THESIS

This is to certify that the

thesis entitled

TDMP: A DATA FLOW PROCESSOR

presented by

George Henry Simmons

has been accepted towards fulfillment
of the requirements for

Ph. D. degree in Electrical Engineering

David Fisher
Major professor

Date February 10, 1981

TDMP: A DATA FLOW PROCESSOR

By

George Henry Simmons

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical Engineering
and
Systems Science

1980

ABSTRACT

TDMP: A DATA FLOW PROCESSOR

By

George Henry Simmons

This research investigates a multiple processor computer structure that has a higher bandwidth than comparable single-processor machines yet is more flexible than existing fixed-array multiprocessors. The basic approach taken here was to develop, model, simulate, and then analyze a computer structure called TDMP - an acronym for Time Division Multiple Processing. TDMP's interprocessor communication network is based on time-division multiplexing and time-division switching techniques and has the following general properties: greater bandwidth than comparable single-processor structures; less complex switching network than a crossbar switch interconnection network; more flexible than fixed-array networks; full-access and non-blocking interconnection capabilities among the processors; simply extended to pipeline operation; and, finally, amenable to VLSI circuit implementation.

The justification for this research investigation is five-fold: First, single-processor systems have intrinsic bandwidth limitations known as the "von Neumann bottleneck." This bottleneck is due to the word-at-a-time style of processing through a single communication channel. As a result, computer structures based on this machine design can not exploit the inherent parallelism in specific tasks. Second, while fixed-array-processor structures can exploit this parallelism, their range of usefulness is limited by their fixed interconnection structure. This results in performance degradation when the task structure doesn't match the physical structure. Third, Dennis, et al., have designed computer structures based on data-flow principles, but they have not explored in detail alternative interconnection networks suitable for

data-driven computation. Fourth, time-division communication techniques have been successfully applied in the telecommunications industry but have not been investigated for use in multiprocessor structures. Yet this technique appears at first glance to hold good promise under certain overall computer system constraints. This fourth justification leads directly to the fifth. Namely, with multiprocessor structures implemented on VLSI circuit chips, the interprocessor and inter-memory communication paths require a disproportionate amount of chip area when compared with that required by the processors and their memory. So the communication channels determine a significant portion of both the cost in chip area and chip speed.

With all of these considerations in mind, the TDMP structure was defined, and APL computer simulation model was developed. This model formally defines the hardware structure, as well as the timing, switching delays, and communication protocols. The model also serves as the basis for analyzing the characteristics of the computer structure and for testing its usefulness in handling various multiprocessor tasks. We limited the model to a sixteen-processor structure.

Results of this investigation show that TDMP indeed has a higher bandwidth than comparable single-processor machines. Sample computations show that the specific TDMP architecture considered has fifteen and ten times the bandwidths of a single-processor system when computing eight-point FFT and second-order digital filter results, respectively. However, in order to achieve these bandwidth improvements over single-processor systems, the granularity of the tasks performed in each of the processors must be large. Results of this research also show that TDMP has greater flexibility than fixed-array multiprocessor structures. This is due to the non-blocking, full-access interconnection capabilities in the TDMP structure, which allows the data paths to be reconfigured for new application programs. Simulations also reveal that time-division techniques can be exploited to route information packets in a data-flow

structure with simple operating system constructs. Moreover, these time-division communication techniques do significantly reduce the VLSI circuit chip area devoted to data paths in data-flow structures.

ACKNOWLEDGMENTS

Never has one individual owed so much gratitude and sincere appreciation to so many people for help making a dream come true. Most notable of the many people is Dr. P. David Fisher, my advisor and thesis committee chairman. I would like to personally thank you for your guidance and support in achieving this ambitious task. You are a gifted professional and your contributions should not go unrecognized. I would also like to thank my thesis committee members for their suggestions and guidance in my doctoral program: Dr. S. Crouch, Dr. J. Forsyth, Dr. H. Hughes, Dr. J. Kreer and Dr. R. Reynolds.

Above all, I would like to thank my wife, Grayce, and my son, Gavin, for the support, tolerance and encouragement they gave me while achieving this task.

TABLE OF CONTENTS

Chapter			Page
I.	INTRODUCTION		1
II.	BACKGROUND		4
	2.1	von Neumann Bottleneck	4
	2.2	Multiprocessing Survey	8
	2.2.1	Illiac IV	8
	2.2.2	Cmmp	10
	2.2.3	PM4	14
	2.2.4	Indirect Binary N-Cube	16
	2.2.5	Star-100	21
	2.2.6	Cray-1	23
	2.2.7	Systolic Arrays	23
	2.2.8	Other Commercial Multiprocessors	26
	2.3	Observations	26
III.	TIME-DIVISION MULTIPLE PROCESSING		31
	3.1	TDMP Architecture	31
	3.1.1	PE Array	31
	3.1.2	Time-Division Switching Network	37
	3.1.3	Control Unit	42
	3.2	TDMP Implementation	43
	3.3	TDMP: A Data Flow Processor	44
IV.	TIMING CONSIDERATIONS		51
	4.1	Timing Diagrams	51
	4.2	Maximum Multiplexing and Demultiplexing Rates	53
	4.3	TDS Maximum Switching Delay	55
V.	TDMP SIMULATION MODEL		62
	5.1	TDS Simulation Model	64
	5.1.1	TDS Control Unit	69
	5.2	PE Simulation Model	74
	5.3	TDMP Simulation Model Operation	83

Chapter			Page
	5.3.1	PE Operation	83
	5.3.2	TDS Operation	86
	5.4	TDMP Simulation Model Summary	90
VI.	TDMP PERFORMANCE EVALUATION		92
	6.1	Performance Data	93
	6.2	Comparison of Computational Performance	98
	6.3	Discussion of Simulation	107
	6.4	TDMP and Dennis Data Flow Models	109
VII.	CONCLUSIONS		111
	7.1	Summary	111
	7.2	Further Research	115
	REFERENCES		117

LIST OF FIGURES

Figure		Page
2.1	Alternative implementation of the 2nd-order difference equation -- non-von Neumann style processor.	7
2.2	A block diagram of the structure of ILLIAC IV.	9
2.3	A block diagram of the data paths between processing units in ILLIAC IV.	11
2.4	A block diagram of the CMU Multiminiprocessor (Cmmp) architecture.	13
2.5	A block diagram of the PM4 architecture.	15
2.6	A block diagram of the indirect binary N-Cube array architecture.	18
2.7	A block diagram of an individual switch node in the indirect binary N-Cube array.	19
2.8	A block diagram of the indirect binary N-Cube array control system.	20
2.9	A block diagram of the Star-100 data paths.	22
2.10	A functional block diagram of the Cray-1 architecture.	24
2.11	The hex connected systolic array for matrix multiplication.	25
3.1	Block diagram of TDMP architecture.	32
3.2	Block diagram of 12x1 PE array and control unit.	33
3.3	Block diagram of a TDMP processing element.	35

Figure		Page
3.4	Example of incoming and outgoing PE communication paths.	36
3.5	Example of time slot interchanging in the TDS.	38
3.6	Example of both space and time switching in the TDS.	39
3.7	Block diagram of a single input/output time-division switching network.	40
3.8	Data flow program graph of the 2nd-order difference equation.	45
3.9	Example of a data flow branch operation.	47
3.10	Example of higher-level conditional expression evaluation.	48
4.1	Timing and control signals for a 16 time slot TDMP system.	52
4.2	Matrix representation of the P and Q control signals.	54
4.3	Flowchart for TDS control algorithm.	56
4.4	An example to illustrate the A_1N versus A_2Log_2N switching delay issue.	59
5.1a	Block diagram of the TDS APL simulation model.	65
5.1b	Part of the TDS APL simulation model.	66
5.2	TDS simulation model memory map.	68
5.3	Block diagram of TDS the control unit.	70
5.4	Addressing modes for the TDS network C-move processor.	72
5.5	Control unit logic network specification table.	73
5.6	Examples of TDS processor move operations.	75

Figure		Page
5.7a	Block diagram of the PE APL simulation model.	76
5.7b	Part of the PE APL simulation model.	77
5.8	PE states for data flow computing.	80
5.9	Logic truth table for the outgoing part of the CILU.	81
5.10	Logic truth table for the incoming part of the CILU.	82
5.11	Flowchart of the PE control algorithm.	84
5.12	PE control program code.	85
5.13	Flowchart for the TDS control algorithm.	87
5.14	TDS control program code.	88
6.1	Block diagram representation of a single PE TDMP model.	94
6.2	Timing diagram for a two time slot model.	96
6.3	Signal flow graph of an 8-point FFT.	99
6.4	Bandwidth comparison as a function of the average number of FFT operations.	103
6.5	Bandwidth comparison as a function of PE execution time for the FFT computation.	105
6.6	Bandwidth as a function of multiplication time for the digital filtering computation.	108

CHAPTER I

INTRODUCTION

In many scientific and engineering computer application areas, the computer's computational speed limits its range of usefulness. Examples of such areas include energy and power modeling, weather modeling and forecasting, fluid dynamics studies, computer-assisted tomography, and artificial intelligence.[1] Several parallel approaches are being taken to extend the useful range of computers. The first approach involves reducing the switching delay times of elementary gates, which embraces both the areas of device physics and integrated circuit (IC) technology.[2] With the second approach, the switching delays are assumed to be fixed but improved methods are sought for performing primitive single-operand and double-operand arithmetic operations, such as transcendental functions, integer arithmetic, and floating-point arithmetic.[3] The third approach assumes that the primitive arithmetic operations are given; instead, the focus is on the design of computer architectures that overcome the intrinsic speed limitations of the von Neumann machine.[4] The first approach is inherently limited in that the effects of reduced IC dimensions and the introduction of new logic families is expected to reduce gate delays and, consequently, improve overall computer system performance by only a couple of orders of magnitude.[1,5] While this represents an important improvement, it does not in itself achieve the long-range improvements needed. Gains with the second approach will result largely due to decreased cost per bit and increased density per bit of semiconductor memories and logic arrays. Older algorithms for primitive arithmetic operations will be modified to fully exploit speed improvements possible with ROM look-up tables or the advantages of performing concurrent conditional computations, such as conditional sum addition.[3,6] While the third approach is ultimately limited by gate delays and the speed of primitive arithmetic algorithms, it provides the potential for making the largest increase in computational speed. Here, multiprocessors are employed; and these multiprocessor architectures exploit the inherent parallelism in specific tasks, thereby eliminating the single processor bottleneck--"von Neumann bottleneck."[4]

We will restrict ourselves to this third approach. Moreover, we will restrict our attention to a class of multiprocessor architectures that can be implemented on a single very large scale integration (VLSI) chip or a small number of such interconnected chips.

Historically, high-performance multiprocessor computer systems have been very expensive, special purpose, and generally research-oriented tools. VLSI technology is changing this so as to make it feasible to build high-performance multiprocessor structures as low-cost, single-chip system components. VLSI technology is a statement about system complexity, not about transistor size or circuit performance. VLSI defines a technology capable of creating systems so complicated that coping with the raw complexity overwhelms all other difficulties.[2] From this definition, we can see that the way in which the computer industry designs multiprocessor computers in VLSI technology must, in fact, be different from the way it has traditionally designed computers in other technologies. For example, in VLSI technology the transistors will be almost "free" and the interconnection data paths - communication - will determine the cost in both area and speed of the chip.[2] This is true because the interconnection paths in VLSI technology are the same width as a transistor, which means that these paths reduce the available active chip area. So, for VLSI implementation of multiprocessor structures simple and regular underlying communication geometry is required to reduce the total amount of interconnection path lengths on the chip. In addition, the processors should be identical to reduce the layout time and effort of the architecture, the design should be partitionable into segments of manageable size and these designs should have a wide range of use to justify the costs for a component manufacturer. As a result, this work is based on the following premise: Continued advances in integrated circuit fabrication technology will permit chip complexities to increase more than three orders of magnitude over what they were at the end of 1979.[7] This research investigates an alternative single-chip microprocessor architecture that exploits this technology to improve significantly the computational capabilities and general usefulness of small computer-based systems suited for signal processing computations such as waveform generation, modulation and

filtering. Specifically, an alternative multiprocessor structure, called time Division Multiple Processing (TDMP), is investigated and has the following general properities:

— greater bandwidth than comparable single-processor architectures;

— less complex switching network than a crossbar switch interconnection network;

— more flexible than fixed-array network;

— full-access and non-blocking interconnection capabilities;

— simply extended to pipeline operation;

— highly compatible with data flow algorithms;

— amenable to VLSI implementation.

The TDMP structure is evaluated by comparing its performance to a single-processor architecture as one boundary of performance and to a fixed-array-processor architecture as the other boundary of performance. Bandwidth, hardware complexity, flexibility, and regularity are the four principal figures of merit.

Chapter 2 of this thesis contains a brief review of several key existing multiprocessor structures, including their performance and range of usefulness. In Chapter 3, the organization and operation of TDMP is presented, along with an analysis of hardware complexity. Chapter 4 presents timing diagrams, closed-form expressions for the maximum multiplexing and demultiplexing rates and the maximum switching delays. Chapter 5 describes the TDMP simulation model. In Chapter 6, the simulation results of two applications are given. The first is a fast-Fourier-transform and the second involves a digital filtering computation. We also compare the computational bandwidth among the TDMP system, single-processor system and fixed-array-processor system for these two applications. Finally, Chapter 7 gives a summary of this research as well as some suggestions for future research.

CHAPTER II

BACKGROUND

2.1 von Neumann Bottleneck

Single-processor computer systems have intrinsic speed limitations that have been ascribed to the "von Neumann bottleneck." The term "von Neumann bottleneck" was coined by Backus[7] to represent the word-at-a-time style of processing that is characteristic of von Neumann machines--the model computer conceived by von Neumann and others about 35 years ago.[7,8] The von Neumann computer is composed ideally of a central processing unit (CPU), a memory that contains data and instructions, and a connecting channel that can transmit a single word at a time between the CPU and the memory (and send an address to memory). The connecting channel is where the von Neumann bottleneck occurs. The reason for its name is that all computational tasks in the von Neumann machine must be accomplished entirely by pumping single words back and forth through this connecting channel. A large part of the traffic in the bottleneck is not useful data but merely names of data, as well as operations and data used only to compute such names. As a result, single-processor bandwidth, defined as the maximum throughput measured in terms of the maximum number of results that can be generated per unit time, is always limited by the von Neumann bottleneck as the computational operations get sufficiently larger or complex.

An example will serve to illustrate the issue. In digital filtering, a group of operations is performed once for each sample (in time) of the signal being processed. For purposes of this calculation, assume that the processor is a 16-bit microprocessor with a 200 nsec cycle time and the floating-point multiplication, division, addition and subtraction operations are performed in a hardware coprocessor.[9,10] Single-precision, floating-point addition and subtraction are performed in 14 μsec and 18 μsec respectively, and double-precision extended multiply and divide

operations are performed in 27 μsec and 39 μsec, respectively. If the second order recursive filter

$$H(z) = \frac{Y(z)}{X(z)} = \frac{A_0 + A_1 z^{-1}}{1 - B_1 z^{-1} - B_2 z^{-2}} \tag{1}$$

is implemented with this processor as a difference equation of order 2 with constant coefficients

$$Y(n) = A_0 X(n) + A_1 X(n-1) + B_1 Y(n-1) + B_2 Y(n-2), \tag{2}$$

then the maximum filter bandwidth is limited to approximately 3.3 kHz. This bandwidth is fixed by the minimum time it takes the processor to perform the group of operations for each sample of the signal.

The group of operations consists of one-at-a-time execution of four multiplication and three addition operations. For simplicity, we assume it takes zero time to move data and to fetch instructions from memory. In this case, the minimum execution time is,

$$T_{\min} = 4t_M + 3t_A = 4(27\ \mu sec) + 3(14\ \mu sec) = 150\ \mu sec$$

where t_M and t_A are the floating-point multiplication and addition times, respectively. Therefore, for every $T_{\min}$ units of time a group of operations is completed, this corresponds to a processor bandwidth of $(T_{\min})^{-1} = 6.6\ kHz$ and a maximum filter bandwidth of $(2T_{\min})^{-1} = 3.3\ kHz$. If the order of the filter gets higher, the number of group operations increases; consequently, the processor and maximum filter bandwidth decreases. The shrinking maximum filter bandwidth is caused by the von Neumann bottleneck and limits the processor's usefulness in many filtering applications, such as those found in telecommunication systems, where the maximum filter bandwidth requirement is 4 kHz and the filter order is 5.[11]

The von Neumann bottleneck can be eliminated by exploiting the inherent parallelism in a group of operations. For example, if the group of operations for the second order difference equation in our previous example is implemented as shown in Figure 2.1, the processor bandwidth is significantly improved. Three successive independent stages of computations are executed in parallel and the output of one stage feeds the next, analogous to an industrial assembly line. If we assume a continuous input stream of values, zero time to move data between stages and ignore any start-up times, the minimum execution time, $T^{\bullet}_{min} = \max\{t_1,t_2,t_3\} =$ speed of the slowest stage in the system, where $t_1 = t_M$, $t_2 = t_A$ *and*, $t_3 = t_A$. Processor bandwidth is $\frac{1}{t_M} = 37\ kHz$ because every $T^{\bullet}_{min} = t_M$, a result can leave the system. This determines a maximum filter bandwidth of

$$BW = \frac{1}{2T^{\bullet}_{min}} = \frac{1}{2t_M} = 18.5\ kHz.$$

A direct comparison shows that $5.6T^{\bullet}_{min} = T_{min}$; hence, the non-von Neumann style processor has a five-fold processor and maximum filter bandwidth improvement over the von Neumann style processor. This improvement was achieved by eliminating the one-operation-at-a-time style of processing through a single channel--the "von Neumann bottleneck."

Recognizing that the von Neumann bottleneck limits processor bandwidth, we must develop processor systems that compute larger units of the task at hand. To accomplish this, multiprocessing concepts are employed in new processor architectures to improve bandwidth.

Two widely used forms of multiprocessing are parallel processing and pipeline processing. Parallel processing improves processor bandwidth by using many processors operating in parallel, either on different data sets or on different portions of the same data set.[12,13] Since, in fact, the processors are not always independent, they may require access to the same data or interchange of results between processors.[14,15] Flexible interconnection networks are needed

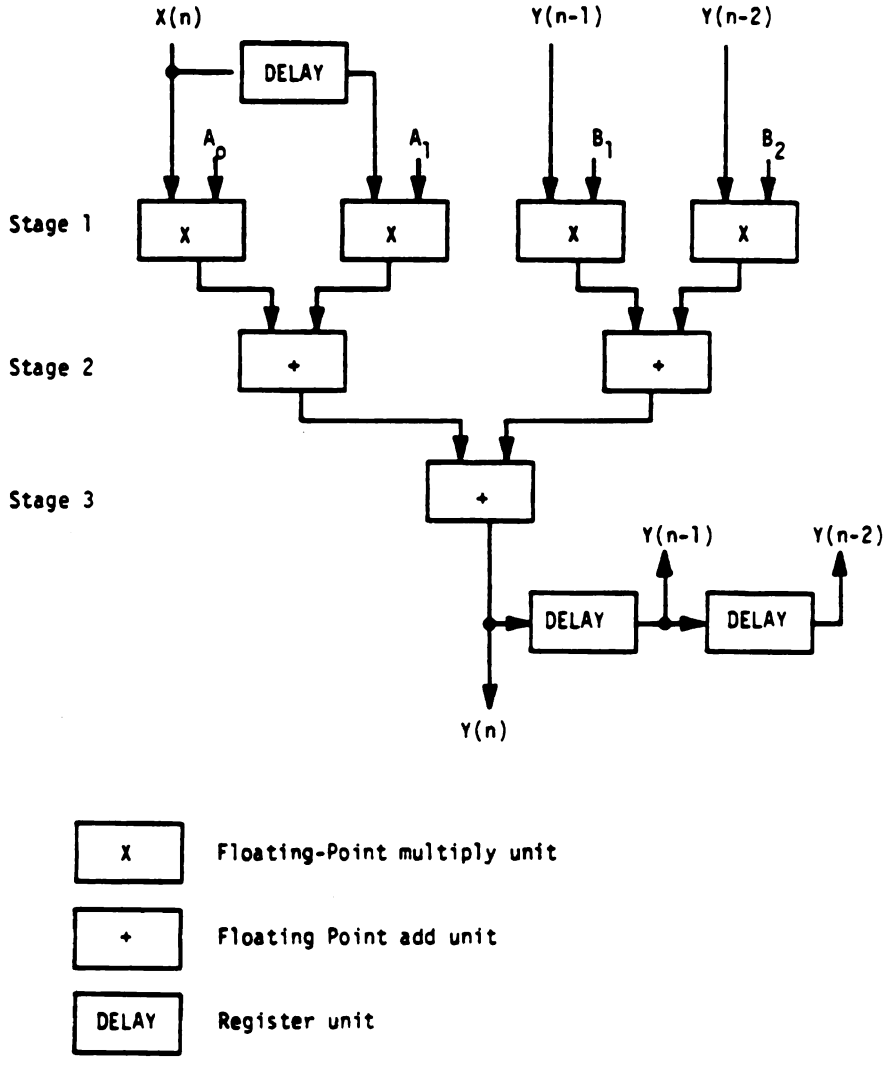


Fig. 2.1 Alternative implementation of the 2nd order difference equation -- non-von Neumann style processor.

here to efficiently handle the communication problems. By flexible interconnection networks, we mean networks that can change their data flow pattern under program control. It turns out that most applications are sped up by transforming their traditional sequential algorithms to use the multiprocessor structure, not by executing on a processor with ever shorter memory access times.[16]

Pipeline processing, on the other hand, improves processor bandwidth by subdividing the processing to be done into sequential functions and then assigning a processor to each function. These functions are then arranged in a "pipeline" so that the output of one stage feeds the next.[13,17,18] The processor bandwidth of this approach is limited by the speed of the slowest processor in the pipeline if fixed interconnection networks are used to feed the pipeline stages. By a fixed interconnection network, we mean a network whose data flow pattern cannot be altered. Jump presents quantitative techniques for the evaluation and comparison of digital system pipelines.[17]

2.2 Multiprocessing Survey

In this section, we briefly review several key multiprocessor structures in order to identify their multiprocessing attributes. We also suggest how well suited these attributes are for VLSI implementation for enhanced computation. Unless dictated by the need for understandability, we have avoided material not directly related to the multiprocessing aspects of these systems.

2.2.1 Illiac IV. Illiac IV is a single-instruction stream multiple-data stream (SIMD) experimental computer designed at the University of Illinois in the late 1960's.[19] The general structure of Illiac IV is shown in Figure 2.2. It contains 64 identical processing units (PU) with a common external control unit (CU), a four nearest-neighbor interconnection structure, an interface to a supervisory host computer and a switching network for interchanging data and instructions between the CU, host computer and memory. A PU comprises a processing

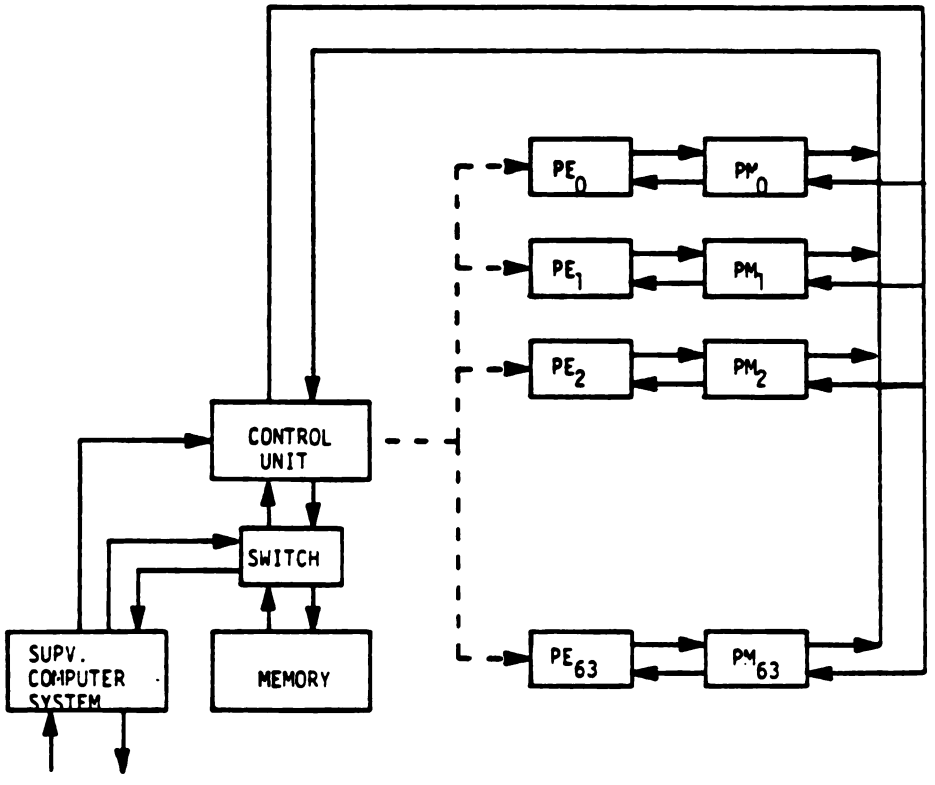


Fig. 2.2 A block diagram of the structure of ILLIAC IV.

element (PE) and a processing memory (PM). A PE is a general-purpose arithmetic logic unit (ALU) capable of executing a conventional instruction set that includes 64-bit floating-point operations. Each PM has a capacity of 2 k, 64-bit words.[13,19] The control unit fetches instructions from a processing element's memory, decodes them, and issues control pulses to the processing elements for execution. It broadcasts memory addresses and data words when they are common to all processors. An instruction can be either a control or a processing unit instruction. The former directs operations local to the control unit, whereas the latter controls the execution of the processing units. The control is designed to overlap the executions of the two different instruction types.[4]

In addition to the common data and control buses that link the PUs to the CU, there are direct data paths connecting each PU to four neighboring PUs. Specifically, PU_i is connected to PU_j if $j = i + 1$ (modulo 64), $j = i - 1$ (modulo 64), $j = i + 8$ (modulo 64) or $j = i - 8$ (modulo 64). The PUs form a two-dimensional array, Figure 2.3. For this reason, Illiac IV is often referred to as an array processor.[4,13,20]

The array organization is very effective in exploiting parallelism when the characteristics of the problem to be solved match the physical structure of the array.[4,12] Matrix operations provide an example of this kind of problem.[12] The uniform processors and simple, regular communication paths satisfy VLSI implementation requirement.[21] A disadvantage of the array organization is the inflexibility in the interconnection structure. This results in performance degradation when the problem structure does not match the physical structure of the array and reduces the universal appeal which brings about some high pressure constraints for VLSI implementation.[4,12,22] The failure of a single processing element can hamper the operation of the entire system; however, by adding alternate data paths in each processor, similar to a Chordal Ring interconnection network, the system would have a graceful degradation.[23]

2.2.2 **Cmmp.** Cmmp is a multiple-instruction stream multiple-data stream (MIMD)

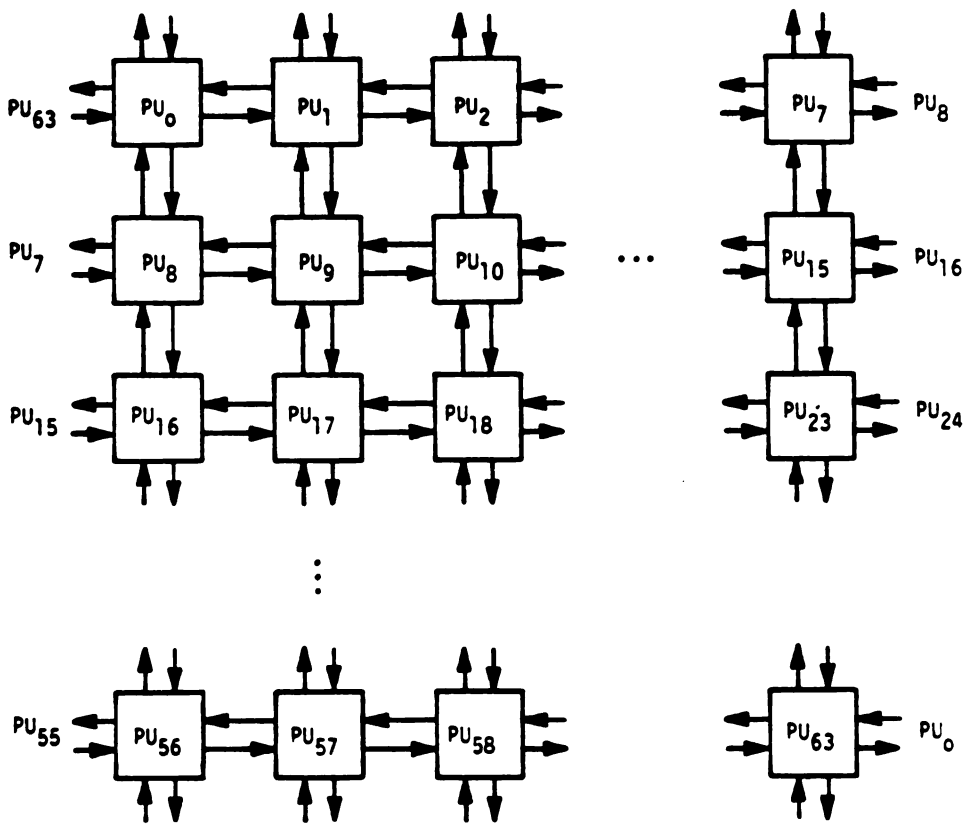


Fig. 2.3 A block diagram of the data paths between processing units in ILLIAC IV.

multiprocessor computer designed and built at Carnegie-Mellon University for computer architecture and artificial intelligence research work.[14] The hardware and software were designed with the goals of symmetry and general purpose in mind. By symmetry, we mean that replicated components, such as processors, are treated as an anonymous pool; no one of them is special in any sense. By general purpose, we mean the multiprocessor character of the machine is used to improve throughput across a set of independent jobs as well as to multiprocess single jobs.[16] The system consists of 16 PDP-11 minicomputers connected to each other through an interprocessor bus and individually connected to 16 memory modules through a full-access, non-blocking, crossbar switching network. Each minicomputer operates like an independent processing system with its own primary memory and controllers. The interprocessor bus allows any processor to generate an interrupt to any subset of the processor configuration at any of several priority levels. No data is carried by this bus. The crossbar switch allows any processor to establish a path to any memory module--full access and all permutations of individual processors connected to individual memory modules are possible--non-blocking. The switch is under both processor and manual control. Collectively, the 16 processors execute six-million instructions per second; the total memory bandwidth is about 500 million bits per second. Despite the fact that Cmmp is built from minicomputers, it is a large-scale machine. Figure 2.4 shows the basic structure of Cmmp along with other system components. The other system components include another crossbar switching network to allow any processor to communicate with any of the various controllers which manage secondary memories, and input/output devices.

