MIT/LCS/TM-93

A LOGIC DESIGN FOR THE CELL BLOCK OF A DATA-FLOW PROCESSOR

KATSUHIKO AMIKURA

DECEMBER 1977

MIT/LCS/TM-93

A LOGIC DESIGN FOR THE CELL BLOCK OF A DATA-FLOW PROCESSOR

Katsuhiko Amikura

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

A LOGIC DESIGN FOR THE CELL BLOCK OF A DATA-FLOW PROCESSOR

by

KATSUHIKO AMIKURA

Submitted to the Department of Electrical Engineering and Computer Science on August 26, 1977 in partial fulfillment of the requirements for the degree of Master of Science

ABSTRACT

Recently studies on parallel computation architecture have yielded a new type of computer architecture known as the data-flow processor. As part of the effort in realizing the data-flow processor, a logic design for the Cell Block of the basic data-flow processor is proposed in this thesis. The resulting design has a modular structure which is derived from a top-down decomposition of the specification given in an Aechitecutere Description Language. The desired speed of operation of the Cell Block is obtained by exploiting the parallelism inherent in its operation. The logic design is carried out using electronic devices available commercially today, but is based on an asynchronous communication protocol.

Thesis Supervisor: Jack B. Dennis
Title: Professor of Computer Science and Engineering

ACKNOWLEDGEMENT

The author wishes to express deep gratitude to and appreciation of Professor Jack B. Dennis, for his patient encouragement and suggestions: His ideas have been most stimulating, often leading to new possibilities and break-throughs.

The author also thanks Mr. Ken Weng, for sharing time and energy in discussions, and above all, for his help in writing this thesis. Improvements on the readability of the thesis, its English, the discussion and presentation of concepts have been made through by his advice and corrections.

The author thanks Mr. Clement Leung for his continuous encouragement, discussions, and overviewing the draft.

The author thanks Mr. David Misunas for discussions, and Mr. Glen Miranker, also for discussions and, above all, for the utilization of his library of manuals and catalogues which have provided the author with a concrete basis for his ideas.

The author thanks the HOKUSHIN ELECTRIC WORKS, LTD. of Japan, for the financial support during this study.

TABLE OF CONTENTS

Chapter 1

1. Introduction

1.1 Background of the Study

Background

To meet the demands for high-speed computation, a completely new approach to the design of computer language and architecture has been proposed. The approach, which exploits the parallelism inherent in a computation to achieve the desired high-speed, is known as the concept of data-flow.

A high-level computer language, known as the data-flow language [2,7,5,6], has been designed to allow a natural expression of the parallelism in programs.

The language is a radical departure from conventional machine languages, and the conventional computer architectures are not organized to execute a program written in this language at a satisfactory speed.

Efforts have therefore been concentrated on the development of a new computer architecture, known as the data-flow processor [5, 6, 7, 12], for the implementation of the language and the evaluation of the computation scheme based upon the data-flow concept.

The Data-Flow Processor

The data-flow processor is actually a generic term for several types of architectural configurations. Each type of processor is defined to implement one of a family of data-flow languages of different but increasing expressive power.

So far, the data-flow processor can be classified according to four levels of capabilities vested to them:

Level 1: processors which can execute reasonably small programs with iteration operations but no data structures.

Level 2: processors which can handle data structures [9, 13].

Level 3: processors which are equipped with multi-level memories to handle large programs (i.e. caché-bulk).

Level 3: processors which can handle procedure activations and streams [22].

The data-flow processors which belong to the Level 1 are the most fundamental, and two types of them have been introduced. One is known as the <u>elementary data-flow processor</u> [5], and the other is known as the <u>basic data-flow processor</u> [6,7].

The Level One Data-Flow Processors

The elementary data-flow processor is developed to execute a program written in the elementary data-flow language [5], while the basic data-flow frocessor is developed to execute a program written in the basic data-flow language [7]. The elementary data-flow language is not equipped with the decision capability, and the basic data-flow language is simply the elementary data-flow language augmented with the decision capability.

The level one data-flow processors are considered the most basic architecture among all types of data-flow processors, since it embodies most of the important characteristics common to all the data-flow processors.

Design Target

The Level one data-flow processors should thus be the first target in the realization of the data-flow processors. It is the objective of this thesis to provide some concrete design proposals for the realization of a module of the basic data-flow processor, the Cell Block Module.

Before proceeding into datails, the computation scheme of the basic data-flow processor is briefly explained in the next section. This scheme is quite different from conventional computation architectures.

1.2 Computation Scheme of The Basic Data-Flow Processor

The basic data-flow processor has the configuration shown in Fig. 1.1. It consists of four sections: the Memory Section, the Arbitration Network, the Functional Unit Section, and the Distribution Network.

The data-flow program to be executed is represented in the Instruction Cells of the Memory Section. Each Instruction Cell consists of four registers, as shown in Fig. 1.2. The first register (register #1) holds an instruction which is composed of an op-code and destination address(es). The op-code specifies the operation to be performed, and each of the destination address(es) specifies the address of a register of a destination Instruction Cell, to which the result of the operation is to be directed. The second, third and fourth registers hold operands to be used in the execution of the instruction.

When an Instruction Cell is loaded with an instruction and the necessary operands, it is enabled and its contents are transmitted as an operation packet. Based upon the op-code, the operation packet is routed through the Arbitration Network, which is a switching network, to an appropriate Functional Unit where the operation specified by the op-code is performed on

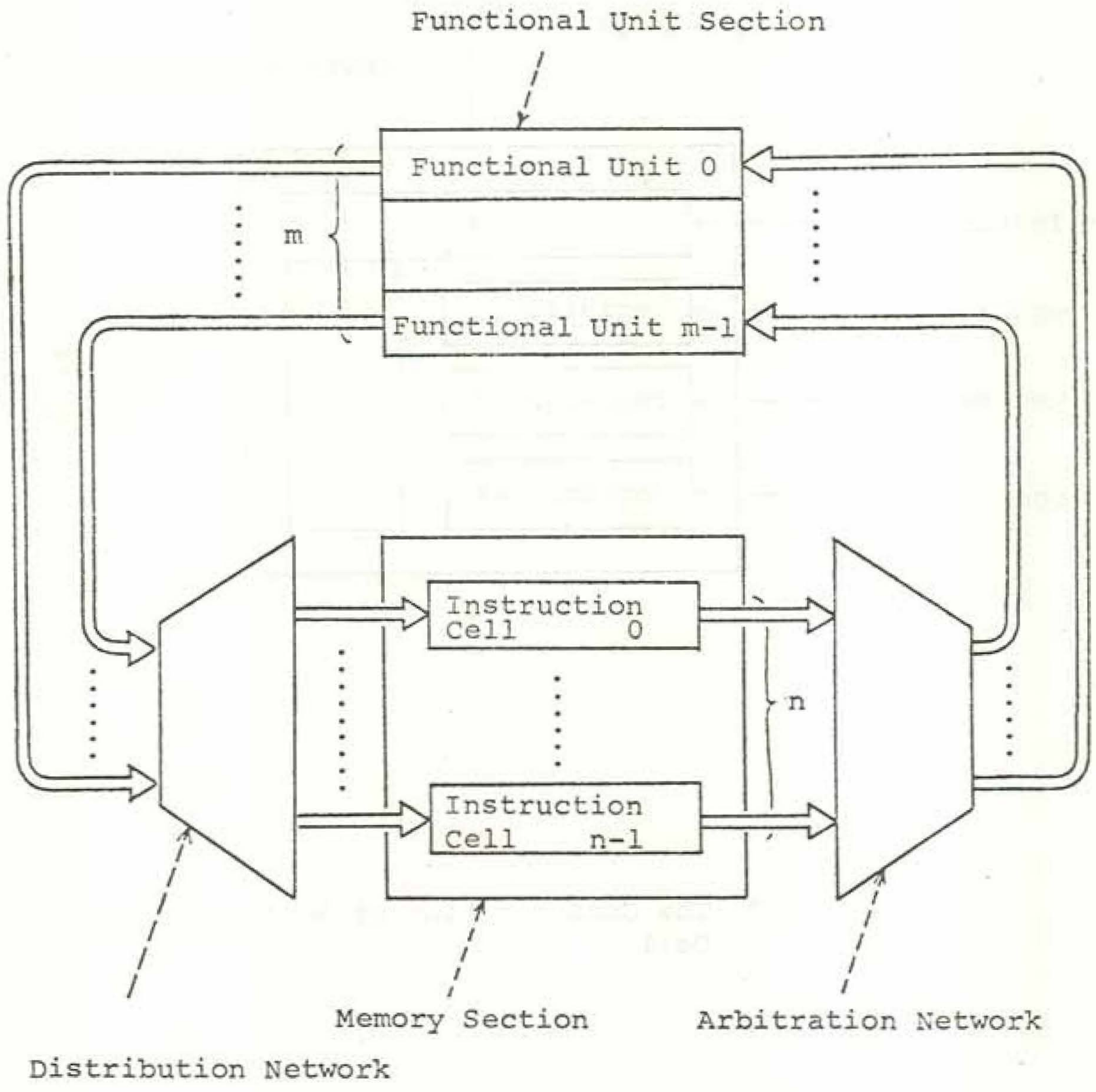


Fig. 1.1 The Configuration of the Elementary Data-Flow Processor

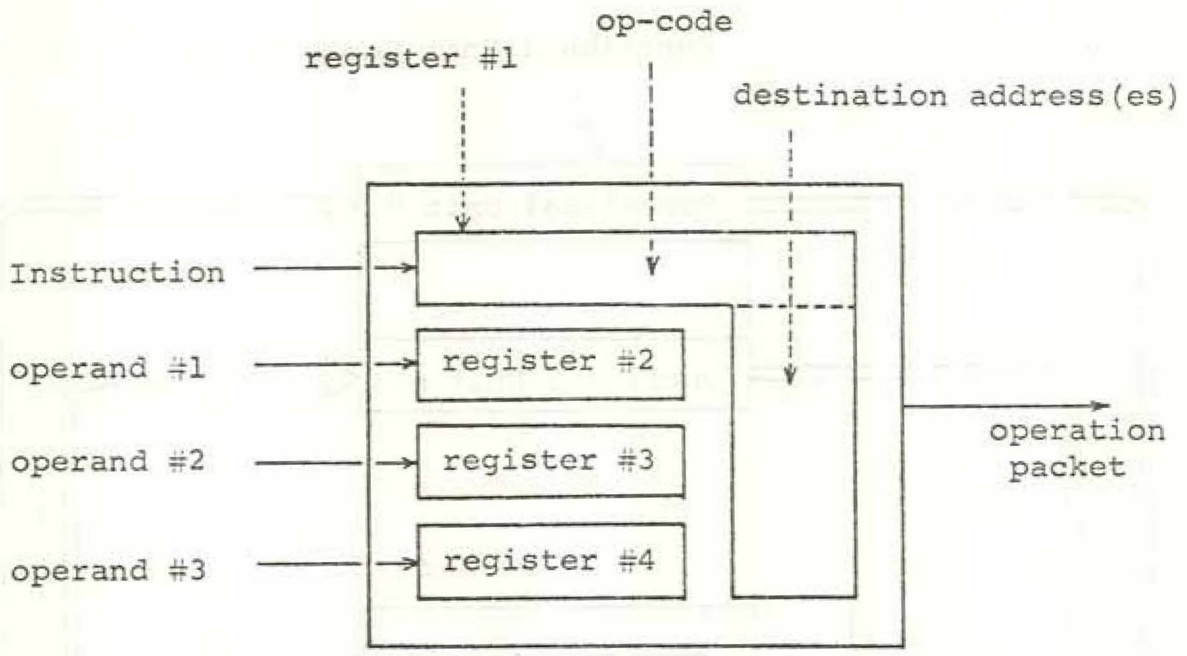


Fig. 1.2 The Configuration of an Instruction Cell

the operands.

The result of the operation is then sent to each destination address as opnd packets. The opnd packets proceed through the Distribution Network, which is also a switching network, and are delivered to the destined register of the destined Instruction Cell.

Each Instruction Cell stores the result as an operand, and may be enabled in turn. If it is enabled, its contents are eventually transmitted as an operation packet as well.

Concurrent Computation Capability

As easily seen, many Instruction Cells may be enabled simultaneously, and it is the function of the Arbitration Network to efficiently deliver operation packets to the Functional Units and to queue operation packets waiting for each Functional Unit. Similarly, the Distribution Network may also have many packets travelling through it simultaneously. In addition, the Functional Units are organized in a pipeline fashion. Thus, all major sections of the processor are organized to operate with a high degree of concurrency.

Deadlock Avoidance

To avoid deadlocks in the highly parallel computation scheme explained above, it is necessary to introduce a third type of packet, called ACK packet [11].

It works as follows: when an operation packet μ from an Instruction Cell α is processed at a Functional Unit, an ACK packet is transmitted to each Instruction Cell which has provided an operand to Instruction Cell α to form the operation packet μ.

The incorporation of the ACK packet to the basic computation scheme requires modification to the fields for destination addresses. An instruction of an Instruction Cell must include the addresses of Instruction Cells to which ACK packets must be sent, as well as the destination addresses to which the computation result is delivered. The condition for an Instruction Cell to be enabled is also modified. An Instruction Cell is enabled when it is loaded with an instruction and the necessary operands, and, in addition, has received an ACK packet from each Instruction Cell to which its computation result has been delivered.

The ACK packets are generated in the Functional Unit Section, and transmitted to the Memory Section together with the opnd packets.

The two types of packets, opnd and ACK, are called <u>event</u> packets. Hence the Memory Section receives event packets and transmits operation packets. When the basic data-flow processor is working under normal condition, which means that neither program loading nor error handling is taking place and program execution is conducted successfully, there are only event packets and operation packets in the whole architecture.

1.3 The Cell Block Module

Because it is the most complex part of the basic data-flow processor, the Memory Section is considered the best section to investigate the feasibility of the realization of the basic data-flow processor. After carrying out a logic design for the Memory Section, it will be easier to estimate with relatively high accuracy the performance and complexity of the entire architecture as well as its cost, and other factors which can only be estimated after a design is carried out.

The Memory Section is actually a collection of identical unit, namely the Instruction Cell. Each Instruction Cell must perform rather complex tasks, including the reception of event packets, the loading of operands, the managerial operations to update the states of the Instruction Cell, the examination of the enabling conditions, and the transmission of the contents of the Instruction Cell as an operation packet.

In addition, a mechanism for loading data-flow programs, a facility to dump out the contents of memory, and an error checking mechanism for the received packets, are considered necessary.

The complexity of the Instruction Cell is considered to lie mainly in the control section of the Cell, therefore sixteen Instruction Cells are consolidated into one. This performance - cost tradeoff has produced a function module, known as the Cell Block Module. The Memory Section, then, is constructed from Cell Block Modules.

To realize the basic data-flow processor in the near future, this Cell Block Module should be implemented as a first step, and it is the objective of this thesis to carry out a logic design of the Cell Block Module to direct the implementation.

1.4 Statement of the Problem

It is strongly desired to realize the basic data-flow processor in the very near future.

A logic design for the Cell Block Module is needed, in particular one which employs conventional commercial electronic devices available today.

A publication, including the specification of the Cell Block Module, has been prepared for this purpose [4].

It is the objective of this thesis to carry out a logic design for the Cell Block Module, utilizing conventional commercial electronic devices available today, according to the specification already published.

1.5 Synopsis of Thesis

In chapter 2, a preliminary discussion is developed to identify the most fundamental characteristics of the resulting Cell Block Module. The discussions concentrate on the choice of the communication scheme, the design tools and the design approach, choice of logic elements, the speed-space tradeoff, the properties of the specification, and the design assumptions.

In chapter 3, the data-flow structure of the Cell Block Module is developed in a top-down fashion. Also the control structure of the Module is developed. Coordination of two independent operation sequences in the Module is discussed.

In chapter 4, a performance analysis is carried out.

In chapter 5, concluding remarks are given and possibilities for further improvements are discussed.

Chapter 2

2. Preliminary Discussions on the Resulting Structure

The first step of the design activity is to choose a possible structure and investigate the fundamental characteristics of the resulting Cell Block Module in a general context.

Two issues which influence significantly the resulting structure are the choise between asynchronous and synchronous control schemes, and the approach to functional decomposition.

In addition, several other issues, though less significant, must be taken into consideration in advance before proceeding into details in carrying out the design trade-offs. They are, for example, the logic elements to be employed and the speed-space trade-offs. These issues are more or less interrelated, and are discussed as a whole to yield design assumptions which are summerized in the last section of this chapter.

2.1 Comparison between Asynchronous and Synchronous Control Schemes

In general, a hardware system can be implemented by employing either a synchronous or asynchronous control scheme, or possibly a combination of both.

A synchronous system does not need mechanisms for detecting the completions of operations. It is assumed for this scheme that every initiated operation always completes within a time duration determined from data on specification sheets. Based on this assumption, the control of operations is performed by making use of a special purpose signal - a master clock - which synchronizes the initiation of operations.

This scheme has been practiced widely, and is regarded as a standard. Almost all of the electronic devices today, the large scale integration devices in particular, are synchronous and design procedures utilizing this scheme are well established and widely known.

But a synchronous system has disadvantages due to the lack of mechanisms for detecting completions of operations. Unexpected delays of operation may cause system malfunctions, and, even worse, the malfunctions may not be detected. The only way to cope with this situation, though still imperfect, is to allow ample margin in the timing estimates for completion of operations to prepare for the worst cases. Consequently, the operation speed of a synchronous system tends to be slow.

Furthermore, the master clock is the only mean to synchronize operations being executed in the system, and the clock signal is sent to all parts of the system. Therefore, if the system grows in size, the propagation delay of the clock signal increases, and the operation speed of the system tends to be suppressed. This situation emerges as a crucial problem in large systems intended to perform high-speed computations. A compromising solution is to adopt a decentralized clock system, but the synchronization of operations at the boundaries, where different clocks meet, is still unsolved as one of the most fundamental problems.

On the contrary, the asynchronous system resolves the synchronization problem by employing special purpose hardware elements such as the arbiter [16,21,23] and the C-module [14, 23]. These elements guarantee the correct operation of the system to be unaffected by changes in either the operation speed or signal delays [3, 8, 10, 20].

The asynchronous system, equipped with these elements for synchronization, exhibits the indispensable property of effective performance when 1) high-speed computation is required, 2) the size of the system grows, 3) the synchronization of operations whose occurences and completions are difficult to predict, and 4) parallel computation is performed.

An asynchronous system employs either one or both hardware elements wherever synchronization is required, and in general many of these elements are required in a system. Although each of the hardware elements has a rather simple configuration, there are no available electronic devices for realizing them except individual gates and flip-flops.

For this reason, an asynchronous system will need much space, because a significant portion of the space is devoted to implementing the basic hardware elenemts, and will result in a large and rather expensive architecture. These are the disadvantages of an asynchronous system.

It is obvious that these disadvantages are eliminated if an asynchronous system is realized as a composition of a small number of devices, each of them being constructed in large scale integration technology.

The Cell Block Module, as a part of the elementary data-flow processor which is inherently an asynchronous system, must observe the asynchronous communication scheme in interfacing with other modules. In addition, it is quite natural to expect that the asynchronous communication scheme is also obeyed in communications internal to the Cell Block Module.

2.2 Design Tools and Functional Decomposition

2.2.1 Design Tools

As a tool for designing an asynchronous system, a descriptive scheme has been developed and utilized widely [23]. This scheme makes use of two graphs - data-flow graph and data-dependency graph - to represent the organization and functions of a system. The data-flow graph describes constituent devices of the system and data paths between them. The data-dependency graph describes the control of data transfer from one device to another and the operations performed on the data upon the completion of the necessary data transfer.

The combination of the data-flow graph and the data-dependency graph is very useful. An alternative to the data-dependency graph, known as Petri Net, has also been utilized widely. The Petri Net is originally developed to model the behavior of systems by studying the occurrences of events and coordinations between them.

The pair of data-flow graph and Petri Net has proven to be a very effective design tool [1], and are employed in the logic design of the Cell Block Module.

Patil's Realization Scheme

The employment of Petri Nets, which are used to describe control sequences, is also supported by previous works by Patil [15, 16, 17, 18, 19], in which it is shown and proven that a control structure can be implemented correctly provided its sequence constraints is given in the form of a Petri Net.

The implementation, in his works, is carried out to the gate-level, and PLA may be utilized instead of gates.

The results of his work are adopted as a basis for the design of the control structure of the Cell Block Module.

2.2.2 Properties of the Specification

The Cell Block Module is defined abstractly, and its specification is written in a high level computer language, ADL (Architecture Description Language), as shown in Appendix 1. There are a couple of noticeable characteristics implied in the specification. The specification describes the operations to be performed in terms of abstract operational primitives. No implications regarding actual characteristics of devices is implied in the specification.

This characteristic enables a designer to enjoy presumably a lot of freedom to make choises in order to incarnate his concept. Interestingly enough, this characteristic motivates the designer to exhibit his full abilities.

Another characteristic is that the specification, utilizing the block structured description in ADL, eliminates the possibilities of leaving unspecified some operations, corresponding to special cases which occur so rarely and are easily overlooked at the very start in the design. A fucntionaly complete specification spares a designer from checking around in an ad hoc manner to get rid of design errors to implement a well engineered system.

The third characteristic, which relates to the operations taking place in the Cell Block Module, is that the module performs concurrent computation in two independent sequences which can be activated independently. From the point of view of operation speed, the concurrent computation capability is quite desirable.

Concurrent Computation in the Cell Block Module

In the Cell Block Module, concurrent computation takes place as follows:

One sequence, called E/C SEQUENCE, is activated upon the arrival of an event/command packet from the Distribution Network, and processes the packet. The processing of the packet may result in the generation of an enabled cell.

The other sequence, called OPT SEQUENCE, is activated when there is at least one enabled cell in the module and a ready signal to accept an operation packet from the Arbitration Network is asserted. The sequence picks up an enabled cell and transmits it as an operation packet for the Arbitration Network. The enabled cell leaves the enabled state when the transmission is completed.

The communication between the two sequences are performed by passing the address of an enabled cell.

Apparently, the operations performed in the Cell Block Module are the ones which take place in a general one producer-one consumer system. The problem regarding the cinflicts in access is typically solved by utilizing a FIFO queue.

It is decided to implement this characteric, rather than execute the E/C SEQUENCE and the OPT SEQUENCE sequencially.

In the course of the design, two synchronization problems arise, one between the E/C SEQUENCE and the OPT SEQUENCE, and the other is the resolution of priorities between the processing of run packets and these two sequences. These problems are dealt with in Sec.2 of Chap.3.

The problem of adopting synchronous devices to form an asynchronous structure also arises. This problem is dealt with in Sec.3 of Chap.2.

2.2.3 The Top-down Decomposition

As is well known, the top-down approach is quite effective in designing well structured systems. In view of the way in which the specification is given, it is considered the approach to be taken up. Following the top-down approach, the original abstract system is decomposed into submodules step by step, and at each step, appropriate choices with regard to the adoption and selection of conventional commercial device may be made.

