531

THE DISTRIBUTED COMPUTER SYSTEM

Yan Hong NG

This Dissertation is submitted for the degree of Doctor of Philosophy

Department of Electrical Engineering & Electronics

Brunel University

1983

CONTENTS

ACKNOWLEDGEMENT

To all those who have support me when I most needed encouragement or help. First of all, I will like to express my upmost gratitude to the ACM (Association for Computing Machinery) and the IEEE Computer Society (Institute of Electrical and Electronics Engineers) for the provision of enormous valuable publications, which like the Polaris has guided me through the "dark ages" of my first two years. Without them the completion of this thesis is almost beyond imagination.

I am indebted to my supervisor, Dr. R. J. Glover for many advices and hleps that he has given to me in the past ten months. I am also grateful to Professor I. Aleksander for enabling the appointment of Dr. Glover as my supervisor who really care for me.

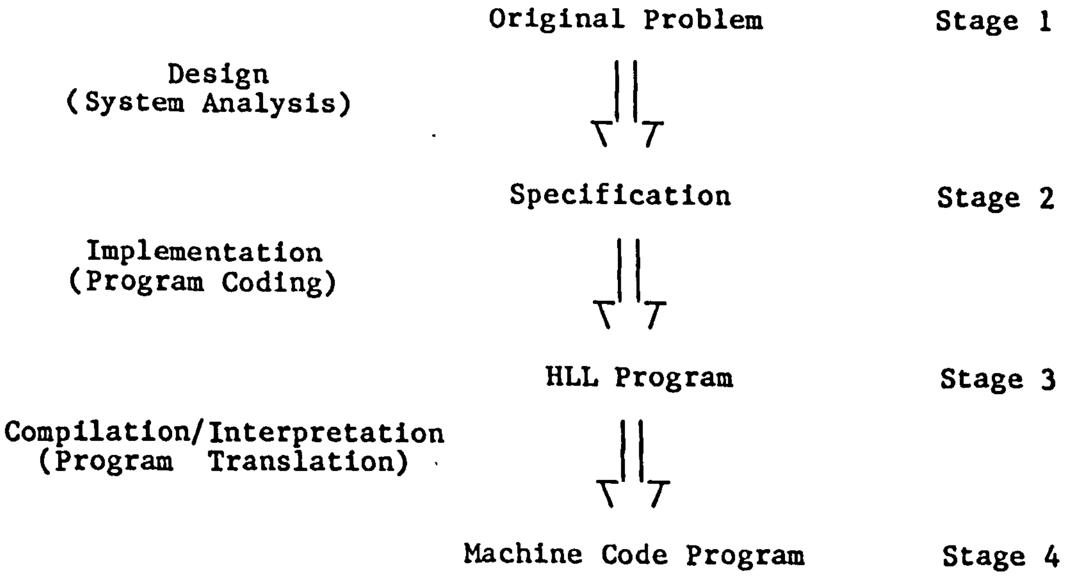
The Work described in this thesis is to the best of my knowledge original, and was done by myself without collaboration. The hardware organization of the Byte-Organized Associative Processor is originated from Mike Lea's Associative Parallel Processors, but to a substantial extent, it is being improved and modified to cater for the implementation of abstract data types. The contribution of my project student Phil. Hardy towards the writing of BOAP simulator is hereby acknowledged.

CHAPTER ONE

INTRODUCTION

Language is a system for the expression of thoughts, feelings, etc., between two parties, by the use of spoken sounds or conventional symbols. However, in order to communicate, they must speak the same language, and be able to decode the meaning behind those sequence of sounds or words generated by his/her counterpart, otherwise, their conversation would end in failure. Although, this obstacle can sometimes be removed, by calling in an interpreter or by using a more explanatory language to put across the message, it is always a very time consumming and frustrating process.

A similar parallel can be drawn, when communicating with a computer. Due to the gap between what we want (expectation) and what the machine can provide (limitation), overheads have to oe included to translate from the original problem into a directly executable machine code program. Problem solving by means of computer programming can be viewed as a process consisting of three translation subprocesses:



(1) Design : translation from original problem to design specification.

(2) Implementation: translation from design specification to a program written in High Level Language (HLL).

(3) Compilation/ Interpretaion: translation from the HLL program into machine code program.