The basic design goals of Cmmp were achieved, and the research revealed some important observations about multiprocessor systems. Specifically, that the raw speed in the design of the switching network and the processor is not very important but reliability of the system is very important.[16] The crossbar switching network provides full-access and non-blocking interconnection capabilities for processor communication. Some criticisms of the system are: the hardware

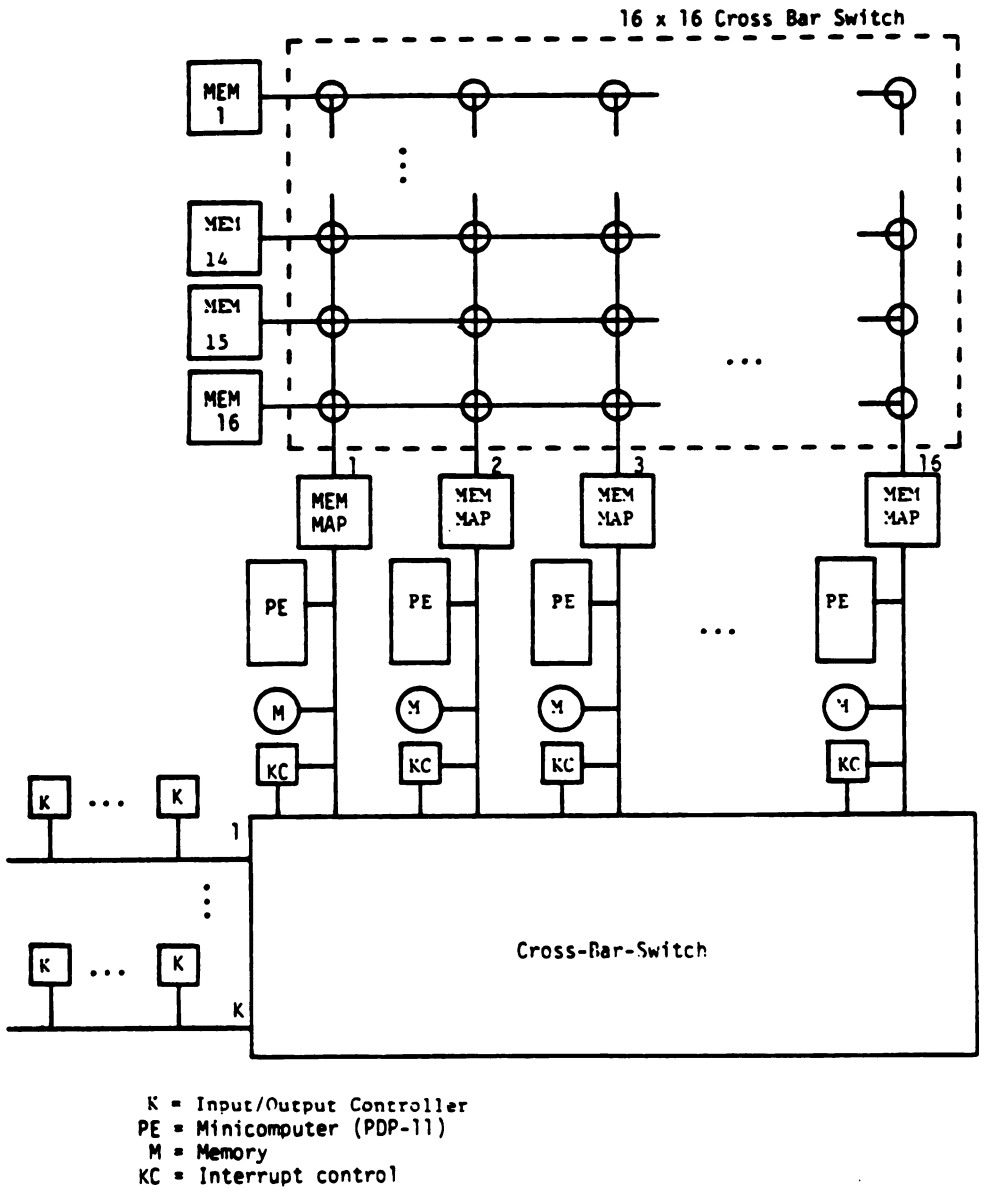


Fig. 2.4 A block diagram of the CMU multiminiprocessor (Cmmp) architecture.

is less reliable than desired, they were unable to partition Cmmp into disjointed systems, and there is not enough human engineering into the software interface to the user.[16] The N^2 growth of the crossbar networks make this architecture a poor candidate for VLSI implementation. The crossbar connecting paths and switches use substantial amounts of the chip area, thereby reducing the size of a network that can be implemented on a given chip.[24]

2.2.3 **PM4.** PM4 is a reconfigurable multiprocessor system for pattern recognition and image processing research work and is currently under development at the Advanced Automation Research Laboratory of Purdue University.[15] The system envisioned consists of hundreds of Large Scale Integration (LSI) bit-slice microprocessors and a three-level hierarchical memory connected by a set of interconnection networks.[15,25]

Figure 2.5 shows a basic block diagram of PM4. The system consists of N identical processors with local memory (PMU), K identical vector control units with local memory (VCU), shared memory connected to the processors through a delta interconnection network (PMIN), file memory connected to processors by a shared bus and a yet undesigned interprocessor connection network which permutes data among processors (IPCN).

The system can reconfigure its resources under system control to assume four different operation modes:

1. SIMD MODE - Single-Instruction Steam and Multiple-Data Stream (SIMD), Illiac IV.[19] In this mode, the same instruction is executed by a subset of the processors operating on different data streams. This mode is used for vector operations with the vector control unit.

2. Multiple SIMD Mode - In this mode, a multiple number of SIMD operations are executed in parallel.

3. MIMD Mode - Multiple-Instruction Stream and Multiple-Data Stream (MIMD), Cmmp.[14]

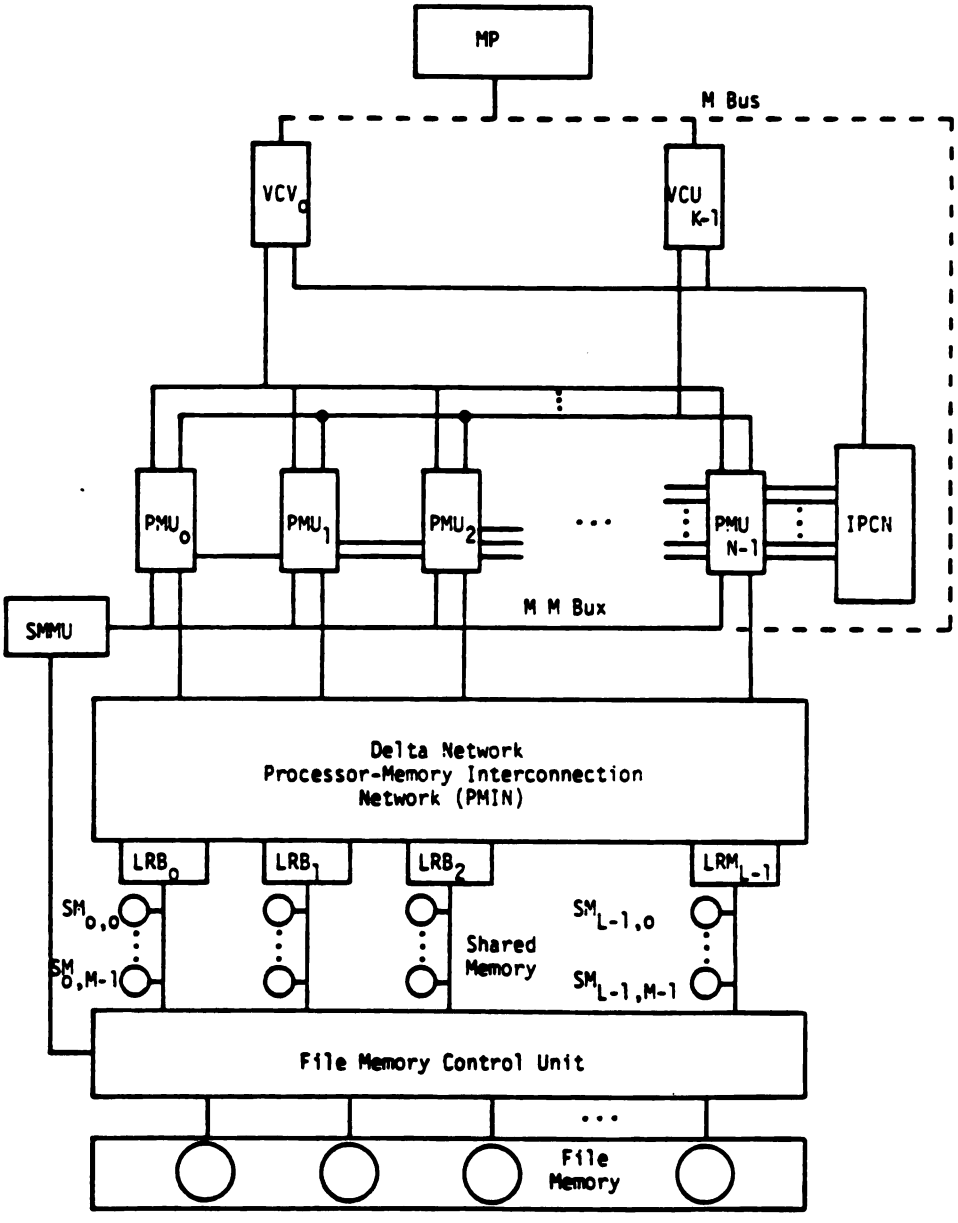


Fig. 2.5 A block diagram of the PM4 architecture.

The processors perform independent tasks on separate data streams concurrently.

4. Distributive Mixed Mode - In this mode, SIMD vector instructions and parallel MIMD processes are simultaneously executed.

The Vector Control Units (VCU) are used in the SIMD mode of operation. The VCU broadcasts instructions on the vector control bus to all processors that are assigned to the SIMD process. The VCU may also send control signals to a time-shared Interprocessor Communications Network (IPCN) to switch the data in a group of processors. The IPCN is time-shared among VCUs during multiple SIMD processes. The VCU has the ability to mask or disable processors so that only the active or unmasked processors execute the broadcasted instructions. During execution, a multiplexor is used to route broadcasted instructions from a VCU to a processor. A delta interconnection network is used to connect the processors to shared memory for block transfer of information.[26,27]

PM4 architectural features are based on existing machines such as Cmmp and Illiac IV.[14,19] The architectural advantages of the above systems have been incorporated into PM4. The flexibility in the PM4 architecture allows it to overcome some of the architectural disadvantages in existing MIMD and SIMD machines. However, because of the processor speed requirements for swift reconfiguration and communication geometry (three different interconnection networks), a single or couple chip VLSI implementation of PM4 may not be applicable. Microprocessors such as Motorola 6800, LSI 11, Intel 8086, and Z8000 do not meet PM4's speed requirements and neither will VLSI processors, since they are likely to be of only moderate speed.[15,28] VLSI performance measures indicate that chip area requirements for delta networks are the same as those for crossbar networks.[24] The difficulties of VLSI implementation of crossbar networks were discussed in Section 2.2.2.

2.2.4 **Indirect binary n-cube.** The indirect binary n-cube microprocessor array is a multiprocessor architecture in which processors are interconnected by a switching network whose set of

connections can be described by the set of edges of the binary n-cube.[28] It is called indirect because the array is not actually interconnected according to the topology of the binary n-cube. The basic form of the array is illustrated in Figure 2.6 for $n = 4$, $N = 2^4 = 16$. The circles in Figure 2.6 represent the microprocessors, indexed from 0 to 15 as indicated by the numbers in the circles. The lines on the right from the switching network connect back to the microprocessors with the indices given in parentheses. Each switch node, indicated by the squares in Figure 2.6, has two input lines, two output lines and can be put into either of the two states shown in Figure 2.7, providing a "direct" or a "crossed" connection. Each level of switching represents a dimension in the n-cube; in this case, n = 4.

The network is used to permute data among microprocessors. In some cases, it may be necessary to make multiple passes through the network to obtain permutations of the data that are otherwise unrealizable. Multiple passes will, of course, entail a sacrifice of speed, but give added flexibility. In addition, as the n-cube dimensions get larger, the number of switching levels in the array increase, which also sacrifices speed in the system.

The system has a two-level control system for the microprocessors, based on variable microprogramming stored within the microprocessors. This means that global commands sent by the main control unit to the microprocessors may be interpreted differently by each microprocessor. For the switch nodes a set of switch controllers are used to control a set of switch nodes. These switch controllers receive global commands from the main control unit. The control system is shown in block diagram form in Figure 2.8.

This processing array can be used effectively for a broad range of SIMD applications. The regularity and modularity of its structure makes it an attractive candidate for VLSI implementation. However, the range of application of the array is limited by the structure of the binary n-cube. The binary n-cube does not have full-access and non-blocking capabilities.[28] In addition, the complexity of implementing indirect binary n-cube networks in VLSI technology is the

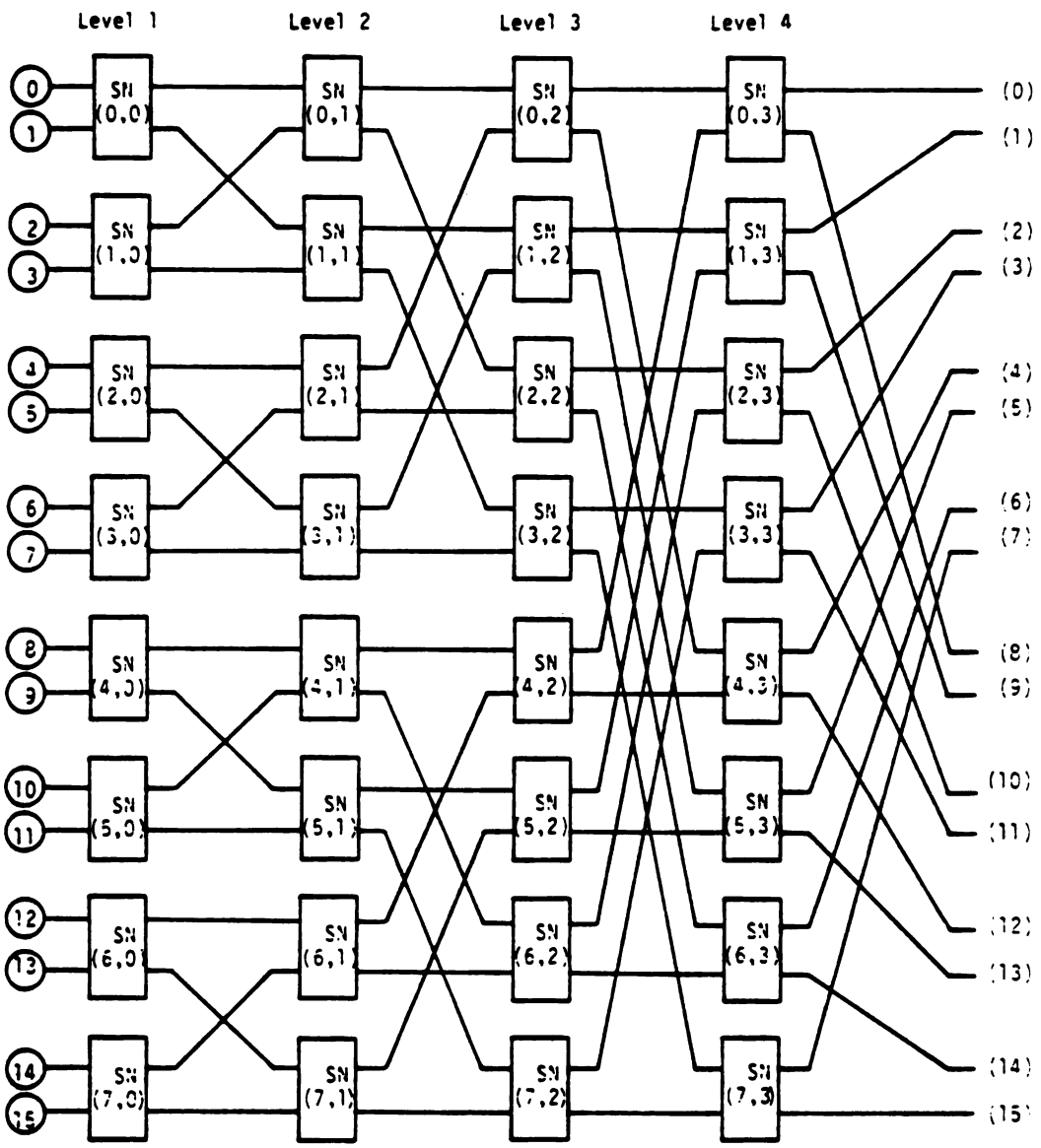
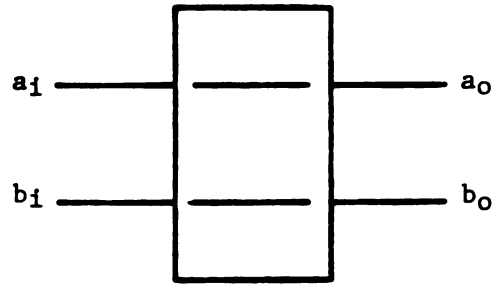
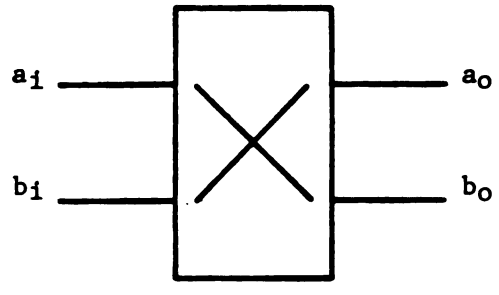


Fig. 2.6 A block diagram of the indirect binary n-cube array architecture.



(a) direct



(b) cross

Fig. 2.7 A block diagram of an individual switch node in the indirect binary n-cube array.

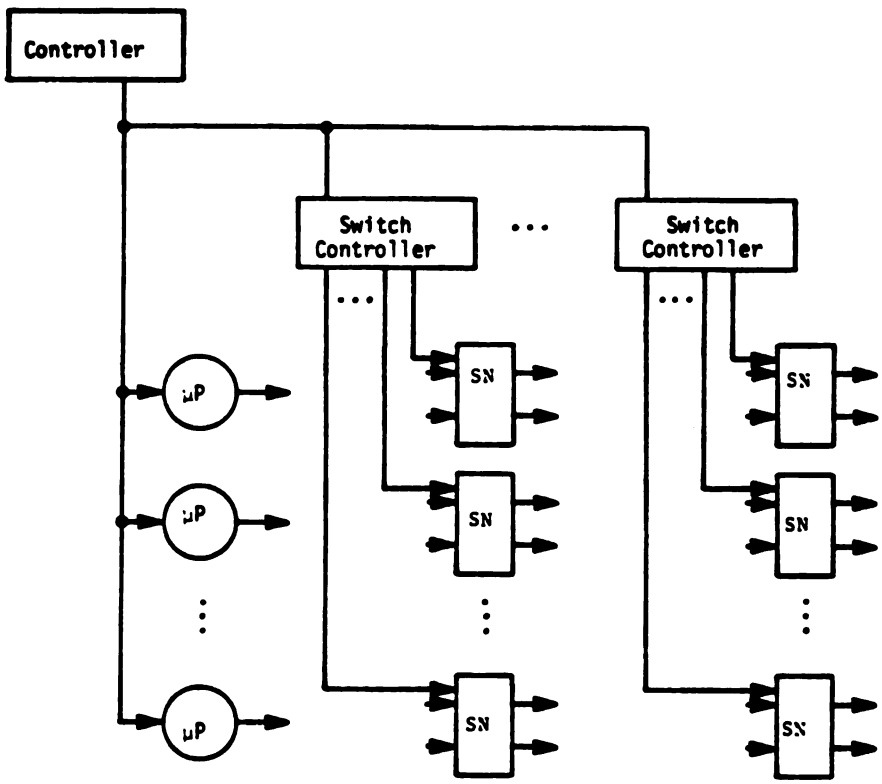


Fig. 2.8 A block diagram of the Indirect binary n-cube array control system.

same as that for crossbar networks.[24]

2.2.5 **STAR-100.** The Control Data STAR-100 computer is a high-performance pipeline processor structured around a four-million byte, high-bandwidth memory.[20,29] Instructions specify operations on variable length data streams. A block diagram of the STAR-100 memory-pipeline data paths is shown in Figure 2.9. The core memory and the data bus configuration have been designed to support a pipeline rate of 100 million 32-bit floating-point operations per second. The core memory has 32 interleaved banks, with each bank containing 2 k 512-bit words. This memory system can support 512 bits of data per minor cycle and there are 32 minor cycles.

The width of the memory data bus for each group of four banks is 128 bits. The data bus transfer rate is 128 bits per minor cycle. Four buses are active with each bus transferring data at a rate of 128 bits per minor cycle. Two of the buses are used for transferring operand streams to the pipeline processor. The third bus is used for storing the resulting stream elements and the fourth bus is shared between input/output storage segments and control vector references. The read and write buffers are used to synchronize the four active buses. The memory segments are buffered to space them eight banks apart, eliminating memory conflict situations.

The floating-point arithmetic section of the STAR-100 consists of two independent pipeline processors. Processor 1 consists of a pipeline floating-point addition unit and a pipelined floating-point multiplication unit. Processor 2 consists of a pipelined floating-point addition unit, a non-pipelined floating-point divide unit and a pipelined multipurpose unit which is capable of performing a floating-point multiplication, divide, or square root operation.

The memory interleaving and pipeline processing makes STAR-100 very efficient for processing vectors. Furthermore, its overall system design allows scalar processing as well. However, with respect to flexibility, the architecture is non-flexible and applications must be

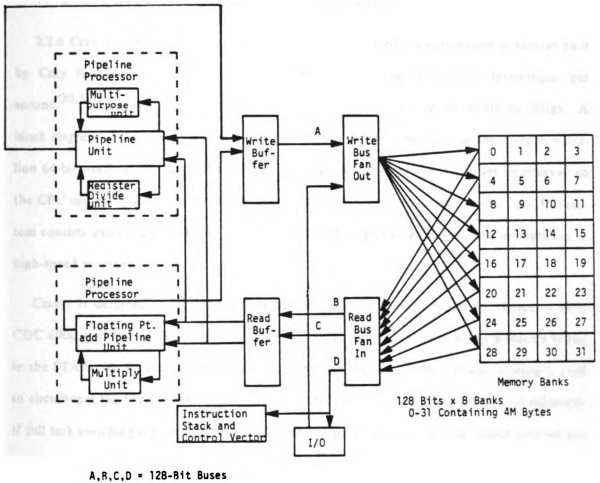


Fig. 2.9 A block diagram of the Star-100 data paths.

structured around a vector for most efficient performance. The long start-up times are seen as a disadvantage and the communication network between the processors and memory banks--wires--is seen as a disadvantage of VLSI implementation.[2,20]

2.2.6 **Cray-1.** Cray-1 is a very high-speed function-oriented general purpose computer built by Cray Research Incorporated and is capable of processing 80 million instructions per second.[20,30] Both scalar and vector processing capabilities are incorporated into its design. A block diagram of the Cray-1 is shown in Figure 2.10. The main memory can be up to one million 64-bit words of 50 nsec cycle-time bipolar memory. The memory is 16-way interleaved so the CPU can easily achieve a data transfer bandwidth of one word per clock cycle. The I/O system consists exclusively of channel connections to other computers and channel connections to high-speed permanently mounted disks.

Cray-1 is designed to extend the independent functional unit concepts developed in early CDC 6000 and 7000 series equipment.[20] While the system avoids some setup problems found in the STAR-100, its architecture suffers from the fixed vector length. Vector chaining is used to circumvent the fixed vector length problem.[30] However, the machine will not run efficiently if full task switching is done very frequently and special designs for each functional unit are not conducive to a single chip VLSI implementation.[2,30]

2.2.7 **Systolic arrays.** Systolic arrays are special-purpose, high-performance multiprocessor devices. A systolic system is described by a network of processors which rhythmically compute and pass data through the system. Every processor regularly pumps data in and out, each time step performing some short computation, so that a regular flow of data is kept up in the network.[21]

Many basic matrix computations can be pipelined on systolic arrays composed of many interconnected inner product step processors. Figure 2.11 shows a hex connected systolic array for matrix multiplication. An inner product step processor is a processor that performs the

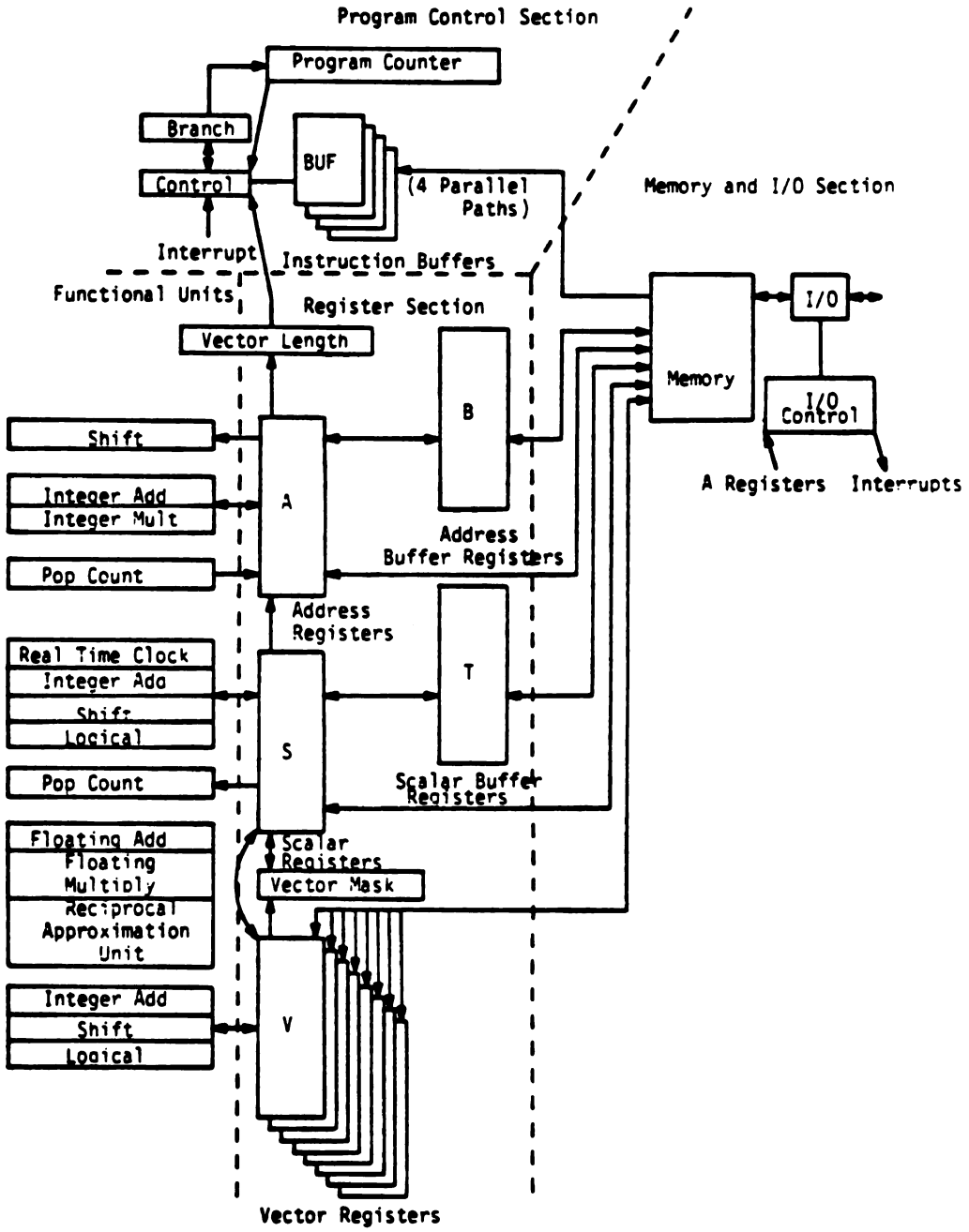


Fig. 2.10 A functional block diagram of the Cray-1 architecture.

