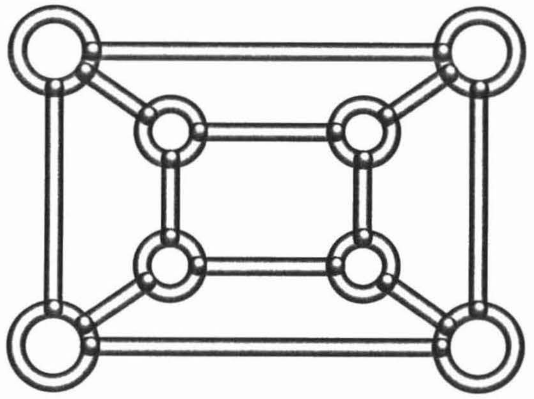
91-PA

PARALLEL ARCHITECTURES

Edited by N. Rishe, S. Navathe, and D. Tal



A postconference publication based upon the proceedings of PARBASE-90, held in Miami Beach, Florida, on March 6–9, 1990

Sponsored by Florida International University

IEEE Computer Society Press The Institute of Electrical and Electronics Engineers, Inc.

Parallel Architectures

Edited by N. Rishe, S. Navathe, and D. Tal

A postconference publication based on the proceedings of PARBASE-90, held in Miami Beach, Florida, March 6-9, 1990

Sponsored by
Florida International University
in cooperation with IEEE and Euromicro

1951-1991

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

ON PARALLEL ARCHITECTURES

Doron Tal*, Naphtali Rishe*, Sham Navathe**, and Scott Graham*

*School of Computer Science
Florida International University
University Park, Miami, FL 33199
Telephone: (305) 348-2025, 348-2744
FAX: (305)-348-3549; E-mail: rishen@fiu.edu

**College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

This paper discusses parallel computer system composed of tightly coupled processors that can coordinate to accomplish a concurrent solution of a common task. Section 2 presents parameters that are frequently used to classify parallel computer architectures. We also define the concept of granularity and discuss its effect on parallel computer architectures. Using Flynn's classification, Section 3 presents those architectures that follow SIMD (Single Instruction Multiple Data) principles. In Section 4 we present MIMD (Multiple Instruction Multiple Data) architecture types.

About the authors

Doron Tal is Assistant Professor of Computer Science at the Florida International University. His Ph.D. is from the Ben Gourion University. Tal's expertise is in parallel architectures.

Naphtali Rishe is Associate Professor of Computer Science at the Florida International University. His Ph.D. is from Tel Aviv University. Rishe's expertise is in databases: design, semantic modeling, implementation, languages, parallel architectures.

Shamkant Navathe is Professor at the College of Computing, Georgia Institute of Technology, Atlanta. His expertise is in database conversion, logical database design, database modeling, distributed database allocation, and database integration. Navathe's Ph.D. is from the University of Michigan.

Scott Graham is a Ph.D. student in Computer Science at the Florida International University.

This research has been supported in part by a grant from the Florida High Technology and Industry Council

1. Introduction

Parallel computer architectures are currently an area of intense research activity. This is evident by the wide variety of new computer architectures that have been introduced in the last decade.

There has always been a need for fast, high performance machines. In fact, in many applications such as weather forecasting, fluid dynamics, simulations, and artificial intelligence, the demand for high performance has always exceeded the current capability. One computer performance measurement can be described as a two dimensional graph where the switching device's speed is plotted along one axis and the amount of parallelism is plotted on the other one. Since the switching speed is limited (the propagation speed can not be faster than the speed of light), parallelism is left as a major hope for computational speedup.

In the course of computer history, many diverse research areas have been characterized as parallel computation. Forty years ago, arithmetic operations which processed words instead of bits were thought of as parallel computations. More recently, low-level parallel mechanisms [Duncan-90] have been introduced to attempt to make the processor execute the current instruction faster. Examples of this work include: overlapping processor work with auxiliary processors (I/O, terminal display, communications, *etc.*); providing concurrent execution of some arithmetic and logic operations by using several independent functional units; using multiprogramming and time sharing to provide a higher degree of resource sharing; and minimizing the transition to the next instruction by using instruction pipelining. Such a pipeline can be described as a flow line of stations that operate simultaneously, where each station always performs the same function (such as decode, fetch, *etc.*) on the stream of instructions that passes through the station. This is also called instruction lookahead. Most of these speedup efforts revolve around making the conventional von Neumann machine work more efficiently by continually attacking the bottlenecks inherent in the von Neumann architecture. Almost all contemporary commercial machines are equipped with the features listed above, but are still considered to be sequential machines.

