AN ABSTRACT OF THE THESIS OF

CHULIT MEESAJJEE for the degree DOCTOR OF PHILOSOPHY
(Name of student) (Degree)

in Electrical and Computer Engineering presented on 12-11-74
(Major department) (Date)

Title: PRELIMINARY DESIGN OF THE RTAF COMMAND AND CONTROL TWO LEVEL PARALLEL COMPUTING SYSTEM

Abstract approved: Redacted for Privacy
Dr. W R. Adrion

This paper investigates the benefit of various parallel processing architectures for a Command and Control system for the Royal Thai Air Force. Parallel processing has been shown to be useful for air defense and air traffic control applications. Its advantages are examined within the constraint imposed by the available resources of a developing nation.

Several alternative types of architecture including array parallel processing, pipeline processing, associative processing, multiprocessing, and computer network are examined and summarized. The consideration criteria are based on cost, performance, reliability and flexibility.

A system architecture based on a generalized multiprocessor configuration is proposed. This system has a second level of parallelism within each processor by providing a number of

independent functional logical and arithmetic units in the processing unit. It is modular in design and hence economical, adaptable, and expandable. Therefore, the basic system cost will be within the financial constraints.

To obtain the optimum designed configuration and to find the limitations of the proposed model, the necessary system parameters such as processing units (P), memory modules (M), memory cycle time, rate of memory reference, are introduced. The memory conflicts (queue statistics) average utilization of a memory module (facility statistics), and other statistics parameters are measured by simulation. The relative system cost is evaluated and the system efficiency curve is plotted against the corresponding memory module utilization to determine the optimum operating configuration.

To illustrate command and control execution, a typical program which solves N simultaneous non-linear equations in N unknowns by the Newton-Raphson iterative procedure is selected as an example. The algorithm of Ramamoothy and Gonzalez is applied for recognizing the parallel processable tasks of the selected FORTRAN program as the system compiler and recognizer. The connectivity matrices corresponding to the analyzed program are illustrated and the parallel processable task tables are constructed.

As an example of second level parallel processing, the necessary PU instructions are proposed and their corresponding

execution time are defined. A number of FORTRAN statements related to the selected program are assumed to be executed by one of the processing units (PU) in the system. A set of machine instructions equivalent to those FORTRAN statements are derived and a long hand simulation performed. The time of both concurrent execution and sequential execution are computer and compared. A number of intermediate ratios of concurrent execution time to the corresponding sequential execution time for the simulated program are computed and plotted as a function of the number of executed machine instructions.

Preliminary Design of the RTAF Command and Control Two-level Parallel Computing System

by

Chulit Meesajjee

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

June 1975

APPROVED:

Redacted for Privacy

Assistant Professor of Electrical and Computer Engineering
in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented 12-11-74

Typed by Ilene Anderton and Lyndalu Sikes for Chulit Meesajjee

ACKNOWLEDGEMENTS

The writer is indebted to Dr. W. R. Adrion, the major professor, for his helpful advice, encouragement and continued assistance throughout all phases of this study. He also wishes to thank the Graduate Doctoral Committee for their valuable criticism on the progress reports of this research work.

He extends his appreciation to the Oregon State University Computer Center Staff especially Mr. Jeffrey Ballance, Manager Operating Systems Development, for providing computing time and useful explanations during the simulation phases of this research, to the faculty members of the Department of Electrical and Computer Engineering for their wise suggestions and cooperation, and to the 1974 Sagamore Computer Conference Committee for their consideration of this research work.

Acknowledgement is made to His Majesty's Government of Thailand, (RTAF) for her financial support and to the Royal Thai Air Force Committee for their approval of this course of study.

Finally, he especially wishes to thank his parents for their constant concern during this study, and Mrs. B. Stroup, for carefully reading the original copy.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDIX TABLES

LIST OF SYMBOLS

A	Address register
A_i	Address register (i)
AM	Associative memory
AP	Associative processor
a	Number of bits, a specific field in a register
B	Index register
B_i	Index register (i)
BR_i	Buffer register (i)
b	Number of bits, a specific field in a register
CAU	Configuration assignment unit
CDC	Control Data Corporation
CFR	Configuration control register
C_B	Base cost
C(I)	Relative cost
$C_N(I)$	Normalized cost
DRU	The I/O device reference unit
F	Instruction field denotes the major class of function
FF	Electronic flip-flop
FN	Function unit
GPS III	General purpose system simulation III, IBM
HBS	Horizontal bus switch

I	M/P, the memory-processor ratio
I_i	Element (i), instruction register in the instruction stack
I/O	Input/output
i	Designation for register identification; subscribe
j	Designation for register identification; subscribe
K	Branch address for long instruction in CDC 6600
K_H	Lumped cost of other hardware (PPU, Crossbar switch, I/O device, etc)
K_M	Cost of a memory module
K_P	Cost of a processor unit
k	Designation for register identification; subscribe
M	Memory module
MAR	Memory address register
MC	Memory cycle time
MDB	Memory data buffer
Mod 8	Modulus 8
MPCU	Micro-program control unit
MRU	Memory module reference unit
M_i	Element (i) of memory address hopper register
N	Specific defined variable
n	Specific defined variable
ns	Nanosecond (or nsec.)
OS-3	Oregon State University Operating System Open Shop

P	Number of processor
PFF_{ji}	Priority flip-flop indicate that PU_j has the highest priority to access memory M_i.
PRU	Processor reference unit
PU	Processing unit
RFF_j	Request flip-flop of processing unit
RIS	Ready for issuing register
RTAF	The Royal Thai Air Force
SFF_{ji}	Switching flip-flop accomplishes the interconnection
T	Signal period
t	Time or time step
VBS	Vertical bus switch
X	Operation register
X_i	Element (i) of operation register
λ	Mean rate of arrival of memory request in the system simulation model
μsec	Microsecond

PRELIMINARY DESIGN OF THE RTAF COMMAND AND CONTROL TWC LEVEL PARALLEL COMPUTING SYSTEM

I. INTRODUCTION

Thailand occupies a strategic position within Southeast Asia. Bangkok International Airport has played the main role of being both Commercial Air Traffic center and Military Air Traffic center for Thailand. To ensure her security and economic development, Thailand needs a high efficiency Air Defense and Air Traffic Control System. Examples of modern systems, are SAGE (Semi-Automatic Ground Environment) for United States and Canadian fighter and missle defenses, BMEWS (Ballistic Missile Early Warning System), in Greenland and Alaska for detecting the launch of Russian missiles and New York International Airport Air Traffic control system. The efficiency of these real time systems depends on the efficiency of the command and control computer at the heart. A command and control computing system requires more function capabilities than a conventional commercial or scientific system. Requirements include Availability: a function of hardware reliability and maintainability; Adaptability: the ability to be dynamically and automatically restructured to a working configuration responsive to the problem-mix environment; Expansibility: expansion without incurring the cost of providing more capability than is needed at one time.

One possible way to provide the functional requirements of a command and control computing system is through parallel processing. One early command and control computer, the Burroughs D 825 employed multiprocessing. This provided parallism at the program or task level. Due to the advance of LSI technology the cost considerations for parallelism at the instruction level make it attractive. This research proposes to design a two level parallel processing system with a multiprocessor architecture at the program and task level and appropriate functional units within each processor to achieve parallelism at the instruction level.

The multiprocessor architecture has been shown to meet the criteria of availability, adaptability and expandability. The increased speed obtained through a second level of parallelism should meet the response needs of a modern command and control system. The research proposed will fall into three main categories. The first will be a survey of parallel processing research as it applies to the design of a command and control system. The second area of work will be the design and specification of the system architecture. Drawing on the experience gained from surveying existing systems, it is hoped that a cost effective combination of task level and instruction level parallelism can be chosen. Problem areas such as programming and operating systems will be outlined. A detailed design for processor subsystems to perform in an efficient

highly parallel manner will be given. The final area will consist of an evaulation of the proposed architecture. Both levels of parallelism will be simulated for a specific algorithm.

To make this system design more relevant to the needs of the Royal Thai Air Force, a detailed study of the air traffic control algorithms used at Bangkok International Airport and their implementation on the proposed system would need to be performed. This is, however, out of the scope of the research presented here. It is hoped that the preliminary design will serve as a first step in finding a satisfactory command and control system for Thailand.

II. SURVEY OF PARALLEL PROCESSING

The first-generation of electronic digital computers was introduced in early nineteen fifties and used vacuum tube technology. They were very slow, when compared to current computers, being capable of performing no more than several hundred arithmetic operations per second. Input-output was performed by paper-tape and punched-card readers, and the computer itself had to wait while each character was being transferred into or out of the machine. The second-generation computer used solid state transistor technology to replace vacuum tubes. These machines were physically smaller and their computationally faster than the older computers, being capable of executing over 50,000 arithmetic operations per second. Many of the input-output devices introduced in the computer system, and an early form of input-output channel was used to speed up overall operation.

The third-generation computer is characterized by solid state integrated circuit technology. Size again decreased, whereas speed increased to over one million arithmetic operations per second. Input-output control systems using multiple I/O channels, memory sharing, and an interrupt system were provided. The modern

third generation computer system is block diagrammed in Figure 2. 1. The difference between the first two generations and the third-generation computer is that the devices whether for input or output, do not report directly to the memory unit but are reached through the channel controller. The channel controller (CC) is a subsystem which acts autonomously by referring to a list of sub-commands stored in main memory. In this sense, it is like a small computer. Since it has its own program in memory, and it has a control unit (CC control) and is attached to I/O devices which supervise. In some systems, separate computers were used to control I/O functions; these computers were called satellites for, although they were autonomous, they were under the regulation of central computer.

The communication between main-memory unit and a channel controller is done by cycle stealing. This task is contrasted with earlier, simpler techniques for I/O communication. When information was transmitted between an I/O device and the memory unit, the rest of the computer was immobilized. No other references to memory were possible as long as an I/O device was activated. The principle of cycle stealing permits information to accumulate while useful work is being done by the processor and control. Information is transferred between channel controller and memory, without awareness of control unit. In this sense, the channel controller "sneaks" into memory and steals a memory cycle.

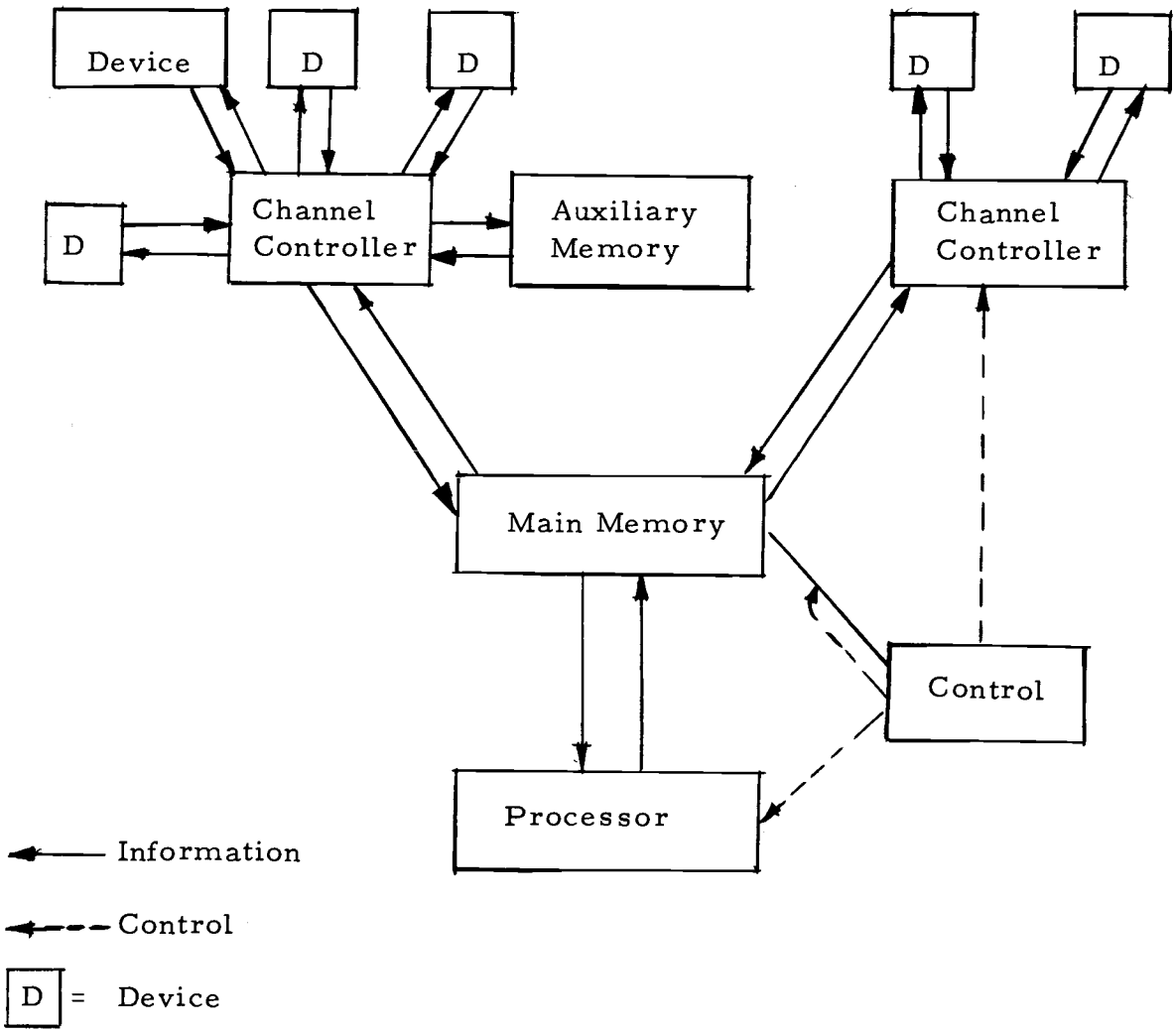


Figure 2.1. Third-generation computer system.

It is extremely difficult to increase the speed of operation of a single computer beyond ten million arithmetic operations per second. Therefore, the fourth-generation computers now being developed and constructed often use some form of multiprocessing or parallel processing to achieve higher speed. This type of machine is called parallel computer and is discussed next.

The speed of conventional computers has increased with the technology of both hardware and software. The transition from vacuum tubes to transistors and from discrete components to integrated circuits has increased the speed of the arithmetic unit. It is certainly possible to speed up execution without an increase in raw (hardware) speed by determining what impact local improvements or different architectures might have. In the balance of this chapter, techniques for local optimization, parallel computer structures and software implications will be surveyed.

2.1 Techniques for Parallelism in a Single Processor System

The basic processing function of a computer is usually divided into subfunctions: instruction fetch, instruction decode, operand fetch and execution. In order to control the memory access time - execution time imbalance, an amount of parallelism or overlap of these subfunctions must be achieved.

In most machines, the subfunctions above are performed by distinct hardware sections. Instruction fetch and decode is implemented by an instruction counter, control logic and memory interface logic (I unit). Operand fetching and the actual instruction execution is performed by a processing (P) unit. The execution of a user program would consist, then, of a series of operations by the I unit, P unit and the memory (M) unit.

Earlier computers were usually single address machines with memory references possible from P unit. The I and P units shared some register interface to memory. This was a bottleneck of the system. To achieve high performance frequent memroy references must be avoided. Both I and P units should have local stores (a small collection of registers) in which they operate much more quickly than main memory. Register to register operations require less time than memory reference operations.

The rate of execution of a uniprocessor system or a single processing unit can be increased by using an instruction look-ahead feature that overlaps the fetching of the next instruction with execution of a current instruction. The concept of instruction look-ahead is illustrated in Figure 2.2.

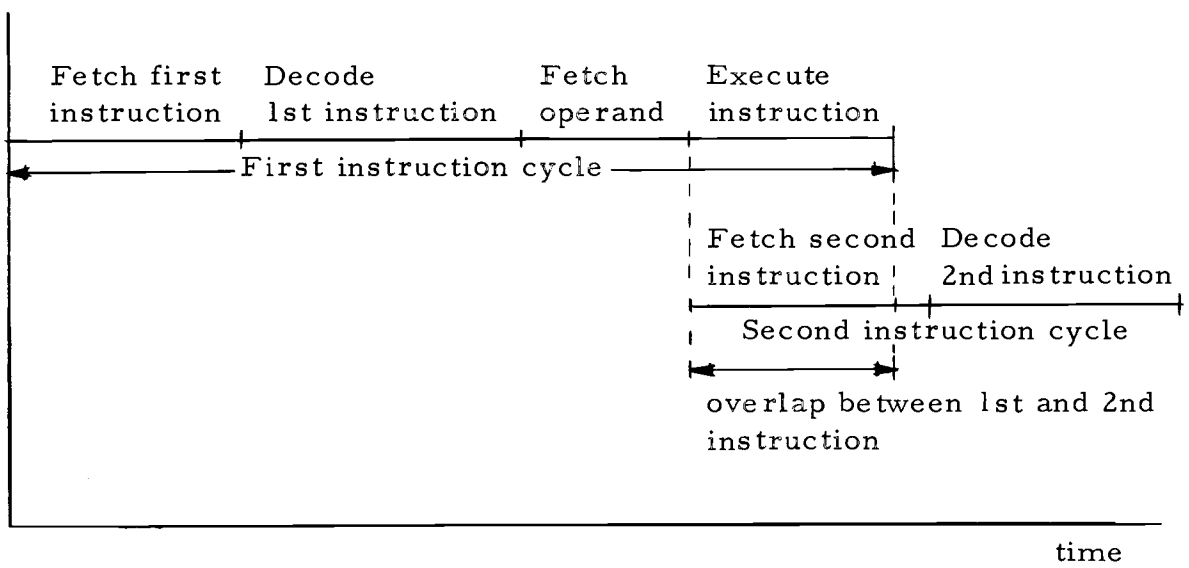


Figure 2.2. Instruction look-ahead diagram.

Instruction look-ahead can be extended to the instruction stack concept introduced in many current high speed uniprocessor machines. For example, an eight-instruction buffer is interposed between I unit and M unit so that whenever I references M, eight instructions will be transferred. One can see that the total time to fetch eight instructions is reduced because of the saving of address deliveries on seven instruction acquisitions. The presence of eight instructions in the Instruction buffer not only provides a repository of instructions to be executed, but, as the system advances through the buffer, it represents a collection of instructions just recently executed. If an instruction towards the end of the buffer transfers control to an instruction near the top of the buffer, then further fetching can be avoided. This concept is called instruction "look-aside".

Another scheme for improving the instruction fetch is to store multiple instructions in a single word, possibly requiring a wider data bus. The disadvantage lies in the constraint imposed on the instruction set and addressing capability through compaction of the instruction word size. The alternative of increasing the memory bandwidth leads to increased hardware costs.

The concept of providing the I unit with local storage increases the instruction delivery rate. Now, a serious imbalance problem between I and P is encountered. The designer must reconsider the

relationship and investigate what can be done to improve the rate of instruction execution.

It is reasonable to adjust the rate by further increasing the potential parallelism of the machine. There are two basic approaches to speeding up the P unit without increasing raw power. One is to increase the parallelism within the unit by possibly increasing the number of function units with the processor. The other scheme is to increase the execution throughput of the machine by employing a number of processors.

A significant scheme for balancing the relation between I and M is through memory interleaving. One can see that a single port M unit when busy serving a request will lock out all other requests. In current large computer systems, these memory delays are reduced by dividing the total memory into "banks" of storage, where each bank contains a subset of the memory address. This is shown in Figure 2.3. The resultant M unit has multiple data ports to reduce conflicts and increase access time.

Memory access time can be improved at both I unit and P unit interfaces by employing a high speed cache memory. Such a memory would be implemented with LSI technology and have access speeds comparable to the execution and decode rates. To overcome the cost problem, the cache memory is kept small and some scheme like paging used to transfer data from the Central M unit to the cache memory.

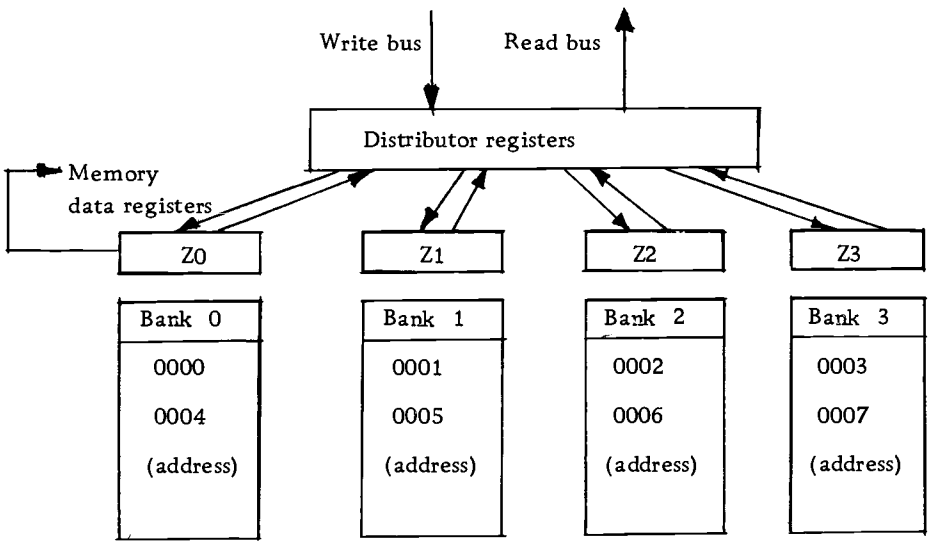


Figure 2.3. Example of interleaving memory (CDC 6600).

In summary, the following techniques have been employed in commercial single processor systems to increase the overall system throughput:

1. Look-ahead technique.
2. Look-aside (Instruction buffer, Instruction stack).
3. Multiple instruction per word, multiple instruction fetch.
4. Multiple functional units within the processor.
5. Wider memory bandwidth.
6. Memory interleaving.
7. Cache memory.

Some of these techniques will be implemented in the design of the command and control system.

2.2 Survey of Parallel Processors

A number of different parallel organizations have appeared in the literature to date. These can reasonably be classified into four categories:

1. Single instruction stream with single data stream (SISD).
2. Single instruction stream with multidata stream (SIMD).
3. Multi-instruction stream with single data stream (MISD).
4. Multi-instruction stream with multidata stream (MIMD).

In 1964, J. C. Muitha and R. L. Beedle classified parallel processor systems into three broad groups which can be described as follows:

1. General purpose network computers. This group can be subdivided into two subgroups; parallel networks with common control and parallel network with independent identical processor elements.
2. Special purpose network computers. These are subdivided into two subgroups; pattern recognition processors and associative processors.
3. Nonglobal computers. Each module is only semi-independent (i. e. locally parallel). In addition this group

is used for other organizations that do not fall into the first two categories.

A third approach should be mentioned which is less formalistic, but more common and includes five categories:

1. Machines with instruction overlap capability and distributed I/O control. This category includes most of the first attempts at implementing some form of parallelism. Although this organization permits different types of operations to be performed concurrently, it is essentially an SISD architecture where parallelism is achieved operation tuning. Examples include the CDC 6000 series machines.
2. Multicomputers or multiprocessors. Computer systems which fall into this category consist of more than one processor or complete conventional computers. Both symmetric and asymmetric organizations are seen. The main processor memory modules are usually shared by all processors. Some type of memory reference conflict resolution mechanism and automatic crossbar switch must be provided in the system. The Burroughs D 825 and Carnegie-Mellon University C.mnp computer systems fall in the category. It is clear that this type of machine is a MIMD system.

3. Array processors. This type of parallel processor consists of a number of processing elements (PE) and associated memories. These identical processors are connected in some kind of network, frequently a matrix array. Each PE can usually communicate with its immediate neighbors directly and in a restrictive fashion, all the other PE's. Processors in the system are controlled by some central array control unit. SOLOMON, ILLIAC IV, the Honeywell PEPE, and the Texas Instruments SIMDA computers fall in this category. The machine physical structure falls into SIMD category.
4. Pipe-line processors. An alternative way of increasing computing speed is to divide the conventional arithmetic and logic unit into a number of functionally independent subunits which can be operated autonomously by the control unit. Each is time division multiplexed doing, for example, one part of a task, passing its partial result to the next module, with all modules operating concurrently. The CDC-STAR and Texas Instruments ASC fall in this category. There are two types of pipe-line processor based on the nature of data flow. One is the flushed pipeline, where data moves completely through the pipe-line before the next data enters the pipeline. The second is unflushed

pipeline, where data moves continuously through the pipeline.

5. Associative processors. These systems are based on utilizing an associative memory and providing processing logic at each cell or word location. An associative memory (AM) is a device capable of retreiving stored data by means of testing part or all of the contents of each memory word simultaneously (by hardware means, mask register operation) in order to find one or more desired words. Associative memory is often called content addressed memory. An associative processor performs one operation on N operands simultaneously. Successive bit positions of all words are sequentially passed through the PE's at each word position. These PE's are driven by a central controller. The operation is serial by bit, but parallel by word and within this context behaves like an array processor and can be categorized as a SIMD system. The Goodyear STARAN is a typical associative processor.

The various architectures discussed above are summarized in Figure 2.4. In succeeding sections, typical commercial systems will be surveyed.

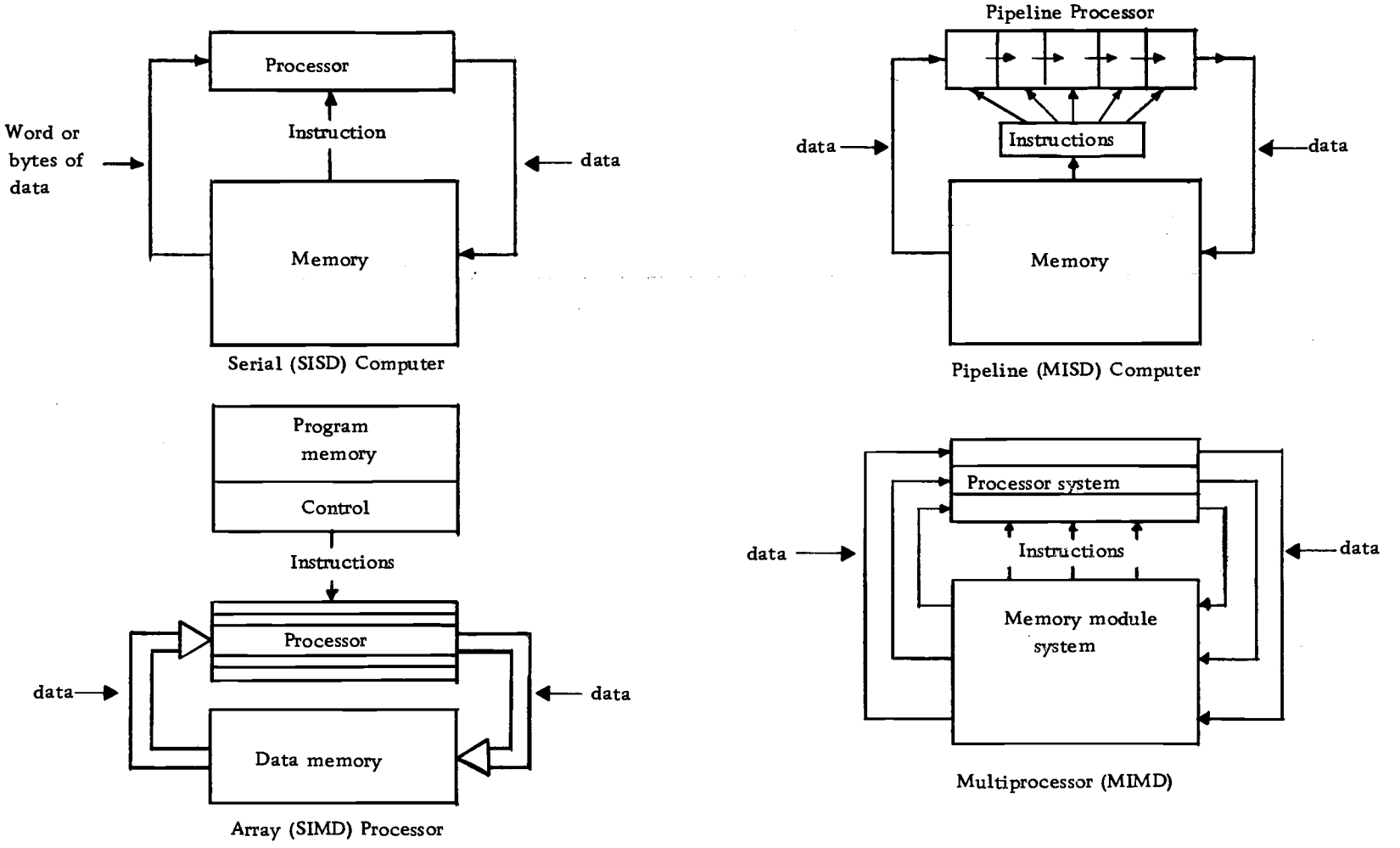


Figure 2.4. Four computer architectures.

The Control Data 6600

The CDC-6600 was the fastest and most powerful single processor machine of its generation. It has one high-speed central processing unit (CPU) and 10 peripheral processing units (PPU). It has up to 32 banks of 1 μsec. core storage, each containing 4096-60 bit words. This computing system was designed with special attention to kinds of use, the very large scientific problem and the time sharing of smaller problems. To handle the large problem, a high-speed floating point central processor with access to a large central memory was obvious. The important idea of the CDC-6600 system was the isolation of this central high speed arithmetic from any peripheral activity. The idea of a multiplicity of peripheral processors is introduced in this system (Figures 2.5 and 2.6). Ten such peripheral processors have access to the central memory on one side and the peripheral channels on the other. Each of the ten peripheral processors contains its own memory for program and buffer areas. All ten peripheral processors have access to twelve input-output channels and may "change hands", monitor channel activity, and perform other related jobs. These processors have access to central memory, and may carry on independent transfers to and from this memory.

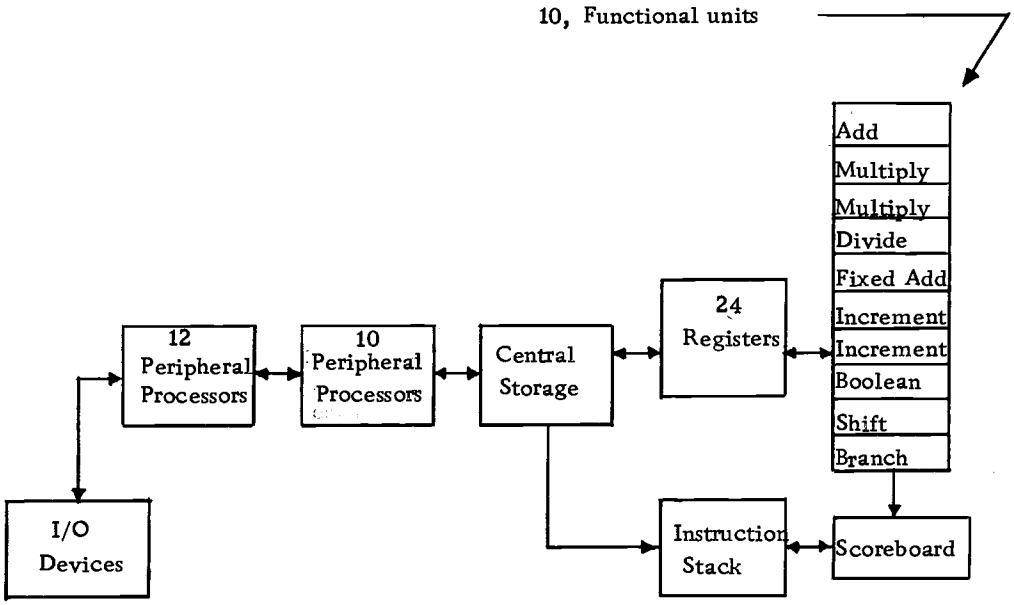


Figure 2.5. CDC-6600 central processor block diagram.

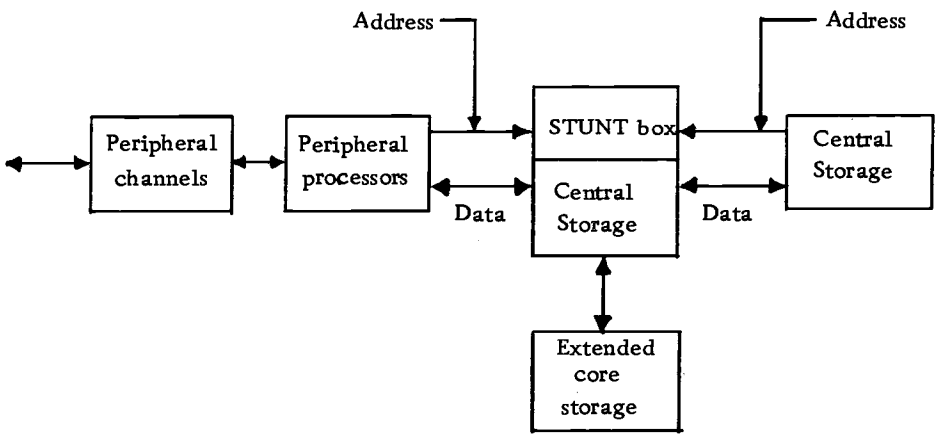


Figure 2.6. CDC-6600 central storage block diagram.

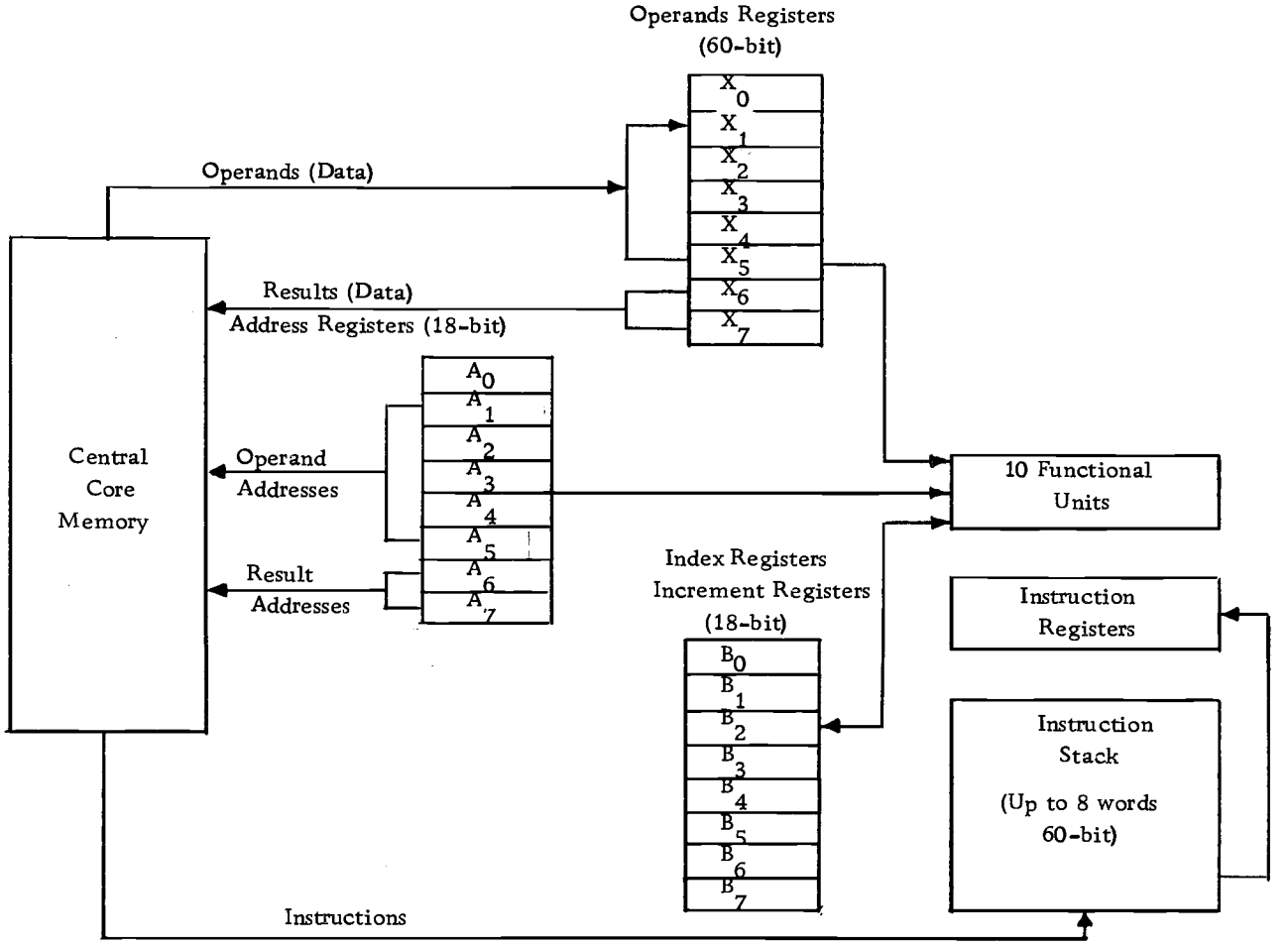


Figure 2.7. Central processor operating registers.

The Central Processing Unit is the key to overall system performance. It is operated as a slave processor to the PPU's and has no I/O capability. Multiple functional arithmetic and logical units with overlap execution capability are provided to achieve high system performance. Several local storage registers are provided, 8 operation registers (X_i, 60 bits each), 8 index registers (B_i, 18 bit each) and 8 address registers (A_i, 18 bit each) to best utilize the overlap potential.

Data and instruction transfer between the Central Storage and the CPU occurs on a number of separately controlled paths. Five of the 60-bit operation registers, (X1, X2, X3, X4, and X5), are assigned as read registers and two , (X6, X7), as store registers (see Figure 2.7). This reflects the typical unbalance of traffic between read and write. Address registers are assigned one-for-one with each of the read registers (A1, A2, A3, A4, and A5), and store registers, (A6, A7). The specified address register is set to the desired address by a CPU instruction. This new address is used to reference storage for a read or a write depending on which register was set (A1-A5 causes a read and access A6-A7 causes a write). The data will enter or leave the operand register in a partner relationship with the address registers, as shown in Figure 2.7). Data and transfer between Central Storage and CPU pass through a set of 60-bit operand registers and their corresponding address

have to pass through a set of address registers. Instruction transfer is accomplished via the instruction stack and fetching mechanism (see Figure 2.8). The CPU is essentially hidden from Central Memory by these operand registers (X), address registers (A) and the instruction stack.

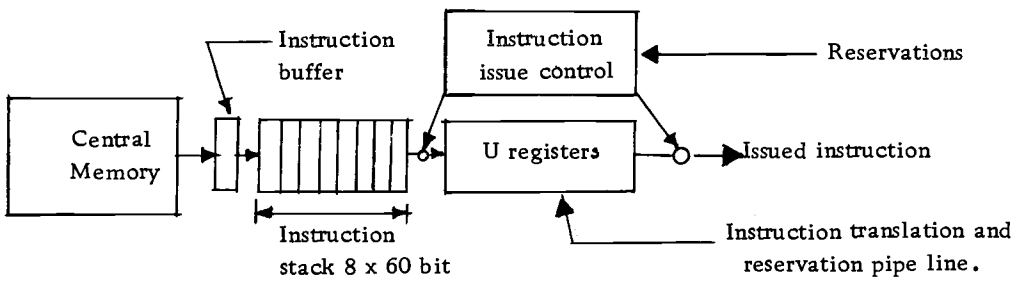


Figure 2.8. CDC 6600 instruction stack and instruction issuing diagram.

To provide a relatively continuous source of instructions, a 60-bit buffer register is associated with an instruction stack capable of holding 32 instructions (see Figure 2.8). Instruction words from memory enter the bottom register of the stack pushing up the old instruction words. In a straight line program (no looping), only the bottom two registers are used, the bottom being refilled as quickly as memory conflicts allow. In programs containing looping, which branch back to an instruction in the upper stack registers, no refills are allowed after branch, holding the program loop completely in the

stack. In this manner, memory access is no longer required, and a considerable speed increase can be obtained.

A unique and essential part of the CDC 6600 Control Processor control is the Unit and Register Reservation Control or Scoreboard. The major premise of the Scoreboard design is that each new instruction be issued to its functional unit as early as possible in order to maximize overlap. The scoreboard maintains a running file of each central register, of each functional unit, and of each of the three operand trunks to and from a unit. The "scoreboard file" is made up of two-, three-, and four-bit quantities identifying the nature of register and unit usage. When a new instruction is brought up, the conditions at the instant of issuance are set into the scoreboard.

A so called "snapshot" is taken of the relevant conditions. If no waiting is required, the execution of the instruction is begun immediately under control of the unit itself. If waiting is required, for example, an input operand may not yet be available in the central registers, the scoreboard controls the delay, and when released, allows the unit to begin its execution.

Most importantly, this activity is accomplished in the scoreboard and the functional unit, and does not necessarily limit later instructions from being brought up and issued. In this manner, it is possible to issue a series of instructions, some related, some not,

until no functional units are available or until an operand or result register conflict is determined. Several independent chains of instructions may proceed concurrently. In the absence of conflicts, instructions may be issued every minor cycle (100 nsec.).

The Central Storage System has three essential features to match the high performance CPU. A synchronous interleaving storage technique is adopted. Central storage is divided into eight chassis modules, each module is subdivided into four banks of 4096, 60 bit-words. Memory interleaving is performed in each module (4 banks). The STUNT Box, a mechanism for referencing control storage, is provided to deliver addresses directly to each of the eight chassis. In principle this mechanism allows a new storage address to be delivered every minor cycle with any "rejected" addresses to be reissued repeatedly until accepted.

Ten peripheral processors (PPU's), are provided to handle I/O processing requirements, transferring data between peripheral device and central storage, controlling the initiation of peripheral device actions, establishing priorities between devices, buffering data between asynchronous devices, and interrupting the central processor for execution of priority tasks. The most interesting feature of the peripheral and control processors (PPU) design is what is called the Barrel (see Figure 2.9). Instead of each PPU having its own control and arithmetic unit, all ten of them share one fast

unit on a time division multiplex scheme. This common unit (the barrel) operates at a 100 nsec. (one minor cycle) clock rate. As the barrel "rotates", one PPU after another comes into the "slot", at which point it is connected to the processor. A "private memory" (4096, 12-bit word) reference is initiated and exactly 100 nsec. later, the memory cycle is finished and the PPU is back in the slot again ready to proceed. The barrel is actually 52 circular-shift registers, each 10-bits long, which are used to store the registers of the 10 PPU's (an 18-bit accumulator, 12-bit program address register and 22-bit instruction register). This technique relies on the synchronism between the barrel's "rotation velocity" and the length of the private storage memory cycle. If a particular PPU doesn't need a memory reference for some instruction, it still has to take a trip around the barrel before starting its next "fetch cycle". Some instructions require several trips around. However, this is an interesting idea that results in a considerable saving in hardware and maintains balance between "memory cycle time" and the "apparent control unit speed".

There are 12-bit and 24-bit-instruction formats and provision for direct, indirect, and relative addressing. The peripheral instructions provide logical, addition, subtraction, shift, and conditional branch. They also provide single word or block transfers to and from any twelve peripheral channels and single word or block transfer to

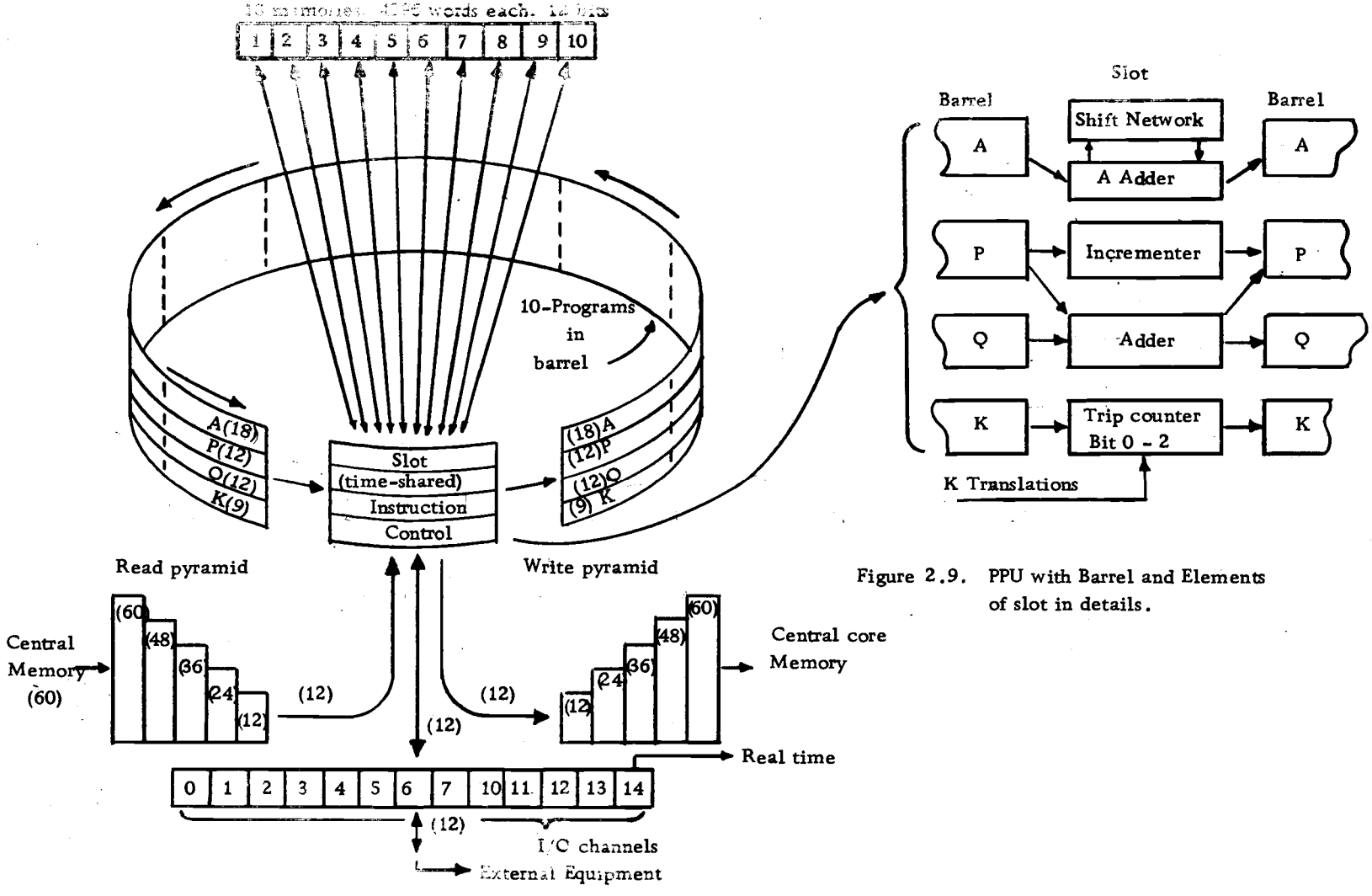


Figure 2.9. PPU with Barrel and Elements of slot in details.

and from central memory. Central memory words of 60-bits length are assembled from and disassembled to five consecutive peripheral words by read/write "pyramids". Each PPU has instructions to interrupt the CPU and to monitor the central program address.

The original Chippewa Operating System was constructed to facilitate high level language usage as well as to perform on-line diagnostic and maintenance programs. It also handled: extended core storage, Central Processor monitor, multiple system overlaps and segmentation.

The important features of the CDC 6600 computer system can be summarized as follows:

Advantages

1. Overlap instruction execution.
2. Instruction stack - capable of holding loop.
3. Interleaved memory.
4. CPU and I/O device PPU control system.
5. Control point structure.

Disadvantages

1. Non-paged or segmented requiring extended core storage for still inefficient for time sharing.
2. Non-optimal use by programmer.
3. High cost.

Burroughs D 825 - computer system

The D 825 was developed for operation in a military environment to meet the data processing requirements for a command and control system. The initial system, constructed for the Naval Research Laboratory with the designating AN/GYK-3(v), has been completed and tested. The system was proposed by James P. Anderson, Samuel A Hoffman, Joseph Shifman, and Robert J. Williams of Burroughs Laboratories (4). A block diagram of the D 825 system is shown in Figure 2. 10.

The current D 825 system processor consists of four identical, general purpose arithmetic and control units. It is designed to use a variable-length instruction format made up of quarter-word syllables (i.e., Zero, one-, two, or three address syllables as required) associated with each basic command syllable. An implied address accumulator stack is associated with its arithmetic unit. A 128-position, thin-film memory is used for the stack and also for many of the registers of the machine: program base register, data base register, the index registers, and limit registers.