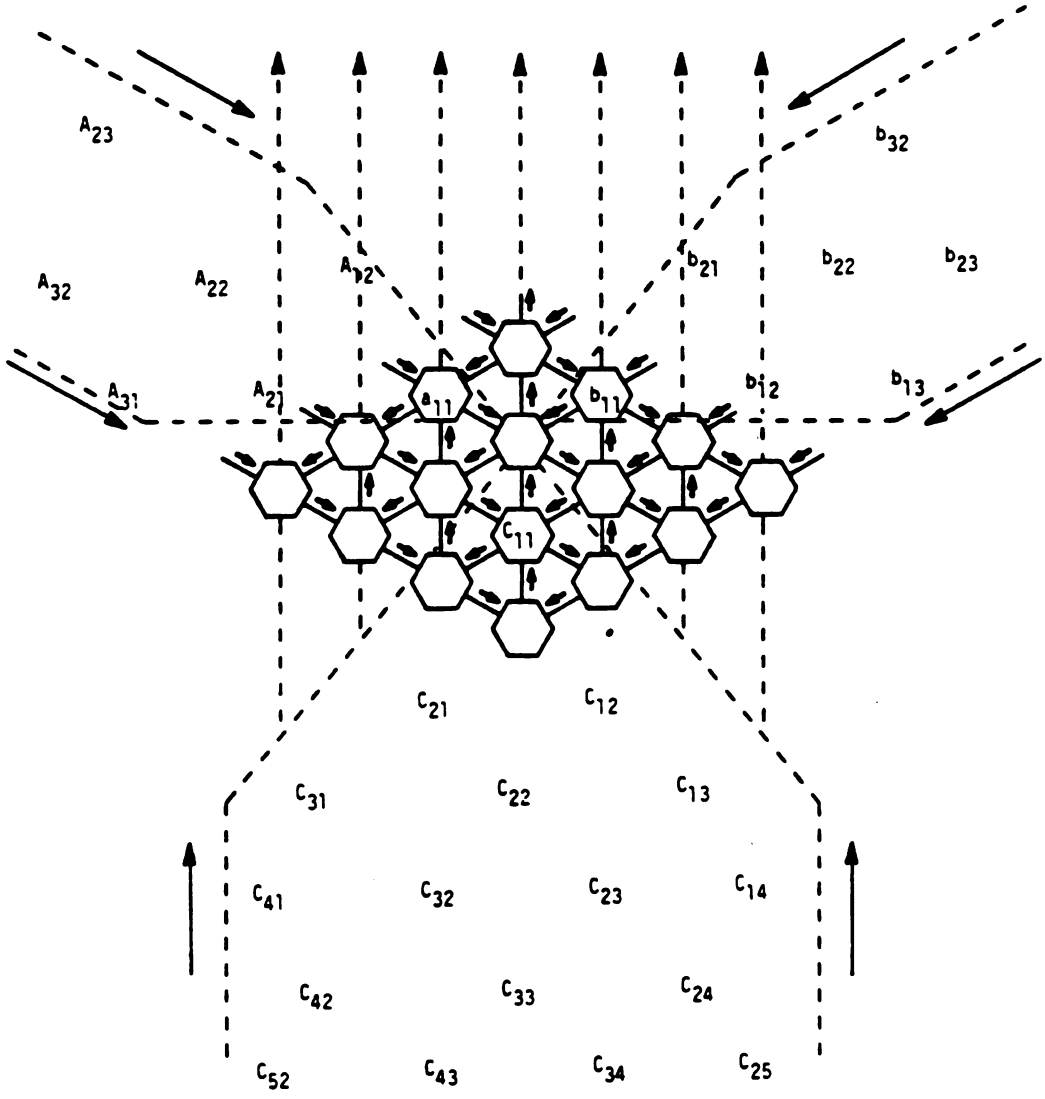


Fig. 2.11 The hex connected systolic array for matrix multiplication.

computation, $R_C = R_C + R_A \times R_B$ and makes the input values for R_A and R_B together with the new value of R_C available as outputs. Kung gives detailed examples of matrix computations on systolic arrays.[21]

The systolic computations are characterized by the strong emphasis upon data movement, pipelining in particular. The arrays have simple and regular communication paths and almost all processors are identical. This makes systolic arrays attractive for VLSI implementation. However, the regular communication paths are obtained by sacrificing flexibility in the network. The rigid data flow paths are not reprogrammable and their structure depends on the computation problem. As a result, a unique array must be provided for each different computation problem and, if the array does not have universal appeal, then the use of such a VLSI part in a new architecture brings about some high pressure constraints.[22]

2.2.8 **Other commercial multiprocessors.** There are other multiprocessor systems besides the ones we have reviewed: IBM 370/168, CDC Cyber 170, Honeywell Series 60 level 66, Univac 1100 Model 80 and Burroughs B7700, to name a few. Each of these multiprocessing systems has distinct advantages and unique constraints with implications for performance. Some of these systems have architectural organizations similar to the previously reviewed systems. The Cyber 170, for example, embodies the principle of functional partitioning similar to the Cray-1 system. A survey which highlights some of the architectural strategies of the above commercial multiprocessor systems is found in reference 31.

2.3 Observations

Our brief review of several key multiprocessing architectures provides us with a basis for making some general comments about multiprocessing systems. One central issue in their design relates to the networks over which the multiple processors communicate to other processors or memory modules. Clearly, as the number of processors increases, the characteristics of this communication network become critical to overall system performance, cost,

and reliability. For example, Cmmp uses crossbar switching networks for communication purposes. An N-by-N crossbar switch has full-access and non-blocking interconnection capabilities. However, the difficulty with crossbar is that network costs grow with N^2. Given VLSI performance bounds, crossbar networks are infeasible for single-chip multiprocessor systems.

Pease's indirect binary n-cube microprocessor array uses the binary n-cube switching network and Purdue's PM4 uses a delta switching network. Both of these networks have similar complexity and cost. The number of switch nodes in these systems grows with $Nlog_2 N$; and, if implemented in small scale integration (SSI) or medium-scale-integration (MSI) technologies, their network costs are cheaper than crossbar. But, if these networks are implemented in VLSI, their network costs are comparable to crossbar.[24] Binary n-cube and delta networks are blocking, which means under certain conditions messages going to different output ports will require use of the same path between two switches. Since, under the assumed protocol, only one message can hold a given path during message transmission, blocking will occur. This blocking reduces the bandwidth and application range of the networks and introduces added delays in the system operation. Other multiprocessor systems such as Illiac IV and systolic arrays have even more restricted communications networks. Illiac IV uses a four-nearest-neighbor interconnection and systolic arrays have fixed data path interconnections. This results in performance degradation when the problem structure does not match the physical system structure and the utility of such systems is limited by their specificity.

The question thus arises, what type of interconnection network can be placed on a VLSI multiprocessor chip that will enhance computation and prove the multiprocessor chip useful in a large number of applications? In Chapter 3, Time Division Multiple Processing (TDMP), we suggest time division switching networks as an alternative solution to this problem.[32] Time-division switching networks have full-access and non-blocking interconnection capabilities like crossbar networks, use very few data paths, and the network costs are dominated by the

cost-per-bit of semiconductor memory. However, no single network is generally considered "best" since the cost effectiveness of a particular design varies with such factors as the computational tasks for which it will be used, the desired speed of interprocessor data transfers, the actual hardware implementation of the network, the number of processors in the system, and the cost constraints on the construction.[33]

The processors used in a multiprocessor system may or may not be identical. The Cmmp, Illiac IV and systolic arrays are examples of systems with identical processors. The Cray-1 (in which the independent functional unit concepts are employed) is a system with different processors. It is essential that the processors used in a VLSI multiprocessing system be identical modules organized in a simple, regular fashion with a minimum number of connecting paths between modules because such geometry leads to high density and, more importantly, to modular design. The TDMP system adapts to these VLSI requirements by using identical processor-memory modules connected to time-division multiplexed buses.

Another issue in the design of multiprocessor systems is their performance. Invariably, the scaling in performance of a multiprocessing system is sublinear. That is, using an N-processor multiprocessing system always yields less than N times the performance of the corresponding single-processor based systems. This is true for both N-unit parallel and N-unit pipeline architectural organization of multiprocessor systems.

One the one hand, we have multiprocessor architectures, like Cmmp, that are very flexible and have a wide range of application. However, this flexibility is at the expense of additional software and hardware operating system overhead needed for communication and control. On the other hand, we have multiprocessor architectures, like systolic arrays, that have very high performance but limited range of application. The very high performance architectures have low operating system overhead, but this low overhead is at the expense of fixed interconnected data paths between processing elements to enhance the computation for a specific problem. So,

in our investigation we seek a balance between these two extremes with a multiprocessor architecture that has performance and flexibility bounded by a general purpose architecture like Cmmp, on one hand, and a non-flexible special purpose architecture, like systolic array, on the other. We also seek an architecture that allows a simple software interface to the user. Conventional architectures speed up most applications by executing algorithms that have been transformed to use the multiprocessor structure or by fixing the transformed algorithm into the interconnection wiring of the multiple processors. Both approaches have proved successful in many ways; however, in every case permitting large amounts of parallel activity, it has proven far more difficult to obtain parallelism in software than to provide it in hardware. In view of the nature of parallel hardware systems and the practical difficulties of keeping them running at full speed, we investigate an architecture that executes data flow programs. Data flow programs are seen as a natural reinterpretation of conventional programs with parallel execution in mind. This approach abandons the classical instruction driven computing. The underlying problem with most current attempts to use parallel hardware is that they are based on traditional concepts of programming.[36] These concepts in turn are based wholly on the serial von Neumann computer design, with instructions executed one at a time. In particular, use of a program counter remains obligatory. In instruction pipelines, no attempt is made to alter the basic von Neumann model. In vector and array processors, one instruction may operate on many pieces of data, but only one instruction executes at a time. In multiprocessors, many program counters step through subprograms simultaneously presenting complex problems of communications such as memory conflict. The use of a program counter is inappropriate when programs are intended for parallel execution. Efforts to develop a model of computation which can effectively express parallelism have yielded a form of program representation known as data flow. Execution of a data flow program is data-driven; that is, each instruction is enabled for execution just when each required operand has been supplied by the execution of a predecessor instruction. Dennis and Misunas, Gurd and Watson have done work on the design of

computers based on the data flow concepts.[37,39] The system we investigate is a new, simpler implementation based on some of their work.

CHAPTER III

TIME-DIVISION MULTIPLE PROCESSING

In this chapter, we present the Time-Division Multiple Processing (TDMP) architecture and examine its hardware complexity. It is called "time-division" because the transmission and switching of information among the multiple processors is done with time-division techniques; i.e., time-division multiplexing and time-division switching. These techniques are useful in meeting the interconnection wiring and pin constraints imposed by VLSI design while enhancing overall arithmetic computation. The key to TDMP is the integrated transmission and switching system that provides the communication channels among the processors. After describing TDMP, we very briefly discuss implementing it using VLSI technology. The intent here is to show that TDMP is very compatible with VLSI implementation. Then we consider TDMP as a data flow processor and investigate how it executes programs expressed in data flow notation.

3.1 TDMP Architecture

The basic TDMP organization is illustrated in Figure 3.1. The components consist of an N x M array of processing elements (PE), time-division switching networks (TDS), and a control unit (CU). We will give a brief description of each component and its interrelationship with the others.

3.1.1 **PE array.** The basic structure of the N x M PE array is shown in Figure 3.2 for N = 12, M = 1. N represents the number of PEs that share a unique pair of two-way incoming and outgoing time-division multiplexed (TDM) buses. The number in each PE box gives the PE time slot position on these buses. M represents the number of unique pairs of incoming and outgoing TDM buses in the array. The incoming buses are shown as two-way communication lines to the left of individual PEs and the outgoing buses are shown as two-way communication lines to the right of individual PEs. The remaining lines are control lines that

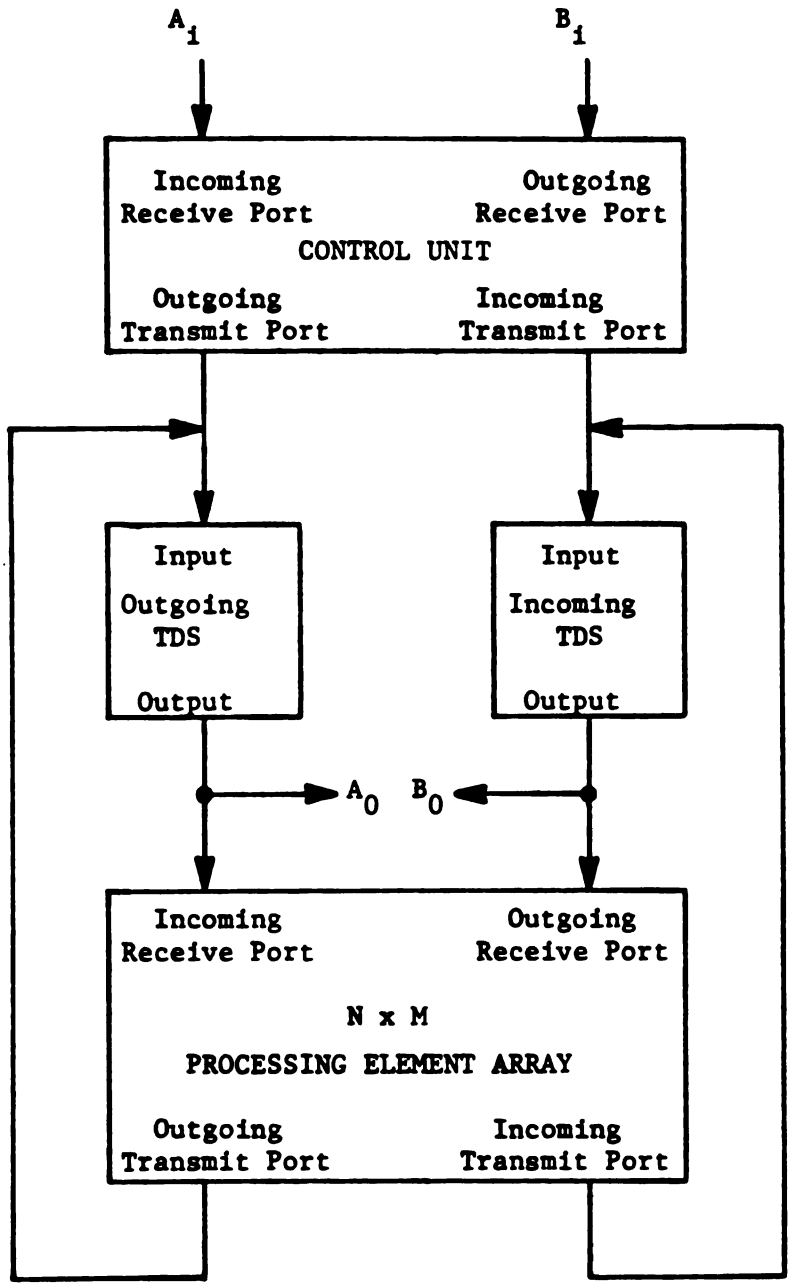


Fig. 3.1 Block diagram of the TDMP architecture.

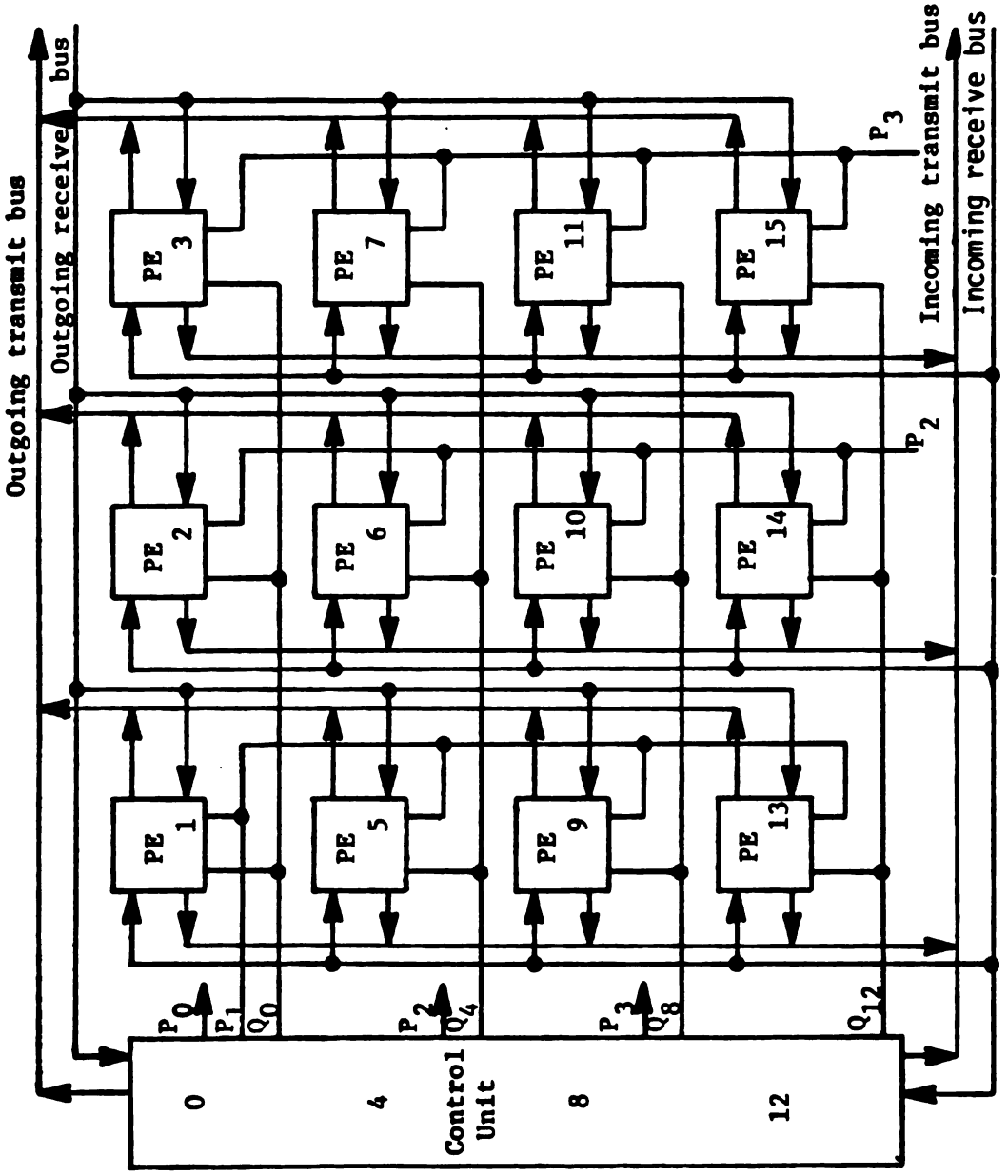
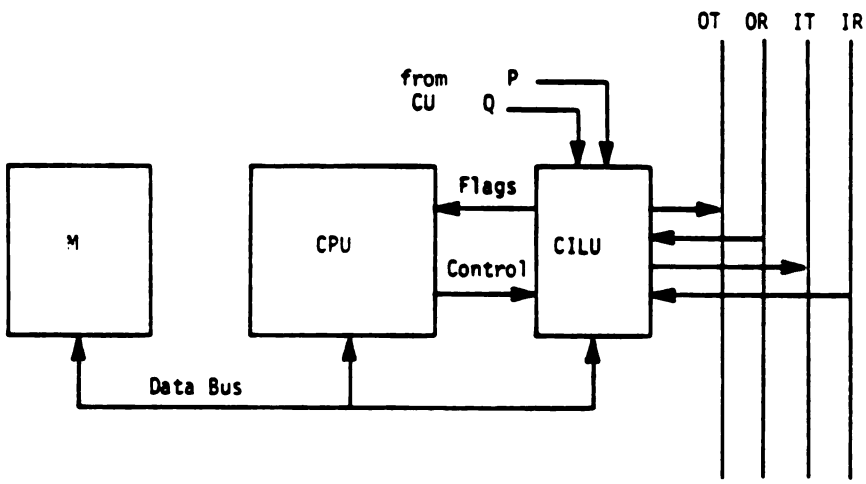


Fig. 3.2 Block diagram of 12x1 PE array and control unit.

convert the PEs into time-division multiplexed circuits for transmitting and receiving information packets consisting of flag bits, switching addresses, destination addresses, operand tags, operands and PE signaling information. All PEs operate independently, are identical (see Figure 3.3) and consist of a central processing unit (CPU), memory and a communications interface logic unit (CILU). The CPU has arithmetic and logic facilities to execute instructions stored in memory. However, these instructions are not enabled for execution until all data operands have been received from predecessor PE operations.

The communications interface logic unit (CILU) interfaces the CPU to the incoming and outgoing transmit and receive TDM buses, Figure 3.3. The incoming transmit and receive TDM buses are used for communication purposes in receiving packets from other PEs. The outgoing transmit and receive TDM buses are used for communication purposes in transmitting packets to other PEs. Both communication arrangements are illustrated in Figure 3.4. Incoming packets contain data needed for the next computation and outgoing packets contain results of a previous computation. Each CPU independently coordinates its use of these buses through its CILU. The CILU consists of an addressable information packet buffer connected to each TDM bus and control logic with control inputs from the CPU and timing signal inputs from the control unit. The CPU can read from buffers connected to receive TDM buses and can write into buffers connected to transmit TDM buses. The timing signals from the control unit are used to convert these buffers into time-division multiplexed circuits. As a time-division multiplexed circuit, each buffer is assigned a time slot in a frame that is regularly repeated. The contents of the transmit buffers are multiplexed onto its respective transmit TDM bus during its time slot and the contents on the receive TDM bus are demultiplexed into its respective receive buffer during its time slot. The time slots given to all buffers for a particular PE are the same. Since we have separate outgoing and incoming communication paths, PEs can simultaneously transmit and receive information packets during the same time slot.



OT - Outgoing transmit bus
OR - Outgoing receive bus
IT - Incoming transmit bus
IR - Incoming receive bus
CPU - Central processing unit
M - Memory
CILU - Communications interface logic unit

Fig. 3.3 Block diagram of a TDMP processing element.

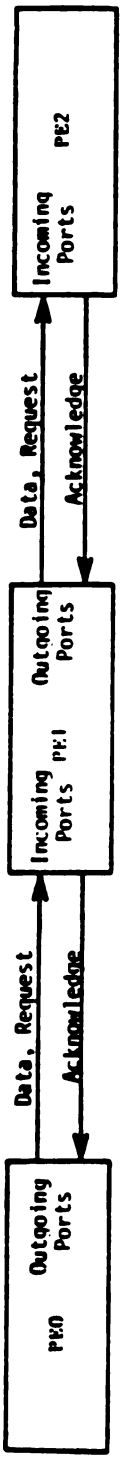
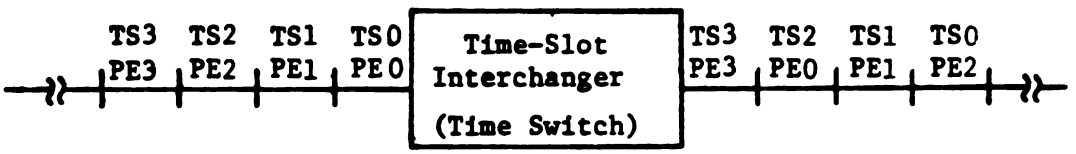


Figure 3.4 Example of incoming and outgoing PE communications paths.

3.1.2 **Time-division switching network.** The time-division switching (TDS) networks are used to permute the information packets among the PEs, Figure 3.1. There are two switching networks in TDMP, an outgoing TDS and an incoming TDS. The outgoing TDS has inputs from outgoing transmit TDM buses and has outputs connected to incoming receive TDM buses. The incoming TDS has inputs from the incoming transmit TDM buses and has outputs connected to outgoing receive TDM buses. Both TDS networks are identical and operate in exactly the same manner. To switch packets between PEs, the packets from individual time slots on one bus are placed in the same or different time slots on other buses. The interchanging of time slots is essential to time-division switching.[32] The operation of time slot interchanging is illustrated in Figure 3.5 for a single time-division multiplexed bus with four time slots. In this example, the input time slots 0 and 2 are interchanged; i.e., the contents on the input TDM bus during time slot 0 are placed on the output TDM bus during time slot 2 and vice versa. To switch information packets between buses, space-division switches are used to interconnect (permute) TDM buses for each time slot period. This is known as "time multiplex switching."[32] Figure 3.6 shows a TDS result in which both space and time switching occurs. Time slot 0 of TDM bus 0 is interchanged with time slot 2 of TDM bus 1 and vice versa.

Figure 3.7 shows a block diagram of a single input/output TDS network. The architecture of the switch is extremely simple, but surprisingly powerful and flexible. The switch consists of addressable input and output registers, dual central processing units (CPU_A and CPU_B). Each of these CPUs has its own program memory (PM_A and PM_B), read/write modules, and multiplexors/demultiplexors. The processors (called network processors) operate concurrently and perform all the switching functions according to a stored program in each PM. All data in read/write memory (RAM) and I/O registers except the input address register are words in a common data memory (DM) and the only operation of each processor is to move a word from some location in DM to another location in DM.[34] A separate DM is associated with each processor. A move operation takes exactly two memory cycles, the FROM address is read from



TSi = Time Slot i
(i=1,2,3,4)

NOTE: PE0 → PE2
PE2 → PE0

Fig. 3.5 Example of time slot interchanging in the TDS.

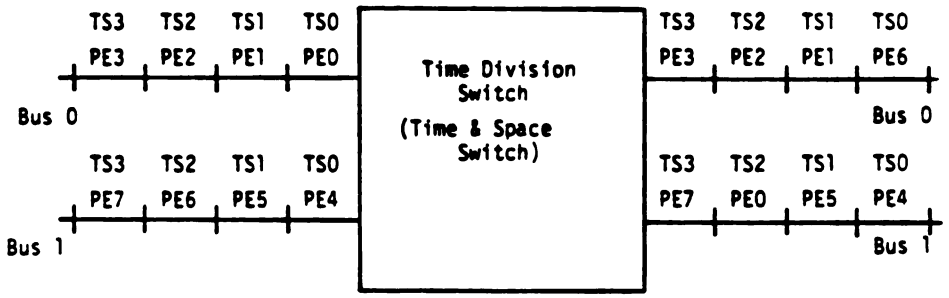


Fig. 3.6 Example of both space and time switching in the TDS.

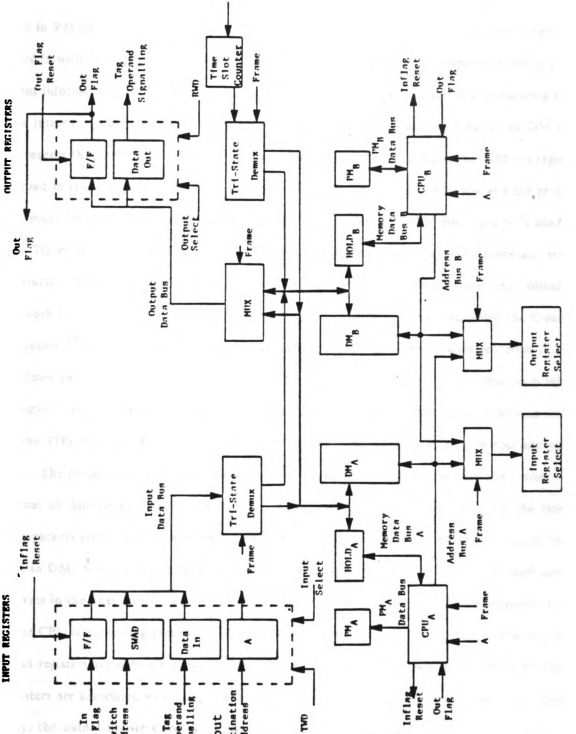


Fig. 3.7 Block diagram of a single input/output time-division switching network.