Most of the difficulties encountered in programming stem from the great distance that separates the initial statement of a problem and its correctly encoded solution, a distance which the programmer himself must traverse. The HLL which the programmer is using will provide him with a collection of data types, operations, and representation schemes that the language supports as primitives. Ultimately, all problems of representation must be solved by finding a way to represent every object of interest as a combination of primitive representations, and all actions must then be expressed as combinations of the primitive operations by means of the semantic and syntactic mechanisms made available by the HLL. Hence, in order to improve the efficiency of problem solving on computer, the distance between stage 1 and 4 must be reduced. We will examine this problem in stages:

(1) System Analysis Process: The Design

A software system is developed in order to meet a need perceived by its user. The system analysis process is therefore used here before any other porcesses to establish and analyse the needs of the user, and specify them in the form of a set of requirements that stating "what" the system is expected to satisfy, but not "how" to achieve them. In other words, it is the process of translating from original problems to design specifications is usually done manually by the system's analyst. The automation of this process still remains an active research area of Artificial Intelligence (A.I.). Nonetheless, we will be taking this subject up again in Chapter 8.

(2) Program Coding Process: The Implementation

One of the first and most important tasks to be accomplished before writing a program is the translation of data of the problem onto the manipulable data types and structures of the HLL, so much so that the data types and structures provided in a programming language determine its user's approach to all problems, as well as profoundly influencing their ways of thinking in programming. However, due to the shortage of powerful primitives available in HLLs, program coding is often a very labour intensive process.

That is, in order to support his design, the programmer need not only have to express his design decisions by means of program, but must also get involved in building data structures and their associated operational constructs as supplements to the HLL primitives. Hence, an inordinate increase in the size and cost of software to such an extent that this unnecessary burden has often been the prime target to be blamed for the so called "software crisis". Therefore, the enhancement of representation primitives in HLL is an area that we will be looking into for reducing the distance between stage 2 and 3.

To do so, we will concentrate on that class of languages which have come to be known as "very high level languages", or nonprocedural languages. A possible definition of a "nonprocedural" program is as Sammet and Leavenworth[13] have noted:

> .... a "nonprocedural" program is a prescription for solving a problem without regard to details of how it is solved.

Briefly put, the nonprocedural languages embody a more abstract approach to data, which provide data structures of greater flexibility and power, including most of the well known data types such as string, list, tree, graph etc., and powerful operations and dictions of mathematics such as set-theoretic and relational types of objects, directly available as primitives. Futhermore, they also allow various powerful techniques such as associative referencing, and nondeterministic programming for the manipulation and combination of processes. However, the crucial challenge facing the procedural languages has been the cost-effective implementation of their programming constructs.

There is reason to believe that current research in nonprocedual programming languages has brought us to the threshold of another advance in programming methodology[13]. The general characteristics of this class of languages can be defined as follows:

1) Associative Referencing

   Associative referencing capability is an important feature of database retrieval langauges, set theoretic languages and A. I. languages, that allow data to be addressed by their contents. In other words, the programmer does not have to specify access paths explicitly or program an algorithm to conduct a search for a specific data structure. Associative referencing is usually provided in those languages that contain sets and algebraic operations, for example, SETL[1] and Codd's ALPHA language[2,3]. Codd defines algebraic operations on relations[4] which effectively provide various types associative referencing.

2) Pattern Directed Structures

   The classical example of a pattern directed structure is given by Markov algorithms[5]. A Markov algorithm consists of a set of replacement or substitution rules which are repeatedly applied to an input string of symbols. The sequencing algorithm is implicit in that the rules are always applied in a determined order. Each rule consists essentially of the directive: if a specified string is contained in the current input string, then replace it with a given string of symbols. The SNOBOL language is an extension and enhancement of the Markov algorithm idea, where the programmer is allowed to depart from the normal sequential control. The pattern directed structure is considered as nonprocedual in the sense that the implementation of its process in a more "conventional" language will involve a complicated series of steps. In other words, pattern directed structure is the abstraction of string data type at the programming language level.

3) Aggregate Operators

In the development of programs by "Top Down" approaches[64] and stepwise refinement[68], the programmer is encouraged to design his algorithms and expresses them in an "abstract" program operating on "abstract" data. He then chooses for the abstract data some convenient and efficient representation in the form of a particular programming language's primitives, and finally programs the algorithms required by his abstract program in terms of these concrete representations. Often, a long and frustrating process that the programmer has to undergo. These abstract data types, known as data structures, or aggregate operators, consist of data elements with important structural relationships. Therefore the provision of aggregate operators, proposed as primitives of nonprocedual languages, has come some distance in compromising with the human tendency to think through problems in terms of aggregate constructs with which more complex aggregates can be built. As it will soon be apparent from Chapter Six and Seven that the algebraic operators defined by Codd are aggregate operators[13], so do many other data structures in various HLLs such as vectors and array (APL)[6], strings and patterns (SNOBOL)[7], sets (SETL[1] and Prolog[8]), lists (LISP[9]), and many other abstract data types such as trees and graphs.