The system consists of 16 independent memory modules of 4096 48-bit-words each. The size of the memory modules was established as a compromise between a module size small enough to minimize conflicts in which two or more computer of I/O modules

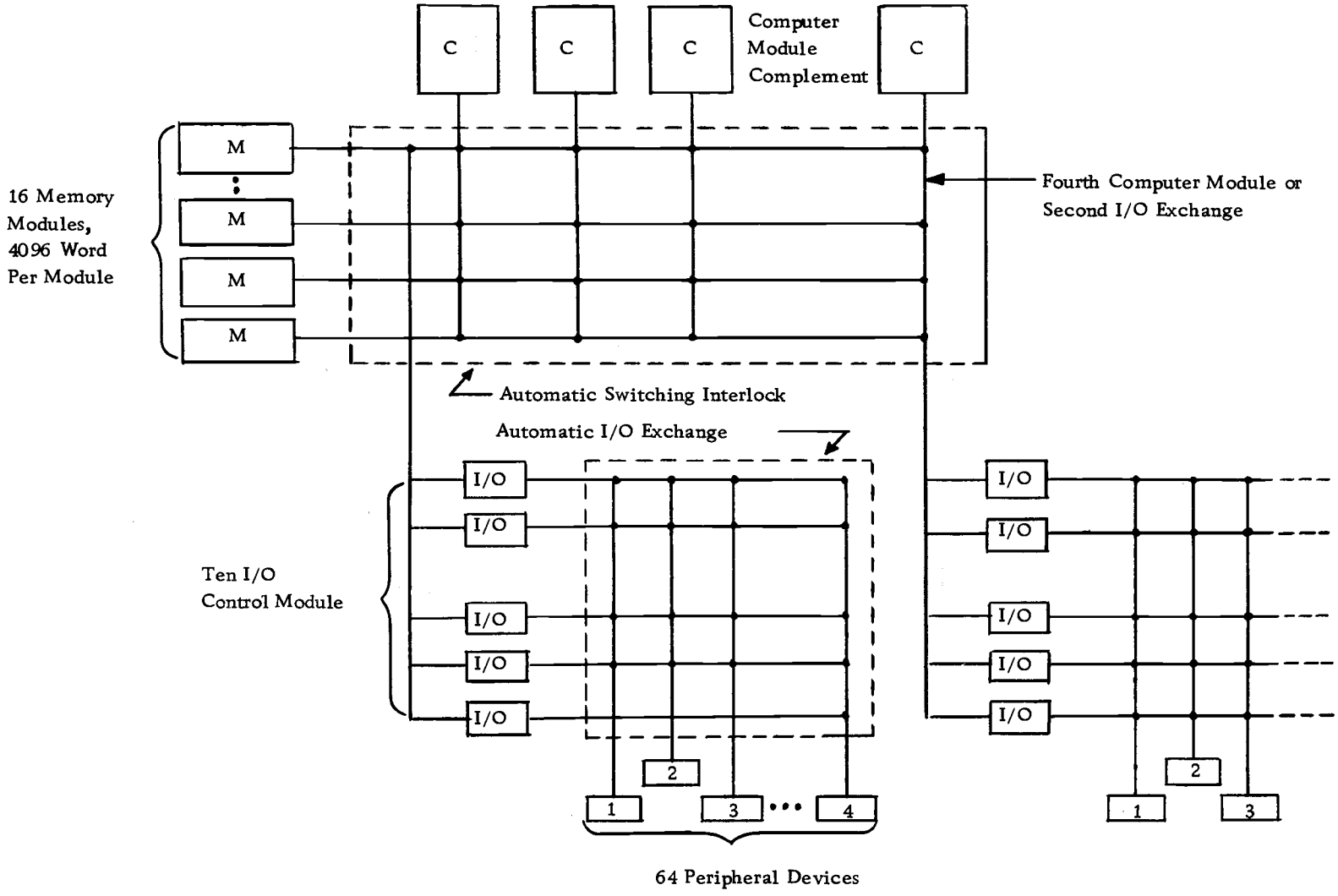


Figure 2.10. D825 System Organization.

alternate access to the same memory module, and size large enough to keep down the cost of duplicated hardware components.

The switch interlock is the most essential part of the system. This crossbar matrix switch provides the communication path to permit access to any memory module by any processor or I/O control module. The switching interlock has the ability to detect and resolve memory reference conflicts.

The priority scheduling function is performed by the bus allocator, a set of logical matrices. The conflict matrix detects the presence of conflicts during interconnection requests. The priority-matrix resolves the priority of each request. The logical product of the states of the conflict and priority matrices determines the state of the matrix, which influences the setting of the crossbar of switching interlock.

I/O control modules are provided to separate I/O routine functions from the computer module. In satisfying the system objectives, I/O control modules are not assigned to any particular computer module, but are treated in much the same way as the memory modules. Automatic resolution of conflicts via the switching interlock function is provided. The I/O action is started by the execution of a transmit I/O instruction in one of the computer modules, which delivers an I/O descriptor word from the addressed memory location to an inactive I/O control module. The I/O descriptor is

an instruction to the I/O control module selects the input-output device and determines the direction of data flow, the address of the first word, and the number of words to be transferred.

When an I/O operation is initiated, it proceeds independently until completion. The I/O control module employs another cross-bar switch as an interface between the I/O control modules and the physical external devices. This automatic exchange, similar in function to the switching interlock, permits two-way data flow between an I/O control module and any I/O device in the D 825 system.

The Automatic Operating and Scheduling Program (AOSP) is the keystone of D 825 system. Its major functions are configuration determination, memory allocation, scheduling, program readying, end-of-job cleanup, reporting, logging, diagnostic and confidence checking and external interrupt processing. This operating system handles retrieval of named objects from files, run-time allocation ofmemory, and I/O control. The primary AOSP function is to provide facilities for multicomputer multiprocessing; i.e., the use of one, two, or more processor modules by time sharing their arithmetic control functions between one or more object-programs under control of an executive system. In this fashion, not only may one program time-share a processor with other programs, but also a processor may time-share the execution

of a program with other processors. The programmer need not indicate that his program may be processed in parallel with other programs; the AOSP will automatically perform parallel processing of job entries as a matter of course. In case a large program involves major independent subtasks, specification of parallelism within the program may be made by the programmer. Therefore, a characterizing feature of the AOSP is that a program may introduce a parallel process. Another feature is that many processors may be (although none are aware of it) simultaneously executing the same set of instructions, referencing different data sets.

The Burroughs D 825 multicomputer system can be summarized as follows:

Advantages

1. The system structure was designed to satisfy the requirements of Command and Control data processing: availability, adaptability, expansonability and programming suitability.
2. This structural freedom is suitable for other types of high speed real-time problems.
3. The memory size (4096 48-bit word per module) was selected to obtain the minimum memory conflicts respect to economic cost.

4. I/O functions are separated from the main CP's and handled by I/O control modules.
5. System hardware in cooperation with the operating system (AOSP) can perform both job level and task level parallel processing.
6. Job level parallel processing is done automatically by AOSP without involving the programmer.

Disadvantages

1. The system cost is high.
2. Switch interlock is costly and complex in construction.
3. I/O control module not capable of independent operation.
4. High level language such as FORTRAN not available.

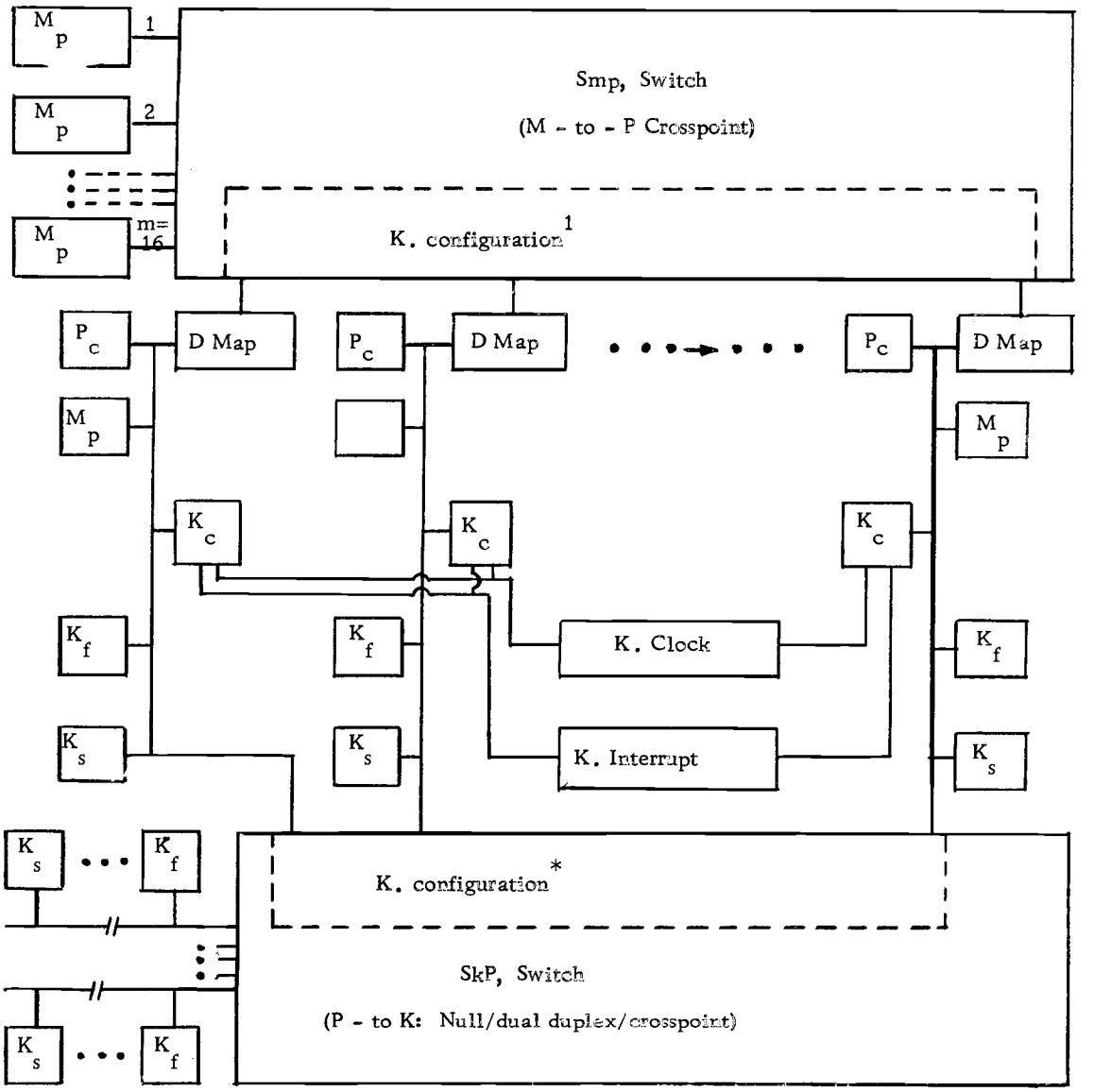
C. mmp-A multi-mini-processor

The C.mmp (Carnegie-Mellon University Multi-Mini-Processor) was proposed by William A. Wulf and C. G. Bell (3, 5) together with the students and staff of the Department of Computer Science at Carnegie-Mellon University (CMU). This project was to design the hardware and software for a multiprocessor computer system using PDP-11's, general purpose minicomputers as processors. The goals of the system design was to obtain an appropriate computing system, to study the characteristic and performance of multiprocessor systems including system software design and the feasibility of utilizing parallel

processing for various research projects (such as speech and pattern recognition) at Carnegie-Mellon University. The C. mmp block diagram is given in Figure 2.11.

The C. mmp consists of the following essential features:

1. Processing units (Pc) are modified versions of DEC PDP-11's general purpose minicomputers. Sixteen processor units are used, each able to execute operating system functions as well as user programs.
2. Memory modules (Mp) contain up to 65 K x 16 bits words and are 8-way interleaved. The system main storage consists of 16 such memory modules. Memory sharing is permitted in this system as in Burroughs D 825 computing system.
3. P&M crosspoint switch (Smp) is the matrix switch which handles information transfer between primary memory, processors and I/O devices. This switch consists of m buses for primary memories and p buses for processors. Up to min (m, p) simultaneous communication is possible through the crosspoint arrangement. The switch Smp can be set either under program control or manually, to provide different configurations. The Smp can be controlled by any of the processors, but only one processor is assigned the control.



Where: P_c / central processor; M_p /The shared primary memory; T/terminals;

K_s / slow device control (e.g., for Teletype);

K_f / fast device control (e.g., for disk memory);

K_c / control for clock, timer, interprocessor communication

*Both switches have static configuration control by manual and program control.

Figure 2.11. Proposed CMU multiminiprocessor computer/C.mmp.

4. Processors and I/O control switch (Skp) is provided to connect one or more of k Unibuses, the common bus for memory and I/O, to an isolated PDP-11 processor. The Unibuses employ several slow controllers (k) (i.e., teletypes, card, readers), and fast controllers (K_t) (i.e., disk, magnetic tape), to be connected to one of the p central processor units.
5. Data operations component (D map) is associated with each processor (Pc). It is used to convert the 16-bit PDP-11 addresses generated in the processor to 18-bit addresses for the Memory module and Unibuses.
6. Central clock unit (K.clock) allows precise time to be measured in the system. This clock is broadcast to all processors for local interval timing.
7. Interrupt Network (K.interrupt) is used to control interrupts for the C. mmp system. Every processor (Pc) is allowed to generate an interrupt to any subset of the Pc configuration at any of several priority levels. Every processor can also cause any subset of the configuration to be stopped and/or restarted. This interrupt capability is under both program and manual control. The console loading function is carried out in this task.

8. System software is, due to lack of experience in organizing computation for parallel execution, incomplete at the present. This operating system will consist of a kernel (called HYDRA) and a standard extension. The kernel will provide a set of structured subroutines for building an operating system. The standard extension will implement an easily modified set of convention operation, (i.e., a scheduler, file system, etc.). An appropriate set of subroutines in which HYDRA should provide includes:
   a. The creation of new virtual resources (e.g., providing filing system).
   b. Editing, program readying, end-of-job cleanup, reporting and logging and I/O control.
   c. Representation of a new resourse in terms of existing ones (e.g., providing virtual memory and paging: data management and memory allocator; system control structures or system configuration determination).
   d. The creation of operations on resources and/or their representation (e.g., providing scheduling).
   e. Protection against illegal operations on a resource both uniformly over a class resource and with regard to specific instances of a resource (i.e., providing

diagnostics and confidence checking, interrupt processing).

Advantages

1. One of the outstanding features, of this computer system is the decentralization of the data base and the computing facilities by employing a number of minicomputers and memory modules tied together in such a way that multiprogramming and multiprocessing of large data bases can be handled as well as a large and expensive computing system can do.
2. In the C. mmp, every processor can execute operating system function as well as user program. It provides more flexibility and reliability since if the assigned host computer is failing, any one or more of the remaining processors can automatically replace it. This is very important for high speed real time processing, such as radar tracking or air traffic control problems.
3. Exhibits task level parallelism.
4. Shared memory and page system are provided.
5. System can be reconfigured by partitioning to match the user program requirements.
6. The system provides the flexibility for easy future expansion to achieve more throughput by just adding another minicomputer.

Disadvantages

1. Executive functions of this system are pure software implementations with only a few hardware features.
2. System still inefficient for time sharing.
3. Non-optimal use by programmers. For example, if a program only uses one machine, that machine is a minicomputer which runs slower.
4. The operating system will be more complex when the system is larger especially when one minicomputer is not enough to handle the system executive program. At this point a large "host" computer may have to be introduced and this symmetric system is automatically becomes master/slave system.

The Solomon Computer

The SOLOMON (Simultaneous Operation Linked Ordinal Modular Network) proposed by Slotnick, Borch and McReynolds (35) was one of the first machines employing parallel processing. This "paper" machine was a parallel network computer under a central control unit. The system is based primarily on the principles of applied parallel[1] operations, thus allowing the speed to increase linearly

[1] Applied Parallelism is the property of a set of computations which enables a number of groups of identical operations within the

with number of processing units. Consequently, the objective of the SOLOMON structure is to be able to control the processing of a number of different data streams with a single instruction stream. The machine has four principle features:

1. A large rectangular array of processing elements (PE's) arranged in a 32 x 32 matrix, is controlled by a single control unit, so that a single instruction stream sequences the processing of many data streams.
2. Store addresses and data are common to all of the data processors, and are distributed on a common bus from the central control unit.
3. Limited local control at the PE level is obtained by permitting each element to enable or inhibit the execution of instructions in the common stream according to locally controlled tests.
4. Each processing element in the array can be communicated to its four nearest neighbors to provide data exchanges.

Figure 2.12 illustrates a simplified block diagram of the SOLOMON organization.

---

set to be processed simultaneously on distinct or the same data bases.

Natural Parallelism is the property of a set of computations that enables a number of groups of operations within the set to be processed simultaneously and independently on distinct or the same data bases.

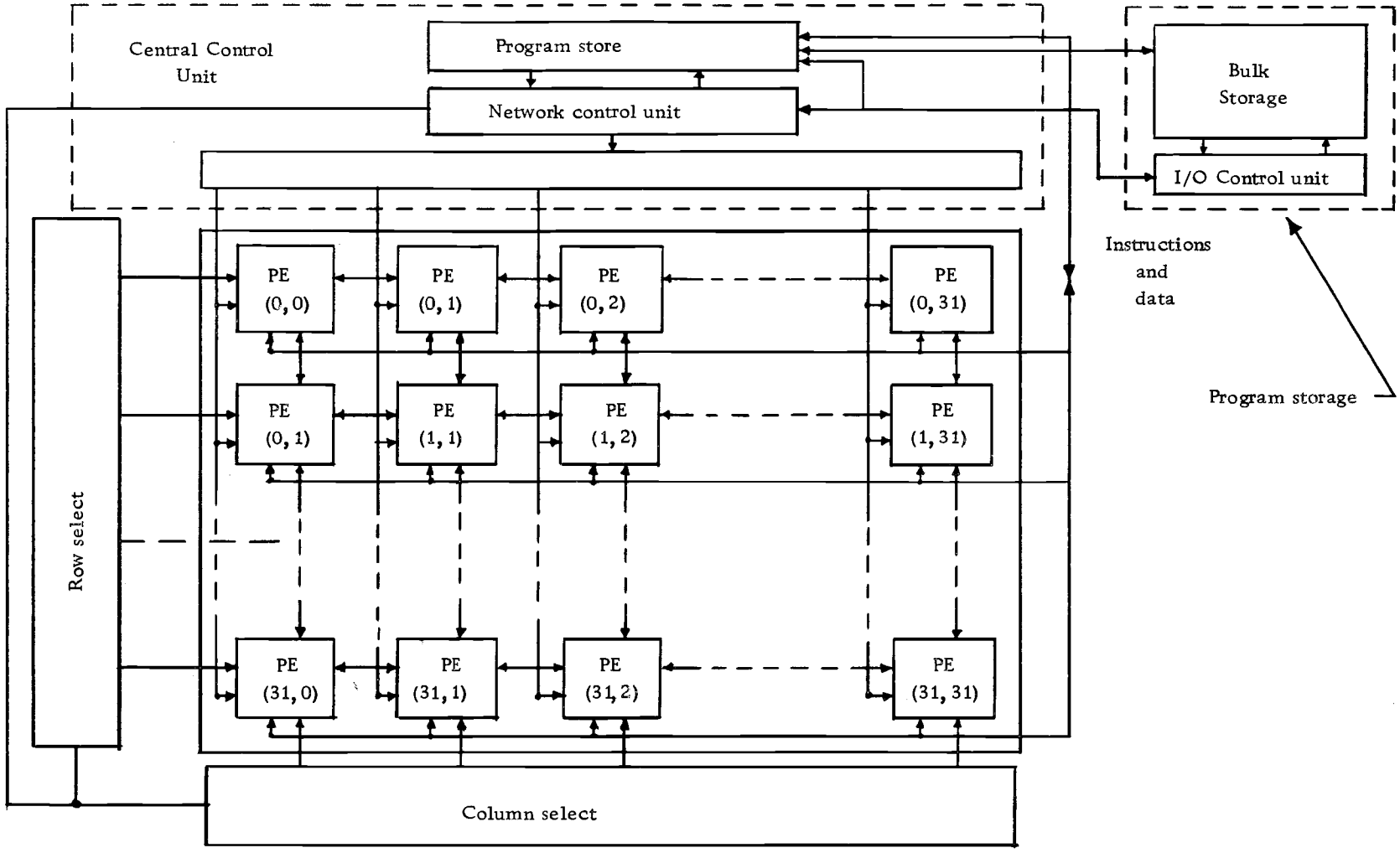


Figure 2.12. The network array processors of SOLOMON computer.

Studies with the architecture of SOLOMON computer indicated that such a parallelism approach was beneficial to a number of important computational areas, particularly well-formulated but computationally massive problems which are limited by the computing power of current conventional computers. Typical computation areas include manipulations of very large matrices (i.e., linear programming), the solution sets of orginary and partial differential equations (i.e., weather models, air traffic control models), and problem of correlation and autocorrelation (i.e., phased away signal processing, radar tracking models). The ILLIAC IV was derived from the SOLOMON machine.

The ILLIAC IV

The SOLOMON computer led to the development of the ILLIAC IV computer system with a view toward solving a number of problems which were too demanding of computer time to be feasible with any existing computer. In some classes of problems, it is observed that the same program steps are executed on many different sets of data. Examples include partial differential equation meshes, matrix operations and fast Fourier transforms. ILLIAC IV was constructed for the University of Illinois by Burroughs Corporation.

The system organization is shown in Figure 2-13. The original design consisted of 256, 64-bit word processing elements (PE),

arranged in four reconfigurable SOLOMON type arrays each consisting of 64 PE's and one control unit, called a Quadrant. The four arrays could be connected together under program control to provide multi- or single-processing operations.

The operating system resides in a Burroughs B6500 general purpose computer, which supervises program loading, array configuration changes, and I/O operations, both internal and external. Back-up storage for the arrays is provided by a large, directly coupled, parallel access disk.

ILLIAC IV system functional units can be described as follows:

1. Array Control Unit (CU) serves the same purpose as the control unit in SOLOMON computer. It decodes the instructions and generates control signals for all the processing elements (PEs), thus insuring that all the PEs execute the same instruction in unison. In fact, the array control unit has five basic functions, to control and decode the instruction stream, to generate the micro-orders needed to execute the instructions in the PEs, to generate common memory addresses, to process and broadcast command data words and to receive and process control signals such as those received from the I/O equipment or the B6500 control computer.

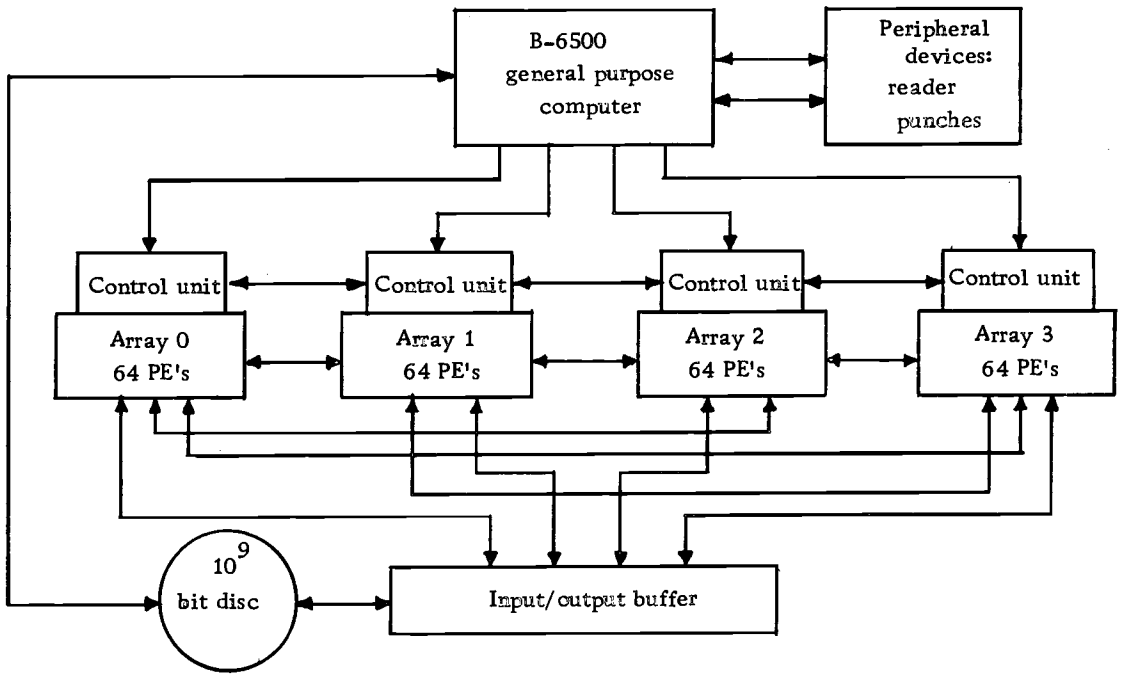


Figure 2.13. ILLIAC IV system organization.

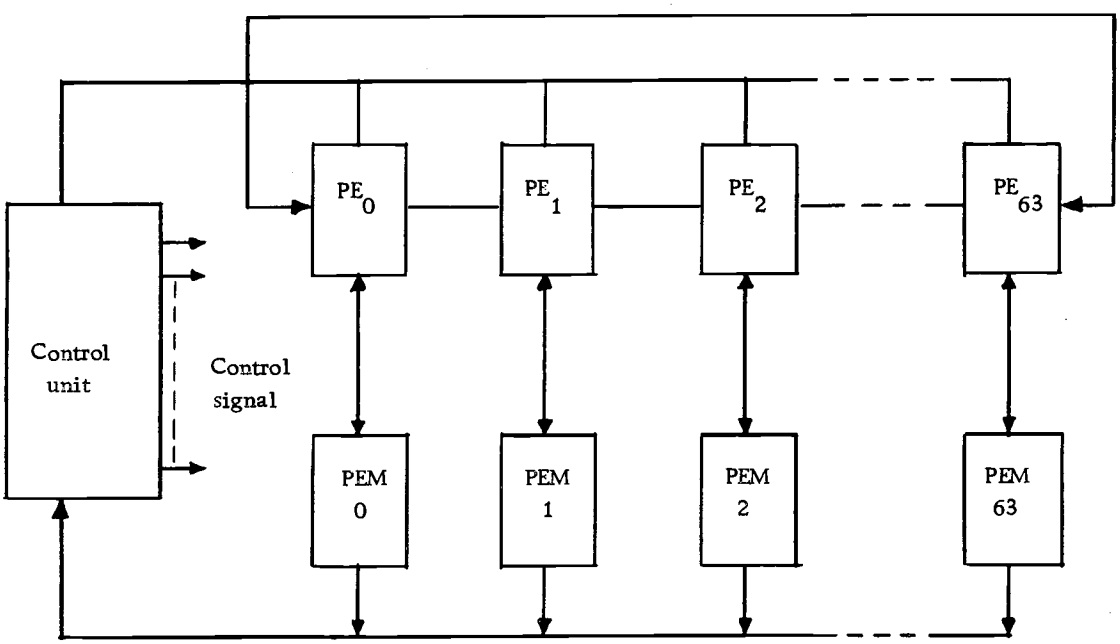
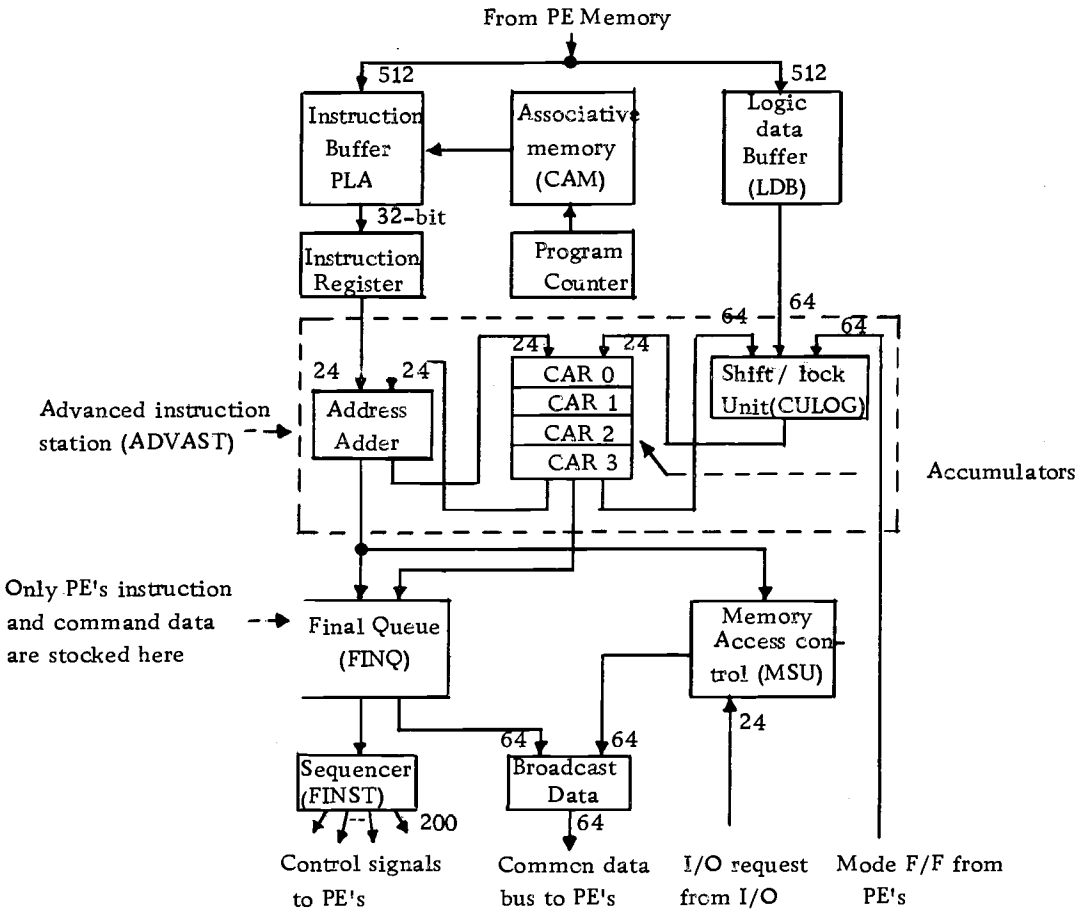


Figure 2.14. ILLIAC IV array structure.

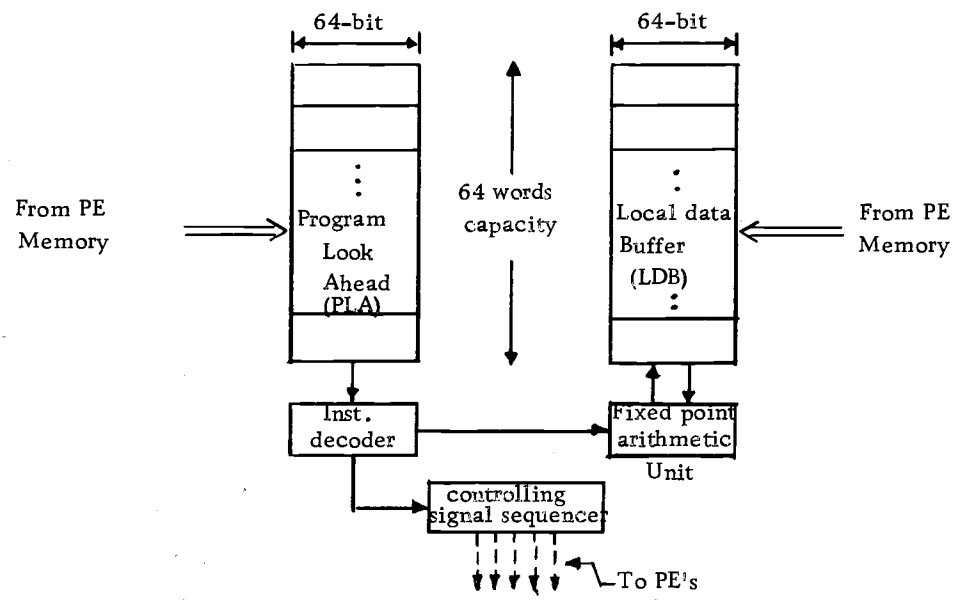
Figure 2.15 gives a functional representation of the major components of an Array Control Unit (CU). There is a Local Data Buffer (LDB), memory, composed of 64 64-bit words. This Local Data Buffer can be filled from any location in any PEM and also stored to any location in any PEM. There is also a block of 64 words called the Program Look-Aheas (PLA) or Instruction Buffer. This buffer provides an instruction queue.

The arithmetic unit in an Array Control Unit (CULOG) is a very simple unit and is limited to performing logical operations, fixed point addition and subtraction, operand fetching and storing results only within the Local Data Buffer (LDB). The four 64-bit accumulator registers (CAR0, CAR1, CAR2, and CAR3) are used for central communication with the CU and hold address indexing information and active data for logical manipulation or broadcasting.

2. The Processing Element (PE) executes the data computations and performs the local address indexing required to fetch the operands from PE-memory (PEM). A PE contains the following subunits: four 64-bit registers to hold operands and results; a floating-point adder/multiplier and a logical unit, capable of performing arithmetic, boolean, and shifting operations; a 18-bit index register and adder for store address modification; an eight bit mode control register. All instructions are 32-bits in length and



(a) Array control-unit in details.



(b) Array control unit simple block diagram. Shows instruction is decoded and transmitted.

Figure 2.15. Array control unit.

belong to one of two categories, array control unit instruction which generate local operations and PE instructions which are decoded in the Array Control Unit (CU) and then transmitted as micro-sequences to all the PEs.

3. The Processing Element Memory (PEM) is the individual memory provided to each Processing Element (PE). Each has a capacity of 2048 64-bit words. It is independently accessible by its attached PE, the CU, or I/O connections.

   In addition, the computing speed and memory of the ILLIAC IV array require a substantial secondary storage memory to maintain program and data files as well as back up memory for programs whose data set requirements exceed fast memory capacity. The disk-file subsystem consist of six-Burroughs model IIA disc storage units.

4. The B6500 general purpose machine is employed as the Control Computer. It is assigned to perform the following functions: executive control of the execution of array programs; control of the multiple-array connections; supervision of the internal I/O processes (disk to array, etc); supervision of the external I/O processing; processing and supervision of the files on the disk file subsystem; and

independent data processing, including compilation to obtain machine language programs for ILLIAC IV array processor.

5. The operating system consists of two interacting parts: a small portion is resident in the array processor memory for fast response to conditions arising from array programs. The bulk operating system is resident in the B6500 control computer. There are three types of scheduling in ILLIAC IV systems; processor array scheduling, job processing scheduling and I/O operation scheduling. The scheduling functions are accomplished by cooperation among the operation program resident in both B6500 and ILLIAC IV, the PEMs, the hardware registers, and the user's program statements.

ILLIAC IV provides both a lower-level language and a high level language called TRANQUIL.

Advantages

1. The instruction parallelism level can be performed in ILLIAC IV with SIMD processing.
2. For certain applications, the ILLIAC IV machine is the fastest computer at the present time. The most suitable applications are well-formulated and high-speed real time problems such as weather models, fast data correlation techniques, phased array signal processing and air traffic control.

3. The flexibility of PEs array configuration.
4. By assigning each PE a PEM, signal line delay and memory reference conflicts can be avoided. This eliminates the cost of switch network between PE and PEM, PE_i can fetch PEM by using route-instruction.
5. The system employs redundant PE units which the CU can enable and disable, increasing system reliability.

Disadvantages

1. System cost is very expensive.
2. Task level parallelism can be performed but is limited.
3. The ILLIAC IV physical system structure is unique and can cause inconvenience to the programmer at both lower and/or higher machine language levels.
4. The ILLIAC IV array computer can perform real-time, computation but not time-sharing multiprogramming.
5. The system efficiency will vary.
6. System control structure is complicated. Array control unit (CU) is the bottleneck to the powerful PE's.

CDC STAR-100 Computer

The CDC STAR-100 (String Array-100) system implements three important concepts, virtual memory, distributed computing, and vector instructions. From the hardware point of view, the CDC

STAR-100 computer is a pipeline processor structured around a four million byte (8 million byte optional) high bandwidth memory. Instructions in the CDC STAR-100 specify operations on variable length streams of data allowing full use of the memory bandwidth and the arithmetic pipelines. In the streaming mode the system has the capability of producing 100 million, 32 bit floating point results per second. A block diagram of the CDC STAR-100 is shown in Figure 2.16.

The CDC STAR-100 system functional units can be described as follows:

1. The pipeline processing unit consists of two independent pipeline processors. The block diagram is shown in Figure 2.16. Processor 1 consists of a pipeline floating point addition unit and a pipelined floating point multiplication unit. Processor 2 consists of a pipeline floating point addition unit, a non-pipelined floating point divide unit (register divide) and a pipelined multipurpose unit which is capable of performing floating point multiplication, dividing or square root operations.
2. The memory unit of CDC STAR-100 is composed of 32 interleaved banks, each bank containing 2048, 512-bit words. The memory cycle time is 1.28 μsec. The minor cycle rate

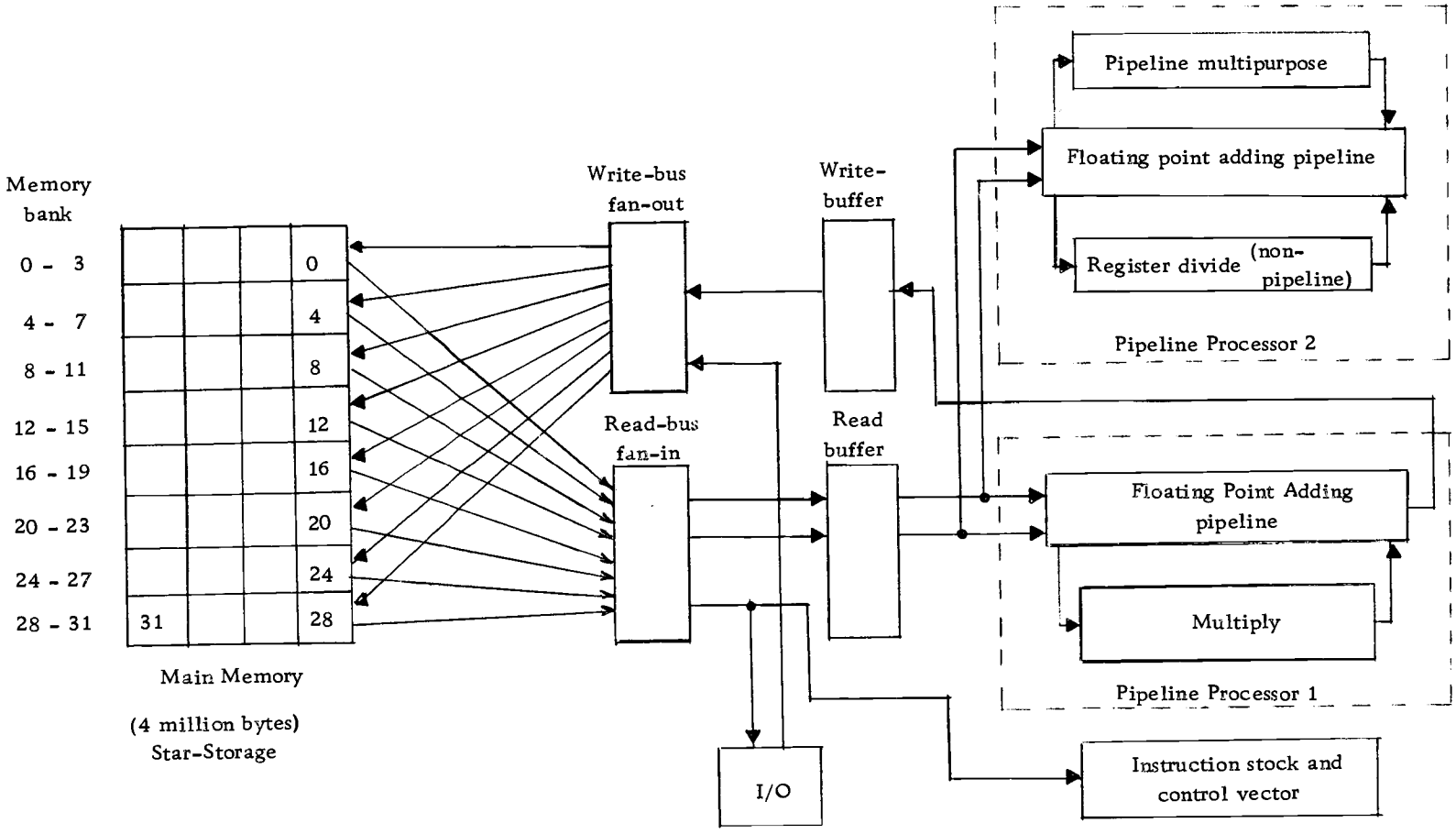


Figure 2.16. STAR-100 memory-pipeline data paths.

in the CDC STAR-100 is 40 nsec. Then it can support a total bandwidth of 512 bits of data per minor cycle.

As shown in Figure 2.16, the width of the memory data bus for each group of four banks is 128 bits (32 bits per bank). The capability of the data bus transfer rate is 128 bits per cycle. During stream operations, four buses will be active with each bus transferring data at the rate of 128 bits per minor cycle. Two of the buses are used for transferring the operational streams to the pipeline processor, the third bus is used for storing the results stream elements, and the fourth bus is shared between input/output storage requests and control vector references.

The read and write buffers as shown in Figure 2.16 are used to synchronize the four active buses. The memory requests are buffered and spaced eight banks apart, to eliminate memory conflicts. Hence, the maximum pipeline rate can be achieved regardless of the distribution of addresses among the four active buses.

3. The CDC STAR-100 system software consists of both a standard operating system and the STAR FORTRAN compiler. Due to its design, the pipeline processors and the STAR FORTRAN compiler increases the speed of processing data vector operations. It is clear that both machine language and high level language (STAR FORTRAN) can be used.

Advantages

1. The pipeline processors increase the computing speed.
2. The system used paging concept (visual memory) to obtain large visual address space.
3. Since the system uses a large page size (4096 bytes x 128 times) and has a large memory bandwidth, the system can handle a large data blocks, reduce the size of the page table and reduce COYE/disk memory transfer time.
4. The CDC STAR-100 is most suitable for vector data problems such as vector multiplication and payroll list processing.

Disadvantages

1. The CDC STAR-100 is large scale machine and its cost is expensive.
2. It is not economical to use the machine for small or medium size computations.
3. The system is still in experimental stage and has software problems.

Goodyear STARAN Computer

Several propriatary versions of the associative processor (AP) are being developed. STARAN is the first working associative array processor, built for the USAF by Goodyear Aerospace Corporation. It

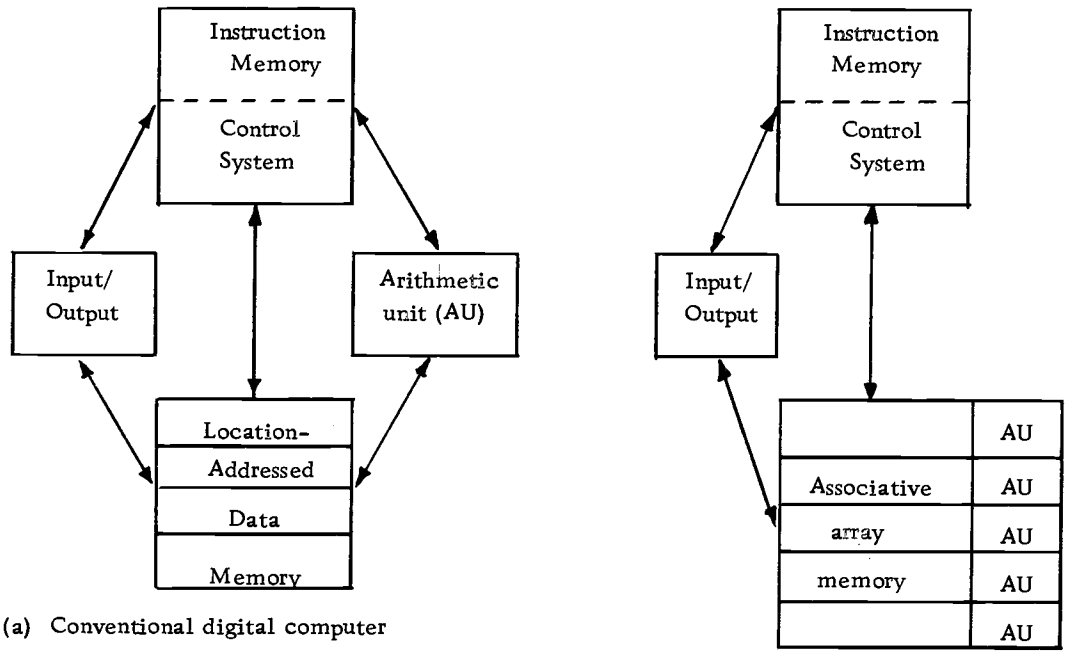
is a digital computer system that simultaneously performs arithmetic, search, or logical operations on either all or selected words of its memory. Four major features distinguish STARAN from the conventional computer:

1. Multidimensional access array memory.
2. Content addressable memory (Associative memory).
3. A simple processing unit at each word of memory.
4. A unique permutation network for shifting and rearranging data in memory.

Figure 2.17 shows the contrast between the conventional computer structure and the associative array processor computer structure. The system architecture of STARAN is shown in Figure 2.18

STARAN system hardware consists of the following Functional Units:

1. An AP control memory is used to store assembled AP application programs and also used for data storage and as a buffer between AP control and the other elements of STARAN. The AP control memory and associative array eyeles are overlapped. The AP control memory is divided into five memory blocks (see Figure 2.19).
2. The AP control logic directly controls data manipulation within the associative array processors and is also the data



(a) Conventional digital computer

(b) Associative array processor

Figure 2.17. The contrast between conventional and associative array machine.

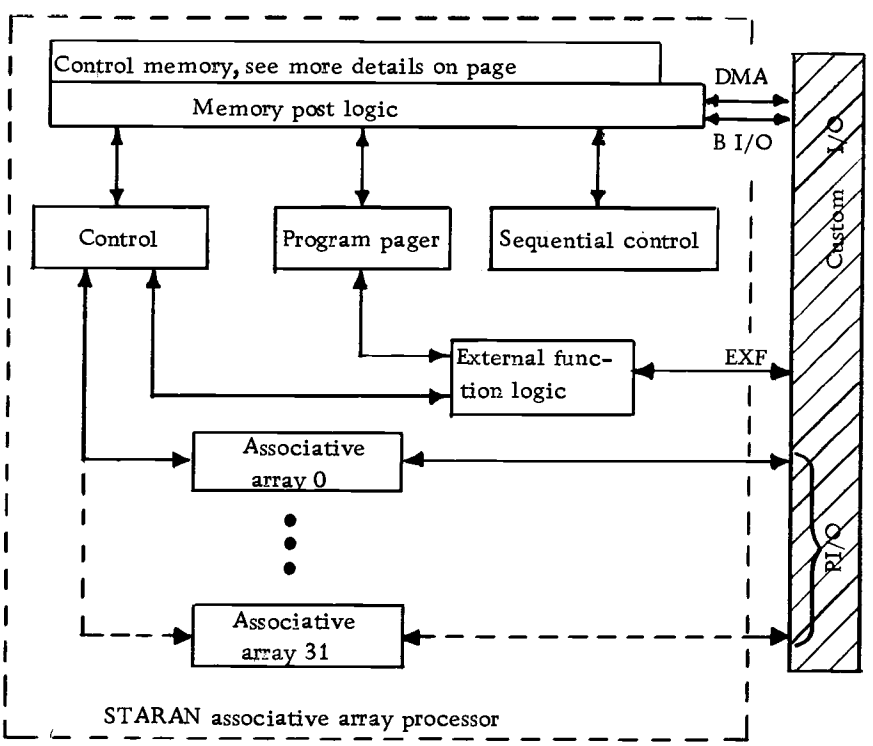


Figure 2.18. STARAN computer block diagram.

communication path between AP control memory and the array processors.