PM and is used to read at that address from DM, then a TO address is read from the next location in PM or from the input destination address register, A, which is the address where the word is written in DM. In the TDS network, this move operation corresponds to putting the input information packet data register contents into the CPU hold register and performing time slot interchanging on this data by storing the data packet in a memory location in DM that represents the same or different time slot position. When the input destination address register is used in the TO cycle of the move operation, the switching is called dynamic and the processor move instruction executed is called "indirect move." When the PM address data is used in the TO cycle of the move operation, the switching is called static and the processor move instruction executed is called "direct move." The operations and capabilities of the individual network processors of this type architecture are developed by Lipovski and called the C-move processor.[34] However, the network processor differs slightly from the C-move processor in that it allows an addressing mode for which the address in TO cycle of the move operation is taken directly from a special address register and not PM. This change enhances the switching speed of the TDS network. Each input packet is serviced sequentially according to its time slot position. The processor completes its service cycle for each packet during one time slot. After one frame, all data packets have been stored into DM. During the next frame, all of the stored data packets are moved out sequentially from RAM locations in DM to the output register location in DM. Since each processor must first store all packets during one frame then read out all packets in the next frame, two CPUs are needed so that when one CPU is inputting packets the other CPU is outputting packets. CPU_A uses DM_A for its input move operations; however, the input registers are only associated with DM_A when CPU_A is inputting packets, otherwise these registers are associated with DM_B which is used by CPU_B in its input move operations. Similarly, the output registers are only associated with CPU_A and DM_A when CPU_A is outputting packets, otherwise they are associated with CPU_B and DM_B when CPU_B is outputting packets. A multiplexor and a demultiplexor are used to connect the input and output registers to a DM

by using framing information to make the correct association. We have two frames called A and B. During frame A the input registers appear as memory locations for CPU_A in DM_A and the output registers appear as memory locations for CPU_B in DM_B. During frame B the output registers appear as memory locations for CPU_A in DM_A and the input registers appear as memory locations for CPU_B in DM_B.

The execution of the processor program in both CPUs is in synchronization with the input and output TDM information packet buses. The input is synchronized by using the input flag bit as a predicate for program execution, using the input switching address as the program counter for PM, and using time slot sizes greater than or equal to the time it takes the network processor to execute the instructions pointed to by this switching address. The output is synchronized by using the time slot counter value as the program counter where the instructions pointed to in PM sequentially output stored packets and flag bits.

3.1.3 **Control unit.** The control unit (CU) interfaces with I/O or a host processor, initializes the time-division switching networks, initializes the PEs, generates and sends timing pulses to PEs. The initialization steps are done prior to the beginning of program execution. Initialization of the time-division switching network consists of storing in program memory the program switching instructions associated with control and time slot interchanging and resetting the network processor to begin execution. The initialization of PEs consists of resetting the CPU and storing instructions, data constants, and initial values into each PE memory. The control unit has direct access to the network processor's program memories for program loading and program modification purposes. However, CU access to the PE memories is through the switching network. The CU has outgoing transmit and receive TDM buses and incoming transmit and receive TDM buses. These buses are connected to the TDS networks in the same fashion as the PEs, Figure 3.1. However, the CU has a multiple number of time slots per frame for communication purposes as compared to the PEs which have one time slot per frame. The CU can

use these time slots to send and receive information packets to and from individual PEs and to send control packets to the TDS network processor. The control packets modify the network processor program counter, which causes a jump to a subroutine that services that input control message.

3.2 TDMP Implementation

There are several important attributes of TDMP that make it well-suited for VLSI implementation. The first attribute is that the majority of the processors are identical. This produces a major benefit in decreasing the layout time and effort of the architecture. The decrease comes about because the use of identical structures reduces the total number of devices which must be individually drawn. In addition, the more structured layouts are easier to validate. A second attribute is that the processors and memory are in close proximity and can be implemented in the same technology. The locality of processing reduces the hardware and software communication overhead. A third attribute is that the underlying communication geometry, time-division multiplexed buses, is simple and regular which leads to high density and, more importantly, to modular design. In this way large TDMP systems can be developed as a collection of many simpler TDMP systems. A fourth attribute is that the switching network is regular and has full-access and non-blocking capabilities with simple and efficient control. The regularity in the switch comes about through the use of semiconductor memory for switching the data. This use is compatible with VLSI and future technologies, since these technologies will decrease both cost per bit and memory access time. The flexibility in the architecture brings about a universal appeal. As a result, VLSI cost advantages can be obtained by the wide variety of applications in which it can be used effectively. However, this is not to say that TDMP is suitable for all applications. The last attribute is that overall architecture of TDMP is highly compatible with algorithms that use a packet form of data movement such as data flow.[35] In data flow algorithms, there is no need to maintain long and continuous connections

between source and destination such as in block transfers of information. So in TDMP, processing elements send information packets to other processing elements only during designated time slot periods.

3.3 TDMP: A Data Flow Processor

TDMP executes programs expressed in data flow notation. These programs are normally described as program graphs which represent the data dependencies between operations. The attractiveness of such a system lies in the fact that it is data-driven; that is, each instruction is enabled for execution just when the required operand(s) has been supplied by the execution of a predecessor instruction(s). Since data flow instructions have no side effects, unrelated instructions can be executed concurrently without interference if each has its required operands. In this sense, the progress of a computation is determined by the passage of data through the system. Principal advantages of the TDMP data flow over conventional designs are reduced complexity of the processor interconnection network, greater use of pipelining, and a simpler representation and implementation of concurrent activity. To illustrate the basic concepts of TDMP data flow operation, consider the data flow program shown in Figure 3.8. This program represents the computation required for a second order recursive digital filter

$$Y(n) = AX(n) + BX(n-1) + CY(n-1) + DY(n-2)$$

where X(n) and Y(n) denote input and output samples for time nT, where T = 1. In this diagram, PE operators 2, 3, 4 and 5 are single-input operators that multiply by the fixed parameters A, B, C and D; PE operators 6, 7 and 8 are two-input operators that perform addition; and PE operator 9 is an identity operator that transmits its input values unchanged. Each small solid dot is a link that receives results from an operator and distributes them to other operators for use as operands. Input operator 1 represents the outgoing transmit control unit port through which an external stream of values that represent the input signal X(n) is presented to

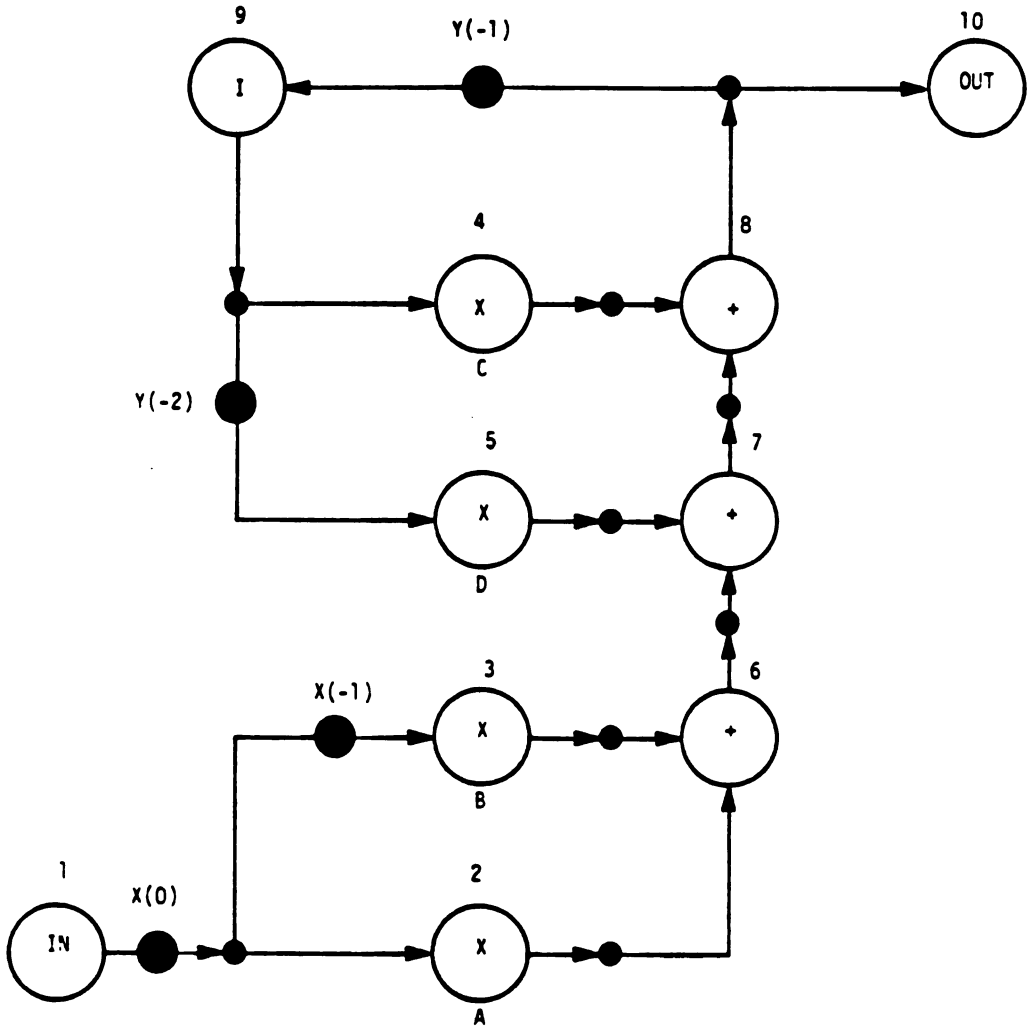


Fig. 3.8 Data flow program graph for a 2nd order difference equation.

the program. Similarly, output operator 10 represents the incoming receive control unit port at which the sequence of values representing Y(n) is delivered during program execution.

The large solid dots (tokens) show the presence of values at certain input arcs of operators and define the initial configuration for program execution. An operator with tokens on each of its input arcs and no tokens on its output arcs is enabled, and may fire by removing the tokens from its input arcs, computing a result using the values associated with the tokens, and associating the result with a token placed on the output arc of the operator. A link is enabled when a token is present on its input arc and no token is present on any of its output arcs. It fires by placing tokens on each of its output arcs and removing the tokens from its input arc. The new tokens distribute copies of the value associated with the input token over each output arc of the link. TDMP can perform conditional execution by using PEs to perform the primitive branch operation. The branch operator selects one or two output arcs (destination addresses) on which to place (send) its first input data, according to the state of a second Boolean input value. The two possible firing states lead to the execution sequences shown in Figure 3.9. The operator can be used to achieve conditional expression evaluation at a higher level. Figure 3.10 shows a natural translation of the high level conditional assignments:

$$\text{ABS: } = \text{If A} \geq 0 \text{ THEN A ELSE - A.}$$

Conditional flow graphs should be constructed with caution, since the absence of tokens flowing down some arcs might leave other tokens stranded at inputs to nodes. Conditional expressions, such as the one in Figure 3.10, are "safe" provided that both THEN and ELSE expressions are stated and are of the appropriate type. Conditional evaluation combined with cyclic or reentrant flow graphs proves TDMP extremely powerful. For example, an iterative or loop construct can be implemented by conditionally deciding whether to send tokens to the next block in a program, or to recycle them through the current block. However, problems can arise in using

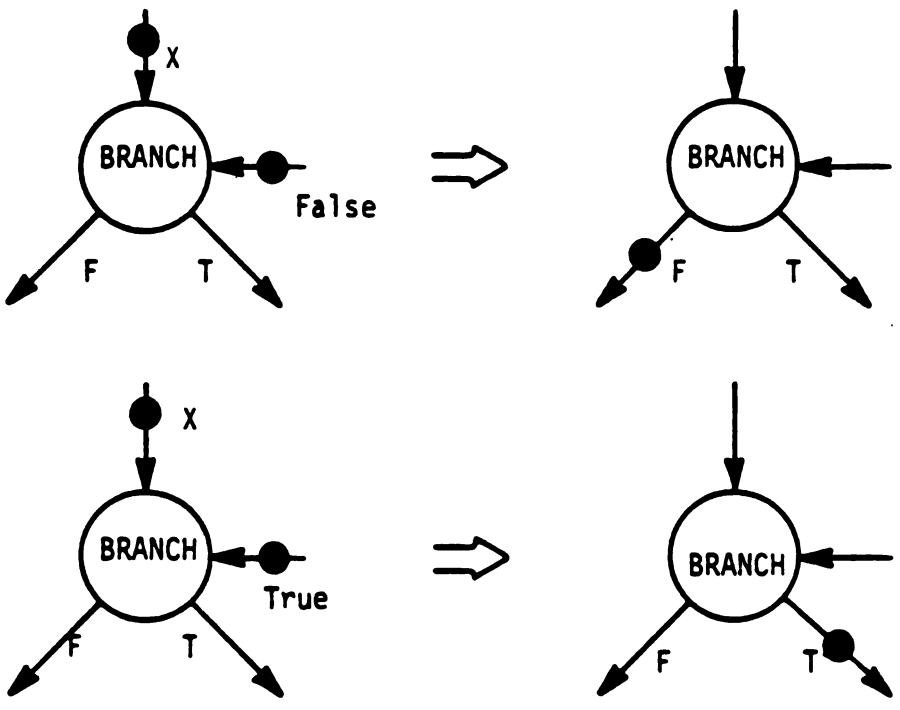


Fig. 3.9 Example of a data flow branch operation.

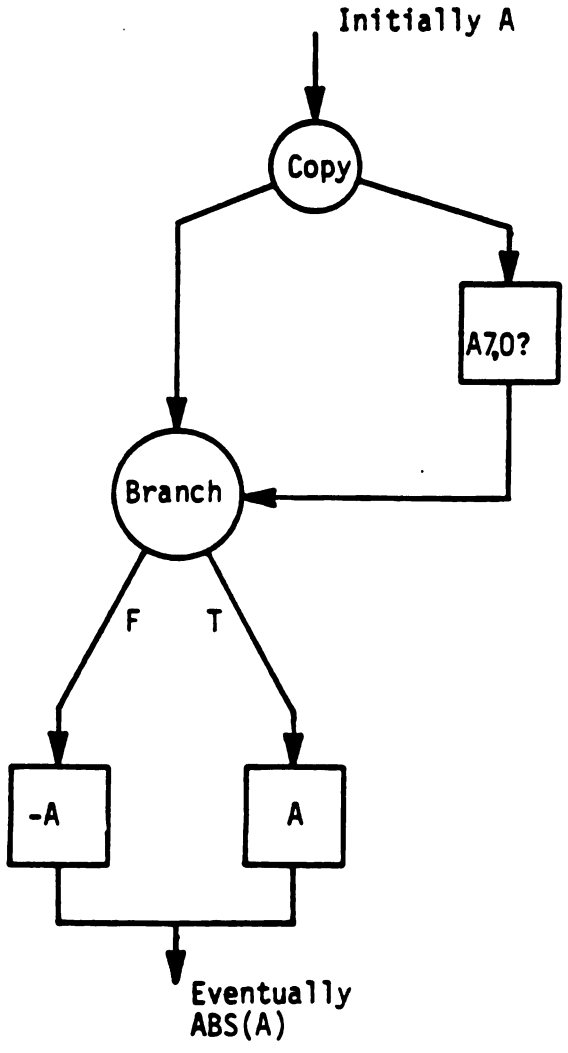


Fig. 3.10 Example of higher level conditional expression evaluation.

some reentry flow graphs because of blocking and token identification. Gurd and Watson discuss these problems and sight several examples to illustrate the issue.[36] Dennis first proposed a very basic version of a data flow language in which instruction execution was limited only by the data dependencies of the program.[37] Dennis and Misunas did preliminary work into the design of a computer based on this language.[35] Raumbaugh has expanded and improved the earlier version of the data flow language proposed by Dennis and has developed a multiprocessor architecture consisting of N identical activation processors.[38] Each processor is capable of executing in a pipeline manner several data flow instructions at a time.

The TDMP data flow processor is conceived as using each processing element as a combined operator and link, or just a link. The time-division switching networks in TDMP are then used as a means by which the link operations can be carried out. If a particular data item is to be used concurrently in more than one place or time in the system, then that data item must be explicitly sent to the multiple processing elements where it will be used. By using submultiplexing in both the PEs and switch, A frame and B frame, two items can be sent to different destinations. Submultiplexing is an important part of TDMP when transmitting to multiple locations in order to prevent two PEs from simultaneously transmitting to the same PE during the same frame.

Data items are transmitted in the form of an information packet. A PE information packet contains five fields; flag, switching address, destination address, tag, operand and signalling. The switching address and destination address are used by the switching network to direct the tag, operand, and signaling fields of each packet to the correct destination. The tag field is used by the PE to identify what operator in PE memory the data operand is associated with. By storing multiple operators in memory, task or program switching is performed more quickly. The number of multiple operators is limited by the PE memory size. The signaling field is used by the PE to transmit and receive request and acknowledge signals when exchanging information

packets between PEs. Once placed on a data path, packets remain there until destroyed as a consequence of being acknowledged by the destination PE. The packet is placed into a firing set of the PE using the tag field. Any PE can execute any machine language program providing that it has sufficient amount of storage. The machine language program defines the operation performed by the PE; e.g., arithmetic or logical type operations. If the incoming packet data does not make the receiving PE firable, then the packet data is stored in the appropriate place in the PE memory. This situation occurs when the received packet does not completely satisfy the needed operand requirements of the PE. For example, if the PE needs two operands to be firable and has previously received none, then the incoming packet will be stored until the second operand has been received. If the incoming packet data makes the PE firable then the PE fires immediately, using the packet data as needed. When the PE produces a result, it formats an outgoing information packet for each copy of the result that is needed.

CHAPTER IV

TIMING CONSIDERATIONS

In Chapter 3 the PEs and control unit were described as time-division multiplexed circuits when transmitting and receiving information packets. The timing relationships of the control signals necessary to convert the PEs and control unit into time-division multiplexed circuits were not discussed in Chapter 3. In Section 4.1 of this Chapter, we describe these timing relationships in detail with a timing diagram. In Section 4.2, we develop an expression that bounds the maximum multiplexing and demultiplexing rates achievable in TDMP. The maximum multiplexing and demultiplexing rates fix the minimum time slot size on the time-division multiplexed buses. The minimum time slot size in turn determines the logic one and logic zero time duration requirements of the timing control signals discussed in Section 4.1. The TDMP switching network has an inherent delay to affect switching of the information packets. In Section 4.3, we present an expression that bounds the maximum switching delay through the time-division switching (TDS) network in TDMP.

4.1 Timing Diagrams

The timing diagram of the control signals that convert the control unit and each PE into a time-division multiplexed circuit is shown in Figure 4.1. These control signals consist of P, Q, TWD, RWD, A and B frame. In this diagram 16 time slots are created (0-15) by using four Q signals (Q_0, Q_4, Q_8 and Q_{12}) and four P signals (P_0, P_1, P_2 and P_3). Each PE has one Q signal and one P signal connection. The control unit has one P signal and four Q signal connections. The simultaneous occurrence of a logic one on both the Q and P signals with logic one on TWD for transmitting and logic one on RWD for receiving select the control unit or a PE for multiplexing and demultiplexing information packets. TWD and RWD are clock signals used to multiplex and demultiplex the information packets to and from the time-division

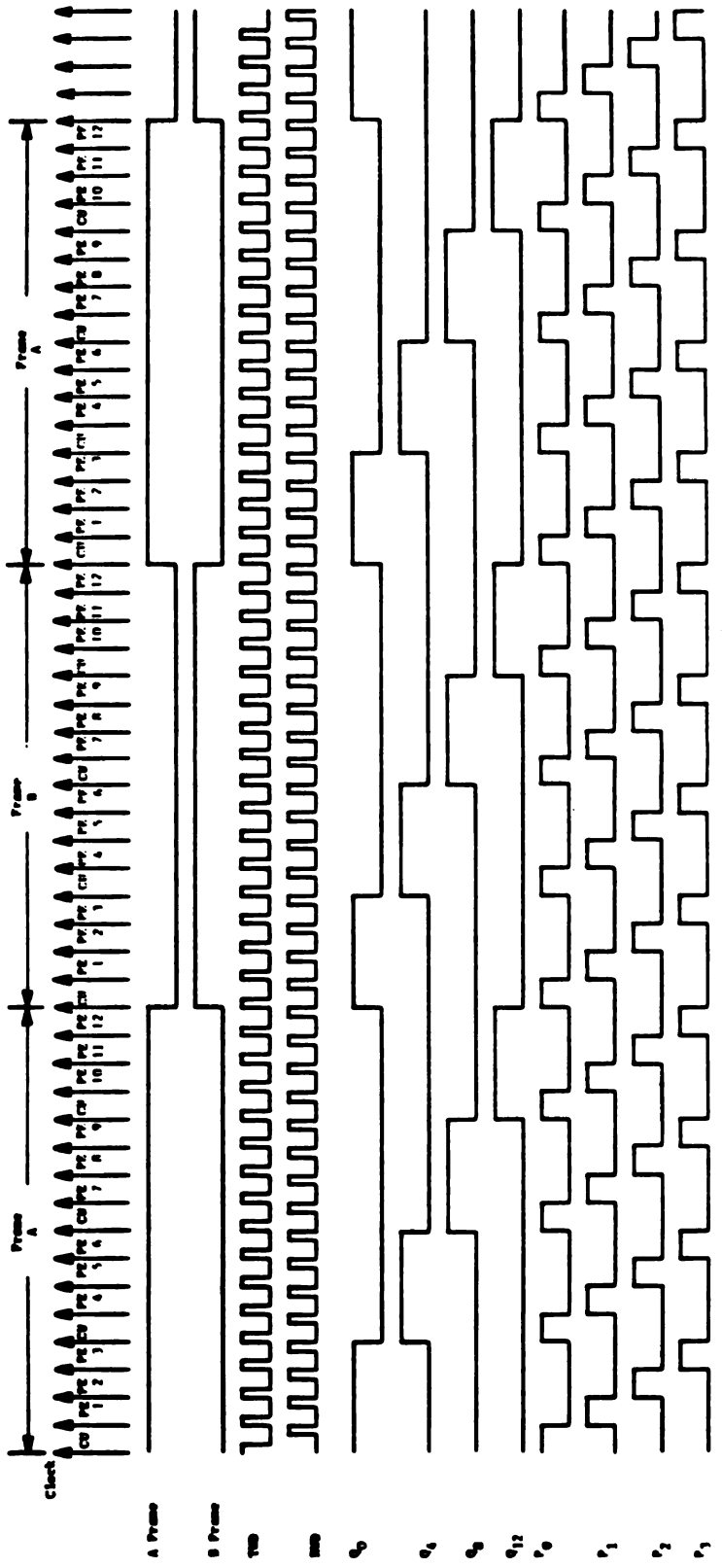


Fig. 4.1 Timing and control signals for a 16 time slot TDMP system.

multiplexed buses. The 4x4 matrix shown in Figure 4.2 is an alternative representation of the P and Q signals used for creating each time slot position. The numbers in each box represent the time slot position created. In addition the time slot position(s) for the control unit and each PE can easily be determined by simply adding together the subscripts on the P and Q signals for that PE or control unit. For example, P_3 and Q_4 generate time slot 7, 3 + 4 = 7. In TDMP time slots 0, 4, 8 and 12 are given to the control unit and the remaining time slots are used by the PEs. The length of time for which both Q and P are simultaneously logic one sets the time slot size. This time slot size is a function of the maximum multiplexing and demultiplexing rates and will be discussed Section 4.2. The A and B frame control signals are used for submultiplexing purposes. Separate registers are used to hold A and B frame information packets. When A frame is logic one only A frame information packets are multiplexed and demultiplexed and when B frame is logic one only B frame information packets are multiplexed and demultiplexed. However, A and B frame cannot simultaneously be logic one. More levels of submultiplexing can be employed in TDMP, however increasing the amount of submultiplexing also increases the transmission delay of individual packets within the system.

4.2 Maximum Multiplexing and Demultiplexing Rates

To determine the maximum TDMP multiplexing and demultiplexing rates we must start at the Time Division Switching (TDS) network. The TDS network works on a sampling basis. The transmitted information packets on the input of the TDS are sampled then stored in random access memory (RAM). The time between information packet samples is called the "sampling period" of the switch, T_s. The minimum sampling period is bounded below by the read/write cycle time, t_m, of the RAM plus the control overhead, t_c, for processing the information packet, $T_s \geq t_m + t_c$. The control overhead consists of the time for scanning the input for new samples, testing the frame condition, determining the type of service requested and reading the destination address. A flow chart for the TDS control algorithm is shown in

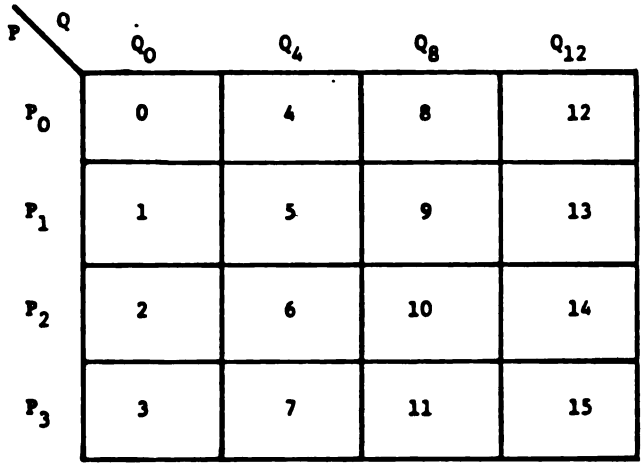


Fig. 4.2 Matrix representation of the P and Q control signals.

Note: The numbers in each box represent the time slot position created from the P and Q control signals.

Figure 4.3. The minimum read/write cycle time, t_m, is fixed by the current state of the art of semiconductor memories and as the technology improves t_m will decrease. The control overhead, t_c, will also decrease as technology improves, however further improvements can be obtained by maintaining precise synchronization of the TDS with the input multiplexed buses thereby eliminating the scanning of flag and frame bits used for synchronization. In addition, if we provide for only one type of switching service, then we can eliminate the control time needed to determine the type of switching service requested. In this investigation we will take the conservative path of keeping the extra control overhead since our primary purpose is to determine the feasibility of using time-division techniques to enhance computation and further enhancements can be made by the elimination of the extra control overhead. As a result, the switch sampling period puts a lower bound on the minimum PE time slot size $T_x \geq T_s$ where T_x is the time slot size and T_s is the switch sampling period. The minimum time slot size fixes the maximum multiplexing and demultiplexing rates in the system,

$$F_{\max} \leq \frac{1}{T_x(min)} = \frac{1}{t_m + t_c}$$

In our control algorithm for the time-division switching network used in TDMP $t_c = 8t_m$, Figure 4.3. Therefore in TDMP the maximum multiplexing rate is,

$$F_{\max} \leq \frac{1}{9t_m}$$

4.3 TDS Maximum Switching Delay

The maximum switching delay of the TDS in TDMP is a function of the time slot size, the number of time slots per frame, and the process of time slot interchanging. In TDMP 16 time slots make up one frame. Each PE is assigned one time slot and the control unit is assigned four time slots. These frames are regularly repeated and alternated via

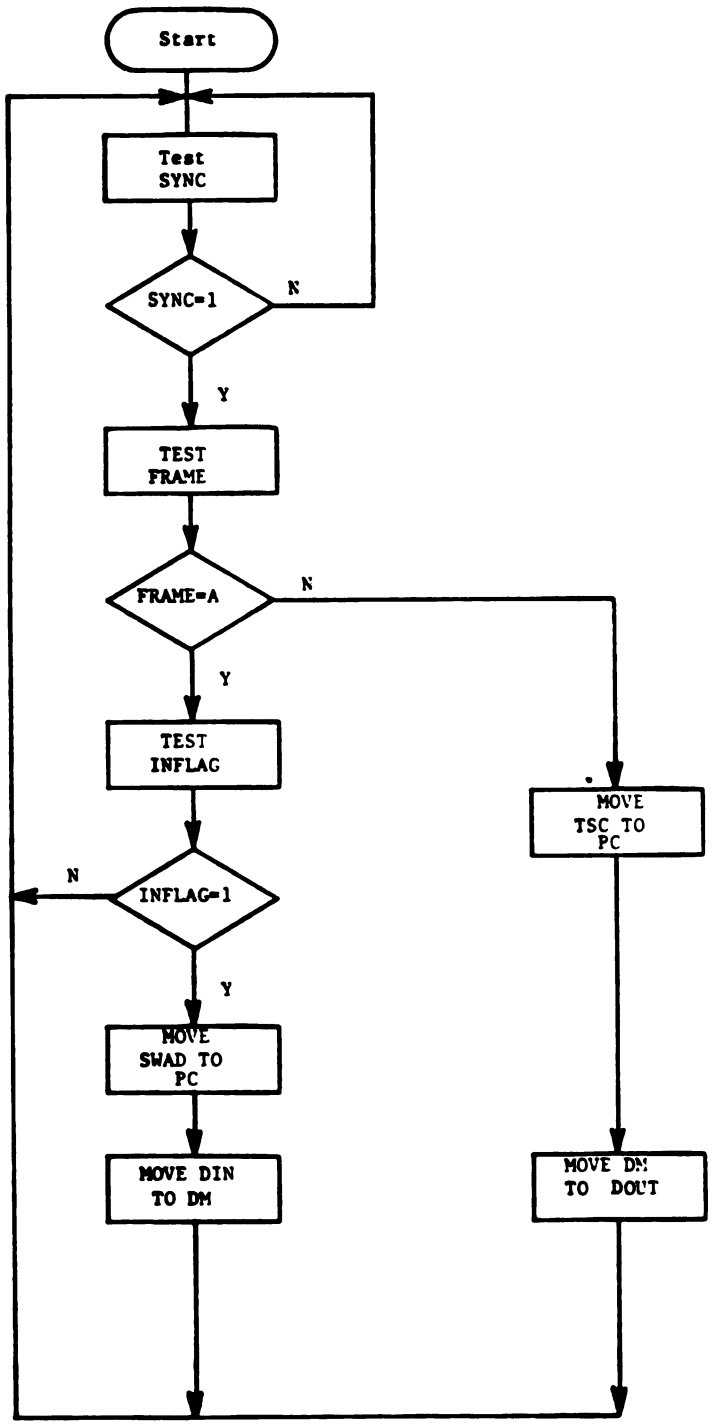


Fig. 4.3 Flowchart for TDS control algorithm.

submultiplexing between A and B frames. As described earlier, in Chapter 3, the TDS network switches information packets by providing for the association of input time slots with output time slots. All input information packets are processed sequentially and stored into RAM, then read out sequentially from RAM. This process of reordering the sequential input information packets by storing them into RAM locations specified by destination addresses and then reading out the RAM contents sequentially is called time slot interchanging. This technique implies a time delay to affect a change in time slot position. The delay is the amount of time it takes the TDS to store all the input information packets for one frame into RAM locations representing time slots positions plus the time it takes the TDS to sequentially output all these stored information packets. As a result, the maximum switching delay can be predicted by the following expression,

$$T_D = 2NT_X$$

where N is the number of time slots per frame, T_X is the time slot size, and the 2 is a result of the time slot interchanging process. In TDMP N = 16 and if we assume that the read/write cycle time of RAM is 30ns then $T_X = 270ns$ and the maximum switching delay in the TDS equals

$$T_D = (2)(16)(270\text{ns})$$

$$T_D = 8.64\ \mu s$$

The TDS switching delays results in a maximum packet switching delay of 17.28 μs. The packet switching delay is twice the TDS switching delay because of the A and B frame submultiplexing in which packets are transmitted every other frame. However, if the same information packets are sent during both A and B frames then, the maximum packet switching delay equals 8.64 μs. If the switching control overhead is reduced to zero the time slot size

decreases, $T_X = 30ns$, and the maximum switching delay becomes $T_D = .96\ \mu s$.

