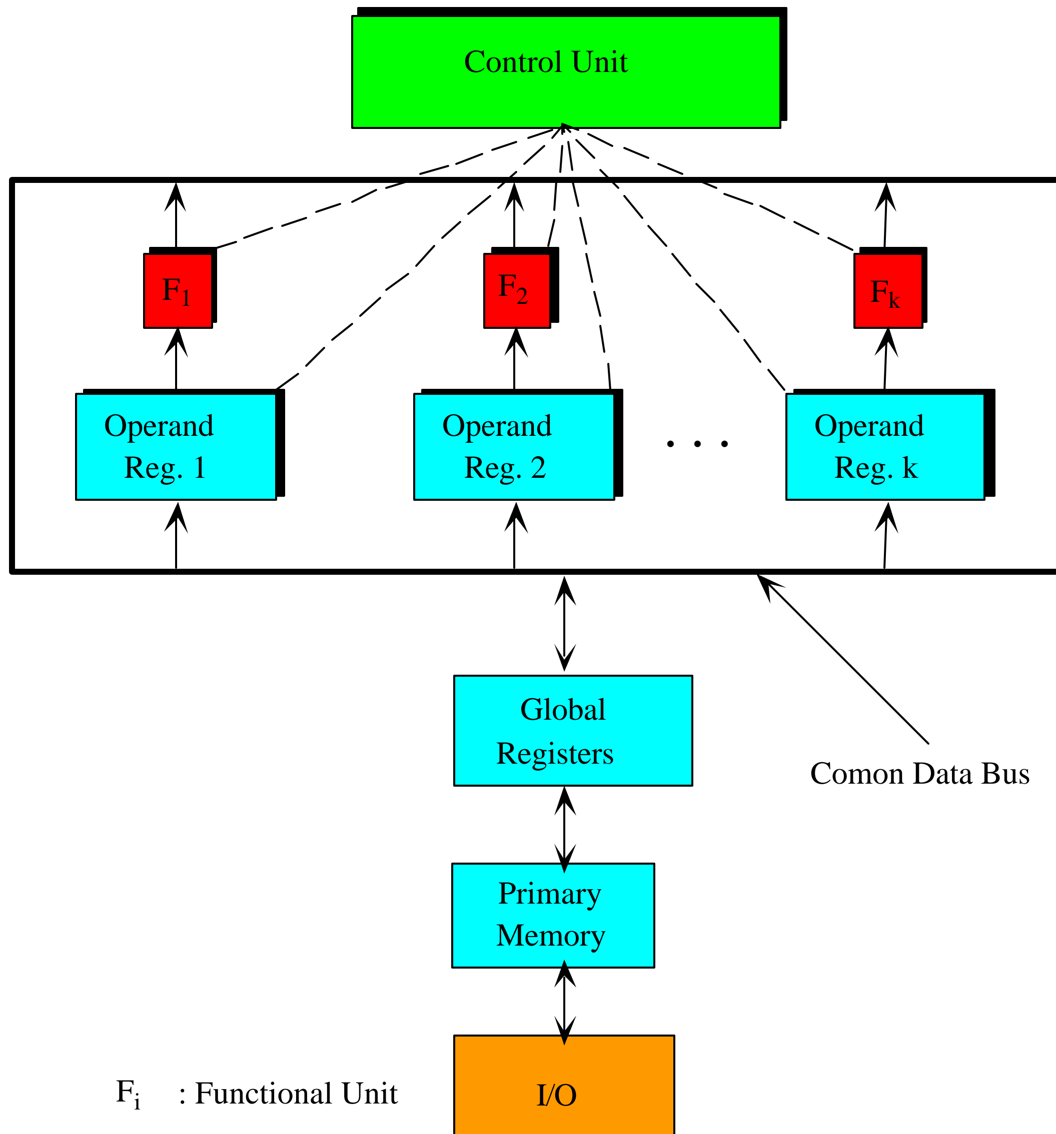


Figure 3. Block Diagram of Multifunctional Processor System.

3.1.2 Array Processors

The schematic diagram of an array processor is shown in Figure 4. The system is composed of *N* identical processing elements (PEs) under the control of a single control unit and a number of memory elements. The processing elements and memory elements communicate with each other through an interconnection network. This network usually provides a uniform

interconnection among processing elements on one hand and processing elements and memory modules on the other hand.

Two general organizations of array processors can be found in the literature. In type 1 organization (Figure 4), any processing element can access any memory elements through a complex and expensive interconnection network. Thus, all the benefits of a shared memory system are present. On the other hand, in a type 2 organization (Figure 5), each processing element is given a dedicated memory that only it can access directly. Therefore, this organization requires a communication mechanism among the processing elements to provide data communication between them. BSP and ILLIAC IV are two classical examples of Type 1 and Type 2 array processors, respectively. ILLIAC IV uses a mesh-structured network, while BSP uses a cross-bar network in order to establish the communication among processing elements and between the processing elements and memory elements. Array processors can also be classified into **coarse grained** or **fine grained** based on the complexity of the processing elements. In the coarse grained system the PEs are multi-bit processors. These processors often have floating-point arithmetic capabilities. However, due to the relative high cost of the PEs, fewer PEs are found in these types of systems. Fine grained array processors, on the other hand, use PEs that are only single-bit processors. As a result, any complex operations must be broken down into a series of single bit operations. However, since fine grained PEs are much less expensive and simpler in structure than coarse grained PEs, many more PEs are connected together in a fine grained system. ILLIAC IV and Connection Machine are classical examples of coarse grained and fine grained array processors, respectively.

In array processors the control unit is a computer with its own high speed registers, local memory and arithmetic unit. As in conventional machines, the instructions are stored in the main memory together with data. The main memory in this system is the collective memory in N processors. Hence, the instructions are fetched from the processors' memory into an instruction buffer in the control unit. If an instruction is either a control or scalar type instruction, it is executed entirely within the control unit. However, in case of a vector instruction it is performed in the processing array. The primary function of the control processor is to examine each instruction to be executed and to determine where the execution should take place. Array processors can further be classified into two classes according to the capability of the processing elements in handling the data manipulation operations. In the first case the processing elements are multi-bit processors, i.e., they operate on a word size that is several bits wide. These processors often have floating point arithmetic capabilities. However, since the cost of one of these processing elements is relatively expensive, fewer processing elements are found in these organizations. BSP and ILLIAC IV are examples of multi-bit array processors. On the other hand, the second class of array processors uses single-bit processing elements. As a result, any complex operations must be broken down into a series of single bit operations; hence the majority of operations take longer to be executed in this group of array processors than in their multi-bit counterparts. However, since single-bit processing elements are much less expensive and simpler in structure than multi-bit processing elements, a far greater number of processing elements can be connected together in a single-bit organization. This increases the processing power of the array processor. In addition, while multi-bit array processors can most efficiently process data elements of the same size, single-bit array

processors can easily operate on variable length data elements. The DAP, MPP, and Connection Machine are examples of single-bit array processors.