3. The program pager logic is used to load the fast page memories from the slow core memory in the AP control memory reaction. While the AP control is executing a program segment out of one page, the pager can be loading the other page with a future program segment.
4. External function logic is used to enable the AP control, sequential control, or an external device to control the STARAN operation. By issuing external function code to the External function logic, a STARAN element can interrogate and control the status of the other elements.
5. The sequential control processor is used to control the sequence of operations in STARAN. It has an 8k 16-bit memory, a keyboard-printer, tape reader/punch unit, and logic capability to interface the sequential processor with other elements of STARAN. The processor is used for system software programs (assembler, operating systems, diagnostic programs, debugging, and housekeeping routines).
6. Input/output is provided through a custom interface unit which communicates with a variety of computer systems and other external devices. Four interface options are

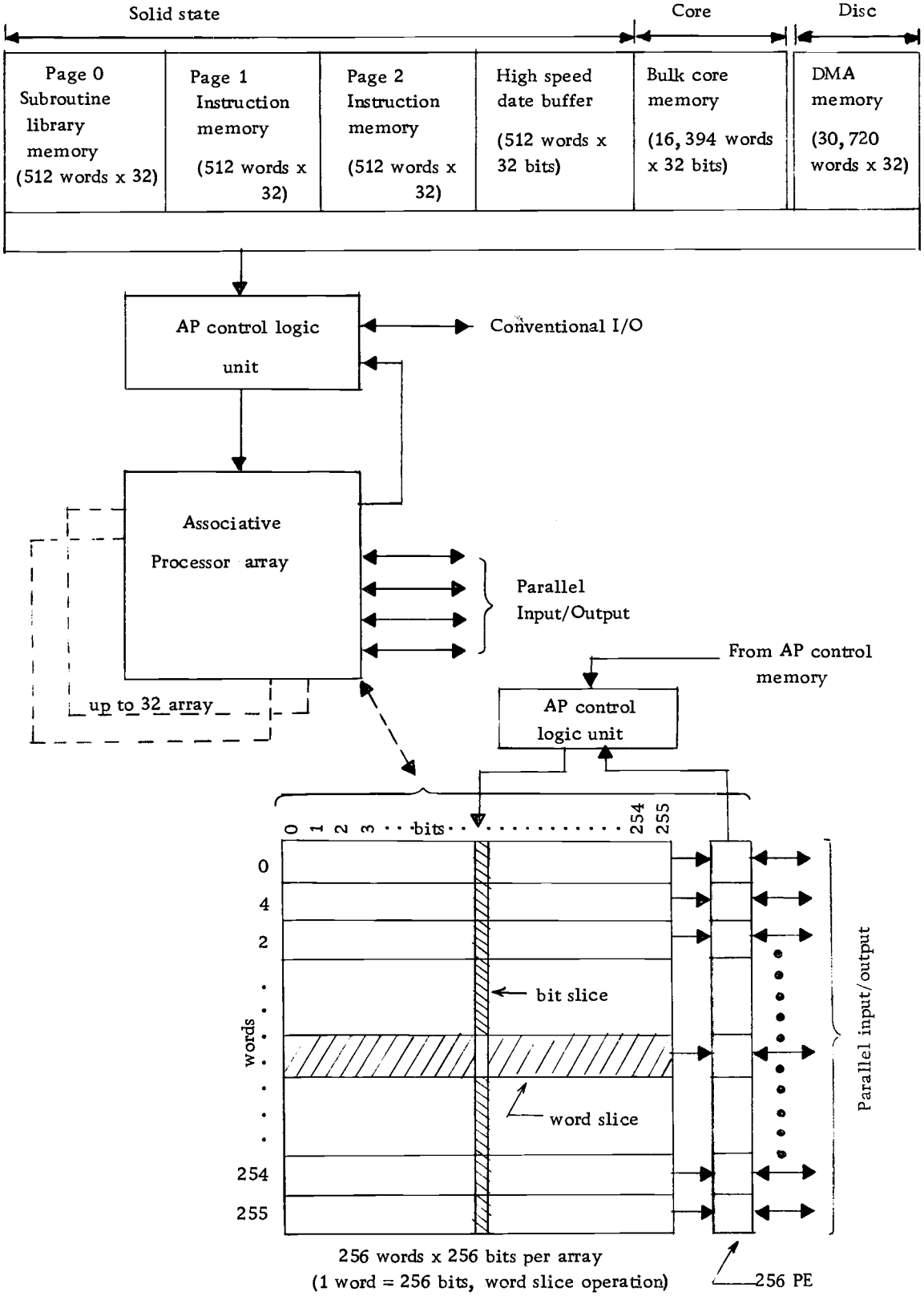


Figure 2.19. Associative processor organization.

available: Direct Memory Access (DMA), buffered input/output (BIO), external function logic (EXF) and parallel input/output (PIO).

7. The STARAN software system consists of an assembler (name APPLE: Associative Processor Programming Language), a utility package, a debugging package, diagnostics and subroutine library. At this time, an associative compiler has not yet been developed for STARAN. Therefore high-level languages such as FORTRAN cannot be used with STARAN.

Advantages

1. The system hardware structure is simple using the associative memory.
2. STARAN can perform simultaneously arithmetic, search, and logical operations.
3. It is high speed and high performance machine.
4. System efficiency is high for high computing rate problem.
5. It is suitable for applications such as sensor processing, tracking, data management, weather prediction and matrix problems.

Disadvantages

1. Associative array processor technique is still premature.

2. The area of applications to limited, relative to other typical computer systems.
3. Cost of STARAN itself is still high.
4. Since the STARAN is still new, and the system is experimental, not much data are available for evaulation and comparison.
5. It limits the major class of user (high level language user) since the associative compiler has not yet been developed for STARAN.
6. Arithmetic capability is less efficient than array processor or conventional processor.

2.3 The Software Implication of Parallelism

In the preceeding sections we have surveyed parallel processing hardware development. It is clear that, in order to gain more advantage in performing parallel operations in a computer system, not only must the system hardware provide parallelism, but the software of that computer system must also. Proper sequencing of instructions and tasks and the allocation of resources to achieve maximum parallelism is a software function.

A definition of levels of parallelism and techniques which the operating system can use to recognize the parallel processable tasks will be discussed in the following sections.

Levels of Parallelism

In general the concepts of software parallelism in a computer system can be classified into the following different levels:

1. Parallelism between jobs.
2. Parallelism between runs in a job.
3. Parallelism between subroutines in a run.
4. Parallelism between instructions in a subroutine.
5. Parallelism between stages in an instruction.

It is necessary to associate different mechanisms with achieving concurrency at these various levels. Parallelism within jobs and runs (1, 2, and 3) are considered as the domain of a given run through subtasking mechanisms. If the system is a single processor one, the parallelism between jobs and runs is handled in a time sharing fashion. In case it is a multiprocessor system, the parallelism between jobs and runs is handled in a time sharing fashion. In case it is a multiprocessor system, the parallelism between jobs and runs is handled in a multiprogramming fashion with overlapped or concurrent I/O operations.

Parallelism between high level language instructions is considered to be the domain of compilers. In either a single processor system or a multiprocessor, an appropriate compiler (compiler and parallel process recognizer) analyses or checks with consecutive

instructions have no conflicts in using the system resources and orders the instructions for simultaneous execution.

Parallelism between stages in an instruction is considered a hardware feature, which was discussed earlier concerning the design of the current higher speed machines such as CDC 6600, CDC STAR, ILLIAC IV and others.

In addition, the scheduling of work at all levels is subject to the same type of analysis and involves the same techniques. A compiler optimizer is a scheduler as much as is the scheduling element of an operating system.

Explicit and Implicit Parallelism

Consider the problem of having a machine know which tasks in a program can be executed in parallel. Two further definitions must be introduced:

1. Explicit parallelism. A piece of code is written with expressions to indicate the point at which parallelism is perceived. In other words, if the programmer himself indicates the tasks which can be executed in parallel within a computational process, such a program is considered to be explicitly parallel. The process is normally done by means of additional instructions in the programming language. The techniques described by Conway (13); the

Fork and Join technique; parallel fork (in ALGOL); TASK, CALL (in PL/1); TRANQUIL (for ILLIAC IV) and others are considered in this category.

2. Implicit parallelism. If indications of parallelism are not within the code or program, but opportunities for parallel execution are discovered by a "recognizer" during the compiling phase, or actually by hardware such as instruction look-ahead and pipeling this category is known as implicit parallelism. An example of implicit parallelism is the FORTRAN Parallel Processable Recognizer (30) discussed later in this chapter.

Explicit parallelism is relatively more simple to implement than implicit parallelism. In fact, in explicit parallelism, the very definition of a job indicates its independence in precedence over other jobs. The implicit approach is associated with complex compiling and supervisory programs. The detection of the inherent parallelism between a set of tasks depends on a thorough analysis of the source program.

Bernstein Conditions

Bernstein (6) has defined conditions for parallelism between tasks. These conditions are general and can be applied to sections of a program of arbitrary size. The conditions are interesting

because they give the ideas concerning the structure of programs amendable to parallel processing and the memory organization of a multicomputer system.

The question encountered is whether or not two tasks can be performed in parallel. This is a function of not only the algorithm being performed, but also of the way it has been programmed. To answer the above question, Bernstein having done research in this area before, came up with a set of conditions for Parallel and Commutative Transformations.

They can be summarized as follows:

1. The concept of parallel processing of two consecutive tasks:

   If T_1, T_2 and T_3 are sequential tasks of a program, their flow chart can be represented as in Figure 2.20(a). If task T_1 and T_2 are completely independent, alternating the order of processing of task T_1 and T_2 will not change the output result. Consider the program flow chart of Figure 2.20(b). This indicates that the time relationship between performance of T_1 and T_2 is arbitrary. If T_1 and T_2 can also be simultaneously performed by two separate processors, the corresponding block diagram flow chart is represented in Figure 2.20(c). Program blocks which are independent in the above sense will be called Parallel.

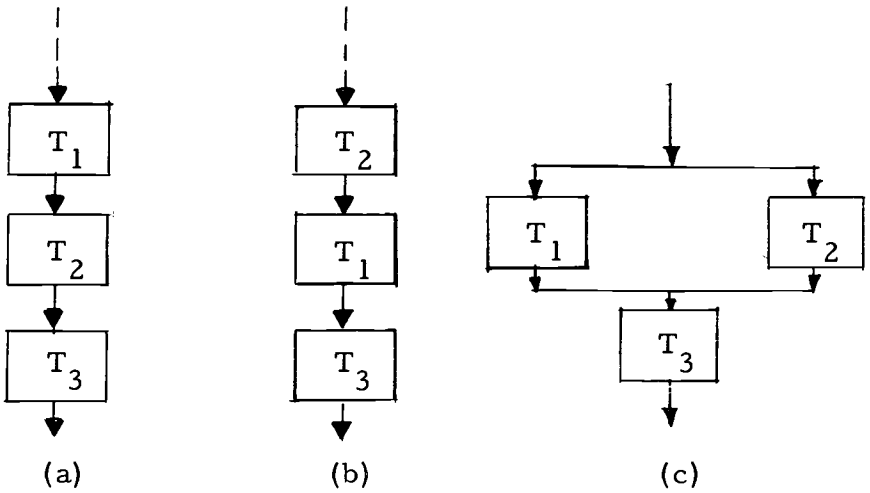


Figure 2.20. Sequential and parallel execution of a computational process.

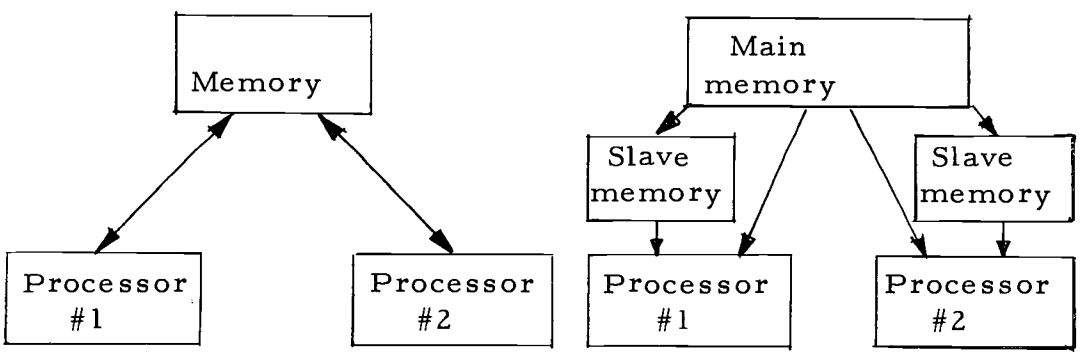
2. The concept of a commutative condition:

   If T_1 and T_2 are defined as above, but, in this case, T_1 and T_2 can be performed in either order as shown in Figure 2.20(a) and 2.20(b) but not in parallel, then the program blocks satisfying this condition is called commutative. The composition of two commutative functions is an example of this situation.

3. Berstein's basic machine models:

   Two, Model A and Model B, are considered by Bernstein to analyse the conditions of programs for parallel processing. Machine A is shown in Figure 2.21 (a) and Machine B in Figure 2.2(b). Each processor communicates directly with a single large memory, shared memory, but Machine B has slave

memories associated with each processor. The purpose of the slave memory is to act as a storage buffer for its associated processor. A processor may fetch information from the main memory but any information to be shared is kept in the slave memory. When a fetch operation is called for, the processor first searches its slave memory to see if that location has previously been stored there. If not, the location is obtained directly from the main memory. When both processors P_1 and P_2 have completed their tasks, the information in the slave memories is transferred to the appropriate locations in the main memory.



(a) Machine A. (b) Machine B.

Figure 2.21. Machine models.

4. Condition for Parallel and Commutative Transformation:

There are four different ways that a sequence of instructions or subprograms can use a memory location:

a. The location is only fetched during execution of the task T_i (W_i)

b. The location is only stored during execution of the task T_i(X_i)

c. The first operation of the task T_i using this location is a fetch. One of the succeeding operations stores into it. (Y_i)

d. The first operation of the task T_i using this location is a store. One of the succeeding operations fetches it. (Z_i)

Where W_i, X_i, Y_i and Z_i are the sets of memory locations falling into categories a, b, c, and d respectively.

Logical relation among T_i, W_i, X_i, Y_i, and Z_i:

a. Only states of memory locations in X_i, Y_i and Z_i and not the state of the entire memory are modified by execution of the task.

b. Execution depends only on the state of the locations referenced in W_i, Y_i, and Z_i, not on the state of the entire machine.

c. $W_i \neq Z_i \neq Y_i$

d. The information obtained from W_i and Y_i has been established in the machine prior to execution of the task.

e. The information extracted from Z_i is actually computed within the task itself.

5. The program flow chart of Figure 2.20(c) is considered regardless of how fast T_2 (or T_1) is executed along the other parallel path. It is required that

$$(W_1 \cup X_1) \cap (Z_2 \cup Y_2 \cup Z_2) = \phi \qquad (2.1)$$

where ϕ = the empty set.

We do not want T_2 destroying results which T_1 is computing for later reference. It is required that

$$Z_1 \cap (X_2 \cup Y_2 \cup Z_2) = \phi \qquad (2.2)$$

The combination of 1. and 2. yields

$$(W_1 \cup Y_1 \cup Z_1) \cap (Z_2 \cup Y_2 \cup Z_2) \qquad (2.3)$$

We do not want T_2 to require information computed in T_1 since the execution of T_1 will no longer necessarily procede that of T_2, thus

$$(W_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2) = \phi \tag{2.4}$$

Also, for the same reason,

$$(Z_1 \cup Y_1 \cup Z_1) \cap Z_2 = \phi \tag{2.5}$$

combining (4) and (5) yields

$$(X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2 \cup Z_2) = \phi \tag{2.6}$$

Finally, we must insure that the partial state of the machine upon which the execution of T_3 depends is the same in the parallel and sequential models. The memory locations involved in this partial state are those contained in W_3 and Y_3. Of the locations modified the combined effect of T_1 and T_2 only those modified by both T_1 and T_2 are affected by the order in which the two programs are executed. Thus we require

$$(X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) \cap (W_3 \cup Y_3) = \phi \tag{2.7}$$

From 2.3 and 2.6,

$$(X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = X_1 \cap X_2 \tag{2.8}$$

Hence, the final condition is

$$X_1 \cap X_2 \cap (W_3 \cup Y_3) = \phi \tag{2.9}$$

Together 2.3, 2.6 and 2.9 assure that tasks T_1, T_2, T_3 in Figures 2.20 (a) and 2.20 (b) will compute the same results for any input data assuming Machine A for the parallel processor.

> The conditions 2.3, 2.6, and 2.9 can be summarized by noting that the areas of memory from which task 1 reads and onto which task 2 writes should be mutually exclusive, and vice versa. With respect to the next task (T_3) in a sequential process, Task 1 and 2 should not store information in a common location. Based on the constraint of using a memory location and logical relations of T_i, W_i, Y_i and Z_i as mentioned before, the conditions for parallel processing of Machine B condition (2.4) alone is sufficient for parallel operation. The similar concepts of analysis can be applied for commutative, branching and loop situations.

More details and examples of applications of Bernstein's conditions are reviewed in (6).

Ramamoorthy and Gonzalez Algorithm

There are many techniques for recognizing parallel processable streams in computer programs. The algorithm described here was developed by C. V. Ramamoorthy and M. J. Gonzalez (30). It

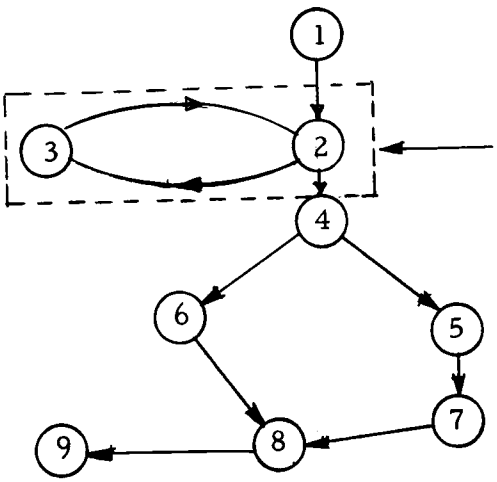
is implemented in the first FORTRAN parallel task recognizer. For convenience in describing the algorithm the following terms must be defined:

Program graph. A computational process, computer program such as FORTRAN program, can be modeled by oriented graphs (program graph) in which the vertices (nodes) represent single tasks. The oriented edges (directed branches) represent the permissible transition to the next task in the sequence.

Connectivity matrix (C). A graph modeling the computation process can be represented in a computer by means of a connectivity matrix (C). C is a n x n dimension matrix such that C_{ij} is a "1" if and only if there is a connection arrow from node i to node j, and C_{ij} is "O" otherwise. The represented computational process can be studied by simple manipulation of the connectivity matrix.

Strongly connected. A graph consisting of a set of vertices is said to be strongly connected if and only if any node in it is reachable from any other.

Subgraph of any graph is defined as consisting of a subset of vertices with all the edges between them retained.



(a) Program Graph

	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0	0
3	0	1	0	0	0	0	0	0	0
4	0	0	0	0	1	1	0	0	0
5	0	0	0	0	0	0	1	0	0
6	0	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0

(b) Connectivity Matrix

Figure 2.22. Program graph of a sequentially coded program and its connectivity matrix.

Maximal Strongly Connected (M.S.C.) Subgraph: is a strongly connected subgraph that including all possible nodes which are strongly connected with each other. In a given connectivity matrix C, all its M.S.C. subgraph can be determined by the well-known methods mentioned in (30).

The reduced program graph, will not contain any strongly connected components. It is obtained by replacing each of its M.S.C. subgraphs by a single vetex and retaining the arrows connected between these vertices and others.

Final program graph: It contains the following types of vertices:

1. The Branch or decision type vertex from which the execution sequence selects a task from a set of alternative tasks.
2. The Fork vertex which can initiate a set of parallel tasks.
3. The Join vertex to which a set of parallel tasks converge after their execution.
4. The Normal vertex which receives its input set from the outputs of preceeding tasks.

The sequence of operations needed to prepare for parallel processing in a multiprocessor computer an input program written for a uniprocessor machine are as follows:

1. Derive the program graph which identifies the sequence in which the computational tasks are performed in the sequentially code program, (See example in Figure 2.22) and build up the connectivity matrix.

2. Determine the possible M.S.C. subgraph by using "simple operations technique". Each M.S.C. is considered as a single task. The corresponding reduced graph is derived. (See example in Figure 2.22).

3. Derive the final program graph by analyzing the inputs of each vertex in the reduced program graph. An element T_{ij} is a "1" if and only if the j-th task (vertex) of the reduced graph has as one of its inputs the output of task i, otherwise T_{ij} is "0".

4. Partition the vertices of the final graph into precedence partitions as follows, using the connectivity matrix T,

   Step 1. A column (or columns) containing only zeros is located. Let this column correspond to vertex V_1.

   Step 2. In this step delete from T both the column and row corresponding to this vertex. The first precedence partition is

$$P_1 = \{V_1\}$$

   Step 3. In obtaining the successor precedence partitions

$$P_2 = \{V_{21}, V_{22}\}$$

P_3, P_4 -----, P_p. The procedure in step 1 and 2 are repeated respectively, until no more columns or rows remain in the matrix T. It can be shown that this partitioning procedure is valid for connectivity matrices which contain no strongly connected components.

The implication of this precedence partitioning is that it P_1, P_2 P_3, -----P_p corresponds to times t_1, t_2, t_3, -----t_p, the <u>earliest time</u> that a task in partition P_i can be initiated is t_i. It is also desirable to provide an indication of the <u>latest time</u> at which a task may be initiated.

5. From precedence partitioning and the final program graph, a Task Scheduling Table can be developed.

The algorithm does not try to determine whether any or all of the iterations within a loop can be executed simultaneously, rather the loop is considered as a single task. Sometimes, a loop may contain a large number of statements, and a great amount of potential parallelism may be lost. For this reason, the recognizer should generate a separate connectivity matrix for each loop within the program. This algorithm will be used later in Chapter to analyze an example program and use it to simulate parallel processing in the proposed architecture.

A survey of techniques for recognizing parallel processable streams in computer programs including the detection of parallel components within compound tasks can be found in (30).

III. DESIGN OF A COMMAND AND CONTROL COMPUTING SYSTEM

The Royal Thai Air Force has need for a suitable computer system to handle their military complex. A system is to be designed having the capacity to monitor and direct all aspects of the Defense Operations: such as radar tracking for air defence and air traffic control for both military and civilian traffic. This computer system will also handle the data processing function for planning and programming for logistics, budget management, financial status, payroll processing, and a Military personal information filing system.

The proposed computing system should satisfy the functional requirements of command and control data processing: availability, adaptability, expansibility and capability for system software. These criteria are considered as the first priority.

System expansion should be possible without incurring the cost of providing more capability than is needed at the time. It should meet the long-range budget constraints. This ability of the system to grow to meet the demands should apply not only to the conventionally expansible areas of memory and I/O but to computational devices, as well.

Two alternative approaches to the system design are considered. An obvious approach is to purchase or rent the whole system from a

reliable computer manufacturer. One could be selected whose characteristics and capability best match the specifications with a reasonable price.

The second approach is to build a system, utilizing the most effective elements available from the leading hardware manufacturers. The general idea of this approach had been mentioned by West (40). This approach also provides the flexibility to implement new ideas and advanced technologies (both hardware and software) in the computing system. The second approach seems to be more promising and challenging in the long-run and the command and control system was designed in this way.

3.1. System Design Philosophy

In attempting to find a suitable architecture for command and control processing, a multiprocessor structure appeared to best meet the design criteria. A typical multiprocessor would consist of modular replicated subunits, usually processors, memories and I/O controllers. Control for the system can either be centralized or distributed. The processing function can consist of identical processor units (symmetric organization) or a collection of special purpose processors (asymmetric organization).

The choice of a symmetric organization will allow the amount of reconfiguration and controlled system degradation capability

necessary to meet the availability and reliability criteria. Distributed and redundant implementation of the control hardware and software will also support this objective.

Since most multiprocessors utilize some form of switching between processors, memories and peripheral controllers, the capability for system expansion is easy to implement.

The general problems attached to multiprocessor configurations such as programming difficulties and system cost must be overcome in the design.

Although multiprocessors are generally high performance systems, this objective will be enhanced by the proposed two level parallel processing concept. The possibility of task/program level parallelism in a multiprocessor system is apparent. Addition of instruction overlap capability within each processor unit will provide an added dimension to the overall performance.

The strict cost limitations will be overcome by keeping the system design as simple as possible and by using state-of-the-art, low-cost LSI components.

The usual system software overhead for recognizing and scheduling parallel processes and allocating necessary resources should be reduced by the realtime fixed program nature of most of the intended applications.

3.2. System Overview

A block diagram of the proposed system is given in Figure 3.1. The system consists of P identical main processor units (PU's), M + 2 memory modules (M's), N peripheral processor units (PPU's) and several crossbar switches.

The main processors are LSI implemented general purpose units. The essential features of the PU include 10 functional units, a collection of operand, address and index registers, an instruction stack and fetching mechanism, a conflict resolution unit, an associative memory page table and a memory access hopper.

Instructions are continuously fed to the PU by the instruction fetch mechanism. The standard machine word length is 48 bits. The instructions are either 16 or 32 bits. Each instruction fetch is a double word fetch including up to six 16 bit instructions. All PU instructions are register-register instructions, with operands held in the PU operand, address and index registers. The conflict resolution unit allows instructions to be issued and executed in the ten functional units at the highest conflict free rate. This allows maximum instruction execution overlap.

The PU can address the memory modules only through the address registers, the page table associative memory and the memory access hopper. In this sense, the PU is hidden from main memory and operates as a slave arithmetic processor to the PPU's.

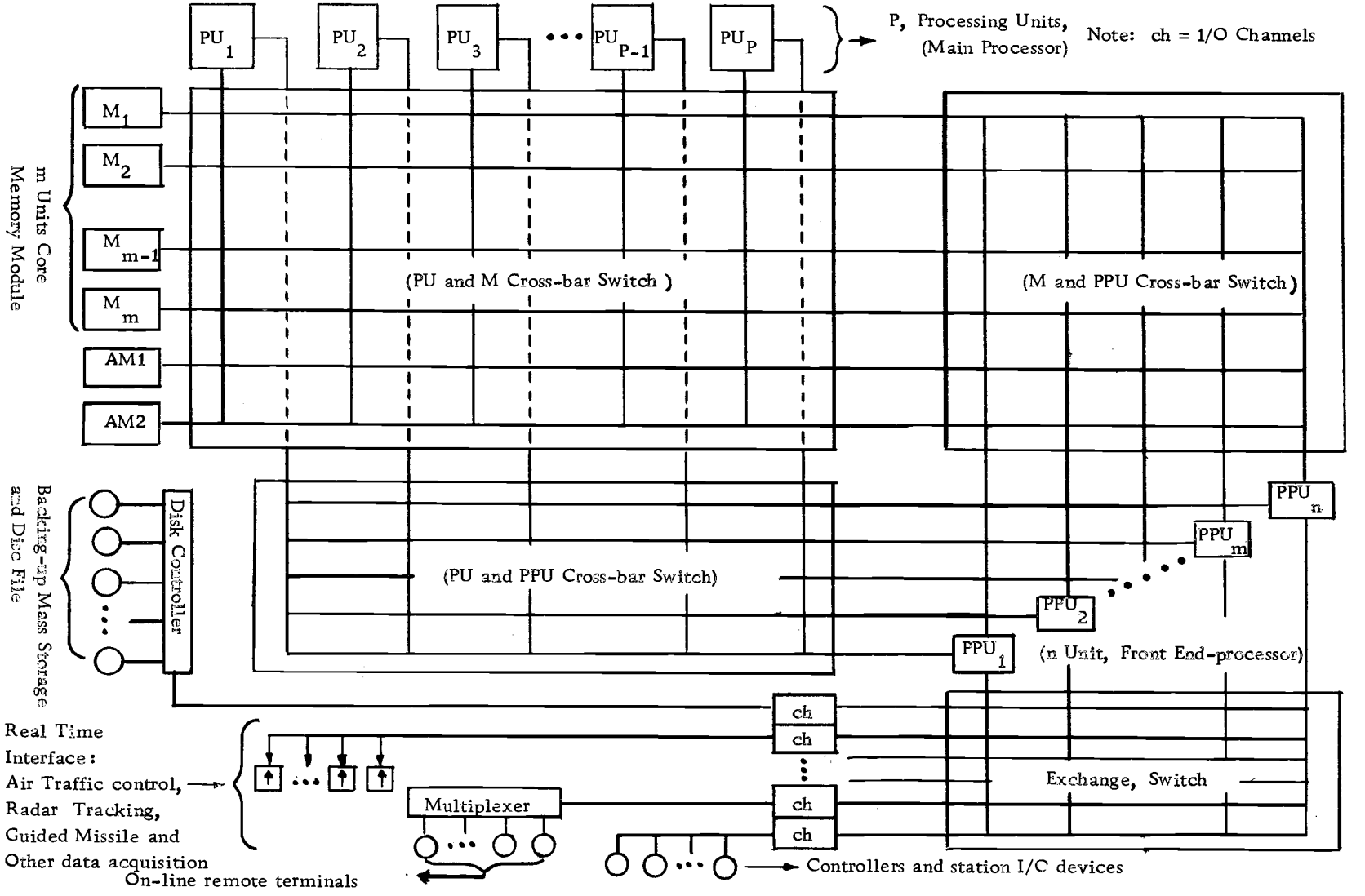


Figure 3.1. The RTAF Command and Control Two-levels Parallel Processing Computer System Organization.

Complete details of the PU design are given in Chapter IV.

The PPU's are similar in structure to standard 16 bit minicomputers. Their implementation will likely be similar to the minicomputer-on-a-card microprocessor based products now commercially available.

The PPU's handle I/O Control, but also execute the "Master Control" operating system routines. These routines are held in one of the protected memory modules. Actual execution of the Master Control function is allowed to float among the PPU's. This provides extra reliability.

One other special assignment on the PPU's is the implementation of configuration control. A PPU under Master Control can reconfigure the system structure to remove or replace faulty subunits and to provide special purpose structures.

The PPU design is summarized in Chapter VI.

The memory modules are 4K, 48 bit units. Memory interleaving is provided to increase bandwidth. Memory modules may be shared by two or more processors. Two associative memory modules are provided for Master Control table storage.

A virtual memory scheme is implemented in the system. It is a simplified paging scheme. Master Control uses a modified LRU algorithm for memory management. The associative memory

page tables along with the AM modules are used as hardware paging aids.

A complete discussion of the memory system can be found in Chapter V.

The final and, perhaps, most important major units are the crossbar switches. These are LSI implemented units and feature internal round-robin contention logic to reduce memory conflicts. The crossbar switch design is given in Chapter V.

The designed system was simulated for a typical FORTRAN program. Both the system level behavior and the PU behavior were investigated and the results compared with sequential processing in Chapter VII. An extensive simulation of memory conflicts was performed and the results used to design the PU memory access hopper, the crossbar switch network, the memory module interleaving and the processor/memory ratio in this system.

3.3. Operating System Summary

Since this is a preliminary design study, a complete operating system was not developed. In this section some constraints and requirements of such a system will be discussed.

The largest class of applications for the designed system is in real time data acquisition and control rather than traditional EDP. Most jobs run in the system will be of this type.

The operating system must be responsive to dynamic tasks as well as static jobs. The essential strategy will be that static tasks will be handled on a priority basis while dynamic requests will be handled by an interrupt system with service provided on a deadline basis. An early model of this scheme was implemented in the Burroughs D 825 system (4).

The Master Control routines are considered to be hidden in the memory with associated data such as memory maps, assignment tables, etc. One PPU will execute these routines and will be said to be in the Master Control mode. A PU executing a given task is in the process mode. When it finishes its present task, the processor will interrupt the PPU executing Master Control. Its status will be switched from process mode to available mode. The corresponding entry in a processor assignment table stored in an AM module will be updated.

As new tasks move into the system, a Task Status Block will be created for each one by the PPU in Master Control Mode. This block will include the following information:

Task name

Priority

Periodicity

Real-time necessity index

Task status: wait for available processor

Active (processor is assigned and ready to process)

Wait for the file (data)

Wait for channel

Wait for I/O device

etc. . . . . .

PU status: Instruction counter

X register contents

A register contents

B register contents

Memory Assignment:

Base address

Block table

Protection code, etc.

I/O device assignment

When a task goes into the wait state, the Master Control PPU will consult the processor assignment table for an available PU. If a PU is free, the Master Control PPU will allocate a sufficient number of physical pages (one page = one memory module) to the task with a LRU paging algorithm. After the necessary task data has been copied from backup storage into the assigned memory

modules, and the appropriate entries in the Task Block Table and PU AM page table are updated, the Master Control PPU will execute an "Exchange Jump" instruction. This instruction will cause the assigned PU registers to be loaded from the PU status portion of the Task Status Block and the PU will begin to execute the task.

As was mentioned, the PU's operate as slaves to the Master Control PPU. The PU's are capable of arithmetic and logical processing. An I/O request will cause an active PU to halt, interrupt the Master Control and go into the available state. The Master Control would then assign a PPU to handle the I/O operation. All I/O interrupts are handled initially by the Master Control PPU. Upon identification of the interrupt type another PPU would be assigned to the interrupt.

Besides processor/task scheduling and interrupt handling, the Master Control program handles memory management, resource allocation, diagnostics and parallel task recognition.

This last function makes use of a parallel task recognizer which will part of the high level language compiler. For example, if a FORTRAN program entered the system as a job, the Master Control PPU would create a task status block, locate a free PU, assign memory and start execution of the compilation. The compiler can be located in a shared memory module and be used by more than one PU simultaneously. During compilation, a task scheduling table

would be created by the recognizer and later used by the Master Control to schedule the compiled routines for maximum parallelism. A typical recognizer is the FORTRAN Parallel Task Recognizer proposed by Ramamoorthy and Gonzalez. This was discussed in Chapter II. The inherent overhead for such a recognizer would be overcome through repeated use of the compiled program.

In later sections, details of the system hardware will be given. Again, although the operating system plays a major role in the overall efficiency of the system, the preliminary stage of the hardware design makes complete specification impossible.

IV. PROCESSING UNIT SPECIFICATIONS

The central theme in the design of our system is the implementation of two level parallel processing, task level parallelism and instruction level parallelism. The former is achieved through the general multiprocessor configuration and the operating system as described in Chapter III. Instruction level parallel execution is performed in the PU's by providing multiple function units and multiple general purpose registers.

A block diagram of a typical PU is shown in Figure 4. 1. There are three groups of eight registers in each PU. They are:

The X registers: These 48-bit operand registers are provided as temporary storage buffers for data words within each PU. In general, data words enter a specified X register from the core storage module through the cross-bar switching network. These data are operated on by arithmetic and logical functional units and are finally returned through the cross-bar switch to the main core memory module from a specific X register, not necessarily the same one they entered.

The A registers: These 18-bit address registers control the main core storage references and can also be indexed by appropriate use of the 12-bit index registers.

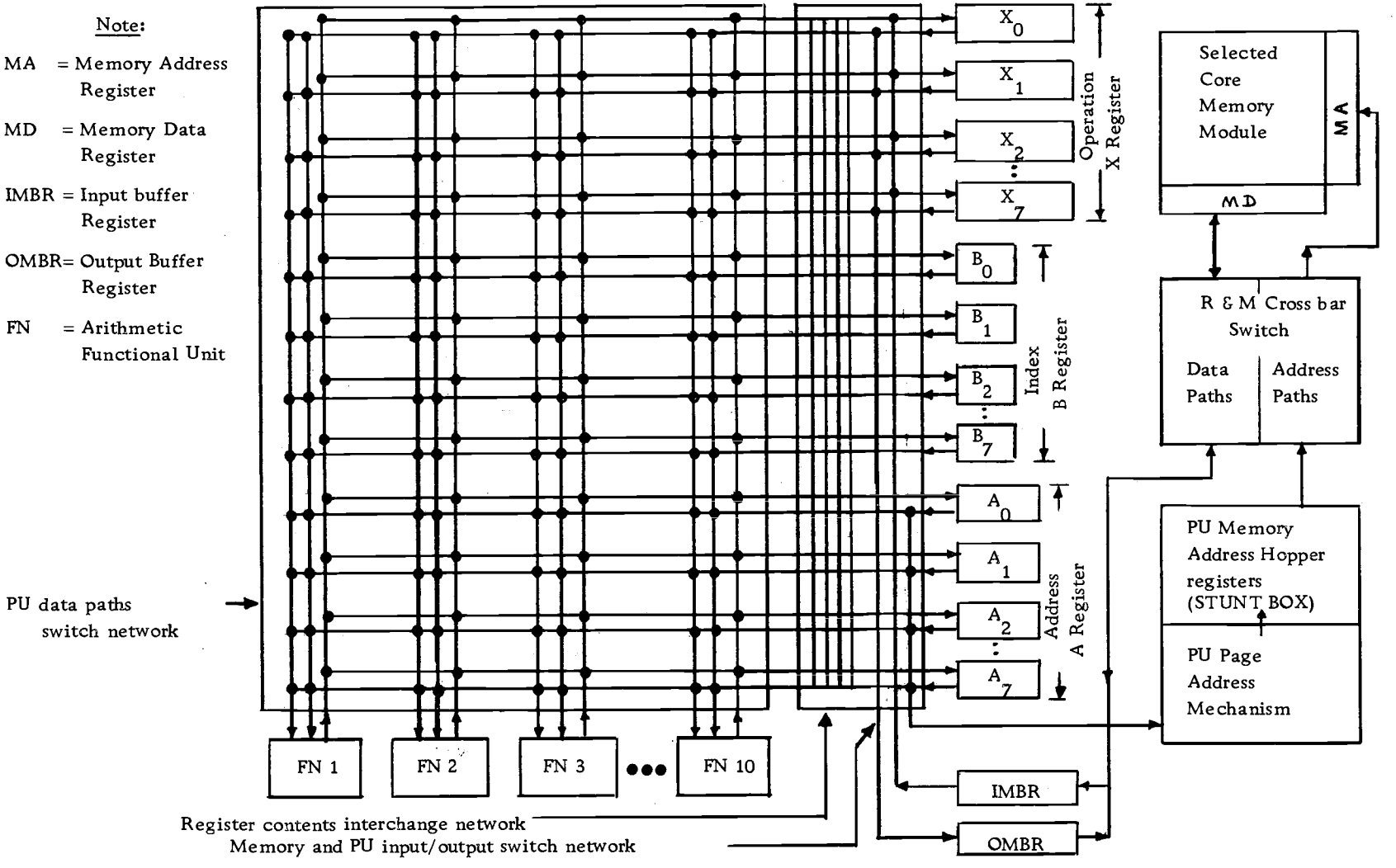


Figure 4.1. Illustration of internal PU organization block diagram.

The B registers: These 12-bit index registers are provided to perform the same indexing function as that in the conventional machine.

Each X register corresponds directly to an A register; X0 to A0, X1 to A1, etc. The first five X registers provide data access from memory. An address placed in A3, for example, will cause an automatic fetch from that location in core to X3. In a similar manner, addresses placed in A5-A7 will cause an automatic store of the corresponding X register.

Addressing to main memory is through the OMBR, IMBR, page addressing mechanism, the PU memory address hopper and the P and M cross bar switch. Details will be given in Chapter IV.

For convenience, two standard cycles are defined in this computer as follows:

1. The Major Cycle: The major cycle is identical to the core storage cycle of the main storage and the PPU storage unit. (A major cycle is equal to 1000 nanoseconds or 1 microsecond).

2. The Minor Cycle: The minor cycle is a measure of the time taken to transfer one data word through the storage distribution system (a minor cycle is equal to 100 nanoseconds, or 0.1 microsecond). The start and stop of internal operations are based on the minor cycle clock.

From the result of investigation and empirical analysis, it is reasonable to provide ten functional units to perform the arithmetic and logical operations in each Processing Unit. The ten functional arithmetic and logical units which make up the arithmetic portion of each PU are shown in Table 4. 1. Each functional unit type will be implemented with available low cost LSI components.

Table 4. 1. PU arithmetic and logical functional units.

Items	Description	Units
1-7	General purpose functional unit can perform the following arithmetic operations: ADD, BOOLEAN, BRANCH, SHIFT, and INCREMENT.	7
8-9	Multiplier units	2
10	Division unit	1

The operation of the functional unit is generally based on three-address instruction format (see figure 4. 2). The basic sequence of operation of a functional unit is as follows:

1. Both operands must be available in PU X registers.
2. The input operands are transferred to the desired functional unit.
3. The execution of the corresponding instruction is performed. On completion, the result is temporarily stored in the functional unit.

4. The control system transfers this result to the appropriate PU register.

5. The corresponding functional unit is released.

There are two types of PU instruction formats, 16-bit and 32-bit, as shown below:

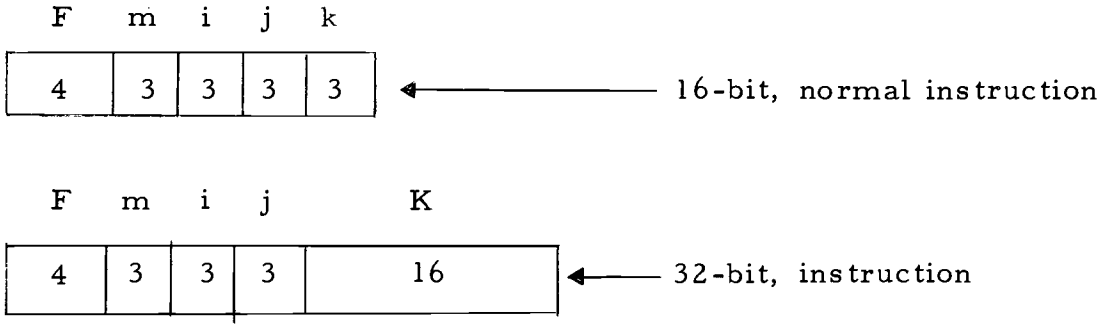


Figure 4.2. FU instruction format.

where F-field, denotes a major class of functions. It identifies one of the 3 different arithmetic and logical functional unit types.

m-field, denotes a mode within the function unit type.

i-field, identifies one of eight registers within the proper group of X, B, or A registers.

j-field, identifies one of eight registers within the proper group.

k-field, identifies one of eight registers within the proper group.

K-field is an 16-bit immediate field used as a constant or branch address.

Each register designator, i, j, and k, refers to one of eight registers and plays a very important role in each PU in issuing the instructions, reserving the registers and functional units to perform concurrent or over-lapping instruction execution. All register designators are used by the PU reservation control unit, block diagramed in Figure 4.3 and detailed in section 4.2. Note that a 48-bit word will contain 3 short format instructions or one long and one short format instruction.

In order for a PU to execute a specific task, the following operations are performed.

1. Initialize: The PU is started, stopped, or otherwise interrupted by means of a special instruction "Exchange Jump". This operation will be initiated by the PPU acting as the "Master Control". After the PPU executes the Exchange Jump, an appropriate block of core storage is referenced. "Master Control" causes the necessary information (program status) to fill up the X, A, B, status and control registers of the referenced PU, and the PU starts to process the task.

2. Instruction Fetch: When an Exchange Jump is executed, the new contents of the program address register are used to locate the first instruction word. This address is sent to the paging mechanism, which computes the actual address. For a normal program start, the first two instruction words (each containing 2 or 3 actual instructions), enter buffer registers "1" and "2" (BR1 and

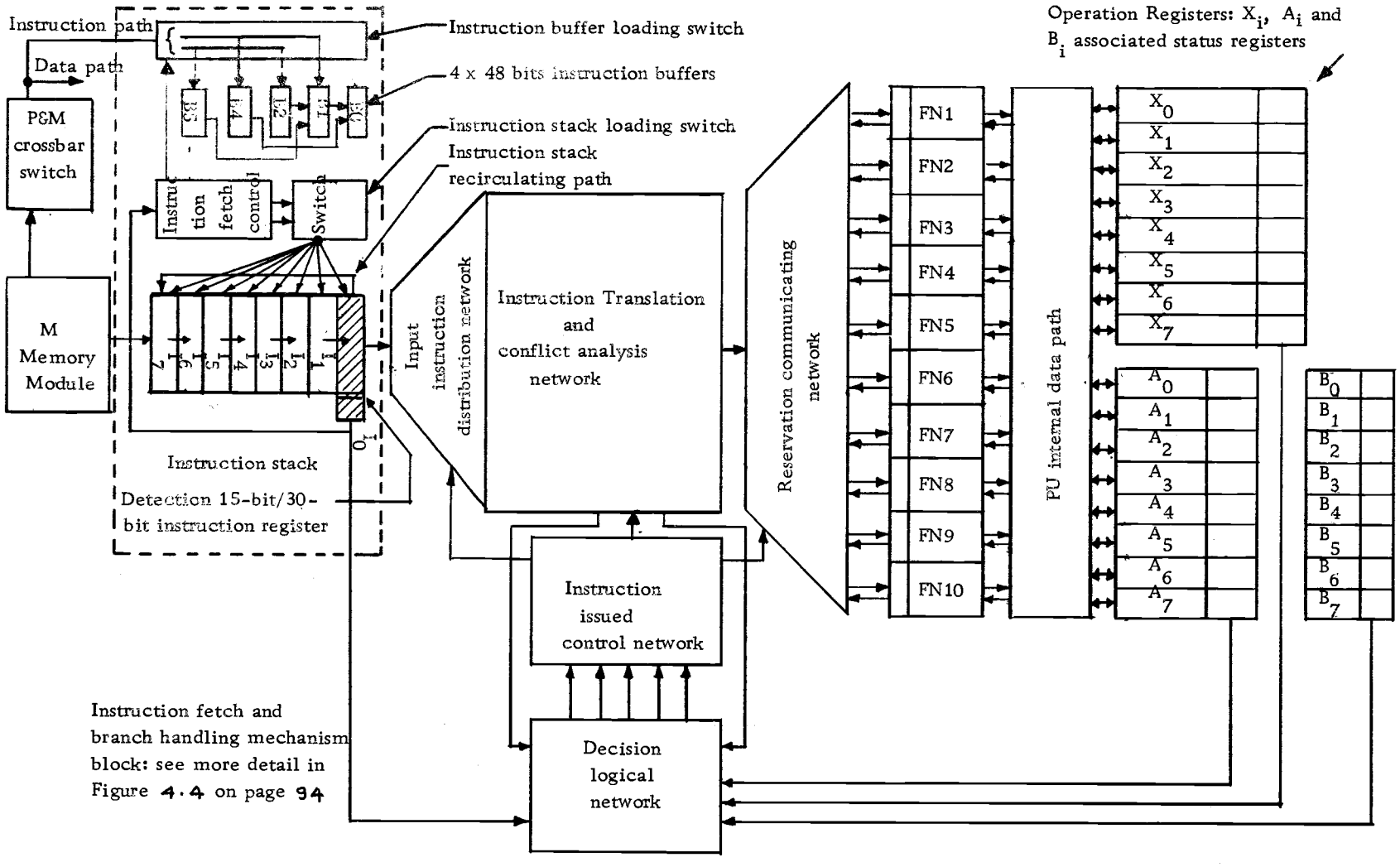


Figure 4.3. The PU reservation control network.

BR2) respectively. From BR1 and BR2, the instructions would move into BR0 and BR1 as shown in Figure 4.4. The two to three instructions in BR0 will be scanned by the prescanning logical switch, the contents transferred to the first available location in the instruction stack (initially, stack (0)), BR1 transferred to BR0, another double fetch initiated, and the process repeated. If the stack fills, the process halts until execution frees a stack location.

3. Branch Handling: If, during the scanning process, BR0 was determined to contain a conditional branch instruction, the conditional branch address will be examined. If the address is outside the instruction stack, the instruction fetch mechanism will fetch the double word at the branch address into BR3 and BR4. If the conditional address is within the instruction stack, issued instructions are circulated back into the stack and no further instruction fetching is initiated. In either case, a hedge is provided so that regardless of the outcome of the conditional branch execution, the next instruction will either be in the stack, BR0 or BR3.

4. Instruction Issue: The reservation control network will issue the instruction under the following conditions:

- The Register designated for the result must not be reserved for a result by a previous instruction.
- The Functional Unit designated must not use as an operand a result of a previous instruction whose execution has not completed.

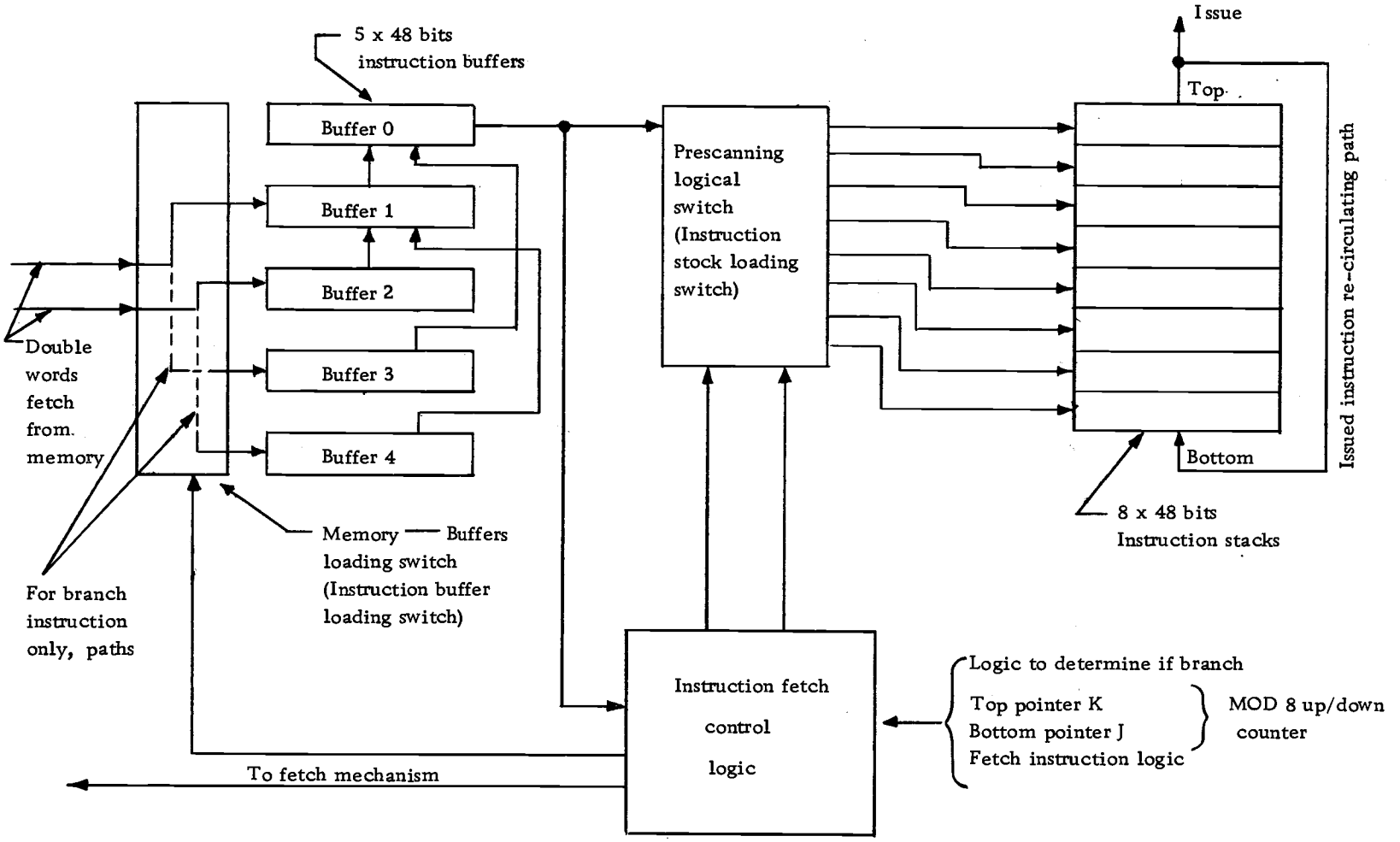


Figure 4.4. The design of PU instruction fetch and branch handling mechanism.