The maximum switching delay in the TDS network increases linearly with N.

$$T_D = A_1 N$$

where N is the number of time slots on the TDS input and output time-division multiplexed buses and A_1 is a weighting coefficient of the TDS network. The coefficient A_1 is determined from the switching implementation, control algorithm, and any other factors that might influence the switching speed of the TDS network. The other factors might include fault detection built with the hardware and diagnostic programs in software to improve reliability of the network. Many of the popular multistage interconnection networks such as binary n-cube, banyan and shuffle exchange have a switching delay which increases logarithmically with N,

$$T_D^* = A_2 \log_2 N$$

where N in these networks represent the number of input and output ports of the network and A_2 is its weighting coefficient.[40] The coefficient A_2 is determined from the same factors that are used to determine A_1 in the TDS network. In comparing the switching delay of interconnection networks the rate of increase, N or $\log_2 N$, is often used as a measure of performance. This is a useful measure when N is very large. However, for practical systems built with technology we can use in the immediate future, we must not only consider the rate of increase but also the weighting coefficients that multiply the rate of increase. If the weighting coefficients are not used in determining the maximum switching delay, then the conclusions drawn from such as analysis might be wrong. As an example, if we compare the maximum switching delay of TDS and the popular interconnection networks based only on the rate of increase, then we would conclude that the popular networks have a maximum switching delay less than that of the TDS network for all $N \geq 1$ since $\log_2 N < N$ for all $N \geq 1$. However, in Figure 4.4 we

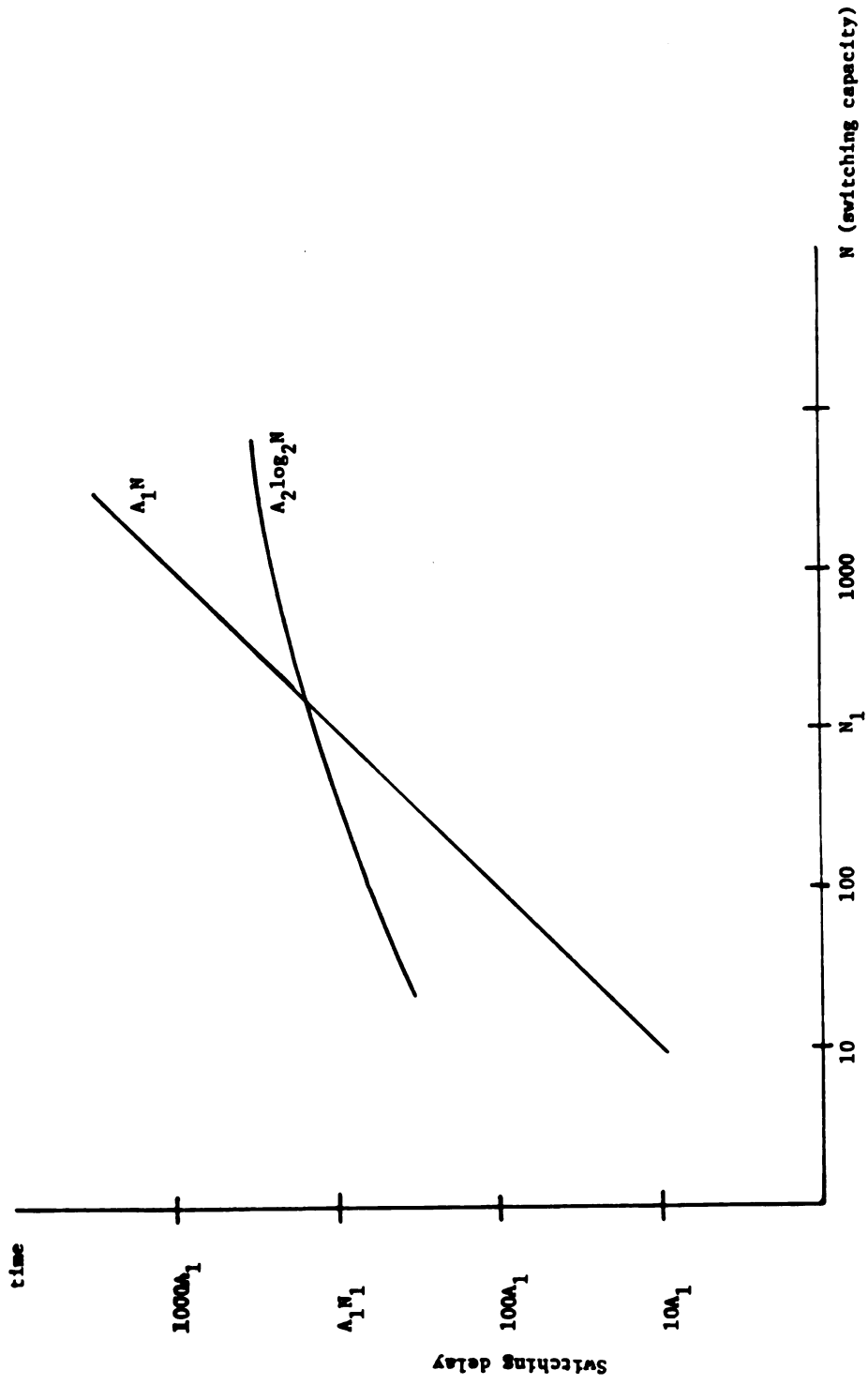


Fig. 4.4 An example to illustrate the A_1N versus $A_2\log_2 N$ switching delay issue.

show that the maximum switching delay of TDS is less than that of the popular networks for some values of $N \geq 1$, if $A_1 < A_2$. This example points out that the issue in comparing maximum switching delay is not just the rate of increase, N versus $log_2 N$, but is $A_1 N$ versus $A_2 log_2 N$. It has not been shown but is conceivable that $A_1 \ll A_2$ since the popular type interconnection networks, referred to as $log_2 N$, may have to make multiple passes through the network to reach the final destination and messages may have to wait as a result of blocking in the network. These factors influence the A_2 coefficient but not A_1. However, the applicability or superiority of one interconnection network over another should not be based only on switching delay. For example, unless efficient algorithms which use these networks and simple operating systems that manage their resources and supervise their processes are found, the discussion of switching delay is moot because the system as a whole will not be cost-effective and will experience considerable amounts of down-time due to software and hardware failures thus reducing the system throughput.

We feel that the TDS network offers some significant advantages over the $\log_2 N$ interconnection networks in algorithm mapping and operating system implementation. The mapping advantages results from the non-blocking and full-access interconnection capabilities of the network. Since all permutations are possible in TDS the programmer can construct or restructure algorithms without concern about interconnection capabilities of the network. And because the network is a C-move processor, many of the operating system functions can be directly implemented into the network, eliminating much of the time and hardware needed in supervising the switching process and in communicating and acting on exceptional conditions arising during switching.

The TDS switching delay is tolerable in TDMP system if we consider that while some PEs are transmitting and receiving information packets through the TDS network other PEs are performing computations. As a result throughput i.e., the quantity of useful information pro-

cessed by the system per unit time, is a better measure of performance for TDMP than is the switching delay of the TDS network in TDMP. And since TDMP is based on data flow principles and the TDS network has full-access and non-blocking interconnection capabilities the decomposition of programs is an easier task to perform in programming sufficient parallel activity into software to keep the parallel hardware fully occupied.

It was shown in this chapter that each PE only needs two signals, P and Q, to convert it into a time-division multiplexed circuit and that the minimum time slot size generated by these signals is determined by the maximum TDMP multiplexing and demultiplexing rates. The multiplexing and demultiplexing rates were found to be dependent upon the sampling period of the TDS network. We also showed that the TDS network has an inherent switching delay that increases linearly with the number of time slots, N. We presented arguments as to why the rate of increase of switching delay in TDS and the popular $\log_2 N$ interconnection networks was not sufficient to determine the superority of the $\log_2 N$ networks over the TDS network. In the next chapter, we describe a simulation model of the TDMP architecture.

CHAPTER V

TDMP SIMULATION MODEL

A simulation approach was chosen to analyze TDMP because it provides a timely and economical means of evaluating preliminary designs. Moreover, the versatility of this approach means that a wide range of alternative designs can be realistically analyzed. The use of a simulation model in our investigation has a two-fold purpose. The first is to analyze TDMP's system performance, e.g., the switching capabilities and switching delay, the communication protocols between processing elements, and the ability to execute a single data flow program at a time. The second purpose is to provide a detailed language description of the TDMP hardware that complements the less detailed block diagrams describing this hardware. As a result, the simulation language in our investigation has the following characteristics:

— capability to simulate concurrent activity;

— capability to simulate timing;

— flexibility to simulate hardware at both the gate-level and functional-level;

— easily translates hardware representation into a computer program.

Several simulation languages were available to us to choose from for our simulation model. These languages were SPICE, ECAP, GPSS, and APL. SPICE and ECAP are circuit level simulation languages and are extremely useful in modeling the dc, ac and transient effects in circuits.[41] However, these languages are too low-level for our current simulation purposes because they model hardware at the transistor-level. These languages would be more useful to us when we have verified the correct operation of the system at the logic gate-level and are ready to test the design performance at the transistor-level for integrated circuit implementation.

GPSS is a simulation programming language used to build computer models for discrete-event simulations.[42,43] GPSS is a process-oriented language and is particularly suited to the translation from a flow-chart representation of a system into a computer program. This is a disadvantage for our simulation purposes because the computer program representation that results is too high-level. We have a hardware representation of our system and would like the computer program model to complement the less detailed block diagrams describing the hardware as well as to simulate the processes in the system. GPSS would be more useful in simulating the protocols in TDMP and not the hardware design.

APL, however, meets most of our desired simulation language needs.[44,45,46] APL has the ability to perform both functional and gate-level simulation and the model implemented in this language neatly complements the less detailed block diagrams describing the hardware. The ability to perform both functional and gate-level simulations gives us the diversity to synthesize some parts of the model at a high-level using functional modules and some parts at a low-level using gates. Functional modules in the TDMP model tend to be those parts of the architecture that are not germain to the TDMP operation, e.g., arithmetic logic units, registers, memory storage units, etc. The gate-level models tend to be those parts of the TDMP architecture that are unique to its operation, e.g, control units, interface logic units, multiplexors, demultiplexors, etc.

All of these simulation languages have short coming when they must simulate concurrent processing. These languages are based on sequential computation with a single processor, as a result they cannot at all or with limited success model asynchronous events or concurrent operations. In addition all of these languages are very costly and require fast, high capacity computers. Given the advantages and disadvantages of each simulation language we chose APL as the language in which to build our simulation model.

This chapter begins with a description of the APL model of the time-division switching net-

work, Section 5.1. Then in Section 5.2, the APL simulation model of the processing element is described. And, finally, in Section 5.3, the operation of the complete APL simulation model of TDMP is discussed.

5.1 TDS Simulation Model

The block diagram of the data paths for the time-division switching (TDS) network simulation model is shown in Figure 5.1. There are two identical sets of data paths in the TDS network, one for switching A frame information packets and one for switching B frame information packets. As a result, only one set of these data paths are described in presenting the TDS network simulation model. The bused data paths in the model consist of DBUS, OBUS, IBUS, INBUS and OUTBUS. DBUS is the network processor data bus. All data moved to or from memory, registers and hardware operational units is placed on DBUS. OBUS and IBUS are internal use buses of the network processor. These buses connect the inputs and outputs of the network processor internal use registers to DBUS. INBUS and OUTBUS respectively connect the switch input and output port registers to DBUS. Many of the remaining data paths consist of gated input and output connections to these buses from registers, memories, and hardware operational units such as an adder, decrementer and incrementer. By gated connection we mean that a component must be selected in order to put data on a bus or take data off a bus. The gates are represented by the small circles on the data paths shown in Figure 5.1.

Specifically, INBUS has gated connections to the input and output of the input flag (INF) register, the outputs of the switching address (SWAD) register and input data (DIN) register. The INF register signals a request for service to the switch during an input packet operation. While the SWAD and DIN registers respectively contain the address of the type of service requested and the input packet data to be switched. The OUTBUS has gated connections to the input and output of the output flag (OUTF) register, the input of the data out (DOUT) register and the output of the time slot counter (TSC) register. The OUTF register signals the switch

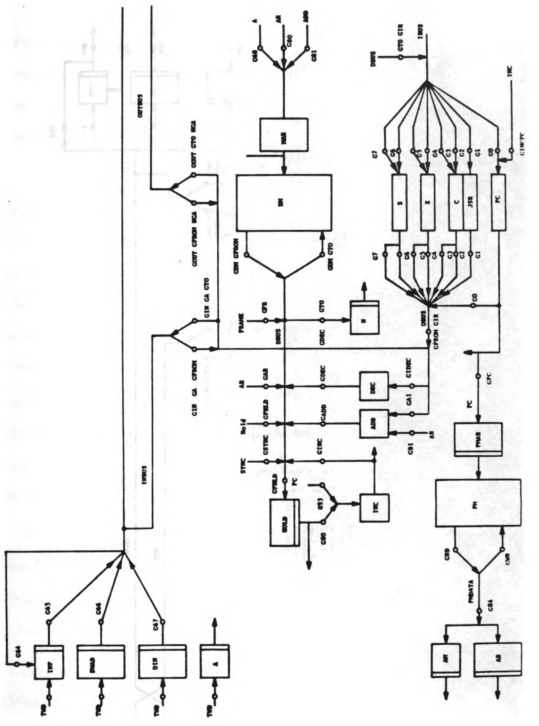


Fig. 5.1a Block diagram of the TDS APL simulation model.

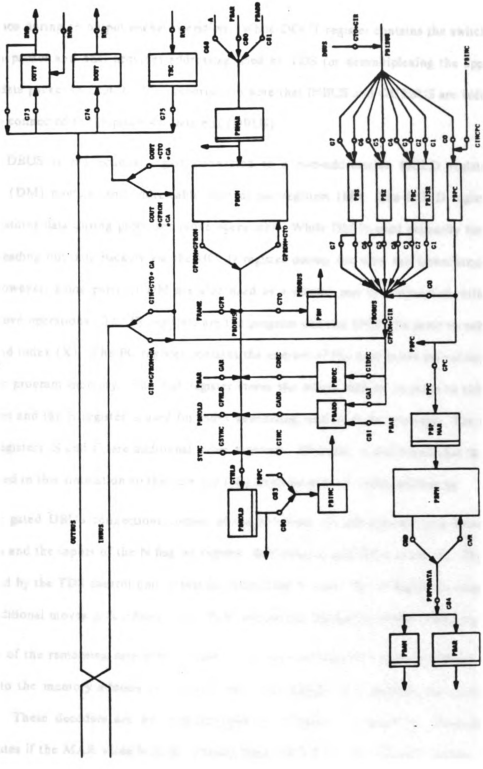


Fig. 5.1b Part of TDS APL simulation model.

for service during an output packet operation. While DOUT register contains the switched output data packet and TSC provides addressing used by TDS for demultiplexing the appropriate output data packet to DOUT. It is important to note that INBUS and OUTBUS are bidirectionally gate connected to the processor data bus (DBUS).

The DBUS is bidirectionally gate connected to a non-addressable HOLD register, data memory (DM) module, and addressable internal use registers (IR). The HOLD register temporarily stores data during processor move operations. While DM is used primarily for writing in and reading out data packets via the HOLD register during the time slot interchanging process. However, some parts of DM are also used as a scratch pad in conjunction with conditional move operations. The IR registers are the program counter (PC), the jump to subroutine (JSR) and index (X). The PC register contains the address of the next move instruction in the processor program memory. The JSR register stores the return address in jump to subroutine operations and the X register is used for index addressing, and up-down counting. The remaining IR registers, S and C, are additional index registers. However, it was found that they were not needed in this simulation so they are not fully implemented for index addressing.

Other gated DBUS connections consist of outputs from the decrementer and incrementer operators and the inputs of the N flag bit register, decrementer and adder operators. The N flag bit is used by the TDS control unit to test for conditional moves. The N flag bit in conjunction with conditional moves is described in the TDS control unit simulation model in Section 5.1.1.

Some of the remaining data paths consist of connections from the memory address register (MAR) to the memory address decoders of the input, output, data memory and internal-use registers. These decoders are not explicitly shown in Figure 5.1. However, these decoders enable gates if the MAR value is in the address space defined for that decoder. In this simulation the internal registers are given memory address locations 0-7, data memory has locations 8-63, the input registers have 64-71 and the output registers have 72-79, Figure 5.2. Gate

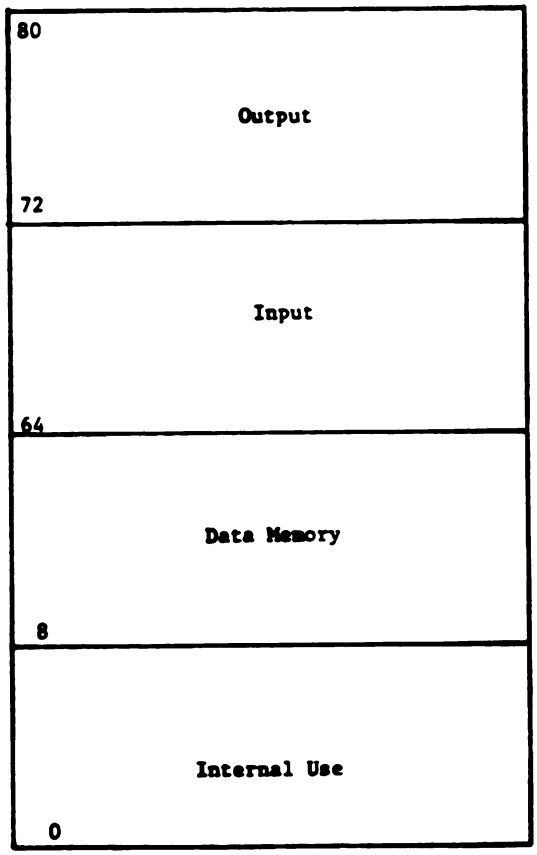


Figure 5.2 TDS simulation model memory map.

signals GIR, GDM, GIN, and GOUT respectively gate the internal registers, data memory, input registers, and output registers data onto and from the DBUS. The remaining decoder signals select specific registers within these groups.

The MAR has gated input data path connections from the destination address register (A), program memory buffer register (AR) and the output of the adder (ADD). Gating A into the MAR is a result of an indirect move, gating AR into MAR is a result of a direct move and gating ADD into MAR is a result of an indexed move.

The few remaining data paths consist of gated connections from the HOLD register to the incrementer input and program counter register connections to the incrementer input and output for incrementing the PC. And finally, we have the data path gated connections from the program counter to the program memory address register (PMAR) and bidirectional program memory data bus (PMDATA) with gated connections from the program memory (PM) to the program memory buffer registers (AM and AR). The program memory contains the TDS switching program consisting of only move instructions. A move instruction consists of an addressing mode and address. The AM register holds the addressing mode value in the From and TO cycles of a move instruction. In the FROM cycle these addressing modes are immediate, direct and indexed. In the TO cycle, these modes are direct, indexed, conditional and indirect. The AR register holds the address value in the FROM and TO cycles or in the case of immediate move the value of the data to be moved. As a result, the AR register has a gated connection to the DBUS for sending data to the HOLD register. Both AM and AR are inputs to the TDS control unit which performs the address evaluation and generates the timing and control signals necessary for all move operations.

5.1.1 TDS Control Unit

Figure 5.3 shows a block diagram of the TDS control unit. The control unit consists of a combinational logic network and memory. The inputs to the control unit are: the least

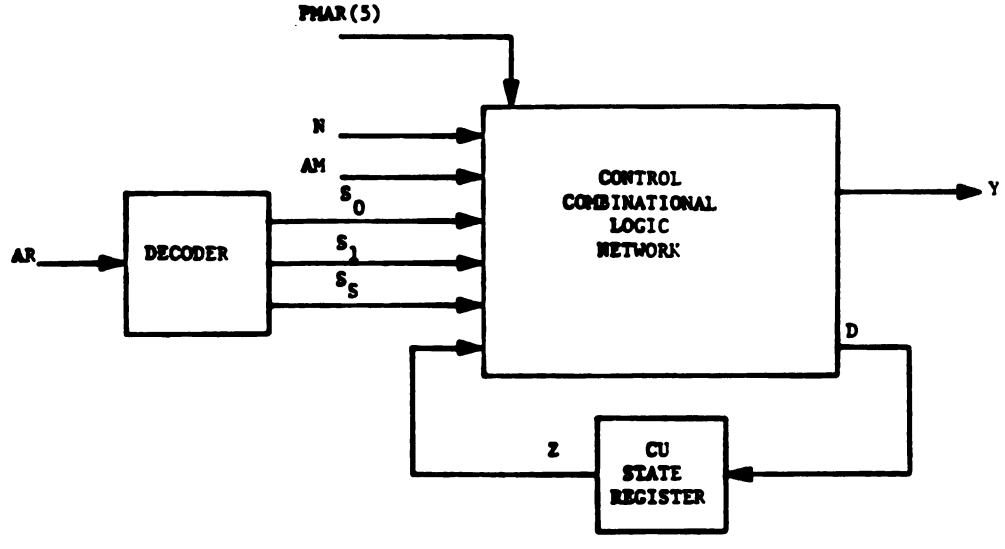


Figure 5.3 Block diagram of the TDS control unit.

significant bit of the program memory address register (PMAR(5)), the addressing mode bits (AM), the N flag bit, the decoder outputs (S_o, S_1, S_5) and the control unit state register output, (Z_o, Z_1, Z_2). The control unit outputs consist of control signals (Y), and the control signals D_o, D_1, D_2. The Y control signals gate in both space and time data paths in the TDS network. The D control signals generate the next state of the control unit. The least significant bit of the program memory address register, PMAR(5), determines the cycle in the move operation. If this bit is logic 1 the move cycle is identified as FROM and if this bit is logic 0 the move cycle is identified as TO. The AM bits are interpreted by the control unit so that the appropriate gate signals are generated for the type of addressing that is required. For example, in an indexed move the control unit must also generate the control signals that gate inputs to the adder from DBUS and the output from the adder to the MAR as well as the normal read/write memory control signals. In Figure 5.4, Table I shows the addressing modes as a function of the AM bits and move cycle. The N bit is used as a test signal in conditional move operations and is the most significant bit of each word moved. This bit is gated into the N bit register off the DBUS during the TO cycle of every move operation. In a conditional move, if the N bit is logic 1 the moved data is written into a memory location and if the N bit is logic 0 the moved data is not written into memory. In both cases the FROM cycle of the move operation is performed, however in the TO cycle the N bit condition is tested. The S_o, S_1 and S_5 inputs to the control unit identify internal use registers that are involved with move operations such as increment, decrement and jump to subroutine.

The network specification table shown in Figure 5.5 gives a detailed logic description of the properties of the control network. It shows for all the inputs to the control unit all the space and time outputs the control unit produces. For example, if the control unit is in state 0, $Z_0 = Z_1 = Z_2 = 0$, the contents of the program counter, PC, is gated into the program memory address register, PMAR. The clock advances the control unit to state 1, $Z_1 = Z_1 = 0$, $Z_2 = 1$. During state 1 the mode and address contents of the program memory are placed in the AM

	PMAR(5)	AM 000	AM 001	AM 010	AM 011	AM 100
From	0	Immediate	Direct	Index x	Unused	Unused
To	1	Unused	Direct	Index x	Conditional	Indirect

Fig. 5.4 Addressing modes for the TDS network C-move processor.

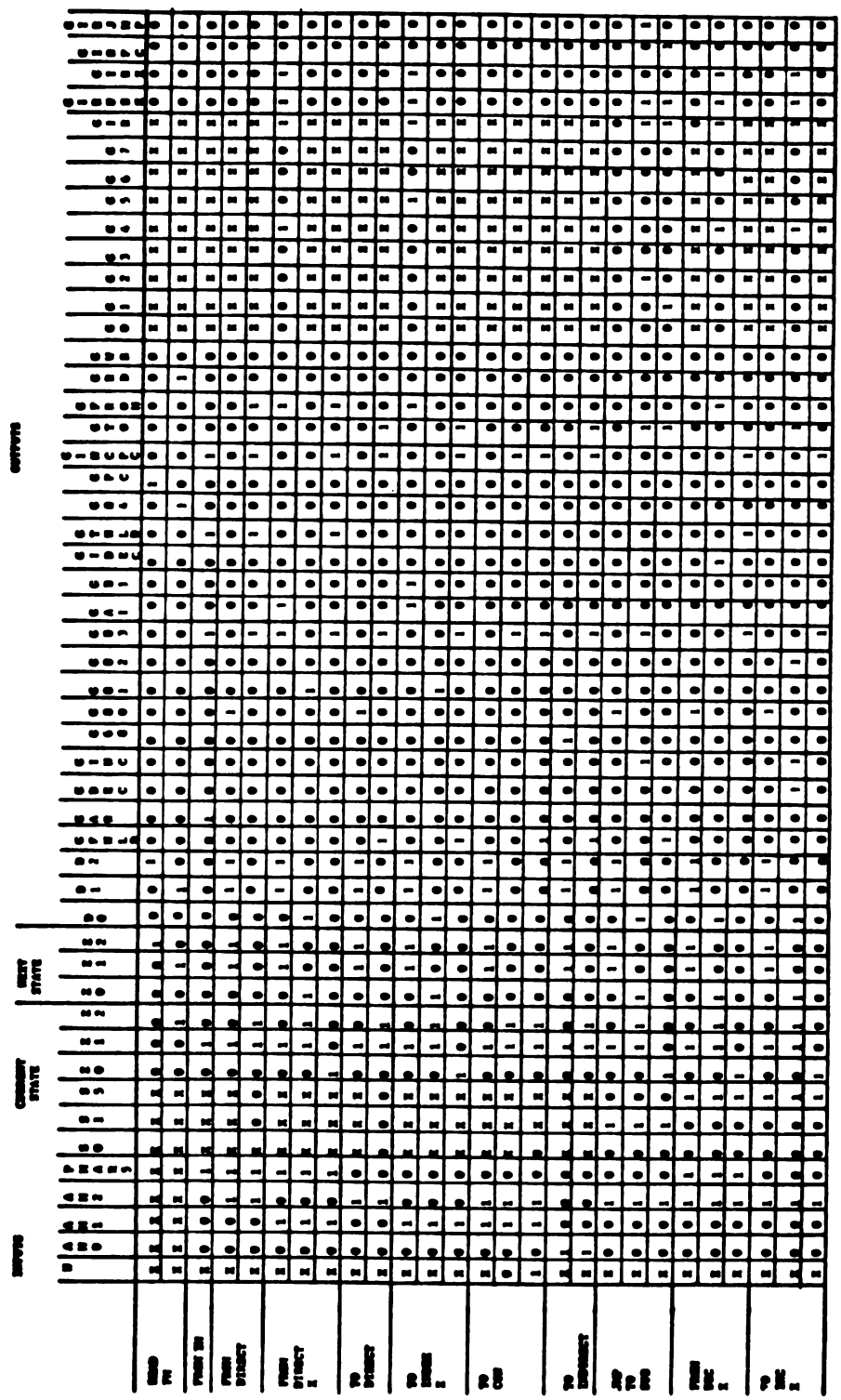


Fig. 5.5 Control unit logic network specification table.

and AR registers and based on the cycle and addressing mode the move operation is executed. If we assume that the addressing mode is direct (AM_0=AM_1=0, AM_2=1) for both cycles and we are in the FROM cycle, PMAR(5)=1. The next clock pulse advances the control unit to state 2 (Z_0=0, Z_1=1, Z_2=0) where it gates the FROM address in the AR register into the MAR register. The next clock pulse advances the control unit to state 3 (Z_0=0,Z_1=Z_2=1) where it gates the contents from the address memory location specified by the MAR to the HOLD register and also gates the data paths necessary to increment the program counter. The next clock pulse advances the control unit back to state 0, where the next three clock pulses cause the next program memory word to be fetched and the MAR to be loaded. However, when the clock advances the control unit to state 3 the cycle will be TO, PMAR(5)=0, so the control unit gates the contents from the HOLD register to the memory address location specified by the MAR and increments the program counter. Figure 5.6 gives examples of all move operations possible with the simulated TDS network C-move processor. In these examples assume initially that memory address location 15 has the value 20 and memory address location 17 has the value 30.