Currently, the term parallel computation is associated with new architectures such as pipelined computers, array processors, and multiprocessor systems. It is evident that there is no universally accepted dichotomy between parallel computer systems and the conventional computer systems, which we would like to exclude from our definition. We define a parallel computer system as one composed of tightly coupled processors that can coordinate to accomplish the concurrent solution of a common task. Distributed computing systems are excluded from this definition since the processors are loosely coupled. Distributed systems typically have different problems than those of parallel systems, since each of the processors is autonomous and may refuse a request for service. All of the extended uniprocessor machines are also excluded from our definition. This report is devoted to a survey of parallel computer architectures which fit our definition.

2. Architecture Classification

Most sequential machines "fit" into a clearly defined architectural model or *paradigm*: a combination of the proverbial von Neumann machine with the more recent additions of virtual memory, direct access media, interrupt structures, concurrent I/O, and so forth. Unfortunately this is not so for parallel machines, whose developments have taken either no clear direction, or many different directions (*e.g.* pipelining [Ramamoorthy-77], connection machine [Hillis-85], systolic arrays [Kung-82], array processors [Padegs-88], data flow machines [Dennis-80, Arvind-84, Treleaven-82], *etc.*). We need a detailed classification scheme which will characterize all computer architectures. In the literature we can find several classification schemes [Hayes-88, Hwang-84]. However, Flynn's classification [Flynn-66, Flynn-72] is the most widely used. Flynn's taxonomy [Hwang-84] is based on the multiplicity of instruction streams and data streams. The following four categories are given:

1. *Single Instruction Single Data* (SISD) computer organization (Figure 2.1 (a)). This is the classical uniprocessor architectural paradigm. It may include parallel mechanisms (*e.g.* pipelining, overlapped CPU and I/O operations), though it is excluded from our definition of parallel computer architectures.

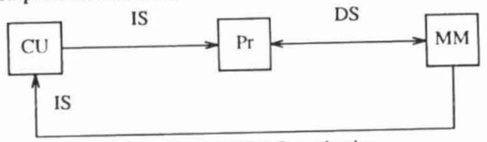


Figure 2.1 (a) SISD Organization

2. *Single Instruction Multiple Data* (SIMD) computer organization (Figure 2.1 (b)). Each processor executes the same instruction simultaneously on its own different set of data. Typically, multiple processors (or processing elements) are supervised by a common control unit. An interconnection network allows the transmission of operands between the processors.

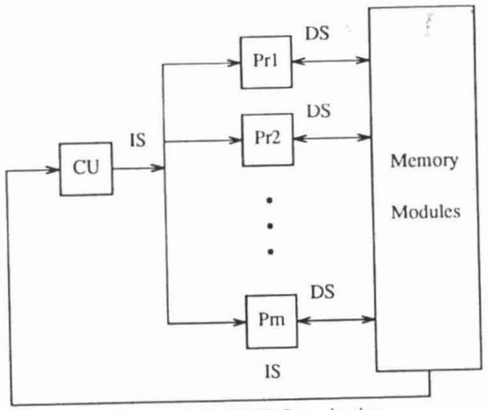


Figure 2.1 (b) SIMD Organization

3. *Multiple Instruction Single Data* (MISD) computer organization (Figure 2.1 (c)). This organization involves several processors, each of which is controlled by a separate control unit. The processors concurrently execute different instructions on a single stream of data. This structure has never been implemented and is considered impractical.

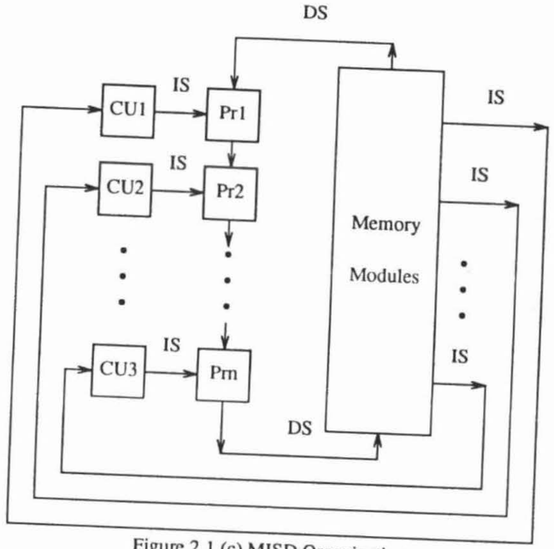


Figure 2.1 (c) MISD Organization

4. *Multiple Instruction Multiple Data* (MIMD) computer organization (Figure 2.1 (d)). In the MIMD computer organization, several processors simultaneously execute different instructions on diverse data. This organization also includes distributed computer systems. If all the data streams are derived autonomously from disjoint memories (or disjoint subsets of shared memories), then we actually have independent uniprocessor systems (multiple SISDs).