4) Nondeterministic Programming

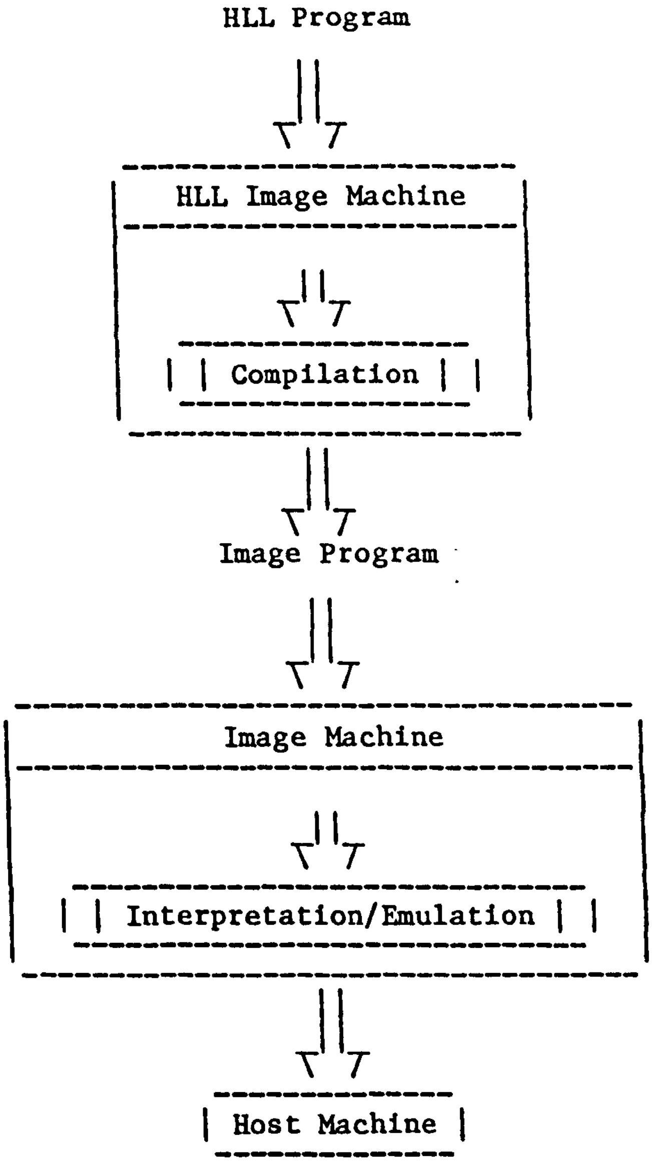
This facility appears in most of the A. I. languages and was inspired by Floyd[11] who introduced new programming primitives for solving combinatorial problems. Essentially, problem solving methods are characterized by searching through a state, or situation space or through a space of alternatives. A solution is a sequence of state transitions from an inital state(s) given in the problem specification, to a final, or goal state. This implies a process of going through a multiple branch in the execution of the program, in order to search for the goal state(s). Each path is computed conceptually in parallel, with its own particular value of the choice as an argument. In most cases, nondeterministic programs are executed as backtracking algorithms[12]. This powerful semantic mechanism enables a programmer to solve complex problems such as chess playing or the "problem of 8 queens" [68]. There is a close correspondence between nondeterministic programming and parallel programming in that the multiple paths of the choice function could be searched in parallel.

Other facilities of interest to the designer of nonprocedural languages can be enumerated, for example elimination of arbitrary sequencing[13] and functional programming[59]. However, it is the criterion first stated that remains the most fundamental; a language is nonprocedural to the degree that it shortens the distance between fomulating and solving some significant classes of programming problems. Summing-up the points discussed so far, it is revealed that the main obstacle towards the implementation of nonprocedual constructs has come from the serious mismatch between the architectures and their supporting machine, which lies right in the heart of the data organisation of all nonprocedural constructs mentioned above: they all belong to the SIMD architecture[10] which can not be efficently implemented on the Von Neumann machine (SISD processor)[10]. Hence, orders of magnitude improvement are only possible if SIMD type array processors can be brought in to support these SIMD constructs.