It should be emphasized that it is possible to eliminate the design flaws which may take place on the interfaces between devices by utilizing the top-down approach intentionally to make the resulting structures always form a modular system at each step. Enforcing this rule provides the resulting structures with two advantages:

(i) it facilitates the realization of the entire structure on account of modularity,

(ii) it is easier to replace modules by equivalent ones implemented with advanced and improved devices.

2.3 Miscellaneous Remarks

Utilization of the Synchronous Devices

Considering the complexity of the Cell Block Module described in the specification, it becomes necessary to employ LSI devices such as RAMs, ROMs, PLAs, and other LSI functional devices as the primary constituents, rather than SSI or MSI devices.

Because these LSI devices are in general manufactured for synchronous machineries, they are not equipped with the function of generating acknowledge signal(s).

To overcome this problem, a delay line is utilized to simulate the function. Upon the acceptance of an activation signal, it is directed to the delay line whose output signal is returned as an acknowledge signal. The delay time is determined by the maximum time of operation calculated from the values contained in manuals or data sheets.

Speed-Space Trade-off

The high-speed computation capability is always pursued as the most significant objective , which is justified in chap. 5.

Cost, size, and power consumption, are taken into consideration as objectives of secondary importance.

2.4 Summary of Design Assumptions

The logic design of the Cell Block Module is carried out with a top-down approach to generate an asynchronous modular configuration.

A set of data-flow graphs and a set of Petri Nets are provided to describe the resulting system. Control modules are provided to specify the operation sequences, but actual device configurations for them are not provided.

The high-speed computation capability is considered the most significant target with respect to the performances of the resulting Cell Block Module.

Implementation devices are chosen from conventional commercial electronic devices available today. LSI devices are in particular preferred.

Chapter 3

3. The Design of the Cell Block Module

In this chapter, a logic design of the Cell Block Module is carried out, yielding the data-flow structure and the control structure.

Although the data-flow structure and the control structure are closely related to each other, the data-flow structure can be determined rather independently. Firstly, the data-flow structure is developed, and then the control structure is developed utilizing the resultant data-flow structure.

It should be remarked that the data-flow structure is designed so that initialization and update of the cell states can be completed in a short time. The reason is explained in sec.2 of chap.3 (exploiting the concurrent computation capability), in terms of the control structure.

3.1 The Design of the Data-Flow Structure

3.1.1 The Level One Design

At the top-most level, the Cell Block Module is viewed as a black box with four ports for communication with other subsystems, as shown in Fig.3.1.

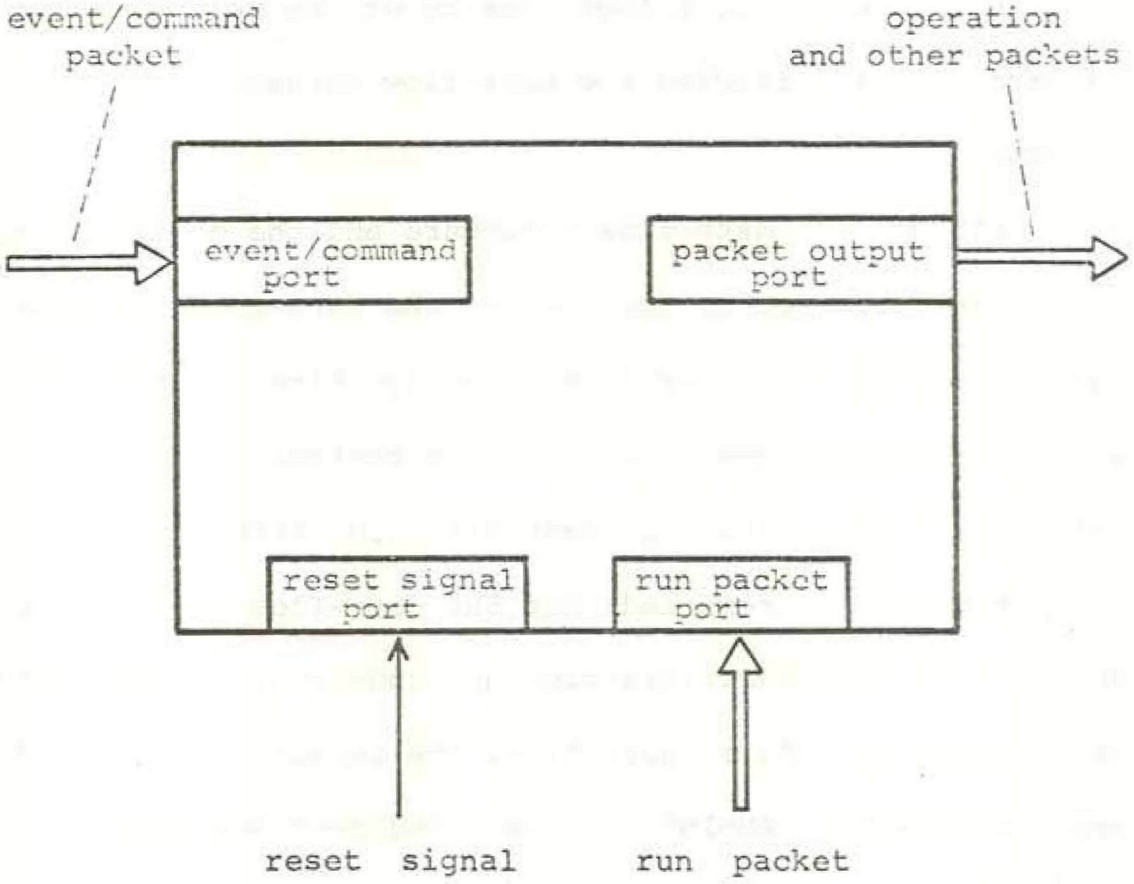


Fig. 3.1 A Black Box Model of the Cell Block Module

The event/command port is dedicated to receiving event and command packets, while the packet output port is dedicated to the delivery of packets from the Cell Block Module. These two ports are utilized to load and to execute a data-flow program.

The other two ports are utilized to control the running stste of the Cell Block Module from the outside. The run packet port receives run packets to change the Cell Block state between the disabled and enabled states. The reset signal port receives reset signals which force the Cell Block Module to be reinitialized.

Each event/command packet is received at the event/command port of the Cell Block Module and appropriate operations are performed on the information it conveys. The operations to be performed are given in the ADL specification (Appendix 1), and the Cell Block Module hardware must execute these operations to fulfill the requirements. During the execution of these operations, a cell may be enabled, and eventually its contents is transmitted at the packet output port as an operation packet.

We are led natually to a two-bus system, one for the input packets and the other for the output packets.

The Cell Block Module deals with two types of information stored in its memory. One type includes all the contents of a cell to be formed into an operation packet, i.e. instruction, operands and addresses for sending the results and acknowledgements. The other includes all information for keeping track of the current state of a cell. The cell content information does not influence the execution of operations in the Cell Block Module. On the other hand, the cell state information is referred to during the execution of operations and controls the operations to be performed. The cell state information is updated during the execution of the operations.

The cell content information and the cell state information, each serves a different purpose and can therefore be stored in separate memory devices. This configuration enables both the cell state information and the cell content information to be accessed independently and utilized concurrently, resulting in obvious improvement in the operation speed by reducing memory conflict.

By naming the memory device for the cell state information C Memory and the memory device for the cell content P Memory, the conceptual configuration of the data-flow structure is depicted as shown in Fig.3.2.

In the configuration, there are two buses and two memories, as described in the preceding section. The buses in the

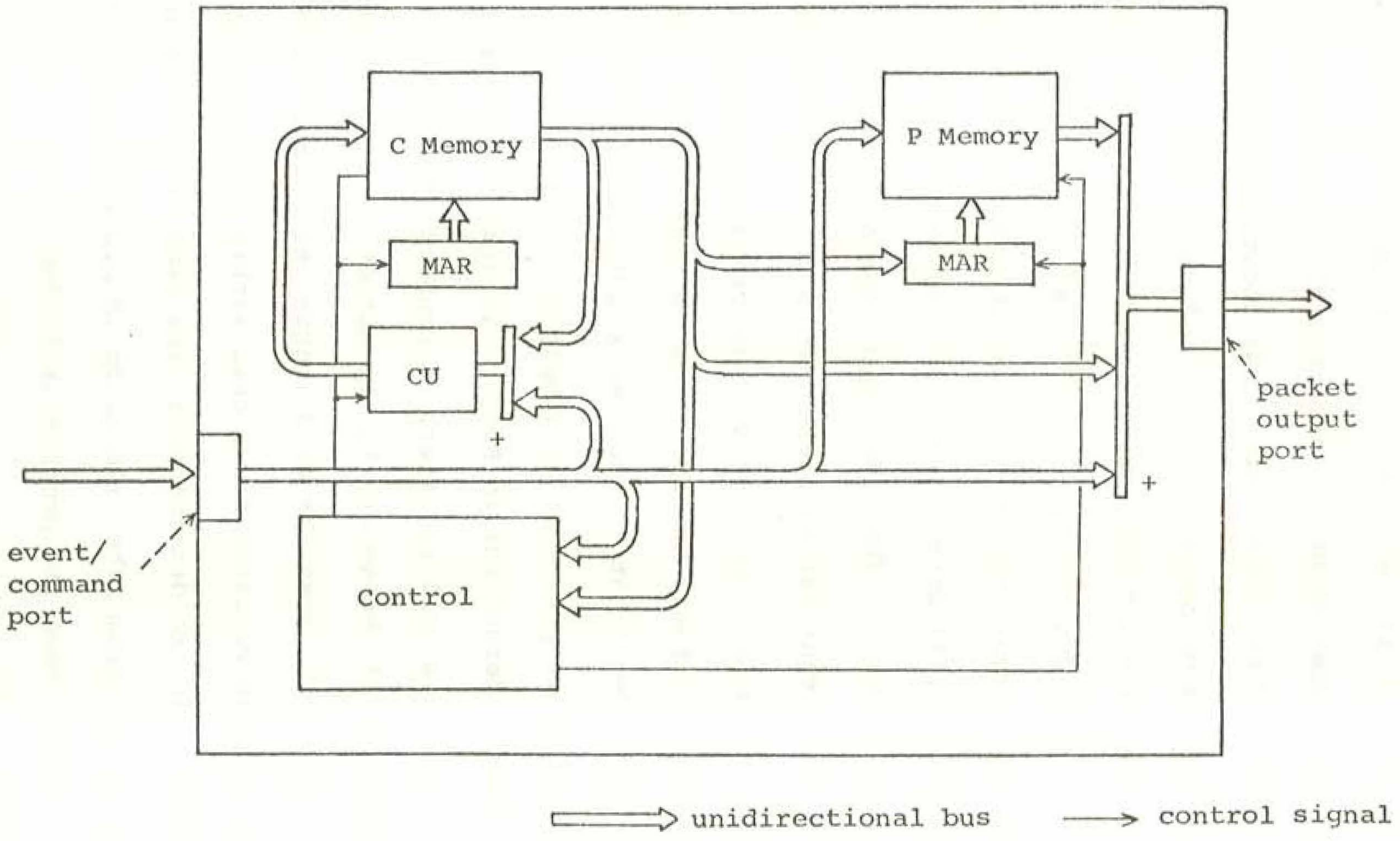


Fig. 3.2 The Conceptual Configuration of the Cell Block Module

structure, including two other internal buses which are explained later, are all unidirectional. By adopting unidirectional buses, the transmission of data naturally incarnates the concept of data-flow. Furthermore, the adoption of an unidirectional bus increases operation speed and eliminates the necessity of registers which are required to hold temporary data. For example, consider the case of an internal bus which connects the output of C Memory to the input of it via a module named CU. CU is a combinatorial circuit, and is used for the update operations of C Memory. The C Memory is assumed to have separate I/O ports. Its output can be presented to CU that performs the appropriate operations on this input to yield the result at the input terminals of the C Memory. Hence, a register for holding the computation result at CU has been eliminated, and the update of C Memory is performed by simply providing a write pulse.

The other internal bus in Fig.3.2 is used to access and store operands in a cell register. It connects the memory address register of the memory to the output of the C Memory. This is necessary because the word length and relative location of an operand register for a cell within the P Memory are not fixed, and are determined at the time of compilation of a data-flow program. The relocation information is stored into the C Memory when the program is loaded.

In Fig.3.2, the T-shaped symbol labeled with a + sign stands for a wired-or connection of buses.

To implement all of the specified operations, a few other buses are incorporated in Fig.3.2. They are utilized to execute operations for processing the error or the dump packets.

3.1.2 The Level Two Design

Based on the conventional configuration in Fig.3.2, datails of the data-flow structure of the Cell Block Module are developed in this subsection.

To explain the resulting data-flow structure, it is necessary to understand the detailed operations of the E/C and OPT SEQUENCEs. We shall, therefore, describe the operations of these sequences and the data-flow structure necessitated by these operations.

The Operations for ACK, SET-ACK, and SET-VAR Packets

The E/C SEQUENCE processes seven types of event/command packets (Appendix 1). These can be categorized into two groups according to their demands on the C and P Memories.

The first category , which includes the types of ACK, SET-ACK, and SET-VAR, consists of packets which utilize only the C Memory when being processed. The contents of the C Memory may be read out and modified, or be simply updated with the information carried by an event/command packet, but the P

Memory is not utilized by any of the operations performed by the E/C SEQUENCE.

The processing of an ACK packet is illustrated below as a typical example.

Example:

When an ACK packet is received, the E/C SEQUENCE extracts the cell address from the packet, and puts it into the memory address register of the C Memory. The addressed cell state in the C Memory is read out, and reloaded into the C Memory at the same location.

The new cell state computed during the update operation may satisfy the conditions for enabling a cell. If so, the cell state is reset to its initialized value.

Thus to route a cell address, two paths are necessary, one from the input bus to the memory address register of the C Memory, the other from the memory address register to the FIFO queue. A closed path also exists for updating the contents of the C Memory.

The Operations for OPND, SET-INSTR, SET-CON, and DUMP Packet

The second category, which includes the types of OPND, SET-INSTR, SET-CON, and DUMP, consists of packets whose processing utilize both the C Memory and the P Memory.

For the types of OPND, SET-INSTR, and SET-CON, the C

Memory is utilized in the same way as for the types in the first category, namely in loading, reading out, and updating, for control purposes.

The packets of these types, however, carry more informations with them. They are values to be utilized as operands, op-codes or destination addresses in an instruction. This information is independent of the cell states and so is saved in the P Memory.

Furthermore, a byte-serial transmission scheme is employed to transmit information from and to the Cell Block Module [4]. Naturally, the information conveyed in a packet will be splitted into a sequence of bytes which are transmitted serially. As for the P Memory, it is reasonable to divide the memory area into sixteen contiguous smaller sections, one section for an entire cell, and to allocate contiguous memory locationd in accordance with the bytewise serial transmission of information. The resulting map of the P Memory is shown in Fig.3.3.

When an entire cell state in the P Memory is retrieved, the order of retrieval is the same as that of loading. Thus, the capability to increment the memory address register of the P Memory by one is desirable for simplifying the control mechanism.

A binary counter suits this purpose, and certain types of conventional commercial counters are furthermore equipped with

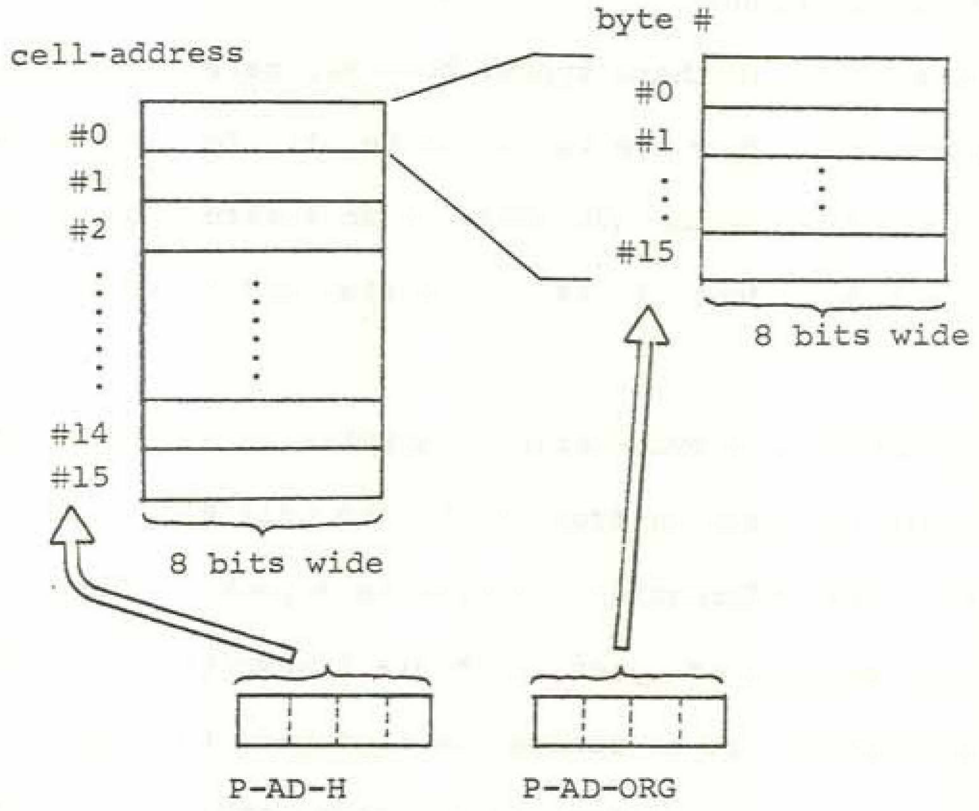


Fig. 3.3 Field Allocation in P Memory

other capabilities which can also be utilized to realize a simple control structure.

Two capabilities: resetting the contents to zero, and generating a carry signal indicating the contents of the counter to be of maximal integer value, are utilized to facilitate access to the entire content of a cell. If the memory locations allocated to a cell are aligned so that the first memory location starts at the zero count of the counter, the succeeding memory location at count one, and so on until the last memory location corresponds to the full count, then the execution of the reset operation on the counter automatically set the memory address register to the first address for a cell. And the assertion of the carry signal generated when the count is full can be used to detect that the last byte in a cell is reached. A separate mechanism to keep track of the memory references for detecting the processing of the last byte is not needed in the data-flow structure.

This scheme is applied in the design of the Cell Block Module, using a 4-bit counter as the lower order bits of the memory address register to access the contents of a cell in the P Memory.

In the specification [4], the total number of bytes of memory for the contents of a cell is fixed. However, the internal allocation of this memory areas to an instruction and all of its operands are left unspecified, allowing flexible and efficient memory utilization.

For each operand, the address of the first byte of the operand, relative to the starting address of the cell, and the length of the operand in bytes, are determined during compilation, and sent to the Cell Block Module by a SET-INSTR packet. They are stored in the C Memory as part of the corresponding cell state.

Upon execution of a loading operation for an operand conveyed by either an OPND or SET-CON packet, the starting address of an operand register and its length are both retrieved from the C Memory.

The starting address for the operand is loaded into the memory address register of the P Memory, and the length is stored into an auxiliary ccunter. The consecutive bytes of the packets are processed by storing their information in the P Memory and by incrementing the contents of the memory address register by one after each loading of a byte.

The contents of the auxilialy counter is also decremented by one at each loading.

The counter is responsible for indicating the arrival of the last byte of an operand through the use of a borrow signal. If the bytes carried by the packet run out before the counter reaches the zero count, zero's are provided to the P Memory as if they are the values carried by the packet, and the loading operations are repeated until the whole memory area for the operand is filled. This operation erases non-zero bit patterns left by the previous utilization of the operand.

Therefore, the counter which serves as part of the memory address register for the P Memory must be capable of parallel loading, as in certain types of conventional counters.

Example: OPND packet case

The operations which take place on the arrival of an event/command packet are illustrated below. As a typical example, the OPND case is considered.

On arrival of an OPND packet at the event/command port, the cell address carried by the packet is extracted and loaded into both of the memory address registers.

The cell state of the addressed cell, to which the packet is read out and utilized to determine whether the packet is an error packet or not.

If the packet turns out to be an error packet, the whole packet, including the cell address, is sent out from the Cell Block Module. No changes are made to the contents of either memory.

If the packet is not an error packet, the cell state is updated based upon the retrieved values. In the course of update, the cell may be enabled upon fulfillment of certain conditions. Then the cell address is directed to the FIFO queue in submission for the transmission of operation packet, and the cell state is reinitialized and stored as a new cell state into the C Memory. If the cell is not yet enabled, the updated state is stored as a new cell state.

After it is determined that the packet is not an error packet two data items, the starting address and length, are retrieved from the C Memory and sent to the memory address register of the P Memory and the auxiliary counter. The memory address register is loaded with the cell address in the higher 4 bits, and with the starting address, named ORIGIN, in the lower 4 bits. The auxiliary counter is loaded with the length in bytes of the memory area to be filled.

The data carried by the packet is stored into the P Memory as described previously.

For the type of DUMP the state and content of the cell addressed are sent out. These are stored in the C Memory and the P Memory respectively. Although both memories are utilized, the contents of the memories are unchanged after the processing of a DUMP packet.

The resulting configuration of the data-flow structure developed so far is shown in Fig.3.4, and 3.5.