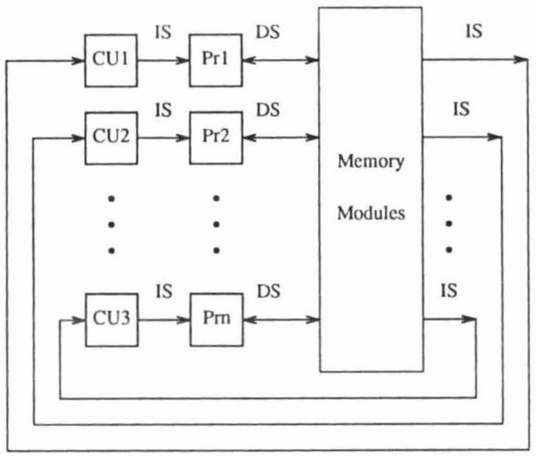


Figure 2.1 (d) MIMD Organization

It is quite interesting that any SIMD computer system can emulate a MIMD system by executing an interpreter and vice versa. Both SIMD and MIMD computer organizations constitute the type of parallel architectures that we are interested in.

The above discussion is too general and may not reveal the wealth and variety of parallel architectures that are currently available. Therefore, we employ one more parameter: *granularity*. When speaking about concurrency in parallel computer systems, one should carefully distinguish between the different granularity levels at which the concurrency is implemented [Tabak-90]. Granularity is discussed in the next sub-section.

This discussion is incomplete without mentioning some other parameters that are frequently employed to classify parallel architectures.

1. *Synchronous vs. asynchronous operation.* Synchronous parallel architectures coordinate their concurrent operations in lockstep, whereas asynchronous architectures do not. Usually, the synchronization is implemented using a global clock or a common control unit. Synchronous parallel architectures are easier to program and to control because the programmer is not responsible for coordination. However, this is achieved at the price of some inflexibility and overhead. The slowest operation determines the duration of an operation cycle. SIMD machines are typically synchronous.

2. *Feng's classification* (degree of parallelism). The number of bits that can be processed within a unit of time, is the degree of maximum parallelism [Hwang-84, Baer-80, Feng-77]. For example, a computer system that consists of M processors each with a bit length of N can process M*N bits concurrently.

3. *Handler's classification.* Wolfgang Handler has proposed a classification scheme based on six independent parameters [Hwang-84, Baer-80, Handler-77]: (a) the number of

processors; (b) the number of ALUs (or processing elements) under the control of a processor; (c) the word length; (d) the number of pipeline stages in each ALU; (e) the number of ALUs that can be pipelined; (f) the number of processors that can be pipelined.

2.1. Granularity of Parallelism

The parallel computation of a problem is done by partitioning the problem into sub-problems, each of which is performed separately. Granularity is defined as the ''size'' of a piece of code obtained by partitioning a problem. The granularity of parallelism varies greatly. At one extreme is the coarsest grain, the conventional von Neumann machines which are multiprocessed in a way such that each processor runs only its own processes. These processors may be symmetric (with each one running the same types of processes) or asymmetric (*e.g.* one may run a device handler only and reside in the device controller). As we move to finer-grained parallelism, we get multiple processors operating on the same process or pieces of a process. The smaller these pieces, the finer the granularity. The smallest possible pieces of a process are its primitive instructions, so at the smallest level of granularity we assign a processor to each instruction. As the granularity gets finer, some classical sequential algorithms become displaced by linear or near linear ones (*e.g.* those for systolic sorting or matrix multiplication).

The partitioning of a problem with running time T into N sub-problems (each running on a separate processor) does not always reduce the running time to T/N. Usually, as N becomes larger, the utilization ratio gets smaller. We are generally confronted with the following problems as we introduce finer and finer granularity solutions to concurrency:

1. *Data dependency.* Not all sub-problems (processes) can be running at the same time because of data dependency. As granularity becomes finer, data dependency usually becomes more of a problem.
2. *Interprocess communication.* The need for more processes to communicate increases an already crowded communication bandwidth and may lead to new bottlenecks. At the finest grain, each instruction execution is involved with receiving input operands and producing a result operand which usually has to be sent to one or more processors.
3. *Processor allocation.* Processor allocation and deallocation algorithms are required. As granularity becomes finer, each processor's computation thread becomes shorter. Hence, the allocation overhead becomes more critical.
4. *Concurrency exploitation.* Programs must be rewritten or rearranged to reveal and exploit maximum concurrency during execution.
5. *Races and contention.* Since several processors have to share resources, synchronization schemes such as semaphores or message based primitives may be required.
6. *I/O.* Even if we achieve a massive degree of parallelism, we still have to handle the I/O bottleneck. Traditional I/O structures are sequential and can not keep pace with the processors' performance.