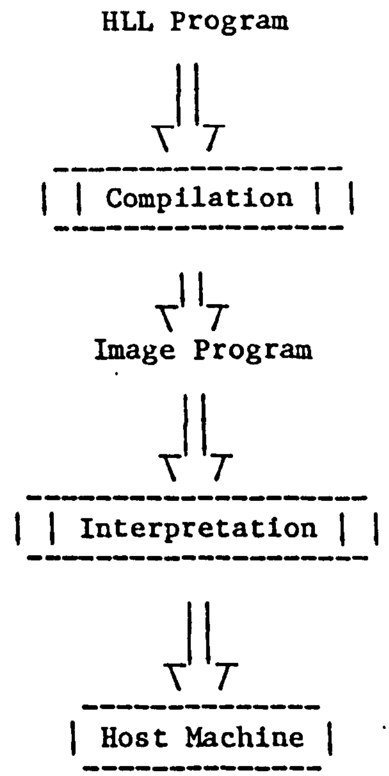
However, there seems in general to be a close relation between the associative referencing and all the other nonprocedural constructs that we are discussing. It is certainly clear that the operations of all known data structures are characterized by searching through the contents of data items and establishing relationships betweem them, so much so that even nondeterministic programming is predominated by searching. Hence, this justifies our choice of an associative array processor for the investigation into implementation of nonprocedural constructs.

(3) Compilation/Interpretation: Program Translation

The machine which understands and runs HLL programs is called the image machine of this HLL. By definition, it must have a set of commands and a storage which are exactly the range and domain of the commands, together with a mechanism that causes the state transitions determined by the commands. This mechanism is often referred to as the host machine.



The host machine itself may be an image machine for another host--hence, there may be several nested levels of interpretation and execution before the actual state transitions. Whilst recognizing this, we will not introduce this unnecessary complexity, especially since this nesting provides very little additional insight into the fundamental mechanism involved. In this thesis, we will assume a simplified model in which a HLL program representation is translated into an image program, that is interpreted by a host machine.



Ideally, one would like to bypass the Compilation/Interpretation state all together and have a host machine which could directly execute the HLLs' program. Yet in practice this can never be the case, the reasons being two fold: Firstly, even with the promise of VLSI technology, it is still not cost-effective to build a well-mapped (between image and host) machine, or a language-oriented image machine[86,87], some HLL instructions will still have to be translated to a certain extent. The best that a computer architect can do is to identify those more frequently used program constructs, and implement them by means of hardware, as a result, bringing the image and host machines closer to each other. Secondly, however direct the direct-execution computer could be, certain phases of the compiler/interpreter process such as Lexical Analysis, Syntax and Semantic Analyses can never be bypassed[14]. Nevertheless, the complexity of Compiler/ Interpreter process could still be minimized. We will first look into the functional organization of the compiler/interpreter:

1) Lexical Analyzer (String & List Processors)
2) Syntax Analyzer (List & Tree Processors)
3) Semantic Analyzer (List Processor)
4) Code Generator (Tree & List/String Processors)

At the first phase of the compilation/interpretation process, a string processor (Lexical Analyzer) takes the HLL source program as input, identifying comments, blanks, quoted strings, identifiers and constants in the source program, grouping characters into tokens, and places them into symbol tables. The List & Tree processors (Syntax Analyzer) are then used at the second phase to group these tokens into syntactic structures such as expressions and statements, before putting them into a convenient form such as postfix Polish notation, tree structure, or quadruples for the Code Generation phase. In the process of these transformations, the list processor (Semantic Analyzer) examines each syntactic construct that has been recognized, checking data types, determining that functions are called with the appropriate number of arguments, and verifying the identifiers which have been declared are typical of what takes place during semantic analysis. Any error encountered, either syntactically or semantically, would prompt appropriate error recovery procedures to be entered for the necessary diagnostics. Finally, another tree & list/string Processor (Code Generator) will process those trees and generate algorithms to drive the host machine.

Traditionally, these special purpose processors are software simulated on a conventional Von Neumann processor, as a result, they increase the overhead of compilation/interpretation process. However, since string, list and tree are part of the well known data structures which the nonprocedural languages support as primitives, inclusion of abstract data types into HLLs will certainly provide a solution for the inefficiency of the program translation process. Therefore, the associative array processor remains the key issue towards the solution of these problems.