- The Functional Unit designated must not be busy.

Steps 2-4, above, are repeated until the PU is stopped by the execution of an Exchange Jump by the Master Control PPU.

In succeeding sections, details of the instruction fetch mechanism, the instruction decode and issue logic and the Reservation Control network will be given.

4.1 Instruction Fetch and Branch Handling Mechanism

The precedure for fetching instructions is best described by the following procedure:

1. Assume the instruction stack is empty as the initialized state.
2. After the desired memory reference is successful, the fetch operation is performed. Two sequential words are fetched through MDB, P and M cross bar switch, and instruction buffer loading switch.
3. The first word is loaded in Buffer 1 and the second word of double word is loaded in Buffer 2.
4. The test is made to check Buffer 0 is empty. If Buffer 0 is empty, pop the Buffer stack (i.e., Buffer 1 → Buffer 0, Buffer 2 → Buffer 1.) and repeat step 4. If Buffer 0 is not empty, continue.

5. A test is made to check if the stack is full or not. If the stack is full, control sets the top stack pointer K = 0 and sets bottom pointer J = 7. After an instruction word is issued, the stack status is rechecked again. Step 5 is repeated until the empty stack location is available.
6. The contents of Buffer 0 are transferred into the first available stack location (J), the bottom stack indicator is updated and buffer stack popped.
7. A test is made to check if Buffer 0 contained a Branch or not by passing the instruction through the prescanner. If it did continue. If it did not, go to step 16.
8. In Buffer 0, a test is made to find out if the branch address is within the instruction stack or not. If it is not, go to step 12. If it is, continue.
9. A test is made to check if branch has been executed yet. If it has not, wait until it is executed, then continue.
10. A test is made to determine if branch was successful or not. If it was not go to step 16. If it was, continue.
11. Set bottom indicator J = 7, and it will decrement as the instruction stack rotates until the target instruction is moved into stack (0). Go to step 16.

12. A Fetch of the instruction double word at the branch target address is initiated. When completed the double word is loaded into Buffer 3 and 4.

13. The same procedure as in step 9 is performed until execution has completed.

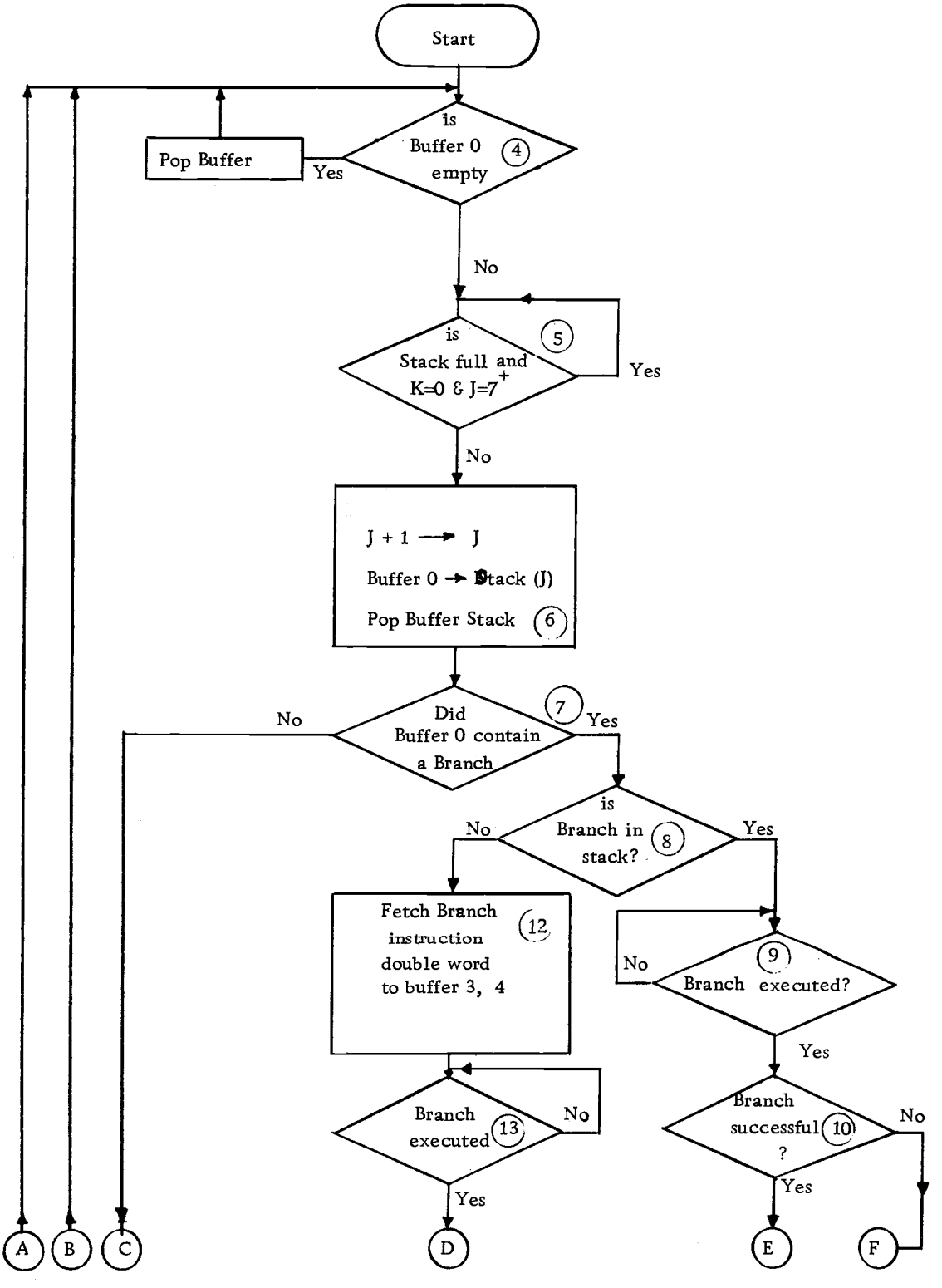
14. If the branch was successful, continue, otherwise, go to step 16.

15. The contents of Buffer 3 $\longrightarrow$ Buffer 0 and Buffer 4 $\longrightarrow$ Buffer 1 simultaneously. The Program Address Counter (PAC) is updated and the buffer stack popped. Go to step 4.

16. If Buffer (0) now contains the second instruction word of a double word continue. If not go to step 4.

17. The control initiates a fetch of the next instruction double word from main memory. Go to step 4.

Figure 4.5 shows the flow chart of Instruction Fetch and Branch Instruction handling. J and K are the bottom and top stack pointers, respectively. The instruction to be next issued is located in Stack (0). Stack (K) is the location of the "oldest" instruction and Stack (J) is the location of the "newest" instruction. Instructions are always circulated in the stack as long as the stack remains unfilled. In other words, an instruction issue causes the stack to rotate, stack (0) $\longrightarrow$ stack (7), stack (i) $\longrightarrow$ stack (i -1), i = 1,..8, and both J and K are decremented. If J = K + 1, (modulus 8) the stack is full.



Flow chart continued over

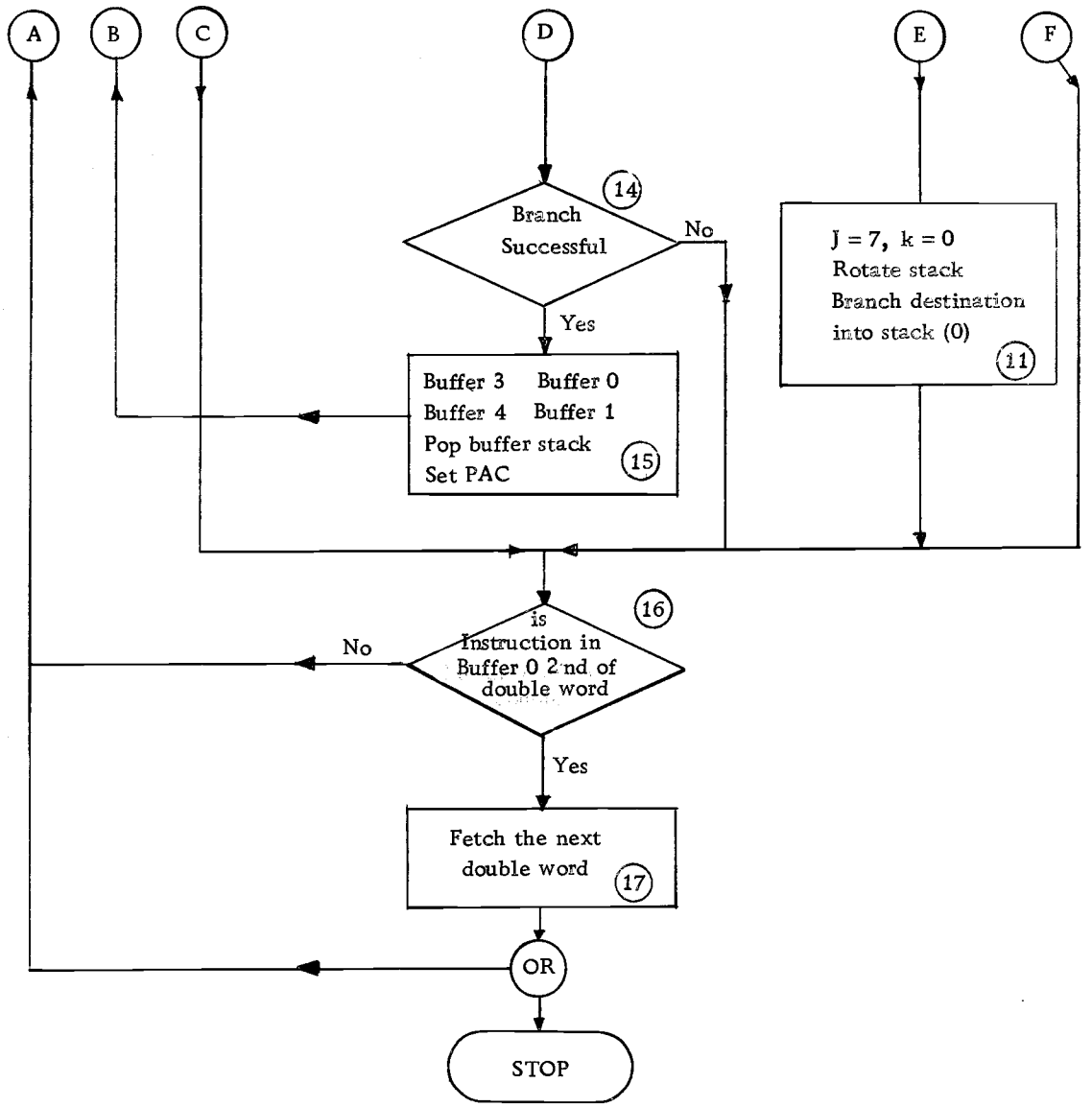


Figure 4.5. Flow chart of instruction fetch and branch instruction handling.

When a new instruction enters the stack from Buffer 0, it does so at stack (J + 1). The only time instructions are prohibited from entering the stack is when the stack is full and K = 0. For this case, the stack holds all non-issued instructions. When K ≠ 0, but the stack is full, the instruction is loaded into stack (J + 1) and both J and K are incremented, destroying the oldest instruction.

To determine if a branch address is within the stack, the program counter and K are compared to the branch address. After successful execution of a branch within the stack, the program counter is updated and both J and K are decremented as the stack rotates the target instruction into stack (0). A branch outside the stack will cause the stack to be cleared.

4.2 Instruction Decode and Issue and Conflict Resolution

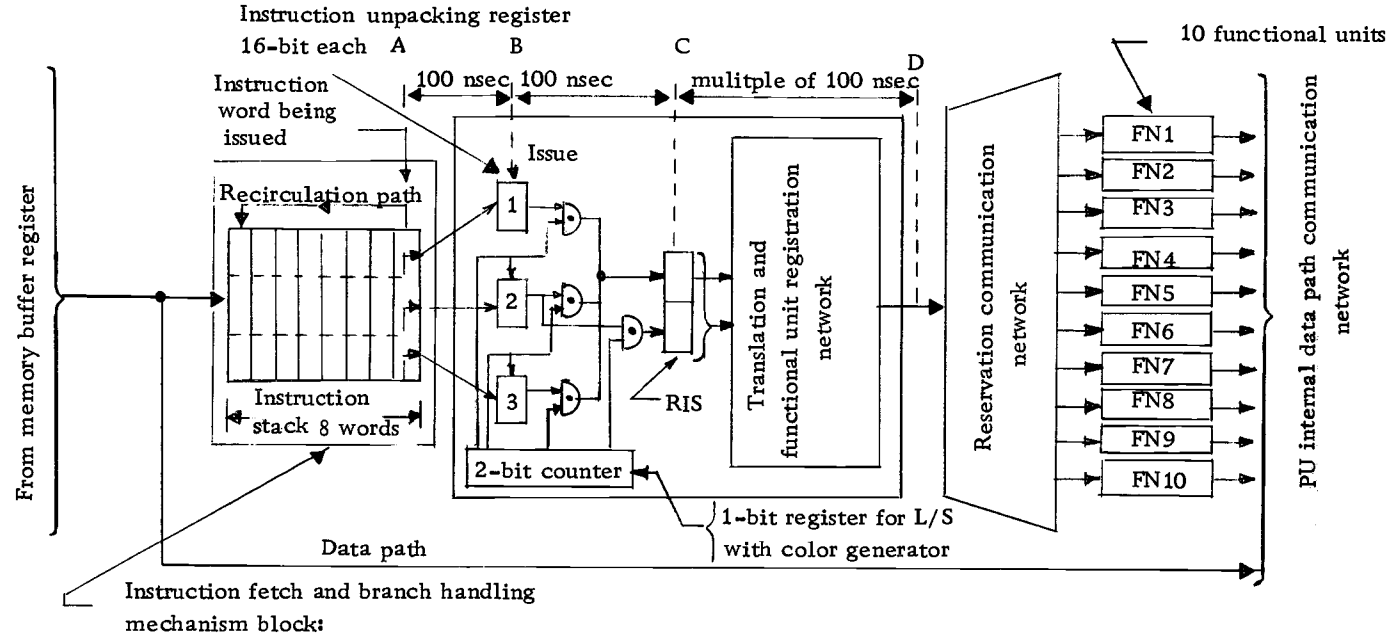
A unique and important part of each PU control is the Functional Unit and Register Reservation Control which are the major components of the Reservation Control Unit. This unit operates the functional units concurrently on a single instruction stream. Several operations are quite independent of others, as the instructions are relatively simple. For example, a sequence of arithmetic or logical operations can often be executed simultaneously with a sequence of control or housekeeping operations. Considerable overlap is possible even in the worst case, as will be shown in Chapter VII.

The major principle of the Reservation Control Unit design is that each new instruction will be issued to its corresponding unit as early as possible in order to allow the following instructions to be issued. In some cases, an issued instruction is held up after issue and has to wait for input operands; at the same time a following instruction may proceed without restraint.

The general structure and organization block diagram of the reservation and control network was shown in Figure 4.3. Details of the Instruction Translation and Conflict Analysis network and the timing are shown in Figure 4.6. The overall structure consists of ten associated blocks:

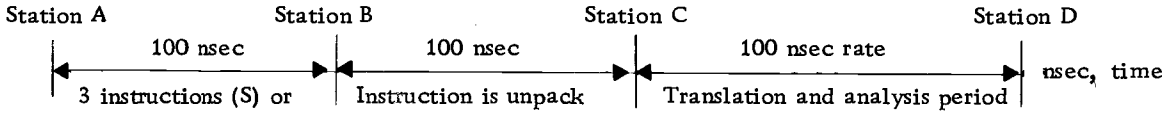
1. Instruction stack.
2. Input instruction distribution network.
3. Instruction translation and conflict analysis network.
4. Reservation communication network.
5. Functional unit private status registers.
6. Operation (X) register private status registers.
7. Address (A) register private status registers.
8. Index (B) register private status registers.
9. Instruction issue control network.
10. Decision logical network.

The instruction stack, as was mentioned earlier, consists of eight 48-bit registers. Each of these instruction registers can contain



(a) Details of instructions un-pack and flow to functional units

Note:
L = Long instruction
S = Short instruction
RIS = Read to issue register



(b) Timing diagram of instruction flow (both short and long instructions).

Figure 4.6. The illustration of instructions flow in reservation control network and timing diagram.

three 16-bit instructions (short instruction) or one 32-bit instruction (long instruction) and one 16-bit instruction. In normal operation, a 48-bit word that contains instructions is fetched sequentially from the main core memory module and enters first available location of the instruction stack through the Input buffer registers as described in Section 4. 1.

The instructions are moved through the stack for issuing and execution. It is assumed that 50 nsec. are required to shift up one register. For branch operation within the stack, the instructions which are contained in registers I_1 through I_7 can be automatically rotated so that the branch address instruction is ready to be issued; this automatic rotation is accomplished by the association branch handling mechanism and requires 50 nsec. for each transfer.

A logical circuit is provided at I_0 register to detect whether the contents include a long instruction or not. If so, this detecting logical circuit will notify the necessary control points, such as Decision logical network, translation and conflict analysis network, etc.

The Input Instruction Distribution Network (Figure 4. 3) is provided to connect the instruction path between the instruction stack and the instruction translation and conflict analysis network. Both short and long instructions are unpacked here, short instructions are loaded in the 16-bit registers 1, 2, and 3 respectively and sequentially

transferred to 32-bit RIS (ready for issue) register at 100 nsec. intervals. The second part of the RIS register is ignored for short instructions. The transfer is accomplished by the cooperation of the instruction issue control network and the associated 2-bit counter. For long instructions, the 16-bit registers 1 and 2 are used simultaneously (considered as a combined 32-bit register). A long instruction is transferred from register I_0 to the RIS register under control of the L/S (long or short instruction) 1-bit register, 2-bit counter and instruction issue control network. It is transferred at the same rate as a short instruction, but both parts of the RIS register are considered.

The Instruction Translation and Conflict Analysis Network is a logical network consisting of a set of 32-bit registers for an instruction that is in the translation and conflict analysis state. This analysis is performed in a pipeline fashion. During the translation state, the operation code is interpreted, the appropriate functional unit status is checked, and the first available functional unit is considered, if no functional unit conflict exists. During the conflict analysis state, the address fields associated with the source (operand) registers and the destination (result) register are interpreted, and the status of the corresponding X, A and B registers are checked. If no conflict exists the analyzed instruction is completely issued and the engaged functional unit starts its operation. After the operation

is completed, the result of the computation is transferred to the result register (X_i, A_i or B_i) and the corresponding status register is updated. The issuing cycle is repeated for a new instruction.

The Reservation Communication Network provides the control signal communication path between the instruction translation and conflict analysis network and the ten functional units, eight operation registers (X), eight address registers (A) and eight index registers (B). The status of the specified functional unit and the specified registers are directly reported to the Decision Logical Network to decide whether the current instruction is ready for issue or not. When decision making is performed involving the current instruction issuing the executive signals are sent to the Instruction Control Network. The appropriate control signals are generated in this Instruction Control Network and sent to the necessary control points in the PU.

In the performance of instruction conflict resolution during the instruction issuing phase, it is necessary that the reservation control network know the status of each PU functional unit, each operating register (X), and each address register (A) and the identity of relevant index registers (B).

A 14-bit function unit private status register is associated with each functional unit. It is divided into four consecutive fields as shown in Figure 4.7. The mode field of an instruction (m) will

identify which kinds of registers (X, A, B) are associated with the referenced instruction.

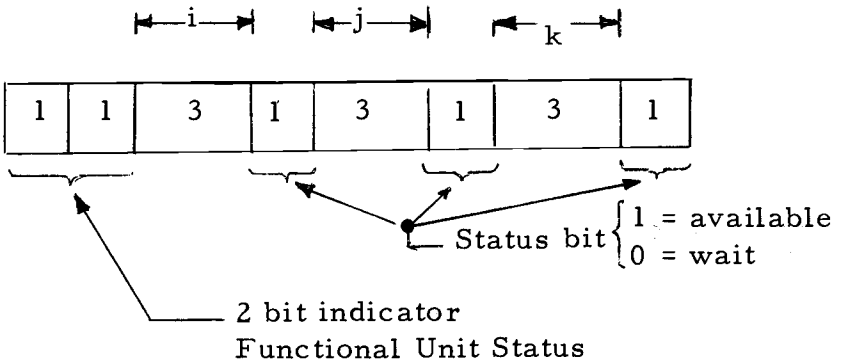


Figure 4.7. FU status register.

where j = 1st operand

k = 2nd operand

i = result

status register is clear when its status is update.

A 6-bit register is associated with each PU 48-bit operation register X_0, X_1, X_2 ..., X_7. It is divided into two consecutive fields as in Figure 4.8. The two leftmost bits represent the status and the four rightmost bits represent the number of the associated functional unit.

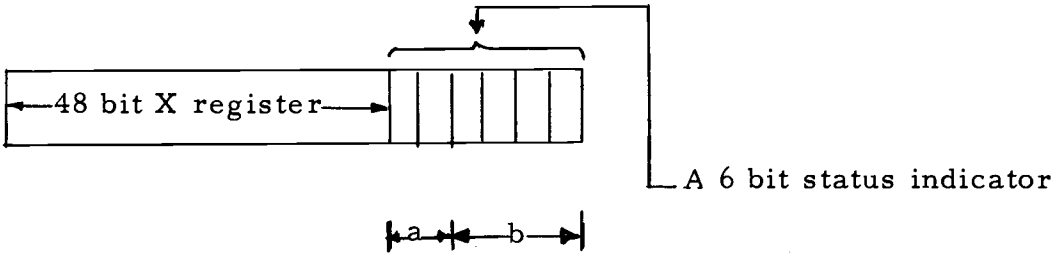


Figure 4.8. Operation register X_i with status indicator.

Field a :

00 = X_i is free.

01 = X_i is reserved to store the result for the associated functional unit.

10 = X_i holds the 1st operand for the associated functional unit.

11 = X_i holds the 2nd operand for the associated functional unit.

Field b : indicates the number of the associated functional unit.

The indicator bits of operation register X_i are cleared when the status of register X_i is updated.

A similar 6-bit register is associated with each PU 18-bit address register A_0, A_1, A_2, ..., A_7 as its private status indicator

whorn in Figure 4.9. The operating function of the status indicator is the same as that for the operation registers.

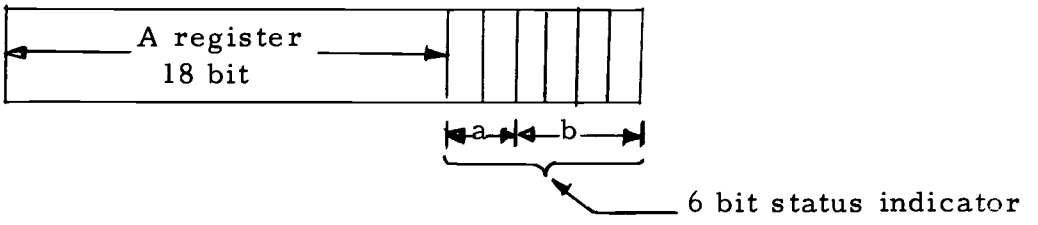


Figure 4.9. Address register A with status indicator.

A 6-bit register is also associated with each PU 8-bit index register B_0, B_1, B_2, ..., B_7 as its private status indicator. The operating function of the status indicator is the same as that mentioned for the X registers. Figure 4.10 illustrates the details of 8-bit index register B and its corresponding status indicator.

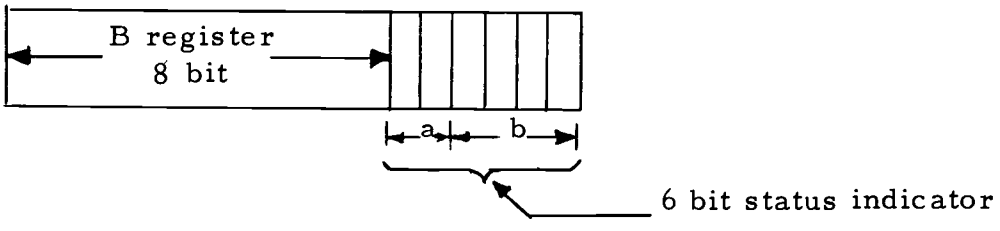


Figure 4.10. Index register B with status indicator.

The logical organization of these four status indicators is shown in Figure 4.11. Physically, they form part of their corresponding registers or functional units, however they can be viewed as a

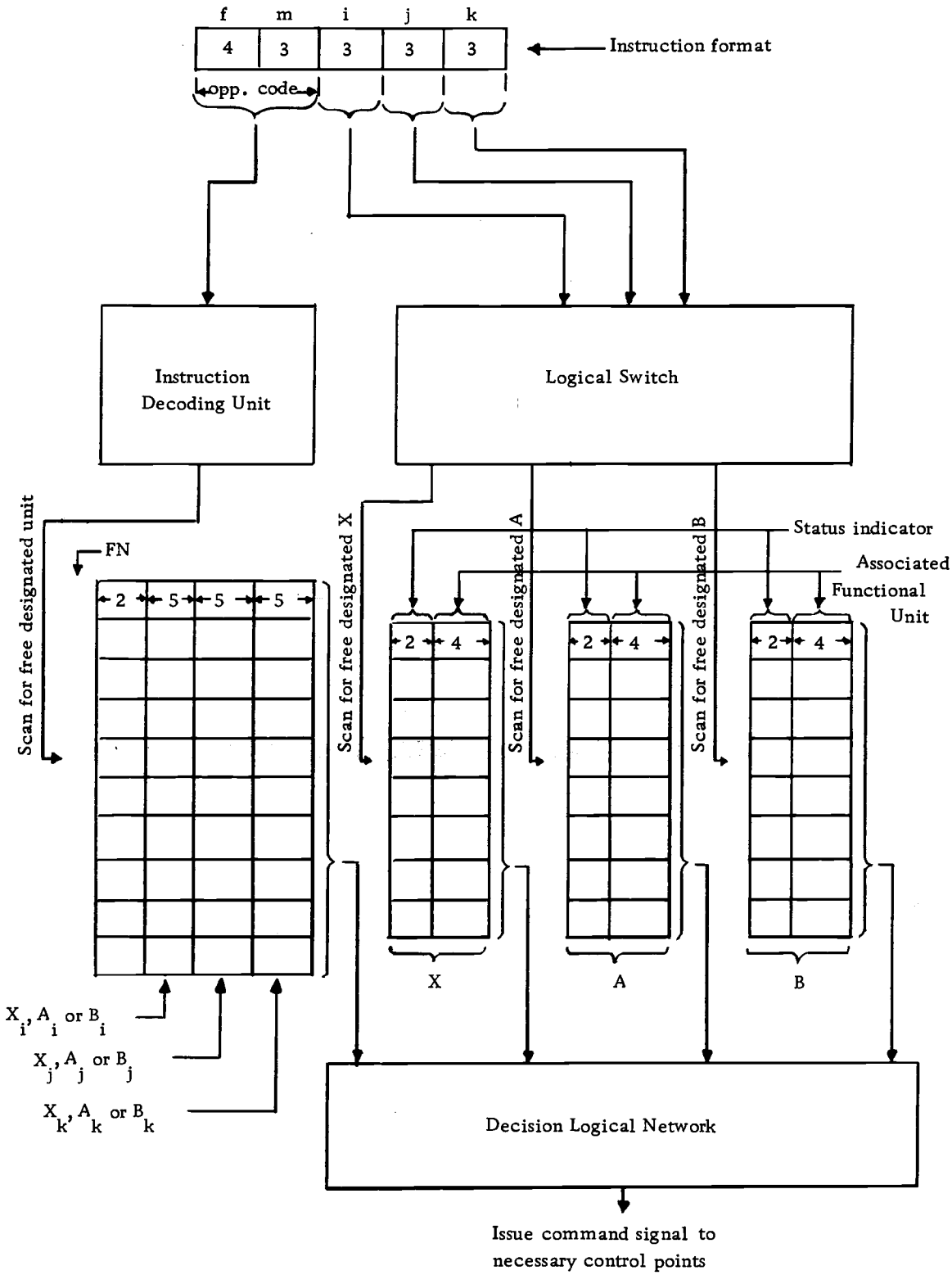


Figure 4.11. Illustration of Logical Scoreboard (logical collection of status indicators) and normal instruction issue reservation block diagram.

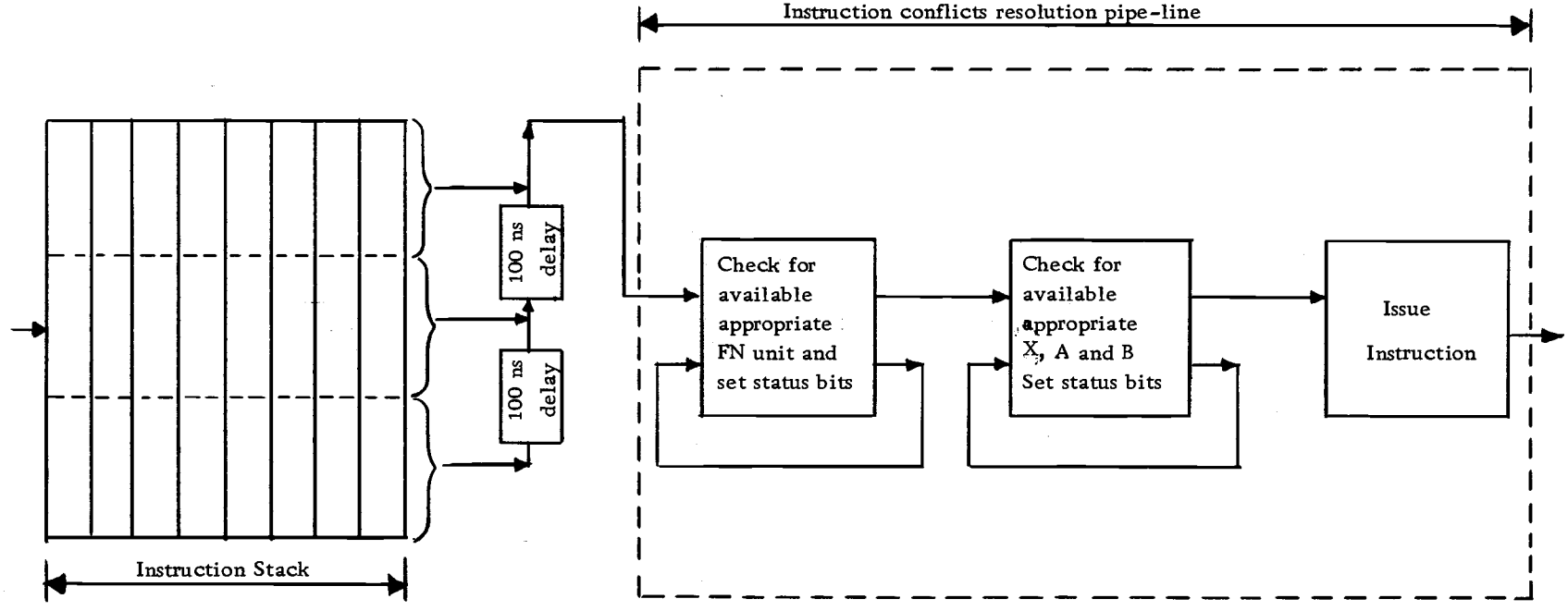


Figure 4.12.Logical block diagram of steps of instruction issue.

collection of four tables used by the logic to resolve conflict conditions.

The algorithm for instruction issue can be described in the following steps:

1. Scan the unpacked instruction in the RIS register.
2. Check to find whether an appropriate functional unit is available or not by scanning the functional unit status registers corresponding to the instruction F field. If an available functional unit exists in the PU, then the first such functional unit is engaged by setting its corresponding status register. If all the appropriate functional units are busy, the control has to wait until the first available unit is free and then it sets its status bits (i.e., the instruction is temporarily held).
3. Check the status indicators for the operand registers (X_j, A_j, or B_j), (X_k, A_k, or B_k) and the result register (X_i, A_i, or B_i) to see if there is a conflict. Types of conflicts are summarized in Table 4.2. If there is no conflict, then set the appropriate status bits in the register status indicator, set the functional unit status and issue the instruction; go to step 1. If there is a conflict, set the wait bits in the free unit's functional unit status register. Issue the instruction,but inhibit further instructions until the

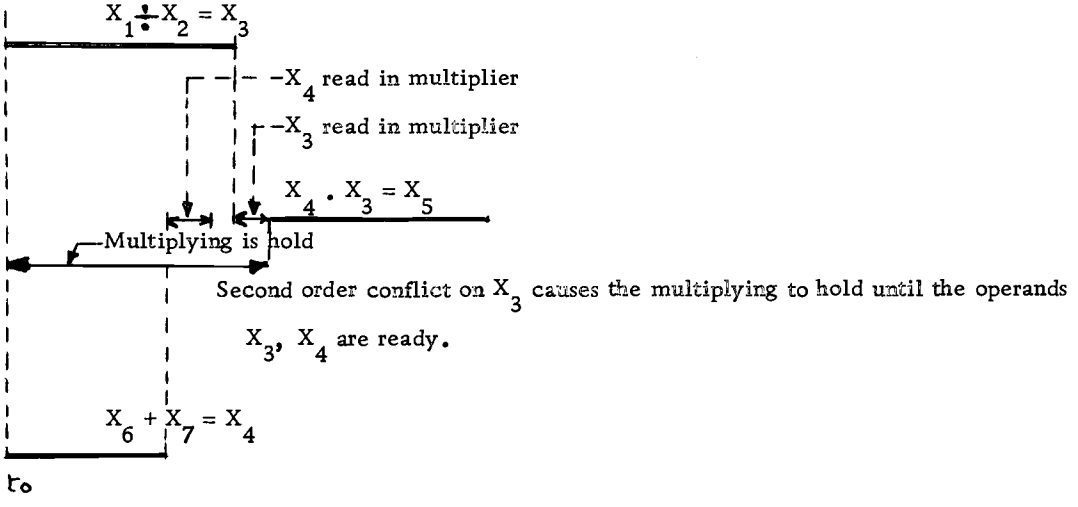
conflict is resolved. On execution of an issued instruction, the status of the associated registers, $X_{(j, k, i)}$ and $A_{(j, k, i)}$ and $B_{(j, k, i)}$ are updated. The functional unit status registers are again examined and the appropriate, if any, wait bits are reset. At this point if conflicts are resolved, go to step 1, otherwise wait.

A detailed example of PU instruction overlap for a specific application program is given in Chapter VII. A list of PU instructions is included in Appendix A.

From the above description of the essential features, it is clear that the PU has been designed to achieve high performance processing with minimum cost by utilizing low cost modular LSI hardware components. Since the PU employs multiple functional units, multiple local storage registers (X, A and B), instruction stack and multiple data paths, it can perform instruction level parallelism and also provide a high degree of reliability. The 48-bit word size is selected to maximize the trade-off between machine capability and cost.

Table 4.2. Type of instruction execution conflicts.

Example of conflict configuration	Type of conflict	Remarks
$X_3 = X_1 + X_2$ $X_5 = X_3 + X_4$	First order conflict case (a).	This is a conflict between instructions which use the same functional unit, case (a), and use the same result registers, case (b).
$X_3 = X_1 + X_2$ $X_3 = X_4 + X_5$	First order conflict case (b).	
$X_3 = X_1 + X_2$ $X_5 = X_4 / X_3$	Second order conflict ($X_3 = X_1 + X_2$ is source instruction and X_3 is source or input operand of $X_5 = X_4 / X_3$)	This conflict occurs when an instruction requires the result of a previously issued instruction, but the source instruction or input operand is not accomplished yet.
$X_3 = X_1 / X_2$ $X_5 = X_3 \cdot X_4$ $X_4 = X_6 + X_7$	Third order conflict	This conflict occurs when an instruction is called on to store its result in a register which is to be used as an input operand for a previously issued instruction, but it is not yet started. This order of conflict can be resulted by holding the result in the functional unit.



V. MAIN MEMORY SYSTEM SPECIFICATION

There is no special requirement on a memory to operate in a multiprocessor environment. In fact, a conventional module of 4K (4096-48 bit words) to 8K (8192-48 bit words), would be adequate for use as a standard storage unit in this command and control computing system. Each memory module has its own address register, data register, parity generator and checker, and module assignment register. This latter register contains the memory module number needed for the system reconfiguration purposes. In this way the physical module number is independent of the logical module number so that system software may assign any number to a memory module.

The choice of 4K to 8K words per module is mainly derived from the cost point of view. Even in the present time, the magnetic core memory still dominates the market, and the lowest dollar per bit figure is obtained from the 4K to 8K size systems.

From the viewpoint of efficiency, the system favors a small module size. For the total amount of memory capacity, the smaller module size will offer less possibility for memory reference conflict from competing processors. Smaller modules are also advantageous from the reliability sense, that is, in case of failure, less amount of memory capacity is to be taken out of the system. Hence, a 4K (4096-48 bit word) module seems to be a suitable choice in this design.

Therefore, a memory module containing 4K words of 52 bits (48 bits including sign, 8 characters, 6 bits each, plus 4 bit parity), is selected as a standard memory module. The memory modules are connected to the computing system through a crossbar switching network.

The 18 bit address field is split as shown in Figure 5.1. The least significant 12 bits of the address are used to define the location of a word within a module and the most significant six bits address the particular memory module.

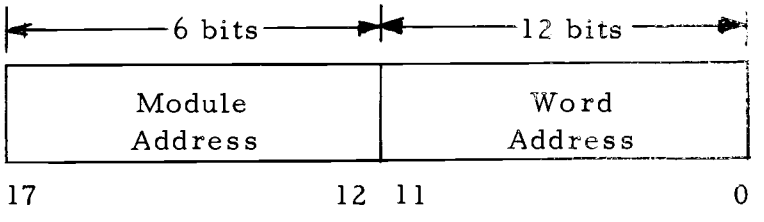


Figure 5.1. 18 bits address format (actual address).

Since the crossbar switch network is modular and easily expandable, it is not necessary to pose any restrictions on the total memory capacity. The ultimate choice of memory capacity is dependent on system requirements, cost, and circuit limitations. Again, for preliminary design discussion purposes, we will assume that this command and control computing system will have a maximum capacity of 16 4K-modules. Since the system employs an 18 bit address field (6 bit memory module address and 12 bit word

address); the system has capability of handling up to 64 4K-modules for maximum future expansion.

If the system employs 16 4K-memory modules, only 4 bits are used for the memory module number assignment registers (6 bits are provided in hardware); and 12 bits for the 4K words, hence a total 16 bit address field is used (18 bits are provided in register hardware).

Inclusion of the memory module assignment register does influence the design and operation of the crossbar switch network control logic. The control logic of the memory access path, instead of responding to a single request line from a processor, has to incorporate a 6 bit comparison circuit to compare the high order 4 bits of an address field from a processor with the contents of each memory module assignment register. A match, (the result of the above mentioned 6 bit comparison) together with the processor request produces an effective memory request signal to be evaluated by the conflict resolution logic.

The logical interconnection of the PU, Effective Memory Request Signal Generator, Address path and Memory Module is illustrated in Figure 5.2. The physical data route between PUs and Memory modules including the crossbar switch is shown in Figure 5.13.

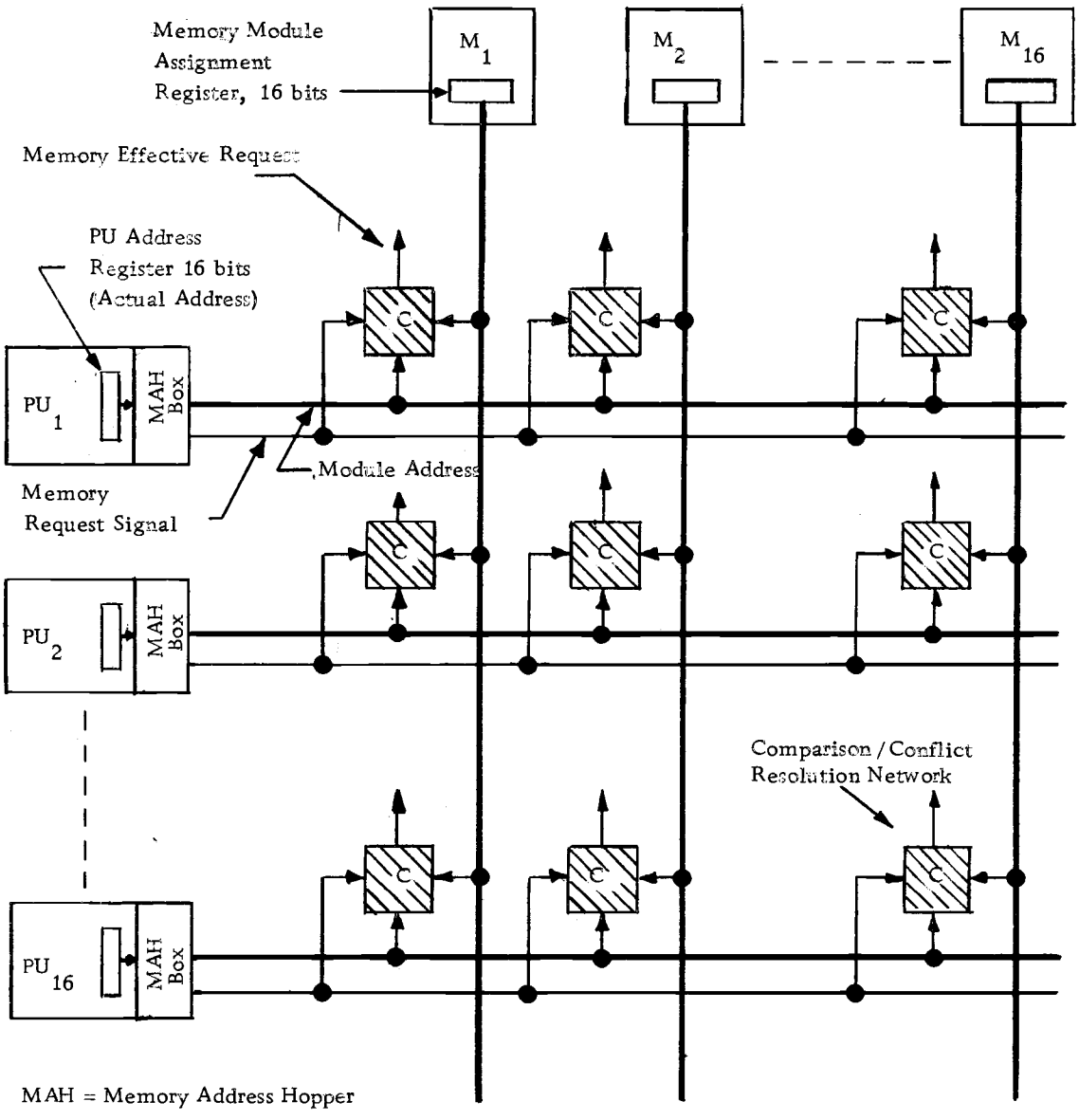


Figure 5.2. Address route of Memory access logical block diagram.

This proposed system attempts to reduce memory delay by using memory interleaving. Each memory module (4096 words) is divided into four banks each containing 1024 words, where each bank contains a subset of three memory addresses. From the results of the analysis of second level parallelism in Chapter VII, it is clear that memory interleaving is very significant technique not only for reducing memory delay but also making the second level of parallelism more meaningful.

5.1 Virtual Memory

To design a computing system in which several processors share the same memories or several programs (or tasks) share the same processor, one must deal with the problem of storage allocation and memory reference conflicts. It is necessary to look for a technique for storage allocation without extensive data moving which will reduce the degree of memory conflicts.

It is assumed that the main storage is divided into logical regions, allocated on a page basis. Swapping techniques are applied to handle transfers between internal main core memory and external storage on a page demand basis. There are advantages and disadvantages to implementing such a virtual memory scheme.

Since the concept of virtual storage used here employs paging, allowing the system to allocate memory to a large number of programs

in the same period of time, it provides support to the multi-processor system in performing parallel task processing. User programs in a multiprogramming system must be protected against one another; otherwise, a program which has been checked out and is known to be correct may start producing wrong results because another program has interferred with it. Virtual addressing can be used for security and protection. Every storage reference produced by a process can be checked to ensure that it lies within the virtual address space assigned to that process. If it does not, the process can be stopped and the reference suppressed before any damage is done. Many user programs can share common data, subroutines, compilers, etc. which reside in the main core memory by using a paging scheme.

For every memory reference in a virtual storage system, the virtual address needs to be converted into an actual core storage address. This process increases the overhead computing time. If the referenced location is in a page which is already in the main core memory, it requires near normal access time, but when the corresponding page is not in the main core memory, a transferring process must be introduced between the main core memory and back-up memory, increasing access time. On the average, the paging system requires longer access time than the nonpaged system. Fragmentation is one nightmare that every operating system designer faces. It occurs whenever user program memory requirements do

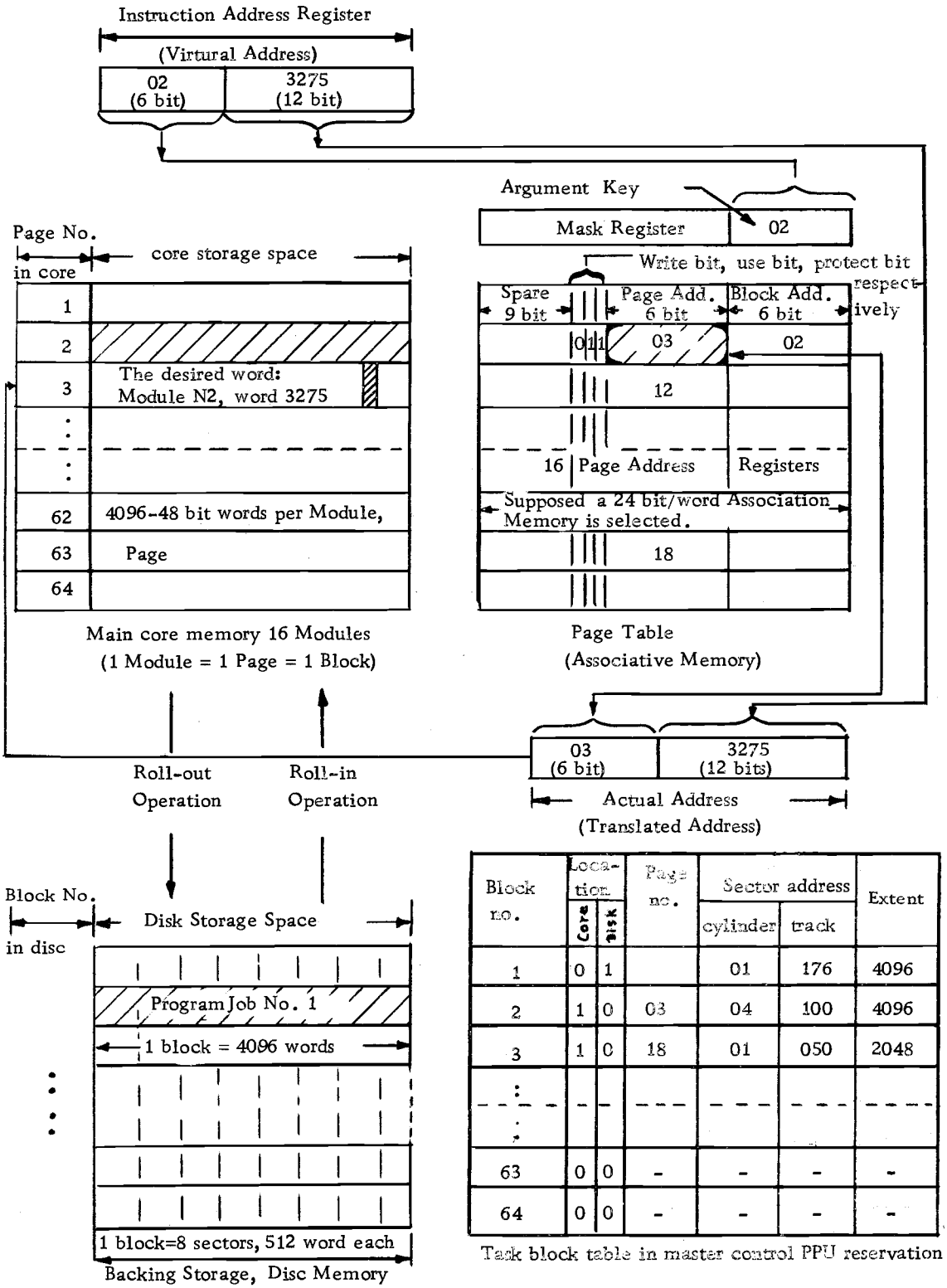
not match the available page size. Fragmentation requires more pages. Swapping a page out of the main core memory just before it is needed is very common. The worst case where the system spends all of its time swapping pages is called thrashing.