These problems give rise to two questions. Is it cost-effective to have all of that parallel and possibly redundant computing hardware waiting around in abeyance so that maximum concurrency will be achieved whenever possible? What is the best granularity and architectural approach for maximizing computing speed and cost effectiveness?

The number and the diversity of the existing architectural concepts shows that the answers for these questions have not yet been determined.

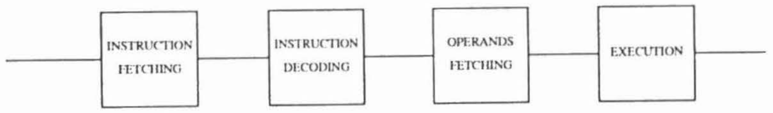
3. SIMD Architectures

3.1. Pipeline Architectures

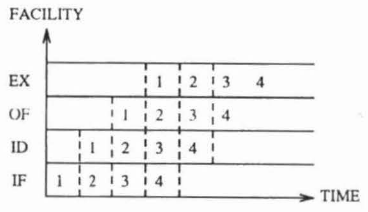
The name pipeline stems from an analogy with a typical industry assembly line in which the work to be done is broken into small pieces which are performed in an overlapping fashion. A part of the work is done at each station and a final result is obtained only after an item has passed through the entire pipeline. Since the pipeline stages must work together to process and forward data, they are triggered synchronously. The time required to perform one stage of the pipeline is called the stage cycle. The length of a pipeline's stage cycle is determined by its slowest stage. If the pipeline is saturated, the execution time is not a function of the total processing time, but rather one of the stage cycle.

Pipelines are classified into two primary categories: arithmetic pipelines and instruction pipelines. Arithmetic pipelines are used to implement complex arithmetic operations such as floating-point or vector operations. Instruction pipelining refers to partitioning instructions into several parts which can each be executed in an overlapped fashion with parts of other instructions. Simple forms of both categories can be found in almost any conventional computer.

An example of a typical instruction pipeline and its execution timing diagram is shown in Figure 3.1. The pipeline timing of Figure 3.1.b is characteristic of assembly lines. Theoretically, in a pipeline like that shown in Figure 3.1, the instruction cycle can be reduced to one-fourth of the original one. However, such a speedup may not be achieved due to any combination of the following difficulties: conditional branches, interrupts, structural hazards (overlapped execution of all possible instruction combinations may not be supported by the hardware), limitation on instruction fetch rate, and data dependency.



(a) Instruction pipeline execution.



(b) Space-time diagram.

Figure 3.1. Linear instruction pipeline.

Vector pipelines fall into the category of SIMD machines. Vector pipelines are characterized by high level operations which operate on one-dimensional vectors of data. The addition of two vectors is an example of such an operation. This kind of vector operation is equivalent to an entire loop of scalar operations. Currently, two primary types of vector pipelines exist: register to register vector pipelines where all vector operations (excluding store and load operations) are performed among registers, and memory to memory vector pipelines where special memory buffers are used instead of registers to hold the vector operands. Examples of the register to register type include the CRAY-1 [Russell-78], the CRAY-2 [CRAY-85], and the Fujitsu VP-200 [Lubeck-85]. The CDC's machines are examples of the memory to memory type.

3.2. Array Processor Architectures

An array processor is a regularly connected array of processing elements. The processors operate simultaneously and synchronously under the control of one central unit. A typical array processor is shown in Figure 3.2. Generally speaking, an array processor may also have a slightly different configuration where the individual processors access the memory modules through an alignment network. The control unit broadcasts a single instruction to be executed by each of the processing elements on its own data stream. Each processing element can perform the instruction slightly differently or can skip the instruction. Usually, the array processor is attached to a host computer through the control unit. Such array processors are suitable for solving very structured problems involved with vector computations. In order to take full advantage of the parallelism embedded in the architecture, the programmer must be aware of the computer's architecture while designing a program's algorithm and while allocating the program's data. Therefore, such arrays are most often used in scientific and special applications.

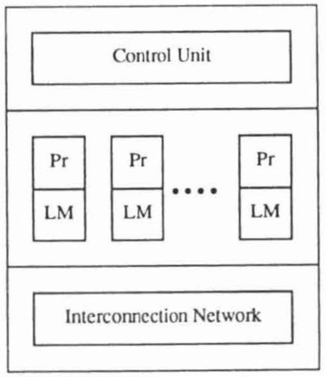


Figure 3.2. Array Processor.

The ILLIAC IV [Barnes-68] is an example of a typical array processor. It has a central control unit and 64 processing elements. Each of the processing elements has six registers and a local memory for the storage of distributed data. Essentially, each processing element is an arithmetic logic unit.