The idea of associative array processors originated during the late 1950's[15]. Although not many associative array processors have actually been built to date, many hundreds of papers have been written on the subject. The asscoiative array processor works on the principle of Content Addressable Memory (CAM) in which each memory cell can be addressed by means of its content rather than its location. From the architectural point of view, the associative array processor is really quite different from the Von Neumann machine. Whereas the latter is based on the ADD operation, the former is based on the COMPARE operation. Just as the Von Neumann computer compares by subtracting and testing for zero, the associative array processor adds by comparison: addition is done by a process analogous to "table look up". Since all memory words are capable of executing the SEARCH operation in parallel, the associative array processor is capable of highly parallel addition. Nontheless, due to their relatively high implementation cost, associative array processors are usually used in conjunction with standard Von Neumann computer systems so that many high-speed parallel processing tasks which cannot be efficiently executed by SISD (single instruction, single data stream) processors[10], are performed by associative array processors. But recent rapid changes of LSI/VLSI technology has greatly reduced the implementation cost of associative array processors, and there is anticipation that associative array processors will be used more extensively in improving the efficiency of problem solving on computer.

In the following chapters, we will be discussing the hardware organisation of associative array processors, and then lead on to the design of the Distributed Computer System (DCS) in which an associative array processor and a Von Neumann processor are integrated together to provide with the best of two computing worlds.

CHAPTER TWO

THE HARDWARE ORGANIZATION OF ASSOCIATIVE PROCESSORS

Interest in associative processing can be traced back to late 1950's, when the computer designers recognized the advantages of the parallel searching of data by content addressing. However, until recently, the main obstacle to the realization of associative processors have been the enormous costs needed in hardware. A good survey by Hanlon[15] appeared in 1966 covered whole range of asssociative memories and their possible applications in the first decade (1956-1966) of development. During this period, many experimental associative memory models were built, culminating in the delivery of a 50-bit, 2K-word associative memory by Goodyear Aerospace to the Rome Air Development Centre in 1968. Since it was then not feasible to construct large scale associative memories, the emphasis of associative processor design during the following decade (1966-1977) tended to focus on subsystems which are capable of both arithmetic and fast search operations. A number of associative processors, notably PEPE and STARAN, were constructed. (Several surveys by Parhami[16], Minker[17], Thurber and Wald[18], Yau and Fung[19] appeared during this period have detailed report of these developments).

However, as a result of recent advances in LSI/VLSI technology, the design and realization of associative processors has entered a new age of development aiming at the construction of large capacity associative processors. For example: the construction of Airborne Associative Processor (ASPRO)[20,21] which consists of 2048 single-bit Processing Elements (PE's) and realized in CMOS/SOS VLSI technology, are designed with a total processing capacity of 64 MOPS (Million Operations Per Second). Also, the Massively Parallel Processor (MPP)[22], which is an extension of STARAN with 16384 PE's, is about 100 times the processing capability as its predecessor in the similar volume. It is able to do floating point operations at speeds better than 100 MOPS, 16-bit integer arithmetic at speeds between 400 MOPS (multiplication) and 3000 MOPS (addition). Such new developments are likely to renew the original interest in applying associative processors to non-numerical processings, and data base applications.

2.1 ASSOCIATIVE MEMORY

An associative memory can be defined as a memory system with the property that stored data items can be retrieved by their content or part of their content, instead of by address of a location as in the Random-Addressed Memory (RAM). From the hardware point of view, the basic element of the associative memory is merely a one bit processing cell that can only perform SEARCH, READ or WRITE operations on the interrogating data. Nevertheless, when a number of these associative bit-cells are brought horizontally together as a word-row, and then linked vertically to each another to form an Associative Memory Array (AMA), it has surprisingly become a very powerful machine, on which every bit-column and word-row of information can be processed in parallel.