There are several reasons for and against using virtual memory techniques for memory management and storage allocation for this computer system. Essentially, it will allow a program to be sub-divided and the sections to be run in parallel, in order to speed up its execution time and to take advantage of parallel architecture and provide necessary security and protection. Thus implementation of a paging technique in this command and control system design will increase the system performance (32).

The virtual storage scheme which will be implemented in this command and control system has the following features:

1. page size equal to memory module size.
2. sector (back up storage) equal to page size.
3. page table implemented in each PU with associative memory.

The translation between the virtual address and the actual address is done by using a page table where each entry of page table corresponds to a module of main core memory, and contains the number of the block currently occupying that page, together with three independent binary digits called a usebit, a writebit and protect bit respectively.



Block no.	Location		Page no.	Sector address		Extent
	Core	Disk		cylinder	track	
1	0	1		01	176	4096
2	1	0	03	04	100	4096
3	1	0	18	01	050	2048
:						
63	0	0	-	-	-	-
64	0	0	-	-	-	-

Task block table in master control PPU reservation

Figure 5.3 Address translation and paging mechanism in each PU interacts to system main storage and booking storage for storage management and allocation.

An associative memory (AM) will be used as a page table with one associative memory associated with each PU. The size of AM is 64 words, corresponding to the maximum number of pages that can be assigned to a task. Contents of words in the associative memory are compared with the address register. A match will yield the corresponding physical module address for the virtual page address in the register. A block diagram is shown in Figure 5.3.

The memory management policy can be summarized as follows:

1. When the Master Control PPU assigns a task to a specific PU for execution, it allocates the required amount of virtual storage blocks to the task. A block table is created by the PPU with 64 entries corresponding to the maximum number of blocks which can be allocated to a task. Each entry specifies the location of the block (central memory, mass memory or not existent), the sector address in mass memory, the page address if in core and the extent (see Figure 5.3).
2. The scheduler function of the Master Control PPU allocates an initial group of physical pages (modules) to the PU, updates the block table and makes an entry in the PU Associative Memory page table.
3. Address translation is handled in the PU.

For example, when the processing unit (PU) executes an instruction with a block address of 02_8 and a word address of 3275_8 within the block:

a. The block address part (02) is extracted from the instruction.

b. This block address is used as an argument key to access the associative memory.

c. By searching the associative memory, the physical block address, page 03 is obtained from the associative memory and combined with the operand address (word address within the block, 3275) to form the actual address to be used in accessing the desired operand in the memory module.

4. During address translation, a search is performed on the associative memory page table for a match to the virtual block address. If a match is encountered in the AM page table, the desired program block is in the central memory and will be accessed. When no match is found in the AM page table, a page-fault has occurred. The address translation mechanism generates a page fault interrupt and activates the Master Control PPU. The type of page fault interrupt generated depends on which of the following two situations is encountered:

a. The mentioned block is not existent. In this case, the system checks that the total space allowance requested by the program will not be exceeded by creation of the new block, then selects a proper page in the core storage, and makes the appropriate correction to both the AM page table and the block table.

b. The block address is in backing storage. In this case, the task block directory is scanned for the address on the disc, drum, or mass storage of the desired block. When this is found, the desired block is read into the main core storage, and its number is placed in the AM page table and the block table.

5. The memory management scheme used by the Master Control PPU is a variation of the Least Recently Used (LRU) algorithm. Upon receiving a page fault interrupt, the PPU determines if core page space is available. If space exists, the PPU proceeds as outlined in 4b, above. Otherwise, an active page must be moved to backup storage before a new block is entered. Each entry in the AM page table for each PU has a "use bit". This use bit is set to 1, whenever the block corresponding to that page is accessed by the PU. At fixed time intervals, the Master Control PPU interrogates the AM page tables. The use bits indicate which pages

have been accessed in the last time interval. The use bits are reset after interrogation. This information is used by the memory management scheme to determine the least recently referenced page. In case of ties, the Master Control will "roll out" the oldest least recently used page.

6. The write bit in each AM page table entry indicates whether the block has been altered while occupying a physical page of central storage. An unaltered page need not be transfered out to back up storage, since a copy is kept in back-up storage when a block is brought into central memory. This reduces transfer time.
7. The third status bit called the protect bit is used to prevent the PU from altering the page of central. This provides protection for shared memory between processor units.

5.2 Memory Reference Conflicts

In a multiprocessor system sharing resources, an unavoidable problem is memory reference conflict. This one factor can severely reduce the performance of the system. A memory reference conflict in this two-level parallel processing system can occur from the following causes:

1. Due to instruction overlap within a PU, two or more independent functional units in the same PU may make

reference to the same memory module at the same time.

2. Due to the multiple PU structure, there are memory reference conflicts when two or more PUs try to access the same memory module simultaneously.

The system was simulated on the Oregon State University Computer Center CDC 3300 using the GPSS III language. The model used is shown for two examples in Figure 5.4 (b) and Figure 5.5 (b) for one processor and two processors, respectively. The equivalent physical systems for these examples are shown in Figures 5.4 (a) and 5.5 (a). The following simulation parameters are defined:

Pij - probability of a request from PPU, Pi to memory module, Mj where

$$\sum_{j} P_{ij} = 1$$

$1/\lambda$ - memory request arrival interval

$1/\mu$ - memory access (service) time -

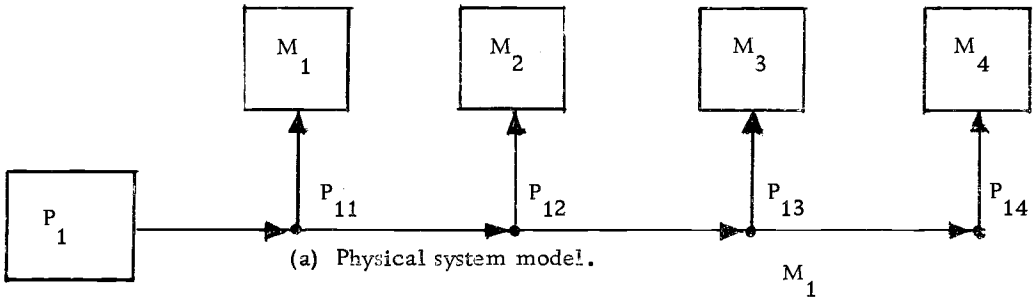
P - number of processors in the system

M - number of memory modules in the system

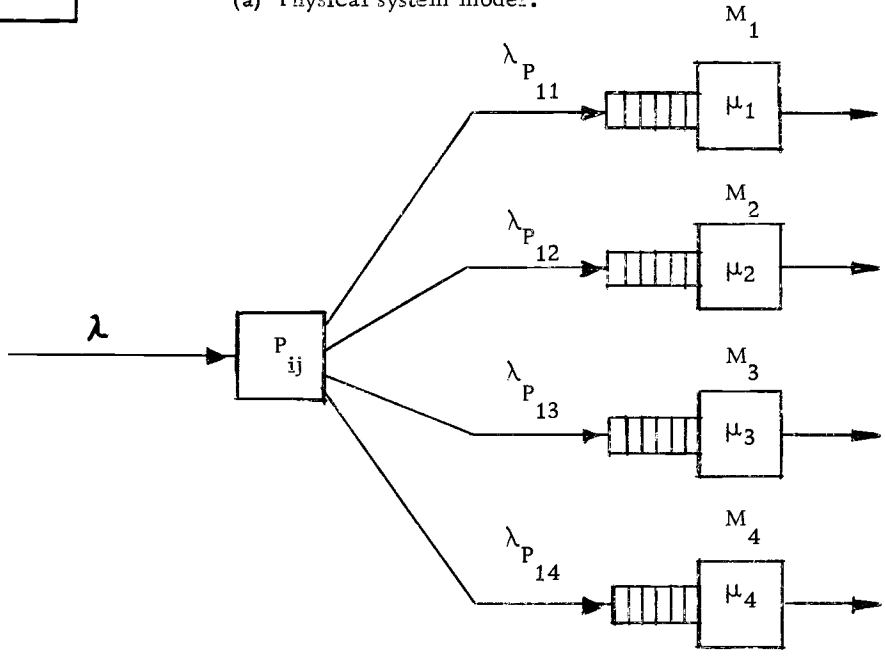
M-P ratio - the ratio of memory modules to PPU's in the system.

The service time μ was set as a fixed value of 1000ns, a major machine cycle. Various values of λ were simulated with arrival times Poisson distributed with the intervals.

Probability distributions were developed for random memory sharing, a high degree of memory sharing and a low degree of memory



(a) Physical system model.

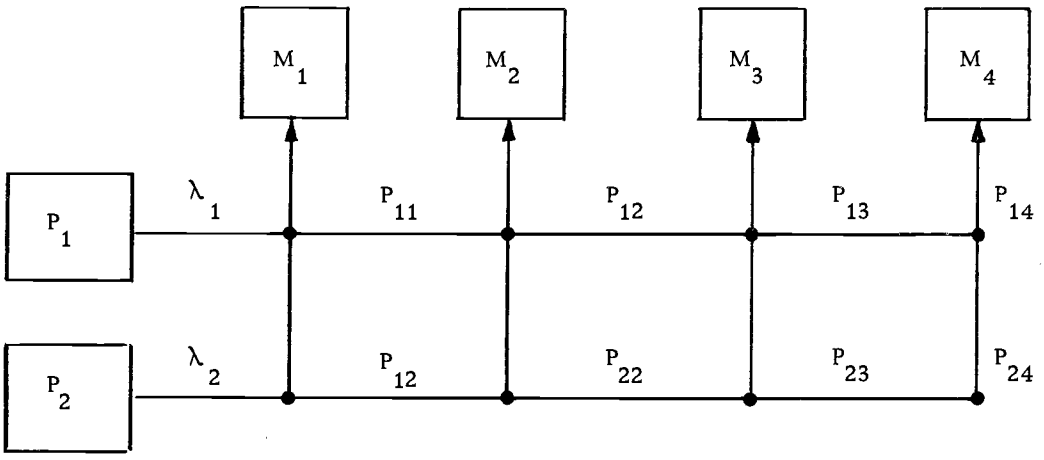


(b) Simulation model.

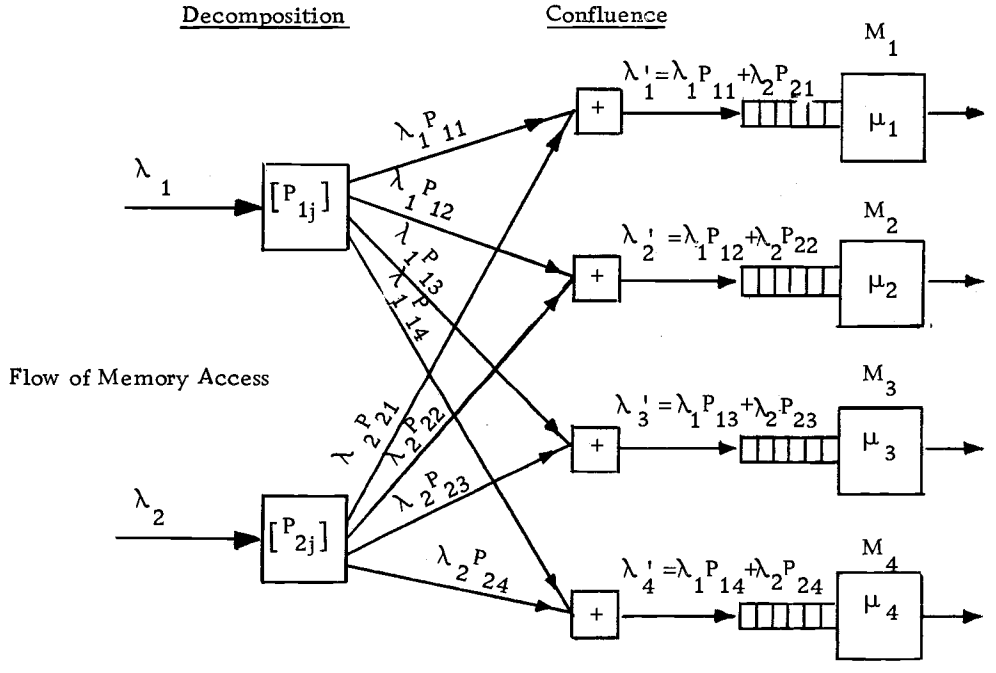
Trials	Processes	M_1	M_2	M_3	M_4	P_{ij}
		P_{11}	P_{12}	P_{13}	P_{14}	
1	P	0.40	0.30	0.20	0.10	1
2	P	0.10	0.40	0.30	0.20	1
3	P	0.30	0.20	0.10	0.40	1
4	P	0.20	0.10	0.40	0.30	1
5	P	0.20	0.30	0.40	0.10	1

(c) Random selected probability of flow

Figure 5.4. Simulation example (P = j, M = 4).



(a) Physical system model.



(b) Simulation model.

Figure 5.5. Simulation example. (P = 2, M = 4)

sharing. A typical example for a single processor system is shown in Figure 5.4 (c). The plots reflect an average of these three cases.

The following relationships were obtained from the simulation and plotted in Figures 5.6 through 5.10.

1. Average memory conflicts per module v.s. M-P ratio.
2. Average memory conflicts per module v.s. M and P.
3. Average successful access per second v.s. system relative cost.
4. Normalized memory conflict rate per module and normalized system cost v.s. M-P ratio.
5. Average percent of memory utilization per module with varying the rate or memory access (i.e. $\frac{1}{2}$ = 250, 500, 700 ns) v.s. M-P ratio.

Normalized system cost was calculated as follows:

K_M = cost of a memory module

K_p = cost of a processor unit

K_h = lumped cost of other hardward (PPU, crossbar switch, I/O devices, etc.)

P = number of processors

M = number of memories

I = M/P, the memory-processor ratio.

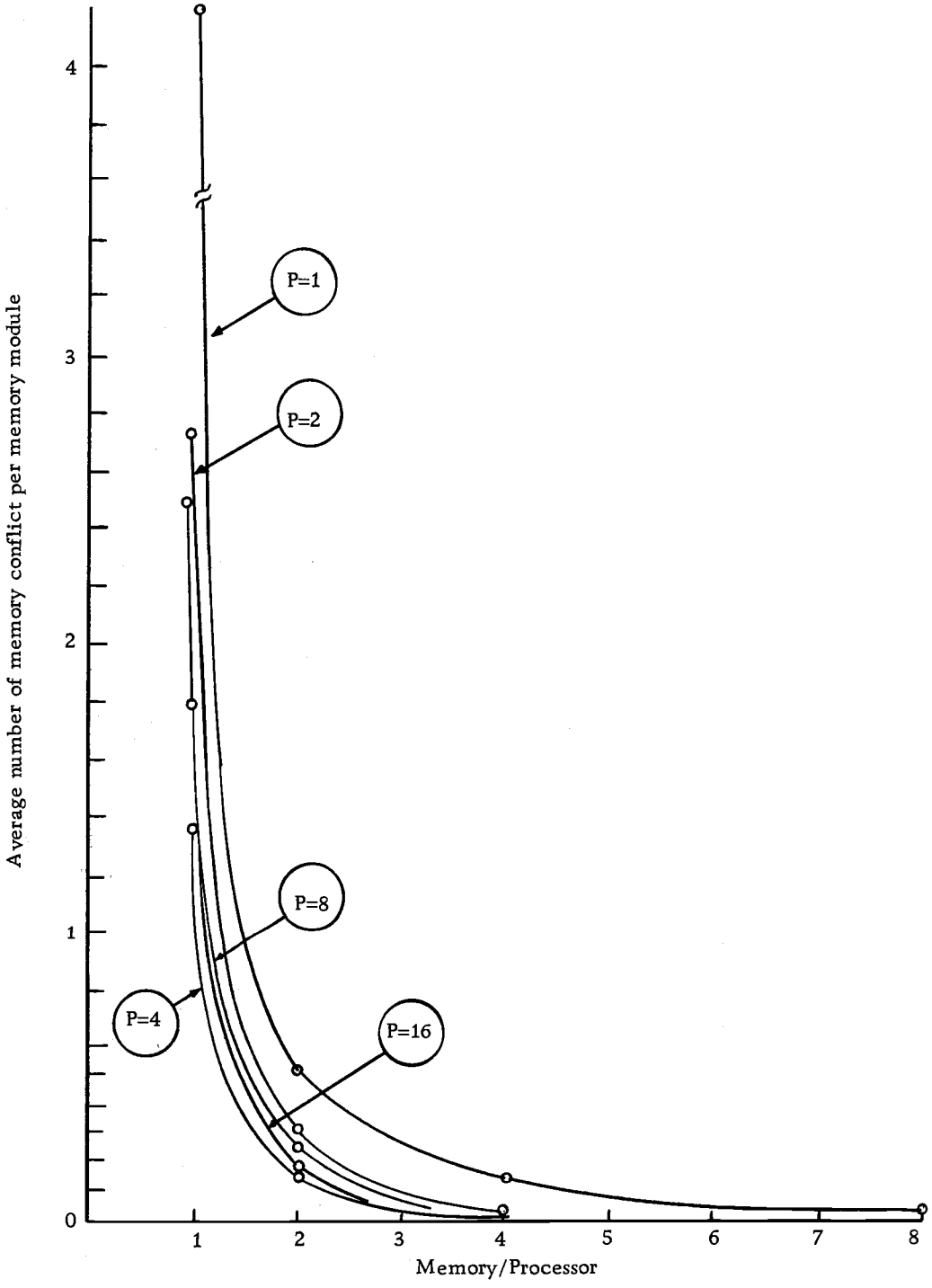


Figure 5.6. The average memory conflict per module as a function of the memory-processor ratio in the system.

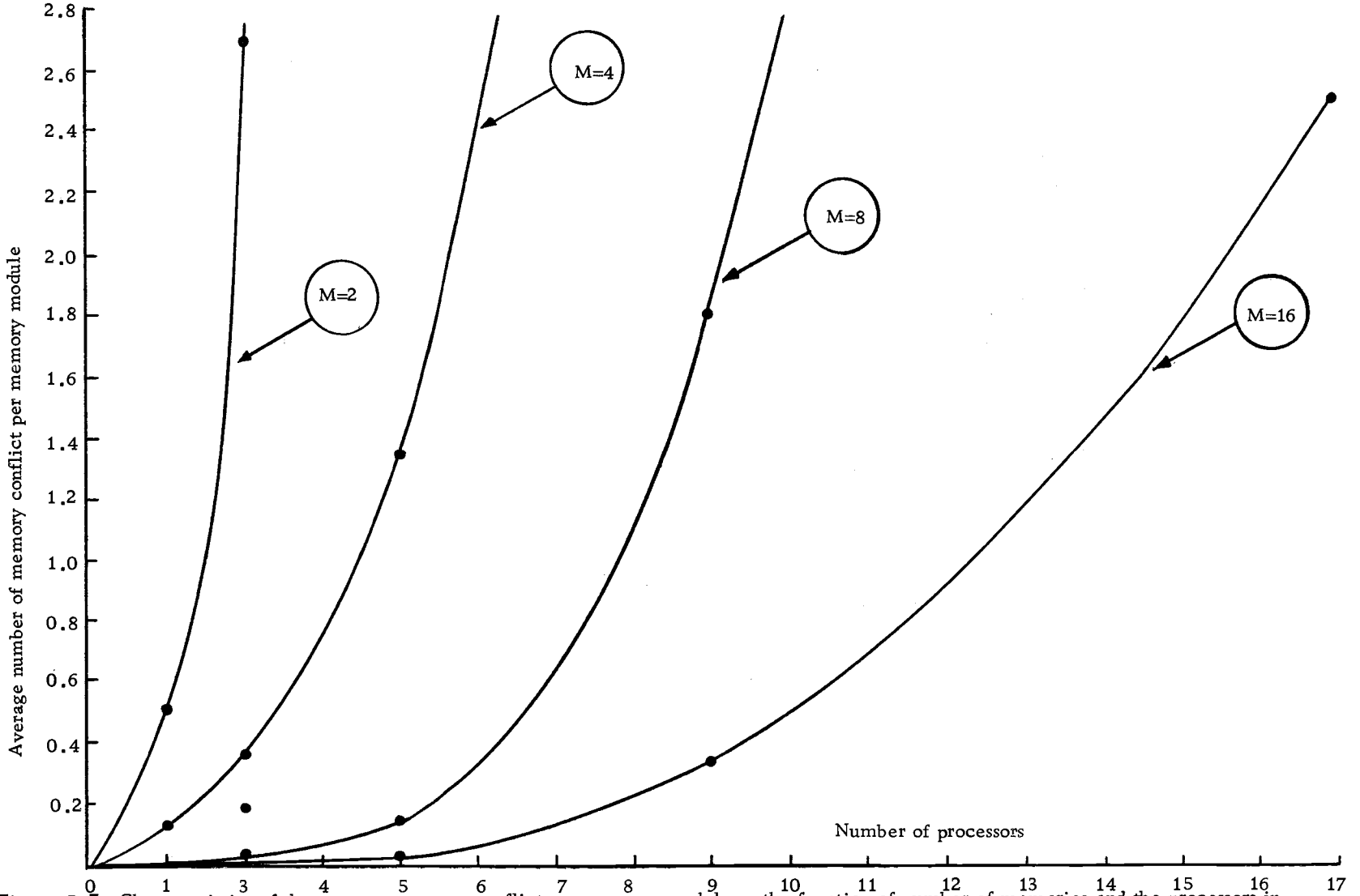


Figure 5.7. Characteristics of the average memory conflict per memory module as the function of number of memories and the processors in the computer system.

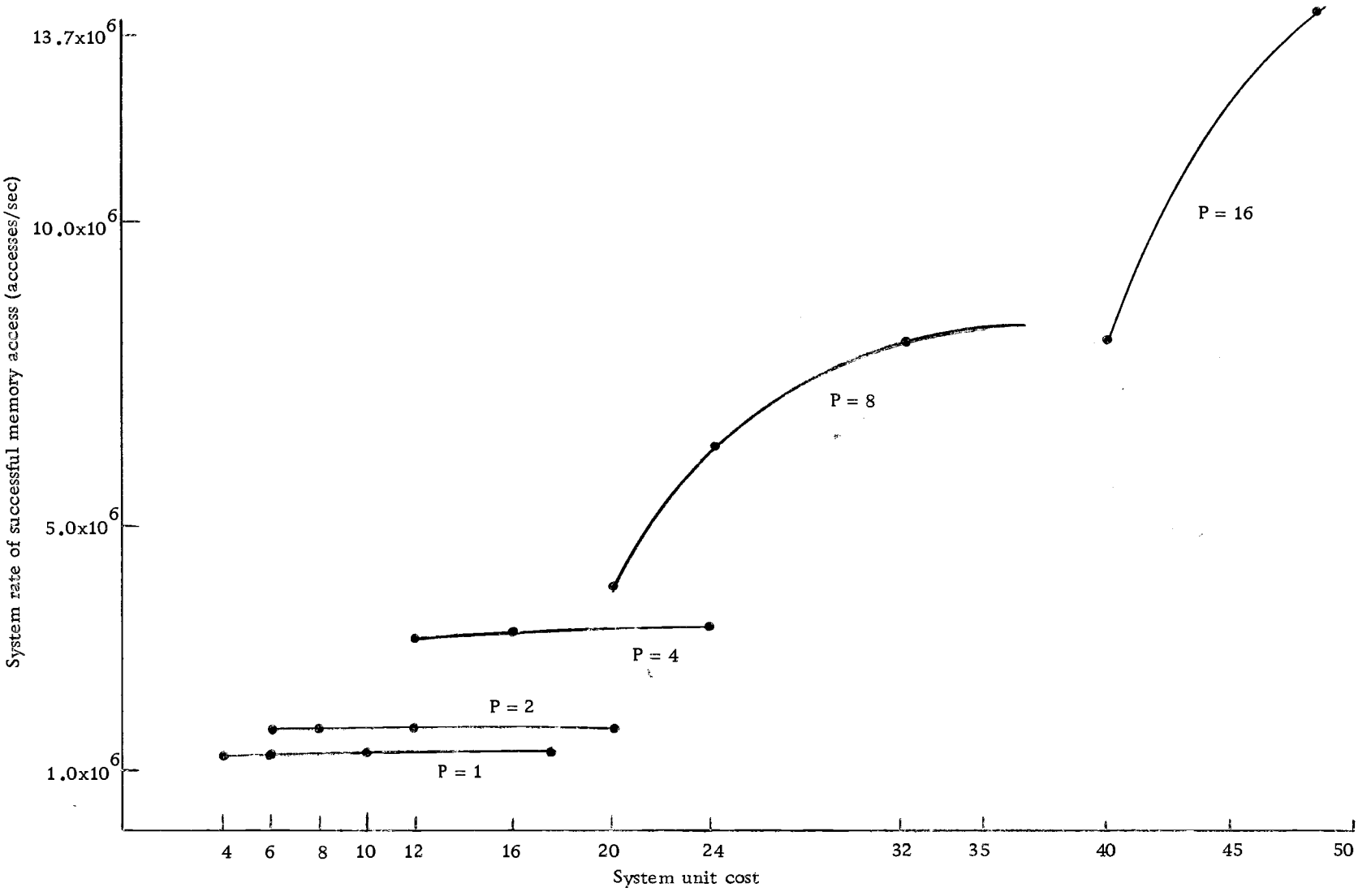


Figure 5.8. The interaction between system rate of successful memory access and system processor-memory per unit cost.

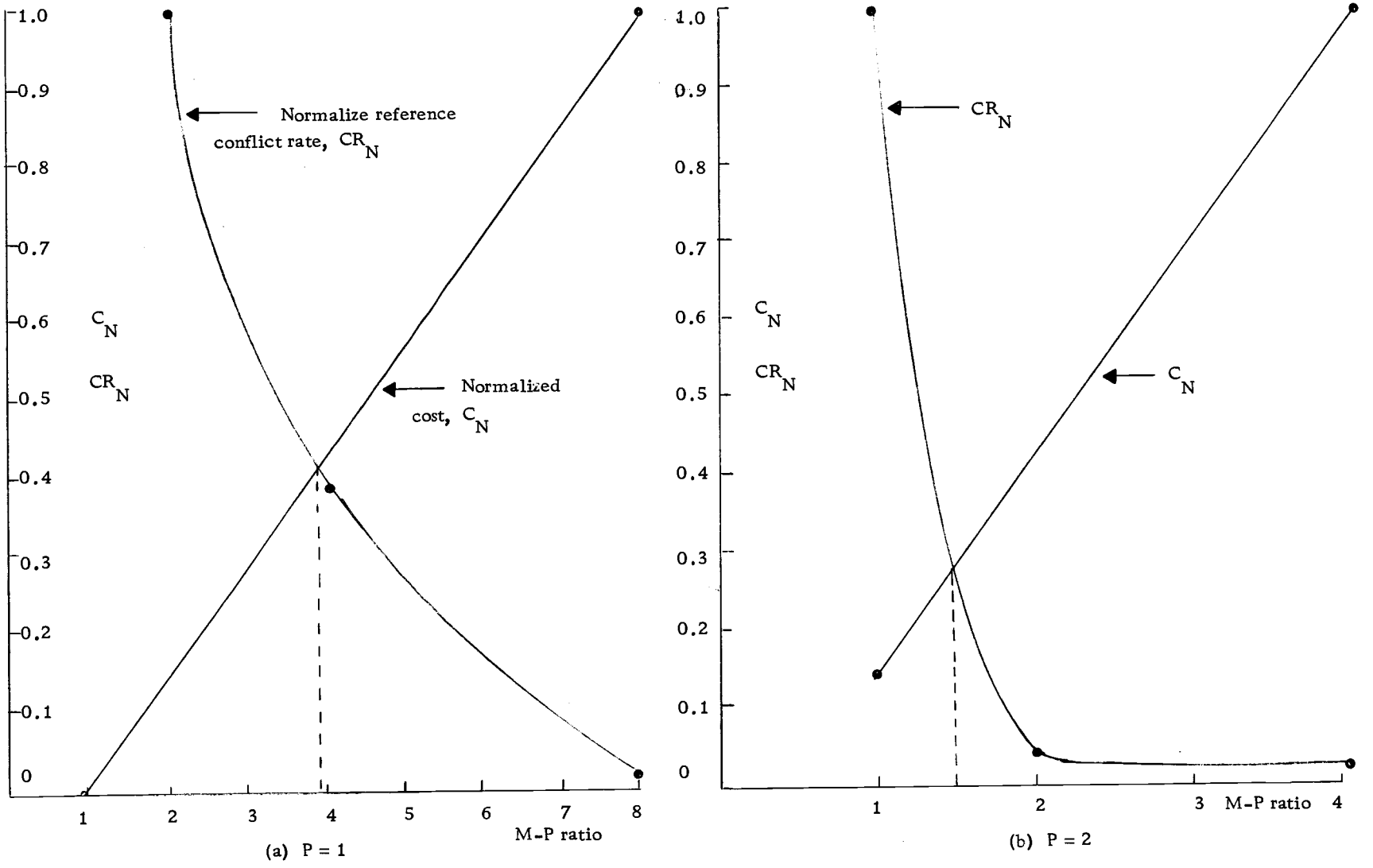
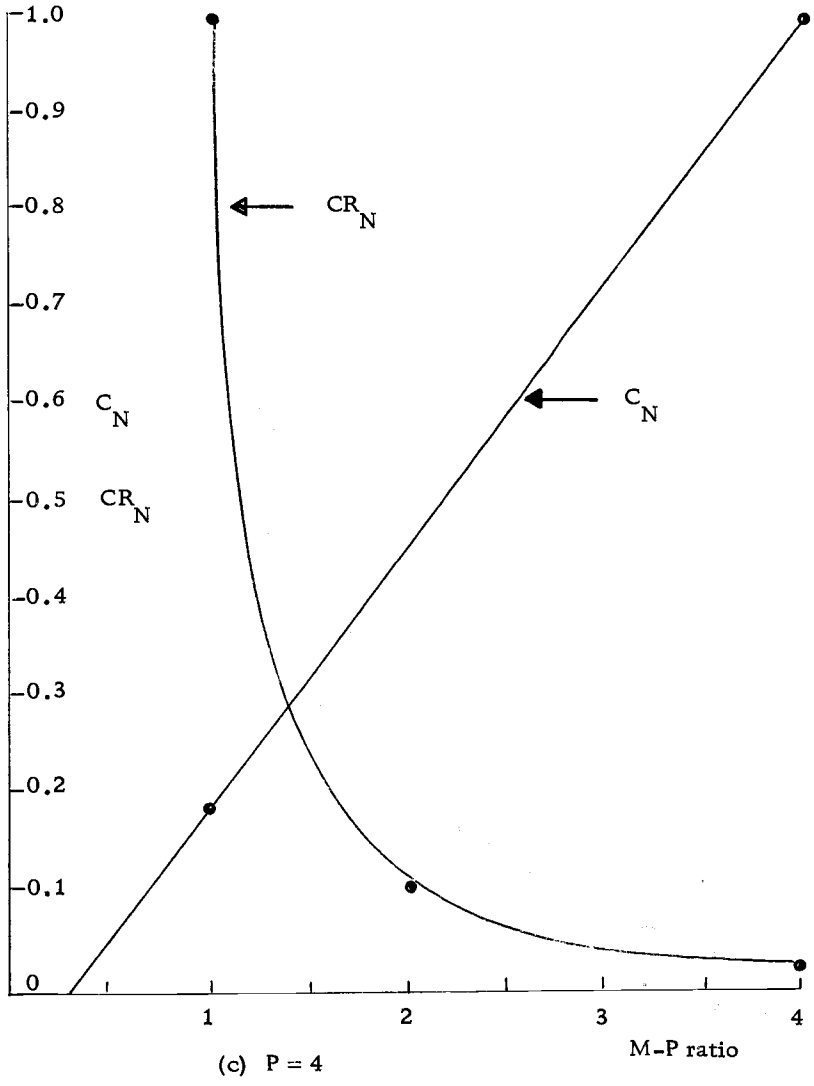
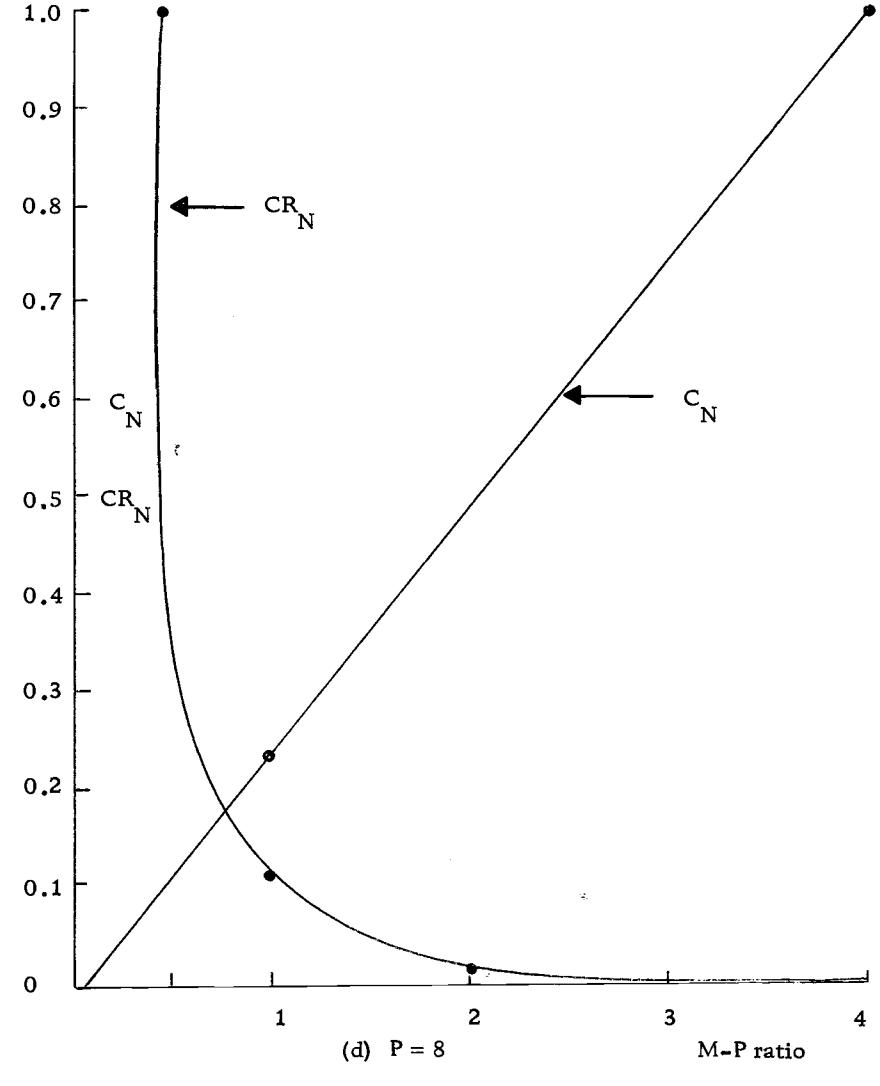


Figure 5.9. The normalized reference conflict rate (CR_N) and the normalized cost function (C_N (I)) plotted as a functional of M-P ratio.



(c) P = 4



(d) P = 8

Figure 5.9. (Continued).

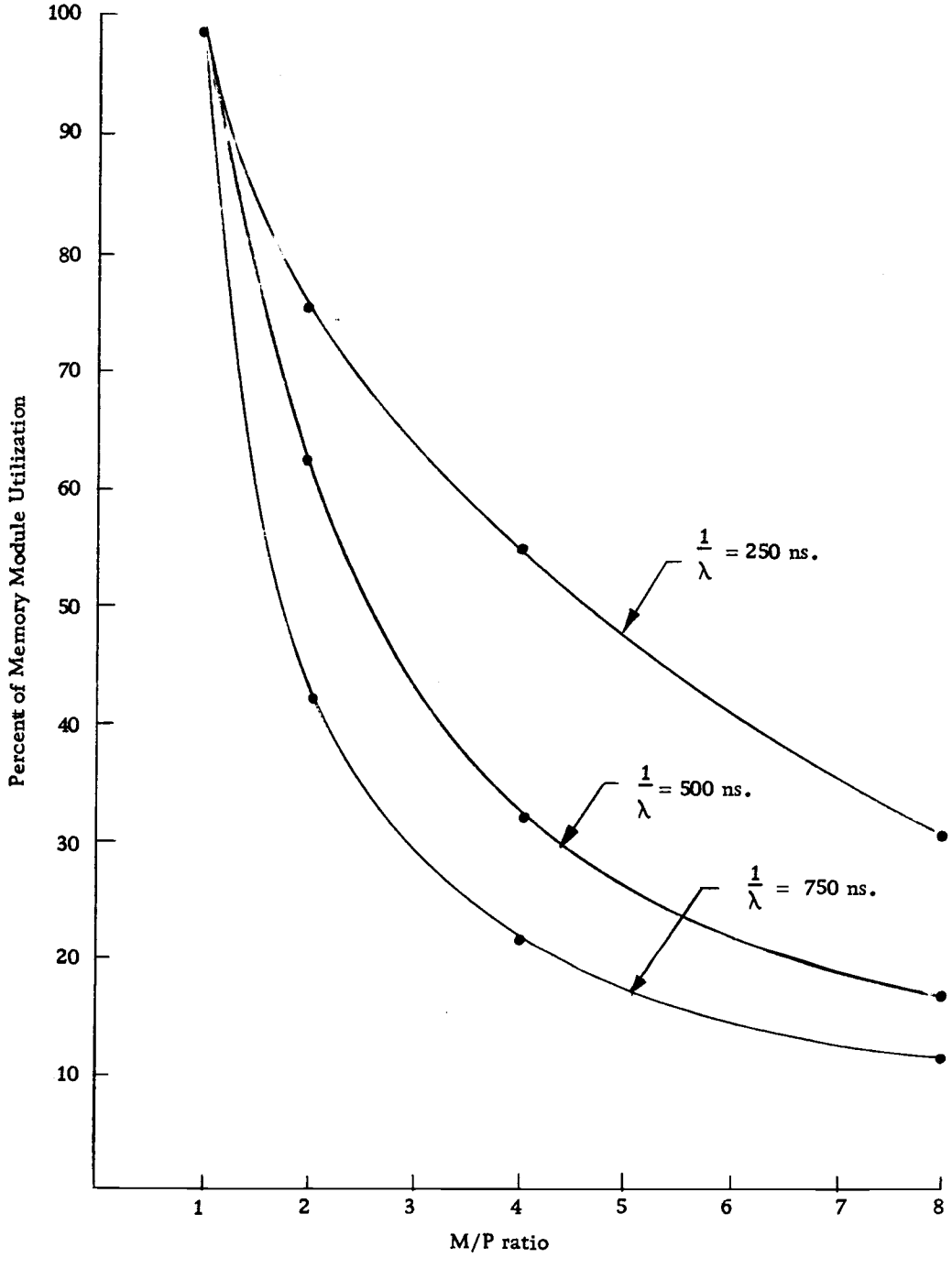


Figure 5.10. Memory module utilization (%) vs. M/P ratio as the rate of memory reference is varied, with P = 2 in the system.

For fixed P,

$$\text{System cost, cost I} = K_H + PK_p + MK_M$$

$$= K_H + P(K_p + IK_M) \qquad (5\text{-}1)$$

$$\text{Base cost, } C_B = K_H + K_M PK_p \qquad (5\text{-}2)$$

$$\text{Relative cost, } C(I) = \frac{\text{Cost}(I) - C_B}{C_B} \qquad (5\text{-}3)$$

$$\text{Normalized cost } C_N(I) = \frac{C(I)}{C(I_{max})} \qquad (5\text{-}4)$$

Here, it is assumed that $K_p = 2K_M$; $I_{max} = 4$. Hence from Equations 5.1 - 5.4:

$$C_N(I) = \frac{C(I)}{C(4)} = \frac{\dfrac{\text{Cost}(I) - C_B}{C_B}}{\dfrac{\text{Cost}(4) - C_B}{C_B}}$$

$$C_N(I) = \frac{\text{Cost}(I) - C_B}{\text{Cost}(4) - C_B} \qquad (5\text{-}5)$$

$$= \frac{K_H + PK_p + MK_M - (K_H + K_M + PK_p)}{K_H + PK_p + 4PK_M - (K_H + K_M + PK_p)}$$

$$= \frac{MK_M - K_M}{(4P - 1)K_M}$$

$$C_N(I) = \frac{PI - 1}{4P - 1}$$

$$= \left(\frac{P}{4P - 1}\right) I - \left(\frac{1}{4P - 1}\right) \qquad (5\text{-}6)$$

This normalized relative cost function $C_N(I)$ is a measure of the cost penalty involved in increasing the M-P ratio. For P = 1, a unit increase in M-P ratio will increase the relative system cost approximately a factor of .3.

Figure 5.6 and 5.7 show the expected result that as the M-P ratio increases the occurence of memory conflicts decreases. Figure 5.8 shows that in increased expenditure is unjustified for $P \leq 4$. For $P > 4$, however, a dramatic increase in successful accesses is obtained with a small increase in cost. This is due to the fact that adding memory modules does not effect as much the cost of the more complex systems.

Figure 5.9 essentially indicates the cost/performance trade off. These four graphs indicate a M-P ratio of 2-3 to be a good choice.

The last graph in Figure 5.10 shows the effect of memory reference rate by a PU on the overall performance.

This study indicates that the average queue length generated by a single processor with a request interval of 75ns on memory modules to be in the range 2-3. In the processor design, a Memory Address Hopper Unit has been included to hold these requests and reduce the conflicts. The design of this unit is discussed in the next section.

To handle inter-processor conflicts, a round-robin contention scheme has been implemented in hardware in the crossbar switch.

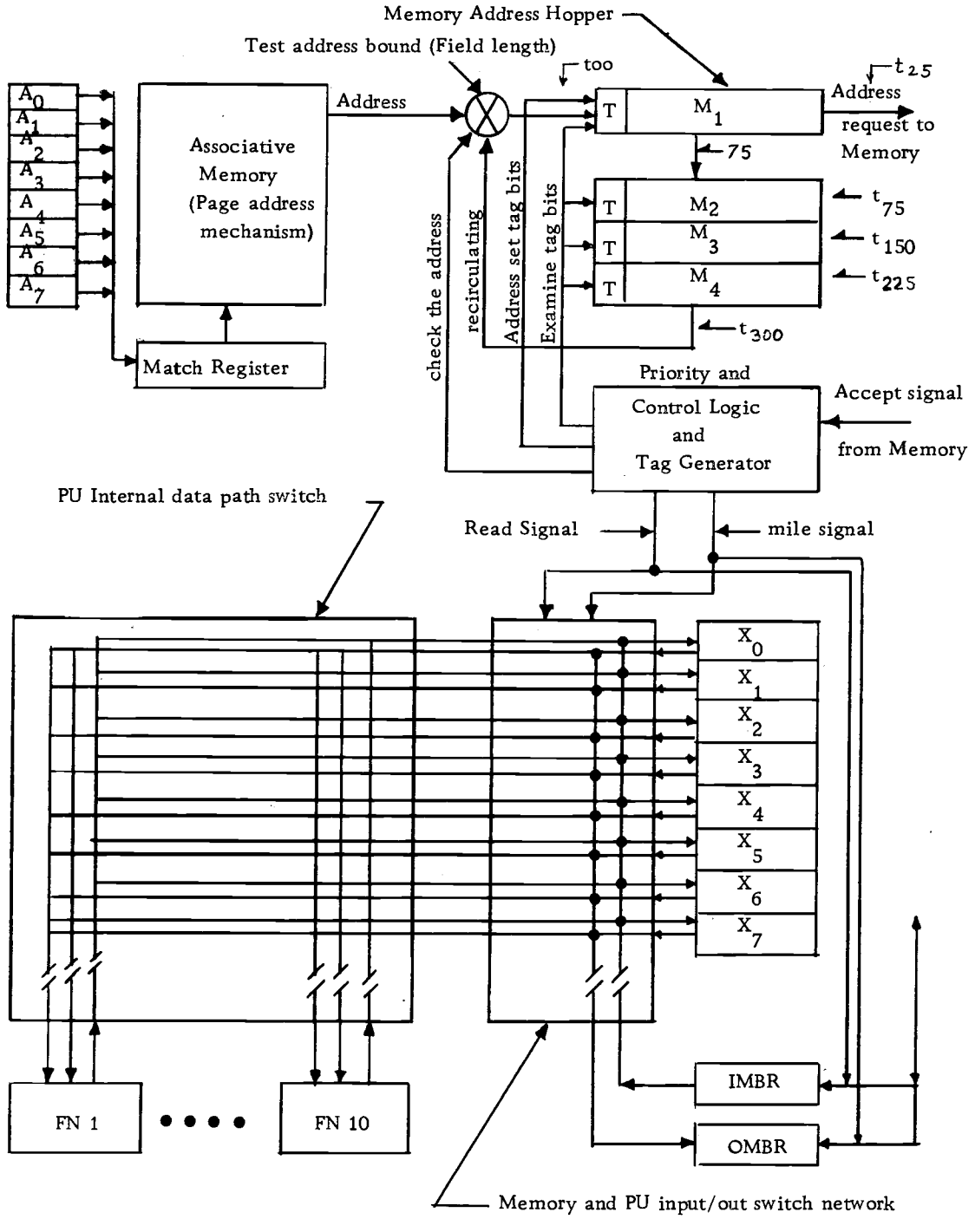
5.3 Memory Address Hopper

The Memory Address Hopper is the PU internal storage reference control unit. It consists of three major parts: Hopper register, Priority network, and Tag generator and distributor. The unit is illustrated in Figure 5.11.

The hopper is a collection of registers used to retain storage reference information until any storage conflicts are resolved. In principle this unit allows a new storage address to be delivered every minor cycle with any "rejected" addresses to be reissued repeatedly until accepted.

There are two types of storage reference: instruction fetch and read and store operands. These two occur in both the PU and the PPU. In the case of simultaneous references, priority is given to operand access. Addresses circulating in the hopper are given priority over new addresses. The priority and control logic handles such conflicts as well as testing for address out of bounds.

The tag generator attaches a status/identifier tag to each address in the hopper. These tags are circulated with the addresses in the hopper and read by the priority control logic as needed. The tags indicate whether the address is a read operand, write operand, fetch instruction and whether the address is being recirculated.



Note:

T = Tag bit

Instruction/Data	1 bit
Read/write	1 bit
Source/Destination	2 bit

Figure 5.11. Illustration of designed Memory Address Hopper. Logical block diagram.

To understand how the Memory Address Hopper works, consider the following example. Assume the three hopper registers are empty. A storage reference address is entered in register M_1 from the PU. This entry is made under the control of priority network. Suppose the time of entry at t_{oo}. A tag from the tag generator is attached which fully identifies the nature of the individual memory reference. The memory address is sent to the memory module via the crossbar switch on the address bus, at t_{25}. The four hopper registers M_1, M_2, M_3 and M_4 are designed to allow the address information to enter in register M_1 and circulate through the other registers and return to M_1 again. Each 75 nsec. time interval, data are switched between registers, which results in a recirculation time of 300 nsec. The time sequence is as follows:

t_{oo}	Memory address and 6 bits tag enter M_1
t_{25}	Address to Memory address bus
t_{75}	M_1 to M_2
t_{150}	M_2 to M_3
t_{225}	M_3 to M_4
t_{300}	M_4 to M_1 (if not accepted)

If the memory module addressed is not busy, an ACCEPT signal is returned on the ACCEPT bus to the Memory Address Hopper. This indicates that no conflict exists, the memory transfer cycle is initiated in that module. The ACCEPT signal reaches the Memory Address

Hopper in time for the control logic to disconnect the path from M_4 to M_1. If the selected memory module is busy, no ACCEPT signal will be sent. In this case, the priority network will give top priority to the recirculation path, causing new addresses to be held up for one minor cycle.

It is clear that the hopper can hold three recirculating addresses. The choice of three was based on the memory conflict simulation which showed the average queue behind each memory module was in the range 2.5-3. The Memory Address Hopper feature should improve the memory accessing capability of the system.