3.3. Fine Grained Array Processors

A particularly interesting extension to array processors resulted in the connection machine [Hillis-85, Hillis-87]. The connection machine is a fine-grained machine which consists of tens of thousands of simple processing elements. Each of the processing elements has a small local memory and a one bit wide arithmetic logic unit. The array processor is connected to a typical super-mini von Neumann machine front-end which originates the instructions to be executed by the connection machine's processing elements. Sixteen processors are connected in a grid. The processor groups are interconnected in the pattern of a Boolean n-cube. As in a conventional SIMD machine, an instruction is broadcast to all the processing elements via an instruction bus. Conceptually, the connection machine designers made further steps toward the implementation of a general purpose machine by introducing the virtual processor concept and a general purpose communication system. The communication system allows any processor to communicate with any other processor using a *send* instruction which passes a message to the other processor's local memory. The virtualization is implemented using a controller that interposes itself between the front end (host) machine and the array processor. The virtualization concept makes the efficient execution of an arbitrarily sized program feasible and allows the number of processors in the system to be expanded. (No matter how many *physical* processors we have, there is always an algorithm that will need more processors.) The connection machine is scalable: more physical processors can be added without having to change the program. The connection machine works efficiently for massive data parallelism applications. However, if the algorithm is characterized by many scalar operations (control parallelism) the performance may converge to that of a SISD machine.

3.4. Associative Array Processors

Some of the SIMD array processors (*e.g.* PEPE [Crane-72], STARAN [Batcher-74]) have been built around associative memories, and are called associative array processors. Associative memory is in a sense an active memory, since its storage cells are provided with special logical capabilities. Simple logic and arithmetic operations can be performed in the memory cells themselves, allowing parallel searches and comparisons to be made. The pieces of data stored in an associative memory are accessed by their contents instead of by their addresses. The characteristics of associativity and parallel processing allow associative array processors to access a large amount of information very quickly. They are considered effective in several applications in the area of information processing, in particular those applications involved with massive searches of very large databases. In addition, they can be used in applications such as weather forecasting computation and signal processing.

Associative processors can be classified into four categories [Haynes-82]: fully parallel, bit serial, word serial, and block oriented. The first category is characterized by having a logical unit in each memory cell which allows for full parallelism capability. This approach is simple, attractive and allows the highest performance. However, its implementation is very expensive and is impractical for large memories. In bit serial processors, a memory operation is applied on one bit column across a large number of words. Each bit column is selected by a special control unit and will be used in subsequent operations. Word serial processors implement a search loop in hardware. Block oriented processors are a hybrid structure of the first two categories and provide a tradeoff between the high cost of the fully parallel associative processor and the relatively slow operation of the bit serial associative processor.

3.5. Systolic Arrays

Systolic arrays have characteristics of both SIMD and MIMD architecture types. However, we choose to describe systolic arrays here in order to highlight their synchronous operation, which usually characterizes SIMD machines. Wavefront arrays [Kung-87] were stimulated by systolic array research and are a special case of systolic arrays where asynchronous operation has been favored. Systolic architectures are typically large, regular arrays of simple processors. These processors operate in parallel, passing data between themselves continuously and synchronously in a fixed, regular pattern. The most striking aspect of the systolic concept is its alternative to the von Neumann approach of instruction execution. As illustrated in Figure 3.3.a [Kung-82], using the von Neumann approach the instruction is fetched from memory and decoded, the operands are fetched, and an execution takes place yielding an operand which will be stored in memory again. In the systolic approach (Figure 3.3.b) data is pumped steadily through the array of cells without any memory reference. Planar systolic arrays with n^2 processors can often implement $O(n^3)$ algorithms in $O(n)$ time. The device efficiency is bounded below by a constant that is independent of n: the array makes maximally efficient use of each processor up to a constant factor. It is able to do this because the flow of data through the interconnection network is fixed, so there is no control overhead and optimized since very few processors ever idly wait for operands. Systolic arrays are generally hard-coded, but it is possible to program them on the fly. Systolic arrays are widely used as optimal solutions to many practical computational problems such as signal processing and matrix computation [Moore-87]. The technology favors the use of a few simple identical processing elements interconnected in a regular pattern yielding a short communication path and an economical VLSI design and implementation. Advances in VLSI technology have made it possible to locate these processors in physical

proximity, eliminating significant inter-chip delays. It must be pointed out that systolic arrays have some weaknesses, such as their inefficient handling of complex data structures and non-scalar operands, and the limitation that only a narrow set of problems can be mapped onto a particular systolic array.

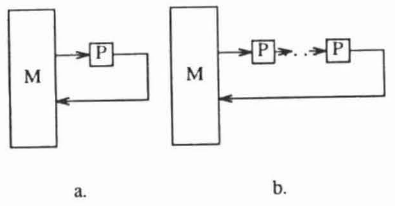


Figure 3.3. a. The von Neumann approach. b. The systolic approach.