More datails of the design are supplied in the following

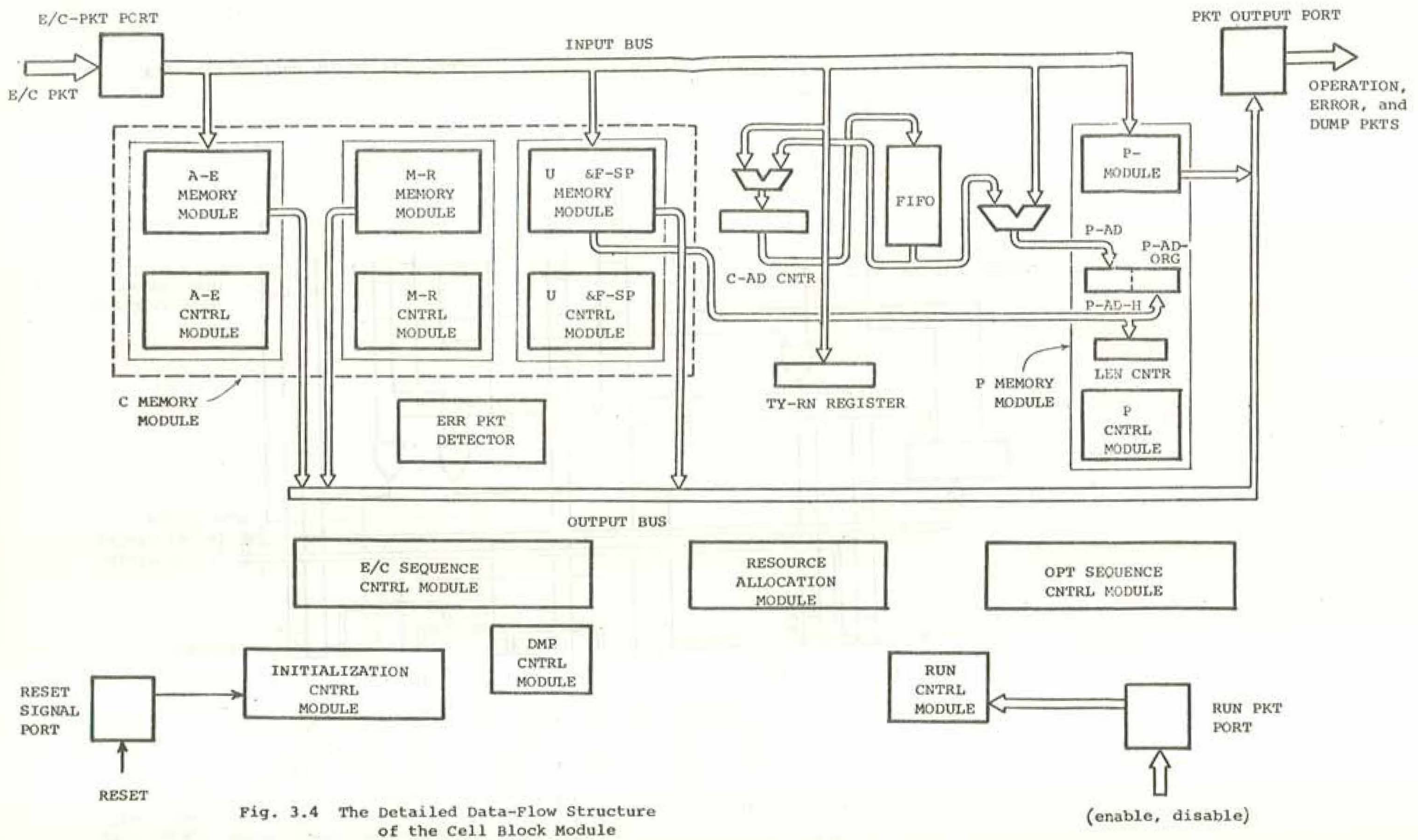


Fig. 3.4 The Detailed Data-Flow Structure of the Cell Block Module

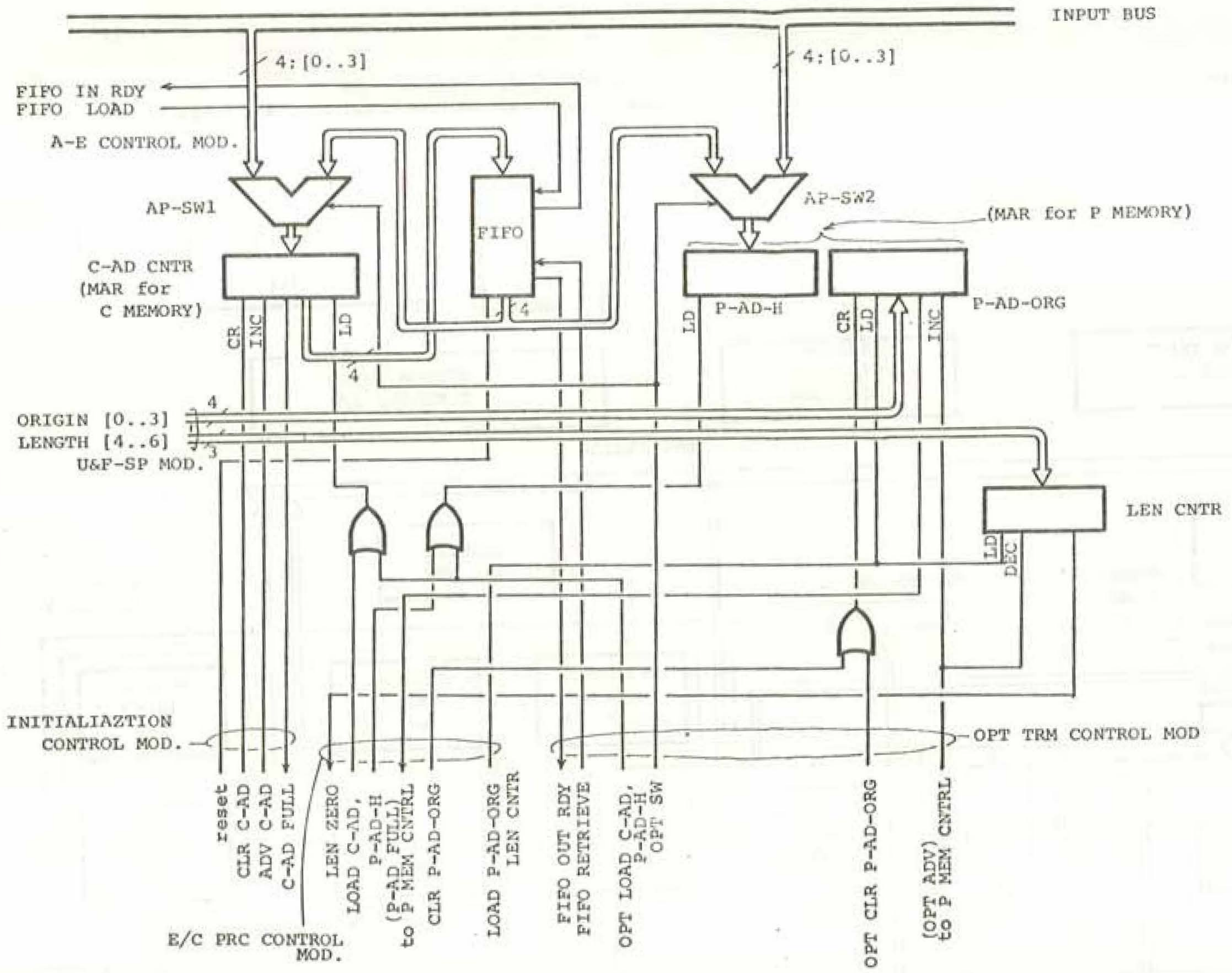


Fig. 3.5 DEVICES AROUND FIFO QUEUE

section.

3.1.3 More Detailed Design and Remarks

The size of the P Memory is 2K bits. A 256x8 configuration is chosen. A memory address register of 8 bits wide is needed to access the contents. The lower 4 bits are provided from a counter as described above, but upper 4 bits may be provided from a conventional register. A cell address is stored in the register for the upper 4 bits while the contents of the cell are accessed by specifying the lower 4 bits in the counter.

The memory address register for the C Memory must also be a counter, since part of every cell state must be initialized. The use of a counter facilitates the operations and yields a simplified structure.

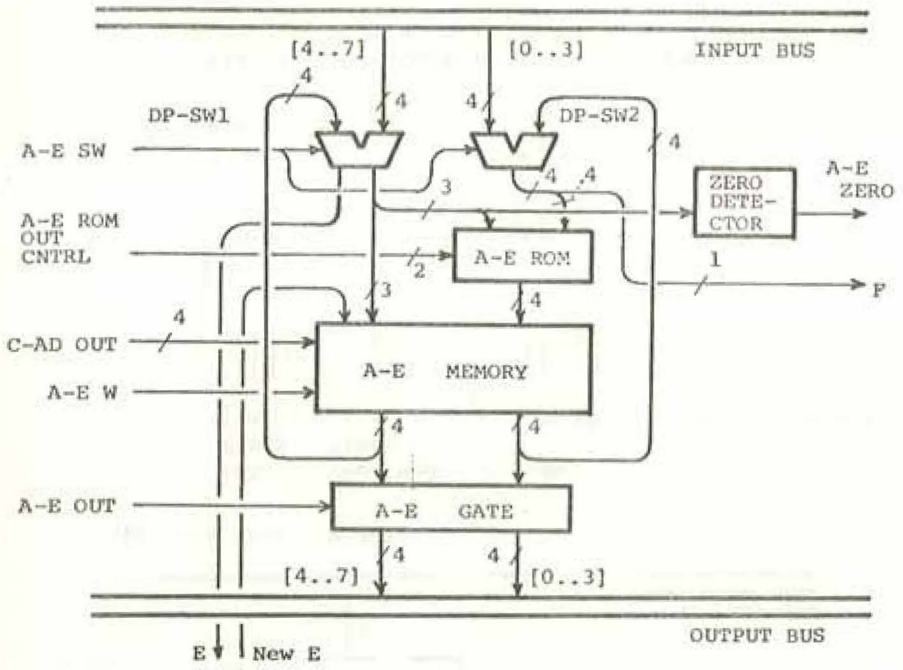
To represent the information contents of a cell state requires a relatively large number of bits. Considering the typical speed of the conventional memory devices, which may be regarded as fairly slow in comparison with the conventional functional devices, and also considering how each constituent of a cell state is utilized in the processing of packets, it is concluded that a horizontal configuration is suitable for the configuration of the C Memory.

A horizontal configuration requires a much larger number of memory outputs than that of a vertical configuration. The operation speed achievable only by a horizontal configuration makes up for this disadvantage because it exploits concurrent computation capability as explained in sec.2 of chap.3 .

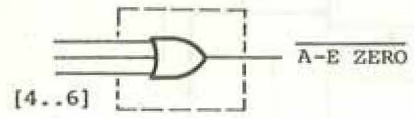
The information contained in a cell state, can be classified into three categories so that no interconnection exists between categories.

In the execution of loading, updating, and initializing operations on a cell state, the information in each category is processed independently and then put together to form a new cell state. Each category of information, therefore, is stored in a separate memory device, as shown in Fig.3.4 and the detailed configurations for the memory devices are shown in Fig.3.6 , Fig.3.7 , and Fig.3.8 respectively. It may be observed that the internal bus mentioned in terms of the module CU in sec.1 of chap.3 is implemented in the detailed configurations, and that operations, the loading and updating in particular, are simple and fast.

The configuration of the P Memory is described in Fig.3.9.

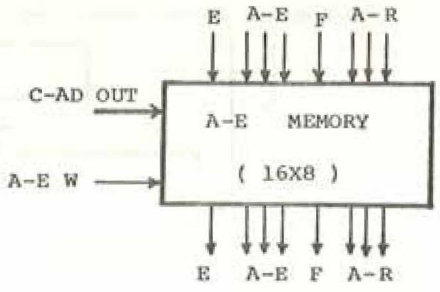


(b) A-E MEMORY Module



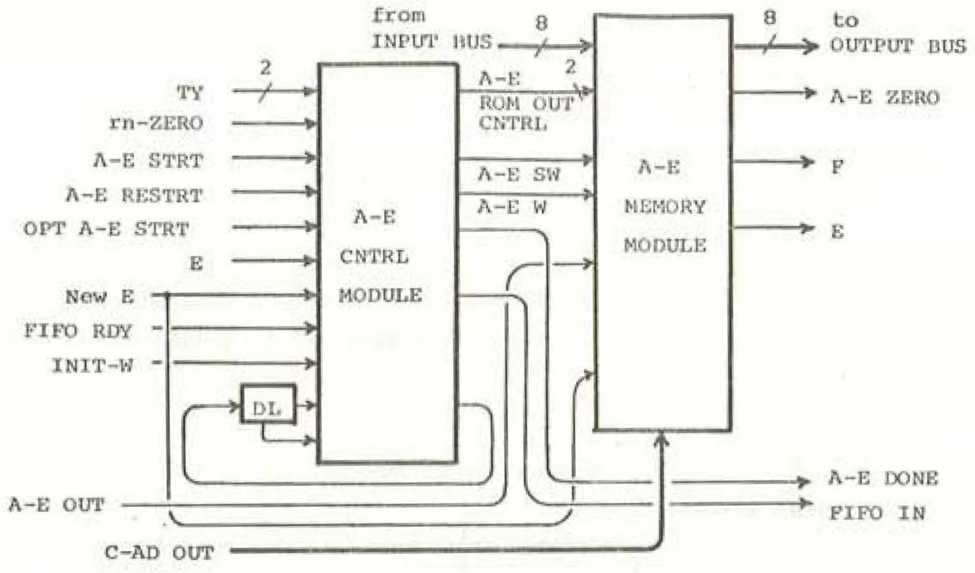
Corresponding to the bits of ack-exp

(c) ZERO DETECTOR

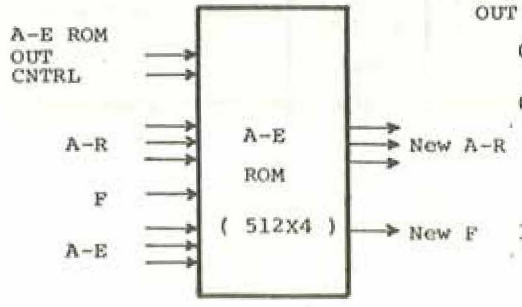


(d) A-E MEMORY Module

Fig. 3.6 The Configuration of A-E Module



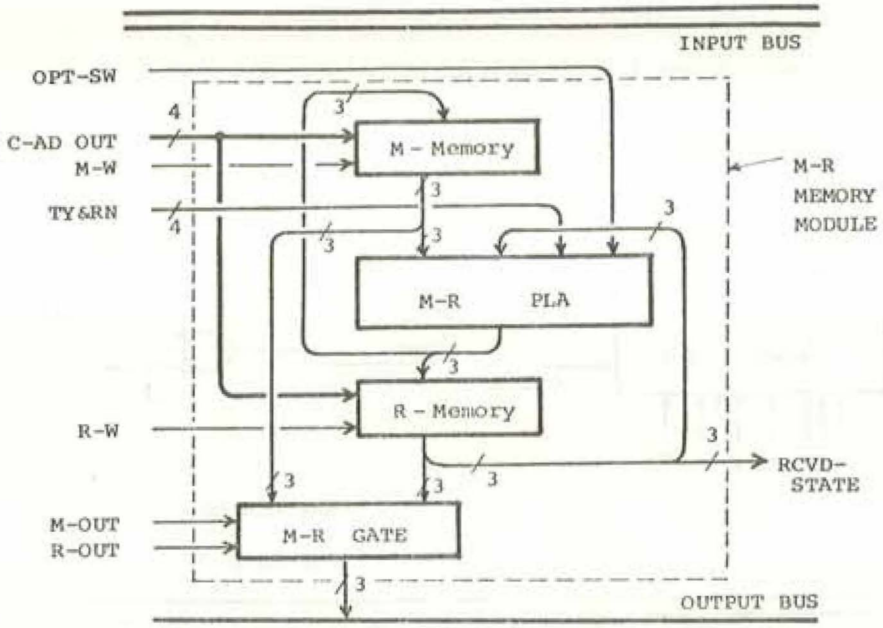
(a) Wiring of A-E Module



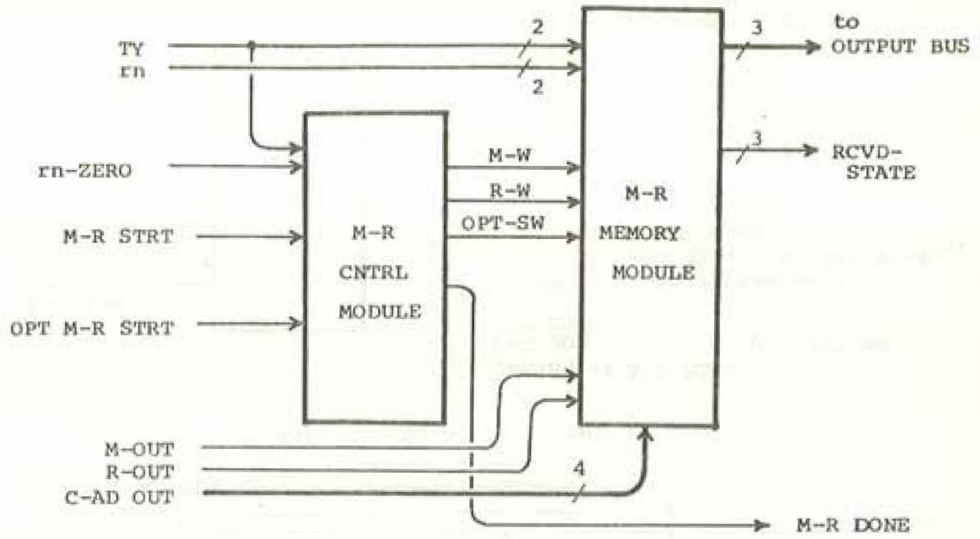
(e) A-E ROM

Coding of A-E ROM:

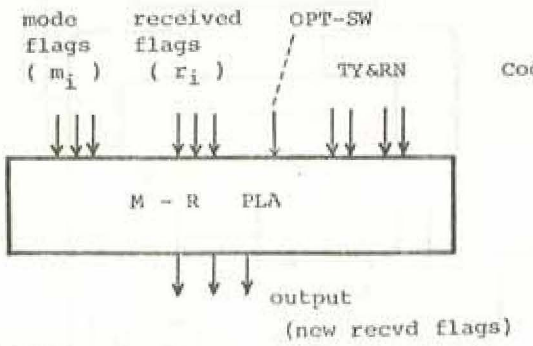
A-E ROM OUT CNTRL	OPERATIONS
00	(No operation) New A-R:=A-R; New F:=F;
01	(ACK Case) If A-E=A-R-1 then New A-R:=0; New F:=1 else New A-R:= A-R+1; New F:= 0;
10	(SET-INSTR,SET-ACK Cases) If A-E=A-R then New A-R:=0; New F:=1 else New A-R:=A-R; New F:= 0;
11	(OPT TRM Case) New F:= 0;



(b) M-R Memory Module



(a) Wiring of M-R Module

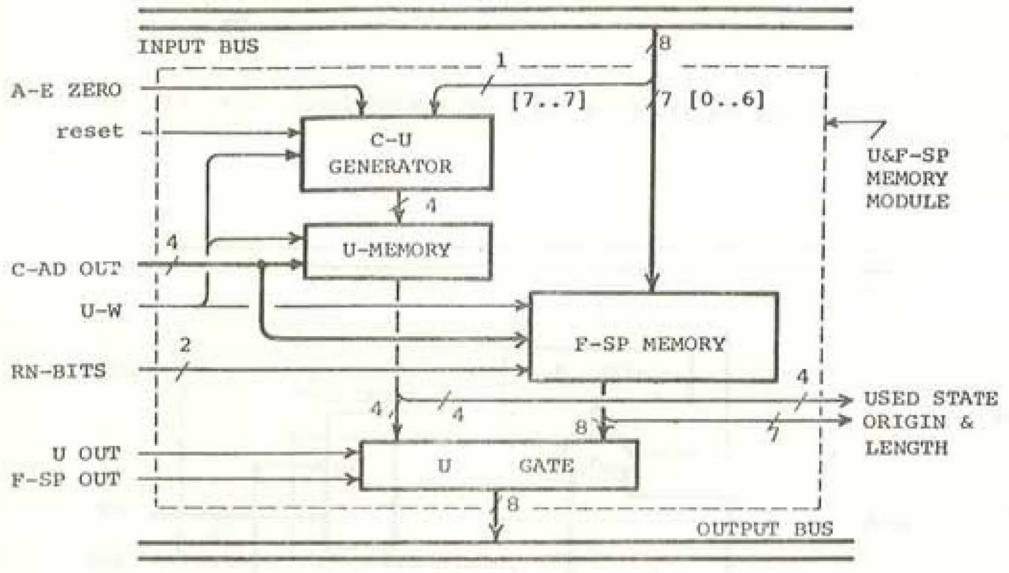


(c) M-R PLA

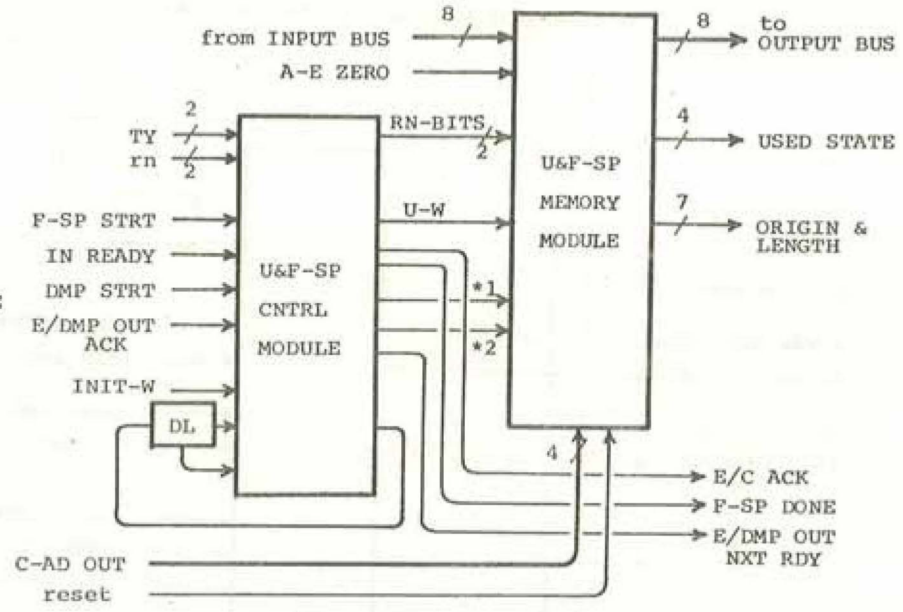
Coding of PLA:

```
[ $o_i$ ] is the output of the i-th bit of PLA
If OPT-SW=true, then [ $o_i$ ]:=[ $m_i$ ]; i=1..3 /*Reset operation of R-Memory*/
   else input packet type of
        SET-INSTR: [ $o_i$ ]:=0; i=1..3   /* Set all $m_i$ to be VAR */
        SET-VAR:   [ $o_i$ ]:=0; i=rn     /* Set the receiver rn to VAR */
                   [ $o_i$ ]:=[ $m_i$ ]; i≠rn
        SET-CON:   [ $o_i$ ]:=1; i=rn     /* Set the receiver rn to CON */
                   [ $o_i$ ]:=[ $m_i$ ]; i≠rn
        OPND:      [ $o_i$ ]:=1; i=rn     /* Set the receiver rn to 1 which */
                   [ $o_i$ ]:=[ $r_i$ ];   /* designates OPND has been received */
                           i≠rn
```