5.4 Crossbar Switch

The crossbar switch plays a very important role in the multiprocessor system, since it provides the data connections and interfaces between processors and memory modules. In general, there are two basic types of switch networks: the Time-Division (Multiplexing) switch and the Space-Division switch. The time division switch network requires less electrical hardware and it is simpler to control, but the implemented system is quite rigid and less efficient because only one processor-memory pair can be connected at one time. There are two conventional schemes for space-division switching, multiport and matrix-crossbar. Both schemes are illustrated in Figure 5.12 Both schemes seem to offer the same

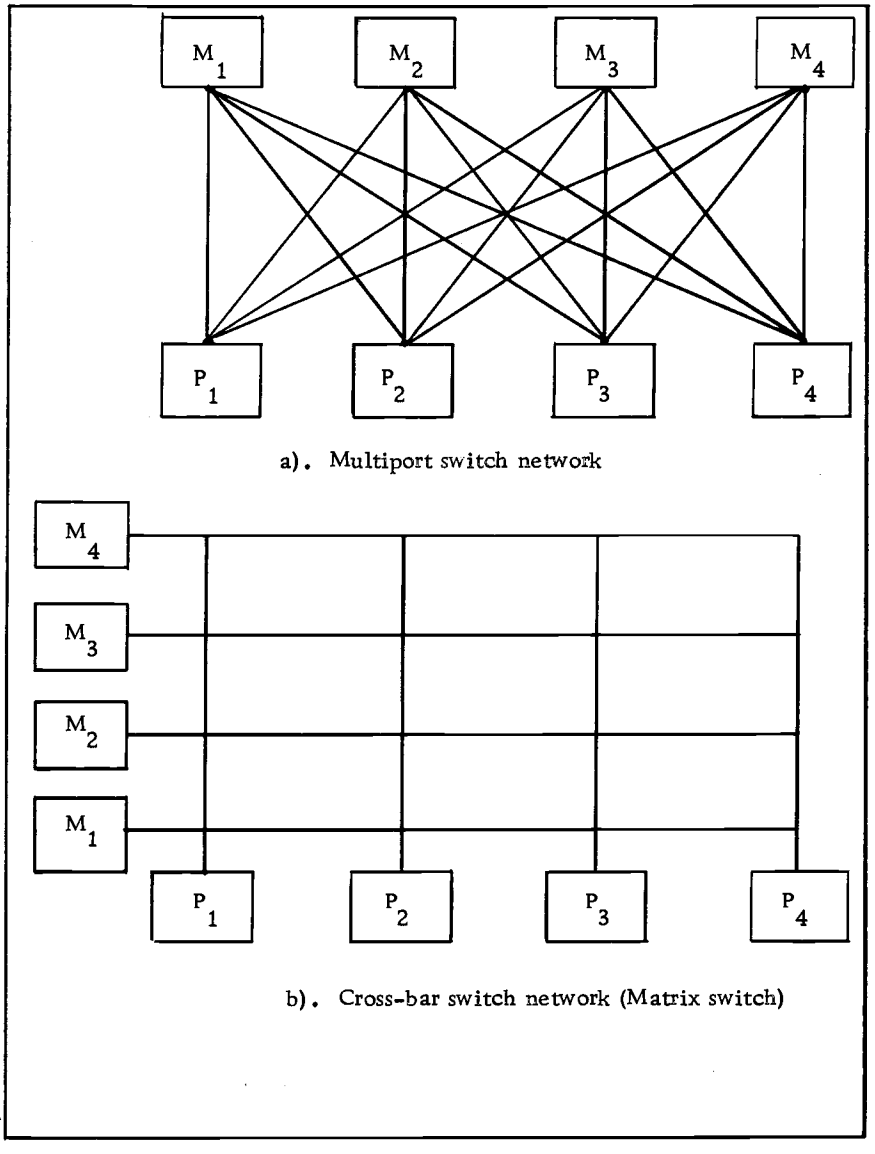
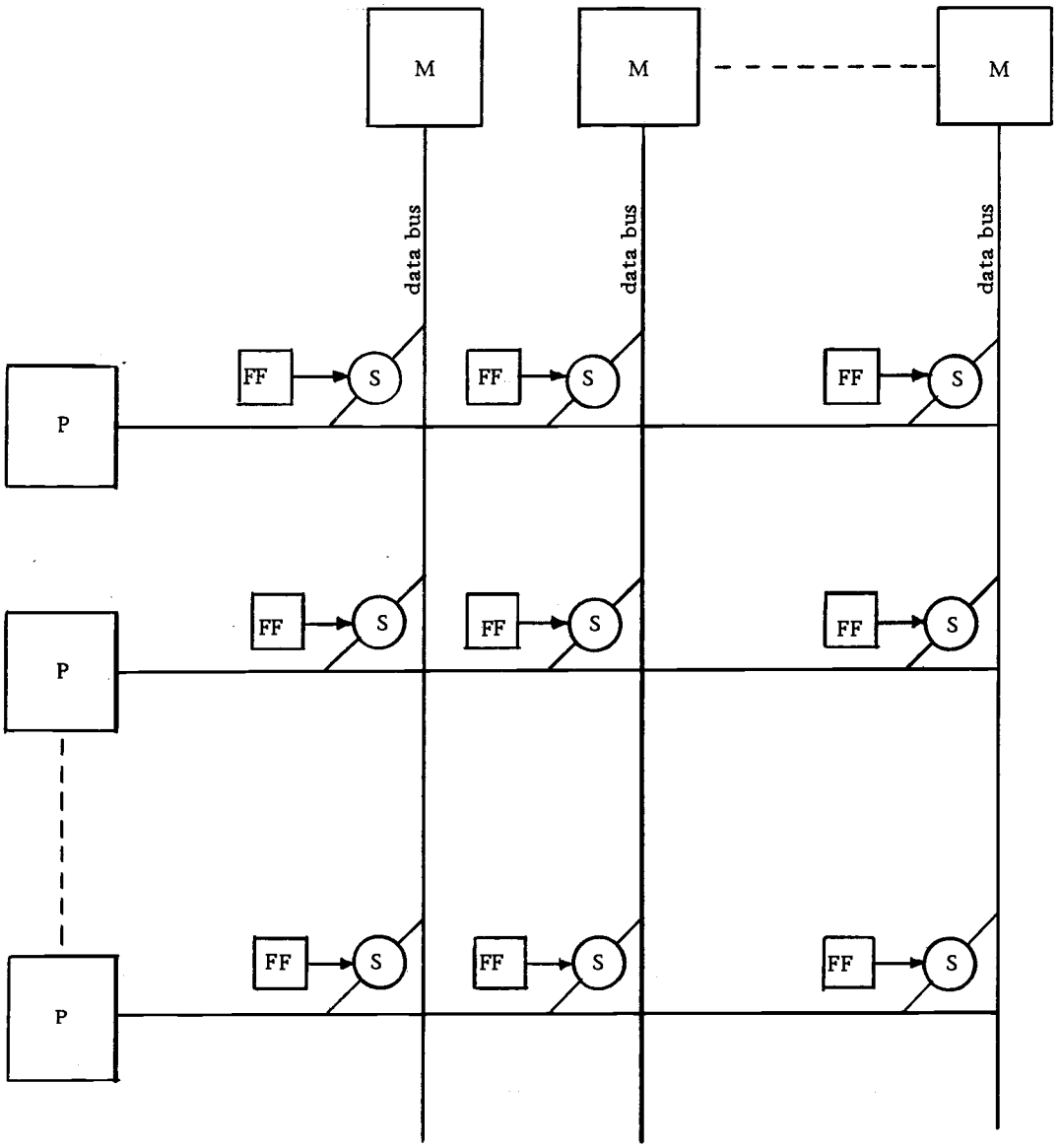


Figure 5.12. Physical multiprocessor interconnection schemes.

speed; however, the multiport scheme requires more interconnection cables (N_c = P x M) than the crossbar switch does (N_c = P + M), causing the multiport scheme to be very expensive for the speed gained. The crossbar switch provides more flexibility in expansion of both the PU module and the Memory module. The Matrix-crossbar switch seems to be more cost effective than the multiport scheme. Hence, in the designed computing system the matrix crossbar switch is selected.

An essential characteristic of the matrix crossbar switch is the capability for allowing simultaneous connections of several processing units and memory module pairs except in conflicting situations. The typical crossbar switch provides tremendous flexibility and efficiency. Disadvantages include high electrical hardware cost and more complex control logic. However, since 1970, due to the advance of LSI technology the hardware cost of a computer no longer has a large influence on the total cost of the system.

Simple control logic is introduced to satisfy the switching matrix functions. The logic of this switching matrix can be considered as having two major parts, one for data/path and the other for control manipulating functions. The data path control is relatively simple as shown in Figure 5. 13. At each intersection of the matrix, a switch S is introduced controlled by a single Flip-flop, FF. Each flip-flop,



Note: FF = Flip-flop
S = Data path switch

Figure 5.13. Crossbar switch data paths.

being set or reset, closes or opens the corresponding switches which connect or disconnect the transmission path between a processor-memory pair.

One can invent many methods to resolve the conflict resulting from the processors competing for a single memory. One possible method is "Round-robin contention". In a contention network the terminal (processing unit) makes a request to transmit the information (address). If the memory module in question is free, transmission goes ahead; if it is not free, the processing unit will have to wait until it becomes free. A queue of contention requests may be built up by a processing unit, (i.e., hopper-registers) or by several contending processors.

The scheduling of the queue can be either in a prearranged sequence or in the sequence in which the requests are made. In this proposed system "Round-robin" priority scheduling is introduced. For example, PU1, PU2, PU3, and PU4, the processing units, could try to make access to the same memory module M_1 in the system through the crossbar switch simultaneously. The basic principle of Round-robin Contention can be described as follows:

1. After a memory request is made to the memory module M_i by processing unit PU_j, the "priority indicator" of memory M_i is passed on to the next processor in a predetermined sequence. Now PU_{j+i} has the highest priority

for access to memory M_i.

2. In case no request is issued to memory M_i by higher priority processors; then lower priority processors, PU_{j+2}, PU_{j+3}, PU_{j+4}, etc. will be able to access memory M_i.
3. The priority logical network will then shift the priority indicator to its successive neighbor.

Figure 5.14 illustrates in detail one possible logical design of the implementation of Round-robin Contention. Only four processing units, PU1, PU2, PU3, PU4 and one memory module M_1 are considered. Flip-flop, RFF_j represents a Request Flip-flop. It is set to "1" when its corresponding PU_j makes a request for memory access; otherwise it is set to "0".

Flip-flop PFF_{ji} is the high priority indicator flip-flop. For example, if Flip-flop PFF_{21} is set to "1", with respect to memory module M_1, processor PU_2 has the highest, PU_3, next, PU_4 third, and PU_1 last. Each column in Figure 5.14 corresponds to one memory module; in a given column, only one PFF_{ji}, priority flip-flop, can be set at a time. The crossbar switch network is synchronized with a standard clock. At every clock pulse, the requests from each processing unit are examined, priority of request is considered, conflicts resolved, and connections are made between a requested memory module and the appropriate processor (PU). In general, the

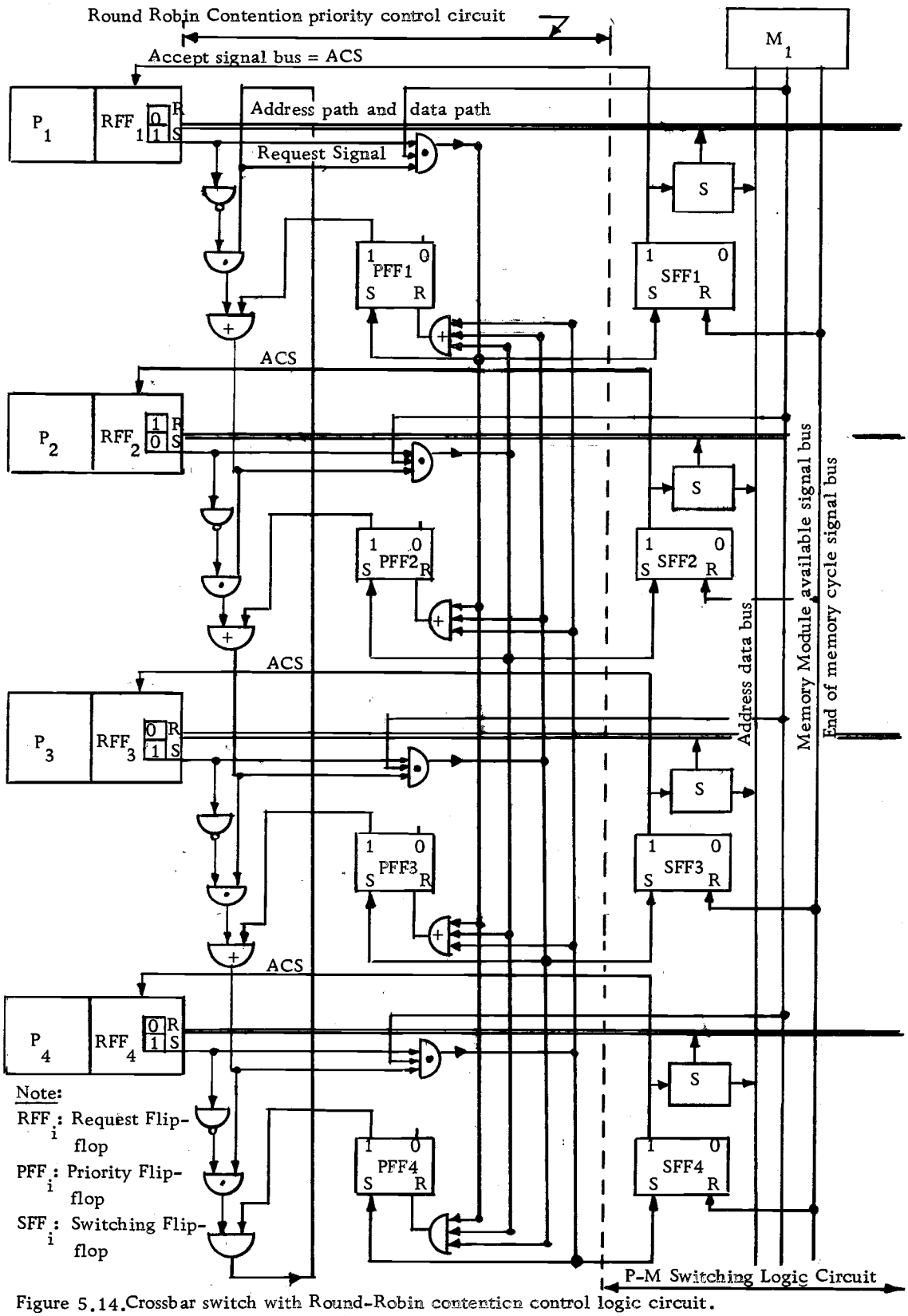


Figure 5.14. Crossbar switch with Round-Robin contention control logic circuit.

connections between a memory module and processing unit pair can occur at any given time.

The above Round-robin Contention Logical control circuit works in the following manner.

1. It may be assumed that at a considered time, PFF_{21}, is set, memory module, M_1 is available, and only PU_3 and PU_4 try to access memory M_1 simultaneously.
2. In this example, since PU_2 is not requesting memory M_1, then the highest priority indicator is passed from PU_2 to PU_3.
3. At the end of the clock period, the priority propagating signal and the PU_3 requesting signal set both PFF_{41} and SFF_{31}, and at the same time reset PFF_{21}.
4. Setting SFF_{31} accomplishes interconnection between PU_3 and M_1 while setting PFF_{41} causes the high priority indicator to be passed on to PU_4.
5. Setting SFF_{31} also starts the memory operation and puts M_1 in a busy state. Furthermore, M_1 sends an accepted signal back to the requesting processor, PU_3. This means that the memory request is accepted.
6. At the end of the memory cycle of M_1, an <u>end-of-cycle signal</u> is issued by the memory M_1, which clears the

switch flip-flop, SFF_{31} to zero and makes the memory M_1 available for the next coming request.

7. Since the highest priority is now posted at PU_4, this time PU_4 is guaranteed to have access to M_1. As memory requests are examined and priority is updated on a clock pulse basis, no single processing unit can be tied up with a memory module for more than one memory cycle at a time.

VI. PERIPHERAL SYSTEM SPECIFICATION

A number of peripheral processing units are provided in the system. This permits processing units to have more available computing time and allows independent concurrent operations between user program processing and I/O processing.

In general, the I/O devices can be classified into two categories: those capable of providing bulk data transference and those which deliver or take one reading at a time. For example, conventional card readers, line-printers, and magnetic tape belong to the first category, while real-time sensors, such as temperature gauge, pressure gauge, radar signal etc. belong to the second category. Devices of the latter category can be simply handled by directly connecting them to a multiplexor. In fact, these devices, are under the direct control of the application programs for real-time operations. The I/O operation is relatively simple. A processing unit (PU) merely issues I/O device number to the corresponding I/O multiplexor which will find the correct I/O device to perform the proper data transfer. This data transfer is conducted here between devices and PU while the PU is waiting. This type of I/O operation is simple and straightforward and will not be mentioned in greater detail here. However, the operations for the bulk data transfer device are relatively complex and need more attention.

Basically, a Peripheral Processing Unit (PPU) performs the following I/O functions:

1. Executes I/O instructions initiated by Processing Unit (PU) and controls the initiation of peripheral device actions.
2. Transfers data between I/O devices and main memory modules.
3. Establishes priorities between devices.
4. Buffers data between asynchronous devices.
5. Interrupts the central processor for execution of priority tasks.

One PPU has the capability of handling a number of I/O devices simultaneously over a set of connected cables. The organizational block diagram of the proposed peripheral processing unit (PPU) is shown in Figure 6. 1.

A small micro program control memory unit is provided in each PPU to execute I/O instructions and to control interface signal exchanges with PU's and main memory modules. Two small scratch pad memories are provided. One is used to store channel status information such as starting memory location, word count, memory protection key, device number, and device status. The other is used to store the I/O data as it is referred to the data-buffer.

This design closely resembles the structure of the conventional minicomputer. In our command and control system, one peripheral

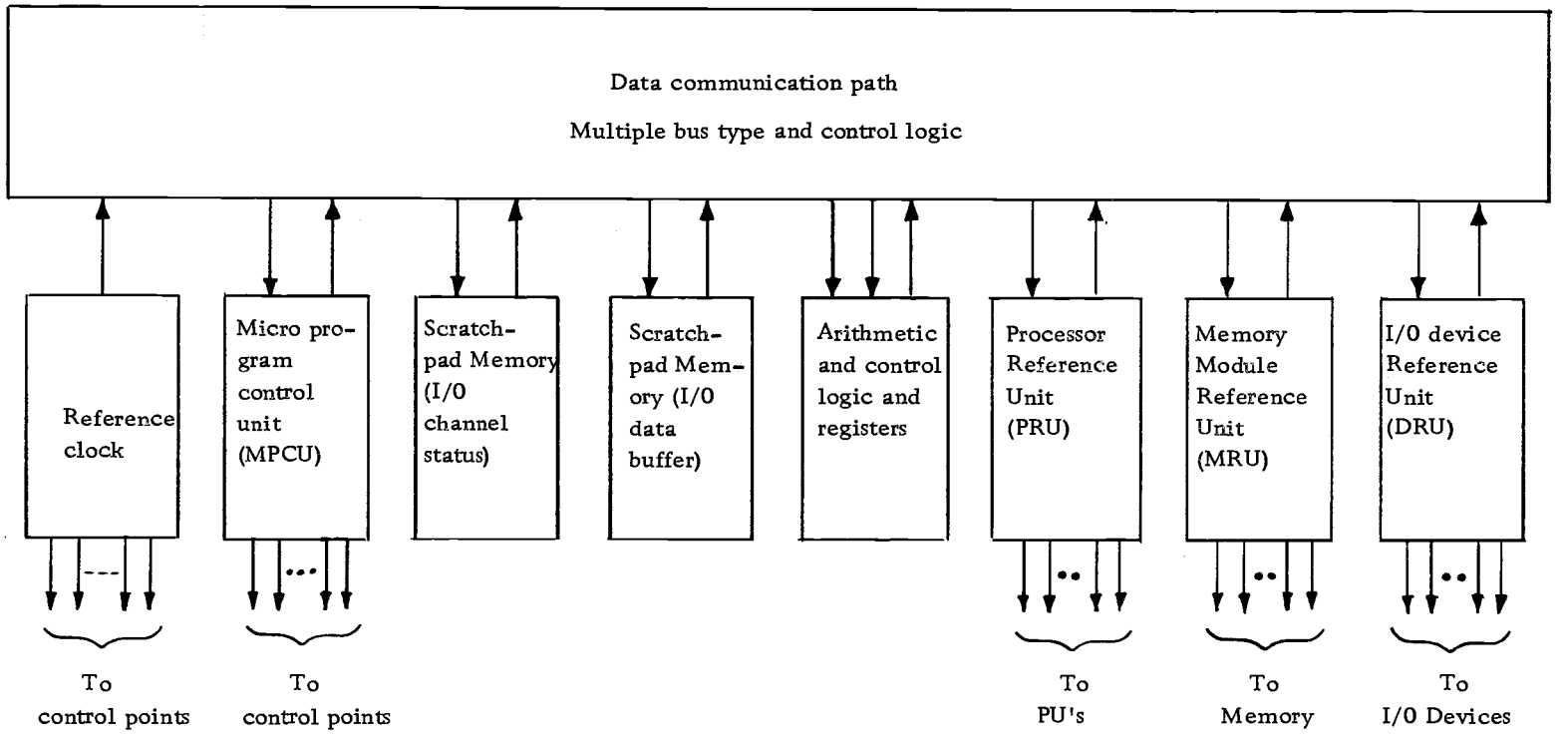


Figure 6.1. Peripheral Processing Unit Organization (PPU).

processing unit will control up to 16 device controllers. The 16 I/O device controllers are connected to a single bus as shown in Figure 6.1, or through an I/O matrix switch as an alternative option.

In each peripheral processor, a simple arithmetic capability is required. This includes counting logic, a shift network, an adder, an incrementer, a comparator and necessary operating registers.

In each peripheral processor, an interfacing unit called the Processor Reference Unit (PRU) is provided to accomplish communication between PPU and PU's. The PRU checks to see whether any processing unit (PU) is waiting to communicate with the associated PPU. If so, communication is initiated. If not, the control is then passed over to another communication port.

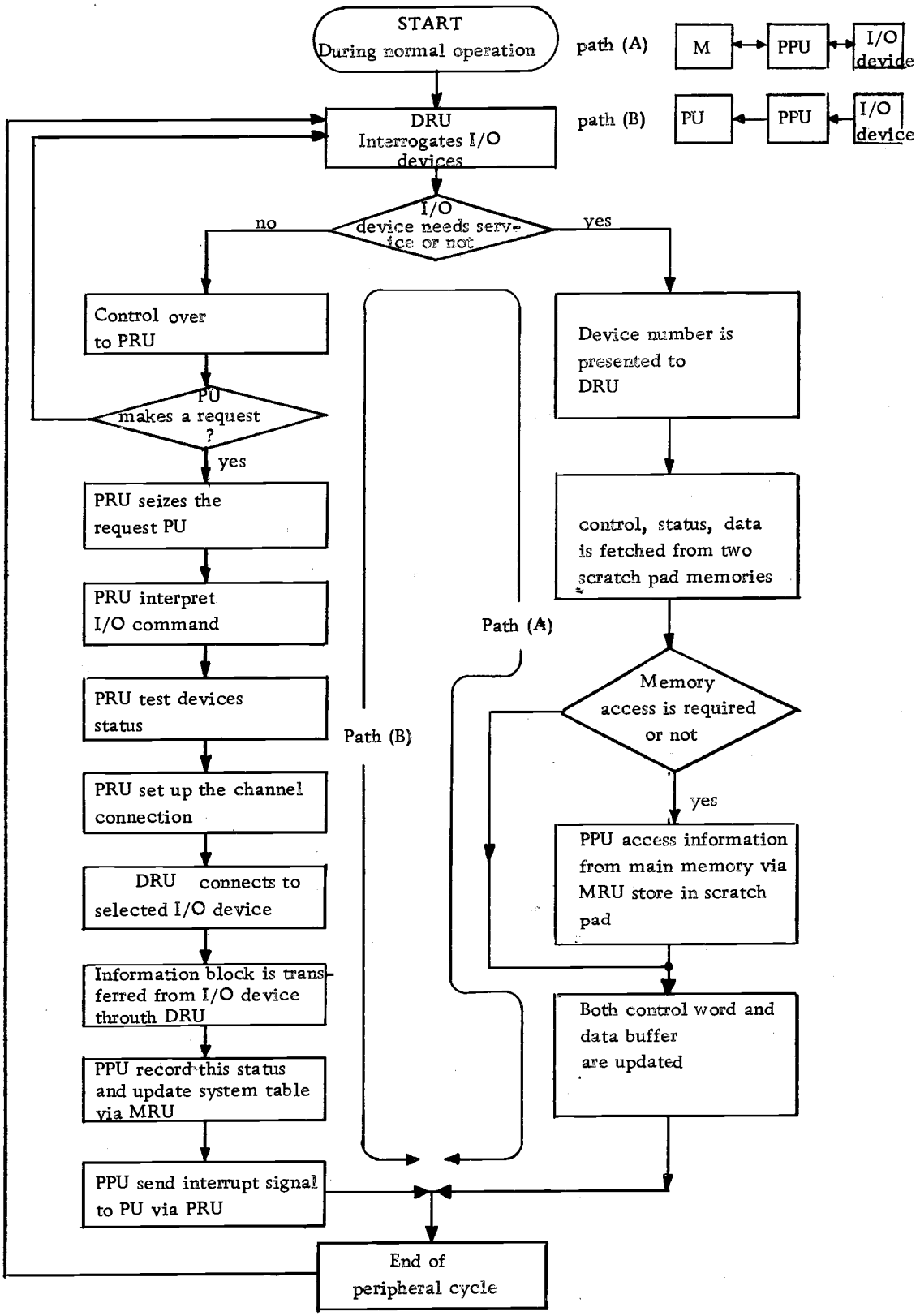
The memory module reference unit (MRU) provides the communication between PPU and Memory Module to perform the input and output data operation.

The I/O Device Reference Unit (DRU) is a communication port which interrogates I/O devices to see whether one needs service. If so, that device is picked up for service.

During a normal operation, the DRU constantly interrogates I/O devices to see whether any one is processing or not. If not, control is then passed over to the processor reference unit (PRU) to see whether any processor is waiting to communicate with PPU. If not, the control is returned to DRU, and so on.

When a device requests service, the device number will be presented to DRU, which then turns to the two scratch pad memories and fetches out of the data buffer the control and status words. In case no memory reference is required, both the data buffer and the control words will be updated. If memory reference is required, the buffer is either filled in case of input or emptied in case of output. The memory address with protection key is delivered to MRU for memory access and both the control word and data buffer will be up-dated. This sequence of operation is called Peripheral Cycle. When it is completed DRU returns to its interrogation routine again. Figure 6.2 shows the flow chart of the Peripheral Cycle.

At some point in the operating cycle when a device requests service and the PRU finds that one PU is making a request, the PRU seizes the next peripheral cycle to communicate with the processing unit (PU). The major functions of the PRU are interpretation of I/O commands, testing I/O device status, setting up channel control words, and initiating the I/O device operation through DRU. After the I/O device has accomplished the block transfer, as indicated either by zero word count in the control word or by the device's physical status, (for example, no card in the hopper), the PPU records this status in the system table via MRU and activates an interrupt signal to processing unit(PU's) through PRU port. From the point of view of hardware control implementation, logic gating will be done in local



(b)

Figure 6.2. Peripheral cycle: (a) block diagram; (b) flow chart.

units, but major sequence control will be governed by the micro-program control memory.

6.1 Configuration Assignment Unit (CAU)

The original purpose for providing reconfiguration capability in a computing system was to establish a high level of fault-tolerance. Such a system is neither a multiprocessor nor even a multicomputer system. However, each of these computing systems has a configuration assignment unit (CAU) capable of interconnecting the various functional units to maintain an operational system. The CAU can also connect multiple memory modules to the same processor. However, since no memory is shared, the system is certainly not a multiprocessor. It is merely a method of maintaining two or three uniprocessors with high availability and reliability.

The CAU in Fault-tolerant design, was basically a crossbar switch matrix, and, as such, had all the problems of complexity found in other crossbar systems. Every multiprocessor system possesses a large degree of fault tolerance due to its physical structure, as compared with a uniprocessor. It is clear that for most multiprocessor systems, the configuration assignment unit was considered to be a dormant crossbar switching network. For example, the first practical multiprocessor system, the Burroughs D 825, and later the Carnegie-Mellon University Cmmp, employed this concept.

In an array processor such as ILLIAC IV, the configuration assignment unit is clearly separated into two parts: (1) logical control in the Array Control Unit, and (2) switching mechanism in the array data routing network. The nature of the circuit hardware seems to be relatively more complex than that implemented in typical multiprocessor system.

In the proposed command and control computing system design the configuration assignment unit (CAU) will provide the following features:

1. continuation of the system multiprocessor function.
2. switching out failed modules of PU, M, and PPU from the system for repair or maintainance.
3. switching in a replacement module from those available in the system or spares
4. re-configuration of system structure, such as partitioning into sub groups, or performing array processor operations like that of ILLIAC IV.

To simplify hardware implementation and to provide more convenient maintainance and repairing, a modular and pluggable type of hardware is considered. In this design, the proposed CAU splits into two major parts. The logical control unit itself is a separate module. In the actual machine this function is provided by any one of the PPUs. The switching mechanism section consists of an

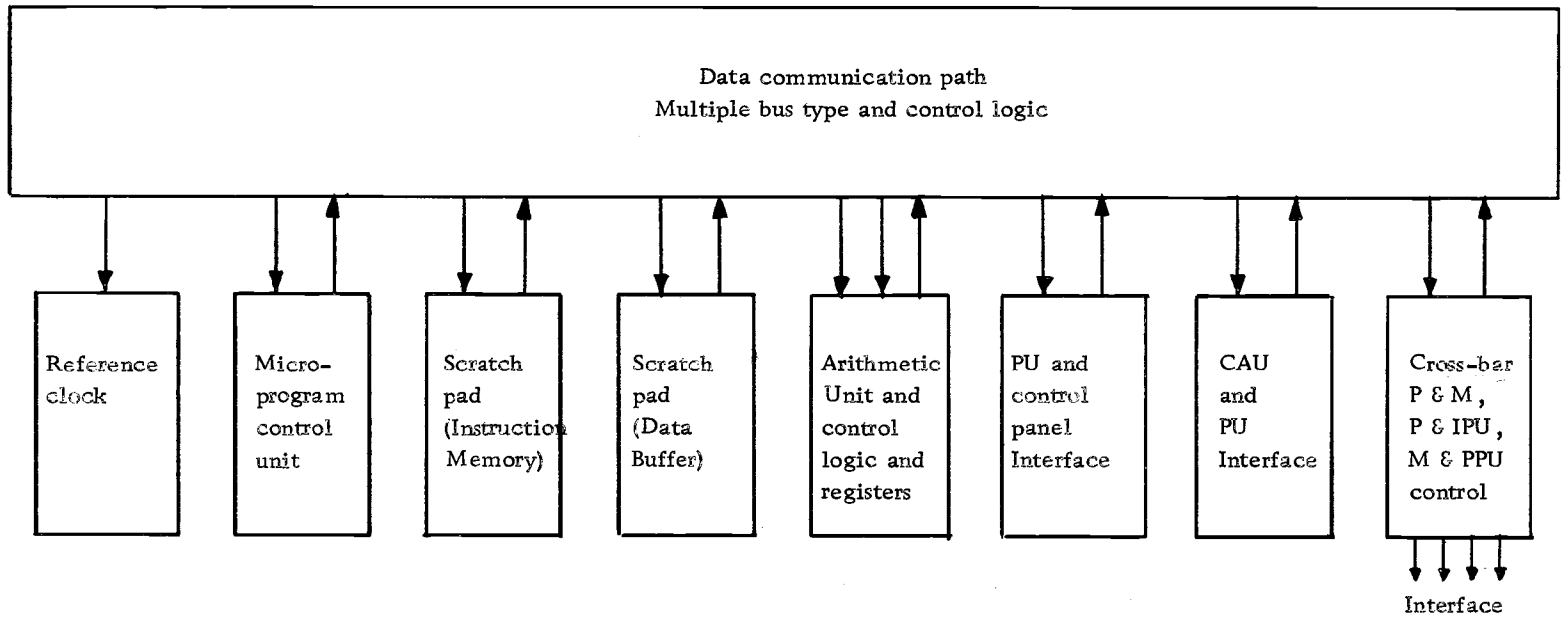


Figure 6.3. Configuration Assignment Unit (CAU) Organization (any one of PPU can be assigned to be CAU).

interfacing unit, a reconfiguration command interpretation network, decoding network, a matrix switch select circuit and data-path switches in the crossbar switch unit.

In addition, all crossbar switches, (PU and M, M and PPU, PU and PPU) are designed to have the same structure and are implemented by L.S I. chips in order to reduce the cost of construction.

The basic goals of configuration control are:

1. that configuration control can be manipulated by the user through the station control key board or by a PPU instruction,
2. that when a PPU is assigned to serve as configuration control unit (CAU), one of its internal 18 bit registers is used as Configuration Control Register (CFR). These 18 bits are subdivided into six fields as shown in Figure 6.4. The fields are defined in Table 6.1.

After the configuration control register (CFR) is set up, the appropriate device is notified. The reconfiguration code is sent to the appropriate crossbar switch. The configuration control code is interpreted and decoded into control signals which select and drive the desired cross-point on or off, depending on the control code.

By the use of a single control code or a combination of the above mentioned control codes, the desired configuration can be achieved. After the reconfiguration is accomplished a configuration

interrupt is triggered and control is then transferred into Master Control mode.

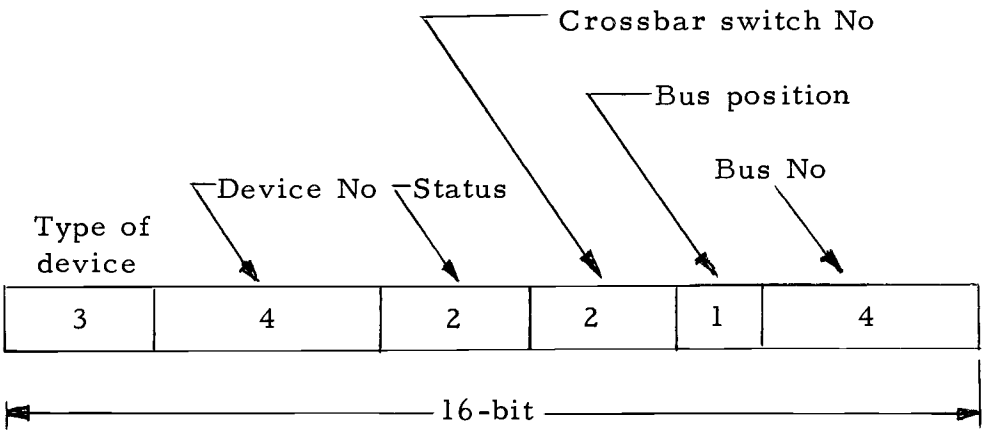


Figure 6.4. Configuration control code format.

6.2 System Master Control Processor

From the literature, there are three basic organizations and modes of operation for an operating system executive of a multiprocessor.

1. Master and slave: Executive always runs in one specific processor.
2. Separate executive: Each processor services its own need. Each processor has its own set of private tables.
3. Symmetric or anonymous treatment of all processors: Master floats from one processor to another.

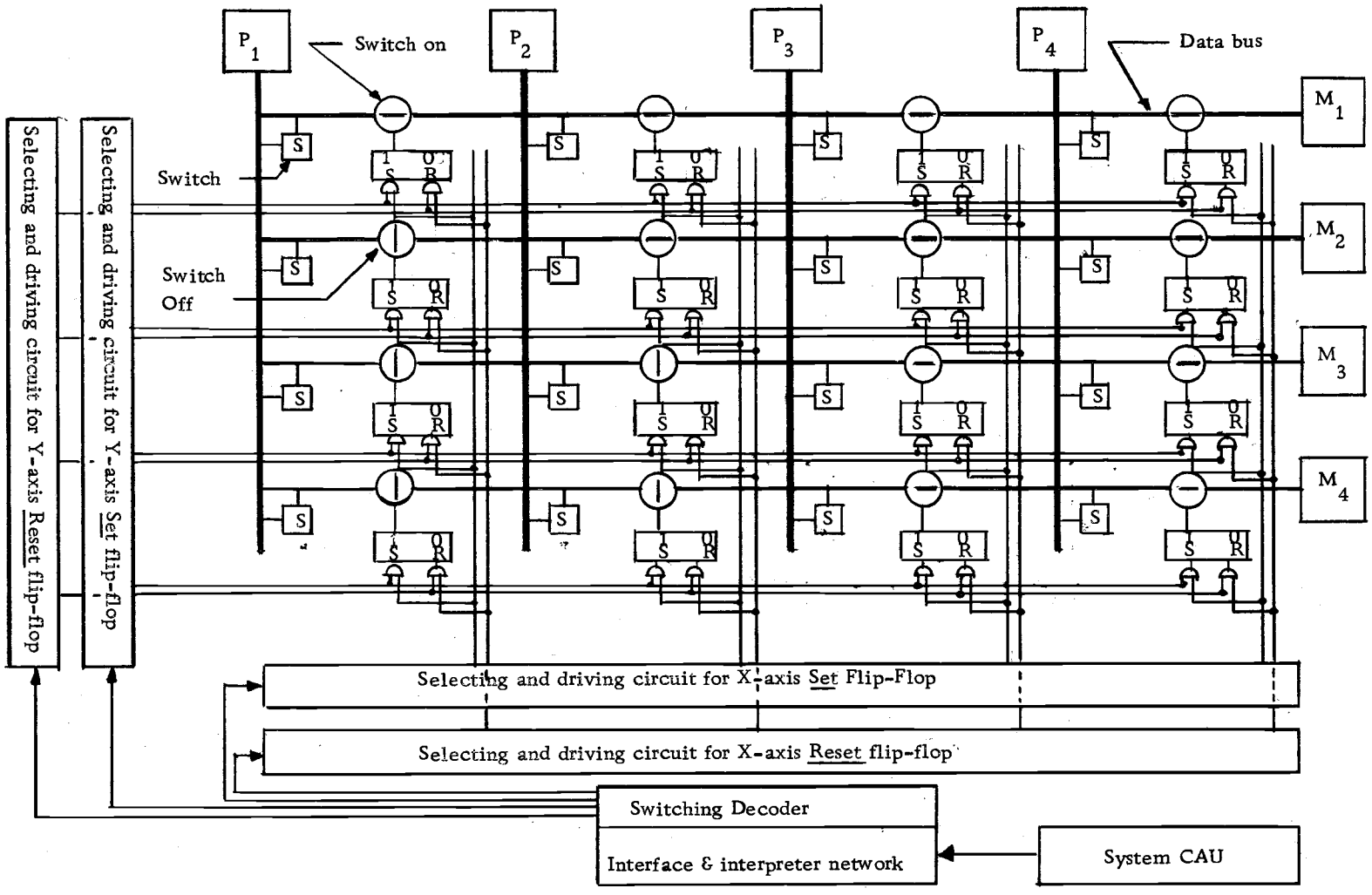


Figure 6.5. The design of switching mechanism for system reconfiguration (P = 4, M = 4, illustration).

Table 6.1. Defined configuration control code format.

Binary code	Representation
	3-Bit Field for type of device.
001	PU = Processing Unit
010	PPU = Peripheral Processing Unit
011	M = Memory Module
100	HBS = Horizontal Bus Switch
101	VBS = Vertical Bus Switch
111	No use (Spare)
000	No use (Spare)
	4-bit Field for number of devices
0000 1111	Device number 0 ⟶ 16
	2-bit Field for device status
00	No use (Spare)
01	Not available (switch out for maintenance or repair)
10	Disconnect
11	Connect
	2-bit Field for cross-bar switch number
00	No use (Spare)
01	PU & M Cross-bar switch unit
10	M & PPU Cross-bar switch unit
11	PU & PPU Cross-bar switch unit
	1-bit Field for position of bus in cross-bar switch
0	Horizontal bus
1	Vertical bus
	4-bit Field for bus number in cross-bar switch
0000 1111	Bus number 0 ⟶ 16

The details and characteristics of these three have been mentioned by Philip H. Enslow, Jr. (17). From an analysis of the proposed architecture of this command and control computer system, the third technique, seems to be the most suitable executive operation.

Since the PU in this computing system is a high power computation unit and so that the system may run at top-efficiency, the PUs should be busy with the user program load instead of with executive programs and for input/output operations. The suggested scheme for delegating system tasks is as follows.

1. PU Processing Unit
    a. Execution of users programs
    b. Compilation and execution of the task recognizing process.
2. PPU, Peripheral Processing Unit
    a. Handling I/O operation
    b. Handling executive program. Any PPU can be assigned to handle the executive task of the system as the Master Control PPU (i.e., executive task floating between PPU's).
3. Provide the CAU function.

In case the Master Control processor (PPU) has failed, and other available PPU can simply be assigned to be Master Control

processor. This provides several advantages including better availability potential, real redundancy, reliability and most efficient use of resources.

Since the executive scheme and the nature of the multiprocessor structure allows several processors to be in a supervisory state simultaneously, access conflicts can occur. However, both conflicts in service requests and those due to memory access can be resolved by priorities and the switching interlock network (crossbar switch).

VII. ANALYSIS AND EVALUATION OF THE PROPOSED SYSTEM

In this chapter, the proposed two level parallel processing architecture will be analyzed and evaluated. A program for solving N simultaneous non-linear equations using the Newton-Raphson Method was chosen as an example task. This example was given in (9).

First level parallelism is simulated by applying the Ramamoorthy-Gonzalez Parallel Task Recognizer algorithm to the sample task and its subroutines. A task table is constructed for scheduling and allocation. The results are compared with sequential processing.

Instruction level parallelism was hand simulated for a subset of instructions within one subroutine. Again, the results were compared with sequential processing. The instruction set used is given in Appendix A.

7.1 Task Level Parallel Simulation

The Ramamoorthy-Gonzalez algorithm (Chapter II) is applied to a FORTRAN program for solving N non-linear simultaneous equations using the Newton-Raphson technique.

The steps followed are:

1. Assign statement numbers as task labels for both main program and subroutines as shown in Figures 7.1, 7.5 and 7.8.

2. Derive the program graphs for the main program and subroutines. Typical graphs are shown in Figures 7.2 and 7.4.
3. Determine the reduced program graphs eliminating strongly connected subgraphs. For subroutine CALCN, graphs were determined with the original DO LOOPS and with the DO LOOPS replaced with the equivalent steps. For subroutine SIMUL, the DO LOOP represented in statements 87 through 118 was complex enough to be treated separately. Typical results are given in Figures 7.3 and 7.10.
4. Using the input/output relationships derive the final program graphs as shown in Figures 7.4, 7.6 and 7.11.
5. Derive the connectivity matrices and the precedence partitions using the Ramamoorthy-Gonzalez algorithm. This was not done for the main program, since it reduced to two tasks. The analysis of subroutines CALCN and SIMUL are shown in Figures 7.7, 7.12 and 7.13.
6. Construct a Task Scheduling Table using the precedance partitions. This is shown for subroutine CALCN and the two parts of subroutine SIMUL in Tables 7.1, 7.2 and 7.3.

The main program depends heavily on subroutine calls and DO LOOPS. Applying the Ramamoorthy-Gonzalez algorithm yields the two task graph shown in Figure 7.4. Obviously, this is a worst case example, where the algorithm produces no parallel processable tasks.

The additional overhead of using a FORTRAN Parallel Task Recognizer with subroutine analysis capability is a necessity for this example.

Applying the procedure to subroutine CALCN, in its original form, produces 38 parallel-processable tasks. This is a reasonable result, which, ignoring the overhead, would provide an immense saving in execution time over in-line sequential processing. An even more impressive result is obtained when the DO LOOPS are replaced by repetitive FORTRAN statements. A Task Scheduling Table was not derived for this case, but an examination of Figure 7.6(b) indicates the amount of achievable parallelism.

The final case examined is subroutine SIMUL. Here, good results are obtained when the subroutine is analyzed in the usual manner. Even better performance is possible when the DO LOOP contained in statements 87 through 118 is considered separately.

The results shown indicate that inclusion of a Parallel Task Recognizer in the operating system will provide the tool necessary to fully utilize the multiprocessor configuration. The recognizer should be capable of analysis at the subroutine and perhaps DO LOOP level to maximize its effectiveness. Restrictions on the language used and intelligent programming by the user will aid the performance.

The problem of the overhead incurred through use of such a recognizer, can be overcome in the command and control system. For our application, the user programs will most often be fixed or

Program Listing

Main Program

```
      C                APPLIED NUMERICAL METHODS,
      C                CHEMICAL EQUILIBRIUM - NEWTON-RAPHSON METHOD
      C
      C                THIS PROGRAM SOLVES N SIMULTANEOUS NON-LINEAR EQUATIONS
      C                IN N UNKNOWNS BY THE NEWTON-RAPHSON ITERATIVE PROCEDURE
      C                INITIAL GUESSES FOR VALUES OF THE UNKNOWNS ARE READ INTO
      C                XOLD(1)...XOLD(N). THE PROGRAM FIRST CALLS ON THE SUBROUTINE
      C                CALCN TO COMPUTE THE ELEMENTS OF A, THE AUGMENTED MATRIX
      C                OF PARTIAL DERIVATIVES, THEN ON FUNCTION SIMUL (SEE PROBLEM
      C                5.2) TO SOLVE THE GENERATED SET OF LINEAR EQUATIONS FOR THE
      C                CHANGES IN THE SOLUTION VALUES XINC(1)...XINC(N). DETER IS THE
      C                JACOBIAN COMPUTED BY SIMUL. THE SOLUTIONS ARE UPDATED AND THE
      C                PROCESS CONTINUED UNTIL ITER, THE NUMBER OF ITERATIONS,
      C                EXCEEDS ITMAX OR UNTIL THE CHANGE IN EACH OF THE N VARIABLES
      C                IS SMALLER IN MAGNITUDE THAN EPS2 (ITCON + 1 UNDER THESE
      C                CONDITIONS). EPS1 IS THE MINIMUM PIVOT MAGNITUDE PERMITTED
      C                IN SIMUL. WHEN IPRING = 1, INTERMEDIATE SOLUTION VALUES ARE
      C                PRINTED AFTER EACH ITERATION.
      C
      C               DIMENSION XOLD(21), XINC(21), A(21,21)
      C
      C               ....READ AND PRINT DATA ....
    1          1      READ (5,100) ITMAX, IPRINT, N, EPS1, EPS2, (XOLD(1), I=1,N)
    2                 WRITE (6,200) ITMAX, IPRINT, N, EPS1, EPS2, N, (XOLD(I), I=1,N)
      C
      C               .... NEWTON-RAPHSON ITERATION .....
    3                 DO 9 ITER = 1, ITMAX
      C
      C               ..... CALL ON CALCN TO SET UP THE A MATRIX .....
    4                 CALL CALCN( XOLD, A, 21)
      C
      C               ..... CALL SIMUL TO COMPUTE JACOBIAN AND CORRECTIONS IN XINC...
    5                 DETER = SIMUL( N, A, XINC, EPS1, 1, 21 )
    6                 IF ( DETER.NE.0. ) GO TO 3
    7                   WRITE (6,201)
    8                   GO TO 1
      C
      C               .....CHECK FOR CONVERGENCE AND UPDATE XOLD VALUES .....
    9          3      ITCON = 1
   10                 DO 5 I = 1, N
   11                 IF ( ABS(XINC(I)).GT.EPS2 ) ITCON = 0
   12          5      XOLD(I) = XOLD(I) + XINC(I)
   13                 IF ( IPRINT.EQ.1 ) WRITE (6,202) ITER, DETER, N, (XOLD(I), I=1,N)
   14                 IF ( ITCON.EQ.0 ) GO TO 9
   15                   WRITE (6,203) ITER, N, (XOLD(I), I=1,N)
   16                   GO TO 1
   17          9      CONTINUE
```

Figure 7.1. Continued

```
      C
1  8                  WRITE (6, 204)
1  9                  GO TO 1
      C
      C               ..... FORMATS FOR INPUT AND OUTPUT STATEMENTS .....
          100         FORMAT ( 10X, 13, 17X, I1, 19X, 13/ 10X, E7.1, 13X, E7.1/ (20X, 5F10.3) )
          200         FORMAT (10X1ITMAX = , 18/ 10H IPRINT = , 18/ 10H N    =,
                     1  18/ 10H EPS1 = , 12E14.1/ 10H EPS2 = , IPE14.1/
                     2  26H0          HOLD(1)...HOLD(, 12, 1H)/ 1H / (1H , 1P4E16.6) )
          201         FORMAT ( 38HOMATRIX-IS-ILL-CONDITIONED-OR-SINGULAR-)
          202         FORMAT ( 10HOITER ---=-, 18/ 10H DETER--= , E18.5/
                     2  26H         XOLD(1)...XOLD(, 12, 1H) / (1H , I24E16.6) )
          203         FORMAT ( 24HOSUCCESSFUL-CONVERGENCE-/-10HOITER---=-, 13/
                     2  26HO       XOLD(1)...XOLD(, 12, 1H) / 1H / (1H, IP4E16.6) )
          204         FORMAT ( 15H NO CONVERGENCE )
      C
2  0                  END
```

Figure 7.1. Main program example.