4. MIMD Architectures

4.1. Multiprocessor Architectures

A multiprocessor computer is defined as one which employs at least two independent processors which operate under a combined control and which share memory modules, I/O channels, and an interconnection network. The interconnection network has a crucial role in determining the multiprocessor computer's performance. We use the interconnection network to classify the multiprocessor into three main categories: time shared busses, crossbar switch systems, and multistage networks.

Bus oriented systems contain one or more busses which connect all the processors and memory modules. The simplest and least expensive is a single-bus system. It is totally passive and is scalable. Components (processors and memory modules) can easily be added to or removed from the bus. Since all the functional units share a common bus, a mechanism must be provided to resolve bus contention. Among the methods used to resolve bus contention are static and fixed priorities, FIFO (First In First Out) queues, daisy chaining, and centralized bus controllers. Unfortunately, the single bus suffers from two primary drawbacks: the contention problem degrades system performance and may lead to a bottleneck, and a bus failure would be catastrophic to the system. Examples of a single common bus system are the IBM Stretch, Univac Larc, CDC 6600, etc. [Enslow-77]

Multibus systems overcome these drawbacks. The bus is no longer the single critical component. The contention problem is relieved because more than one interconnection can be set up at a time. Multibus systems do have their own set of problems. The system complexity is increased because extra logic is needed and because the multibus interconnection sub-system becomes an active device. Multiport capability is also required for all of the system's components. The 1108 Univac [Enslow-77] is an example of a multiport, multibus system.

The crossbar switch is the optimal solution for the contention problem. In a crossbar switch system, all the components are physically interconnected via crosspoint switches. Provided that no two accesses involve the same component, the contention problem is completely alleviated. The crossbar system is characterized by simple protocols and high hardware complexity. This system is the antithesis of the single common bus approach. The crossbar switch is impractical for a large number of components. If a system contains N components, it needs 2^N crossbar switch units. The Alliant multiprocessor system [Alliant-88] uses a crossbar switch between its processors and the cache memory modules.

A multistage network is a general representation for switching networks. In order to understand the network's operation we must first introduce the principles of a simple crossbar switch. The crossbar switch is the basic element of any multistage network. Figure 4.1 shows all of the possible interconnections of a 2x2 switch. If both inputs require a connection to the same output, then one request has to be delayed or rejected. In such a switch, the performance is limited by the switch setup time. The performance can be improved by using a buffer.

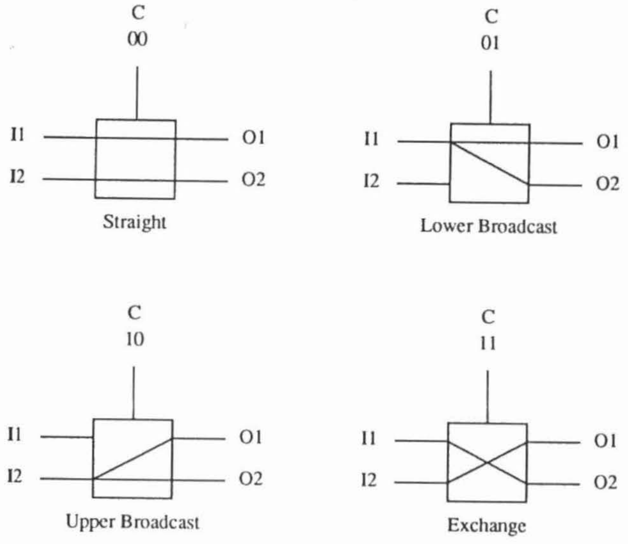


Figure 4.1. A 2x2 switch and all of its possible interconnections

A switching network that connects N inputs to M outputs is called an NxM switching network. It is composed of one or more stages of switching elements. Figure 4.2 [Hayes-88] shows an example of a three stage 8x8 switching network. One of the basic requirements for a multistage network is the full access property: each processor can be interconnected to any other processor. The network shown in Figure 4.2 has that property. A network which allows for the interconnection of any processor pair that is not currently interconnected without resetting the network is called a non-blocking network. The switching network shown in Figure 4.2 is a blocking network which can lead to communication delays. Among the popular interconnections in multistage networks are Banyan [Goke-73] and shuffle [Siegel-85].

One of the most common methods of implementing parallelism is through the use of a multiprocessor architecture. These architectures are the traditional design for parallel computers which are based on several coordinated von Neumann processors, each of which handles a portion of a problem. This approach entails memory access conflicts and high hardware and software overhead for processor synchronization and allocation.