5.2 PE Simulation Model

The block diagram of the data paths for the PE simulation model is shown in Figure 5.7. The data paths consist of DBUS, registers, communication interface logic unit (CILU), arithmetic logic unit (ALU), program memory, data memory and a C-move processor. DBUS is the PE data bus. All data moved to or from data memory, registers, CILU and ALU is placed on DBUS. The registers are partitioned into groups called an outgoing communication port and an incoming communication port. These ports are identical but are used for different purposes. The outgoing port transmits request signals and data and receives acknowledge signals. The incoming communication port receives request signals and data and transmits acknowledge signals. Since both ports have the same hardware for communication, only the

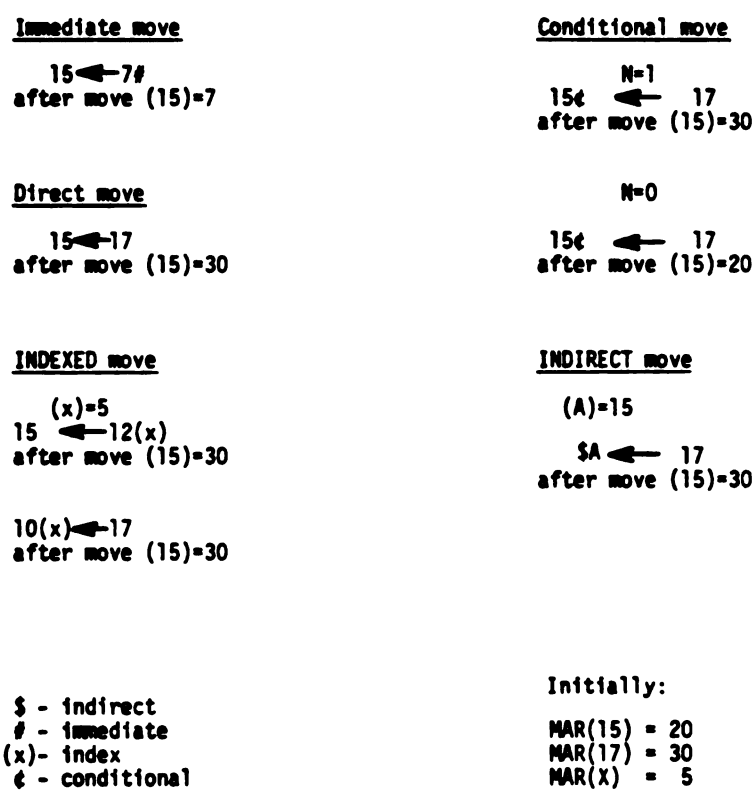


Fig. 5.6 Examples of TDS processor move operations.

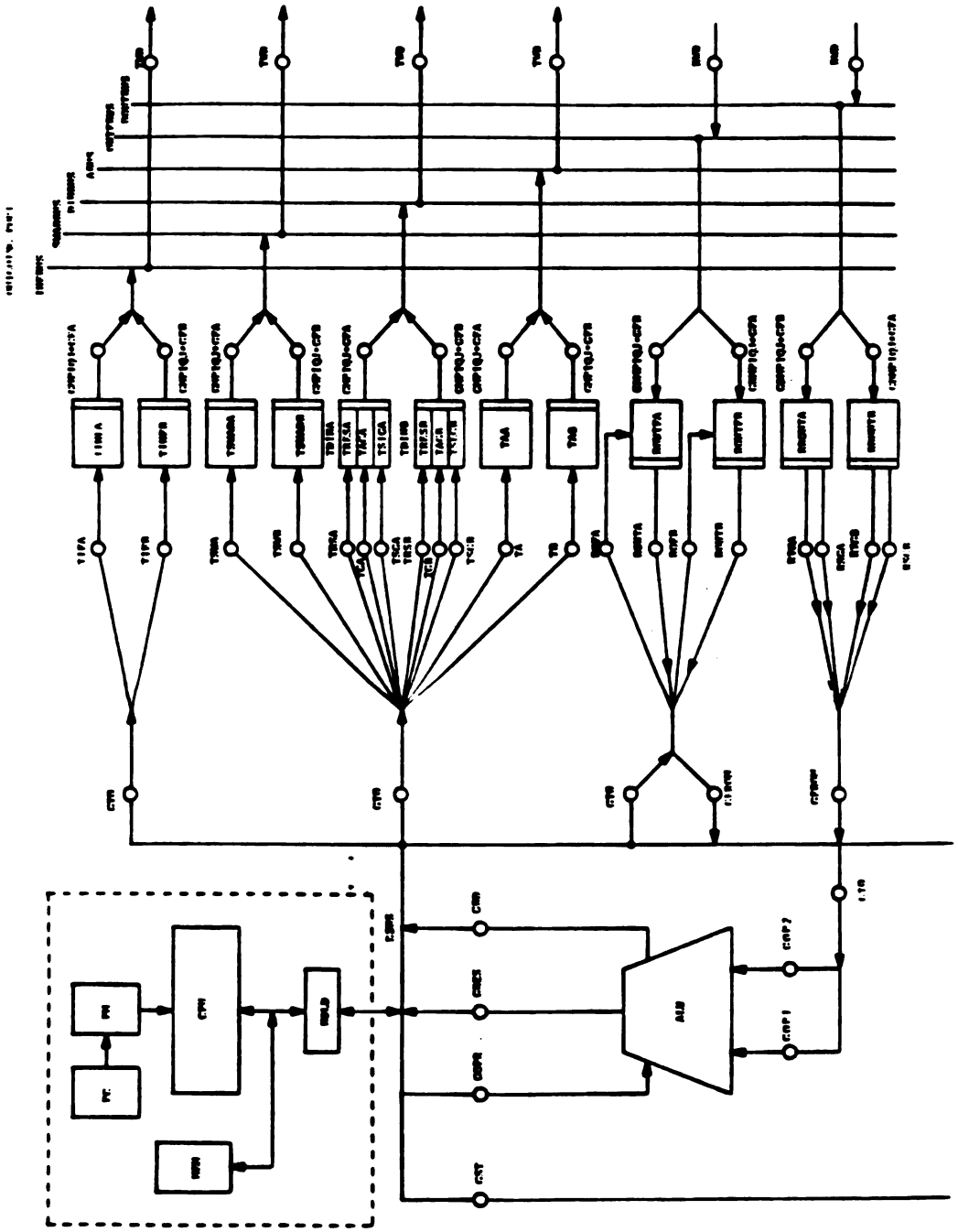


Fig. 5.7a Block diagram of the PE APL simulation model.

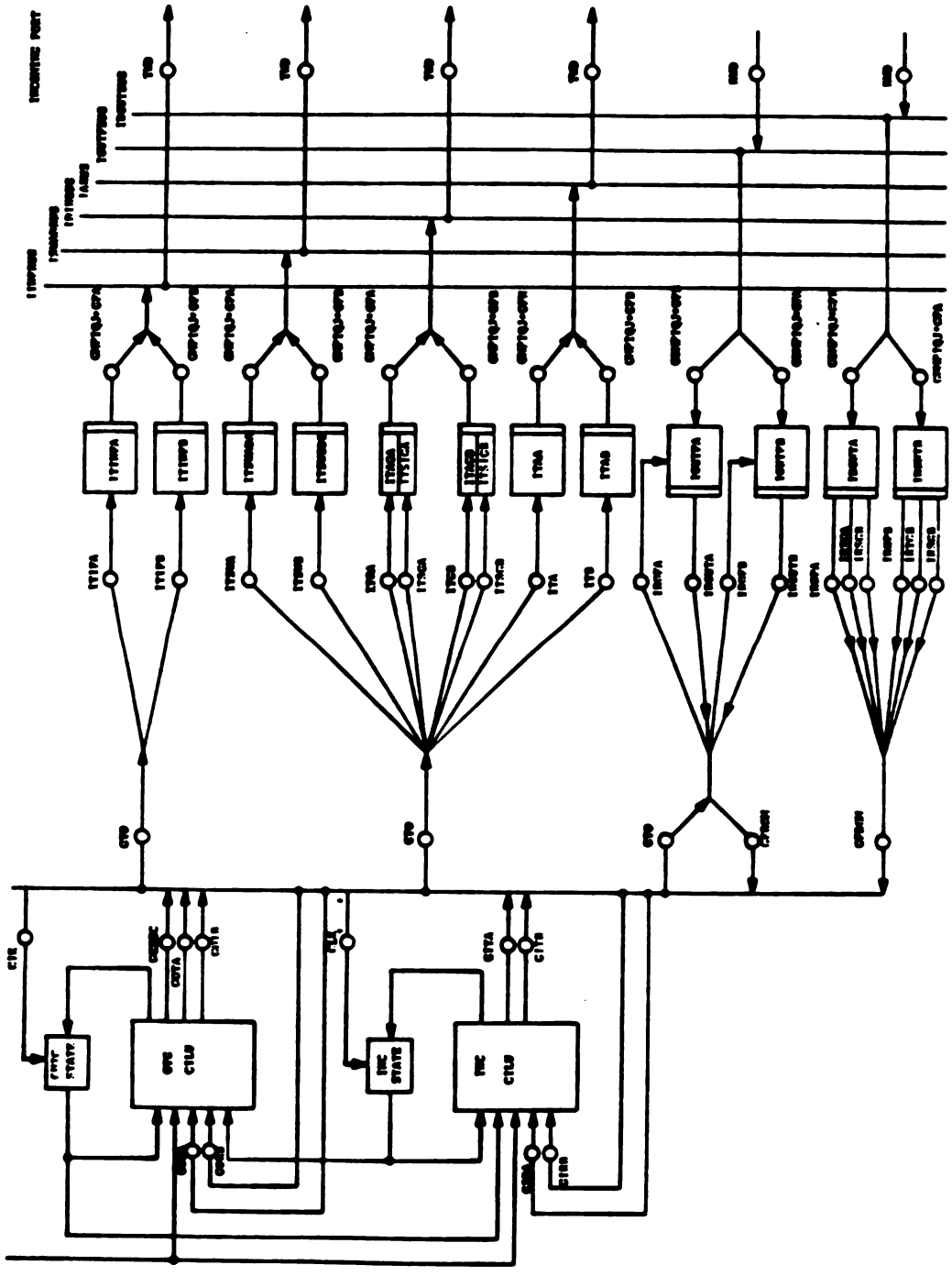


Fig. 5.7b Part of the PE APL simulation model.

outgoing port will be described.

The registers that make up the transmitting section of the outgoing communication port are TINFA, TINFB, TSWADA, TSWADB, DINA, DINB, TAA, and TAB. All of these registers are gate connected to DBUS on their input side and gate connected by multiplexor gates to their respective time division multiplex bus on their output side, Figure 5.7. TINFA and TINFB are respectively the A and B frame transmit flag bit registers. These registers signal requests for service to the switch during the A and B frame PE time slots. The request for service is basically an enable signal that informs the switch if it should perform or not perform a switching operation on its current input data. Since each PE has its own unique time-slot to seize the switch there is no chance of collision between PEs requesting for switching service. This helps to improve the throughput of the system. TSWADA and TSWADB are the A and B frame switch address registers. These registers are initialized by the PE with an address value of a location in the switch program memory. This location in the program memory contains move instructions that perform the switching operations on the switch input data. In our investigation the switch operations will be limited to moving the input data directly and indirectly to the switch data memory module. Direct and indirect move operations were described earlier in Chapter 3. DINA and DINB are respectively the A and B frame input data registers. The input data registers contain the data that is to be switched. Each of these registers is composed of three registers with separate gated inputs and a common gated output. The registers are labeled TRES, TAG, and TSIG with A and B subscripts added to each name to signify their association with either A or B frame. The TRES register contains the result of a previous PE computation. The TAG register contains a value used to identify which operation this result is to be used with in the next step of computation. The use of a TAG is required when the PEs are programmed to perform multiple operations or operations in which data values are reentered into the data flow program. In this simulation each PE is programmed for a single operation and data flow programs that use feedback are not simulated. As a result, the TAG bit is shown but

is not actually used in this simulation. TSIG is the signalling bit register used to transmit the idle and request PE signalling states. Transmit registers TAA and TAB are respectively the A and B frame data packet destination address registers. These registers contain an address representing the same or different time slot position of the PE. The switch uses this address to carry out the time slot interchanging operation on the input packet data.

The receiving section of the outgoing port consist of OUTFA, OUTFB, DOUTA and DOUTB. OUTFA and OUTFB are respectively the A and B frame receive flag bit registers. These registers are sensed by the PE to determine if new receive data is available. For the outgoing port the receive data consists of tag and PE signalling. For the incoming port the receive data consists of operands, tag and PE signalling. The receive data is contained in the DOUTA and DOUTB registers, respectively, for A and B frame data. The receive data registers have a single gated input, whereas the output has a separate gated connection to DBUS for each field in the register, Figure 5.7.

The CILU data paths generate the outgoing and incoming PE transmit signalling bits as well as the PE execution enabled signal. The inputs to the CILU consist of the outgoing and incoming receive signalling bits, the ALU status bit and the current state of the PE. The receive signalling bits were discussed earlier in this section, the ALU status bit, PST, is either in the idle or busy condition, the PE state information is stored in a register and new state values are generated based on the ALU status, the receive signalling conditions and the previous PE state value. However, no state changes can occur until the CILU has received a clock pulse generated by the PE control program. One clock pulse is generated each time the PE cycles through its control program. The various PE states and explanation of these states is given in Figure 5.8.

The CILU is modeled at the logic gate-level. Figures 5.9 and 5.10 show the truth table representation of the CILU design. The CILU was decomposed into an outgoing and incoming

PE State				
Outgoing		Incoming		Explanation
POS0	POS1	PIS2	PIS3	PE State
0	0	0	0	idle
0	0	1	0	received operand A
0	0	0	1	received operand B
0	0	1	1	enabled
1	1	0	0	results available
1	1	1	0	results available and received operand A
1	1	0	1	results available and received operand B
1	1	1	1	results available and received operands A & B
0	1	0	0	A frame results available
1	0	0	0	B frame results available
0	1	1	0	A frame results available and received operand A
0	1	0	1	A frame results available and received operand B
0	1	1	1	A frame results available and received operands A and B
1	0	1	0	B frame results available and received operand A
1	0	0	1	B frame results available and received operand B
1	0	1	1	B frame results available and received operands A and B

Fig. 5.8 PE states for data flow computing.

INPUTS						OUTPUTS				
Outgoing Current State		Outgoing Receive Signalling		ALV Status		Next State		Outgoing Transmit Signalling		
POS0	POS1	PORA	PORB	PST	PIS2•PIS3	POS0	POS1	POTA	POTB	ENABLE
0	0	X	X	X	0	0	0	0	0	0
0	0	X	X	0	1	1	1	0	0	1
0	0	X	X	1	1	0	0	0	0	0
1	1	0	0	0	X	1	1	1	1	0
1	1	1	0	0	X	0	1	0	1	0
0	1	X	1	X	X	0	0	0	0	0
1	1	X	X	1	X	1	1	0	0	0
1	1	0	1	0	X	1	0	1	0	0
1	0	1	X	X	X	0	0	0	0	0
1	1	1	1	0	X	0	0	0	0	0
0	1	X	0	X	X	0	1	0	1	0
1	0	0	X	X	X	1	0	1	0	0

Fig. 5.9 Logic truth table for the outgoing part of the CILU.

INPUTS						OUTPUTS			
Incoming Current State		Incoming Receive Signalling		ALU Status		Next State		Incoming Transmit Signalling	
PIS2	PIS3	PIRA	PIRB	PST	POSO v POS1	PIS2	PIS3	PITA	PITB
0	0	0	0	X	X	0	0	0	0
0	0	1	0	X	X	1	0	1	0
0	0	0	1	X	X	0	1	0	1
1	0	X	0	X	X	1	0	0	0
1	0	X	1	X	X	1	1	0	1
0	1	0	X	X	X	0	1	0	0
0	1	1	X	X	X	1	1	1	0
1	1	X	X	0	0	0	0	0	0
1	1	X	X	1	X	1	1	0	0
0	0	1	1	X	X	1	1	1	1
1	1	X	X	0	1	1	1	0	0
0	0	1	1	X	0	1	1	1	1

Fig. 5.10 Logic truth table for the incoming part of the CILU.

combinational logic designs to reduce the size of the network, since many of the inputs that affect the incoming network have no effect an the outgoing network and vice-versa. The ALU performs the arithmetic operations in the PE. High-level functional modules were used to simulate the ALU. High-level modules were used because the specific operation of the ALU was not important in analyzing the operation of the TDMP architecture. However, the capability to move input operands to and results and status information from the ALU is important to the operation. As a result in this model the ALU has two input operand registers called A and B that are gate connected to DBUS. Outputs from the ALU consist of a gated connection from the results (R) register and the status (ST) register to DBUS. The remaining data paths for the PE are the C-move processor, program memory and data memory. These data paths are exactly the same as those described in Section 5.1 for the TDS network. The purpose of the processor is to move under stored program control data and signalling to and from the data paths in the PE. In this simulation the data memory is used as a scratch pad by the processor during conditional move operations and the program memory contains the PE control program.

5.3 TDMP Simulation Model Operation

In describing the TDMP simulation model operation, we will assume that the PE and TDS program memories have been initialized prior to program execution. We will begin by discussing the PE simulation model operations followed by a discussion of the TDS simulation model operations. We end this chapter with a summary discussion of the overall TDMP simulation model.

5.3.1 **PE Operation** The flowchart shown in Figure 5.11 and the program code shown in Figure 5.12 describes the programmed control operations of the PE. These instructions are stored and fetched from the processor program memory. The PE begins by moving the ALU status, the incoming port A and B, and outgoing port A and B receive signalling bits to the incoming and outgoing communication logic networks, respectively. The processor

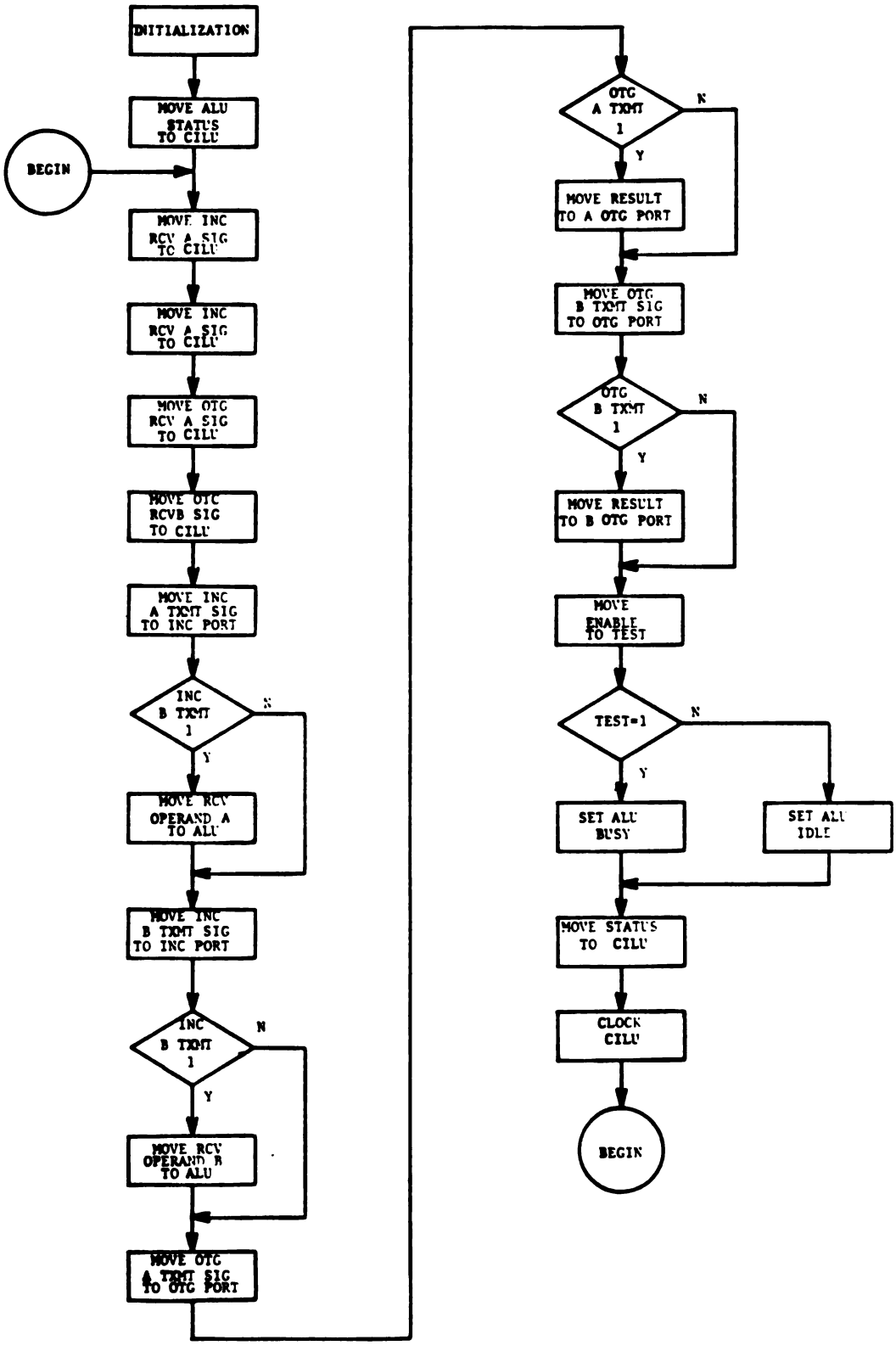


Fig. 5.11 Flowchart of the PE control algorithm.

	PM Address	PE Program	
0,1	57,58	1,95 <- 1,125	... Move status to CILU
2,3	59,60	1,86 <- 1,82	... Move inc RCVA signalling to CILU
4,5	61,62	1,87 <- 1,85	... Move inc RCVB sig to CILU
6,7	63,64	1,90 <- 1,118	... Move outgoing RCVA sig to CILU
8,9	65,66	1,91 <- 1,121	... Move outgoing RCVB sig to CILU
10,11	67,68	1,70 <- 1,66	... CILU PITA to inc TXMT A sig
12,13	69,70	3,123 <- 1,80	... Conditionally move inc A operand to ACU
14,15	71,72	1,73 <- 1,89	... CILU PITB to inc TXMT B sig
16,17	73,74	3,124 <- 1,83	... Conditionally move inc B operand to ACU
18,19	75,76	1,106 <- 1,92	... CILU POTA to outgoing TXMT A sig
20,21	77,78	3,104 <- 1,126	... Conditionally move results to outgoing A results
22,23	79,80	1,109 <- 1,93	... CILU POTB to outgoing TXMT A sig
24,25	81,82	3,107 <- 1,126	... Conditionally move results to outgoing B results
26,27	83,84	1,20 <- 1,96	... Move enable (PEXEC) to test location
28,29	85,86	2,0 <- 0,91	... Conditionally jump to set Inst=1
30,31	87,88	1,122 <- 0,0	... Set Inst=0
32,33	89,90	1,0 <- 0,93	... Jump to move status to CILU
34,35	91,92	1,122 <- 0,1	... Set Inst=1
36,37	93,94	1,95 <- 1,125	... Move status to CILU
38,39	95,96	1,94 <- 0,1	... Clock CILU CIK=1
40,41	97,98	1,94 <- 0,0	... Clock=0
42,43	99,100	1,0 <- 0,59	... Jump to move inc RCVA signalling to CILU

Fig. 5.12 PE control program code.

moves the generated incoming port A transmit signalling bit (ITA) from the CILU to the incoming port A transmit signalling register, ITSIGA. If this bit is logic 1, the processor moves the incoming A operand from IDOUTA to the ALU A input. If this bit is logic zero the incoming A operand is not moved to the ALU. The ITA bit represents acknowledgment of receiving the incoming A operand. Similarly, ITB is moved from the CILU to the incoming port B transmit signalling register, ITSIGB. If this bit is logic 1, the processor moves the incoming B operand from IDOUTB to the ALU B input. If this bit is logic 0 the incoming B operand is not moved to the ALU. Next the processor moves the outgoing port A transmit signalling bit, OTA, from the CILU to the outgoing port A transmit signalling register, TSIGA. OTA represents a request to send results as determined by the communications logic network. If this bit is logic 1 the processor moves the results from the ALU output, R, to the outgoing port A transmit results register, TRESA. If this bit is logic zero, the results are not moved. Similarly, the processor moves the outgoing port B transmit signalling bit, OTB, to the outgoing port transmit B signalling register, TSIGB. If this bit is logic 1 the processor also moves the results from the ALU output, R, to the outgoing port B transmit results register, TRESB. The processor now tests the communication logic network enable execution, EXEC. If this signal is logic 1 the processor moves an arithmetic instruction and start execution signal to the ALU input, INST. If EXEC is a logic 0 the arithmetic instruction and start execution signal are not moved to the ALU. The processor now clocks the communication logic network so that it can generate its next state based on its current input and outputs. The processor jumps to the beginning of its control program to repeat this cycle of move instructions.

5.3.2 **TDS Operation** The flowchart shown in Figure 5.13 and the program code shown in Figure 5.14 describe the TDS network processor operations. The SYNC bit is a synchronization pulse that occurs at the beginning of each time slot. This bit is used to

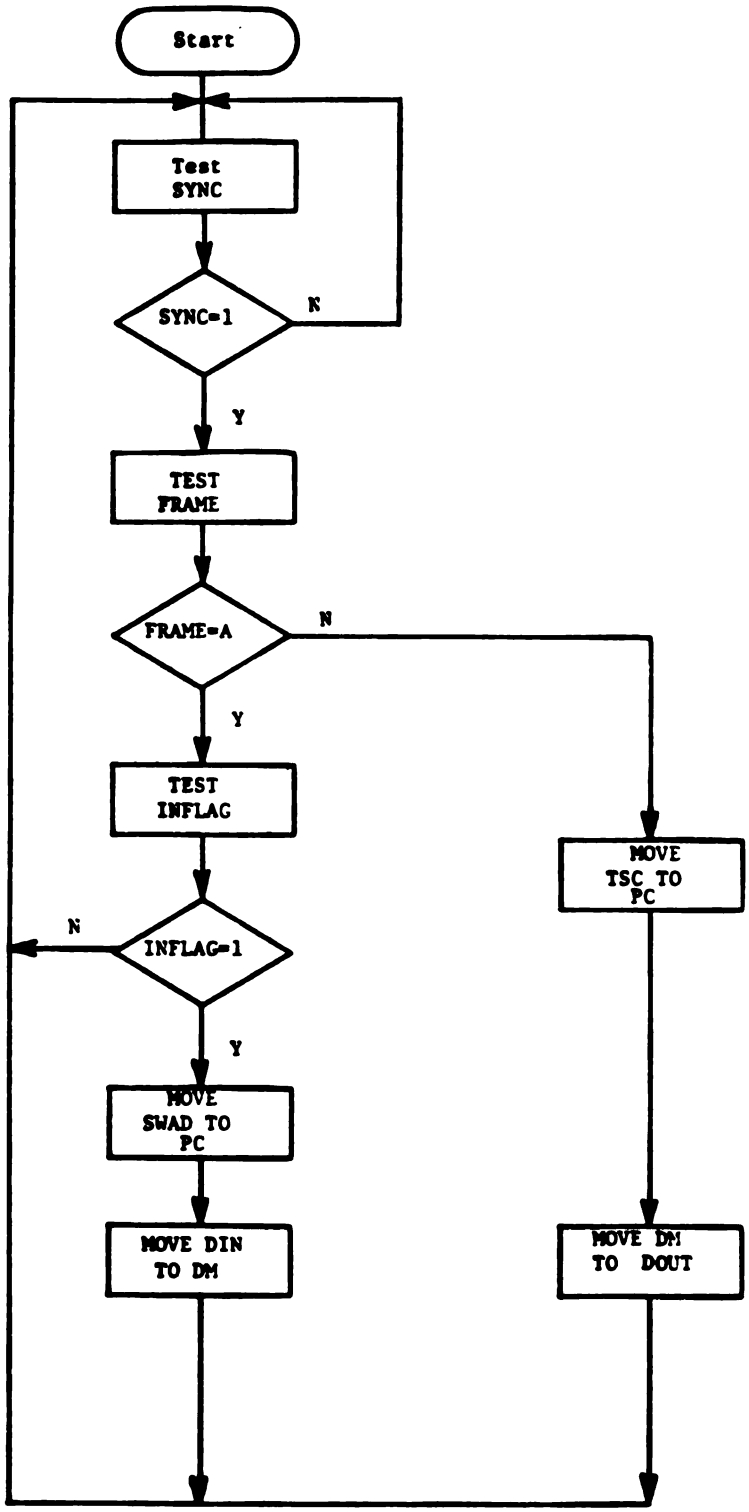


Fig. 5.13 Flowchart for the TDS control algorithm.