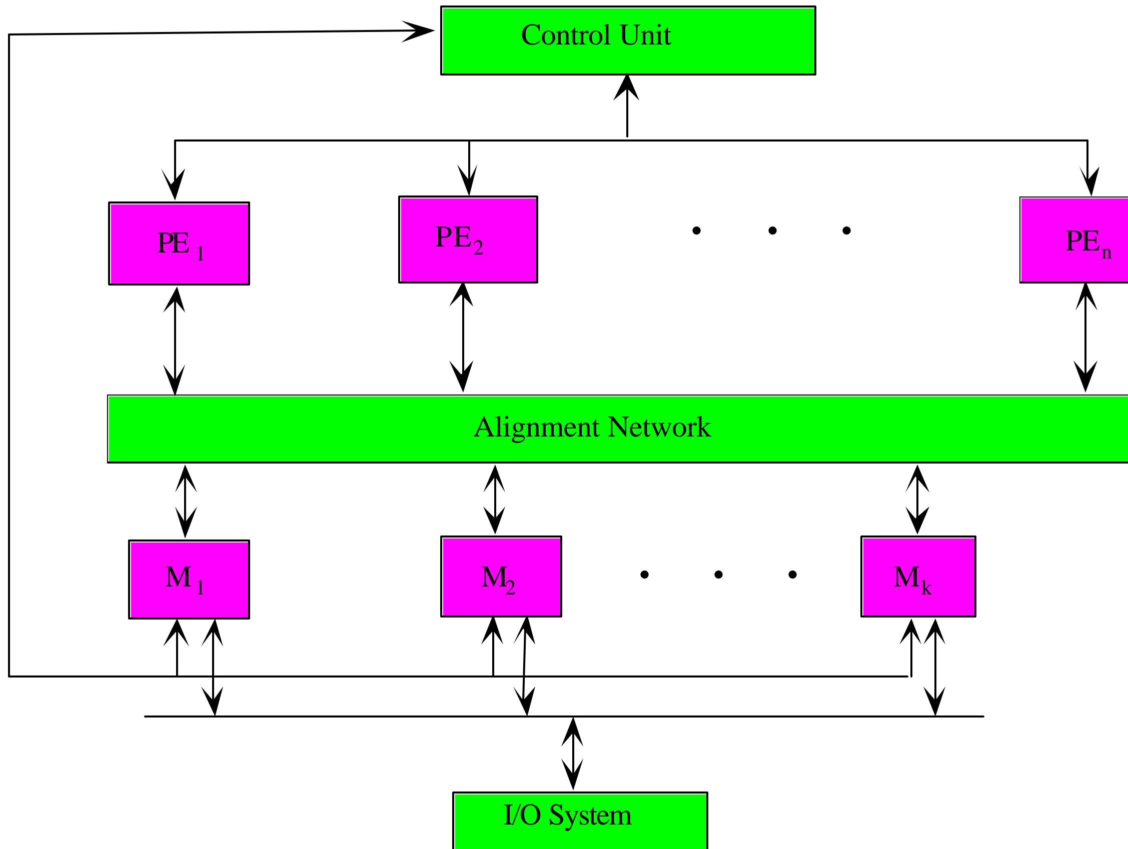


Figure 4. An array of processor (Type 1).

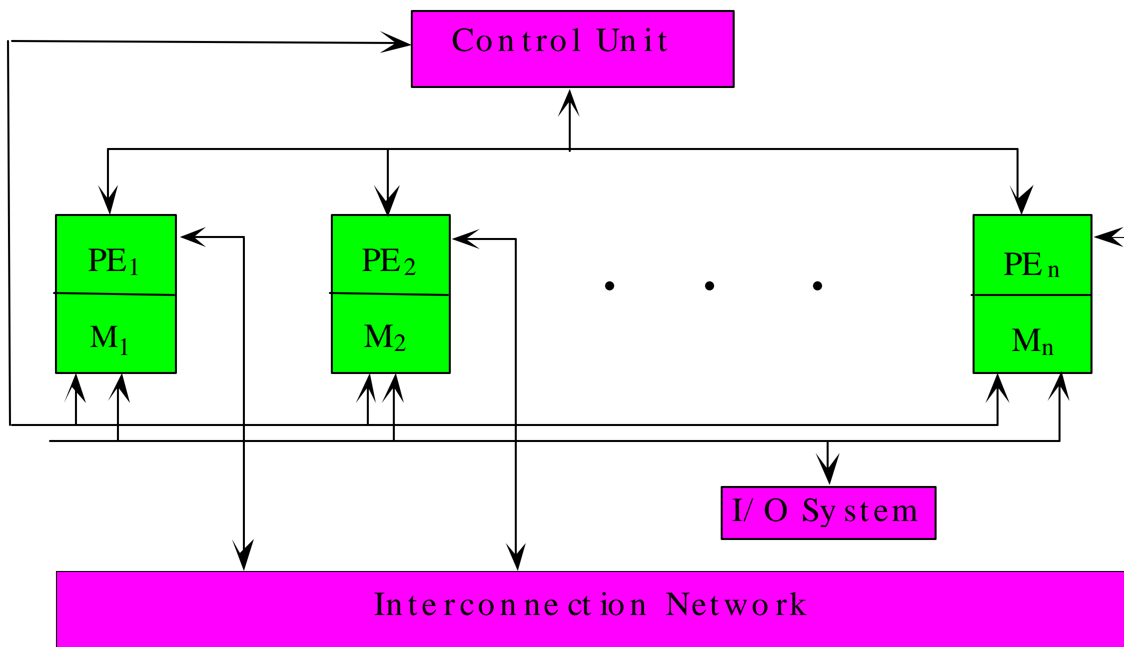


Figure 5. An array of processor (Type 2).

An array processor is a synchronous parallel computer. Processing elements are synchronized to perform in parallel the same function at the same time. The problem of data structuring and detecting parallelism in a program is a major bottleneck, although the design of the control unit is simple and is most like the one in sequential systems. In array processors an operation such as:

x(i) = A(i)*B(i) i = 1, 2, ..., N

could be executed in parallel, if the elements of *A* and *B* arrays are distributed properly among the processors, e.g., the ith processor is assigned the task of computing x(i). However, if we have to compute:

$$Y = \sum_{i=1}^{N} A(i) * B(i)$$

the product terms are generated in parallel as discussed before. Additions will be performed in $\log_2 N$ iterations, assuming that the intermediate operands are properly aligned and only a subset of processors which handle these operands become active at successive iterations. Thus the speed up ratio becomes at the expense of a poor resource utilization.

$$S = \frac{2N-1}{1+\log_2 N} \approx \frac{N}{\log_2 N} \qquad (13)$$

3.1.3 Associative Processor

Associative memories have been generally defined as a collection or assemblage of data storage elements which are accessed in parallel on the basis of data content rather than by specific address or location. As a result, each associative cell should have hardware capability to store and search its contents against the data which is broadcast by the control unit. With such a definition in mind one could conclude that, while read and write are the basic operations in the conventional random access memory (RAM), search is the basic operation for associative processing. The typical components of an associative memory are depicted in Figure 6. The Memory array provides storage to store the data. The comparand register holds the data to be compared against the contents of the memory array. However, by proper setting of the bit pattern in the mask register one can mask off portions of the data words from comparison and other operations. A response bit indicates the success or failure of a search against the content of the corresponding associative word. Finally, the multiple match resolver is used to narrow the result of a search to a specific word in case of multiple responses (e.g. matches).

An associative processor is then defined as an associative memory capable of performing arithmetic and logic operations. Usually, in such an organization, arithmetic and logic

operations are performed one bit at a time. An associative computer then is defined as a system that uses an associative memory or processing as an essential component for storage or processing, respectively. An obvious advantage of associative processing can be found in its application in non-numeric processing, radar signal tracking and processing, image processing, and simple arithmetic and logic operations on large sets of data. The main motivation for the study of the associative systems centers around its capability in:

i) Reducing the existing semantic gap and bottleneck in the conventional systems, and
ii) Increasing the performance due to the parallel operations at the storage level and elimination of address computation.

The first electronic associative memory was introduced by Slade and McMahon who described the design of a cryogenic memory system. Since then associative memories have been implemented using techniques such as tunnel diodes, evaporated organic diode arrays, magnetic cores, plated wires, semiconductors, bubble memory, integrated circuits, and recently optical technology. Moreover, literature has addressed several modifications to the basic associative operations (e.g. Hybrid associative memory, read only associative memory). However, up to the last decade there was no wide spread general application of associative memories. This was due to the hardware complexity and cost of associative cells in comparison with RAM cells, conservatism, and lack of suitable associative algorithms. As a result, the statement such as "the superiority of the content addressable memory is implied not proven" was a true statement. However, since the mid 1970s there is growing evidence that the above claim of an associative memory's superiority may be justified. This is due to the advances in technology and its effect on cost and size of the hardware components, and the strong applications of associative processing in non-numeric operations, image processing and pattern recognition (Table 3).