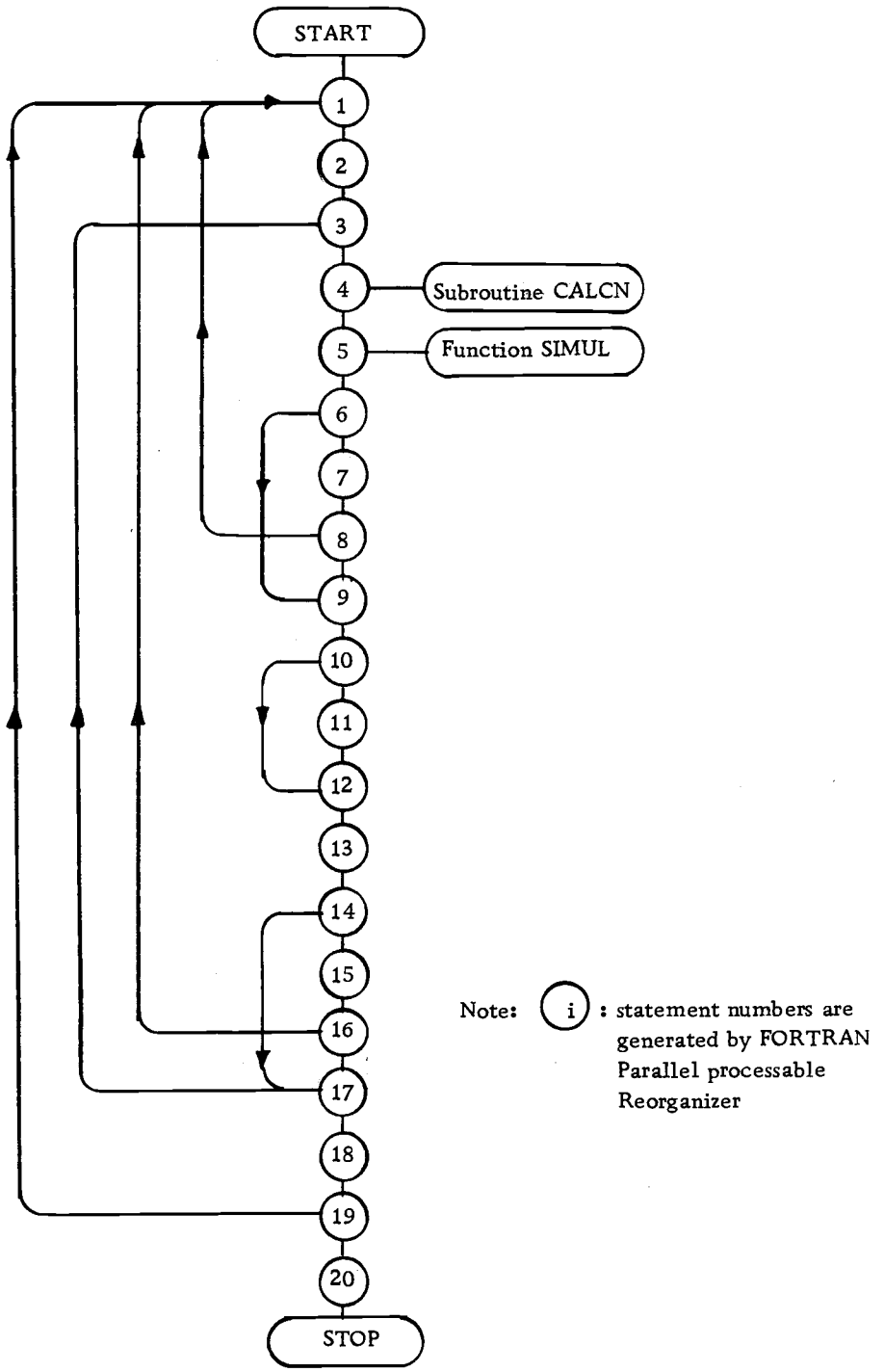


Figure 7.2. Flow-chart diagram of program in Figure 7.1.

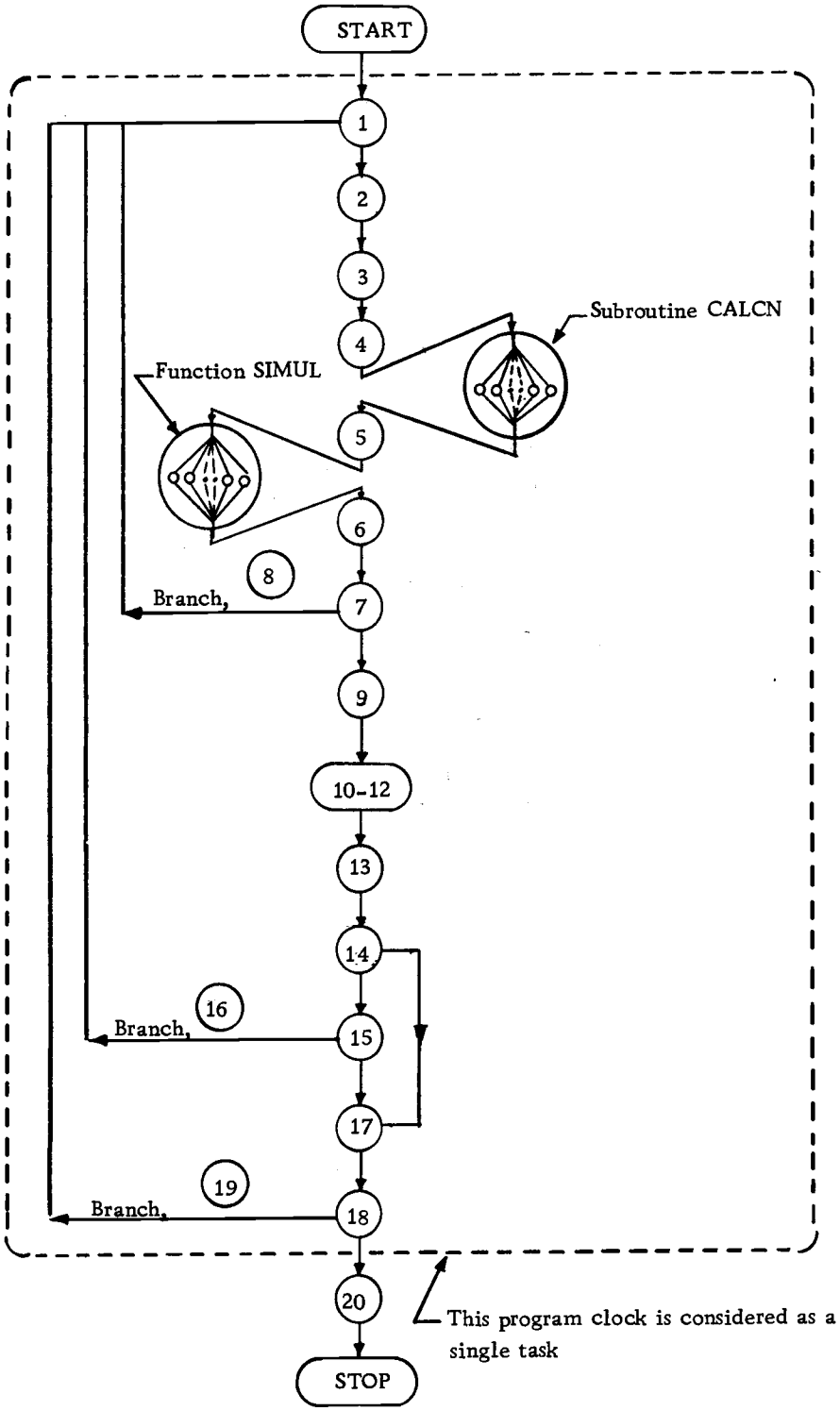


Figure 7.3. A partially reduced graph and Final Program graph of the Main Program.

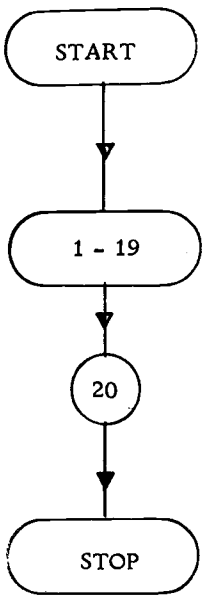


Figure 7. 4. Final reduced program graph for the main program of Figure 7. 1.

Program Listing (Continued)

Subroutine CALCN

```
2  9      C                SUBROUTINE CALCN( DXOLD, A, NRC )
          C
          C                    THIS SUBROUTINE SETS UP THE AUGMENTED MATRIX OF PARTIAL
          C                    DERIVATIVES REQUIRED FOR THE SOLUTION OF THE NON-LINEAR
          C                    EQUATIONS WHICH DESCRIBE THE EQUILIBRIUM CONCENTRATIONS
          C                    OF CHEMICAL CONSTITUENTS RESULTING FROM PARTIAL OXIDATION
          C                    OF METHANE WITH OXYGEN TO PRODUCE SYNTHESIS GAS. THE PRESSURE
          C                    IS 20 ATMOSPHERES. SEE TEXT FOR MEANINGS OF XOLD(1)...XOLD(N)
          C                    AND A LISTING OF THE EQUATIONS. DXOLD HAS BEEN USED AS THE
          C                    DUMMY ARGUMENT FOR XOLD TO AVOID AN EXCESSIVE NUMBER OF
          C                    REFERENCES TO ELEMENTS IN THE ARGUMENT LIST.
          C
                           DIMENSION XOLD(20), DXOLD(21), A(1) A(21,21)
          C
                           P = 20.
          C
          C                ..... SHIFT ELEMENTS OF DXOLD TO XOLD AND CLEAR A ARRAY .....
3  0                     DO 1  I = 1, 7
3  1                     XOLD(I) = DXOLD(I)
3  2                     DO 1  J = 1, 8
3  3              1      A(I,J) = 0.
          C
          C                ..... COMPUTE NON-ZERO ELEMENTS OF A .....
3  4                     A(1,1) = 0.5
3  5                     A(1,2) = 1.0
3  6                     A(1,3) = 0.5
3  7                     A(1,6) = -1.0/XOLD(7)
3  8                     A(1,7) = XOLD(6)/XOLD(7)**2
3  9                     A(1,8) = -XOLD(1)/2. - XOLD(2) - XOLD(3)/2. + XOLD(6)/XOLD(7)
4  0                     A(2,3) = 1.0
4  1                     A(2,4) = 1.0
```

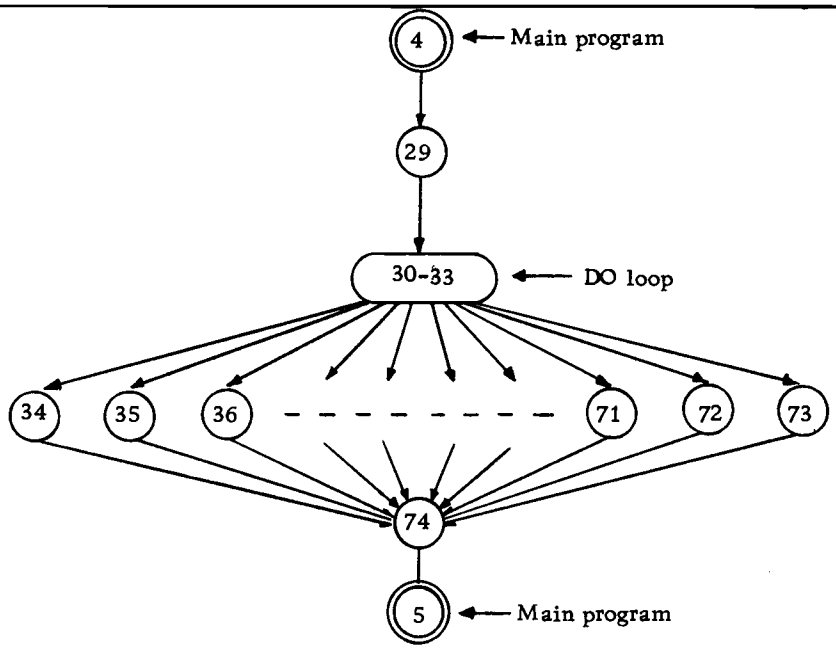
Figure 7.5. Continued

```
4  2                A(2, 5) = 2.0
4  3                A(2, 7) = 2.0/XOLD(7)**2
4  4                A(2, 8) = -XOLD(3) - XOLD(4) - 2.0*XOLD(5) + 2.0/XOLD(7)
4  5                A(3, 1) = 1.0
4  6                A(3, 2) = 1.0
4  7                A(3, 5) = 1.0
4  8                A(3, 7) = 1.0/XOLD(7)**2
4  9                A(3, 8) = -XOLD(1) - XOLD(2) - XOLD(5) + 1.0/XOLD(7)
5  0                A(4, 1) = -28837.
5  1                A(4, 2) = -139009.
5  2                A(4, 3) = -78213
5  3                A(4, 4) = 18927.
5  4                A(4, 5) = 8427.
5  5                A(4, 6) = -10690./XOLD(7)
5  6                A(4, 7) = (-13492. + 10690. *XOLD(6))/XOLD(7)**2
5  7                A(4, 8) = 28837.*XOLD(1) + 139009.*XOLD(2) + 78213.*XOLD(3)
                   1         -18927.*XOLD(4) - 8427.*XOLD(5) - 13492./XOLD(7) + 10690.
                   2         *XOLD(6)/XOLD(7)
5  8                A(5, 1) = 1.0
5  9                A(5, 2) = 1.0
6  0                A(5, 3) = 1.0
6  1                A(5, 4) = 1.0
6  2                A(5, 5) = 1.0
6  3                A(5, 8) = 1.0 - XOLD(1) - XOLD(2) - XOLD(3) - XOLD(4) - XOLD(5)
6  4                A(6, 1) = P*P*XOLD(4)**3
6  5                A(6, 3) = -1.7837E5*XOLD(5)
6  6                A(6, 4) = 3.0*P*P*XOLD(1)*XOLD(4)**2
6  7                A(6, 5) = -1.7837E5*XOLD(3)
6  8                A(6, 8) = 1.7837E5*XOLD(3) - P*P*XOLD(1)*XOLD(4)**3
6  9                A(7, 1) = XOLD(3)
7  0                A(7, 2) = -2.6058*XOLD(4)
7  1                A(7, 3) = XOLD(1)
```

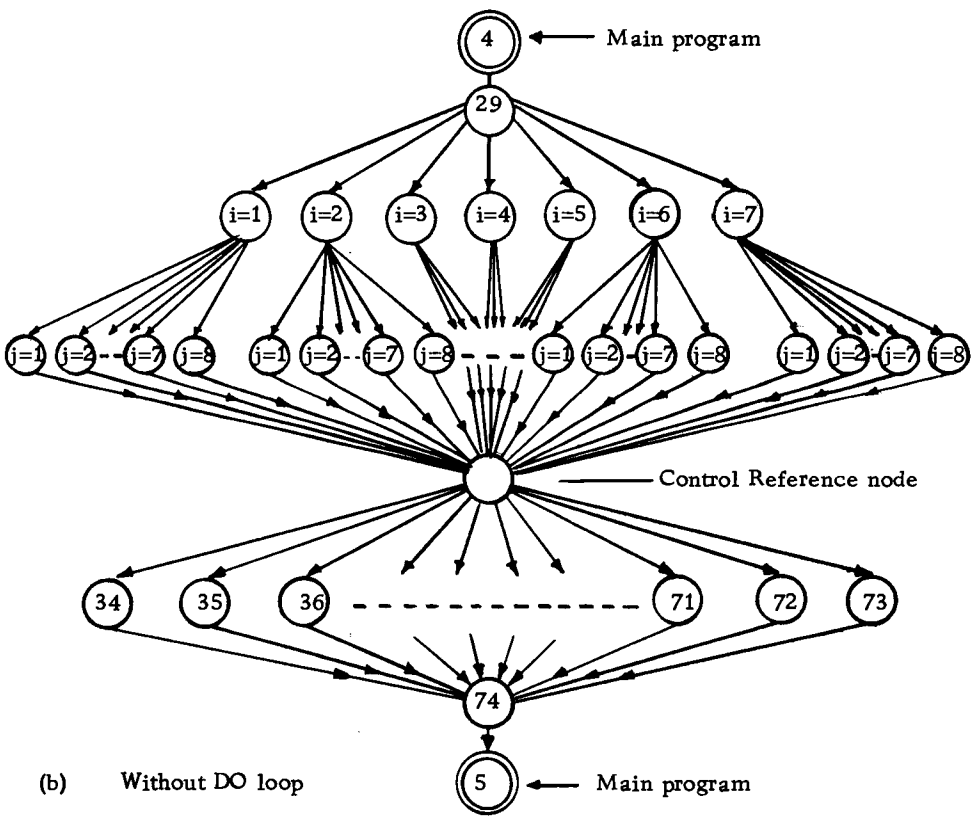
Figure 7.5. Continued

```
7   2                   A(7, 4) = -2.6058*XOLD(2)
7   3                   A(7, 8) = 2.6058*XOLD(4)*XOLD(2) - XOLD(1)*XOLD(3)
7   4                   RETURN
        C
7   5                   END
```

Figure 7.5. Subroutine CALCN.



(a) Final graph of CALCN subroutine.



(b) Without DO loop

Figure 7.6. Final program graph of CALCN with DO loop.

Successive Tasks

Current Tasks

	29	30*	34	35	36	.	.	.	.	.	71	72	73	74
29	0	1	0	0	0	0	0	· · ·	0	0	0	0	0	0
30*	0	0	1	1	1	1	1	· · ·	1	1	1	1	1	0
34	0	0	0	0	0	0	0	0	0	0	0	0	0	1
35	0	0	0	0	0	0	0	0	0	0	0	0	0	1
36	0	0	0	0	0	0	0	0	0	0	0	0	0	1
.	0	0	0	0	0	0	0	0	0	0	0	0	0	1
.	0	0	0	0	0	0	0	0	0	0	0	0	0	1
.	0	0	0	0	0	0	0	0	0	0	0	0	0	1
.	0	0	0	0	0	0	0	0	0	0	0	0	0	1
.	0	0	0	0	0	0	0	0	0	0	0	0	0	1
71	0	0	0	0	0	0	0	0	0	0	0	0	0	1
72	0	0	0	0	0	0	0	0	0	0	0	0	0	1
73	0	0	0	0	0	0	0	0	0	0	0	0	0	1
74	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a) Connectivity matrix of final program of subroutine CALCN

Groups	Table Precedence position of CALCN	Remark
1	[29]	
2	[30*]	
3	[34, 35, 36, . . ., 71, 72, 73]	
4	[74]	

Note: 30* = 30 → 33

(b) Precedent partition of subroutine CALCN

Figure 7.7. Analysis of subroutine CALCN.

Table 7.1. Task scheduling table of subroutine CALCN.

Time	Input to tasks	Task number	Task Type	
			Input	Output
t_1	--	29		
t_2	29	30*		Fork
t_3	30*	34		
t_3	30*	35		
t_3	30*	36		
t_3	⋮	⋮		
t_3	⋮	⋮		
t_3	30*	71		
t_3	30*	72		
t_3	30*	73		
t_4	34, 35, 36. . . 72, 73	74	join	

t_i, i = 1, 2, 3, 4 based on this task scheduling table only.

```
79              FUNCTION SIMUL( N, A, X, EPS, INDIC, NRC )
      C
      C         WHEN INDIC IS NEGATIVE, SIMUL COMPUTES THE INVERSE OF THE N BY
      C         N MATRIX A IN PLACE. WHEN INDIC IS ZERO, SIMUL COMPUTES THE
      C         N SOLUTIONS X(1)...X(N) CORRESPONDING TO THE SET OF LINEAR
      C         EQUATIONS WITH AUGMENTED MATRIX OF COEFFICIENTS IN THE N BY
      C         N+1 ARRAY A AND IN ADDITION COMPUTES THE INVERSE OF THE
      C         COEFFICIENT MATRIX IN PLACE AS ABOVE. IF INDIC IS POSITIVE,
      C         THE SET OF LINEAR EQUATIONS IS SOLVED BUT THE INVERSE IS NOT
      C         COMPUTED IN PLACE. THE GAUSS-JORDAN COMPLETE ELIMINATION METHOD
      C         IS EMPLOYED WITH THE MAXIMUM PIVOT STRATEGY. ROW AND COLUMN
      C         SUBSCRIPTS OF SUCCESSIVE PIVOT ELEMENTS ARE SAVED IN ORDER IN
      C         THE IROW AND JCOL ARRAYS RESPECTIVELY. K IS THE PIVOT COUNTER,
      C         PIVOT THE ALGEBRAIC VALUE OF THE PIVOT ELEMENT, MAX
      C         THE NUMBER OF COLUMNS IN A AND DETER THE DETERMINANT OF THE
      C         COEFFICIENT MATRIX. THE SOLUTIONS ARE COMPUTED IN THE (N+1)TH
      C         COLUMN OF A AND THEN UNSCRAMBLED AND PUT IN PROPER ORDER IN
      C         X(1)...X(N) USING THE PIVOT SUBSCRIPT INFORMATION AVAILABLE
      C         IN THE IROW AND JCOL ARRAYS. THE SIGN OF THE DETERMINANT IS
      C         ADJUSTED, IF NECESSARY, BY DETERMINING IF AN ODD OR EVEN NUMBER
      C         OF PAIRWISE INTERCHANGES IS REQUIRED TO PUT THE ELEMENTS OF THE
      C         JORD ARRAY IN ASCENDING SEQUENCE WHERE JORD(IROW(I)) = JCOL(I).
      C         IF THE INVERSE IS REQUIRED, IT IS UNSCRAMBLED IN PLACE USING
      C         Y(1)...Y(N) AS TEMPORARY STORAGE. THE VALUE OF THE DETERMINANT
      C         IS RETURNED AS THE VALUE OF THE FUNCTION. SHOULD THE POTENTIAL
      C         PIVOT OF LARGEST MAGNITUDE BE SMALLER IN MAGNITUDE THAN EPS,
      C         THE MATRIX IS CONSIDERED TO BE SINGULAR AND A TRUE ZERO IS
      C         RETURNED AS THE VALUE OF THE FUNCTION.
      C
                DIMENSION IROW(50), JCOL(50), JORD(50), Y(50), A(1), X(1)
      C
80              MAX = N
81              IF ( INDIC.GE.0 ) MAX + N + 1
      C
```

Figure 7.8. Continued

```
      C              .....IS N LARGER THAN 50 .....
 82                  IF ( N.LE.50 )  GO TO 5
 83                  WRITE (6, 200)
 84                  SIMUL = 0.
 85                  RETURN
      C
      C              ..... BEGIN ELIMINATION PROCEDURE
 86         5        DETER = 1.
 87                  DO 18  K = 1, N
 88                  KM1 = K - 1
      C
      C              ..... SEARCH FOR THE PIVOT ELEMENT .....
 89                  PIVOT = 0.
 90                  DO 11  I = 1, N
 91                  DO 11  J = 1, N
      C              ..... SCAN IROW AND JCOL ARRAYS FOR INVALID PIVOT SUBSCRIPTS .....
 92                  IF ( K.EQ.1 )  GO TO 9
 93                  DO 8   ISCAN  =  1, KM1
 94                  DO 8   JSCAN  =  1, KM1
 95                  IF ( I.EQ.IROW(ISCAN) )  GO TO 11
 96                  IF ( J.EQ.JCOL(JSCAN) )  GO TO 11
 97         8        CONTINUE
 98         9        IF (BABS(A(I, J)).LE.BABS(PIVOT)  GO TO 11
 99                  PIVOT = A(I, J)
100                  IROW(K) = 1
101                  JCOL(K) = J
102        11        CONTINUE
      C
      C              ..... INSURE THAT SELECTED PIVOT IS LARGER THAN EPS .....
103                  IF ( BABS(PIVOT).GT.EPS )  GO TO 13
104                  SIMUL = 0.
105                  RETURN
      C
```

Figure 7.8. Continued

```
      C               ..... UPDATE THE DETERMINANT VALUE .....
106         13        IROWK = IROW(K)
107                   JCOLK = JCOL(K)
108                   DETER = DETER*PIVOT
      C               .....NORMALIZE PIVOT ROW ELEMENTS .....
109                   DO 14 J = 1, MAX
110         14        A(IROWK, J) = A(IROWK, J)/PIVOT
      C
      C               ..... CARRY OUT ELIMINATION AND DEVELOP INVERSE .....
111                   A(IROWK, JCOLK) = 1./PIVOT
112                   DO 18 I = 1, N
113                   AIJCK = A(I, JCOLK)
114                   IF ( I.EQ.IROWK ) GO TO 18
115                   A(I, JCOLK) = - AIJCK/PIVOT
116                   DO 17 J = 1, MAX
117         17        IF ( J.NE.JCOLK) A(I, J) = A(I, J) - AIJCK*A(IROWK, J)
118         18        CONTINUE
      C
      C               ..... ORDER SOLUTION VALUES (IF ANY) AND CREATE JORD ARRAY .....
119                   DO 20 I = 1, N
120                   IROWI = IROW(I)
121                   JCOLI = JCOL(I)
122                   JORD(IROWI) = JCOLI
123         20        IF ( INDIC.GE.0 ) X(JOCLI = A(IROWI, MAX)
      C
      C               ..... ADJUST SIGN OF DETERMINANT .....
124                   INTCH = 0
125                   NM1 = N - 1
126                   DO 22 I = 1, NM1
127                   IP1 = I + 1
128                   DO 22 J = IP1, N
129                   IF ( JORD(J).GE.JORD(I))  GO TO 22
```

Figure 7.8. Continued

```
130                     JTEMP = JORD(J)
131                     JORD(J) = JORD(I)
132                     JORD(I) = JTEMP
133                     INTCH = INTCH + 1
134           22        CONTINUE
135                     IF ( INTCH/2*2.NE.INTCH )  DETER + - DETER
      C
      C                 ..... IF INDIC IS POSITIVE RETURN WITH RESULTS .....
136                     IF ( INDIC.LE.0 )  GO TO 26
137                     SIMUL = DETER
138                     RETURN
      C
      C                 ..... IF INDIC IS NEGATIVE OR ZERO, UNSCRAMBLE THE INVERSE FIRST BY ROWS .....
139           26        DO 28  J = 1,  N
140                     DO 27  I = 1,  N
141                     IROWI = IROW(I)
142                     JCOLI = JCOL(I)
143           27        Y(JCOLI) = A(IROWI, J)
144                     Do 28  I = 1,  N
145           28        A(I, J) = Y(I)
      C                 ..... THEN BY COLUMNS .....
146                     DO 30  I = i,  N
147                     DO 29  J = 1,  N
148                     IROWJ = IROW(J)
149                     JCOLJ = JCOL(J)
150           29        Y(IROWJ) = A(I, JCOLJ)
151                     DO 30  J = 1,  N
152           30        A(I, J) = Y(J)
      C
      C                 ..... RETURN FOR INDIC NEGATIVE OR ZERO .....
153                     SIMUL = DETER
```

Figure 7.8. Continued

```
154                     RETURN
      C
      C                 ..... FORMAT FOR OUTPUT STATEMENT .....
          200           FORMAT( 10HON TOO BIG)
      C
155                     END
```

Figure 7.8. Function SIMUL

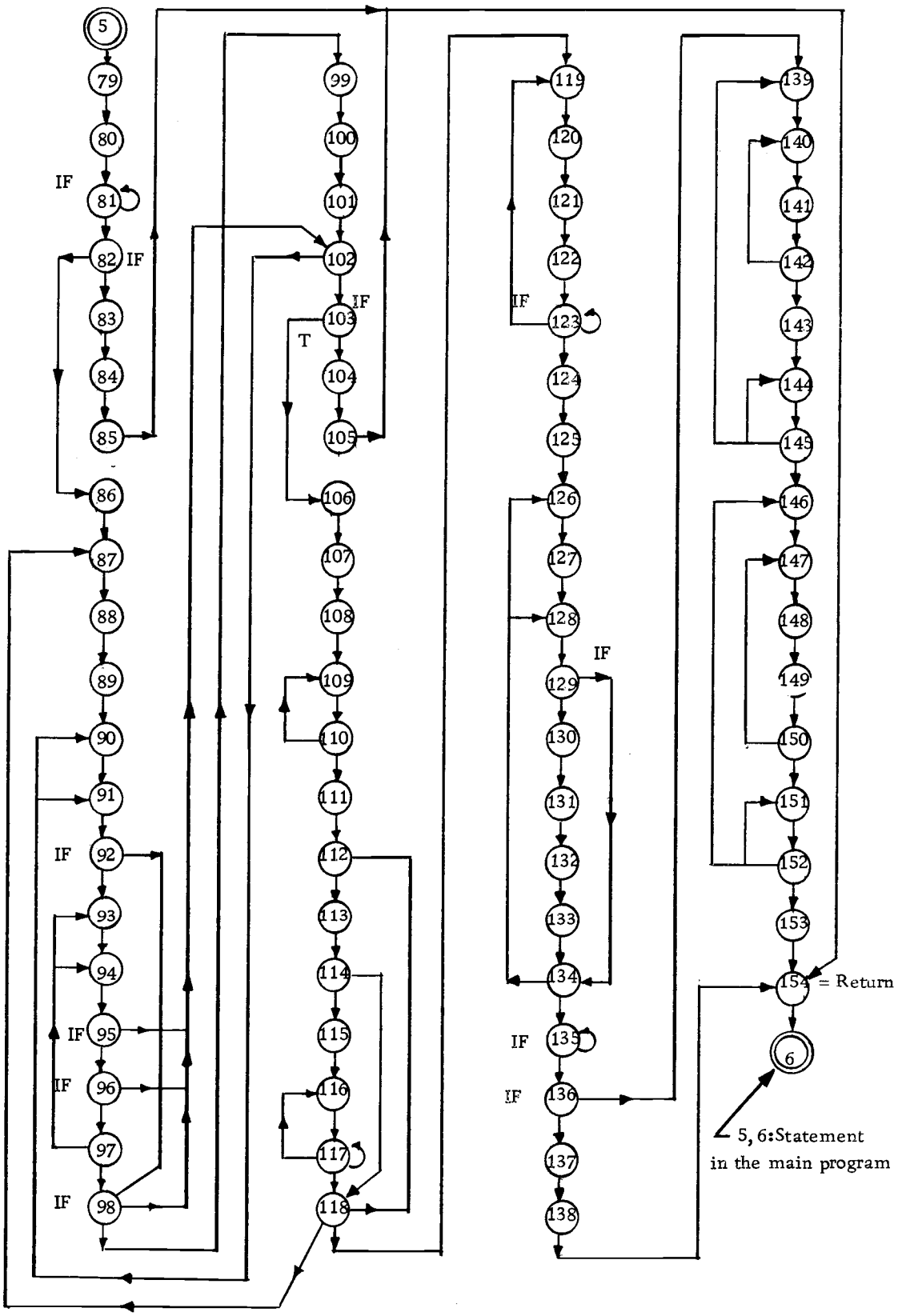


Figure 7.9. The flow chart of Function SIMUL.

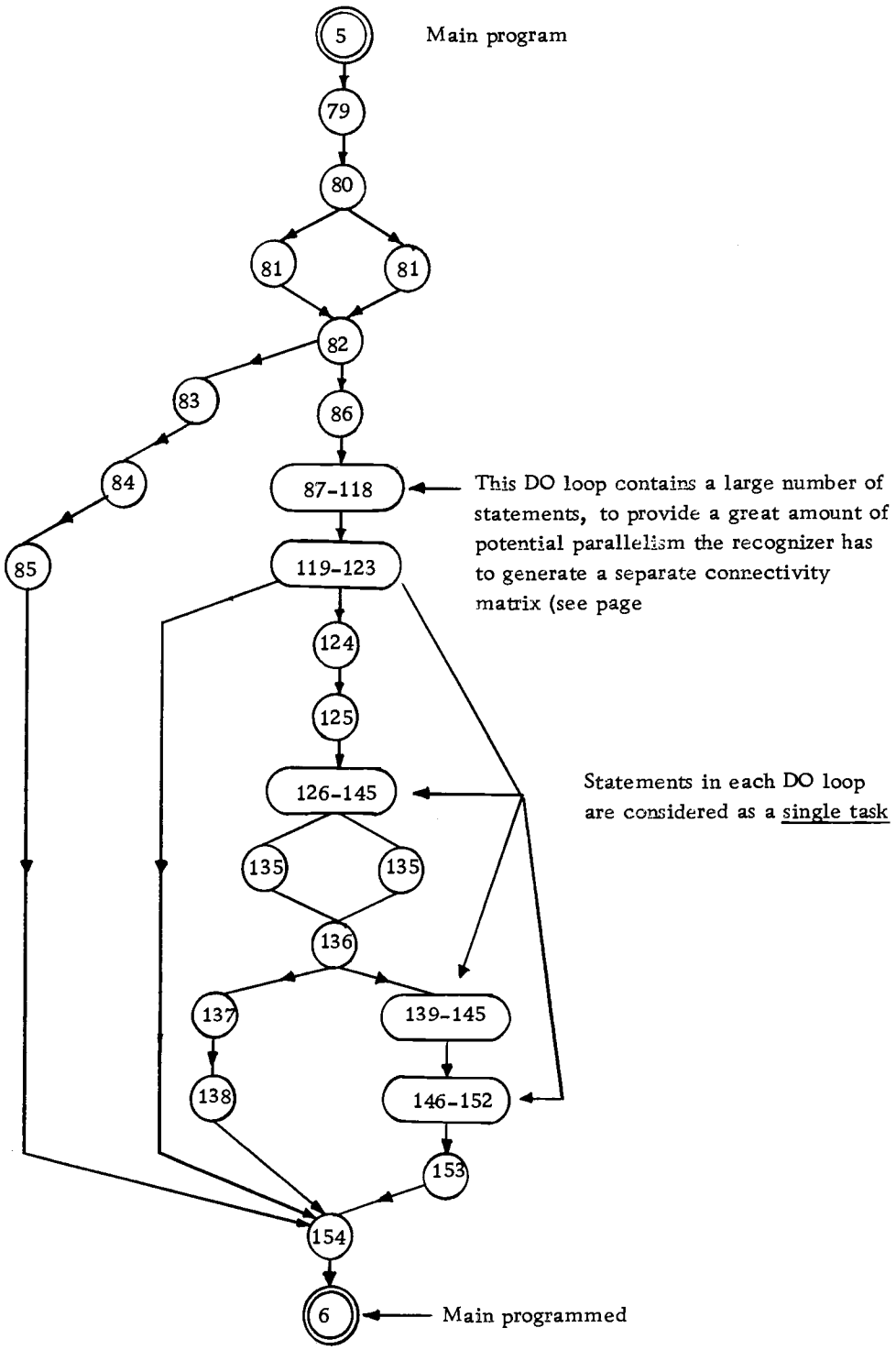


Figure 7.10. Reduced Program graph and Function SIMUL.

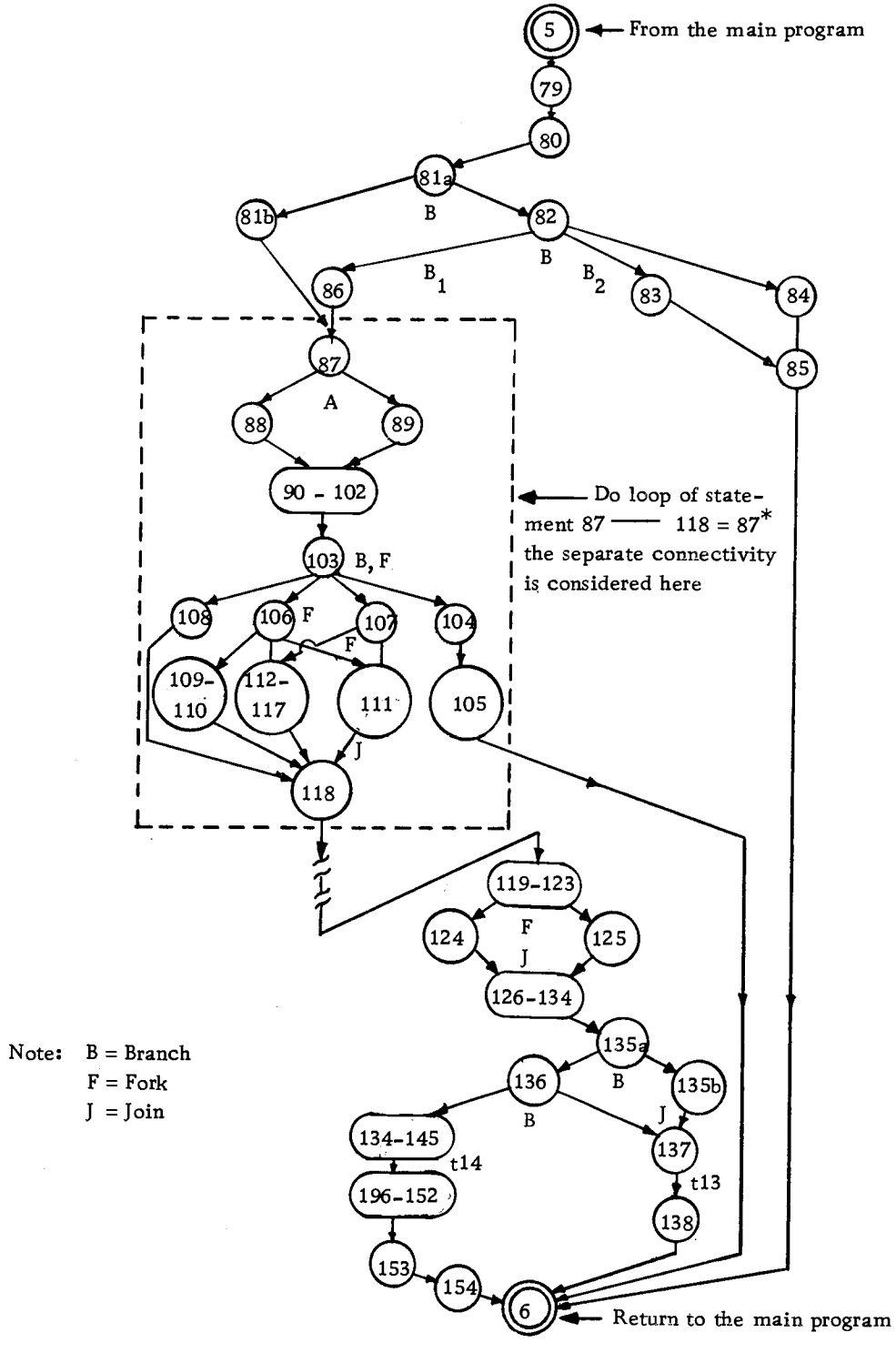


Figure 7.11. Final program graph of function SIMUL.

Successive tasks

Current tasks	79	80	81a	81b	82	83	84	85	86	87*	119*	124	125	126*	135a	135b	136	137	138	139*	146*	153	154	6
79	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
80	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
81a	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
81b	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
82	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
83	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
84	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
85	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
86	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
87*	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
119*	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
124	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
125	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
126*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
135a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
135b	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
136	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
137	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
138	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
139*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
146*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
153	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
154	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a) Connectivity matrix of final program graph of function SIMUL.

Groups	Table Precedence position of SIMUL	Remark
1	[79]	
2	[80]	
3	[81a]	
4	[82, 81b]	
5	[83, 84, 86]	
6	[85, 87*]	
7	[119*]	
8	[124, 125]	
9	[126*]	
10	[135a]	
11	[136, 135b]	
12	[137, 139*]	
13	[138, 146*]	
14	[153]	
15	[154]	

(b) Precedence partition SIMUL.

Note: $87^*_* = 87 \rightarrow 118$
$119^*_* = 119 \rightarrow 123$
$126^*_* = 126 \rightarrow 134$
$139^*_* = 139 \rightarrow 145$
$146^*_* = 146 \rightarrow 152$

Figure 7.12. Analysis of function SIMUL.

Table 7. 2. Task scheduling table of function SIMUL.

Time	Input to tasks	Task number	Task Type	
			Input side	Output side
t_1	--	79		
t_2	79	80		
t_3	80	81a		Branch
t_3	81a	81b		
t_4	81a	82		Fork Branch
t_5	82	83		
t_5	82	84		
t_5	82	86		
t_6	83, 84	85	Join	
t_6	81b, 86	87*	Join	
t_7	87*	119*		Fork
t_8	119*	124		
t_8	119*	125		
t_9	124, 125	126*	Join	
t_{10}	126*	135a		Branch
t_{10}	135a	135b		
t_{11}	135a	136		Branch
t_{12}	135b, 136	137	Join	
t_{12}	136	139*		
t_{13}	137	138		
t_{14}	139*	146*		
t_{15}	146*	153		
t_{16}	153	154		
t_{17}	85, 138, 154	6		

Successive Tasks

Current Tasks	87	88	89	90*	103	104	105	106	107	108	109*	111	112*	118	6
87	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
88	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
89	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
90*	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
103	0	0	0	0	0	1	0	1	1	1	0	0	0	0	0
104	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
105	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
106	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
107	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
108	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
109*	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
111	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
112*	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
118	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a)

Groups	Table Precedence position of 87 - 118	Remark
1	[87]	
2	[88, 89]	
3	[90*]	
4	[103]	
5	[104, 106, 107, 108]	
6	[105, 109*, 111, 112*]	
7	[118]	

(b) Precedence partition of statements 87 118

Note: $90^{*} = 90 \rightarrow 102$
$109^{*} = 109 \rightarrow 110$
$112^{*} = 112 \rightarrow 117$

(a) Connectivity matrix of final program graph of statements 87 ⟶ 118, function SIMUL.

Figure 7.13. Analysis of function SIMUL.

Table 7.3. Task scheduling table of statements 87 → 118 of function SIMUL.

Time	Input to tasks	Task number	Task Type	
			Input side	Output side
t_1	81b, 86	87		Fork
t_2	87	88		
t_2	87	89		
t_3	88, 89	90*	Join	
t_4	90*	103		Branch, Fork
t_5	103	104		
t_5	103	106		Fork
t_5	103	107		Fork
t_5	103	108		
t_6	104	105		
t_6	106	109*		
t_6	106, 107	111	Join	
t_6	106, 107	112*	Join	
t_7	109*, 111, 112*, 108	118	Join	
t_8	105	6		

Note:

(1) *See code description on page

(2) t_i, $i = 1, 2, \ldots 8$ based on this task scheduling table only.

"canned" programs to handle air traffic control. The overhead problem, then, has to be faced only once, at the initial run. For later execution, the compilation - parallel task recognition phase need not be repeated.

7.2 Instruction Level Parallelism Simulation

A series of FORTRAN statements from subroutine CALCN are examined to determine the amount of overlap execution that can be performed. A detailed timing analysis is derived for both overlap and sequential execution. Three basic assumptions are used in the timing analysis of the examples.

1. Consecutive instruction words from main storage require a minimum of eight minor cycles (800 nsec.) to fetch.
2. Both 16-bit and 32-bits instruction require one minor cycle to issue.
3. A functional unit is free one minor cycle after the result is placed in a register.

Timing analysis of the concurrent execution simulation is given in Tables B.1-B.5 of Appendix B and are listed by minor cycle count, with six headings corresponding to the execution stages. These are:

ISSUE = Time of instruction issue

START = Start of function operation

RESULT = Function complete with result available

UNIT FREE = Functional unit is ready for reuse

FETCH = Operand fetched from storage and available in X register

STORE = Result stored in location specified by A register

A STORE MACRO (BASE, SIM1, DIM2, I, J) is created and timed. The MACRO machine language instruction is used to represent the array addressing necessary to store array data. The parameters are BASE, the base address of the array; DIM1, the first dimension of the array; DIM2, the second dimension of the array; I and J, the array subscripts. A similar MACRO for fetch is also defined.

Since the FORTRAN statements chosen for simulation are generally array statements, it was felt that the creation of the FETCH and STORE MACROS would aid in performing the timing analysis. Compilation of array FORTRAN statements would necessarily rely on some array addressing scheme. The STORE MACRO is analyzed first and the results used in complete simulation.

Three cases were considered:

1. Memory not interleaved, average memory conflict = 3
2. Memory not interleaved, average memory conflict = 0
3. Memory interleaved, average memory conflict = 0

A complete analysis of each case for both parallel and sequential execution is given in Tables B. 1-B. 3 of Appendix B. A

timing diagram for the third case, comparing overlap and sequential execution is given in Figure 7.14.

The part of subroutine CALCN to be simulated consists of six FORTRAN statements shown below:

Task No.	FORTRAN Statement
C	Computed non-zero elements of A ...
34	A(1, 1) = 0.5
35	A(1, 2) = 1.0
36	A(1, 3) = 0.5
37	A(1, 6) = -1.0/XOLD(7)
38	A(1, 7) = XOLD(6)/XOLD(7)**2
39	A(1, 8) = XOLD(1)/2. -XOLD(2)-XOLD(3)/

Assume that the following storage locations contain the data shown:

Location	Stored Content
K5	(0.5)
K6	(1.0)
K7	(1.0)
K8	(-1.0)
M1=XOLD(1)	(a)
M2=XOLD(2)	(b)
M3=XOLD(3)	(c)
M4=XOLD(6)	(d)
M5 = XOLD(7)	(e)

The elements of matrix A are mapped in the main core memory in a linear-array fashion. Each time an element A(i, j) of matrix A is referenced (fetch or store), the indexes i and j and other information such as base address (BASE), and row dimension (DIM1), are used to compute the actual physical address of the corresponding linear array element. The selected mapping function can be expressed by the following mathematic model:

$$L_k = BASE + (I-1) * DIM1 + J$$

where

L_k = the address of linear array element.

The equivalent machine language routine for the mapping function and the time analysis for both concurrent and sequential execution is just that already shown for the FETCH or STORE MACROS.

An equivalent machine language program for the six FORTRAN statements is given in Tables B 4 and B. 5 of Appendix B. Figure 7. 15 compares timing for concurrent and sequential execution.

The intermediate sums for both concurrent execution times and sequential execution times with respect to the same executed FORTRAN statement are determined and the ratios between these times are computed and shown in Figure 7. 16. This plot shows the ratio of accumulated execution times versus the number of machine

1. Fetch BASE
2. Fetch I
3. Fetch J
4. Fetch D1M1
5. Form BASE + I
6. Set B1 = -1
7. Form J-1
8. Form (J-1)*D1M1
9. Form BASE + I + (J-1)*D1M1
10. Store X7 at loc. A (I, J)

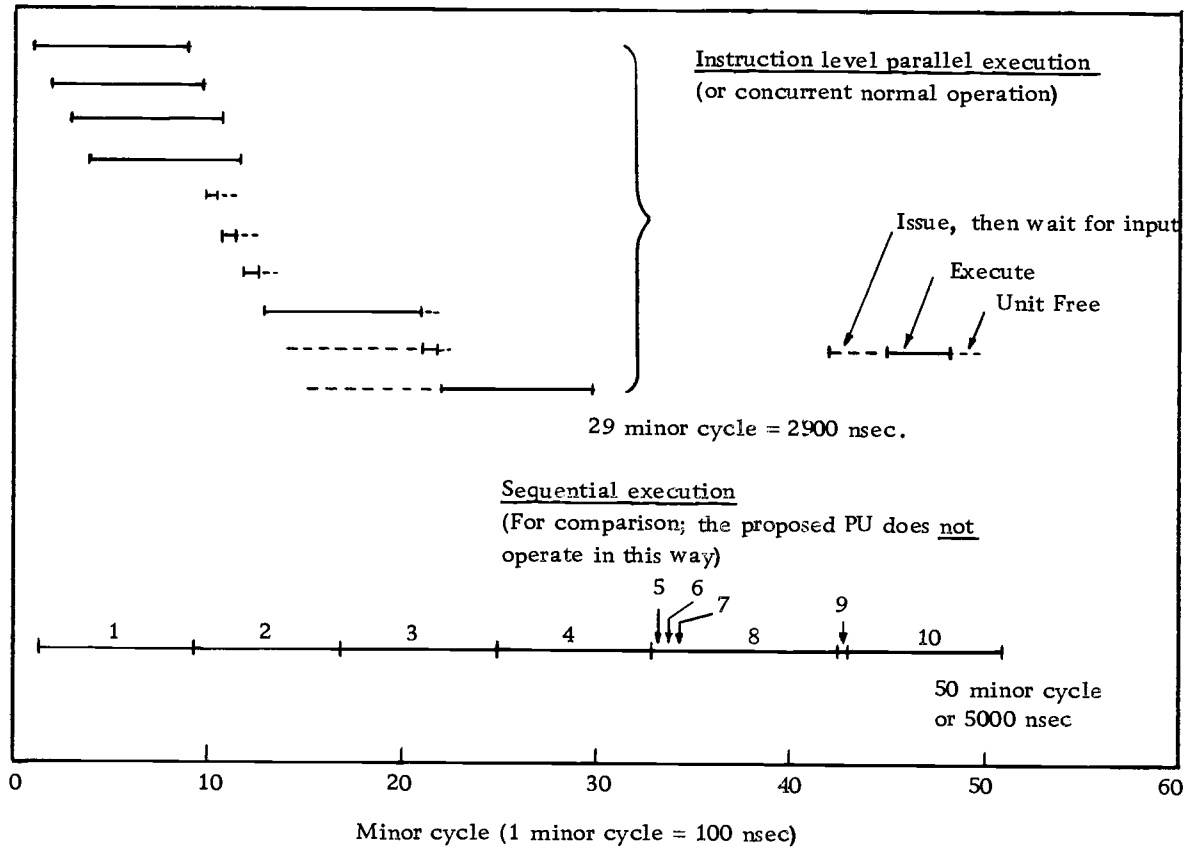


Figure 7.14. Timing analysis of FETCH/STORE MACRO instruction and comparison between Concurrent and Sequential execution.