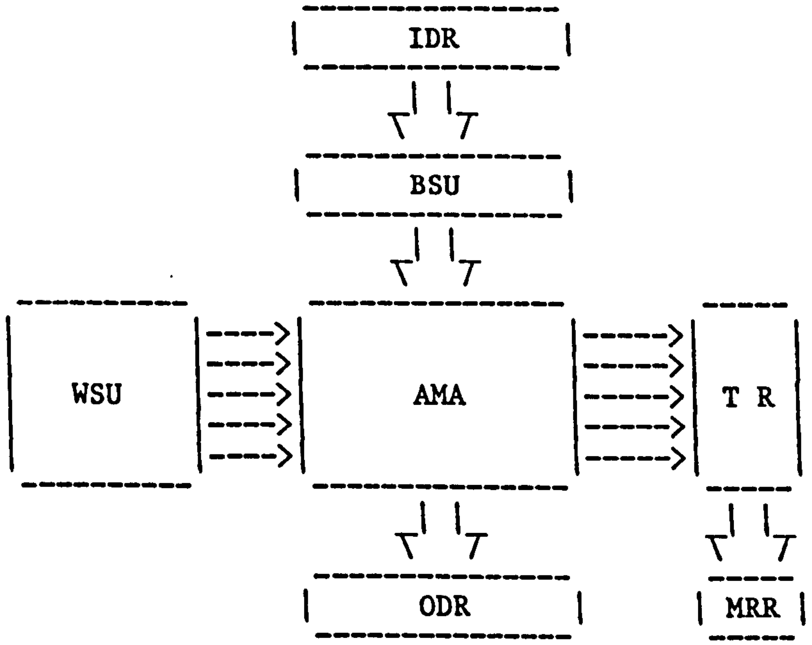


Fig. 2.1 The Organization of the Associative Memory

The SEARCH operations, which consist of masking and comparison, are executed in a fashion that depends on the organization of the associative memory. The search-key can be compared with all the words in the AMA, or some or part of the words through the control and selectivity of the Word Select Unit (WSU) and the Bit Select Unit (BSU). The possibility of matching multiple words to a search-key requires that the associative memory has some way of tagging the matched words. The tag function and matched-word indication are performed by the word-match tag networks called Tag Register (TR) and Match Reply Register (MRR). The IDR and ODR are the Input Data Register and the Output Data Register of the associative memory.

2.1.1 The Classification of Associative Memory

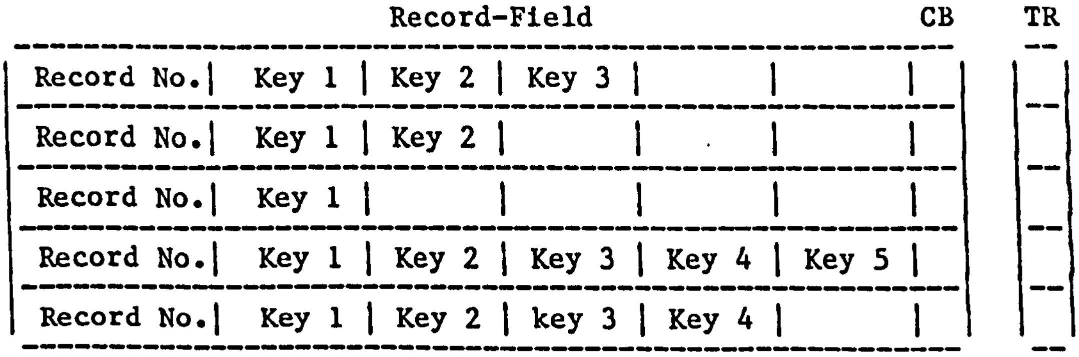
Although, many types of associative memories have been reported, Lea[23] had generally classified them into three categories of associative memory:

1) The Record-Organized Associative Memory
2) The Field-Organized Associative Memory
3) The Byte-Organized Associative Memory

2.1.1.1 The Record-Organized Associative Memory

In this configuration, every word-row of associative memory array has a fixed but long word length which ideally could be allocated to each record. It is by far the fastest configuration in which all records can be processed in parallel with only one instruction; either the SEARCH operation or READ/WRITE operation. Therefore, it is sometimes referred to as the fully parallel word-organized associative processor[19].

Associative Memory Array



Record-Field						CB	TR
Record No.	Key 1	Key 2	Key 3				
Record No.	Key 1	Key 2					
Record No.	Key 1						
Record No.	Key 1	Key 2	Key 3	Key 4	Key 5		
Record No.	Key 1	Key 2	key 3	Key 4			

Fig 2.2 The Record-Organized Associative Memory

The control bit CB is used as a "bit-map" or "access vector" to assist in the resolution of multiple responses and in boolean search combinations. This configuration is designed for applications in which all records are of similar length; consequently, for other applications in which records are of dissimilar length, some considerable redundancy could exist within the associative memory. However, this problem could be solved at the expense of the execution speed, if records are broken up at the field level.

2.1.1.2 The Field-Organized Associative Memory

This configuration is designed to provide the solution for dissimilar length applications, in which it subdivides the records into several fields, and allocates word-rows at the field level, in such a way that they can be joined up in almost any number to achieve a variable record-length data structure.