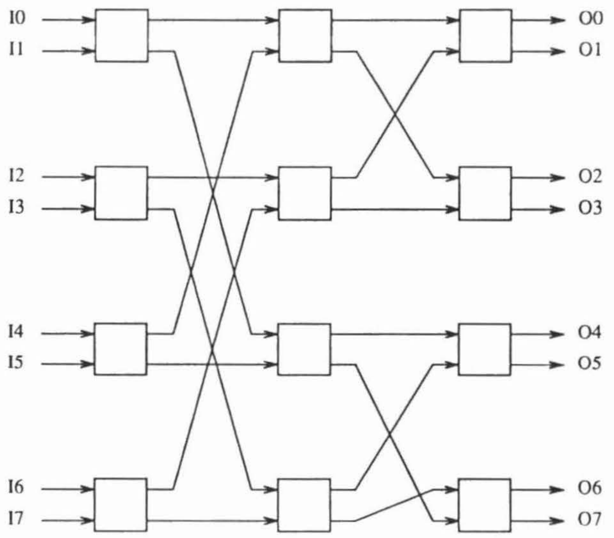


Figure 4.2. Three stage switching network.

4.2. Data Flow Architectures

The data flow model is a radical change and an alternative to the classical sequential instruction execution in the von Neumann approach. In the data flow model all the instructions are considered as independent entities which can each be executed as an independent concurrent action whenever each instruction has its input data. The data flow model of computation is based on two principles:

1. *Asynchrony*. All operations are executed when and only when the required operands are available.
2. *Functionality*. All operations are functions: there are no side effects.

The first principle provides for an execution mechanism in which data values pass through data flow graphs as tokens and in which an operation is triggered whenever all of the required input tokens are present at a node in the graph. The second principle implies that any enabled operations can be executed in any order or concurrently.

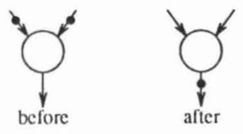


Figure 4.3. Data flow principles.

The data flow principles are illustrated in Figure 4.3. Large circles represent instructions, arrows represent arcs between instructions, and black circles represent data tokens. For an instruction to be enabled, tokens must be present at each input arc. Any enabled operator can be fired by removing one input token from each input arc, applying the specified function to the data, and placing the tokens labeled with the resulting value on the output arcs.

Figure 4.4 shows an example data flow graph which implements the computation of Z=(X+Y)*(X-Y). In this example, one computation by the graph can produce a value on arc Z, while a new computation on new values on arcs X and Y is ready to begin in a pipeline fashion.

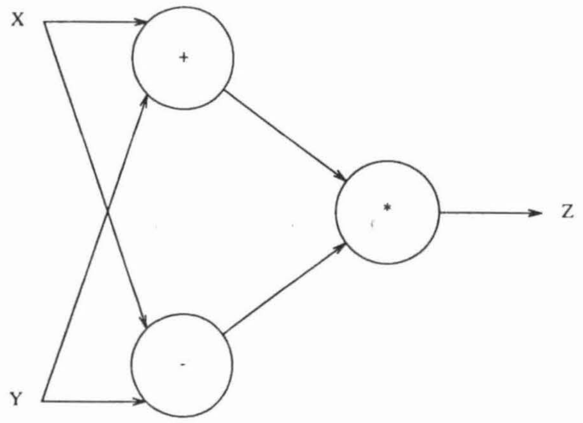


Figure 4.4. A data flow graph which implements the computation of Z=(X+Y)*(X-Y)

The data flow concept is very attractive because control is distributed out to the level of operations on scalar operands and parallelism in execution flows naturally as operands become available, thus following an algorithm's natural parallelism. Unfortunately, some problems have plagued the data flow concept since its inception:

1. A method to control and support a large amount of interprocessor communication is needed.
2. The handling of arrays, data structures, and large static data bases, is often inefficient, especially in ring architectures.
3. The law of granularity further impacts an already crowded communication bandwidth.
4. The absence of explicit storage imposes serious overhead problems such as the need to circulate storage constants and literals.
5. Operand accumulation causes storage and retrieval congestion.
6. The data-driven philosophy prevents lookahead and instruction overlap parallelism.
7. A process' data flow graph representation is limited to the size of the physical program storage.

5. Summary

We have shown a large number of different parallel architectural solutions. Many of these solutions are difficult to compare because of their differences in granularity, algorithmic approach, hardware implementations, and applicability in different problem situations. However, this discussion has tried to cover most of the existing architectural concepts and to explain how they work. In the following chapters we find novel proposals for parallel architecture paradigms. We hope that the review that has been conducted here will help the reader examine these proposals in a clearer light.

References

[Alliant-88] *Alliant FX/Series Product Summary*, Alliant Computer Systems Cor., 1988.