Associative memories have been classified into four categories namely fully-parallel, bit-serial, word-serial and block-oriented. This classification is in accordance with the basic unit of data to which the search operation is applied, and reflects a compromise between speed and cost.

a) Fully-Parallel: In a fully parallel organization, each basic unit of information (e.g., bit) has its own search circuitry. Therefore, the associative operation can be performed along two dimensions simultaneously. Such a direction implies larger cell size and more expensive modules in comparison with a bit in the random access memory. Point *D* in Figure 1 represents a fully parallel associative memory. In practice, fully parallel associative memories have been realized as a two or one dimensional memory arrays. In the two dimensional organization (word organized) memory is composed of fixed length entities called words. In a one dimensional organization (distributed logic), memory is arranged as a string of search character cells where each cell communicates with its neighbors and the control unit. Naturally fixed length record size is an obvious shortcoming of a word organized model. This will limit/complicate the implementation of the variable length word applications. However, one should remember that associative operations in a word organized memory are handled easier than the ones in a distributed logic organization.

b) Bit-Serial: This organization represents point *C* in Feng's concurrent space (Figure 1). Memory could be organized as a collection of circular shift registers in which search capability is

associated with a designated bit within each word (e.g. bit-slice). To achieve efficiency at a reasonable cost, a variation of this organization (i.e. byte serial associative memory) has also been proposed in the literature. In a byte serial model, byte search capability is associated with each associative word.

c) Word-Serial: In this class, search capability is associated with a word. This will represent point *B* in Figure 1. However, one should recognize the difference between this organization and word-parallel ALU systems, based on the fact that in word-serial organization operations are performed in associative fashion. This organization represents a hardware realization of a simple program loop in linear search.

d) Block-Oriented System: In this class, associative capability is provided at the mass storage level (e.g. secondary storage). This concept is an extension of fixed head rotating secondary storage technology. However, the fixed read/write heads are extended as a small processor (i.e. logic per track). As the data passes under the read/write heads, it will be investigated and marked for the later accesses. During the 1970s, the concept of logic per track, originally proposed by Slotnick, was used as a guideline in the design of many database machines.

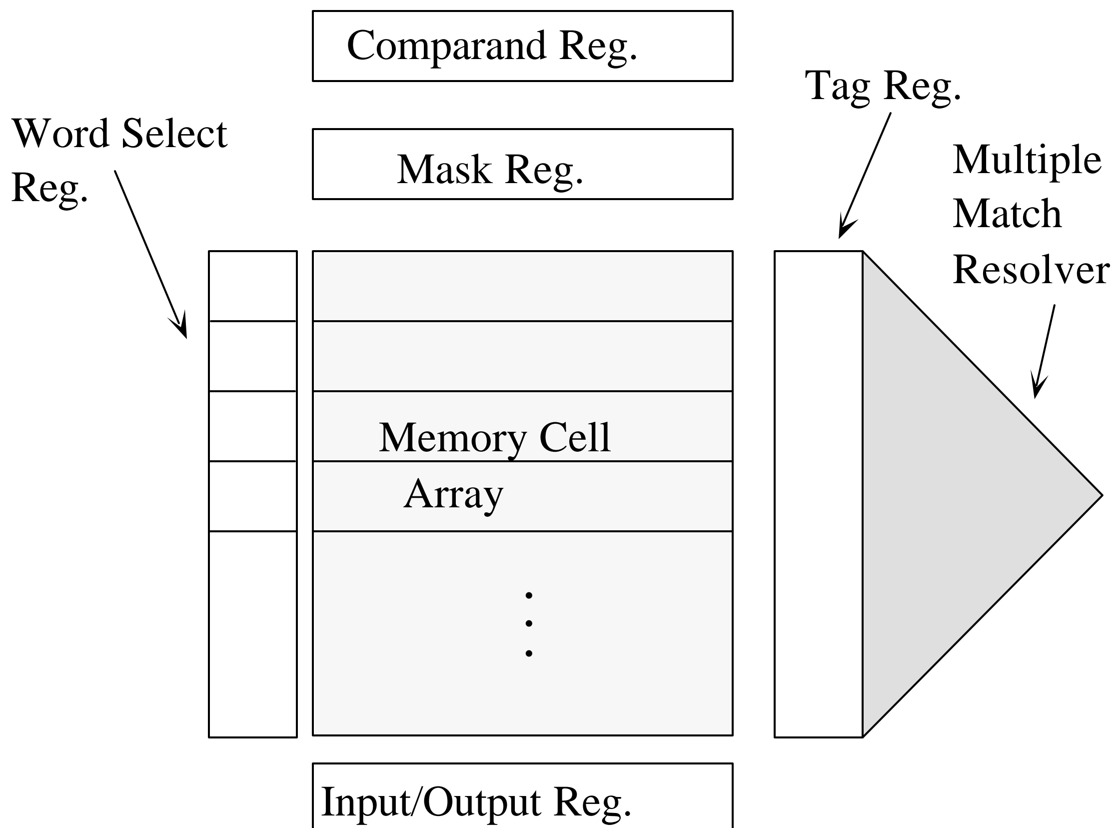


Figure 6. A Word Parallel Associative Memory.

Table 3. Developments in the Design of Associative Chip.

Year	Capacity (bit)	Cell Size l^2	Application
1985	4-K	2652	Artificial Intelligence
1985	8-K	1080	Dataflow Processing
1986	20-K	1438	Artificial Intelligence
1987	16-K	5000	Database Processing
1988	6-K	14092	Database Processing
1988	2-K	3040	PROLOG-based Computers
1988	9-K	1656	Parallel Processing
1989	8-K	9544	Database Processing

e) **Application of Associative memory:** The use of CAMs to improve the performance of memory management is already well established. Associative memories can be used to quickly execute the table entry look-up and modification operations used in memory management systems. For this reason, CAMs are often used as translation look- aside buffers in virtual memory systems and as tag directories in fully-associative cache organizations. For both these applications the CAM needs to perform equality searches on its contents.

Associative memories have often been used in the architecture of database machines (unit 4). The parallel search capabilities of CAMs make these devices ideally suited for the database environments. Typically, a CAM used for database operations should have at least maskable equality-search, maskable write, and multiple write capabilities. However, many database applications often perform θ-searches (where θ is the element of the set $\{<, >, =, \neq, \leq, \geq\}$). As a result, making an associative memory which can implement a θ-search directly in hardware is very desirable. The concept of content addressable processing for handling character matching operations was proposed by Lee and Paull. This bit parallel word serial organization was composed of an array of identical cells, each acting as a small finite state machine capable of communication with its left and right neighboring cells. The system was proposed for text retrieval operations, but because of the hardware cost it was not possible to be used in practical applications. Later on, a number of variations to this organization were proposed in the literature. In the early 1970s some new hardware for handling databases using associative processors was designed. Comparisons between an associative processor based architecture and a similar von-Neumann architecture have shown its superiority with respect to retrieval, update and storage operations. DeFiore and Berra have shown that in a database

environment, associative processors need to use three to fifteen times less storage compared to a database system using inverted list organization and have a response time faster by an order of magnitude.

Associative memories are also being used in the design of the prolog machines for efficient handling of backtracking and unification operations. It has been shown that a CAM with a maskable equality search, maskable write, and the garbage collection abilities can reduce the backtracking time to a small, constant value regardless of the number of bindings, also a CAM can be used to speed performance on unification through clause filtering. Finally, we have recently witnessed a surge of interest in the application of associative memory and associative processing in the area of Computer Vision.

Despite the great advantages of associative operations in some applications, there are very few associative memories currently available in the market either as general purpose chips or as components in standard cell libraries for VLSI design. One reason for this is that there exists a perceived belief that the cost of an associative memory is much too large to be practical. At first glance, this seems to be a valid concern. After all, compared to a conventional RAM a fully parallel CAM suffers the additional cost of search circuitry at every bit position (a CAM also has more complex control circuitry to operate it, but in large CAMs this circuitry is overshadowed by the size of the storage elements and search circuitry). In some cases, though, this size and cost increase should be acceptable. For example, the equality-search bit cell contains only about twice as many transistors as a standard CMOS static RAM cell, a penalty more than offset by the increased functionality of the associative memory.

However, the perceived cost of a CAM might not be the only reason that CAM production has been discouraged. As discussed earlier, applications which use CAMs to speed up their execution often require CAMs with varying degrees of functionality. As a result, it would be nice to have a general CAM design which allows CAMs of different functionality to be built easily. However, most of the proposed fully parallel CAM designs are for special purpose CAMs with a very specific set of functions. While these CAM designs might be suitable for an application which requires those specific functions, the special purpose nature of the design may make it difficult to incorporate other types of functions into the CAM.