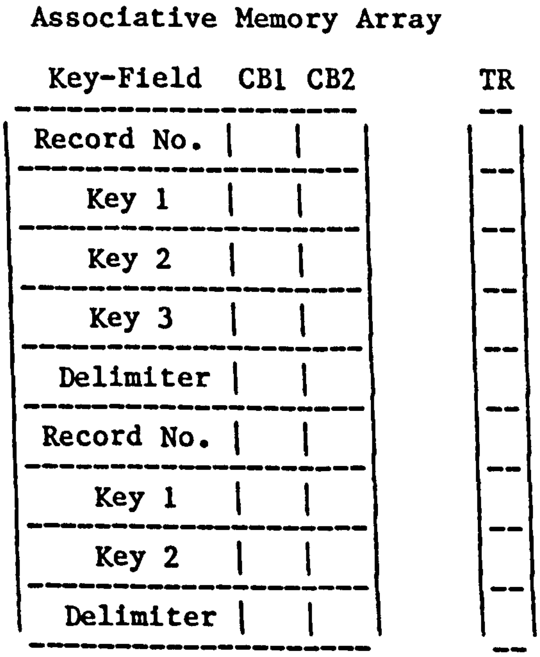


Fig. 2.3 The Field-Organized Associative Memory

However, one (or more) control-bits are needed to be used as delimiters, in addition to the control-bit of the Record-Organized Associative Memory. Nonetheless, the Field-Organized Associative Memory can support very different record lengths without incurring redundancy, provided that their keywords(fields) are of the similar length.

2.1.2.3 The Byte-Organized Associative Memory

If dissimilar length keywords do occur within the Field-Organized Associative Memory, then it will run into the similar problem as the Record-Organized Associative Memory. Thus, a Byte-Organized Associative Memory has commonly been suggested.

The Byte-Organized Associative Memory configures the data at the byte level, in which each word-row of the AMA is allocated to only a single byte of data, and a variable length keyword field is constructed by bringing together any number of bytes (word-rows), in such a way that it just like a record is constructed by linking up a chain of keyword fields. Nevertheless, more control bits are needed as markers to form and break the chain of keyword fields or records:

1) One bit (e.g. CB1) to mark the beginning of records.
2) One bit (e.g.CB2) to mark the beginning of keyword fields within records.
3) Two bits (e.g. CB3 & CB4) to act as "Tag Images".

Associative Memory Array

	CB1	CB2	CB3	CB4	TR
R	1	2			
E					
C					
O					
R					
D					
N					
o					
.					
K		2			
E					
Y					
1					
K		2			
E					
Y					
2					
K		2			
E					
Y					
3					
#					

Fig. 2.4 The Byte-Organized Associative Memory

Although the Byte-Organized Associative Memory leads to the slowest file searching among all the other associative memories, it incurs the least possible redundancy and it is by far the most economical to implement in hardware.

2.2 ASSOCIATIVE PROCESSORS

From a computer architect's point of view, associative processors belong to the category of so called SIMD (Single Instruction Stream Multiple Data Stream) machines[10]. A SIMD machine is a computer that performs operations on all selected processing elements with only one single instruction execution. But, unlike other array processor type SIMD machines[24,25], an associative processor is an SIMD machine whose processing elements and data addressing satisfy the following two properties:

1) The property of associative memory.

2) Data transformation operations, both arithmetic and logical, can be performed on a SIMD basis.