[Arvind-84] Arvind, D. E. Culler, "Why Dataflow Architectures", The 4th Jerusalem Conference on Information Technology, pp. 27-32, 1984.

[Baer-80] J. L. Baer, *Computer System Architecture*, Computer Science Press, Rockville, MD, 1980.

[Barnes-68] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes, "The Illiac IV Computer", *IEEE Transactions on Computers*, Vol. C-17, No. 8, pp. 746-757, 1968.

[Batcher-74] K.E. Batcher, "STARAN Parallel Processor System Hardware", *Proceedings AFIPS 1974 National Computer Conference*, ,Vol. 43, AFIPS Press, Montvale, N.J., pp. 17-22, 1974.

[Crane-72] B.A. Crane, M.J. Gilmartin, J.H. Huttenhoff and R.R. Shively, "PEPE Computer Architecture", *IEEE COMPCON*, pp. 57-60, 1972.

[CRAY-85] *CRAY-2 Computer System Functional Description*, Publication HR-2000, Cray Research, Inc., Mendota Heights, MN, 1985.

[Dennis-80] J. B. Dennis, "Data Flow Supercomputer", *IEEE Computer*, Vol. 13, No. 11, pp. 48-56, 1980.

[Duncan-90] R. Duncan, "A Survey of Parallel Computer Architectures", *IEEE Computer*, Vol. 23, No. 2, pp. 5-16, 1990.

[Enslow-77] P.H. Enslow JR, "Multiprocessor Organization - A Survey", *Computing Surveys*, Vol. 9, No. 1, pp. 103-130, 1977.

[Feng-77] T. Feng, "An Overview of Parallel Processors and Processing", *Computing Surveys*, Vol. 9, No. 1, pp. 1-2, 1977.

[Flynn-66] M. J. Flynn, "Very High Speed Computing Systems", *Proceedings, IEEE*, Vol. 54, No. 12, pp. 1901-1909, 1966

[Flynn-72] M. J. Flynn, "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, Vol. C-21, No. 9, pp. 948-960, 1972.

[Goke-73] R. Goke and G.J. Lipovski, "Banyan Networks for partitioning on Multiprocessor Systems", *Proceedings of the first Annual Symposium on Computer Architecture*, pp. 21-30, 1973.

[Handler-77] W. Handler, "The Impact of Classification Schemes on Computer Architecture", Proceedings, Int. Conference on Parallel Processing, pp. 7-15, 1977.

[Hayes-88] J. P. Hayes, *Computer Architecture and Organization*, 2nd. ed., McGraw-Hill, NY, 1988.

[Haynes-82] L.S. Haynes, R.L. Lau, D.P. Siewiorek, D.W. Mizell, "A Survey of Highly Parallel Computing", *IEEE Computer*, Vol. 15, No. 1, pp. 9-24, 1982.

[Hillis-85] W. D. Hillis, *The Connection Machine*, The MIT Press in Artificial Intelligence, U.S.A., 1985.

[Hillis-87] W.D. Hillis, "The Connection Machine", *Scientific American*, Vol. 75, No. 3, pp. 86-93, 1987

[Hwang-84] K. Hwang, F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, NY, 1984.

[Kung-82] H.T. Kung, "Why Systolic Architectures?", *IEEE Computer*, Vol. 15, No. 1, pp. 37-46, 1982.

[Kung-87] S.Y. Kung, S.C. Lo, S.N. Jean, and J.N. Hwang, " Wavefront Array Processors - Concept to Implementation", *IEEE Computer*, Vol. 20, No. 7, pp. 18-33, 1987.

[Lubeck-85] O. Lubeck, J. Moore, and R. Mendez, "A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and CRAY X-MP/2", *IEEE Computer*, Vol. 18, No. 1, pp. 10-29, 1985.

[Moore-87] W. Moore, A. McCabe, R. Urquhart eds., *Systolic Arrays*, Adam Hilger, Bristol England, 1987.

[Padegs-88] A. Padegs, B. B. Moore, R. M. Smith, and W. Buchholz, "The IBM System/370 Vector Architecture: Design Considerations", *IEEE Transactions on Computers*, Vol. 37, No. 5, pp. 509-520, 1988.

[Ramamoorthy-77] C.V. Ramamoorthy, "A Pipeline Architecture", *Computing Surveys*, Vol. 9, No. 1, pp. 61-102, (1977).

[Russell-78] R.M. Russell, "The Cray-1 Computer System", *Communication of the ACM*, Vol. 21, pp. 63-72, 1978.

[Siegel-85] H.J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing*, Lexington Books, D.C. Heath and Co., Lexington, MA, 1985.

[Tabak-90] D. Tabak, *Multiprocessors*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Treleaven-82] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", *Computing Surveys*, Vol. 14, No. 1, pp. 93-142, 1982.