Fig. 3.7 The Configuration of M-R Module

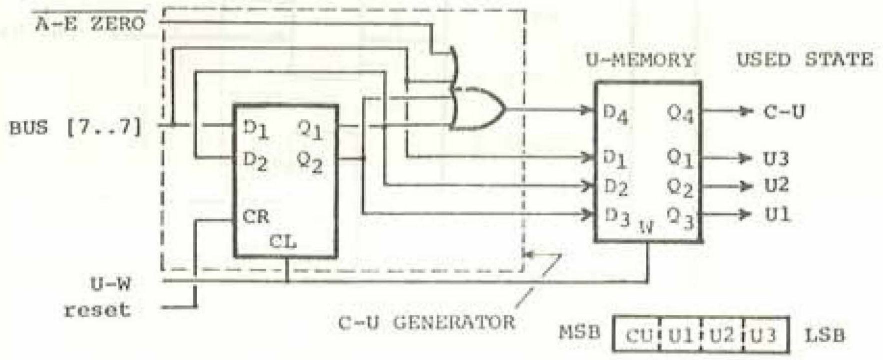


(b) U&F-SP MEMORY MODULE



*1 – U OUT, *2 – F-SP OUT

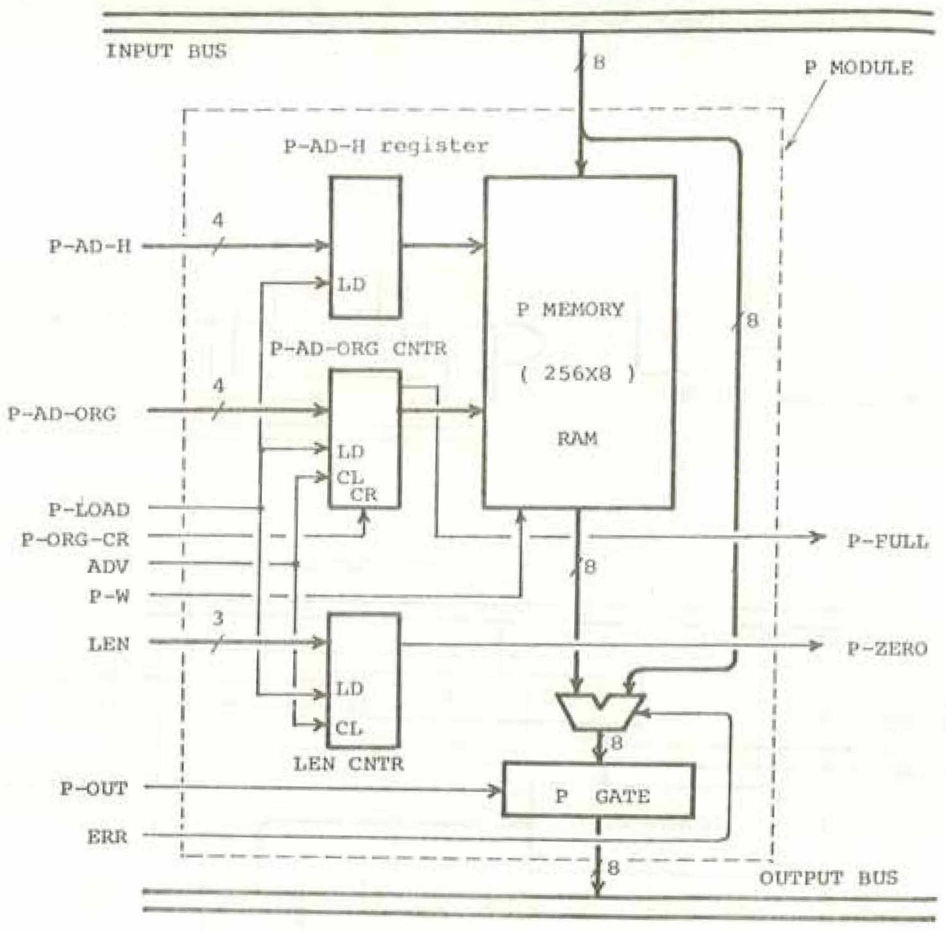
(a) Wiring of U&F-SP MODULE



(c) C-U GENERATOR

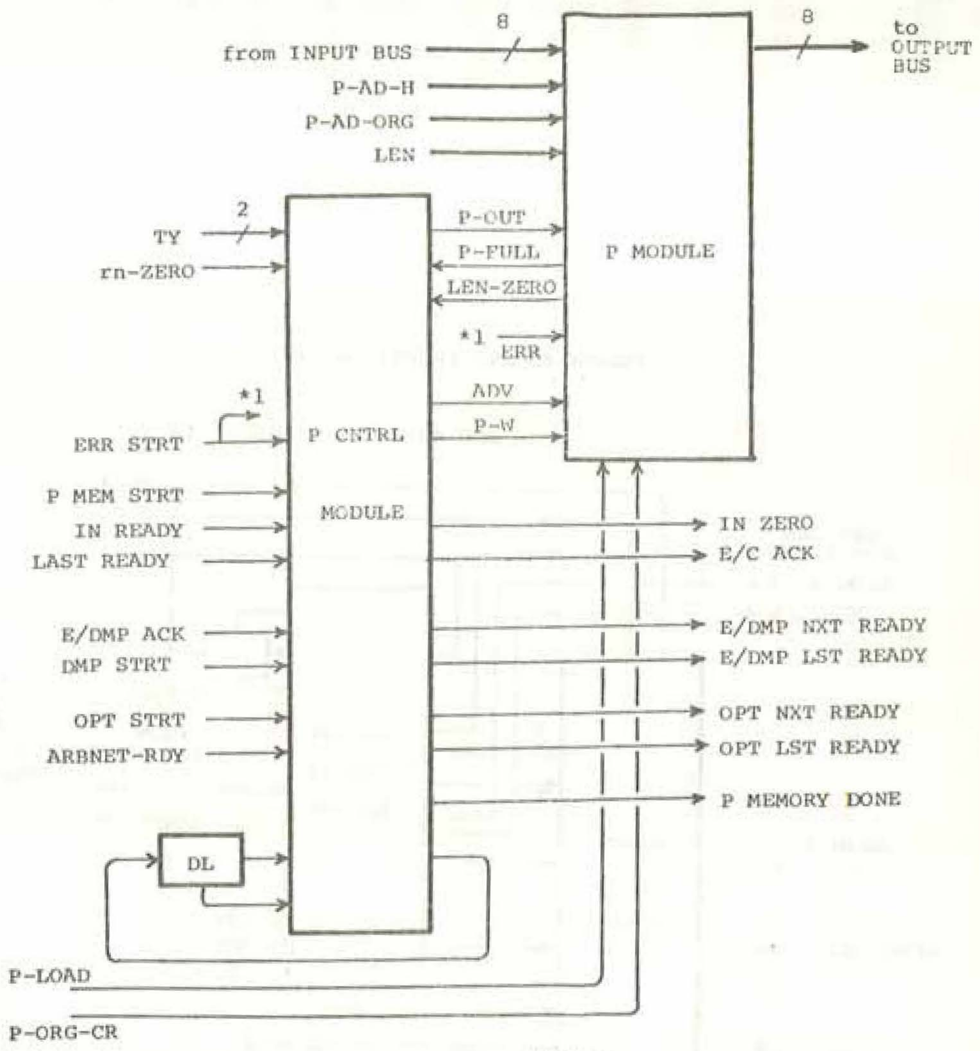
Bit configuration

Fig. 3.8 The Configuration of the U&F-SP Module



(b) P Module

Fig. 3.9 Configuration of the P Memory Module



(a) Wiring of P Memory Module

We have now completed the design of the data-flow structure of the Cell Block Module. The next section explains the design of the corresponding control structure.

3.2 The Design of the Control Structure

As for the data-flow structure, the control structure of the Cell Block Module is developed step by step following the top-down approach, resulting in a modular structure.

The original abstract specification in ADL (Appendix 1) is translated into a representation utilizing Petri Nets (Appendix 2), and the goal of the design activity with respect to the control structure is described in terms of the Petri Net representation as follows:

Given the Petri Nets representing the original abstract specification, the Petri Nets are decomposed into a modular system with each module also represented by a Petri Net. The resulting Petri Nets will be augmented with the operations concerning the characteristics of the actual devices.

Each module of the resulting modular system is further decomposed into submodules acquiring more datails. This procedure is repeated until each of the resulting modules becomes simple enough to be easily implemented from the detailed Petri Nets, by Patil's realization scheme.

Priorities between Control Sequences

At the outset of the design process, the priorities associated with the execution sequences in the Cell Block Module, as given in the ADL specification, are examined and incorporated in the control structure.

Four processing sequences are distinguished in the origial abstract specification: two of them are the E/C SEQUENCE and the OPT SEQUENCE, and the other two sequences are a sequence, called INIT SEQUENCE, which performs initialization operations for the Module upon the arrival of a reset signal, and a sequence, called RUN SEQUENCE, which changes the running state of the Module upon the receipt of a run packet.

Each of the four SEQUENCEs is activated independently upon arrival of the corresponding signal or packet.

The priorities given by the specification are as follows:

The reset signal has priority over other signals and packets in the sense that the arrival of the reset signal forces the Cell Block Module into a state in which the INIT SEQUENCE is executed. This transfer must be performed immediately on the arrival of a reset signal regardless of the operations being executed in the Cell Block Module. As a result, if the Cell Block Module is processing a packet, part or all of the information conveyed by the packet may be lost upon the arrival of the reset signal.

Once this state is reached, the INIT SEQUENCE is executed to reset every cell state.[*1]

Processing run packets may be inhibited by the reset signal, but it takes precedence over the processing of other signals. On the arrival of a run packet at the Module, the running state is switched over from the enabled state to the disabled state or vice versa according to the type of the run packet. This switching, however, must not interrupt the E/C SEQUENCE and the OPT SEQUENCE, if one or both of them are in progress, to avoid destroying any information being precessed. By inhibiting further processing of packets by these two sequences (see below), all sequences in progress eventually terminate. The state switching is then carried out.

On receipt of an enabling run packet the running state is switched over to the <u>enabled</u> state. An event/command packet or an operation packet may be processed in this state.

On receipt of a disabling run packet, the running state is switched to the disabled state. The transmission of an operation packet is inhibited in this state, while event/command packets are processed. The <u>disabled</u> state of the running state is automatically achieved after the INIT SEQUENCE is

---

[*1] The abstract Cell state is reinitialized although it may involve resetting only certain memory words.

executed.

An E/C SEQUENCE or an OPT SEQUENCE in progress may be interrupted and terminated by a reset signal. Further initiation of these sequences are inhibited by a run packet which disables the Cell Block Module, although any such sequences in progress is allowed to run to completion.

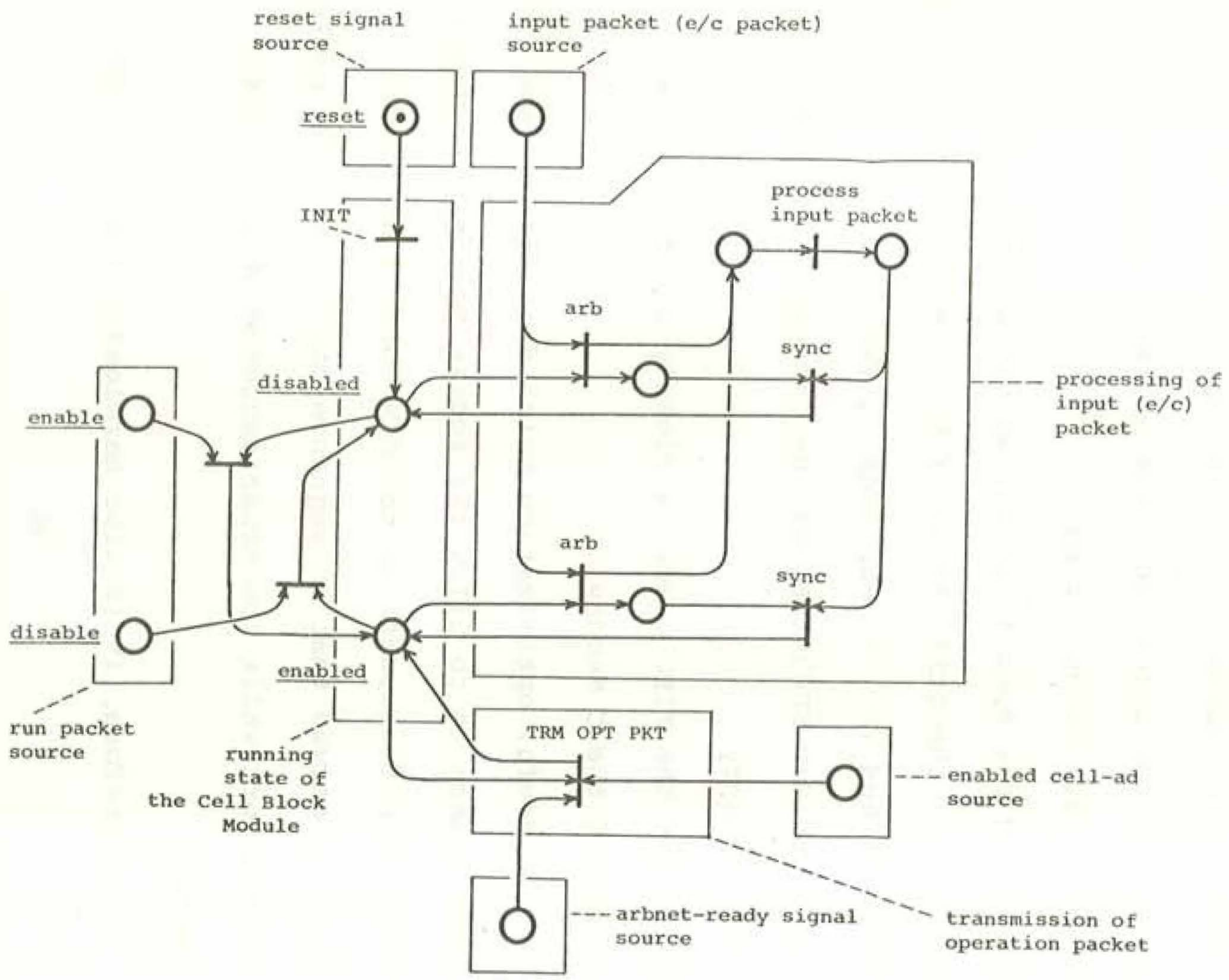
The behavior of the Cell Block Module as governed by these priority rules can be represented by a Petri Net as shown in Fig.3.10. The hardware module implementing this strategy is called the RUN CONTROL module and is easily realized using arbiters and other conventional devices.

Resolution of Conflicts

Our design allows the concurrent execution of the E/C SEQUENCE and the OPT SEQUENCE. They may access several sections of the Cell Block Module simultaneously, generating conflicts in these sections. These conflicts must be resolved to ensure correct module behavior.

There are four such sections which are introduced in the specification or by the characteristics of the chosen devices: the packet output port, the C Memory, the P Memory, and the FIFO queue.

The packet output port is utilized by the OPT SEQUENCE for transmission of operation packets and by the E/C SEQUENCE for


reset signal source
input packet (e/c packet) source
reset
INIT
process input packet
arb
sync
processing of input (e/c) packet
disabled
enable
arb
sync
disable
enabled
run packet source
running state of the Cell Block Module
TRM OPT PKT
enabled cell-ad source
arbnet-ready signal source
transmission of operation packet

arb ---	arbitration
sync ---	synchronization
INIT ---	initialization
TRM OPT PKT ---	transmission of operation packet

Fig. 3.10 A Petri Net Representation of Run Control Module

the transmission of DUMP packets or ERROR packets. In the specification, the packet output port is defined to serve both sequences, yielding the possibility of conflicts.

The P and C Memories are selected from conventional LSI memory devices. None of them allow simultaneous access, and hence yield the possibility of conflicts.

Regarding the FIFO queue, there already are several types of asynchronous FIFO queues available today, which are capable of processing two concurrent accesses to them, one at the input port and another at the output port.

By examining the specification, we can determine some characteristics of the utilization of these shared resources by the E/C SEQUENCE and the OPT SEQUENCE: (Table 3.1)

(i) Both the E/C SEQUENCE and the OPT SEQUENCE always use the C Memory.

(ii) Usage of the FIFO queue is closely related to the usage of the C Memory.

A useful optimization which does not degrade performance is to allow the access to the FIFO queue only when the access to the C Memory is granted.

(iii) It is observed that no sequence can utilize the packet output port while the other sequence utilizes the P Memory.

Therefore, it is also harmless and helpful to

	C Memory	FIFO queue	P Memory	Output port
OPND	○	△	○	⧄
ACK	○	△		
SET-INSTR	○	△	○	⧄
SET-ACK	○	△		
SET-VAR	○	△*2		
SET-CON	○	△	○	⧄
DUMP	○	⧄	○	○
OPERATION	○	○	○	○
ERROR			*1	○

○	Always utilized.
△	Not always, but utilized if the associated conditions are satisfied.
⧄	Never utilized, but no way to be utilized anyway.

*1 Input port is secured. Impossible to utilize the input port for processing the succeeding packets, anyway.

*2 Actually never utilized (result packet arrives later). But specified in the Specification.

Table 3.1 Interrelations between the Resources with respect to E/C and OPERATION Packets

simplify the situation by specifying that the packet output port is accessible to one of the SEQUENCEs only when it is allowed to utilize the P Memory.

With these additional constraints the conflicts are resolved without deadlocks, as shown in Fig.3.11 by a Petri Net, by utilizing two arbiters one for coordination at the C Memory and the other for the P Memory.

The hardware module which implements this behavior is called the RESOURCE ALLOCATION module.

RESOURCE ALLOCATION Module

The E/C SEQUENCE and the OPT SEQUENCE must be granted permission to use the C Memory to begin operation.

The E/C SEQUENCE, then, reads the cell state from the C Memory, which may be sufficient to process certain types of packets, and, if necessary, requests the RESOURCE ALLOCATION Module for using the P Memory. The module checks if the P Memory is utilized by the OPT SEQUENCE and grants permission to the E/C SEQUENCE when the P Memory is released by other sequences.

The E/C SEQUENCE, then, makes use of the memory, and completes its operation. Each of the memories may be released independently on the completion of the corresponding shared resources in each sequence.

If a ready signal from the Arbitration Network is asserted and there is at least one enabled cell, the OPT SEQUENCE waits for the release of the C Memory, and secures the C Memory upon receipt of a grant signal issued by the RESOURCE ALLOCATION Module. The OPT SEQUENCE then waits until the P Memory is released by the E/C SEQUENCE and starts its operations.

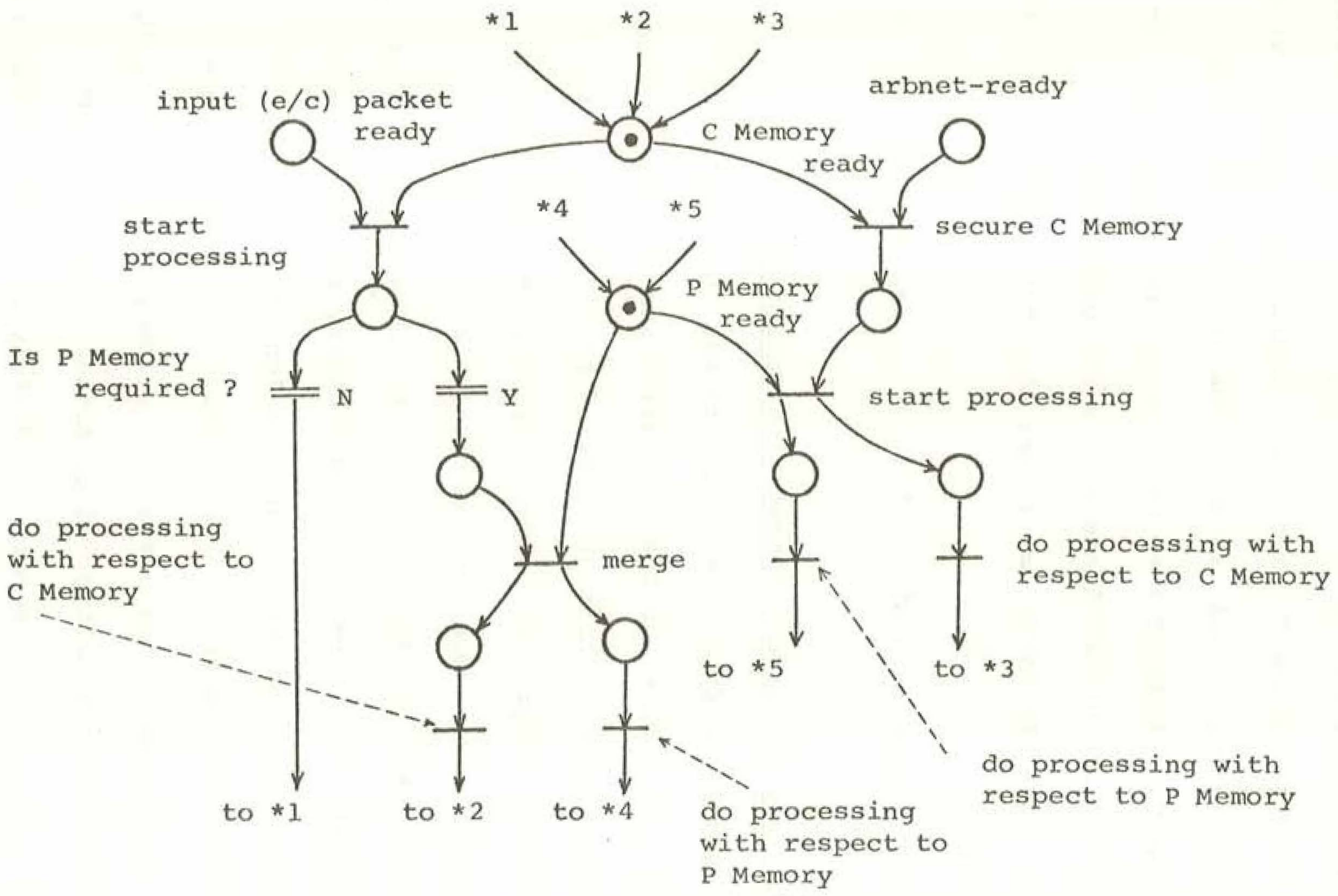
Thus, both SEQUENCEs can access the shared resources without deadlocks.

This resource allocation strategy is depicted in Fig.3.11.

Exploiting the Concurrent Computation Capability

The completed data-flow structure has the capability of performing the update and the initialization operations of a cell state in a very short time. Based on this characteristic, the concurrent computation capability may be exploited.

Consider the scheme for the resolution of conflicts, as explained in the preceding section. The OPT SEQUENCE starts packet transmission using both the C Memory and the P Memory at the same time. However, it takes a long time to complete its sequential memory access to the P Memory. The total access time to the P Memory includes as the primary factor the cycle time of the P Memory multiplied by sixteen, while the OPT SEQUENCE releases the C Memory immediately after the completion of its operations on the C Memory. Therefore, there is a long



Remark: The path passing through the link *5 consumes the longest time. Thus the path passing through the link *1 can be executed while P Memory is utilized by the transmission sequence.