1. Fetch Loc. K5 $0.5 \rightarrow X_7$
2. STORE MACRO (13532, 21, 21, 1, 1)
3. Fetch Loc. K_6 $1.0 \rightarrow X_6$
4. STORE MACRO (13532, 21, 21, 1, 2)
5. Fetch Loc. K_5 $0.5 \rightarrow X_6$
6. STORE MACRO (13532, 21, 21, 1, 3)
7. Fetch Loc. K_8 $-1.0 \rightarrow X_1$
8. Fetch Loc. M_5 XOLD(7) $\rightarrow X_2$
9. Form -1.0/XOLD(7) $\rightarrow X_7$
10. STORE MACRO (13532, 21, 21, 1, 6)
11. Fetch Loc. M_4 XOLD(6) $\rightarrow X_1$
12. Fetch Loc. M_5 XOLD(7) $\rightarrow X_2$
13. Form XOLD(7)**2 $\rightarrow X_0$
14. Form XOLD(6)/XOLD(7)**2 $\rightarrow X_6$
15. STORE MACRO (13532, 21, 21, 1, 7)
16. Fetch XOLD(1) $\rightarrow X_1$
17. Form XOLD(1)/2 $\rightarrow X_1$, by shift right
18. Fetch XOLD(2) $\rightarrow X_2$
19. Fetch XOLD(3) $\rightarrow X_3$
20. Form XOLD(3)/2 $\rightarrow X_3$ by shift right
21. Fetch XOLD(6) $\rightarrow X_4$
22. Fetch XOLD(7) $\rightarrow X_5$
23. Form XOLD(6)/XOLD(7) $\rightarrow X_6$
24. Form X_6 - XOLD(3)/2 $\rightarrow X_4$
25. Form X_4- XOLD(2) $\rightarrow X_5$
26. Form X_5-XOLD(1)/2 $\rightarrow X_7$
27. STORE MACRO (1352, 2, 21, 21, 1, 8)

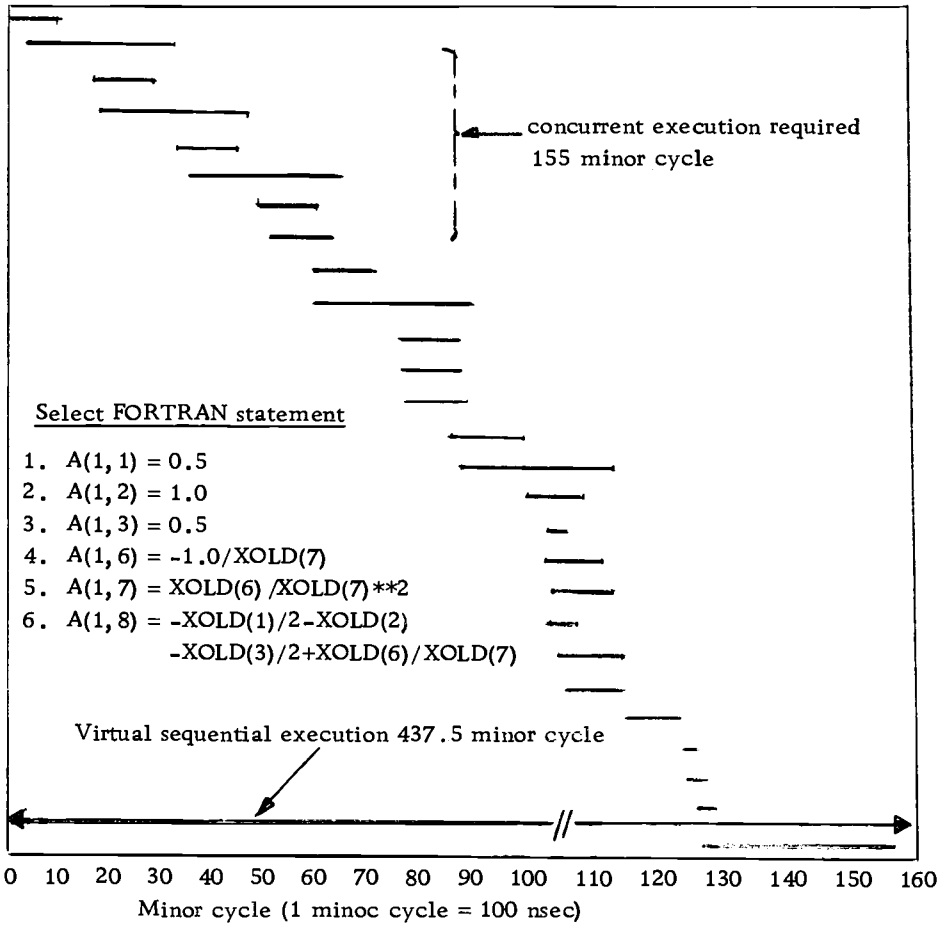


Figure 7.15. Timing analysis of selected FORTRAN statement concurrent v.s. sequential execution.

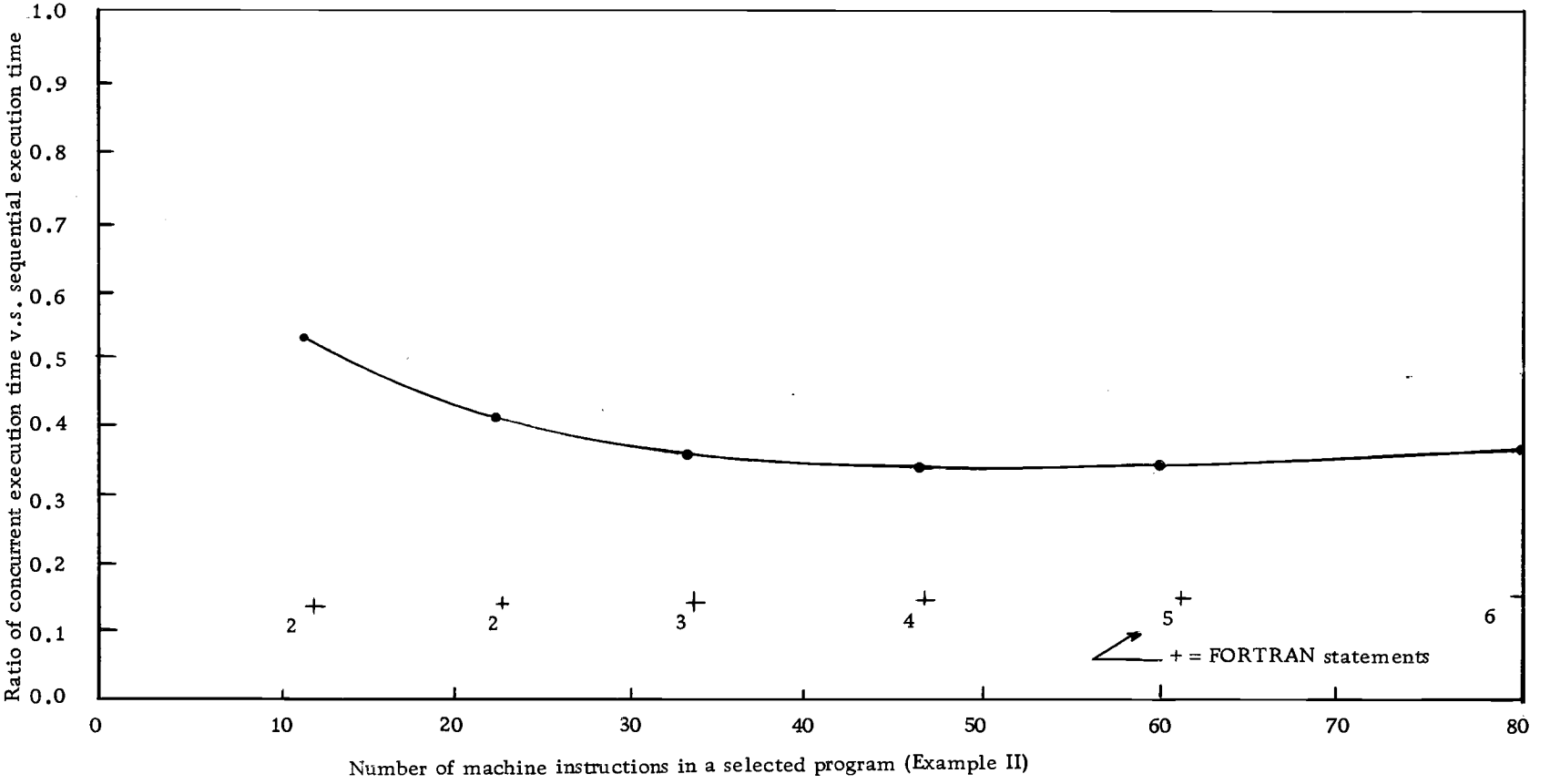


Figure 7.16. The ratio of accumulated concurrent execution time to accumulated sequential execution time as a function of the number of machine instructions in the selected program.

language statements. Figure 7.16 indicates that the increase in execution speed obtained by overlapped execution in the PU approaches a factor of three for this example.

7.3 Summary

From the results of both simulations, the proposed two level parallel architecture shows a large improvement over sequential processing.

The results imply a definite need for an operating system with a parallel task recognition capability. The overhead inherent in such a system should be reduced due to the "canned program" nature of most of its applications.

The instruction overlap simulation indicated that memory interleaving should be included. The results obtained in the simulation implied a gain of 200-300% in execution speed could be expected with the proposed PU architecture.

VIII. CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

A two-level parallel processor architecture has been proposed as a possible system to implement a command and control function for the Royal Thai Air Force. Suggested as an air traffic control and air defense system, the proposed model was demonstrated to possess the desired features; availability, adaptability and expandability. Furthermore, the system provides a high level of efficiency through the use of state-of-the art LSI hardware and parallel processing capability, while meeting the cost constraints of a developing nation.

The proposed architecture is a multiprocessor configuration with a number of identical hardware modules employed in the system. A number of processing units (P) and a number of peripheral processing units (PPU) are connected directly to a number of memory modules (M) through a cross-bar switch. The proposed system has many features which distinguish it from other multiprocessor system.

The concept of allowing master control to float from one PPU to another, along with the modularity will provide a high degree of system availability. Availability is a primary requirement of a command and control data processing system and depends on the inherent fault-tolerant capability.

The configuration assignment unit (CAU) allows a faulty module to be easily taken out from the system for maintenance or repair. This redundancy clearly provides a high degree of reliability.

The system modularity provides both convenience and expansion capability matching to both computing load growth and the available budget. In order to maintain system performance at the highest level, the average memory reference conflict per memory module must be kept as low as possible. Expansion of the system must maintain the processor/module ratio found to be optimum by the memory conflict simulation.

During future expansions, the additional units including the processing module need not necessarily be identical to the existing units. Different module types such as associative memories, associative processors, special purpose processors, etc., can be interfaced to the proposed system through the crossbar communication structure to make the total system match a specific application.

Since the system consists of a number of PU's, PPU's, and M's, it can process jobs or tasks simultaneously. In addition to this first level of parallelism, each PU includes seven general purpose functional units, two multipliers and one divider unit, giving it the capability for concurrent execution of a sequence of machine instructions. The concept of two level parallel processing is the feature of the proposed system architecture.

In the illustrated example, a worst case program was considered. The first level parallelism simulation was performed by analyzing the program graph. The connectivity matrices and a parallel processable tasks table were constructed. The second level parallelism was simulated by long hand calculation. The execution times for both overlap and sequential execution were computed and compared.

The results of both simulations show the proposed models to be superior to both conventional systems and array structured systems for the application programs simulated.

Memory conflict, a possible weak point of the system, is handled by providing memory address hopper registers in each PU. The number of hopper registers was selected based on the memory conflict simulation. A hardware round-robin contention was implemented in the crossbar switch design.

The designed system is shown to have a relatively low cost with a high degree of performance. The basic system components are modular in design and can be built with high quality, low cost commercial LSI hardware. Cost of shipment of electronic hardware components is less than for shipment of a complete computer system. The labor cost for the machine assembly is much lower in Thailand. Since the system cost is based on the above three mentioned costs,

it would seem to be lower than directly purchasing a completed computing system that offers the same capability.

To make this investigation and analysis more feasible, the author recommends that the following features be considered for future research.

1. System software

   a. Suitable high level language. The higher languages such as FORTRAN IV, ALGOL, and others should be considered and compared as to the nature of the real assigned problems.

   b. Operating system functions. In the modern computer system, overall efficiency is dependent not only on its high speed hardware components but also on its operating system functions. Hence, in order to have the desired system run at high efficiency, it is necessary to evaluate and optimize the following system tasks:

      1. Scheduler
      2. Memory managements
      3. System translator and parallel task recognizer

2. Cost Analysis

   Cost analysis is as important a factor in choosing the computer system as is the system capability. It is the

indicator for the final decision as to which to select among a number of computer systems. In practice, both cost behavior and system capability behavior are measured, plotted, and compared. This procedure is quite obvious if the candidate computer systems are already available in the market. But it is not possible for a computer system that is only in a preliminary state of design. Hence, to accomplish a real cost analysis of the proposed computing system, it would be necessary to complete the PU design for the exact cost, and also analyze the cost of expansion versus the increase in capability.

3. Special Purpose Processors

In order to achieve the most suitable computer structure for the Royal Thai Air Force command and control system, special purpose processors such as the Fast Fourier transform processor or autocorrelation processor, should be considered as additions to the system. Again the system cost and capability must be analyzed and compared with those of the proposed system.

4. System Configuration

Also, it is recommended that not only the special purpose processors be considered but also that other promising types of computer system configurations be considered in

analysis and evaluating, such as the associative array processor (STARAN structure) or parallel network of array processor (ILLIAC IV structure).

5. Simulation of Air Traffic Control Algorithms

To accomplish the system cost analysis and evaluation, the specific algorithms for air traffic control used at Bangkok International Airport should be simulated. New algorithms which more efficiently use the proposed structure should be investigated.

BIBLIOGRAPHY

1. Adrion, W. R. Lecture notes from class EE 597X: Computer System Architecture, taught in Department of Electrical and Computer Engineering, Oregon State University, Corvallis 97331. Winter term 1973.

2. Barnes, George H., Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick and Richard A. Stokes. 1968. The ILLIAC IV Computer. IEEE Transactions on Computers, Vol. C-17, No. 8, August 1968. p. 746-757.

3. Bell, C. Gordon, and Peter Freeman. 1972. C. ai-A computer architecture for AI research. AFIPS conference proceedings. vol. 41. Part II. 1972. Fall Joint Computer Conference. Baltimore, MD., Spartan Books, p. 779-790.

4. Bell, C. Gordon and Allen Newell. 1971. Computer Structures: Readings and Examples. McGraw-Hill Book Company, 1971. p. 447-496.

5. Bell, C. Gordon and William A. Wulf. 1972. C.mmp-A multi-mini-processor. AFIPS conference proceedings, vol. 41, Part II. 1972. Fall Joint Computer Conference. Baltimore, MD., Spartan Books, p. 765-777.

6. Berstein, A. J. 1966. Analysis of programs for Parallel processing. IEEE Transaction on Computers, vol. C-15. No. 5, October 1966. p. 757-763.

7. Blumberg, Donald F. 1963. Computer Applications for Industry and the Military - A critical review of the last ten years. AFIPS conference proceedings. vol. 24. 1963. Spring Joint Computer Conference. Baltimore, MD., Spartan Books, p. 179-790.

8. Burnett, G. J., L. J. Koczela and R. A Hokom. 1967. A distributed processing system for general purpose computing. AFIPS Conference Proceedings. vol. 31. 1967. Baltimore, MD., Spartan Books, p. 757-768.

9. Carnahan, B., H. A. Luther and J. O. Wilkes. 1969. Applied Numerical Methods. New York, John Wiley and Sons, 1969. p. 282-328.

10. Coffman, E. G., Jr., and P. J. Denning. 1973. Operating systems theory. Englewood Cliffs, New Jersey. Prentice-Hall, p. 144-186.

11. Cohen, Ellis. 1973. Symmetric multi-mini-processors: a better way to go? Computer Decisions, January 1973. vol. 5. Hayden Publishing Company, Rochelle Park, New Jersey, p. 16-20.

12. Colin, A. J. T. 1971. Introduction to operating systems. MacDonald, London and American Elsevier Inc. New York, p. 18-59.

13. Conway, Melvin E. 1963. A Multiprocessor System Design. AFIPS conference proceedings. vol. 24. Fall Joint Computer Conference. Baltimore, MD., Spartan Books, p. 139-146.

14. Critchlow, A. J. 1963. Generalized Multiprocessing and Multiprogramming Systems. AFIPS conference proceedings. vol. 24, Fall Joint Computer Conference. Baltimore, MD., Spartan Books, p. 107-126.

15. Davis, R L., S. Zucker, and C. M. Campbell. 1972. A building block approach to multiprocessing. AFIPS conference proceedings. vol. 40, 1972. Baltimore, MD., Spartan Books, p. 685-703.

16. Dennis, S. F. and M. G. Smith. 1972. LSI-Implication for future design and architecture. AFIPS conference proceedings. vol. 40. Spring Joint Computer Conference. Baltimore, MD., Spartan Books, p. 343-351.

17. Enslow, Philip H., Jr. 1974. Multiprocessors and Parallel Processing. New York, John Wiley and Sons.

18. Erwin, F. D. and E. Bersoff. 1969. Modular computer architecture strategy for long term missions. AFIPS conference proceedings. vol. 35. Baltimore, MD., Spartan Books, p. 337-345.

19. Finch, T. R. 1967. Main fram memory technology. AFIPS conference proceedings. Fall Joint Computer Conference. vol. 31. Baltimore, MD., Spartan Books, p. 593-598.

20. Flores, Ivan. 1969. Computer Organization, Englewood Cliffs. Prentice-Hall, p. 49-70.

21. Harold, Lorin. 1972. Parallelism in hardware and software: Real and Apparent Concurrency. Englewood Cliffs, New Jersey, Prentice-Hall.

22. Hobbs, L. C., D. J. Theis, Joel Trimble, Harold Titus and Ivar Highberg. 1970. Parallel Processor Systems, Technologies and Applications. New York, Spartan Books.

23. Husson, Samir S. 1970. Microprogramming: Principles and practices. Englewood Cliffs, Prentice-Hall, p. 22-78.

24. International Business Machines. 1965. General Purpose System Simulator III Introduction. White Plains, New York, Technical Publications Department.

25. Katzan, Harry, Jr. 1970. Operating systems architecture. AFIPS conference proceedings. Spring Joint Computer Conference. vol. 36, Baltimore, MD., Spartan Books, p. 109-118.

26. Knight, E. Kenneth. 1966. Changes in computer performance. Datamation, vol. 12, No. 9. September 1966. p. 40-54.

27. Lass, Stanley E. 1968. A fourth-generation computer organization. AFIPS conference proceedings. Spring Joint Conference, vol. 32. Baltimore, MD., Spartan Books, p. 435-441.

28. Lewin, Douglas. 1972. Theory and Design of Digital Computers. New York, John Wiley and Sons, p. 306-333.

29. McMillan, Claude, R. F. Gonzalez, Thomas J. Schriber. 1973. Systems Analysis a Computer Approach to Decision Models. Homewood, Illinois, Richard D. Irwin, p. 280-356.

30. Ramamoorthy, C. V. and M. J. Gonzalez. 1969. A survey of techniques for recognizing parallel processable in stream in computer programs. AFIPS conference proceedings. vol. 35. Fall Joint Computer Conference. Baltimore, MD., Spartan Books, p. 1-15.

31. Riley, Wallace B. 1971. Electronic Computer Memory Technology. New York, McGraw-Hill Book Company. p. 61-192.

32. Rosene, A. Frederick. 1967. Memory Allocation for Multiprocessors. IEEE Transactions on Electronic Computers. vol. EC-16, No. 5. October 1967. The Institute of Electrical and Electronics Engineers, New York, NY 10017. p. 659-665.

33. Sharpe, William F. 1969. The Economic of Computers. New York, RAND Corporation, p. 211-278.

34. Slotnick, Daniel L. 1967. Unconventional systems. AFIPS conference proceedings. vol. 30. Spring Joint Computer Conference. Baltimore, MD., Spartan Books. p. 467-469.

35. Slotnick, Daniel L., W. Carl Borch and Robert C. McReynolds. 1962. The SOLOMON Computer. AFIPS conference proceedings. vol. 22. Fall Joint Computer Conference. Baltimore, MD., Spartan Books, p. 97-107.

36. Soucek, Branko. 1972. Minicomputers in data processing and simulation. New York. John Wiley and Sons, p. 151-291.

37. Thornton, J. E. 1970. Design of a computer the Control Data 6600. Glenview, Illinois, Scott, Foresman and Company.

38. Wallace, V. L. and D. L. Mason. 1969. Degree of multiprogramming in Page-on-Demand Systems. Communications of the Association for Computing Machinery, vol. 12, No. 6, June, 1969. p. 305-318. (Communications of the ACM).

39. Wang, Gary Y. 1970. System Design of a Multiprocessor Organization. University of Texas at Austin.

40. West, P. George. 1967. The best approach to a large computing capability. AFIPS conference proceedings. vol. 30, 1967. Spring Joint Computer Conference. Baltimore, MD., Spartan Books, p. 467-469.

41. Wilkes, M. V. 1965. Slave Memories and Dynamic Storage Allocation. IEEE Transactions on Electronic Computers Vol. EC-14, No. 2. April 1965. The Institute of Electrical and Electronics Engineers, New York, NY 10017, p. 270-271.

APPENDIX A

Instruction Set Summary

The table which follows is a complete preliminary list of PU instructions for the RTAF Command and Control System. Both octal and binary code equivalents for the function fields are given. Estimated execution times are also included.

Table A.1. Proposed PU instruction and execution time.

Octal code	Binary code	Mnemonic code	Description	Execution time n sec
			Add Unit (Floating Point)	
1 0	0 0 1 0 0 0	ADDFLS	Floating sum of $x_j + x_k \rightarrow x_i$	400
1 1	0 0 1 0 0 1	ADDFLD	Floating difference of $x_j - x_k \rightarrow x_i$	400
1 2	0 0 1 0 1 0	ADDDPS	Floating DP sum of $x_j + x_k \rightarrow x_i$	400
1 3	0 0 1 0 1 1	ADDDPD	Floating DP difference of $x_j + x_k \rightarrow x_i$	400
1 4	0 0 1 1 0 0	ADDRFS	Round floating sum of x_j and $x_k \rightarrow x_i$	400
1 5	0 0 1 1 0 1	ADDRFD	Round floating difference of $x_i + x_k \rightarrow x_i$	400
			Long Add Unit	
1 6	0 0 1 1 1 0	LADDIS	Integer sum of $x_j + x_k \rightarrow x_i$	400
1 7	0 0 1 1 1 1	OADDID	Integer difference of $x_i + x_k \rightarrow x_i$	400
			Divide Unit	
2 0	0 1 0 0 0 0	DIVFL	Floating divide $x_j \div x_k \rightarrow x_i$	1000
2 7	0 1 0 0 0 1	RFDIV	Round floating divide $x_j \div x_k \rightarrow x_i$	1100
			Multiply Unit	
3 0	0 1 1 0 0 0	FMUL	Floating product of $x_j * x_k \rightarrow x_i$	800
3 1	0 1 1 0 0 1	RFMUL	Round floating product of $x_i * x_k \rightarrow x_i$	900
3 2	0 1 1 0 1 0	DPFMUL	Floating DP of $x_j * x_k \rightarrow x_i$	800
			Boolean Unit	
4 0	1 0 0 0 0 0	TRANS	Transmit $x_j \rightarrow x_i$	50
4 1	1 0 0 0 0 1	LGPX	Logical product $x_j \cdot x_k \rightarrow x_i$	50
4 2	1 0 0 0 1 0	LGSX	Logical sum $x_j + x_k \rightarrow x_i$	50
4 3	1 0 0 1 0 1	LGDX	Logical difference $x_j \sim x_k \rightarrow x_i$	50
4 4	1 0 0 1 0 0	TRANSCP	TX_k comp $\rightarrow$ xi	50
4 5	1 0 0 1 0 1	LGPC	$x_j \cdot x_k$ comp $\rightarrow x_i$	50
4 6	1 0 0 1 1 0	LGSC	$x_j + x_k$ comp $\rightarrow x_i$	50
4 7	1 0 0 1 1 1	LGDC	$x_j \sim x_k$ comp $\rightarrow x_i$	50

Table A.1. (Continued).

Octal code	Binary code	Mnemonic code	Description	Execution time n sec
			Increment Unit	
5 0	1 0 1 0 0 0	SUMAKA	$= A_j + K \rightarrow A_i$	50
5 1	1 0 1 0 0 1	SUMBKA	$= B_j + K \rightarrow A_i$	50
5 2	1 0 1 0 1 0	SUMXKA	$= x_j + K \rightarrow A_i$	50
5 3	1 0 1 0 1 1	SUMXBA	$= x_j + B_k \rightarrow A_i$	50
5 4	1 0 1 1 0 0	SUMABA	$= A_j + B_k \rightarrow A_i$	50
5 5	1 0 1 1 0 1	DIFABA	$= A_j - B_k \rightarrow A_i$	50
5 6	1 0 1 1 1 0	SUMBBZ	$= B_j = B_k \rightarrow Z_i$	50
5 7	1 0 1 1 1 1	DIFBBZ	$= B_j - B_k \rightarrow Z_i$	50
6 0	1 1 0 0 0 0	SUMAKB	$= A_j + K \rightarrow B_i$	50
6 1	1 1 0 0 0 1	SUMBKB	$= B_j + K \rightarrow B_i$	50
6 2	1 1 0 0 1 0	SUMXKB	$= x_j + K \rightarrow B_i$	50
6 3	1 1 0 0 1 1	SUMXBB	$= x_j + B_k \rightarrow B_i$	50
6 4	1 1 0 1 0 0	SUMABB	$= A_j + B_k \rightarrow B_i$	50
6 5	1 1 0 1 0 1	DIFABB	$= A_j - B_k \rightarrow B_i$	50
6 6	1 1 0 1 1 0	SUMBBB	$= B_j + B_k \rightarrow B_i$	50
6 7	1 1 0 1 1 1	DIFBBB	$= B_j - B_k \rightarrow B_i$	50
7 0	1 1 1 0 0 0	SUMAKX	$= A_j + K \rightarrow x_i$	50
7 1	1 1 1 0 0 1	SUMBKX	$= B_j + K \rightarrow x_1$	50
7 2	1 1 1 0 1 0	SUMXKX	$= x_j + K \rightarrow x_i$	50
7 3	1 1 1 0 1 1	SUMXBX	$= x_j + B_k \rightarrow x_i$	50
7 4	1 1 1 1 0 0	SUMABX	$= A_j + B_k \rightarrow x_i$	50
7 5	1 1 1 1 0 1	DIFABX	$= A_j + B_k \rightarrow x_i$	50
7 6	1 1 1 1 1 0	SUMBBX	$= B_j + B_k \rightarrow x_i$	50
7 7	1 1 1 1 1 1	DIFBBX	$= B_j - B_k \rightarrow x_i$	50
			Shift Unit	
2 3	0 1 0 0 1 1	SHIFL	Shift x_i left jk places	50
2 4	0 1 0 1 0 0	SHIFR	Shift x_i right jk places	50
2 5	0 1 0 1 0 1	SHIFNL	Shift x_i nominally left B_j places	50
2 6	0 1 0 1 1 0	SHIFNR	Shift x_i nominally right B_j places	50
2 7	0 1 0 1 1 1	NORMAL	Normalize x_k in x_i and B_j	50

Table A.1. (Continued).

Octal code	Binary code	Mnemonic code	Description	Execution time n sec
3 3 0	1 1 1 1 0	ROUNOR	Round and normalize x_k in x_i and B_j	60
3 4 0	1 1 1 0 0	UNPAC	Unpack x_k to x_i and B_j	50
3 5 0	1 1 1 0 1	PACXXB	Pack x_i from x_k and B_j	50
3 6 0	1 1 1 1 0	FORM	Form jk mask in x_i	50
			Branch Unit	
0 0		STOP	Stop	
0 1		RIJK	Return jump to k	
0 2			Go to $K = B_i$ (Note 1)	100
0 3 0			Go to K if x_j = zero	100
0 3 1			Go to K if $x_j \neq$ zero	100
0 3 2			Go to K if x_j = positive	100
0 3 3			Go to K if x_j = negative	100
0 3 7			Go to K if x_j = in given range	100
0 3 5			Go to K if x_j = is out of range	100
0 3 6			Go to K if x_j = is definite	100
0 3 7			Go to K if x_j = is indefinite	100
0 4			Go to K if $B_i = B_j$	100
0 5			Go to K if $B_i \neq B_j$	100
0 6			Go to K if $B_i \geq B_j$	100
0 7			Go to K if $B_i < B_j$	100

Binary code of this part is not shown here

Mnemonic code of this part is not shown here

Note 2

Note 1

Note:

1. Go to $k + B_i$ and go to k if B_i .... tests made in increment unit, in FN
2. Go to k if x_j .... tests made in long add unit in FN
3. * means add 600 nsec. (6 minor cycles) to branch time for a branch to an instruction which is out of the instruction stock (assume no memory conflict).
4. * is also mean duplexed units-instruction go to free unit
5. Comp = Complement
6. DP = Double Precision

APPENDIX B

Detailed Timing for the Instruction Level Simulation

Tables B. 1-B. 3 list the machine code and a complete timing study for the STORE MACRO for the following conditions

1. No memory interleaving, average memory conflict = 3
2. No memory interleaving, average memory conflict = 0
3. Memory interleaving, average memory conflict = 0

Table B. 4 is a machine code listing and timing analysis for the five FORTRAN statements included in the simulation.

Table B.1. List of the machine code and timing study for STORE MACRO case 1.

				Concurrent Execution						
	Example of STORE MACRO execution simulation			ISSUE	START	RESULT	UNIT FREE	FETCH	STORE	Sequential execution (mic)
	Case 1									
	Assumptions:									
	(1) Non-overlap and no interleave are provided a store reference requires 1000 nsec.									
	(2) Average memory conflicts per access = 3(P = 16, M = 16)									
	(3) Both 15-bit and 30-bits instruction require one minor cycle to issue									
	(4) A functional unit is free one minor cycle after the result is placed in a register									
	(5) STORE MACRO execution time is equal to FETCH MACRO execution time									
	STORE MACRO (13532, 21, 21, 1, 2)									
1	A1 = A1 + K1	Fetch BASE	= X1 = 13532	1	1	1.5	2.5	31	-	30.0
2	A2 = A2 + K2	Fetch I	= X2 = I = 1	2	31	31.5	32.5	61	-	30.0
3	A3 = A3 + K3	Fetch J	= X3 = J = 2	3	61	61.5	62.5	91	-	30.0
4	A4 = A4 + K4	Fetch D1M1	= X4 = D1M1 = 21	4	91	91.5	92.5	121	-	30.0
5	X5 = X1 + X2	Form BASE + I	= X5 = 13534	61	61	61.5	62.5	-	-	0.5
6	B1 = BO + (-1)	Set B1 = -1	= B1 = -1	62	62	62.5	63.5	-	-	0.5
7	X1 = X3 + B1	Form J-1	= X1 = 1	91	91	91.5	92.5	-	-	0.5
8	X3 = X1 * X4	Form (J - 1) * D1M1	= X3 = 21	121	121	129	130	-	-	8.0

Table B.2. List of the machine code and timing study for STORE MACRO case 2.

			Example of STORE MACRO execution simulation	Concurrent execution						Sequential execution (mic)
				ISSUE	START	RESULT	UNIT FREE	FETCH	STORE	
9	X6	= X5 + X3	Form BASE + I + (J-1)*D1M1 = X6 = 13554	122	129	129.5	130.5	-	-	0.5
10	X7	= BO + X6	Set A_7 = 13554 and store content of X_7 at location 13554	123	130	130.5	131.5	-	160	30.0
			Total execution time ——						159	160
			Assume location K7 stored 13532, K2 stored 1							
			K3 stored 2, K4 stored 21							
			Case 2							
			Assumptions:							
			(1) Non-overlap and no interleave are provided: each storage reference requires 1000 nsec							
			(2) No memory conflicts							
			(3) Both 15-bit instruction and 30-bit instruction require one minor cycle to issue							
			(4) A functional unit is free one minor cycle after the the result is placed in a register							
			(5) STORE MACRO execution time is equal to FETCH MACRO execution time							
			STORE MACRO (13532, 21, 21, 1, 2)							
1	A1	= A1 + K1	Fetch BASE = X1 = 13532	1	1	1.5	2.5	11	-	10.0
2	A2	= A2 + K2	Fetch 1 = X2 = I = 1	2	11	11.5	12.5	21	-	10.0

Table B.3. List of the machine code and timing study for STORE MACRO case 3.

			Example of STORE MACRO execution simulation		Concurrent Execution						Sequential execution (mic)
					ISSUE	START	RESULT	UNIT FREE	FETCH	STORE	
3	A3	= A3 + K3	Fetch J	= X3 = J = 2	3	21	21.5	22.5	31	-	10.0
4	A4	= A4 + K4	Fetch D1M1	= X4= D1M1 = 21	4	31	31.5	32.5	41	-	10.0
5	X5	= X2 + X2	Form BASE + I	= X5 = 13534	21	21	21.5	22.5	-	-	0.5
6	B1	= BO + (-1)	Set B1 = -1	= B1 = -1	22	22	22.5	23.5	-	-	0.5
7	X1	= X3 + B1	Form J - 1	= X1 = 1	31	31	31.5	32.5	-	-	0.5
8	X3	= X1 * X4	Form (J - 1)*D1M1	= X3 = 21	41	41	49	50	-	-	8.0
9	X6	= X5 + X3	Form BASE+I+(J-1)*D1M1	= X6 = 13554	42	49	49.5	50.5	-	-	0.5
10	A7	= BO + X6	Set A7 = 13554 and store contents of X7 at location 13554		43	50	50.5	51.5	-	60	10.0
				Total execution time ⟶						59	60

Assume location K1 stored 13532, K2 stored 1,

K3 stored 2, K4 stored 21

Case 3

Assumptions:

(1) Consecutive instruction words from storage require a minimum of eight minor cycles (memory interleaved)

(2) A functional unit is free one minor cycle after the result is placed in a register.

Table B.3. List of the machine code and timing study for STORE MACRO case 3 (cont.).

			Example of STORE MACRO execution simulation		Concurrent execution						Sequential execution (mic)
					ISSUE	START	RESULT	UNIT FREE	FETCH	STORE	
			(3) Every instruction (both 15-bit and 30-bits) requires one minor cycle to issue (4) A functional unit is free one minor cycle after the result is placed in a register (5) STORE MACRO execution time is equal to FETCH MACRO execution time								
			STORE MACRO (13532, 21, 21, 1, 2)								
1	A1	= A1 + K1	Fetch BASE	= X1 = 13532	1	1	1.5	2.5	9	-	8.0
2	A2	= A2 + K2	Fetch I	= X2 = I = 1	2	2	2.5	3.5	10	-	8.0
3	A3	= A3 + K3	Fetch J	= X3 = J = 2	3	3	3.5	4.5	11	-	8.0
4	A4	= A4 + K4	Fetch D1M1	= X4 = D1M1 = 21	4	4	4.5	5.5	12	-	8.0
5	X5	= X1 + X2	Form BASE + I	= X5 = 13533	10	10	10.5	11.5	-	-	0.5
6	B1	= BO + (-1)	Set B1 = -1	= B1 = -1	11	11	11.5	12.5	-	-	0.5
7	X1	= X3 + B1	Form J - 1	= X1 = 1	12	12	12.5	13.5	-	-	0.5
8	X3	= X1 * X4	Form (J - 1)*D1M1	= X3 = 21	13	13	21	22	-	-	8.0
9	X6	= X5 + X3	Form BASE _ I + (J-1)*D1M1	= X6 = 13554	14	21	21.5	22.5	-	-	0.5
10	A7	= BO + X6	Set A7 = 13554 and store content of X7 at location 13554		15	22	22.5	23.5	-	30	8.0
				Total execution time ——						29	50.0
			Assume Location K1 stored 13532, K2 stored 1,								
			K3 stored 2, K4 stored 21								

Table B.4. List of the machine code and timing study for FORTRAN statement execution.

FORTRAN statement number	Example of concurrent execution simulation	Concurrent execution						Sequential execution (mic)
		ISSUE	START	RESULT	UNIT FREE	FETCH	STORE	
	Example of selected FORTRAN statement execution							
	BASE = base address of matrix A D1M1 = 1" dimension of A, No. of rows D1M2 = 2" dimension of A, No. of column							
	BASE = 13532 DIMA (21, 21)							
1	A(1, 1) = 0.5							
	$A_7 = BO + K_5$ 0.5 → X_7	1	1	1.5	2.5	9		8
	STORE MACRO (13532, 21, 21, 1, 1) Store X_7 → Loc. A(1, 1)	16	23	23.5	24.5	-	31	50
	Intermediate sum after statement No. 1 is executed →						30	58
2	A(1, 2) = 1.0							
	$A_6 = Bo + K_6$ 1.0 → X_6	17	17	17.5	18.5	25	-	8
	STORE MACRO (13532, 21, 21, 1, 2) Store X_6 → Loc. A(1, 2)	32	39	39.5	40.5	-	47	50
	Intermediate sum after statement No. 1 - 2 are executed →						46	116
3	A(1, 3) = 0.5							
	$A_7 = BO + K_5$ 0.5 → X_7	33	33	33.5	34.5	47	-	8
	STORE MACRO (13532, 21, 21, 1, 3) Store X_7 → Loc. A(1, 3)	48	55	55.5	56.5	-	63	50
	Intermediate sum after statement No. 1-3 are executed →						62	174

Table B.4. List of the machine code and timing study for FORTRAN statement execution (cont.).

FORTRAN statement number	Example of concurrent execution simulation		Concurrent execution						Sequential execution (mic)
			ISSUE	START	RESULT	UNIT FREE	FETCH	STORE	
4	A(1, 6) = -1.0/XOLD(7)								
	A_1 = BO + K_8	-1.0 → X_1	49	49	49.5	50.5	57	-	8
	A_2 = BO + M_5	XOLD(7) → X_2	50	50	50.5	51.5	58	-	8
	X_7 = X_1/X_2	-1.0/XOLD(7) → X_7	58	58	68	69	-	-	10
	STORE MACRO	(13532, 21, 21, 1, 6) Store X_7 → Loc. A(1, 6)	73	80	80.5	81.5	-	88	50
		Intermediate sum after statement No. 1-4 are executed →						87	250
5	=	A(1, 7) = XOLD(6)/XOLD(7)**2							
	A_1 = BO + M_4	XOLD(6) → X_1	74	74	74.5	75.5	82	-	8.0
	A_2 = BO + M_5	XOLD(7) → X_2	75	75	75.5	76.5	83	-	8.0
	X_3 = X_2 * X_2	X^2 → X_3	76	76	84	85	-	-	8.0
	X_4 = X_1/X_3	X_1/X_2^2 → X_7	84	84	94.	95.	-	-	10.0
	STORE MACRO	(13532, 21, 21, 1, 7) Store X_7 at Loc. A(1, 7)	99	106	106.5	107.5	-	114	50.0
		Intermediate sum after statement No. 1-5 are executed →						113	334
6	A(1, 8) = - XOLD(1)/2. -XOLD(2) - XOLD(3)/2. + XOLD(6)/XOLD(7)								
	A_1 = BO + M_1	XOLD(1) → X_1	100	100	100.5	101.5	108	-	8.0
	SHIFT X_1 RIGHT 1 place	XOLD(1)/2 → X_1	101	101	101.5	102.5	-	-	0.5
	A_2 = BO + M_2	XOLD(2) → X_2	102	102	102.5	103.5	110	-	8.0
	A_3 = BO + M_3	XOLD(3) → X_3	103	103	103.5	104.5	111	-	8.0

Table B.4. List of the machine code and timing study for FORTRAN statement execution (cont.).

FORTRAN statement number	Example of concurrent execution simulation		Concurrent execution						Sequential execution (mic)
			ISSUE	START	RESULT	UNIT FREE	FETCH	STORE	
	Shift X_3 Right 1 place	XOLD(3)/2 → X_3	104	104	104.5	105.5	-	-	0.5
	$A_4 = BO + M_4$	XOLD(6) → X_4	105	105	105.5	106.5	113	-	8.0
	$A_5 = BO + M_5$	XOLD(7) → X_5	106	106	106.5	107.5	114	-	8.0
	$X_6 = X_4/X_5$	XOLD(6)/XOLD(7) → X_6	114	114	124	125	-	-	10.0
	$X_4 = X_6-X_3$	XOLD(6)/XOLD(7) - XOLD(3)/2 → X_4	124	124	124.5	125.5	-	-	0.5
	$X_5 = X_4-X_2$	XOLD(6)/XOLD(7) - XOLD(3)/2 - XOLD(2) → X_5	125	125	125.5	126.5	-	-	0.5
	$X_7 = X_5-X_1$	XOLD(6)/XOLD(7) - XOLD(3)/2 - XOLD(2) -XOLD(1)/2 → X_7	126	126	126.5	127.5	-	-	0.3
	STORE MACRO (13532, 21, 21, 1, 8)	Stored X_7 in Loc. A(1, 8)	141	148	148.5	149.5	-	156	50.0
		Intermediate sum after statement No. 1-6 are executed →						155	437.5

APPENDIX C

A Tabulation of Arithmetic and Logical Function Type Frequencies For Various Common Programs

In order to determine which functional units would best be chosen for the processing unit, a study of many common programs was undertaken. Table C. 1 lists for each program, algorithm or routine a rating (A, B, or C) which indicates the relative frequency of occurence of various arithmetic and logical operations.

For example, LaGrangian interpolation algorithms require a large number of additions and multiplications (both rated A), but relatively fewer shift operations (rated B).

Table C.1. The investigation of Logical and Arithmetic function - references for G.P. applicatis problems.

Applications	Add & Subtract Floating Point Reference	Multiplication Floating Point Reference	Divide Floating Point Reference	Fixed Add Reference	Increment Unit Reference	Boolean Function Unit Reference	Shift Reference	Branch Reference
Engineers and Scientists								
1. Interpolation and Approximation								
Interpolation with finite divided differences	A	A	A	B	A	A	B	B
Longrangian interpolation	A	A	A	A	A	A	B	B
Chebyshev economization	A	A	A	A	A	A	B	B
2. Numerical Integration								
Integration by Simpson's Rule	A	A	A	A	A	A	B	B
Fourier coefficients using Romberg integration	A	A	A	B	A	A	B	B
Gauss-Legerdre Quadrakue	A	A	A	B	A	A	-	B
Velocity distribution using	A	A	A	B	A	A	B	B
3. Solution of Equations								
Graeffe's root-squaring Method mechanical vibration frequencies	A	A	A	B	A	A	-	B
Interactive factorization of polynomials	A	A	A	B	A	A	-	B
Solution of an equation of state using Newton's method	A	A	A	B	A	A	-	B
Grauss-Lagondre base points and weight factors by the half interval method	A	A	A	B	A	A	-	B
4. Matrices and Related Topics								
Matrix operations	A	A	A	A	A	A	B	B
The power method (method for determining Eigenvalue and Engenvectors	A	A	A	-	A	A	B	B
Rutishauer's method (to find (x) and R	A	A	A	-	A	A	B	B

Table C.1. Continued.

Applications	Add & Subtract Floating Point Reference	Multiplication Floating Point Reference	Divide Floating Point Reference	Fixed Add Reference	Increment Unit Reference	Boolean Function Unit Reference	Shift Reference	Branch Reference
Jacobi's method (of finding (x) and R.	A	A	A	-	A	A	B	B
5. System of Equations								
Gauss-Jordan reduction (voltage and current E.E.network)	A	A	A	A	A	A	B	B
Calculation of the inverse matrix using the maximum Pivet strategy (member forces in a plane truss)	A	A	A	-	A	A	B	B
Gauss-Seidel method	A	A	A	-	A	A	B	B
Flow in a pipe network (successive-substitution method)	A	A	A	-	B	B	B	B
Chemical equilibrium (Newton-Rophson method)	A	A	A	-	B	B	B	B
6. The Approximation of the Solution of Ordinary Differential Equations								
Taylor's expansion approach	A	A	A	-	A	A	B	B
Euler's method	A	A	A	-	A	A	B	B
Fourth-order Ruange-Kunge method (transient behavior of a resonant circuit)	A	A	A	-	A	A	B	B
Hamming's method	A	A	A	-	A	A	B	B
Bourdary-value problem in fluid mechanics	A	A	A	-	A	A	B	B
7. Approximation of the Solution of Partial Differential Equations								
Unsteady-state heat condition in an infinite, parallel-sided slab (Explicit method)	A	A	A	-	A	B	B	B
Unsteady-state heat conduction in an infinite, parallel-sided slab (Implicit method)	A	A	A	-	A	B	B	B
Unsteady-state heat conduction in a long bar of square cross-section (Implicit alternating direct method)	A	A	A	-	A	B	B	B

Table C.1. Continued.

Applications	Add & Subtract Floating Point Reference	Multiplication Floating Point Reference	Divide Floating Point Reference	Fixed Add Reference	Increment Unit Reference	Boolean Function Unit Reference	Shift Reference	Branch Reference
Unsteady-state heat conduction a solidifying alloy	A	A	A	-	A	B	B	B
Natural convection at a heated vertical plate	A	A	A	-	A	B	B	B
Steady-state heat conduction in a square plate	A	-	A	-	A	B	B	B
Deflection of a loaded plate	A	A	A	-	A	B	B	B
Torsion with curved boundary	A	A	A	-	A	B	B	B
Unsteady conduction between cylinders (characteristic value problem)	A	A	A	-	A	B	B	B
8. Statistical Methods								
Distribution of points in bride hand	A	A	A	-	A	B	B	B
Poisson distribution random number generator	A	A	A	-	A	B	A	B
Tabulation of the standardized normal distribution	A	A	A	-	A	B	B	B
X^2 test for goodness of fit	A	A	A	-	A	B	B	B
Polynomial regression with plotting	A	A	A	B	A	B	B	B
Correlation theory (air traffic control, radar tracking, weather forcast)	A	A	A	-	A	A	B	B
Operation Research								
9. Deterministic Simulation								
Missile attack problem	A	A	A	-	A	A	B	B
Motion of a rocket	A	A	A	-	A	A	B	B
10. Probabilistic Simulation								
Queuing problem	A	A	A	-	A	A	B	B
Neutron diffusion	A	A	A	-	A	A	B	B
Airplane traffic flow	A	A	A	-	A	A	B	B
Processing computer program	A	A	A	-	A	A	B	B

Table C.1. Continued.

Applications	Add & Subtract Floating Point Reference	Multiplication Floating Point Reference	Divide Floating Point Reference	Fixed Add Reference	Increment Unit Reference	Boolean Function Unit Reference	Shift Reference	Branch Reference
11. Linear Programming								
Linear programming problem (Simplex method)	A	A	A	-	A	A	B	B
Data Processing								
12. Basic Data Processing								
Accounting (compound interest, discounted value)	A	A	A	-	A	A	B	B
A payroll problem	B	B	B	-	A	A	-	B
An inventory control problem	A	-	-	-	A	A	-	B
13. Information Retrieval								
Sequential search	-	-	-	-	A	A	-	B
Binary search	A	-	A	-	A	A	A	B
Probabilistic search	A	A	A	-	A	A	-	B
Hash-search	A	A	A	-	A	A	-	B
14. Artificial Intelligent								
Picture processing	A	-	-	A	A	A	B	B
Pattern recognition	A	A	A	A	B	B	B	B
	54^{A} 1^{B} 0x	50^{A} 1^{B} 5x0	52^{A} 1^{B} 3x0	7^{A} 9^{B} 40x0	54^{A} 2^{B} 0x0	39^{A} 17^{B} x0	2^{A} 49^{B} 5x0	0^{A} 56^{B} 0x0
Divide by	56	56	56	56	56	56	56	56
Average	0.965^{A} +0	0.895^{A} +0	0.93^{A} +0	0.0125^{A} +	0.915^{A} +0...	0.69^{A} $+0.3^{B}$	0.875^{B}	n
Approximation	A	A	A	B	A	A	B	B

Note:

A = a lot of references.

B = rare number of references.

APPENDIX D

Summary of Existing Command and Control Systems

Table D. 1 lists manufacturers, types, applications and architectures of the currently available commercial command and control systems.

Table D.1. Summary of existing command and control system and air traffic control computer system.

Computer system	Type of system	Major application	System resource	
			Number of processor	Number memory module
D-825 Burrough	Multicomputer system conventional hardware processor	Command and control system (USAF)	P = 4	M = 16 4096, 48 bit/word shared memory
PEPE with IBM 360/95 as host computer	Multicomputer system associate processor	Air-traffic control, missile tracking (object tracking: real time radar processing)	PE = 16	M = 16 512 words/M 32 bit/word
ILLIAC IV with B6500 as host computer	Array-network computer conventional type processor	Can handle object tracking: real time radar processing	PE = 64	M = 64 2048 words/PEM 64 bit/word, 0.240 μsec
UNIVAC 1206 (at Ascension Island AMR station 12)	Conventional type computer	Object tracking by radar. (Military real time computer system)	P = 1	16384 words 30 bit/word 8 μsec cycle time
Sigma 5 computer	General purpose computer, medium size moduler type computer	Air-traffic control system	P = 1	M = 3 16k/bank 32 bits/word plus parity, 0.85 μsec cycle time

Table D.1. Continued.

Computer system	Type of system	Major application	System resource	
			Number of processor	Number of memory module
STARAN Goodyear Aerospace Corporation	Associative array processor	Air traffic control and object tracking: real time radar processing	PE = 256	Bulk core 10384 32768 words cycle time 1/μsec
T1ASC Toya Instrument Advance Scientific Computer	General purpose multi-processor Pipe-line processor	Experiment computer: Advance Scientific computer	P = 1	M = 8 128 k/module