To help simplify the development of different types of CAMs, we believe that a general CAM organization suitable for creating CAMs of various degrees of functionality is needed. Ideally, this organization should contain a high-level CAM architecture composed of a set of mostly-independent modules and a list of common features shared by every CAM regardless of its functionality. These common features are implemented by modules whose designs usually remain the same regardless of the type of CAM being developed. A set of special-purpose features, different for each CAM implementation, determines the exact functionality of the CAM. These features are implemented by special-purpose modules in the CAM. Finally, one needs to develop proper tools for automatic design and fabrication of modular associative chips with various functionalities.

It is worthwhile to mention that, besides the above special purpose designs based on associative processors, there are some associative processors which are capable of performing general purpose operations. STARAN is an example of such a system; it is composed of an associative array processor (one to 32 modular associative processor) with an interface (custom interface unit) to the users. It also has conventionally addressed control memory for program storage and data buffering. Each associative processor is a matrix of 256 words by 256 bit,

with parallel access up to 256 bits at a time in the word, bit, or mixed direction. Control signals generated by the control logic unit are fed to the processing elements in parallel, and all processing elements execute the instruction simultaneously. The STARAN symbolic assembly language APPLE provides a flexible and convenient assembler for programming without complex and costly indexing, nested loops, and data manipulation constructions required in conventional systems. Instruction execution time is dependent upon the number of bits in the operations involved in the instruction.

3.2 Multiprocessor Systems

The attribute that characterizes a multiprocessing system is the sharing of a global memory by several independent processors making up the system. Two arguments justify such an approach. The first is that by assigning a different job to each processor, the total throughput of the system is increased. This is due to the ability to overlap both computation intensive and I/O intensive jobs in the overall system. The second argument for multiprocessors is that there exists a large class of problems where the problem can be split up into a number of independent tasks. In a multiprocessor system, each task can be simultaneously run on a different processor. This reduces the execution time of the problem and hence increases the system's throughput. Figure 7 depicts the general organization of a multiprocessor system.

In general, a multiprocessor system can be characterized by the following features:

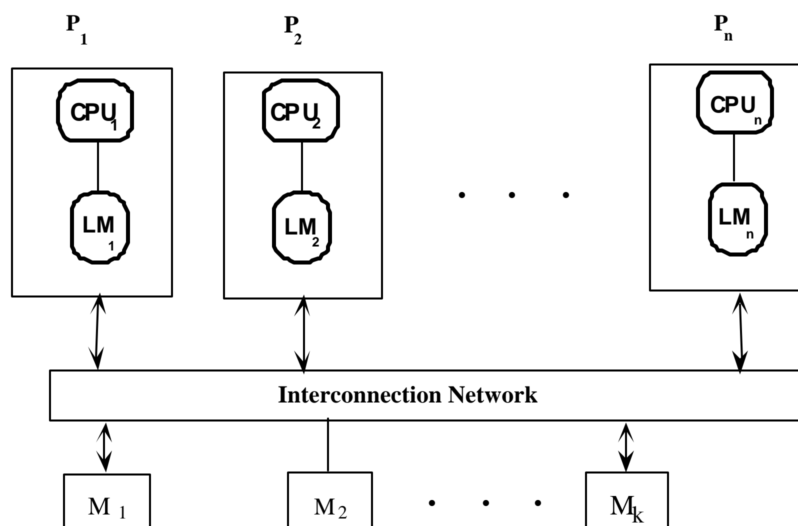
- The system contains two or more processors, each with its own control unit. These processors can be homogeneous or non-homogeneous, but most currently developed systems use homogeneous processors. Since it is generally required that the processors can perform general purpose operations, this rules out systems with a central processor and highly specialized I/O processors as multiprocessors. Note that the execution performance of a single processor can vary dramatically between different systems. For example, the Cm* multiprocessor system uses processors with the computing power of a PDP-11, while each processor in the S-1 multiprocessor is about as powerful as a CRAY-1 computer.

- The processing elements share a main memory that usually consists of several independently accessible modules. This memory holds common data needed by the various processors in the system. The shared memory can be organized in one of two ways. In the first organization, all of the shared memory modules are separated from the independent processors by an interconnection network or a multiport interface. Hence the access time of the shared memory (assuming no conflicts) is independent of the module being accessed. This type of system is known as a tightly coupled system. In the second organization each processor has a local-public (as opposed to local-private) memory; the shared memory of the multiprocessor is the aggregate of all these memory modules. Each processor can directly access its memory module, but all other accesses to non-local memory modules must be made through an interconnection network. Note that in this organization, called a loosely coupled system, the access time to the shared memory depends upon whether the desired address is local to the processor. The CRAY X-MP and the HEP are both examples of tightly coupled multiprocessor systems, while the Cm* is an example of

a loosely coupled system. In both tightly and loosely coupled systems the access time to shared memory may be increased due to the memory contention — i.e. more than one processor accessing the same module at the same time.

- Each processor might also have a local-private memory in addition to the shared memory. A programmer may or may not be able to directly reference this memory. In the first case, the memory can be used to store local variables and global values that would otherwise be frequently referenced from the shared memory. In the second case, the memory is a hardware controlled cache which holds recently referenced data in the expectation that it will be referenced again soon. Since private memories can potentially increase the overall performance of a system, recent multiprocessor designs are equipped with local-private memory modules. In such an environment the issue of coherence problem with private memories has attracted many research efforts.

- Besides a common memory, the processors usually (but not always) share other resources such as I/O channels and devices. This amortizes the costs of these resources over the several processors in the system. This also allows the rest of the system to continue using I/O devices even though a processor may fail. However, it is possible for processors to have private I/O devices in the multiprocessor system.



P : Processor
CPU : Central Processing Unit
LM : Local Memory
M : Memory Module

Figure 7. Multiprocessor System.

- The whole multiprocessor is under the control of a single integrated operating system. This operating system provides the means of interaction among different modules in the system. To help do this, it uses the unique hardware features of the system such

as an interprocessor communication mechanism, if any. Note that some operating systems allow the processors to work on several different problems at the same time (multiprogramming) while others require that all the processors be dedicated to a single job.

Multiprocessors have several advantages over conventional computer systems. First, as mentioned above, problems can often be solved faster if they are broken up into several concurrently executing tasks. Second, multiprocessor systems are more reliable since failure in any one of the redundant components can be tolerated by reconfiguring the system. Finally, multiprocessor systems are cost effective due to resource sharing among the processors in the system.

It is clear to see that multiprocessor systems naturally evolved from earlier systems. Like array processors, multiprocessor systems achieve concurrency through hardware replication (i.e. redundancy). Thus it is reasonable to say that the multiprocessor organization is a logical extension of array processor organization. The degree of freedom associated with the processor is much higher than in an array processor. That is, the processors are more independent with respect to each other. However, this independence of the processors and the sharing of resources among the processors, both desirable features, do not come without a price. Instead, these features increase the complexity of a dynamic communication system between the processors and the shared resources (e.g., memory) and between themselves. In addition, the operating system and other software supports must be more complex than their array processor counterparts in order to efficiently map application programs onto the hardware features.

If a multiprocessor has P processors, its throughput is certainly less than P times the throughput of a single processor. This is due to several factors. For example, I/O operations usually take longer in multiprocessor systems due to the overhead caused by software which resolves resource conflicts. The performance of individual jobs can be degraded by the delays caused by interprocessor communication, the need for processor synchronization, and memory and other resource conflicts. However, the number of resource conflicts can be reduced by scheduling a balanced mix of I/O intensive and computation intensive tasks. In multiprocessor systems with 2 and 4 processors, typical values of throughput are 1.5 and 2.5, respectively.

3.3 Pipeline Systems

The term pipelining refers to the design technique that introduces concurrency into a computer system by taking some basic function to be involved repeatedly in the system and partitioning it into several sub-functions with following properties:

- Evaluation of the basic function is equivalent to some sequential evaluation of the sub-functions.