Fig. 3.11 Resource Allocation Strategy

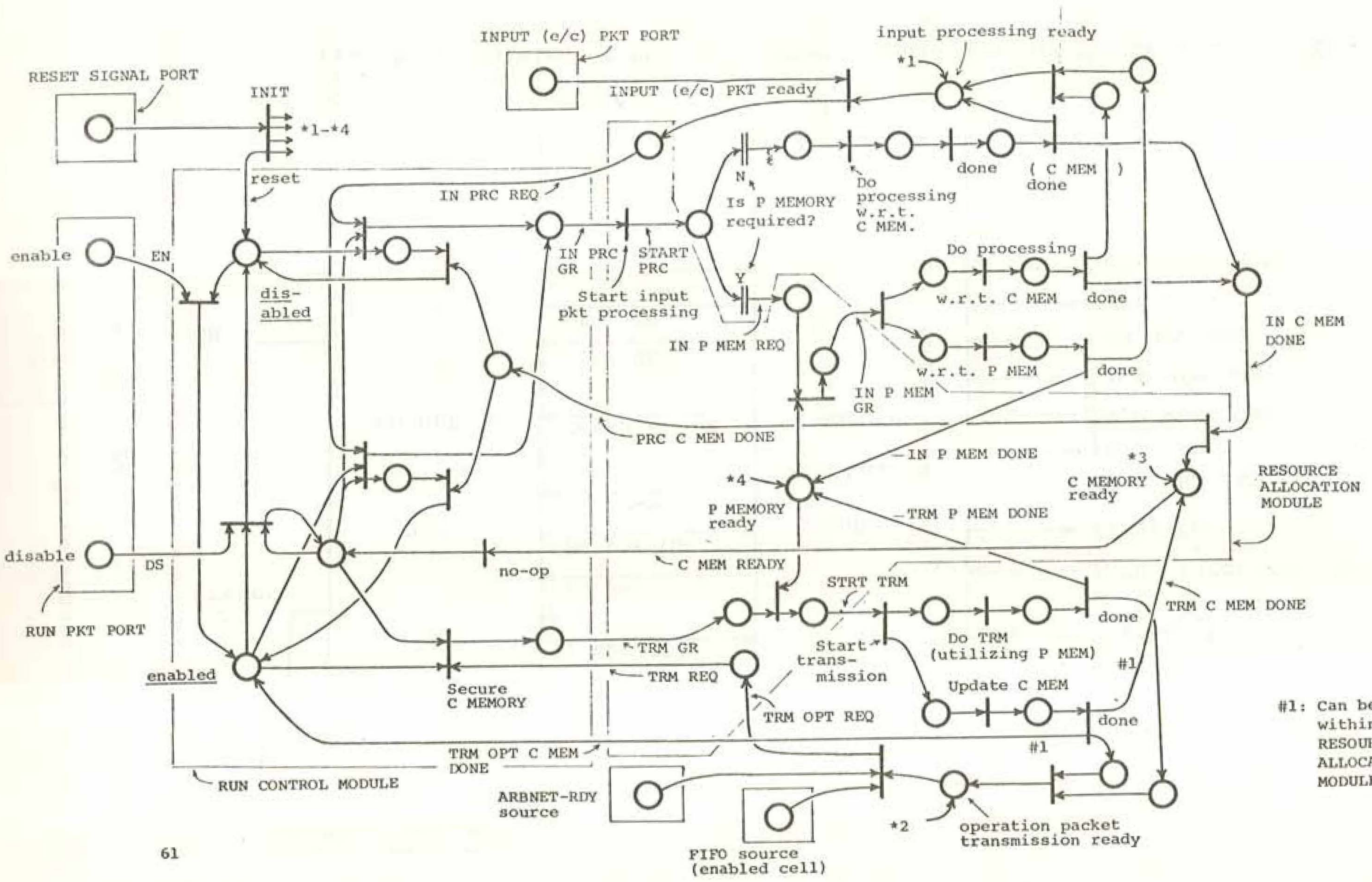
gap in time between the release of the C Memory and that of the P Memory by the OPT SEQUENCE.

The E/C SEQUENCE, then, can process packets in parallel by taking advantage of this time gap in the OPT SEQUENCE's demand on the C Memory, as long as each of the packets can be processed without using the P Memory. This time gap is estimated to be long enough to process three ACK packets.

This scheme still suffers from limitations due to the lack of parallel accessing capability in conventional memory devices. Even though the C Memory is often available to the E/C SEQUENCE, it is not able to take advantage of the opportunity if it processes a packet which requires accessing the P Memory.

More parallelism can be incorporated if the configuration of the P Memory consists of multiple memory devices to allow parallel accesses, rather than just one as shown in this design. This becomes clear when we consider the execution of a data-flow program.

During the normal execution of a program, only three types of packets are in the entire architecture, namely the OPND, the ACK, and the OPERATION packets. Assume that an OPND is received at the event/command packet port when an OPERATION packet is under transmission, and that the destination cell of the OPND packet differs from the cell address from whose contents the OPERATION packet is formed (which has very high probability



#1: Can be ramified within the RESOURCE ALLOCATION MODULE ated with a control module, and operated only through the control signals issued by the control module. Each of the control modules receives command signals specifying the sequence to be executed and initiates the activation of the sequence.

This configuration has the following advantages. Firstly each module can be accessed from both the E/C SEQUENCE and the OPT SEQUENCE. Therefore, both sequences need not have separate mechanisms to control the same data-flow module. Obviously, the number of hardware devices is reduced.

Secondly, the number of control signal lines, connecting a data-flow module and the associated control module, remains almost the same whether the control module is expected to serve only one sequence or several sequences. The obvious reason is that the complexity associated with the capability of executing several sequences is transformed into the complexity of programs memorized by PLAs or ROMs, and does not appear in the form of the increased number of devices and connections. In addition, if PLAs are utilized in the implementation, even the execution speed will not be slowed down significantly by complexity. Therefore, the interconnection is simplified and will result in the reduction of the number of control signal lines.

Thirdly, a control module never receives more than one command signal at a time due to the mutual exclusion which is guaranteed by the RESOURCE ALLOCATION Module, and therefore no mechanism to coordinate the requests is necessary. A control module must return to the sequence generating the command signal a done signal on the completion of operation. The acknowledge scheme can be implemented correctly by broadcasting done signals to all modules, because there exists only one such SEQUENCE. Therefore, no other coordination mechanism is necessary.

The fourth advantage is related to the second one. The INIT SEQUENCE, which has not been examined until now, is easily incorporated into this structure at the cost of slightly increased complexity of programs stored in some of the control modules.

Employing the configuration discussed above, the E/C SEQUENCE and the OPT SEQUENCE must perform the following : 1) to request permission to utilize shared resources, 2) to issue command signals to the appropriate control modules, 3) to release the shared resources on the completion of the corresponding operations, and 4) to return to an initial state for further packet processing. This behavior is described in the Petri Nets shown in Fig. 3.14 through Fig. 3.20.

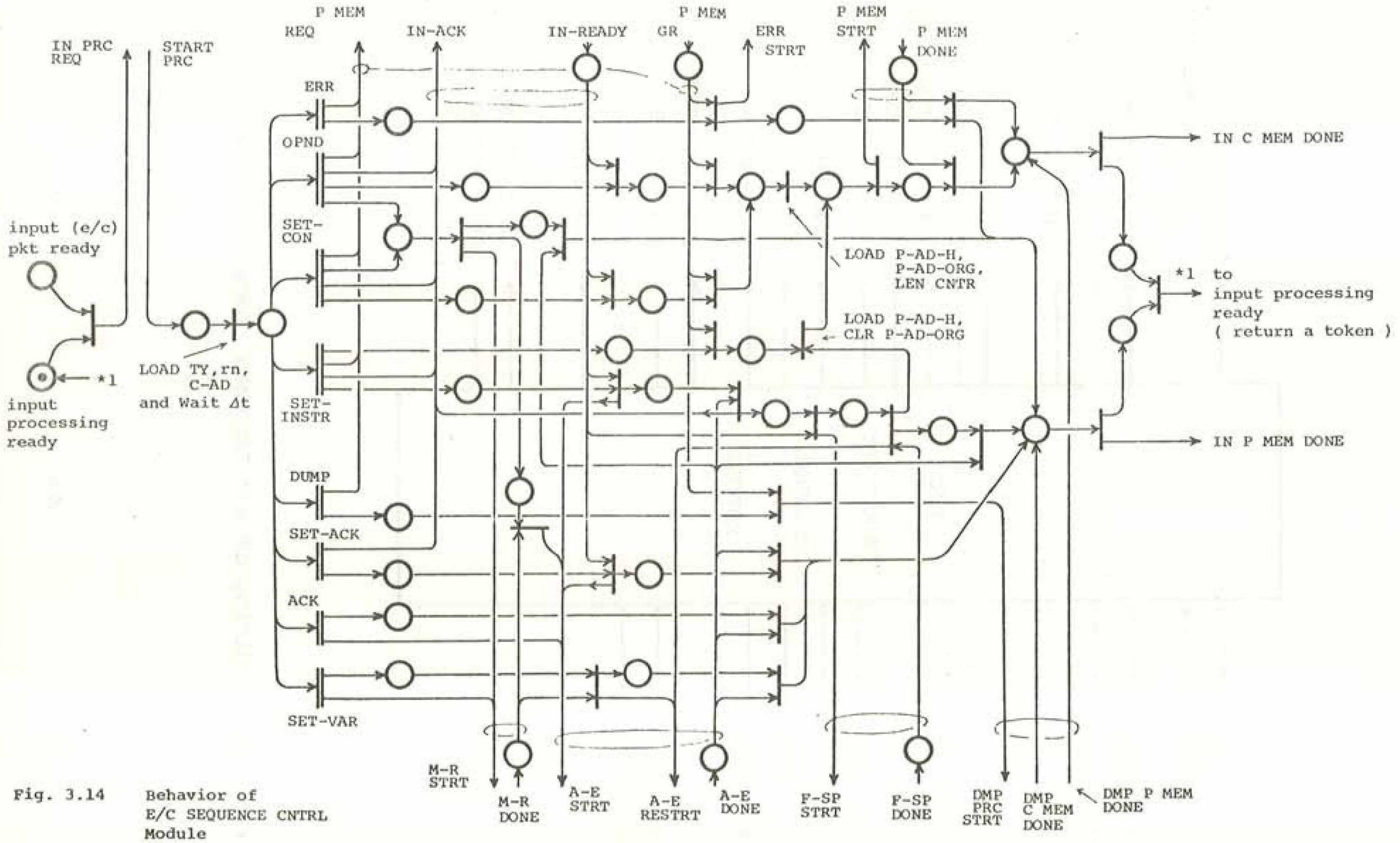


Fig. 3.14 Behavior of E/C SEQUENCE CNTRL Module

65

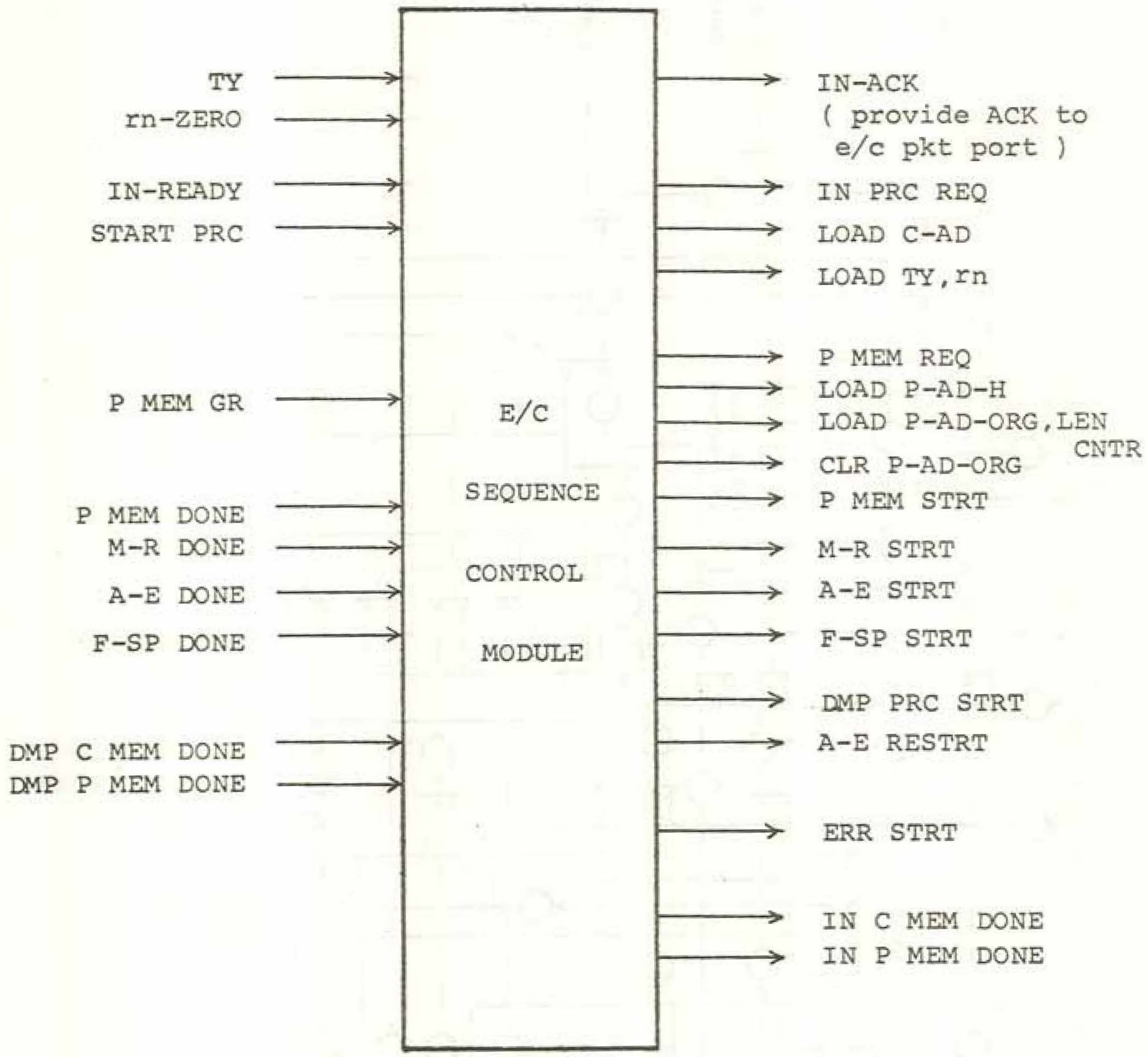
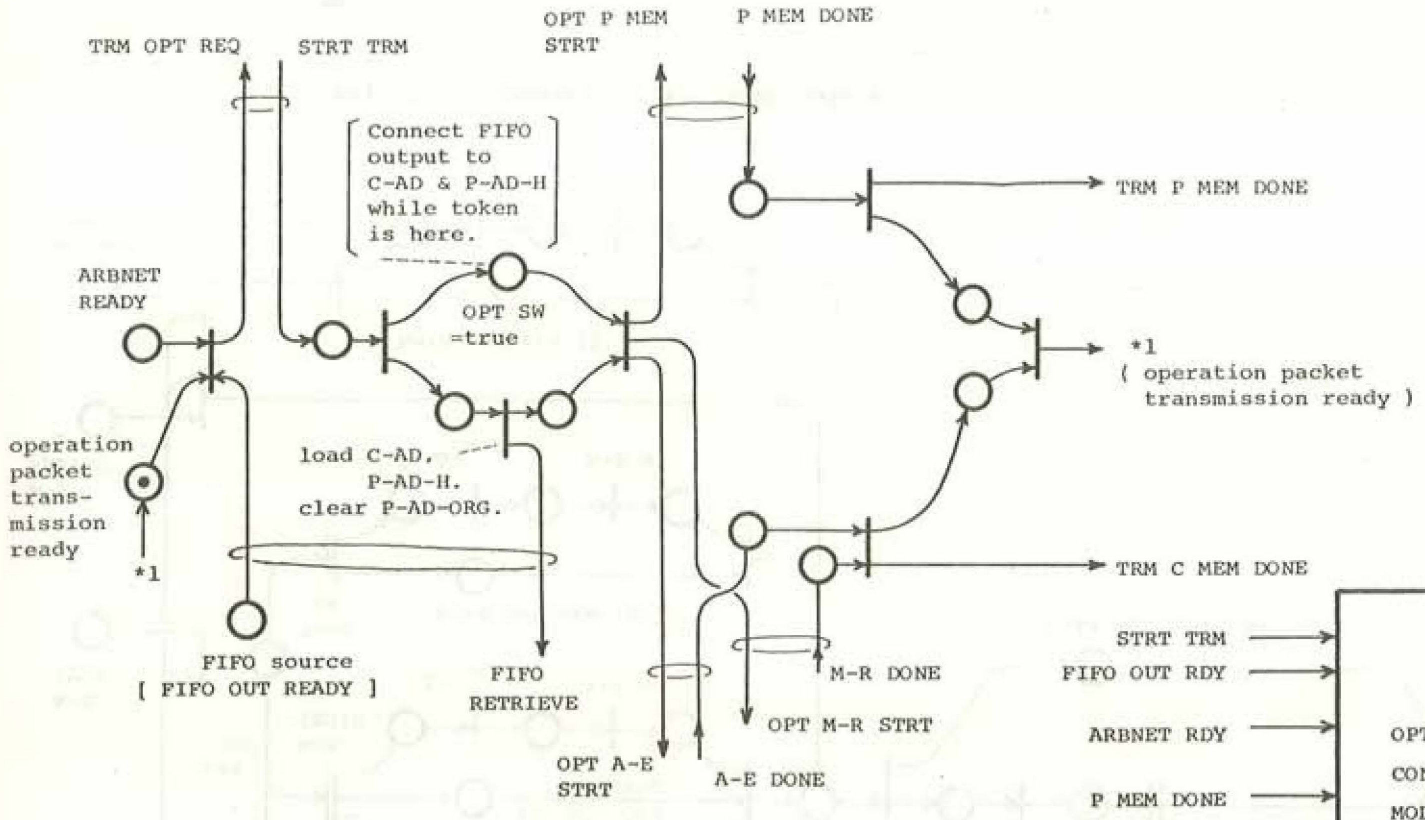


Fig. 3.15 Wiring of E/C SEQUENCE CONTROL Module


TRM OPT REQ
STRT TRM
OPT P MEM STRT
P MEM DONE
Connect FIFO output to C-AD & P-AD-H while token is here.
ARBNET READY
OPT SW =true
TRM P MEM DONE
*1
(operation packet transmission ready)
operation packet trans-mission ready
*1
load C-AD, P-AD-H. clear P-AD-ORG.
FIFO source [FIFO OUT READY]
FIFO RETRIEVE
OPT A-E STRT
A-E DONE
OPT M-R STRT
M-R DONE
TRM C MEM DONE