A Program Memory

	PM Address		
	9,10	1,20 <- 1,77	... Move SYNC bit in test location
	11,12	3,0 <- 0,15	... If SYNC=1, jump to test frame
	13,14	1,0 <- 0,9	... Return to Sync
*	15,16	1,20 <- 1,76	... Move frame to test location
*	17,18	3,0 <- 0,21	... If F=1, jump to input routine
*	19,20	1,0 <- 0,27	... If F≠1, jump to output routine
	21,20	1,20 <- 1,65	... Move input flag (INF) to test location
	23, 24	3,0 <- 1,66	... If INF=1, jump to location specified by SWAD
	25,26	1,0 <- 0,9	... If INF=0, return to SYNC
	27,28	1,0 <- 1,75	... Jump to location specified by TSC
	29,30	4,68 <- 1,67	... Move DM to DM location specified by A loc
	31,32	1,0 <- 0,9	... Return to scan
	33,34	1,74 <- 1,32	... Time slot zero data
	35,36	1,0 <- 0,9	... Return to scan
	37,38	1,74 <- 1,33	... Time slot one data
	39,40	1,0 <- 0,9	... Return to scan

TSC = 33 => TS0
TSC = 37 => TS1

* Note: B Program memory is exactly the same as A with the one exception for testing the frame.

15,16 1,20 <- 1,76
17,18 3,0 <- 0,27
19,20 1,0 <- 0,21

Fig. 5.14 TDS control program code.

synchronize the TDS with the input and output time-division multiplexed buses. The TDS processor moves this bit to a scratch pad location in data memory. If SYNC is a logic one, each of the TDS processors move the FRAME bit to a scratch pad location in their respective data memories. If SYNC is logic 0 the TDS processors continuously moves the SYNC bit to the same scratch pad location in their data memories until SYNC is logic 1. If the FRAME bit is logic one, TDS processor A performs the operations in the input processing branch of the TDS flow chart and TDS processor B performs the operations in the output processing branch of the TDS flowchart. If FRAME is logic zero processor A performs the output processing operations and processor B performs the input processing operations.

The input processing operations begin with a test of the INF bit. The INF bit is moved to a scratch pad location in memory. If INF is logic one the processor moves the contents of the SWAD register to the program counter register. The instruction pointed to at this address is a direct move or an indirect move of the DIN register contents or some other type of requested switching service function. For this example, assume the instruction pointed to by the PC is an indirect move, then the processor moves the contents of the DIN register to a data memory location specified by the contents of the A register. The program now jumps back to the test SYNC instruction. If the INF bit was logic zero the program does not perform any switching service and immediately jumps back to the test SYNC instruction.

The output processing operations begin by testing the OUTF register. If OUTF is logic zero the contents of TSC are moved to the processor program counter. The instruction pointed to at this address causes a jump to a program memory location that directly move the contents from a location in data memory to the DOUT register and returns to the test SYNC instruction. If OUTF is logic one the processor immediately jumps back to the test SYNC instruction without moving any data to the DOUT register.

5.4 TDMP Simulation Model Summary

Because of APL space limitations, all of the PE variables needed to simulate a twelve PE TDMP architecture could not be created and stored in one file. However, by using a set-up and end-up program, we were able to achieve the same effect. The set up program takes shared variables such as program memory, incoming and outgoing port registers, etc., and makes them specific variables for a particular PE by adding a suffix to these variables. When done computing the end-up program stores the contents of these specific variables and removes the subscripts to make them shared variables again. Each time a PE is used it must use the set up and end-up program.

The APL simulation model provides us a means of simulating the TDMP system operation; however, timing problems can occur since APL simulation does not include timing simulation methods which could predict timing problems. A timing simulator allows verification of proper circuit behavior in the presence of variation in gate delays. The worst case circuit behavior is obtained based on the minimum and maximum transition delays assigned to the gates is the model. By allowing observation of the circuit whenever simulation time is incremented rather than waiting or assuming the circuit has stabilized, a better understanding of circuit operation may be obtained to aid in circuit design or diagnosing problems. Detail logic gates and timing simulation can be implemented in the APL model by writing programs that perform these functions. However, because these functions are not built into the APL simulation the cost for implementing them in our model is expensive.

APL is a sequential simulation language, which means it cannot simulate concurrent activity. However, we can achieve the effect of performing concurrent activity by stopping the simulation timing clock and sequentially performing computations then restarting the clock again. This is an awkward and expensive way to simulate concurrent activity; still, it is the best alternative available. But even given these short comings of the simulation language, it does allow

us to obtain some useful and significant information in predicting TDMP's system performance. This performance will be discussed with the aid of sample computations in the next chapter.

CHAPTER VI

TDMP PERFORMANCE EVALUATION

Chapter 5 described the TDMP APL simulation model and its operation. In this chapter we evaluate the performance of the TDMP design based on this model. In our investigation we did not design many evaluation tools to measure system performance. For example, we didn't design any hardware monitors for measurements of hardware activity, software monitors for event recording during program execution nor did we develop workloads that represent work expected of the system. We feel that a rigorous performance evaluation of multiprocessor systems employing these techniques is a research project unto itself. In addition there is a question of whether new performance tools are needed or how do the currently used performance tools need to be extended to be applicable to multiprocessor systems. As a result, we relied primarily on monitoring signals and registers in simulated clock time for recording the activity of small portions of the system and monitoring computational results in simulated clock time for measuring total system performance. Section 6.1 discusses the qualitative and quantitative performance results obtained from this simulation model.

On the one hand, we have the conventional single-processor systems which have many uses but the range of use is limited by the von Neumann bottleneck. On the other hand, we have fixed-array-processor systems which overcome the von Neumann bottleneck but at the expense of limiting its range of use by the fixed interconnection structure. TDMP eliminates or improves upon the range of use limiting factors in both single-processor and fixed-array-processor systems. In Section 6.2 we use data bandwidth as a measure of computational performance and compare the data bandwidths of a single-processor, TDMP, and fixed-array-processor systems in computing a fast-Fourier-transform (FFT) and a digital filtering algorithm. This comparison is done to show how much TDMP improves arithmetic computation over single-processor systems and how much data bandwidth we must sacrifice to obtain

flexibility over fixed array processor systems. The two algorithms chosen for the comparison represent the important class of recursive algorithms used in digital signal processing. In Section 6.3, we discuss what has and has not been learned from the simulation. And, finally, Section 6.4 describes the differences in TDMP data flow processor and Dennis's data flow processor.[37]

6.1 Performance Data

The performance data obtained from the simulation model is based on a communication saturated TDMP system with one processing element. By communication saturated we mean that the system bandwidth is dominated by the interprocessor communication and the processor arithmetic execution time is negligible compared to this communication time. With this approach the bottlenecks, delays and throughput we examine are a result of the TDMP architecture only. The PE computes results and transmit these results back to itself through the TDS network. Figure 6.1 shows a block diagram of the simulation model with the PE shown in two parts for clarity purposes. In the model, time slot zero, TS0, identifies the outgoing port of the PE and time slot one, TS1, identifies the incoming port of the PE. With this model we were able to test the communication protocols between PE outgoing and incoming communication ports, the A and B frame submultiplexing, the time-division switching network and the data flow control.

The simulation model is synchronized to a master clock. Four simulation clock pulses correspond to one read/write memory cycle, t_m. From the simulation we found that the maximum rate a PE can generate result packets, T_{PG}, is once every three cycles through its control program. The first cycle loads the operands into the ALU, the second cycle initiates computation and the third cycle transmits the results. In the PE control program shown in Section 5.3, this corresponds to 120 t_m, assuming that the needed operands have been received and the ALU execution time is zero. However, if the operands are not immediately available the time

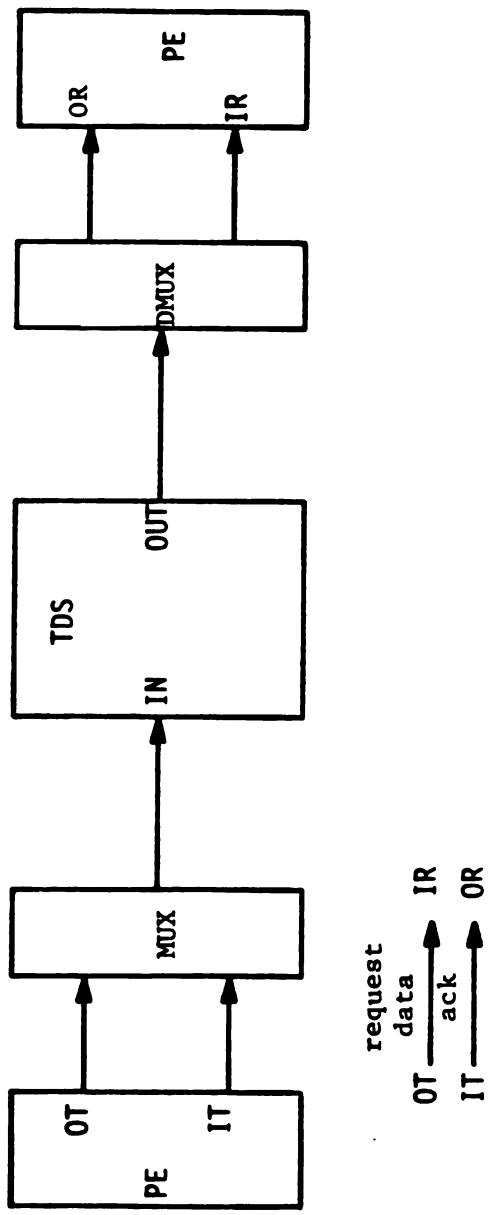


Fig. 6.1 Block diagram representation of a single PE TDMP model.

delay in receiving these operands reduces the rate at which the PE generates result packets. This time delay in receiving operands is attributed to the transmission, switching, and packet flow control. The worst case transmission delay, T_t, is caused by the time-division multiplexing and submultiplexing of the information packets. In the model $T_t = 2NT_X$, where N is the number of time slots, T_X is the time slot size, and 2 is the number of submultiplexed frames. The worst case switching delay, T_D, also equals $2NT_X$, since in the model the switch sampling period equals the time slot size. The transmission and switching delays are fixed and each packet transmitted must sequentially experience these delays. This results in a pipeline realization of packets with the sections of the pipe consisting of the packet generation, transmission and switching. Without including the packet flow control and assuming worst case, we found that the model computing with two input operands produces the first result packet in $T_{PG} + T_t + T_D$ units of time and the following result packets every T units of time where T is the maximum of (T_{PG}, T_t, T_D). The timing diagram shown in Figure 6.2 is the one used for timing and control of the two time slot model. From this diagram we find that the time slot size is 98 simulation clock pulses which correspond to 24.5 t_m. If we assume $t_m = 30$ nsec, then

$$T_D = T_t = (2)(2)(24.5)(30 \text{ nsec})$$
$$= 2.94 \ \mu\text{sec}$$

and
$$T_{PG} = (120)(30 \text{ nsec})$$
$$= 3.6 \ \mu\text{sec}.$$

As a result $T = T_{PG}$ since $T_D = T_t < T_{PG}$. For a large number of operations this corresponds to a data bandwidth of 277 kHz where data bandwidth is defined as the maximum number of results that can be generated per unit time, $b_D = \frac{1}{T}$. However, if $N > 2$ time slots then $T_D = T_t > T_{PG}$ and $T = T_D = 1.47N \ \mu$sec, resulting in $b_D = 1/1.47N \ \mu$sec.

If the packet flow control is added to the model, the previously computed b_D decreases

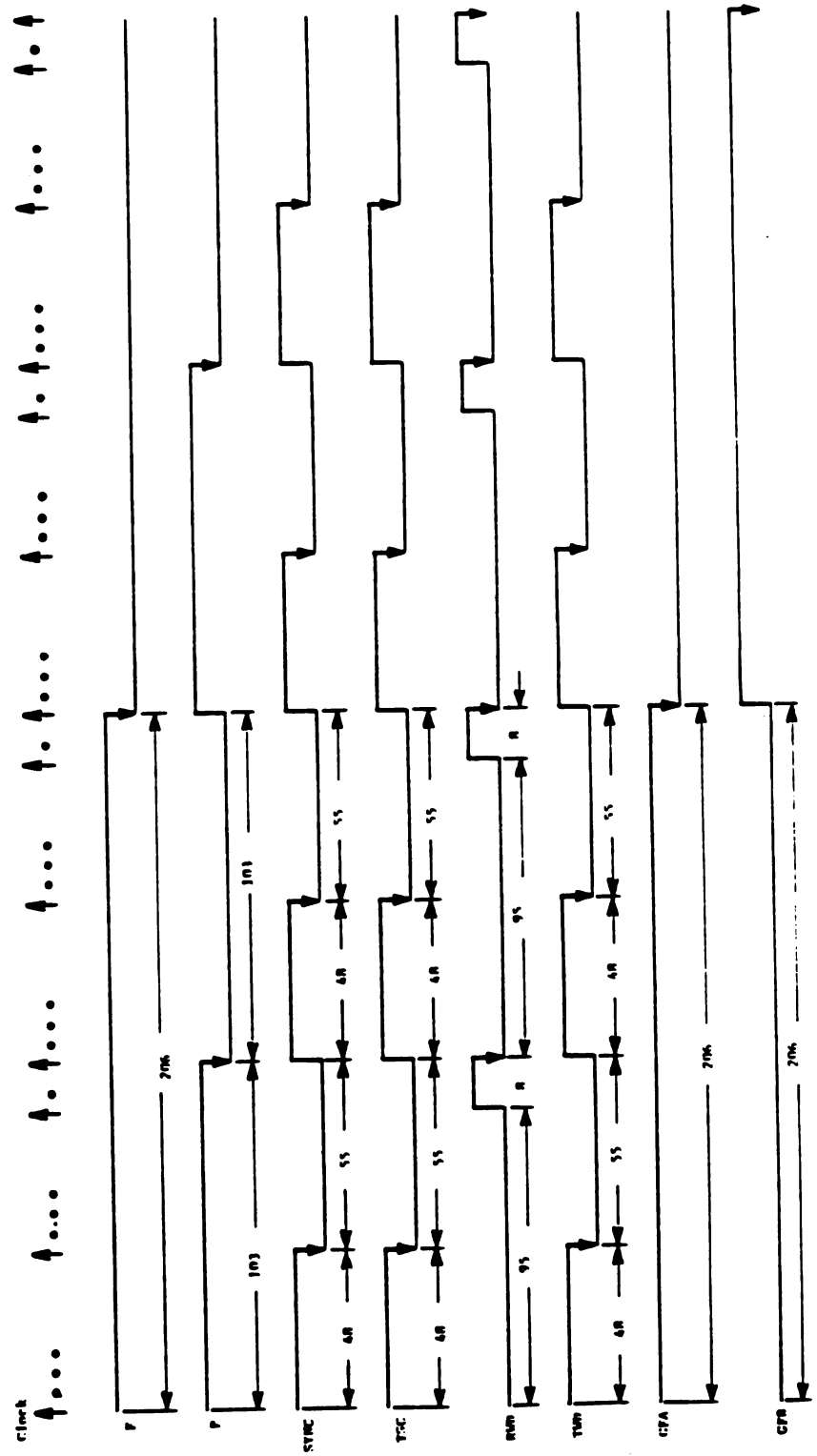


Fig. 6.2 Timing diagram for a two time slot model.

significantly. The decrease in bandwidth is due to the interruption of the pipeline operation and added delay both caused by the packet flow control protocol. The packet flow control protocol is responsible for preventing new result packets from being generated before previously transmitted result packets have been acknowledged. Since the acknowledge signal is returned after the request signal and data have been received, the time between generating new result packets increases by the amount of time it takes the receiver PE to generate an acknowledge signal plus the time it takes the transmitter PE to receive it. The worst case time for generating an acknowledge signal, T_{ack}, is 1 cycle through the PE control program or 40 t_m. This time plus the transmission and switching delay time experienced by the acknowledge signal is added in the packet generation, transmission and switching time of the result packet sent. As a consequence

$$\begin{aligned} T &= T_{PG} + 2T_t + 2T_D + T_{ack} \\ &= (4.8 + 2.94N)\ \mu\text{sec} \end{aligned}$$

and from the model where N = 2

$$b_D = \frac{1}{T} = 93.6 \text{ kHz.}$$

If sufficient size receiver first-in-first-out (FIFO) queues or receive buffers with arbitration circuitry are used to store and control incoming packets to the PE, then the packet flow protocol can be omitted and the b_D is increased to 277 kHz for N = 2 or $\frac{1}{1.47N}$ *MHz* if N > 2 time slots. These are the data bandwidths we obtained for the model when we didn't consider the packet flow control. The cost of the bandwidth improvement is a decrease in system reliability because the acknowledge signal used for packet flow control was our only verification that the transmitted packet was received successfully.

6.2 Comparison of Computational Performance

We decided to use the performance data obtained from the simple one PE model of TDMP to determine and calculate the (data) bandwidth of a large TDMP system suitable for implementing FFT and digital filtering algorithms. This decision was based on the high cost of the TDMP APL simulation and the small amount of new information that would be gained in simulating a large system. This single PE model is justified because TDMP is a synchronous system and there is no contention for communication. Each PE is given a designated time slot to transmit and receive data. As a result all PEs are equal and we can determine on a worst case basis the time required for each PE to compute, transmit and receive results. This is different from an asynchronous system where the time for each PE to transmit and receive results varies and depending on the type of interconnection network there may be some contention for communication. The (data) bandwidth was chosen as the performance index in comparing the computational performance of TDMP, single-processor and fixed-array-processor systems. This choice is more appropriate than the memory cycle time or instruction execution time since it allows for the fact that instructions may be executed concurrently. The evaluation of the b_D's is based on each system performing the same FFT and digital filtering algorithms with comparable processors. We begin this analysis with a comparison of the b_D's resulting from the processing systems performing an FFT algorithm.

The FFT algorithm implemented on each processing system is based on the decimation in time principle and is illustrated for an 8-point sequence in the signal flow graph shown in Figure 6.3. In the signal flow graph, there are four columns and each column contains eight entries. For the sake of clarity, the two-dimensional variable y(k,i) is used to denote the value of a given node in the array, where k is the number of the column and i is the number of the component within the column. At the node corresponding to column k and row i, the variable y(k,i) is found from an equation of the form:

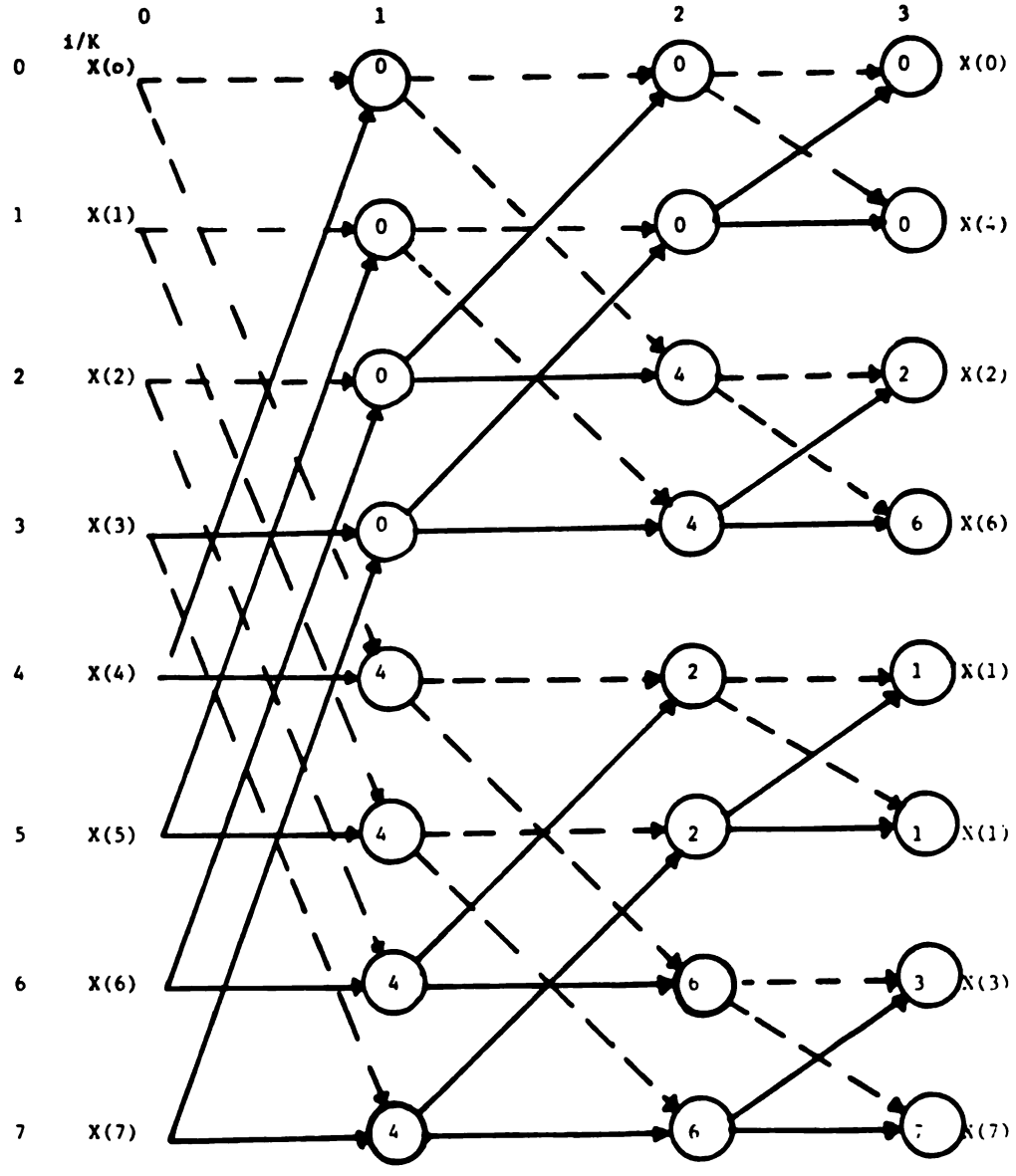


Fig. 6.3 Signal flow graph of a 3-pt. FFT.

$$y(k,i) = y(k-1, i_1) + w^r y(k-1, i_2) \qquad (1)$$

where i_1, i_2 and r are functions of the location within the array and $W = e^{-(j2\pi/N)}$. In each case, the dashed line connecting the variable in column k-1 with column k refers to the first term on the right-hand side of Equation (1), i.e. the nonweighted term. The solid line refers to the second term on the right hand side of (1), i.e., the weighted term. The number in the circle is the degree of W as indicated by term r in (1). The important parameters needed to determine and compare the (data) bandwidth among the three systems implementing this FFT algorithm are defined:

N = the number of data points in the FFT

T_E = execution time of the processor in performing the butterfly computation, $y_3 = y_1 + w^r y_2$

M = the average number of FFT operations to be performed

T_X = time slot size

T = worst case TDMP system delay

t_r = register delay

t_m = read/write memory cycle time

$b_D(M)$ = data bandwidth for a finite number of FFT operations

b_D = data bandwidth when the number of FFT operations is large (M -> ∞)

The b_D analysis assumes that all initialization of the processors has occurred prior to the beginning of the FFT program execution and that each individual processor performs the same computation, $y_3 = y_2 + w^r y_2$, in the same amount of time. To compute y_3 each processor must perform 4 real multiplications and 2 additions. We also assume that the input and output interfaces for each system is manageable and not a limiting factor in the b_D performance. With

these assumptions given, the b_D of a single-processor system, can be determined from the following equation,

$$b_{D_1} = \lim_{M \to \infty} b_{D_1}(M) = \frac{1}{T_E(N \log_2 N) + 6t_m(N \log_2 N)} \tag{2}$$

In equation (2) $T_E(N \log_2 N)$ is the arithmetic execution time needed to compute one FFT operation with a single-processor, $6t_m(N\log_2 N)$ is the accessing and storage time of the real and imaginary parts of the two input operands and the results during one FFT operation.

The TDMP b_D is based on a TDMP system with $N\log_2 N$ PEs. Each PE is assigned a location in the N x $\log_2 N$ pipeline array of processors as represented in the signal flow graph shown in Figure 6.3 for $N = 8$. Each PE is programmed to execute the butterfly computation as a single data flow operation and to transmit the results of this computation to the next column of PEs based on its location assignment within the array. This arrangement of PEs form an asynchronous pipeline flow of result packets with the synchronization of the computations controlled by the data flow principles at the architecture level. The worst case b_D for TDMP is characterized by the following equations;

$$b_{D_2}(M) = \frac{M}{(T_E + T) \log_2 N + (T_E + T)(M-1)} \tag{3}$$

and

$$b_{D_2} = \lim_{M \to \infty} b_{D_2}(M) = \frac{1}{(T_E + T)} \tag{4}$$

where T is the worst case TDMP system delay. T is (4.8 + 2.94N) μsec if packet flow control protocol is used and 1.47N μsec if it is not used. These two cases correspond to not using and using input queues as buffers in the PEs. The term $(T_E + T)\log_2 N$ is the system delay, i.e.,

the time before the first results appear at the output of the system. After the system delay, a FFT operation result appears every $T_E + T$ units of time for an unbounded number of operations.

The fixed-array-processor directly implements the signal flow graph of Figure 6.3 with hardwired interconnections between the processors. Each interconnection path contains a register to synchronize the data flow between processors. The execution time, T_E, and the register delay time, t_r, determine the processor b_D,

$$b_{D_3}(M) = \frac{M}{(T_E + t_r)\log_2 N + (T_E + t_r)M - 1}$$

$$b_{D_3} = \lim_{M \to \infty} b_{D_3}(M) = \frac{1}{(T_E + t_r)}$$

Figure 6.4 shows a graph of the FFT data bandwidths for the single-processor, TDMP, and fix-array-processor as a function of the number of operations, M. The curves were generated assuming that N=8, $t_m = 30$ *nsec*, $t_r = 10$ *nsec* and

$$T_E = (4T_M + 2T_A)$$

where T_M and T_A are respectively the hardware processor floating-point multiplication time (27 μs) and addition time (14 μs).