- Other than the exchange of inputs and outputs, there are no interrelationships between sub-functions.

- Hardware may be developed to execute each sub-function.

- The time required for these hardware units to perform their individual evaluations is usually approximately equal.

Therefore, in a pipeline system a process is decomposed into a series of sequential sub-processes. Each sub-process is executed on a dedicated module called a stage or station. In addition, since the logic that actually performs the sub-processes at each stage is without memory, the presentation of data to each stage usually demands some kind of storage (buffer) to be included at either the beginning or end of each stage. This will help to synchronize the overall flow of data through out the pipe. Thus, within a pipeline several partial operations can be in progress concurrently, which will result in an increase in throughput. The concept of pipelining has been implemented in system such as: Amdahl 470 V/8, CDC 7600, CDC STAR-100, Cray , Fijitsu VP-200, Hitachi S-810, IBM 360-91, NEC SX-2, and TI-ASC.

Suppose we want to compute the elements x(i) defined as:

x(i) = A(i)*B(i) i = 1, 2, ..., N.

Assuming that the multiplier unit is a pipeline of 5 stages, then the overall execution time will be $[(N-1)+5]\Delta t$ (Δt is the delay time due to operation in a stage) provided that a constant flow of data is always available to the pipeline and the system can store the x(i)s as fast as they are generated. Now, suppose one has to calculate

$$Y = \sum_{i=1}^{n} A(i) * B(i)$$

using the same pipe for addition. The formation of products will take (N+4) stage delays. Then the pipeline is drained out and set for addition operations. Due to the data dependence, additions are performed in several passes. After the first pass, the pipeline yield $\lceil N/2 \rceil$ results, in the second pass it yields approximately $\lceil N/4 \rceil$ results, ... etc. Hence the total execution time would be $5+(\lceil N/2 \rceil - 1) + 5 + (\lceil N/4 \rceil - 1) + ... +5 + (1-1) = 4 \log_2 N + \lceil N/2 \rceil + \lceil N/4 \rceil + ... = 4 \log_2 N+N$ stage delays. Hence, the total execution time is $=2N+4\log_2 N+4$ stage delays. A serial process would have taken 5(2N-1)=10N-5 stage delays. As a result the speed up ratio is equal to

$$S = \frac{5(2n-1)}{2n+4 \log_2 n+4} \approx 5 \quad \text{for large n} \tag{14}$$

Pipelines can be classified according to their capabilities. A **unifunction** pipeline is the one that is capable of only one kind of operation. On the other hand, a **multifunction pipeline** is the one that is capable of handling several different kinds of functional evaluation. A multifunctional pipeline can be further grouped into **statically configured** and **dynamically configured** pipeline. This classification is based on the frequency of changes in the functions they perform. A concept known as the **hazard** is a major concern in a pipeline architecture. A hazard prevents the pipeline from accepting data at the maximum rate that the staging clock might support. Hazards are the result of structural and data dependencies. A structural hazard is one where two different pieces of data attempts to use the same stage at the same time (e.g., collisions). Data dependent hazards occur when a pass through a stage is a function of the data value. For statically configured pipelines, the designers could predict precisely when a structural hazard might occur and hence they can schedule the pipeline so that the collisions do not occur.

Data dependent hazards are clearly system and usage dependent and are not as amenable to analytical study as are structural hazards.

Application of pipelining as a technique to improve the performance, and hence to reduce the computation gap, can be traced in the evolution of the CDC-6600. In 1969 the CDC-7600, an upgraded version of the CDC-6600 was introduced. The major innovation in the CDC-7600 was that all but one of the functional units of the CDC-6600 (i.e. divide unit) were replaced with pipelined functional units. As a result, not only could all of the functional units operate concurrently, but most could also be operating on several pieces of data at the same time. Following in this trend, the CRAY-1 was introduced in 1976. Like the CDC-7600 the CRAY-1 had pipelined functional units. But it also had a more extensive register structure that better complements the pipelined functional unit design. In 1982 the CRAY X-MP was released. This supercomputer incorporated multiple independent CPUs (up to four by 1984) in its design to further increase the computer system's power. In 1985 Cray research introduced the CRAY-2. While much of its performance increase was due to technological factors, it was mainly a pipelined organization.

This trend in application of pipelining scheme produces two positive side effects. First, the design of the control unit was made easier because pipelined functional units can accept a new operand (or pair of operands) every clock cycle and thus are never considered "busy". Second, the pipelining scheme made the ALU ideal for solving problems that are vector in nature. It should be noted, however, that since the vector and scalar operations of this computers perform mainly large-scale arithmetic, some types of algorithms, such as sorting algorithms, can not use the vector capabilities very effectively.

3.3.1 Hazard and Collision

As discussed earlier, to utilize a pipeline effectively we have to provide stream of data to the pipe. Otherwise, hardware resources can not be overlapped and system throughput decreases. In a linear pipeline the delivery of the data to the pipe (e.g., Latency) can be easily synchronized with the time delay of the slowest stage. However, in a feedback pipe because of the internal conflicts, due to the collisions among different data set, such a simple synchronization scheme can not be employed. Though the scheduling algorithm for a generalized pipeline is NP complete, but under some strict practical restrictions one could develop an optimum solution for such a hard problem. The set of restrictive conditions well applicable to the practical operations are:

- The execution time of all stages is a multiple of some basic unit.
- Once an activation is started, its time pattern of stage utilization is fixed and defined

The first condition can be easily enforced and the second condition excludes the class of dynamic pipelines. The time-pattern of stage utilization can be defined by a two dimensional table called **reservation table**, in which rows represent stages of the pipeline and columns represent the time slots. The reservation table shows at each time instant which stage of pipeline in this table is in used by the computation. A reservation table represents exactly one pattern taken by one input data set. A mark in the entry (i,j) of the reservation table indicates that for that pipeline stage i is needed j time units after its initiation. Therefore, the **compute time** of the table is defined by the number of columns in the reservation table. Existence of several marks in one row represents the fact that the corresponding stage is utilized several times during one

initiation. Moreover, consecutive marks in a row indicate that the execution of the stage is a multiple of basic time unit.

Figure 8 depicts a pipeline and its reservation table. Note that the pipeline has seven stages and a compute time of 9. It should be noted that each operation with respect to a pipeline system has a unique reservation table.. However, a reservation table might represent several pipelines.

According to the definition of the reservation table, then one can make the following conclusions: A pipeline is statically configured if the same reservation table is used by all activations. A multifuntion pipeline has several reservation tables. Finally, in a dynamic pipeline the computation does not have a predetermined reservation table. From Figure 8 it can be concluded that two initiations which are four time units apart will collide at stage S_1. Moreover, no collision will occur if two initiations are two time units apart. Therefore, one has to design a control mechanism which allows the data to be delivered in a time pattern which prevents any possible collisions in the future with the previously initiated computations.

According to our discussion so far we can conclude that, two computations should not be initiated if they are l time units apart and l is the distance between two marks in the same row. Such a time distance is called a **forbidden latency**. From the definition of forbidden latency, then one can determine the **forbidden list (L)** which is the collection of all the forbidden latencies in a pipeline.

$$L = (l_1, l_2, \ldots l_k) \quad (15)$$

the **collision vector (C)** in then defined as a binary vector of length **K**

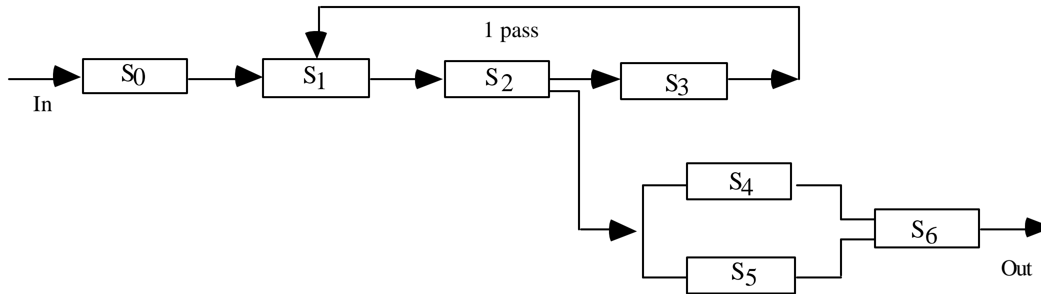
$$C = (C_K\ C_{K-1} \ldots C_2\ C_1) \text{ where } C_i = 1 \text{ iff } i \in L$$
$$C_i = 0 \text{ otherwise} \quad (16)$$

The bit pattern of a collision vector determines the forbidden latencies of the pipeline. Therefore, by investigating such a pattern one can initiate two computations without any collisions. For the pipeline of Figure 8 then,

$$L = (1,4) \text{ and } C = (1001)$$

As a result, a new collision free computation can be started only after 2 and 3 time units after the initiation of the first computation. A simple logical shift register can be used to control the initiation of a new computation. Upon the activation of the pipeline for the first computation, the collision vector is loaded into the shift register. The register is shifted right one position at a time. A collision free computation is allowed at time instant t+i if and only if a bit 0 is being shifted out of the register after i shifts from the time t.