(a) Behavior of OPT TRM CONTROL Module

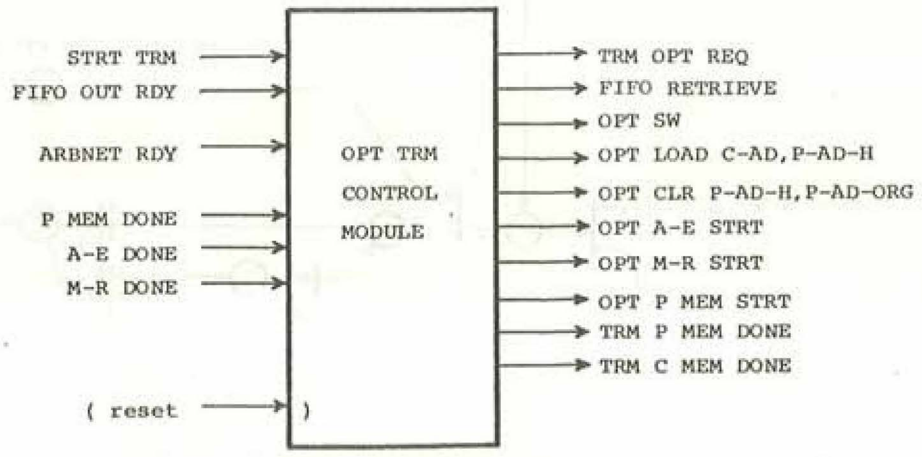


Fig. 3.16 Behavior of OPT SEQUENCE Control Module

(b) Wiring of OPT TRM CONTROL Module

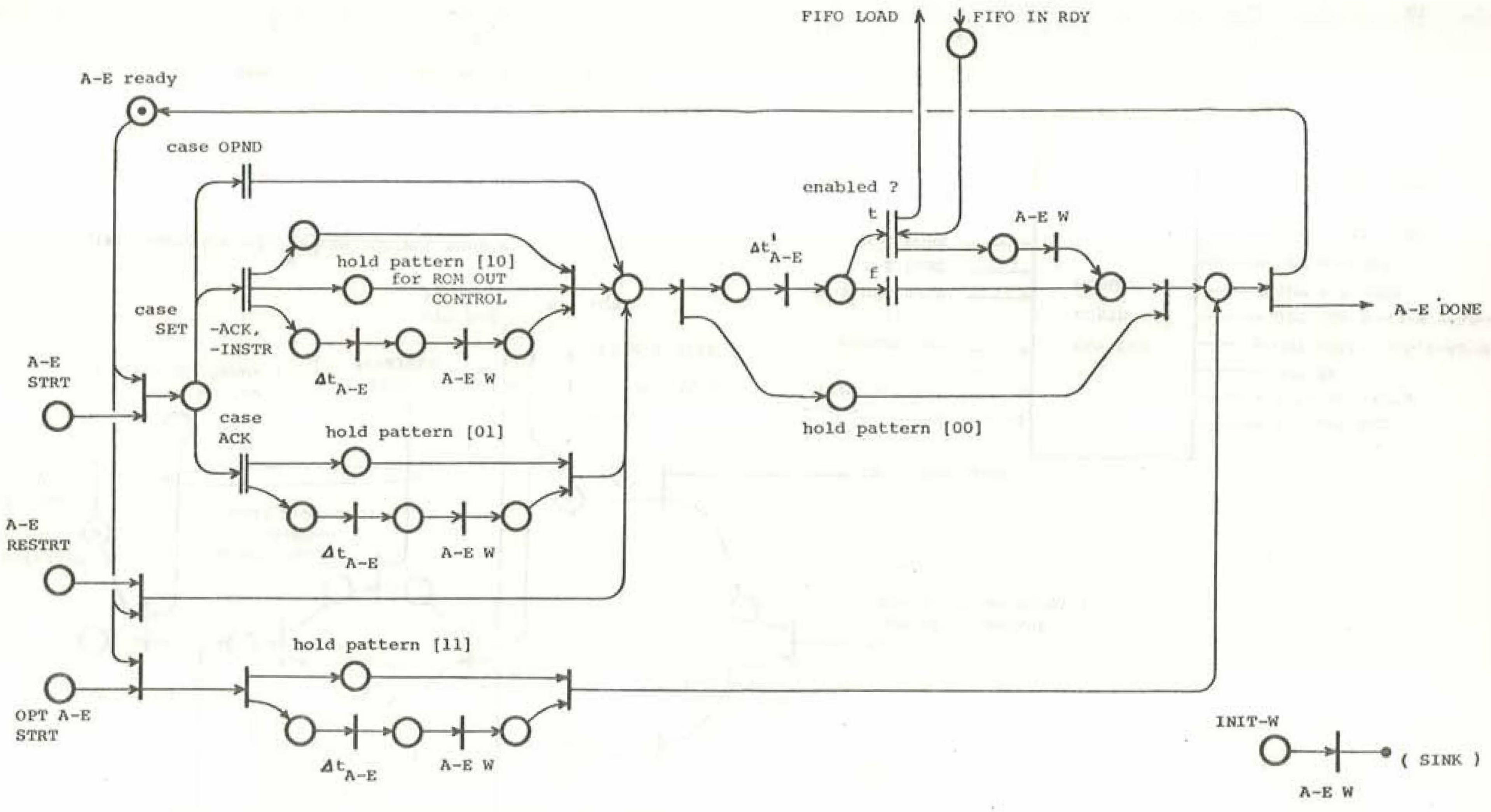


Fig. 3.17 Behavior of A-E CNTRL Module

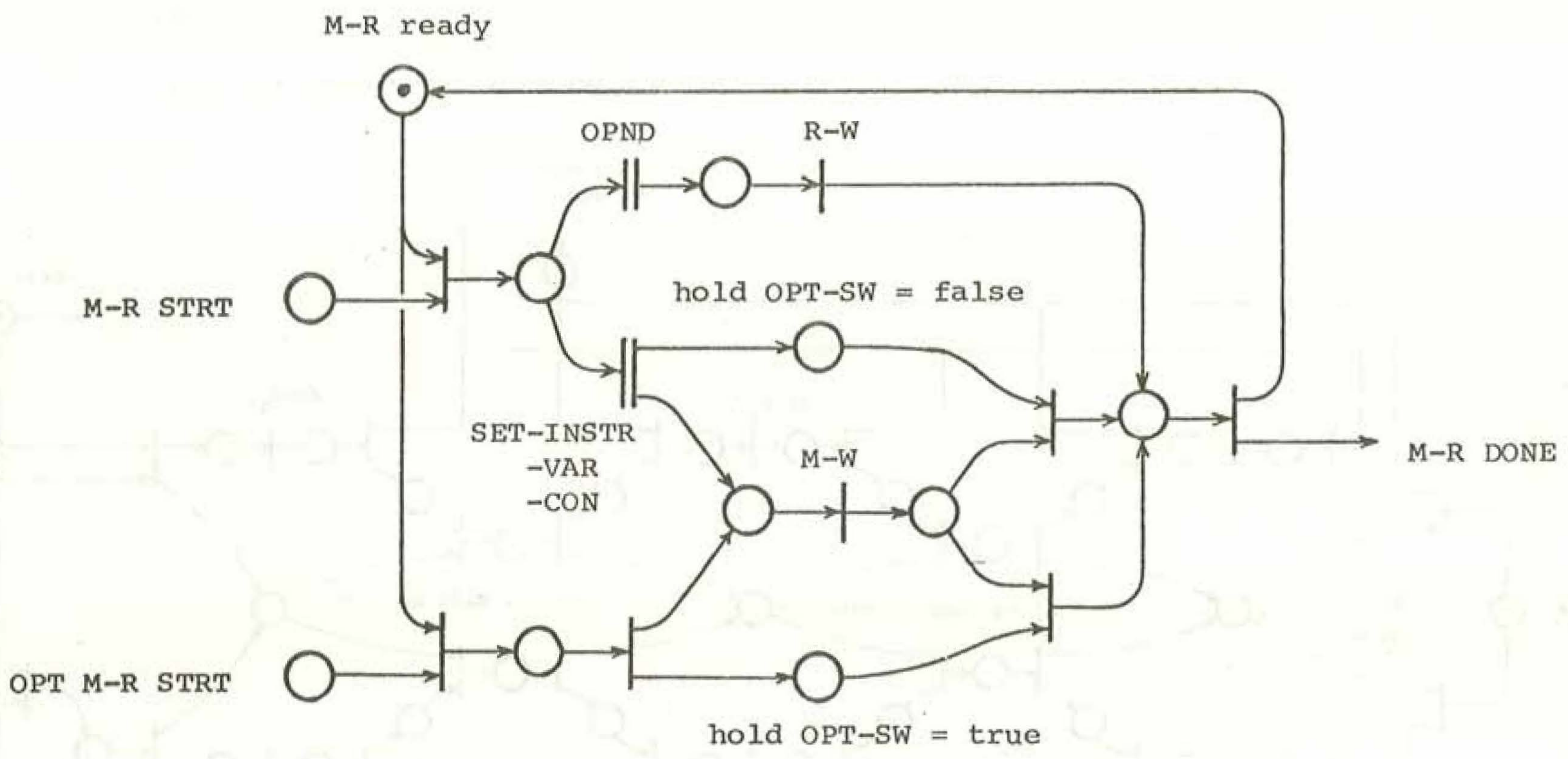


Fig. 3.18 Behavior of M-R CNTRL MODULE

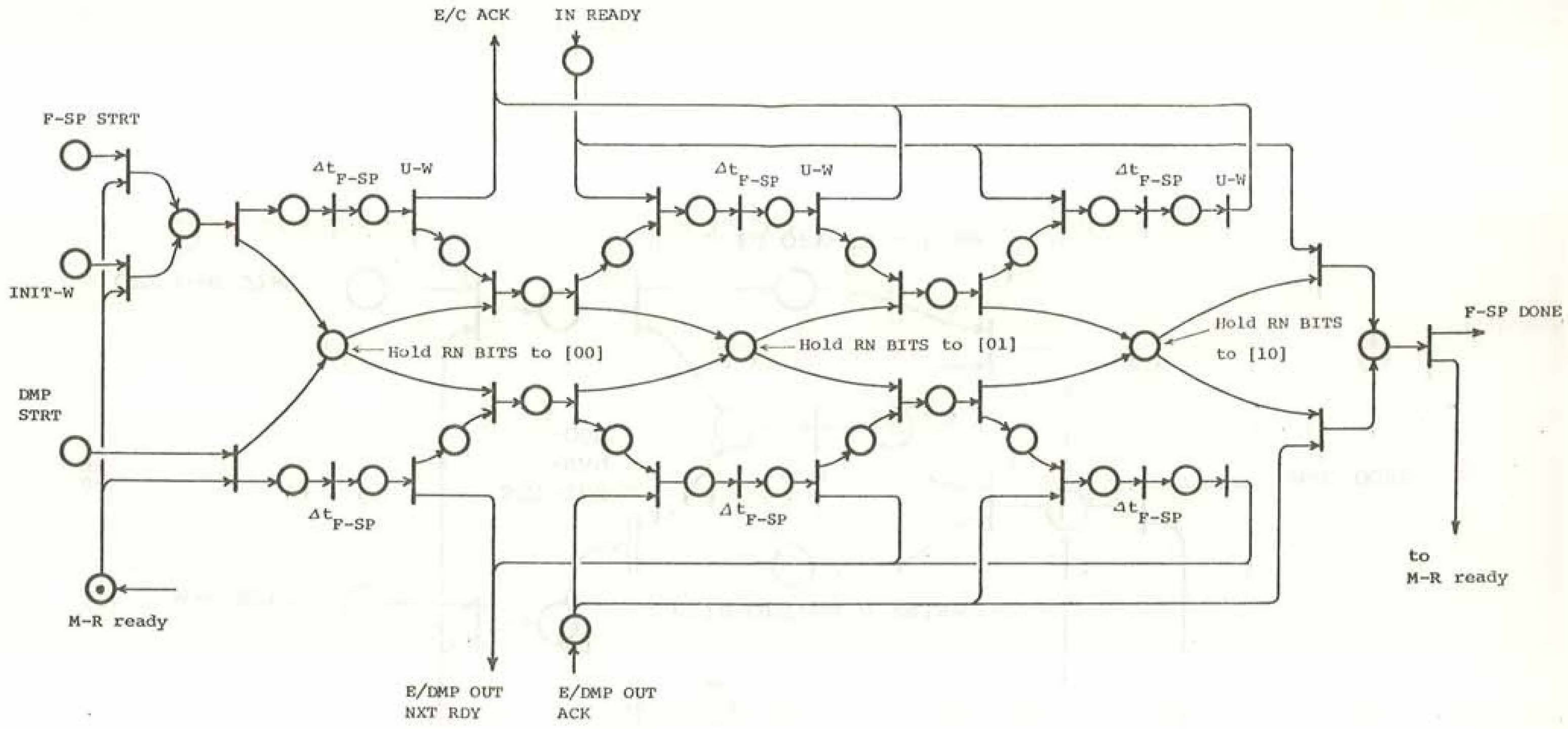


Fig. 3.19 Behavior of U&F-SP CNTRL Module

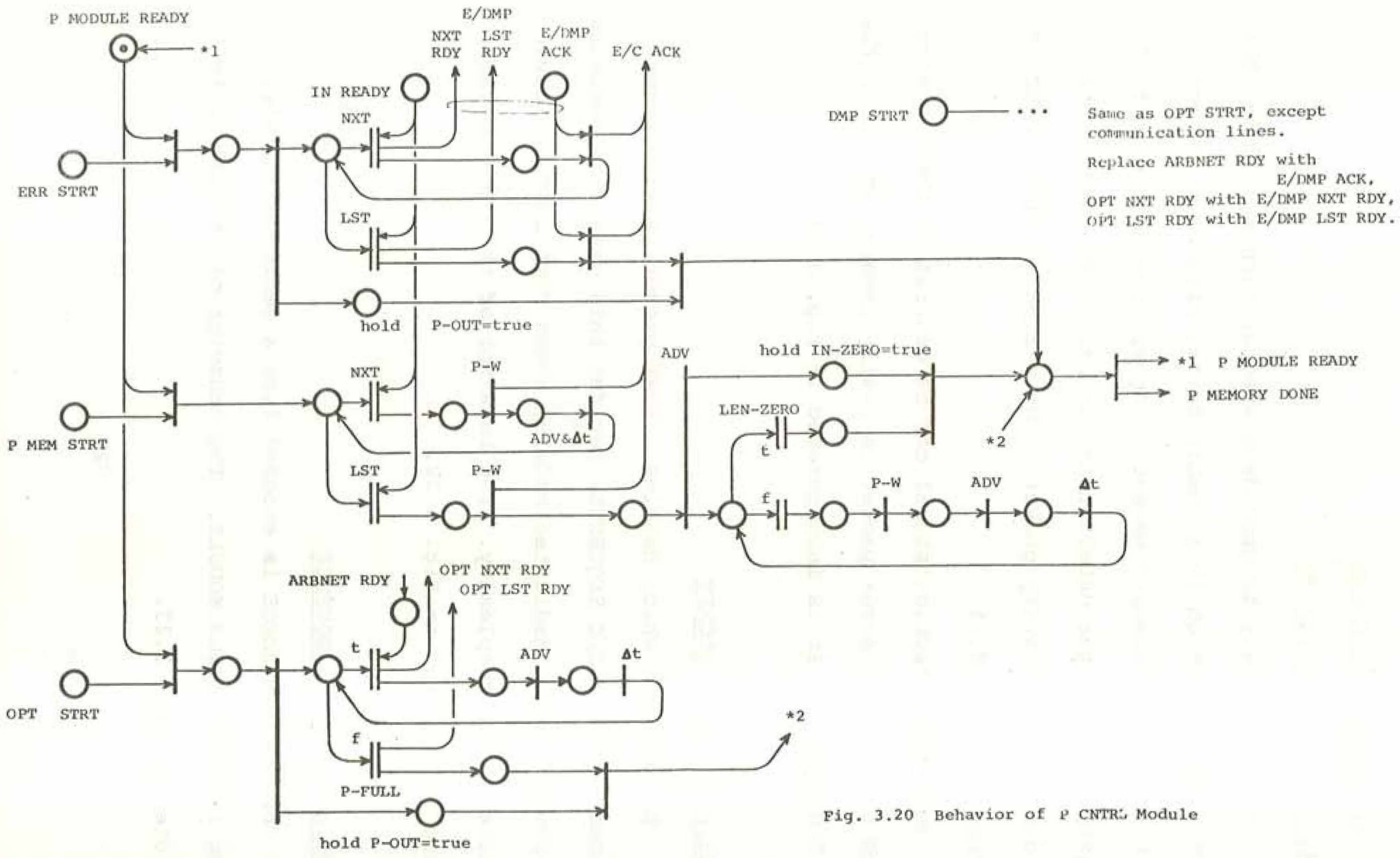


Fig. 3.20 Behavior of P CNTRL Module

3.3 Miscellaneous Cases

Handling of ERROR PACKET

It is necessary to provide a special purpose module, the ERROR DETECTOR, which is a combinatorial circuit to detect whether an event/command packet is an error packet or not. It is expected that the detector is realized with PLA which is encoded to detect every packet. The wiring of ERROR DETECTOR is shown in Fig. 3.21.

Once the detection signal can be generated, the sequence which handles the error packet is easily incorporated into the E/C SEQUENCE. (It is incorporated in Fig. 3.14.)

Handling of DUMP PACKET

The sequence which handles a DUMP packet is also incorporated into the E/C SEQUENCE, but for this case, the sequence is encoded into a dedicated module, named DUMP CONTROL Module, because of its complexity. The behavior of the DUMP CONTROL Module is depicted in Fig. 3.22.

Handling of INIT SEQUENCE

The INIT SEQUENCE is encoded into a dedicated module, named INITIALIZATION MODULE. The behavior of the module is depicted in Fig. 3.23.

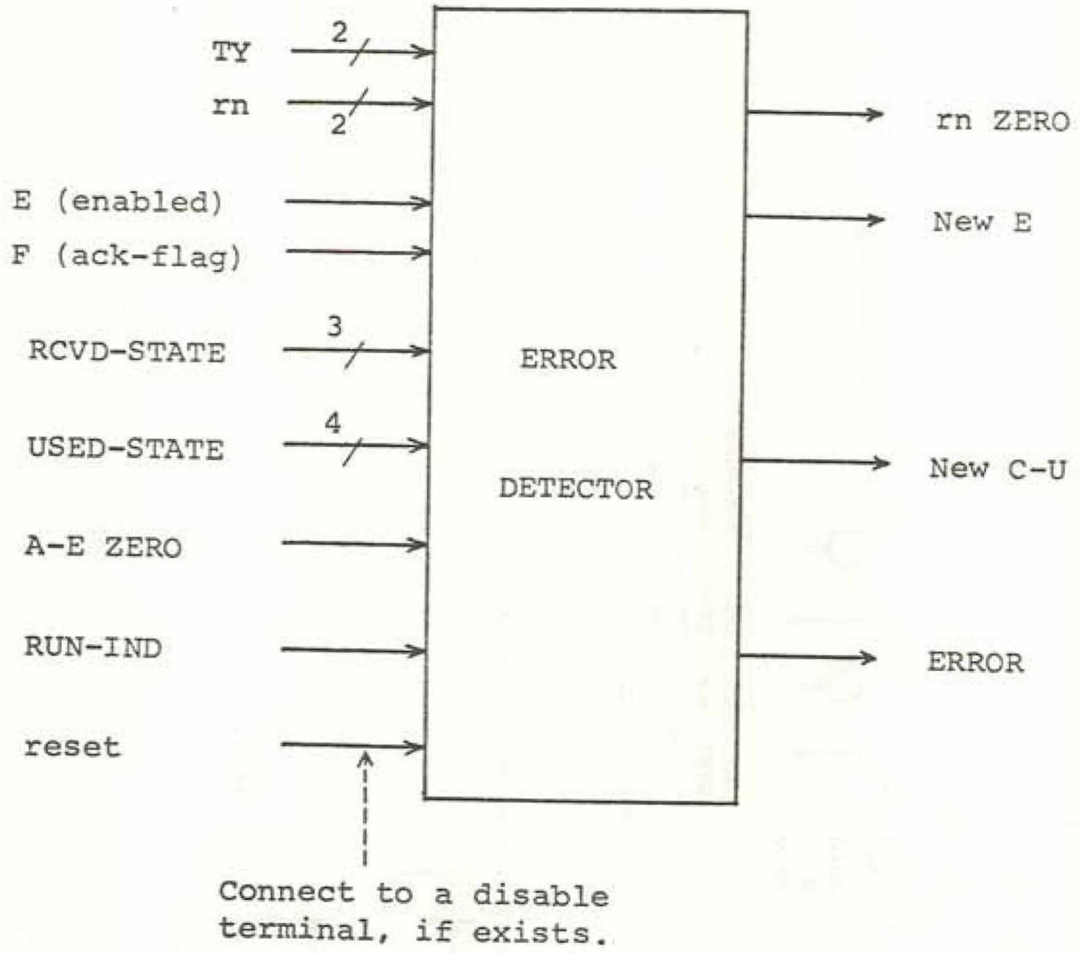


Fig. 3.21 Wirings of ERROR DETECTOR Module

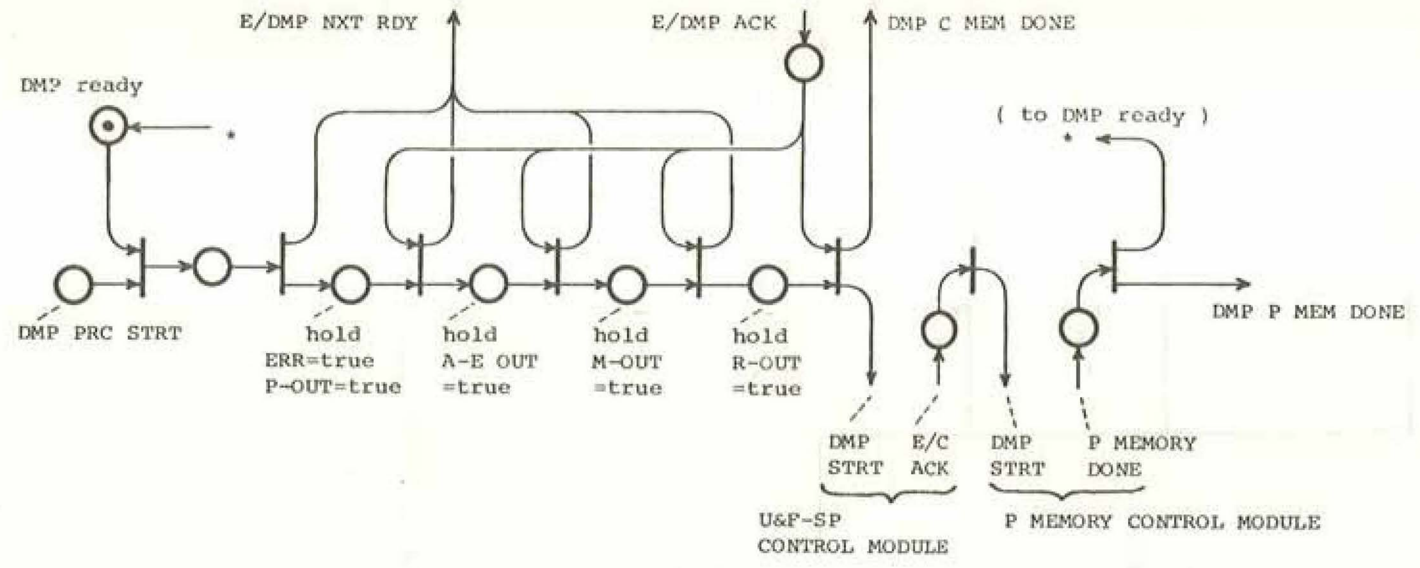


Fig. 3.22 Behavior of DMP Control Module

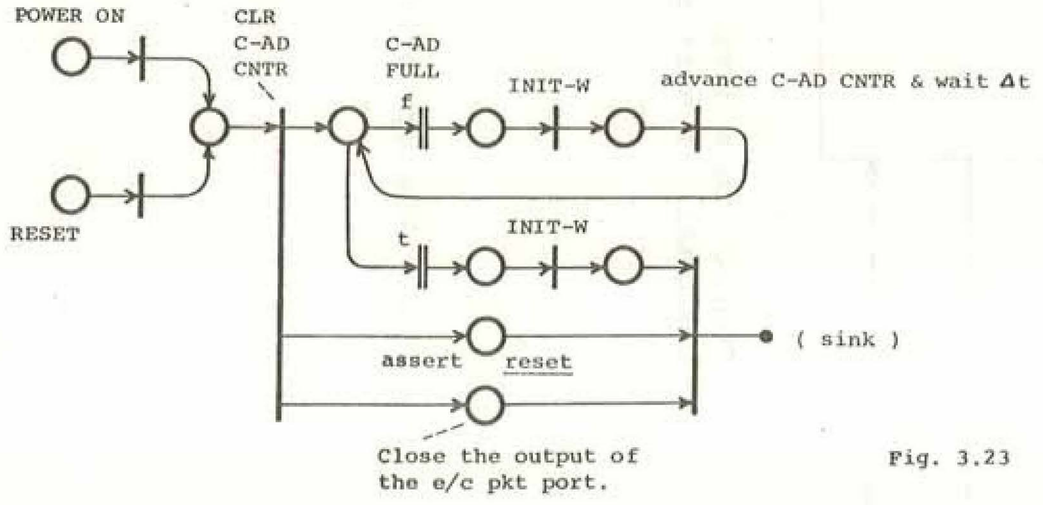


Fig. 3.23 Behavior of Initialization Control Module

74

An additional circuit section which generates a signal IN-READY is depicted in Fig.3.23.

We have completed the design of the control module, and combined it with the data-flow structure developed in Sec. 3.1 to construct the complete Cell Block Module. This proposed design is evaluated in the next chapter.

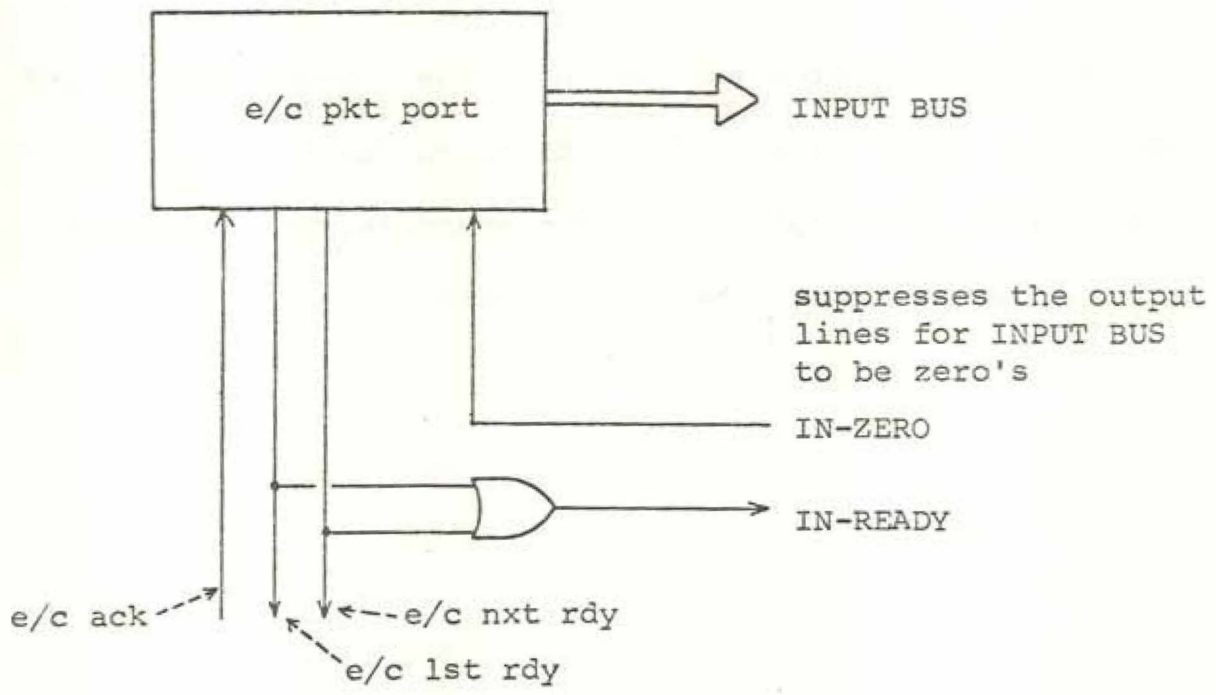


Fig. 3.23 WIRINGS OF E/C PACKET PORT

Chapter 4

4. Performance Analysis

There are several parameters which determine the cost performance of a processor, such as its architecture, power consumption, physical size, the number of devices and their operating speed. In our design for the Cell Block Module, the control structure is not implemented with random logic, but is assumed to be constructed from PLA's (Chapter 2). The actual construction may result in any of a number of hardware configrations depending upon the specific PLA's to be utilized. There is a significant difference between the various configrations regarding speed, power consumption, size and the number of devices. In this section we will only analyze the performance of our proposed design based on time estimate for executing the various control functions and data operations. Consideration of other factors are postponed until a specific implementation of the control structure is chosen.