The curves in Figure 6.4 show that for a large number of FFT operations, TDMP b_D is more than 15-times larger than the b_D of a single-processor system and 1.5-times smaller than that of a fixed-array-processor in computing an 8-point FFT. The b_D of the fixed-array-processor can be viewed as an upper-bound and the maximum b_D improvement we can obtain over the single-processor with TDMP. In order to achieve this maximum improvement we must reduce the interprocessor communication overhead either with hardware or

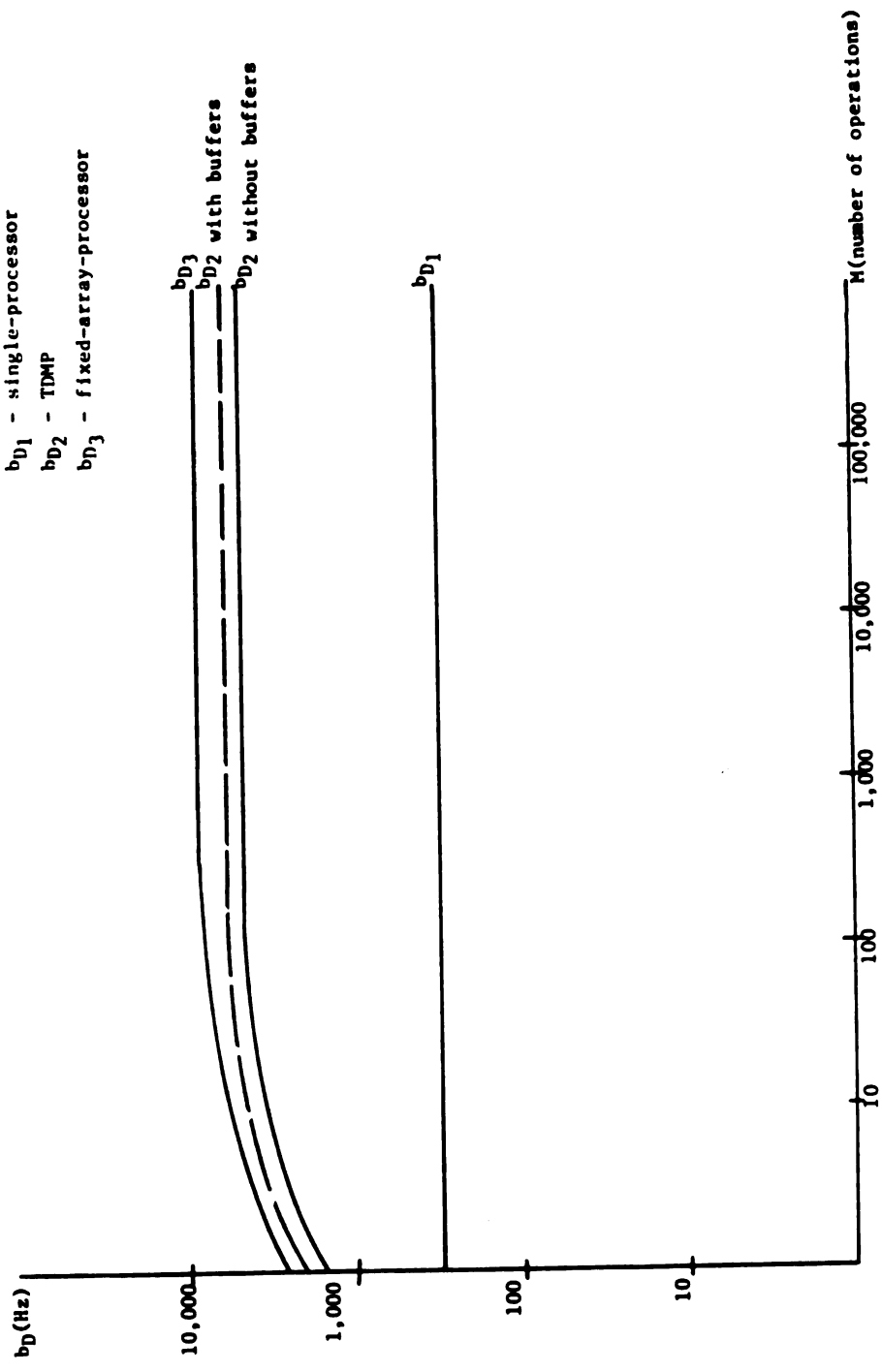


Fig. 6.4 Bandwidth comparison as a function of the average number of FFT operations

reducing the communication protocols that are in software. For example the dash curve shows the improvement in TDMP b_D if input queues are used. In this case TDMP b_D is more than 19-times larger than the b_D of a single-processor and 1.2-times smaller than that of a fixed-array-processor. This comparison shows that for a small reduction in b_D of fixed-array-processor, TDMP enhances both single-processor and fixed-array-processor range of usefulness, e.g., increases b_D over single processors and adds flexibility over fixed-array-processor.

Figure 6.5 shows the b_D's as a function of the processor execution time for large M. From these curves we can see that the single-processor system has a larger b_D than TDMP if the execution time is less than 2.8 μs and as the execution time increases above 2.8 μs this situation flip-flops. From the same curves we see that if T_E is greater than a 1000 μs, the b_D of TDMP and fixed-array-processor is about the same. These results are very important and can be easily explained. In the first case where TDMP b_D is less than that of a single-processor, the PEs in the TDMP are spending most of their time communicating and waiting for new operands because the computation time to communication time ratio is small ($<.04$). In the second case, where the TDMP b_D is comparable to the fixed-array-processor, the computation time to communication time ratio is large (>13) so that PEs spend the majority of their time computing instead of idling. This points out the importance of granularity in TDMP. Granularity is the size task a PE must perform before it is required to communicate with other PEs in the system. For TDMP the granularity must be large such that the computation time to communicate time ratio is large for a given problem.

To illustrate the advantage that TDMP has over fixed-array-processors for this class of problems, we simply change the application problem. Changing the application problem in TDMP requires no hardware changes. We simply reprogram the data paths and input new information packets with new tag values, assuming of course that the program for the new task has been

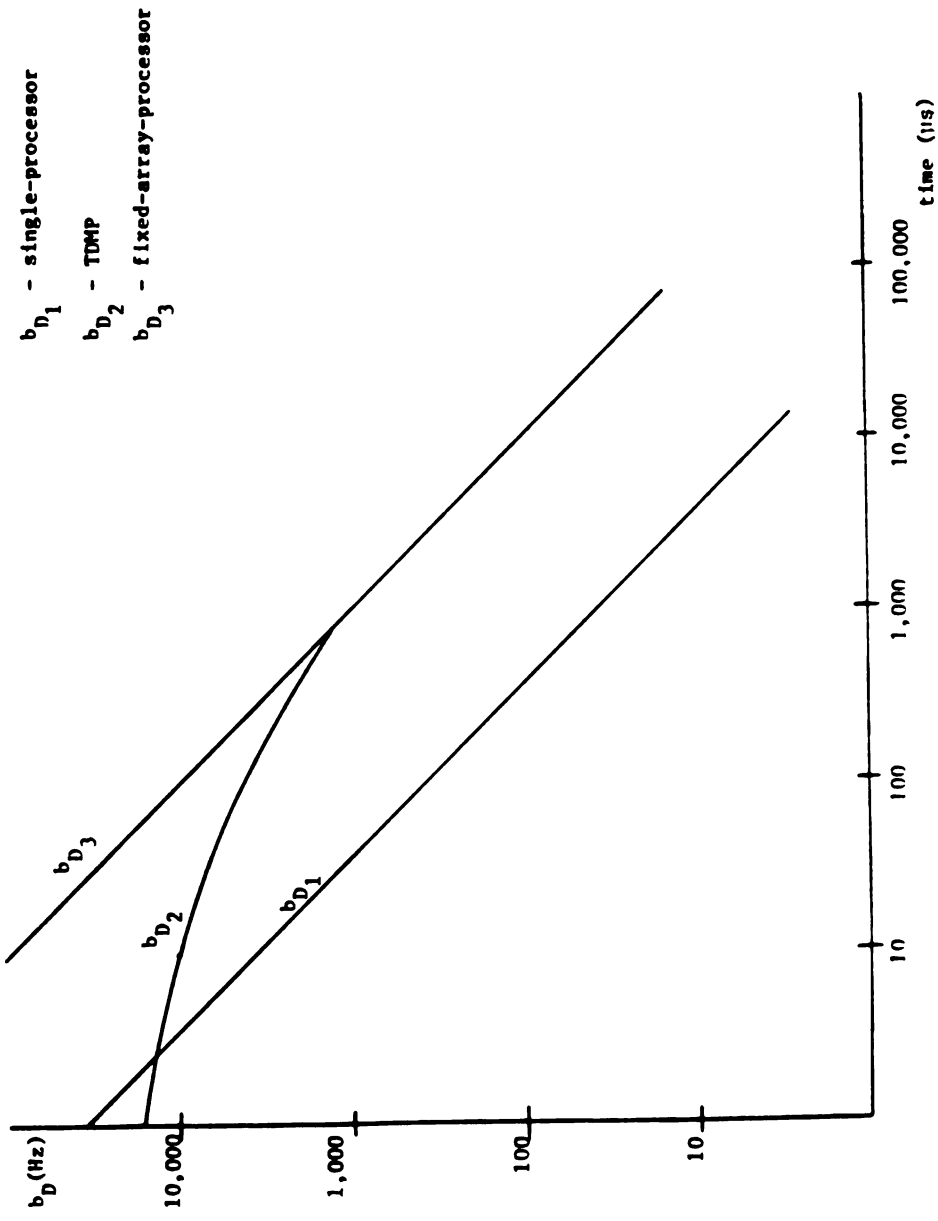


Fig. 6.5 Bandwidth comparison as a function of PE execution time for the FFT computation.

previously stored in the PE. However, changing the application in a fixed-array-processor requires changing the hardwired interconnection paths between processors or possibly experiencing a significant degradation in the b_D. In addition if the problem size changes unneeded PEs in TDMP architecture can be utilized for multiprogramming purposes. Since TDMP is based on data flow principles and suffers no side effects of concurrent processing, we can execute two or more different application problems simultaneously. There is no way in which fixed array-processors can do this. Because in fixed-array-processors the correctness of the results requires all processors to be synchronized together. As a result, unneeded processors must remain idle, thus reducing the utilization of the fixed-array-processing system.

As a second example of TDMP performance, we will compare the b_D's of the three processing systems implementing the digital filtering algorithm given in Chapter 3 and shown in Figure 3.8. In the single-processor system 4 multiplications, 3 additions, and 8 memory accesses and stores must be accomplished to compute one filter result. Thus

$$b_{D_1} = \frac{1}{4t_M + 3\,t_A + 8\,t_m}$$

For the TDMP system, we assume N = 8 to make the architecture fit the problem size and directly implement the signal flow graph shown in Figure 3.8. This results in

$$b_{D_2} = \frac{1}{t_M + T}$$

This b_D assumes that many filter computations are performed and the multiplication time is larger than the addition time. Similarly for the fixed-array-processor system we assume an architecture size to fit the problem size and directly implement the signal flow graph which results in

$$b_{D_3} = \frac{1}{t_M + t_r}$$

Figure 6.6 shows the b_D's for each processing system as a function of the processor multiplication time. In this example we see that the b_D of TDMP is larger than that of the single-processor for all values of processor multiplication time greater than 15 μsec and less than that of the fixed-array-processor until the multiplication time exceeds 500 μsec. The starting point of 15 μsec was used so that our equations for b_{D2} and b_{D3} will be valid, since $t_A < t_M$ is required for these equations to be correct. The results from this analysis again shows us the effect that communication overhead has on the TDMP b_D and the amount TDMP enhances b_D over single processor systems.

6.3 Discussion of the Simulation

The simulation model was the driving force behind designing the specific components and the interrelationship of these components to make a TDMP system work. By using a simulation model of the design, we did not have to purchase, build and test printed circuit boards and the LSI components on these boards to observe the performance and operation of the TDMP system. With simulation we were able to test new ideas quickly and to make changes easily. We also were able to develop the precise timing relationships for the system operation and determine what influences these relationships. We did not find out how TDMP performs when the system hardware components are not ideal but vary with temperature and age. We were not able to truly simulate concurrent PE activity because the simulation processor was sequential. The capacitance of the bus data paths that influence signal propagation could not be simulated inexpensively. To simulate this capacitance we need a stix layout of the intergrated circuit design of the TDMP architecture in order to determine, at best, a good guess of the bus lengths necessary to calculate a capacitance value that could be simulated for each bus data path in this model. This approach is very expensive, time consuming and not

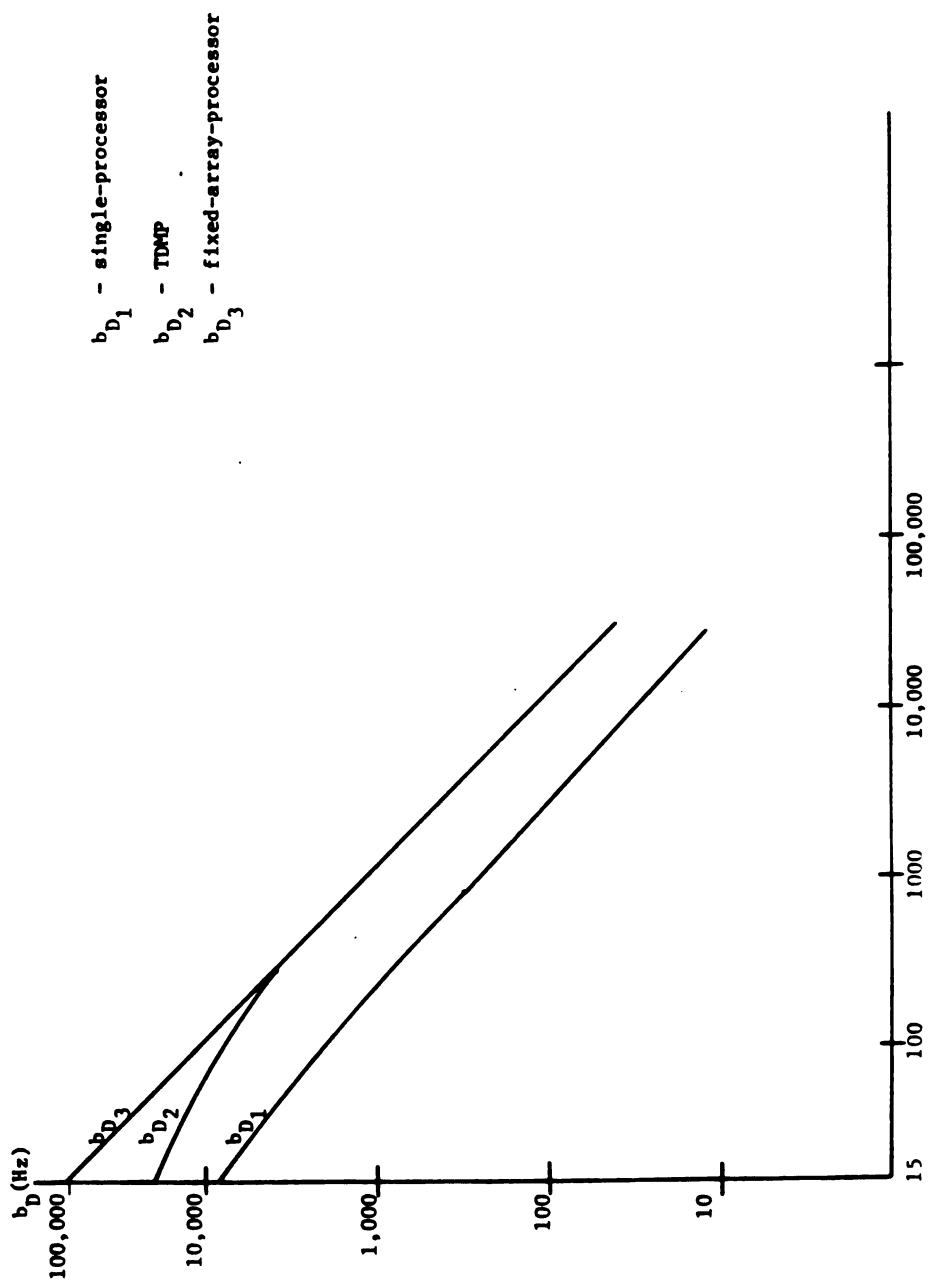


Fig. 6.6 Bandwidth as a function of multiplication time for the digital filtering computations.

necessary for this initial simulation model, so it was not done. However this should be done when simulating the architecture for actual VLSI implementation to study its effect on performance.

6.4 TDMP and Dennis Data Flow Models

There are two primary differences in the TDMP data flow architecture and the data flow architecture developed by Dennis at MIT.[37] The first difference is that in TDMP the data flow operators and memory cells are in the same PE module with an interprocessor communication network used to interchange result information packets between PEs. In Dennis's model the operators and memory cells are separated. All information packets are transmitted through an arbitration network from the memory section to the operators. And all result information packets are transmitted from operators through a distribution network to the memory cells. The TDMP approach does not increase the amount of concurrent activity over Dennis's model but rather allows for locality of information flow. This approach reduces the amount of parallelism in a problem required for high PE utilization. The argument about locality of information flow is true for any multiple processor machine in which each processor holds a part of an overall program. However, two factors increase the likelihood that such good decomposition can be achieved for TDMP. First, since TDMP is based on data flow principles, operations are constrained only by the availability of data. Tolerance of asynchronous behavior at the architecture level allows greater flexibility in mapping programs. Second, since data flow languages are free of side effects decomposition is an easier task to perform. The second difference is that TDMP uses time-division communication techniques for the transmission and switching of information packets and Dennis's model is designed for space-division techniques. The time-division techniques are better in conserving valuable chip area over the space-division techniques for single-chip VLSI implementation. And the use of an intelligent switching network in TDMP allows for the possibility of performing operations other than "plain old switch-

ing". For example, the switch could modify the data as it is being transmitted to the destination PE or implement some of the system operating functions to improve reliability and diagnosability in the system.

CHAPTER VII

CONCLUSIONS

7.1 Summary

Advances in improving the computer's computational speed result from reducing the switching delay times of elementary logic gates, improving methods for performing primitive arithmetic operations and designing computer architectures that overcome the intrinsic speed limitations of the von Neumann machine. While the first two approaches improve the computational speed, the third approach provides the potential for making the largest increase in computational speed. Here, multiprocessors are employed; and these multiprocessor architectures exploit the inherent parallelism in specific tasks, thereby eliminating the single-processor bottleneck -- "von Neumann bottleneck". In addition, VLSI is making it feasible to build these high-performance multiprocessor structures as low-cost, flexible, single-chip system components. So, the purpose of the research reported here was to investigate a multiple processor computer structure that exploits this technology to improve the bandwidth of comparable single-processor machines and increase the flexibility of existing fixed-array-multiprocessors. Within the research reported here, the investigation attained several objectives. First, it surveyed the previous work on multiprocessor architectures to identify their principal attributes, and it suggested how well suited these attributes are for VLSI implementation for enhanced computation. Desirable multiprocessors VLSI attributes include:

— majority of the processors are identical;

— processors and memory all in close proximity;

— simple underlying communication geometry;

— regular and modular structure;

— few interconnection data paths between processors;

— flexible datapaths.

Next, it presented an architecture called TDMP (Time Division Multiple Processing) and described how it achieves the following general properties:

— greater bandwidth than comparable single-processor architectures;

— less complex switching network than a crossbar switch interconnection network;

— more flexible than fixed-array network;

— full-access and non-blocking interconnection capabilities;

— simply extends to pipeline operation;

— highly compatible with data flow algorithms;

— amenable to VLSI implementation.

A computer based simulation model of TDMP was developed to investigate the computational capabilities, hardware components and the interrelationship of these components with the others. Finally, through two algorithms that represent the important class of recursive algorithms used in digital signal processing, it compared the computational data bandwidth of TDMP to comparable single-processor and fixed-array-processor systems to show how much TDMP improves arithmetic computation bandwidth over single-processor systems and how much bandwidth we must give up to obtain flexibility over fixed-array-processor systems. By satisfying these objectives, the investigation achieved the overall purpose of the research project.

During this investigation the research project achieved several key results. First it described, from a hardware point of view a basic TDMP architecture with twelve processing

elements, two time-division switching networks and a control unit. The processing elements perform the arithmetic operations and synchronize computation for correct execution at the architecture-level using data flow principles. The time-division switching networks were shown to have full-access and non-blocking interconnection capabilities for switching the information packets and flow control signals among the multiple processors. The switch was also shown to possess several important attributes that make it well-suited for VLSI implementation. These attributes include simple underlying communication geometry (time-division multiplexed buses), modularity regularity and logical simplicity. The logical simplicity implies that incremental changes in the switch architecture (e.g., increasing the number of processing elements connected to the switch) the degree to which the changes affects the software in the switch is small. Next, we explained how TDMP executes programs based on data flow principles in which computations are allowed to proceed as soon as it operands became available. The idea of organizing a computer to operate on data as soon as its operands became available has been discussed by Dennis, Gurd and Watson, Arvind, Miller and Cocke.[36,37] However, none of these authors has suggested a detailed and efficient scheme for communicating information packets to processors for computation. This investigation has shown through simulations that an integrated interprocessor communication system using time-division multiplexing and time-division switching offers an attractive solution to this problem. Also it was shown that TDMP is unique because time-division communication techniques have not been previously investigated in data flow or conventional multiprocessor architectures.

To develop TDMP, we built and tested an APL simulation model of the architecture. The investigation found that the use of such a model was helpful in the design and analysis of the architecture. Using the performance data of a single PE TDMP model, it was shown that TDMP can improve the data bandwidth (computational speed) over single-processor systems by factors of 15 and 10, when computing FFT's and digital filtering results, respectively. The analysis also indicated that in order to achieve this improvement the granularity of tasks for

each PE must be chosen such that the computation time to communication time ratio for each PE is large. The investigation also revealed some negative attributes and deficiencies of the simulation languages available for our simulation purposes. The negative attributes, common to all of the simulation languages considered in this investigation, result from these languages not being able to truly simulate concurrent processing or asynchronous events. The reason for this is these languages are based on a single-processor executing instructions sequentially. The deficiencies of each simulation language varies with the language type. For example, APL does not have a built in timing simulator but GPSS does. However, GPSS is not particularly suited to the translation from a hardware representation to a computer program but APL is. The negative attributes and deficiencies of each simulation language considered in this investigation is discussed in more detail in Chapter 5 of this thesis.

Using the results of this study, designers of multiprocessor systems will be able to create advance systems with increased computational performance over single-processor systems, a wider range of use over fixed-array-processor systems and a hardware design that copes with interconnection data path complexity imposed by VLSI implementation. Second, since TDMP is based on data flow principals, i.e. operations are constrained only by the availability of data, the architecture allows greater flexibility in mapping programs and since, data flow languages are free of side effects decomposition of programs is an easier task to perform. The second results will aid programmers in developing parallel software to keep the parallel hardware fully occupied. The third result is an APL computer model of the TDMP system which provides a model for developers of multiprocessor systems to use in investigating the computation bandwidth of TDMP if some of the parameters of the structure are varied. For example the bandwidth for a specific problem can be investigated as the number of time slots, the time slot size and the number of multiplexed buses is varied. Currently, the model has the flexibility to vary the number of time slots, the time slot size and the control software for both the PEs and time-division switch. However, it can not vary the number of time-division multiplexed

(TDM) buses. This feature can be implemented by duplicating existing parts of the model and adding a new time multiplex switch part. This change results in a new time-division switch with multiple input and output lines. The new switch can switch data between processors on the same or different time-division multiplex bus. The changing of the number of time slots on a TDM bus or the number of TDM buses correspond to varying the N and M parameters of the model, respectively. And finally, the use of an integrated communication system based on time-division techniques has not been previously investigated in multiprocessor structures for enhanced computation. So the results of this investigation provide designers with a new trajectory for multiprocessor design and implications of using these techniques in those designs.

7.2 Further Research

As with any research project, the investigation reported here points toward several areas of additional study. Further investigation can be made into showing new options possible in multiprocessor architectures as a result of using time-division communication techniques. For example, since the switching network is a C-move processor, we can now think of ways to enhance computation by modifying the data as it is being switched. We may also be able to modify the switch to improve reliability and diagnosability in the system. The research can be extended into analyzing the data bandwidth, hardware and software requirements of TDMP if input queues or buffers with arbiters are added to the processing elements. This corresponds to making the processing elements into multiple input data flow operators. Research into dynamically allocating multiple time slots to PEs can be investigated. This improves the time division multiplex bus bandwidth utilization because unused time slots caused by idle or non-communicating PEs can be allocated to PEs that need to transmit or receive data more frequently. Research into the modularity of TDMP can be further investigated. By employing the concepts of time multiplex switching and pipelining larger TDMP systems can be developed from simpler TDMP systems to solve larger problems. And, finally, further research into the

development of higher level programming languages are needed. The underlying trouble with most current attempts to use parallel hardware is that they are based on traditional concepts of programming. These concepts in turn are based wholly on the serial von Neumann computer design, with instructions executed one at a time. In view of the nature of parallel hardware systems and the practical difficulties of keeping them running at full speed, we suggest further investigation of data flow programs based on the TDMP computer model.

REFERENCES

REFERENCES

1. R. Sugarman, "Superpower Computers," IEEE Spectrum, April 1980.

2. C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley Publishing Company, Inc., 1980.

3. K. Hwang, Computer Arithmetic, John Wiley and Sons, Inc., 1979.

4. S. S. Reddi and E. A. Feustel, "A Conceptual Framework for Computer Architecture," Computing Surveys, Vol. 8, No. 2, June 1976.

5. A. Mohsen, "Device and Circuit Design for VLSI," Caltech Conference on VLSI, January 1979.

6. G. A. Jullien, "Using ROM Arrays to Implement Computer Arithmetic," IEEE Computer Arithmetic, 1979.

7. C. Mead, "VLSI and Technological Innovation," Caltech Conference on VLSI, January 1979.

8. R. Benard, "Computers: Emphasis on Software," IEEE Spectrum, January 1980.

9. Intel Corporation, 8086 Family User's Manual, October 1979.

10. Intel Corporation, 8087 Co-processor, private communication, May 1980.

11. R. Friedenson, "RC Active Filters for the D3 Channel Bank," Memorandum for File, Bell Labs, December 1971.

12. K. Thurber, "Associative and Parallel Processors," Computing Surveys, Vol. 7, No. 4, December 1974.

13. J. P. Hayes, Computer Architecture and Organization, McGraw-Hill Book Company, pp. 209-236, 1978.

14. W. Wulf and C. Bell, "Cmmp--A Multi-Mini-Processor," Fall Joint Computer Conference, pp.765-777, 1972.

15. F. Briggs, K. S. Fu, K. Hwang and J. Patel, "PM4--A Reconfigurable Multiprocessor System for Pattern Recognition and Image Processing," National Computer Conference, pp. 225-265, 1979.

16. W. Wulf and S. Harbison, "Reflections in a Pool of Processors--An Experience Report on Cmmp/Hydra," National Computer Conference, pp. 939-950, 1978.

17. J. R. Jump and S. Ahuja, "Effective Pipelining of Digital Systems," IEEE Transactions on Computers, Vol. C-27, No. 9, September 1978.

18. C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," Computing Surveys, Vol. 9, No. 1, March 1977.

19. G. H. Barnes, et al, "The Illiac IV Computer," IEEE Transactions on Computers, Vol. C-17, pp. 746-757, August 1968.

20. K. J. Thurber, "Parallel Processor Architectures," Computer Design, pp. 89-114, February 1979.

21. K. T. Kung, "Let's Design Algorithms for VLSI Systems," Caltech Conference on VLSI, January 1979.

22. A. L. Davis, "A Data-Driven Machine Architecture Suitable for VLSI Implementation," Caltech Conference on VLSI, January 1979.

23. B. W. Arden and H. Lee, "Analysis of Chordal Ring Network," Workshop on Interconnection Network for Parallel and Distributed Processing, April 1980.

24. M. A. Franklin, "VLSI Performance Comparison of Baynan and Crossbar Communications Networks," Workshop on Interconnection Networks, pp. 20-28, April 1980.

25. F. Briggs, PM4, Private communication, May 1980.

26. H. J. Smith and D. S. Smith, "Study of Multistage Interconnection Networks," Proceedings of the Fifth Annual Conference on Computer Architecture, April 1978.

27. D. M. Dias and J. R. Jump, "Analysis and Simulation of Buffered Delta Networks," Workshop on Interconnection Networks, April 1980.

28. M. C. Pease, III, "The Indirect Binary n-Cube Microprocessor Array," IEEE Transactions on Computers, Vol. C-26, No. 5, May 1977.

29. R. G. Hintz and D. P. Tate, "Control Data STAR-100 Processor Design," proceedings of 6th Annual IEEE Computer Society International Conference (COMPCON '72), San Francisco, CA, pp. 1-4, September 1972.

30. R. L. Sites, "An Analysis of the Cray-1 Computer," Computer Design, January 1979.

31. M. Satyanarayanan, "Commercial Multiprocessing Systems," IEEE Computer Society, May 1980.

32. A. E. Joel, Jr., "Digital Switching--How it has Developed," IEEE Transactions on Communications, Vol. C-27, No. 7, July 1979.

33. H. J. Siegel, "Interconnection Networks for SIMD Machines," Computer, pp. 57-65, June 1979.

34. D. Tabak and G. J. Lipovski, "Move Architecture in Digital Controllers," IEEE Journal of Solid State Circuits, Vol. SC-15, No. 1, February 1980.

35. J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," Computer Architecture, pp. 126-132, 1975.

36. J. Gurd and I. Watson, "Data Driven System for High Speed Parallel Computing - Part 1: Structuring Software for Parallel Execution," Computer Design, June 1980.

37. J. B. Dennis, "First Version of a Data Flow Procedure Language," Project MAC, MIT Cambridge, MA, 1974.

38. J. Rumbaugh, "A Data Flow Multiprocessor," Transactions on Computers, 1977.

39. J. Gurd and I. Watson, "Data Driven System for High Speed Parallel Computing - Part II: Hardware Design," Computer Design, July 1980.

40. M. Malek and W. W. Myre," Figures of Merit for Interconnection Networks," Proc. of Workshop on Interconnection Networks, April 1980.

41. H. Levin, "Introduction to Computer Analysis: ECAP for Electronic Technicians and Engineers," Prentice-Hall, 1970.

42. T. J. Schriber, "Simulation Using GPSS," Wiley, 1974.

43. P. A. Bobillier, B. C. Kahan, and A. R. Probst, "Simulation with GPSS and GPSS V," Prentice-Hall, 1976.

44. G. A. Blaauw, "Digital System Implementation," Prentice-Hall, Inc., 1976.

45. L. Gilman and A. J. Rose, "APL An Interactive Approach," John Wiley and Sons, Inc., 1974.

46. Control Data Corporation, "APL Version 2 Reference Manual," 1979.