After the initiation of the second computation then the collective collision vector of the pipeline for two initiations should be calculated. This collective collision vectors should then be investigated to determine the initiation of the next computation. The collective collision vector is calculated by oring the contents of the shift register with the initial collision vector. In general, after initiation of a new operation, the contents of the shift register should be modified to represent the overall status of the pipeline.



	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
S_0	X								
S_1		X				X			
S_2			X				X		
S_3				X	X				
S_4								X	
S_5								X	
S_6									X

Figure 8. An Example of a pipeline and its Reservation Table.

As far as Figure 8 is concerned, suppose the second computation is initiated after two-time units, the content of the shift register is:

$C_{t2} = C_{t1} \vee C = (0010) \vee (1001) = 1011.$

Now, the third computation can be initiated at time 3, this gives a collision vector of:

$(0001) \vee (1001) = 1011$

which is a transition to the initial state.

This gives rise to the idea of a **state graph**, representing all the collision free transitions in the pipeline. Cycles in the state diagram correspond to the possible cycles of collision free initiations of the computations. Every cycle has an **average latency** which is the average of the latencies of its constituent edges. An optimum cycle is the one that has the minimum average latency. After determination of the proper optimum cycle then one can design a simple finite state machine which can control the new collision free initiation of new computations.

4. PROGRAMMING CONTROL FLOW COMPUTERS

In the preceding sections we examined the architectures of some pipelined and parallel computers. Through advancement in technology and architectural innovations, each system is characterized by a very high peak processing rate for numerical computation. However, only a very limited subset of instructions available on these computers can execute at these peak rates. Since most non-trivial applications use a wide mix of instructions, the processing rates of these applications are often much less than the theoretical performance of the system.

From the programming point of view, we may state that a supercomputer is merely a computer with some added features allowing it, under certain conditions, to perform groups of operations at high speed. The key to designing programs that execute quickly on these machines is to learn how to utilize the resources of the supercomputer effectively for a given

application. This section surveys some techniques to improve the resource utilization of the aforementioned organizations.

4.1 Array Processor

An array processor can be classified as having a shared memory (i.e., BSP) or private (i.e., ILLIAC IV) memory organization. In a shared memory organization each processing element can access any memory module, while in a private memory architecture each processor has its own private memory module. In a shared memory organization (as long as array elements are distributed uniformly across a collection of prime memory modules) most common array operations can be performed without any memory contention. Since the representation of vector instructions in a shared memory array processor is similar to that in a piplined processor, much of the discussion that applies to pipelined organizations applies to this organization as well. For this reason, only private memory models are discussed in this section and the word "array processor" will refer to this model of computation.

In general, a private memory array processor has the following features:

- It contains a finite number of processors (natural parallelism) that execute the same operation in lockstep on different operands.
- Each processor in the array has a private memory which it can access very quickly.
- The processors in the array can exchange information via an interconnection network which has a particular topology. While only topological neighbors can directly communicate with each other, any two processors can communicate indirectly by passing values through a finite number of intermediate processors.
- At any given moment, all data passing through the interconnection network is traveling in the same topological direction.

Computationally, an array processor algorithm should try to perform the same operations on many independent sets of data. Often during the course of an algorithm, though, processors need the results calculated in other processors before another operation can be performed. This requires that data be shifted through the interconnection network. The communication complexity of an algorithm is defined as the number of data routings needed on a particular array processor to implement the data flow between processors that is specified by the algorithm. Since a data route operation often takes the same amount of time as other processors operations, the communication complexity of an algorithm cannot be ignored.

To reduce the communication complexity of an algorithm, communication should be limited to processors that are as topologically close together as possible. Ideally, processors should communicate only with direct topological neighbors. This implies that the optimal algorithm for an array processor is a strong function of the topology of the interconnection network. For example, an array processor with an interconnection network that can efficiently implement a tree-like communication network (such as hypercube interconnection network) can add N numbers in $0(\log_2 N)$ time using a tree structured cascade sum algorithm. However, an optimal summing algorithm for a mesh connected array processor takes 0(sqrt(N)) time and has a substantially different data flow than a tree structured cascade sum algorithm.

In many cases it is not possible to distribute data in the processor memories such that all processors need to communicate with their direct topological neighbors. For these situations, it may be beneficial to break up the processors into several clusters where the processors in each cluster are topologically close together. Data is distributed among all the processors such that

the majority of communication occurs between the processor in a cluster and not between different clusters. By localizing most of the communication to inside the clusters, the overall communication cost of an algorithm might be reduced.

In general, it is not wise to have an algorithm where any processor can be connected to any other processor. Such a connection scheme requires each piece of data that is sent through the interconnection network to have a processor destination address associated with it. The routing algorithm used in the array processor would have to examine the destination address of that piece of data and determine where to route it so that it eventually arrives at its destination. For most array processors, the overhead for such a routing algorithm would be self-defeating. However, on the Connection Machine this routing algorithm is performed in hardware concurrently with the instruction execution of the processors. This makes the implementation of any logical connection structure much more feasible than in most other array processors, which do not have such routing algorithms implemented in hardware.

4.2 Multiprocessor Systems

There are three general classes of parallel algorithms that can be applied towards multiprocessor systems. They are synchronized algorithms, asynchronous algorithms, and semi-synchronized algorithms.

A synchronized algorithm is conceptually the easiest of the three classes of algorithms to understand. In a synchronized algorithm, a processor must wait for a synchronization signal from another processor before it can execute a certain portion of its program. Because process synchronization is a costly operation, synchronization calls should be avoided as much as possible. This introduces the concept of granularity to synchronized algorithms. Granularity can be defined as the time spent executing concurrent code between processor synchronizations. To reduce the total percentage of time spent synchronizing the processors, the grain size of an algorithm should be made as large as possible. The amount of time that characterizes a large grain application varies between different computers. For example, a granularity on the order of milliseconds is considered large for the CRAY X-MP. Sometimes, however, it is necessary to execute small grain algorithms on a multiprocessor. In such cases, the direct use of hardware instead of operating systems calls for synchronizations can significantly reduce the execution times of these algorithms. The advantage of synchronized algorithms is that they are relatively easy to design and analyze.

Synchronized algorithms can be further subdivided into two types of algorithms: partitioned and pipelined algorithms. In a partitioned algorithm, a task is divided into several subtasks, each hopefully of the same size. Each processor is assigned one or more of the subtasks to execute, and all of the processors begin executing their subtasks at the same time. Synchronization points are inserted inside a subtask only when interactions between subtasks are needed. Whenever a processor finishes executing its subtask(s), it synchronizes itself with the remaining processors. After the synchronization, the results which were generated by the individual subtasks can then be combined to solve the original problem.

A pipelined algorithm, like its hardware counterpart, consists of several independent stages (subtasks) where each stage accepts input data from the preceding stage in the pipe and sends its output to the next stage of the pipeline. The input to the algorithm is given to the first stage of the pipeline, and the output of the last stage is the output of the algorithm. Each stage in the pipeline is assigned to a processor, and all the processors have to be synchronized before

data is passed between the stages. Pipelined algorithms are sometimes called macro-pipelined algorithms to distinguish them from hardware pipelining.