The operation speed of the entire Cell Block Module can be estimated by estimating the operation speed of the various control structures and data operations. The latter is easier since the data-flow structure is given in terms of memory modules, counters, registers, and FIFO buffers. Every control sequence at the lowest level is described by a Petri Net whose events are the data operations. We estimate the operation speed of each sequence by postulating a reasonable time estimate for the control functions necessary and calculating estimates for the data operations from data on the operation speed of such devices as memory modules, counters, registers, and FIFO buffer. Since the design does not specify particular devices available on the market, an average speed for these devices is used.

The estimation is as follows:

an access to memory devices and FIFO queue (a bipolar speed is assumed.)	--- 50 ns
an operation by other functional devices in the data-flow structure	--- 20 ns
each action at the RESOURCE ALLOCATION Module such as <u>IN</u> <u>PRC</u> <u>RQ</u>	--- 30 ns
transmission delays at the event/command port and the packet output port	--- 20 ns

Because the operation speed of the Cell Block module for the normal program execution is of the utmost importance, we shall only provide the performance estimate for the processing of packets associated with program execution.

Assuming that both the C Memory and P memory are available when accessed:

type OPND case: 1090 ns

This estimate includes 7 x 70 ns for receiving the 7 bytes of oprand data and storing them into the P Memory. The other necessary operations are the extraction of a cell address from the operand packet, retrieval of the starting location and the operand register length in bytes from the C Memory and setting them up in counters and the memory address register of the P Memory, updating the contents as the operand is stored.

type ACK case:	420 ns	if the ACK packet enables the corresponding cell
	320 ns	otherwise

An operation packet is transmitted by executing the OPT SEQUENCE, its estimated execution time is, again assuming that no conflict at the C Memory or the P Memory arises:
type OPERATION case: 1820 ns

This includes 16 x 70 ns for fetching 16 bytes from the P Memory and transmitting them to the Arbitration Network. The rest of the time is consumed in accessing the cell state and setting up and maintaining counters and memory registers.

Assuming that 1) every Instruction Cell utilizes 2 operands and 3 ACK packets, and 2) every ACK packet is processed concurrently with the execution of the OPT SEQUENCE, the total time to process the enabling and firing of an Instruction Cell becomes 4000 ns (1820ns + 2180ns). Instead of 2), if we assume, as for the worst case, that 2)' no ACK packet is ever processed while the execution of the OPT SEQUENCE is performed, then the total time becomes 5060 ns (4000ns + 320ns + 320ns + 420ns). It may be reasonable to regard that half of the ACK packets are processed concurrently with the execution of the OPT SEQUENCE on the average, thus 4500 ns is obtained as the average execution time of an Instruction Cell.

Because of the conflict resolution, the OPT SEQUENCE or the E/C SEQUENCE is once in a while forced to wait before being granted access to the memory modules. Taking this into consideration, 5 μs is estimated to be the average execution time for an Instruction Cell, to transmit an operation packet, which can be restated as 0.2 MIPS (Million Instruction Per Second) for the throughput of the Cell Block Module as designed.

Therefore, a Memory Section consisting of, for example, 512 Cell Block Modules has a 20 (51) MIPS throughput when 20% (50%) of the Instruction Cells are always enabled.

It should be noted that the above values change significantly under different conditions, for example the number of operands per Instruction Cell, as well as the number of ACK packet.

Chapter 5

5. Conclusions

A logic design for the Cell Block Module has been completed, employing conventional commercial electronic devices available today.

The resulting architecture forms an asynchronous modular system which is derived from a top-down decomposition of the specification.

The configuration of modules, as well as the internal configuration of each module, is optimized to yield a structure as simple as possible. Conventional synchronous LSI devices are heavily employed in the modules, yet the modules behave asynchronously because of the use of delay lines allowing an asynchronous communication protocol to be simulated.

The resulting configuration is considered to be small in size for the relatively complex operations to be performed.

High Speed Computation Capability

Throughout the design process, the high speed computation capability is pursued, leading to an architecture which can process operation packets and certain types of event/command packets concurrently. This capability does help the resulting architecture to achieve a high speed which is estimated to be comparable to the projected speed in the specification note [ ]. However, a higher degree of concurrency is achievable. Exploiting this concurrency fully can result in a faster computation speed, not available in a conventional scheme for many applications.

Improvement

The desired improvement in the processing speed can be obtained by employing a multiple memory configuration for the P Memory, so that the P Memory can be accessed by the OPT SEQUENCE and the E/C SEQUENCE concurrently provided the cell address dealt with by each SEQUENCE are different.

Because the conflicts at a cell address by the two SEQUENCEs will rarely occur under normal program execution, event packets can be processed almost in parallel. The overhead at the RESOURCE ALLOCATION Module may introduce problems.

BIBLIOGRAPHY

1. Dennis, J.B., " Modular, Asynchronous Control Structures for a High Performance Processor," Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York, 55-80 (1970).

2. Dennis, J.P., " First Version of a Data Flow Procedure Language," Lecture Notes in Computer Science 19, Springer-Verlag, New York, 326-376 (1974).

3. Dennis, J.B., " Packet Communication Architecture," Proceedings of the 1975 Sagamore Conference on Parallel Processing, IEEE, 224-229 (August 1975).

4. Dennis, J.B., Leung, C.K., and Misunas, D.P., " Specification of the Instruction Cell Block for a Data Flow Procrssor," Data Flow Design Note 1, Computation Structures Group, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., December 1975.

5. Dennis, J.P., and Misunas, D.P., " A Computer Architecture for Highly Parallel Signal Processing," Proceedings of the ACM 1974 National Conference, 402-409 (November 1974).

6. Dennis, J.B., and Misunas D.P., " A Preliminary Architecture for a Basic Data-Flow Processor," Proceedings of the Second Annual Symposium on Computer Architecture IEEE, 126-132 (1975).

7. Dennis, J.B., Misunas, D.P., and Leung, C.K., " A Highly Parallel Processor Using a Data Flow Machine Language," submitted for publication.
Computation Structures Group Memo 134, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., January 1977.

8. Dennis, J.B., and Patil, S.S., " Speed Independent Asynchronous Circuit," Proceedings of the Fourth Hawaii International Conference on System Science, 55-58 (January 1971).

9. Dennis, J.B., and Weng, K.-S., " Application of Data Flow Computation to the Weather Problem," Computation Structures Group Memo 147, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., May 1977.

10. Misunas, D.P., " Petri Nets and Speed Independent Design," Communication of the ACM, Vol. 16, No. 8, 474-481 (August 1973).

11. Misunas, D.P., " Deadlock Avoidance in a Data-Flow Architecture," _Proceedings of the Milwaukee Symposium on Automatic Computation and Control_, IEEE, New York, April 1975.

12. Misunas, D.P., " A Computer Architecture for the Data-Flow Processor," S.M. Thesis, Department of Electrical Engineering and Computer Science, Mass. Inst. of Technology, Cambridge, Mass., June 1975.

13. Misunas, D.P., " Structure Processing in a Data-Flow Computer," _Proceedings of the 1975 Sagamore Computer Conference on Parallel Computation_, IEEE, 230-234 (August 1975).

14. Muller, D.E., " Asynchronous Logics and Application to Information Processing," _Switching Theory in Space Technology_, Stanford Press, Stanford, California, 1963.

15. Patil, S.S., " Coordination of Asynchronous Events," Technical Report 72, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., June 1972.

16. Patil, S.S., " Synchronizers and Arbiters," Computation Structures Memo 91, Laboratory for Computer Science, Mass. Inst. of Techno;ogy, Cambridge, Mass., October 1973.

17. Patil, S.S., " Cellular Arrays for Asynchronous Control," _Proceedings of the ACM 7th Annual Workshop on Microprogramming_, 178-185 (September 1974).

18. Patil, S.S., " Micro-Control for Parallel Asynchronous Computers," Computation Structures Group Memo 120, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., March 1975.

19. Patil, S.S., " An Asynchronous Logic Array," Technical Memo 62, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., May 1975.

20. Patil, S.S., " The Description and Realization of Digital Systems," Proceedings of the Sixth Annual IEEE Computer Society International Conference, San Francisco, September 1972.

21. Plummer, W.W., " Asynchronous Arbiters," IEEE Trans. on Computers, Vol. C-21, No. 1, 37-42, (January 1972).

22. Weng, K.-S., " Stream-Oriented Computation in Recursive Data Flow Schemas," Technical Memo 68, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., October 1975.

23. Course Notes on 6.032 Computation Structures. Department of Electrical Engineering and Computer Science, Mass. Inst. of Technology, Cambridge, Mass.

```
                /* Packet Definitions */

     /* An event packet, a command packet or a run packet is an input
     packet to a cell block */

event/command-pkt = union[ event-pkt, command-pkt ];

     /* An event packet is sent to the cell block from other modules
     of a data flow processor. It is either an operand packet or an
     acknowledge packet */

event-pkt = union[ opnd-pkt, ack-pkt ];

     /* A command packet is sent to the cell block from the host com-
     puter, for initializing four different ( not necessarily disjoint )
     sets of state variables and for dumping the state of a specified
     cell */

command-pkt = union[ set-cmnd-pkt, dump-cmnd-pkt ];

set-cmnd-pkt = union[ set-instr-pkt, set-ack-pkt, set-var-pkt, set-con-pkt ];

     /* A run packet enables or disables the transmission of operation
     packets by a cell block */

run-pkt = union[ enable-pkt, disable-pkt ];

     /* An operation packet, an error packet or a dump packet is an
     output packet of the cell block. An operation packet is formed
     from the contents of an enabled cell and sent to other modules
     of the data flow processor */
```

```
operation-pkt = array[ 0..15 ] of byte;

     /* An error packet or a dump packet is sent to the host computer.
     A dump packet dumps the contents of a cell, in response to a
     dump-cmnd-pkt received */

error/dump-pkt = union[ error-pkt, dump-pkt ];

     /* An error packet is sent to the host computer in response to an
     event packet or command packet that arrives when not expected. The
     contents of the error packet is identical to the conents of the
     input packet whose processing leads to an error. */

error-pkt = union[ event-pkt, set-cmnd-pkt ];


/* packet Format Definitions */

/* OPND, ACK, SET-INSTR, SET-ACK, SET-VAR, SET-CON, DUMP-CMND, ENABLE,
DISABLE, and DUMP are literal packet type identifiers */

cell-address = 0..15; /* specifies one of the sixteen cells in a cell block */

operand = array of byte;

opnd-pkt = packet[ type: (opnd), cell-ad: cell-address,
                   rec-num: 1..3; /* specifies one of 3 receivers in a cell */
                   opnd: operand];

ack-pkt = packet[ type: (ACK), cell-ad: cell-address];
```

```
set-instr-pkt = packet[ type: (INST), cell-ad: cell-address,
                        ack-exp, ack-rec: 0..7, /* up to seven acknowledgements */
                        format-spec: array[ 1..3 ] of opnd-spec,
                              /* opnd-spec specifies which bytes of instr comprise
                                 the contents of the corresponding receiver */
                        instr: array[ 0..15 ] of byte ];

          opnd-spec = record[   used: boolean, /* is receiver used ? */
                              origin: 0..15,    /* location and */
                              length: 0..7 ];   /* length of receiver contents in
                                                   instr */

set-ack-pkt = packet[ type: (SET-ACK),
                      cell-ad: 0..15,
                      ack-exp: 0..7,
                      ack-rec: 0..7 ];

set-var-pkt = packet[ type: (SET-VAR),
                      cell-ad: cell-address,
                      rec-num: 1..3 ];

set-con-pkt = packet[ type: (SET-CON),
                      cell-ad: cell-address, rec-num: 1..3,
                      opnd: operand ];

dump-cmnd-pkt = packet[ type: (DUMP-CMND),  cell-ad: cell-address ];

enable-pkt = packet[ type: (ENABLE) ];

disable-pkt = packet[ type: (DISABLE) ];
```

```
        /* for an explanation of the components of a dump packet, refer to
        Section 7 of " Data Flow Design Note 1 --- Specification of the Instruction
        Cell Block for a Data Flow Processor," Computation Structures Group,
        Laboratory for Computer Science, M.I.T., Cambridge, Mass. */

    dump-pkt = packet[ type: (DUMP),
                       cell-ad: cell-address,
                       enab: boolean,
                       ack-flag: boolean,
                       ack-exp: 0..7,
                       ack-rec: 0..7,
                       recs: array[ 1..3 ] of record[   used: boolean,
                                                       recvd: boolean,
                                                      origin: 0..15,
                                                      length: 0..7,
                                                        mode: (VAR, CON) ],
                       instr: array[ 0..15 ] of byte ];
```

```
Cell-Block: module
            event/command input port receives event/command-pkt,
            run input port receives run-pkt,
            arbnet-ready input port receives signal,
            operation output port sends operation-pkt,
            error/dump output port sends error/dump-pkt,
            enabled output port sends signal;

behavior:

/* State Variable */

            run-ind: boolean;
            instruction-arry: array[ 0..15 ] of record operation-pkt;
            control-arry: array[ 0..15 ] of cell-state;
            enab-count: 0..16;
            xmit-flag: boolean;

/* Cell State Record Format */

            cell-state = record[   used: boolean,
                                   enab: boolean,
                                 ack-flag: boolean,
                                 ack-exp: 0..7,
                                 ack-rec: 0..7,
                                 recs: array[ 1..3 ] of rec-state ];

/* Receiver State Record Format */

            rec-state = record[  used: boolean,
                                recvd: boolean,
                               origin: 0..15,
                               length: 0..7,    mode: (VAR, CON) ];
```

```
/* Procedure to Transfer Operand Value from Input Packet to Instruction Array */

enter-opnd: procedure(opnd: operand, rs: rec-state; op: record operation-pkt );

     begin  org := rs.origin;
            lth := rs.length;
            for k := 0 to lth-1 do
                   if k < length(opnd)
                        then op[org+k] := opnd[k]
                        else op[org+k] := 0;
    return
end enter-opnd;

/* Procedure to Select an Enabled Cell */

priority-search: procedure( ca: array[ 0..15 ] of cell-state );

    j: own := 0;  /* j is initialized to 0, but its content is retained between
                      calls to priority search */
    until ca[j].enab = true do
                   if j < 15 then j := j+1  else j := 0;
    return j;
end priority-search;

/* Procedure to Transmit an Operstion Packet */

xmit-operation: procedure( cs: cell-state, op: array[ 0..15 ] of byte );

   send op at operation;
   cs.ack-flag := false;
   for i:= 1 to 3 do  if cs.recs[i].mode = VAR then cs.recs[i].recvd := false;
   cs.enab := false;
   return
end xmit-operation;
```

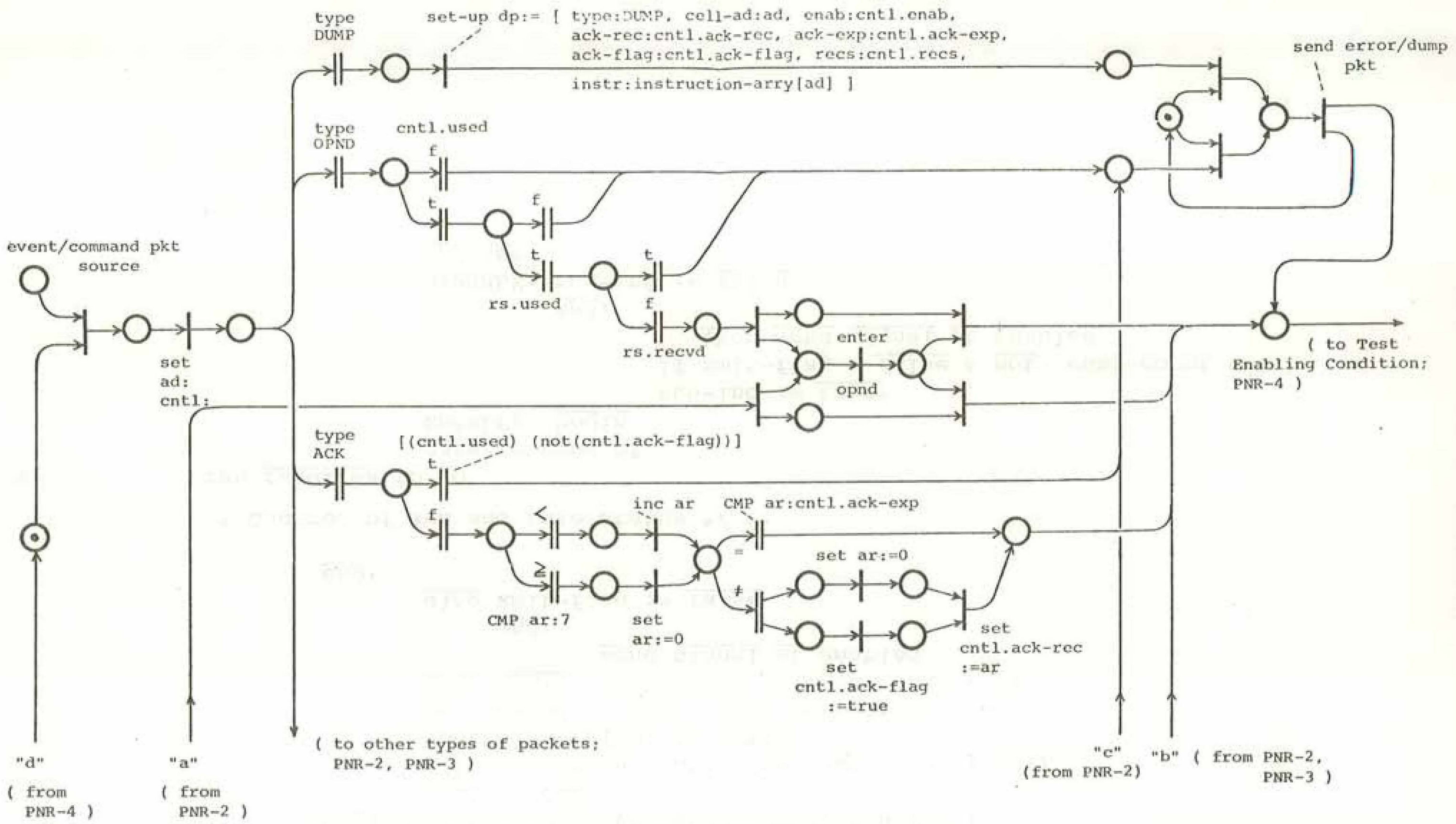
```
/* Initialize State Variables */

        run-ind     := false;
        enab-count  := 0;
        xmit-flag   := false;
        for j := 0..15 do
                          control-arry[j].used := false;
                          control-arry[j].enab := false;

/* Process Packet */

repeat begin
        when begin
        event/command receives e-c-pkt do
                begin
                ad   := e-c-pkt.cell-ad;
                cntl := control-arry[ad];
                case e-c-pkt.type of

                     /* Process Operand Packet */

                     OPND: begin
                           if cntl.used = false
                                then send e-c-pkt at error/dump;
                           rn := e-c-pkt.rec-num;
                           rs := cntl.recs[rn];
                           if rs.used = false
                                then send e-c-pkt at error/dump;
                           if rs.recvd = true
                                then send e-c-pkt at error/dump;
                           enter-opnd(e-c-pkt.opnd, rs, instruction-arry[ad]);
                           rs.recvd := true;
                           end;
```

```
/* Process Acknowledge Packet */

ACK: begin
     if cntl.used = false ∨ cntl.ack-flag = true
          then send e-c-pkt at error/dump;
     ar := cntl.ack-rec;
     if ar < 7 then ar := ar+1  else ar := 0;
     if ar = cntl.ack-exp then
          begin ar := 0;
                cntl.ack-flag := true  end;
     cntl.ack-rec := ar
     end;

/* Process Set-Acknowledge Packet */

SET-ACK: begin
     if run-ind = true  then send e-c-pkt at error/dump;
     if cntl.used = false  then send e-c-pkt at error/dump;
     cntl.ack-exp := e-c-pkt.ack-exp;
     cntl.ack-rec := e-c-pkt.ack-rec;
     if cntl.ack-rec = cntl.ack-exp
          then begin
               cntl.ack-rec := 0;
               cntl.ack-flag := true
               end
          else cntl.ack-flag := false;
     end;

/* Process Set-Instruction Packet */

SET-INSTR: begin
cntl.ack-exp := e-c-pkt.ack-exp;
cntl.ack-rec := e-c-pkt.ack-rec;
```

```
 if cntl.ack-rec = cntl.ack-exp
      then begin
           cntl.ack-rec := 0;
           cntl.ack-flag := true
           end
      else cntl.ack-flag := false

 /* uf, indicating whether the cell is used or not, is false
    if and only if the cell expects no acknowledgement and
    none of the receivers is used */

if cntl.ack-exp = 0
      then uf := false  else uf := true;
for i := 1..3 do
      begin  os := e-c-pkt.instr.format[i];
             if os.used = true  then uf := true;
             cntl.recs[i] := [   used: os.used,
                               origin: os.origin,
                               length: os.length,
                                 mode: VAR,
                                recvd: false ];
      end;
cntl.used := uf;
instr-arry[ad] := e-c-pkt.instr
end;

/* Process Set-Variable and Set-Constant Packet */

SET-VAR, SET-CON: begin
      if run-ind = true  then send e-c-pkt at error/dump;
      if cntl.used = false  then send e-c-pkt at error/dump;
      rn := e-c-pkt.rec-num;
      rs := cntl.recs[rn];
      if rs.used = false  then send e-c-pkt at error/dump;
```