The second general class of multiprocessor algorithms is the asynchronous algorithm. In this type of algorithm, several processors are cooperating to solve a particular problem. However, communication between processes is accomplished by accessing global data and shared variables instead of processor synchronization. Each process works in stages. At the beginning of a stage, the process reads data from the global variables and determines whether the desired goal of the algorithm has been reached. If not, it calculates some new values based on the current values found in the global variables. These newly calculated values are usually stored into the global variables, but even this action may depend on the current values of other global variables. Since updating global variables is almost always performed in critical sections, processor idling occurs only when two or more processes try to update the same global variable at the same time. This updating takes a small amount of time, so even an idled processor would not be stopped for long. After updating the global variables, the process then repeats the stage. Note that the computations performed in a stage are designed to bring the whole system closer to the solution of the problem.

Asynchronous algorithms have several advantages over synchronous algorithms. First, the large synchronization overhead found in synchronous algorithms is non-existent in asynchronous algorithms. In addition, asynchronous algorithms are more reliable than their synchronous counterparts since it is guaranteed that at least one non-blocked processor will be working toward the solution of the problem. Another benefit of asynchronous algorithms is that they can take advantage of fluctuations in processor speeds due to factors such as memory conflicts, etc. Asynchronous algorithms do have some tradeoffs, though. If processor speeds do not fluctuate much in the system, processors which perform the same computations in their stages may end up redundantly performing the same operations on the global data. The data in the global variables also become inconsistent due to several asynchronous processes updating them at the same time. This makes it hard to analyze how fast an asynchronous algorithm converges in on a solution. Despite these disadvantages, asynchronous algorithms can be competitive with synchronous algorithms, especially if the system contains many processors and the fluctuations in speed of these processors are large.

The third class of multiprocessor algorithms is the semi-synchronized algorithms. This type of algorithm may be thought of as an asynchronous algorithm with an additional feature: if one processor gets "too far in front of" or "too close to" the calculations of another processor, it is blocked (synchronized) until the blocking condition no longer applies. Under most circumstances this blocking rarely occurs; thus the algorithm has the asynchronous algorithm's advantage of a low synchronization overhead. The exact blocking conditions for a semi-synchronized algorithm are determined by the special features of the problem to be solved. These blocking conditions are chosen to guarantee favorable rates of convergence for the algorithm, a feature not always possible in pure asynchronous algorithms. Thus semi-synchronous algorithms have the advantages of both synchronous and asynchronous algorithms. Unfortunately, favorable blocking conditions for a particular problem are not always easy to determine.

4.3 Pipeline Processors

Pipelined processors used pipelined functional units to perform operations on large vectors of data much quicker than could be done using scalar operations. Basically, a vector instruction can be thought of as a hardware implemented DO loop which takes one or two independent input vectors, performs an operation on them, and produces a single output vector. For example, the software loop

```
DO I = 1, 100
    A(I) = A(I) + B(I+1)
ENDDO
```

can be represented by a single vector instruction

```
A(1:100) = A(1:00) + B(2:101)
```

The key to designing high performance algorithms on pipelined processors is to utilize these vector operations as efficiently as possible. It is important to recognize that pipelined supercomputers are similar to conventional computers in many respects. Thus many of the rules that can help to speed up programs for conventional systems may be generalized to pipelined processors. One such rule is that values often used in a program should be kept in internal registers for quick access. For pipelined supercomputers with vector registers, an additional rule can be stated: perform as many operations on an input vector as possible before storing the result vector back in the main memory. Note, however, that these two rules are probably more suitable for a compiler to implement since most algorithms are not initially designed with register usage in mind.

Whenever a vector operation is executed, some overhead time is needed to initially fill the functional pipeline being used. Thus the total execution time T of a vector instruction operating on an N-element vector is

$$T = [s + (N-1)]\,\Delta t$$

where $s * \Delta t$ is the startup time of the pipe. Since the startup time is amortized over the N elements of the vector, the best average processing rates occur for the largest values of N. Depending upon the architecture of the computer, however, the largest value for N may have some limitations placed upon it. For this reason two classes of pipelined computers, memory-to-memory (i.e. STAR-100) and register-to-register (i.e. CRAY-1) pipelined computers, are discussed separately. Memory-to-memory pipelined supercomputers have virtually no limit placed upon N. Thus the size of the vectors used in an algorithm should be as large as possible to achieve the highest processing rates. There are several basic ways to increase the vector size in an algorithm. Since only inner loops can be vectorized, the first is to make the innermost loop of a multiple-nested loop as large as possible. This can sometimes be accomplished by changing the nesting order of the loop. For example, the code fragment

```
DO I = 1,100
    A(I,1:60) = 0
ENDDO
```

can be rewritten as

```
DO J = 1,60
    A(1:100,J) = 0
ENDDO
```

since both forms are functionally equivalent. Note that the vector size of the innermost loop is now 100 instead of 60. The second method is to convert multi-dimensional arrays into one-dimensional arrays, if possible. For example, the same code fragment above can be written as

```
A(1:6000) = 0
```

without changing its meaning (this assumes the elements of the two-dimensional array are stored contiguously in memory). The third technique is to rearrange data into unconventional forms so that several smaller vectors may be combined into a single larger vector. For example, successive over-relaxation (SOR) methods for solving finite differential equations appear to be non-vectorizable due to an abundance of short vectors. However, as mentioned in, clever reordering of the grid blocks used in SOR techniques can substantially increase the length of vectors in the algorithm.

In a register-to-register pipelined computer, the maximum vector size that can be processed is limited by the size of the vector registers, N_{VR}. The size of N_{VR} is relatively small (e.g. $N_{VR} = 64$ for the CRAY-1) and for most supercomputer applications vectors of size $N > N_{VR}$ need to be processed. In this case, the vector must be sectioned into several smaller vectors of length N_{VR} plus perhaps a final remainder vector with a length less than N_{VR}. The best average execution time occurs whenever N is a multiple of N_{VR}. In some instances, the nesting of loops can be rearranged to make the vector length of the innermost loop a multiple of N_{VR}. In many instances, though, one of the indices in a nested loop structure is not a multiple of N_{VR}. In these situations, the total startup time must be checked and the case with the smallest overall startup time should then be chosen. Finally, as mentioned for memory-to-memory computers, multidimensional array should be collapsed to one-dimensional arrays if possible.

All pipelined computers, whether memory-to-memory or register-to-register, use an interleaved main memory. By storing sequential elements of a vector in different memory banks, consecutive elements of a vector can be accessed quickly. Almost all pipelined computers allow accesses to vectors with any constant stride between the elements. However, some constant strides can cause the same memory bank to be accessed before its memory cycle is complete. This causes memory conflicts and slows the performance of the computer. For example, accessing every eighth element of a vector on a CRAY-1, which has 16 interleaved memory banks, can cause memory conflicts. Thus, the memory layout of data in an algorithm should be designed to minimize memory bank conflicts. This can sometimes be done by adding dummy rows and columns to an array. Random accesses to the main memory should also be avoided.

Another feature shared by all pipelined computers is the ability to use a bit vector, also called a control vector, to control the execution of a vector operation. A control vector for an N-element vector operation is an N-bit vector where the ith bit of the vector is set if the ith pair of elements in the input vectors are to be operated upon. Control vectors can be used to vectorize conditional statements within a DO loop. For example, the code

```
DO I = 1,64
        IF (A(I).GT.0)    A(I) = A(I) + B(I)
ENDDO
```

can be vectorized into

```
L(1:64) = A(1:64).GT.0
WHERE L(I) A(1:64) = A(1:64) + B(1:64)
```

Note, however, that whenever a control vector is used to control a vector operation, all elements of the input vector must be fetched even if an operation will not be performed upon them. Thus if a control vector is relatively sparse then much of the time spent during a controlled vector operation is wasted in fetching operands that are not needed. One way to

speed up the performance of a conditional vector operation is to collect all of the input elements that are going to be operated upon into a single, smaller vector, performing the desired operations upon this smaller vector, and then returning the output elements to their proper positions in the larger vector.