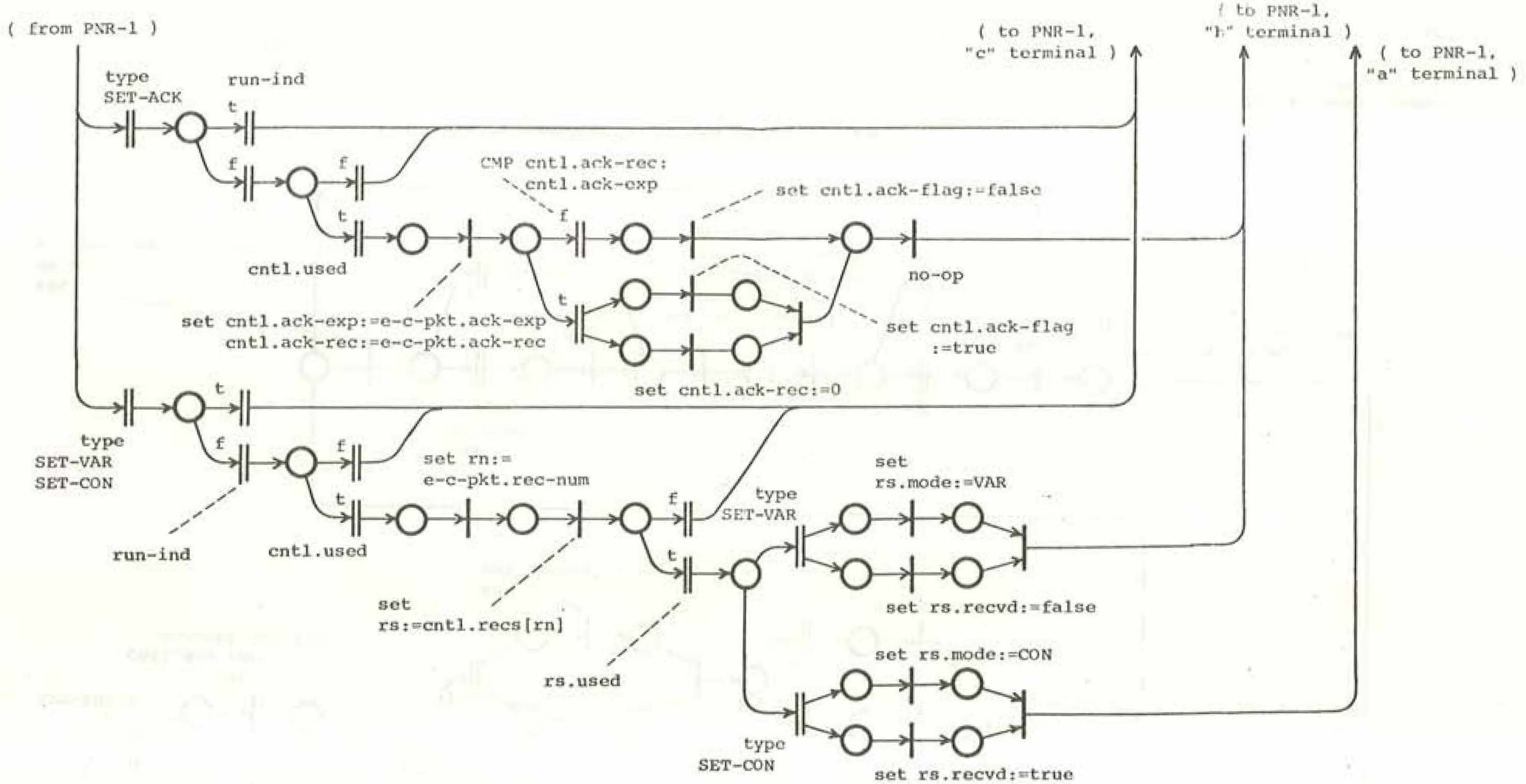
```
        case e-c-pkt.type of
        SET-VAR: begin  rs.mode := VAR;
                        rs.recvd := false  end;
        SET-CON: begin  rs.mode := CON;
                        rs.recvd := true;
                        enter-opnd( e-c-pkt.opnd, rs,
                              instruction-arry[ad])  end
        endcase;
        cntl.recs[rn] := rs
        end;

   /* Process Dump Command Packet */

   DUMP-CMND:  begin
        dp: record dump-pkt;
        dp := [ type: DUMP,  cell-ad: ad,  enab: cntl.enab,
                ack-flag: cntl.ack-flag,
                ack-exp:  cntl.ack-exp,  ack-rec: cntl.ack-rec,
                recs: cntl.recs,  instr: instruction-arry[ad] ];
        send dp at error/dump;
        end

/* end of the case statement for handling different types of
   event and command packets */

endcase;
```

```
/* Test Enabling Conditions and Transmit Operation Packets */

en := cntl.ack-flag;
for i := 1 to 3  do
     begin
           rs := cntl.recs[i];
           if rs.used = true & rs.recvd = false   then en := false;
     end;
if en = true  then
     begin
           cntl.enab := true;
           if run-ind = true & xmit-flag = false
                 then begin
                            send signal at enabled;
                            xmit-flag := true
                      end
                 else enab-count := enab-count + 1
     end;
control-arry[ad] := cntl

end; /* Processing of input packets from input port
        event/command ends here */
        /* Arbitration Network signals ready */

        arbnet-ready receives signal do
                begin  ad := priority-search (control-arry) ;
                       cntl := control-arry[ad];
                       xmit-operation (cntl, instruction-arry[ad]);
                       control-arry[ad] := cntl;
                       if run-ind = true & not( enab-count = 0 )
                       then begin  enab-count := enab-count - 1;
                                   send signal at enabled
                            end
                       else xmit-flag := false
                end;

        /* Control of Run and Idle Status */

        run receives rp do
                       case rp.type of
                       ENABLE:  begin
                                        run-ind := true;
                                        if xmit-flag = false & not( enab-count = 0 )
                                          then send signal at enabled
                                end;
                       DISABLE: run-ind := false
                       endcase

        end when

    end repeat

end Cell-Block;
```


type DUMP
set-up dp:= [type:DUMP, cell-ad:ad, enab:cntl.enab, ack-rec:cntl.ack-rec, ack-exp:cntl.ack-exp, ack-flag:cntl.ack-flag, recs:cntl.recs, instr:instruction-arry[ad]]
send error/dump pkt
type OPND
cntl.used
f
t
f
t
t
f
rs.used
rs.recvd
enter
opnd
event/command pkt source
set ad: cntl:
(to Test Enabling Condition; PNR-4)
type ACK
[(cntl.used) (not(cntl.ack-flag))]
t
f
<
≥
inc ar
CMP ar:cntl.ack-exp
=
≠
set ar:=0
CMP ar:7
set ar:=0
set cntl.ack-flag :=true
set cntl.ack-rec :=ar
"d"
(from PNR-4)
"a"
(from PNR-2)
(to other types of packets; PNR-2, PNR-3)
"c" (from PNR-2)
"b" (from PNR-2, PNR-3)

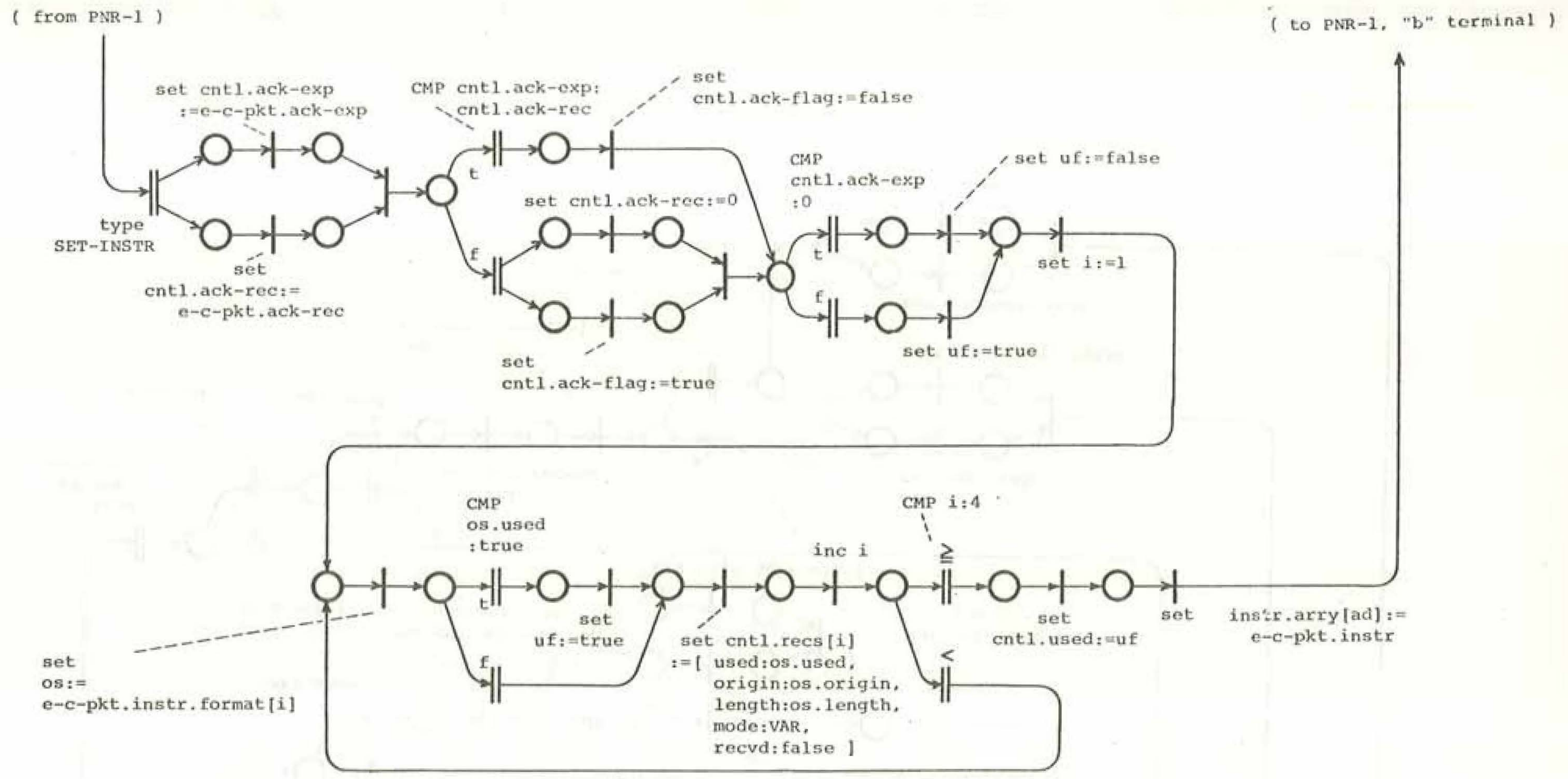
Appendix 2 Petri Net Representation of the ADL Specification

[PNR-1: event/command: DUMP, OPND, ACK]



Appendix 2 Petri Net Representation of the ADL Specification

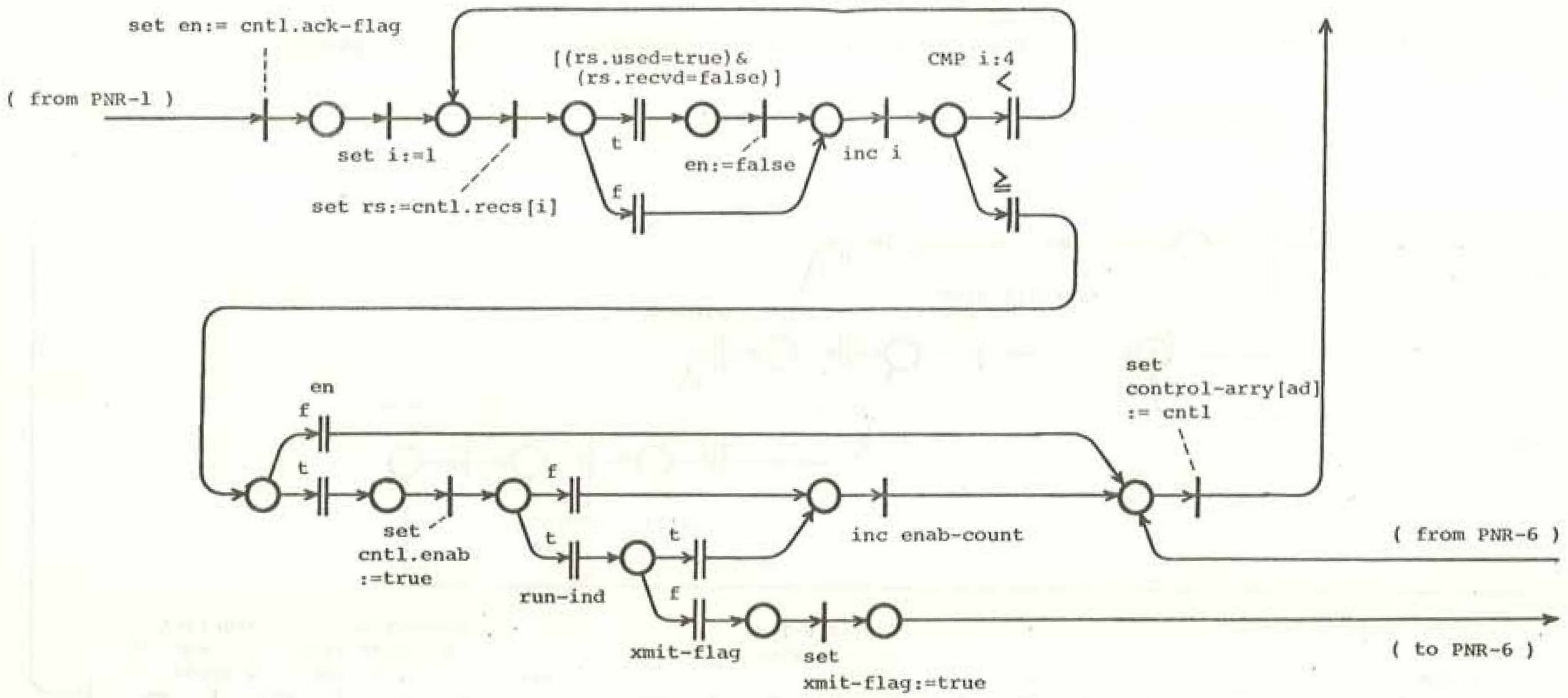
[PNR-2: event/command: SET-ACK,SET-VAR,SET-CON]



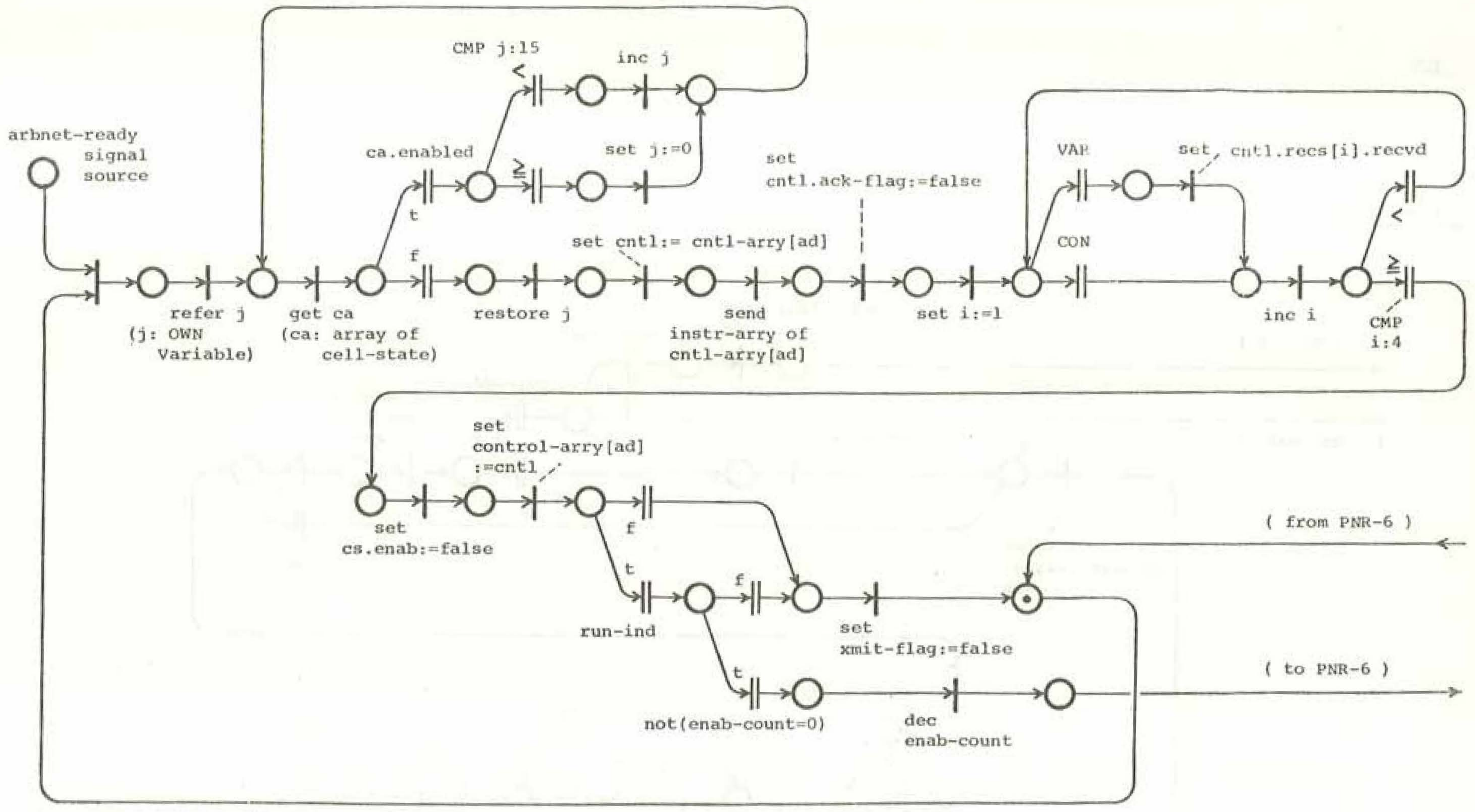
(from PNR-1)
(to PNR-1, "b" terminal)
set cntl.ack-exp :=e-c-pkt.ack-exp
CMP cntl.ack-exp: cntl.ack-rec
set cntl.ack-flag:=false
type SET-INSTR
set cntl.ack-rec:= e-c-pkt.ack-rec
t
f
set cntl.ack-rec:=0
set cntl.ack-flag:=true
CMP cntl.ack-exp :0
set uf:=false
t
f
set i:=1
set uf:=true
CMP os.used :true
t
f
set uf:=true
inc i
CMP i:4
≥
<
set cntl.used:=uf
set
instr.arry[ad]:= e-c-pkt.instr
set os:= e-c-pkt.instr.format[i]
set cntl.recs[i] :=[used:os.used, origin:os.origin, length:os.length, mode:VAR, recvd:false]

Appendix 2 Petri Net Representation of the ADL Specification

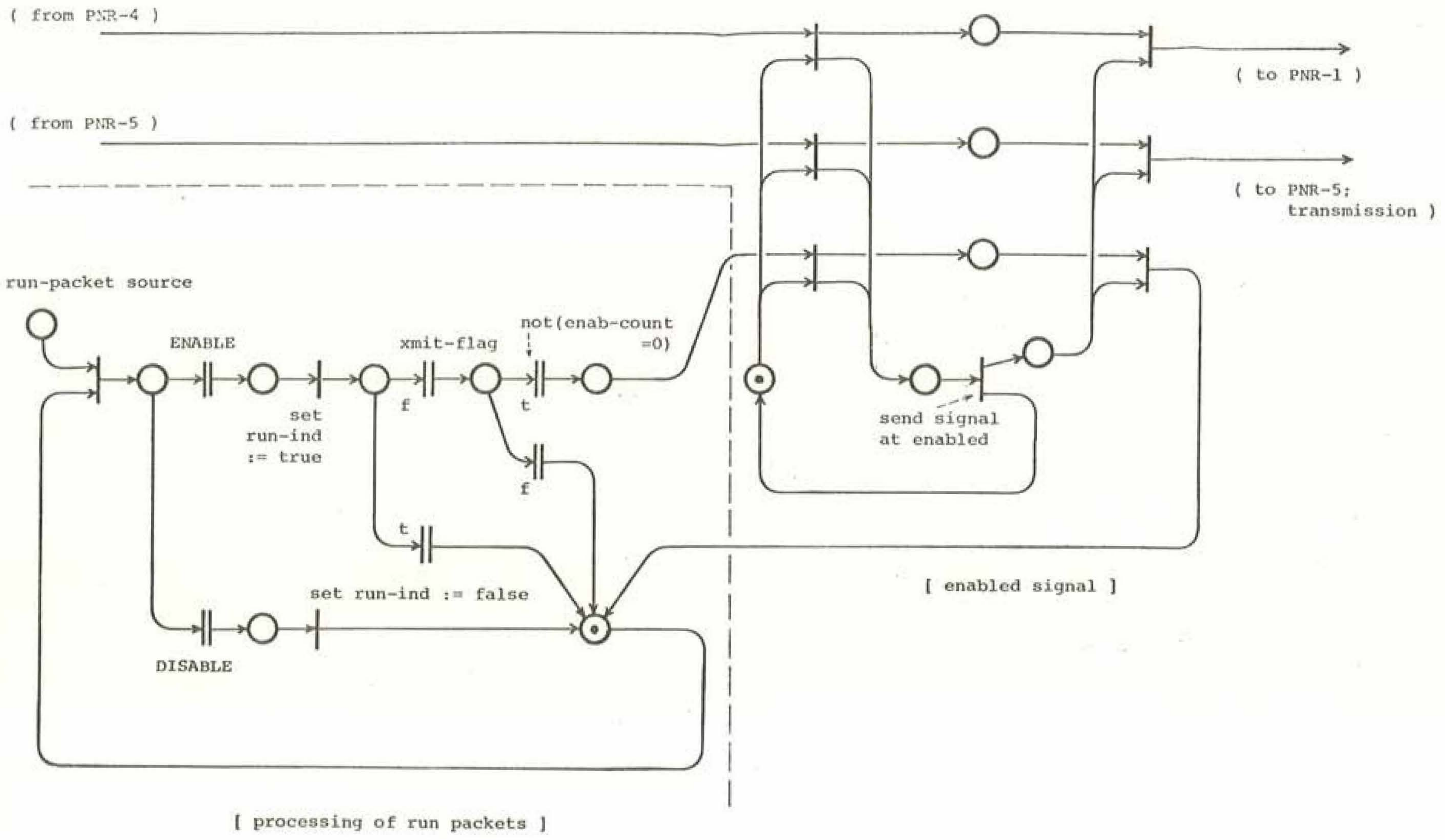
[PNR-3: event/command: SET-INSTR]



Appendix 2 Petri Net Representation of the ADL Specification [PNR-4: Test Enabling Condition]



Appendix 2 Petri Net Representation of the ADL Specification [PNR-5: Transmission]



Appendix 2 Petri Net Representation of the ADL Specification

[PNR-6: run packet & enabled signal]