Finally, most DO loops found in algorithms contain more than one array assignment statement. There are many techniques used to transform multi-statement DO loops into a series of vector statements, most of which can be used by vectorizing compilers. It should be noted that among the aforementioned techniques to increase the performance of the algorithms for pipelined organizations, some can be used when designing an algorithm, while others can be used by an intelligent compiler to vectorize an already existing program.

4.4 Self Test Problems #2

1) Compare and Contrast:

   a) Single-bit and Multi-bit array processor organizations against each other.

   b) Global-memory and Dedicated-memory array processor organizations against each other.

2) True or false: In a database environment associative memory (associative processing) can improve the memory utilization (justify your answer).

3) The following pipeline organization is assumed:

   where $\delta_i = \Delta t$ $0 = i = 6$ and $\delta_i = 2\Delta t$ for $i = 3$. Determine the list of the forbidden latencies, the collision vector, and the state diagram.

4) For a fully parallel word organized associative memory, write an algorithm to find the greatest value stored in the memory. Assume numbers are all positive and each word contains one number.

5. ISSUES IN CONTROL FLOW CONCURRENT SYSTEMS

5.1 Parallelism vs. Pipelining

The theme of this section is two-fold: First, the general characteristics of parallel and pipeline systems are compared. Our comparison is intended to clarify the existing ambiguity between pipelining and parallelism. However, one can extend such a general discussion for other classes of concurrent systems. Second, the shortcomings of the conventional concurrent systems are addressed to motivate our discussion in the next section of this article.

Based on our earlier discussion, both parallelism and pipelining attempt to increase the performance of some functions by increasing the number of simultaneously operating hardware modules. For a conveniently designed module to perform some generic function, either technique can be used to derive a new design running up to N times faster. However, parallelism is achieved through the replication of the basic hardware unit N times, with all replicated units running simultaneously, while pipelining is the result of staging the hardware unit into a sequence of N subunits. The difference between pipelining and parallelism also shows up in memory organization and bandwidth, internal interconnection of modules, and control. For example, in a pipeline system, memory organization should support a constant and smooth flow of data to the pipeline. On the other hand, in a parallel system, accesses to the memory system are not smooth, and each processor could initiate an access to any memory module. This implies a complex and dynamic interconnection network between processing modules and memory modules. Such a network should provide simultaneous accesses to the memory modules where one access does not block other accesses to the memory modules. Since, a pipeline can not be broken up into an arbitrary number of stages, one can conclude that parallel systems are more expandable than pipeline systems. Reliability is another feature which separates these two techniques. In general, parallelism offers a more reliable system. This is due to the fact that in a parallel system, the task of any faulty module can be distributed among other replicated modules; this can not be done in a pipeline system. Architectural analysis of the so called supercomputers reveals a trend in the design of concurrent systems, which can be classified as parallel pipelined systems.

5.2 Shortcomings of the Conventional Concurrent Systems

The class of concurrent systems and its successor (e.g., supercomputers) have shown their effectiveness in many real time applications. These computers by their very nature are more complex than their predecessor architectures. This complexity is mainly due to the simultaneous competition/cooperation of several modules over common resources, which leads to more complexity and sophistication at the:

i) Control structure, in order to manage the flow of data and operations within the system's modules.

ii) Interconnection network, to allow simultaneous interactions among the system's modules.

While the growth in complexity could result in higher cost, lower resource utilization, and performance degradation, the major disadvantage of these systems is associated with two interrelated factors, namely **specialization** and the **semantic gap**.

In contrast to the conventional von-Neumann architectures, concurrent systems are specialized architectures. For example, while concurrent systems are superior in handling computation bound applications, they offer low performance in I/O bound applications such as database systems. In addition, these machines demand specific domain(s) to guarantee the performance improvement. Studies on ILLIAC-IV type architecture have shown that the allocation of data within the memory modules has drastic effect on the performance. Experiences on the CRAY computers have proven that vector operations for small vectors demonstrate performance degradation over scalar operations. These examples reflect the fact that conventional concurrent systems require specialized and sometimes different programming skills for efficient resource utilization. As a result, in a multi-functional unit system a mixed sequence of instructions increases the performance, while in a pipeline system a uniform sequence of instructions increases the performance. Therefore, we can conclude that conventional concurrent systems introduce a wider semantic gap than conventional systems in handling general purpose applications. Thus, they require an extensive software support to determine the inherent parallelism in an application program.

The performance improvement of concurrent systems is greatly dependent on the proper utilization of the hardware resources. However, it has been shown in practice that in many applications such a performance improvement has not been achieved. This problem is contributed to the: i) lack of suitable "parallel" algorithms for various applications, ii) lack of suitable "parallel" high level languages which enables the programmer to express the inherent parallelism explicitly in the problem being encoded, iii) lack of suitable compilation techniques to detect embedded "parallelism" in a sequential program, and iv) lack of suitable control algorithms to distribute hardware resources among concurrently running programs.

High level languages rooted in the 1950's have been developed as programming tools to increase the machine's independence and productivity. Naturally, these languages reflect the structure of the conventional uni-processor systems - i.e., the existence of a primitive set of arithmetic operations which are carried out sequentially on data stored in some form of memory device. However, for a concurrent system there is a need to express the "concurrency" in an algorithm for parallel execution. This goal can be achieved either through the definition of new parallel languages or the addition of parallel constructs in the definition of the conventional sequential high level languages. Since the introduction of concurrent systems, there has been a surge to design and develop parallel languages to facilitate the utilization and performance of these computers. The so called Parallel Fortran (P-FOR) proposed for PEPE architecture, TRANQUIL for ILLIAC IV, and APPLE for STARAN are among the pioneer efforts in this area.

Generation of parallelism from sequential constructs (e.g., vectorization) requires an extensive analysis of the sequential programs. This analysis must check that the ordering is in fact arbitrary and that there are no sequential dependencies in the process. This approach is a means to increase the adaptability of the parallel systems and to protect the previous investments of the users. Naturally, this direction requires the development of sophisticated compilers (e.g. vectoring compilers) to generate parallel machine instructions from sequences of operations without violating the program semantics. This means more sophisticated compilation techniques, more complex operating systems, and more advanced program development tools. The growth of the software overhead and its by-products in the concurrent systems is the source of our discussion in the upcoming units.

Glossary

Address accessible memory	A storage unit in which an storage element is accessed by means of its address.
Array Processors	A collection of synchronous processing elements under the control of a single control unit.
Associative Memory	A memory organization in which storage elements are accessed in parallel on the basis of data contents.
Associative Processor	An associative memory capable of performing arithmetic and logic operations.
CDC	Control Data Corporation.
CMOS	Complementary Metal Oxide Silicon.
Computation Gap	The difference between computational power demanded by application areas and the computation power of the computer systems.
Concurrency	Ability of the computer hardware to simultaneously execute many actions at any instant.
Content addressable Memory	See associative memory.
Control flow model of computation	A computation model in which the execution of an instruction activates the execution of the next instruction.
DAP	Distributed Array Processor.
Data flow computation	A computation model in which the availability of the data activates the execution of the next instruction(s).
Data dependent hazard	A pass through a pipeline stage is a function of the data value.
ENIAC	A pioneer computer organization (1948).
Greedy Cycle	For a pipeline organization, a greedy cycle is a cycle which allows the new activation at the earliest possible instant.

MIMD	Multiple Instruction stream, Multiple Data stream.
MISD	Multiple Instruction stream, Single Data stream.
Multiprocessor system	A collection of asynchronous processing units under the control of a shared operating system.
Parallelism	Ability to achieve concurrency via duplication/replication of hardware units.
RAM	Random Access Memory.
Pipelining	Ability to achieve concurrency via staging the hardware units.
Semantic gap	The difference between the features in the high level languages and the hardware features of the underlying architecture.
SIMD	Single Instruction stream, Multiple Data stream.
SISD	Single Instruction stream, Single Data stream.
Structural hazard	Is the one when two different pieces of data attempt to use the same pipeline stage at the same time.
TI	Texas Instrument
Q-search associative	An associative memory organization which allows memory elements to be accessed based on Θ {=,<,>,=,=,ð} relationship.
ULSI	Ultra Large Scale Integration.
VLSI	Very Large Scale Integration.
WSI	Wafer Scale Integration.