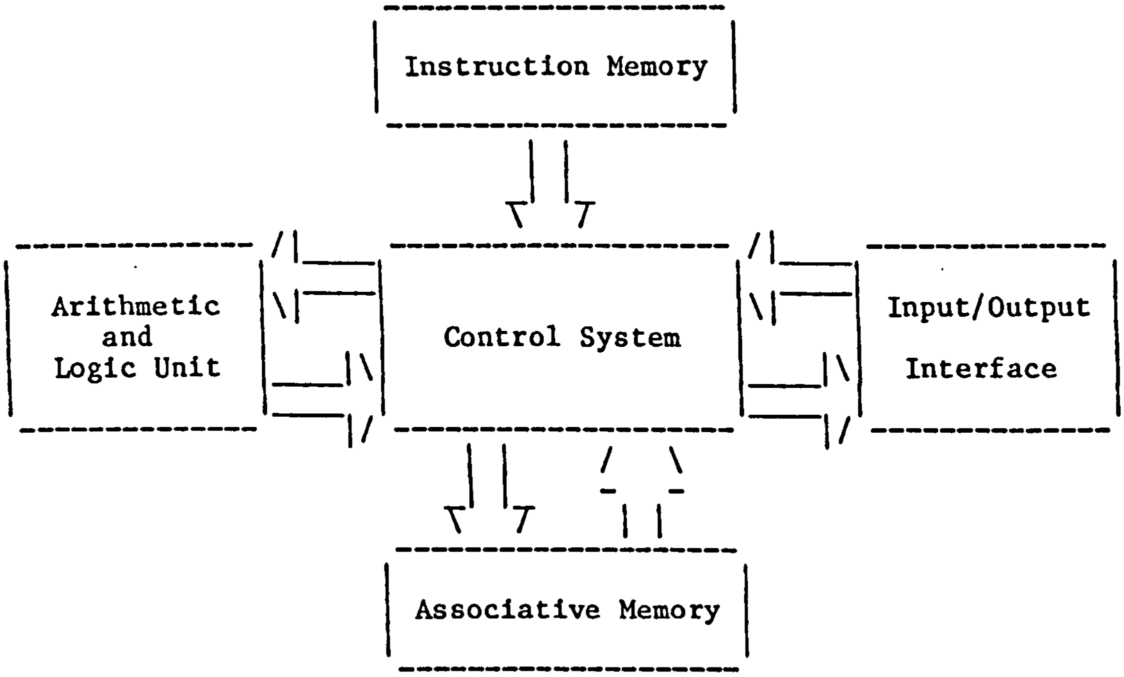


Fig. 2.5 The Block Diagram of an Associative Processor

An associative processor usually consists of an associative memory, arithmetic and logic unit (ALU), control system, instruction memory, and input/output interface. The major difference between an associative processor and a von Neumann machine is the use of associative memory, so much so that the classification of the associative memory is often used as a means to classify the architecture of associative processors, regardless of the details of their individual hardware implementation.

1) The Record-Orgánized Associative Processor

2) The Field-Organized Associative Processor

3) The Byte-Organized Associative Processor

Regarding of our intention to integrate associative processor into the Distributed Computer System for the implementations of abstract data types, the Byte-Organized Associative Processor (BOAP) has been chosen to serve this purpose. As it will soon be apparent that all data types regardless of theír structures, are ultimately mapped on to the physical hardware storage of the computer system, and abstract data types with their symbolic manipulation characteristic are best mapped on to the BOAP.

```
 ----------------------------------
| Character Field |CB1|CB2|CB3|CB4|
 ----------------------------------
|<--- 8  Bits --->|
```

The representations and manipulations of all known data structures to date are mostly non-numerical processing[130], which use character structure as a basic building block: String structures are built from a sequence of characters. Similarly, List, Tree, Graph structures are built from a chain of character fields with pointers to link them together. As a result, we shall be concentrating only on the Byte-Organized Associative Processor in the sections that follow.

2.3 THE BYTE-ORGANIZED ASSOCIATIVE PROCESSOR

The Byte-Organized Associative Processor (BOAP) shares the same general organization as shown in Fig. 2.5, except that it has replaced the ALU with a scratch Pad buffer for data transformations.

2.3.1 The Data Transfomations of BOAP

Traditionally, a ALU is integrated into the associative processor for performing complicated data transformations. For instance, all selected matched word-rows in the associative memory can be fetched serially into the ALU for specified data transformations, and the results are then stored back into the memory. But serious problems do exist in this approach:

1) If large amount of data are involved in data transformation, bottlenecks may occur in the I/O transfer, and the advantages gained from the use of associative memories will be lost.

2) It is contradictory to the design philosophy of the SIMD machine that a major part of its data transformations have to be done on the SISD basis.

However, the contemporary approach towards the solution of these problems is to provide a scratch pad buffer for holding intermediate results, and at the same time, increase the complexity of the Word Select Unit, to such an extent that data transformations can be performed within the associative memory by using table look-up procedures. This is achieved by the insertion of tag manipulated operations (as shown in Table 2.1) to activate other word-rows, before the execution of the READ/WRITE function. Nonetheless, the formal definition of the Associative Assembly Language (AAL), together with the tag manipulated operations which shown in the Table 2.1, will be the subject of Chapter four.

MNEMONIC	TAG MANIPULATION/WORD-ROW ACTIVATION
	No activation
PTT	Activates all tagged word-rows
PCT	Activates all untagged word-rows
RSTTU	Activates the first tagged word-row from T-end (Top end)
RSTTD	Activates the first tagged word-row from B-end (Bottom end)
RSCTU	Activates the first untagged word-row from T-end
RSCTD	Activates the first untagged word-row from B-end
EIR	Activates all rows from T-end or B-end to first tagged row
MOR	Activates all rows from first tagged row to T-end or B-end
GR	Activates all rows from tags in TR1 to TR2 (Group Run)
RSGSU	Activates only ending rows of every group during GR
RSGSD	Activates only starting rows of every group during GR
RSFGU	Activates only first group from B-end during GR
RSFGD	Activates only first group from T-end during GR
RSFGSU	Activates only first ending row from B-end during GR
RSFGSD	Activates only first starting row from T-end during GR

Table 2.1 The tag Manipulations of BOAP

2.3.2 The Hardware Organization of BOAP

The Byte-Organized Associative Processor (BOAP) that we are proposing is shown in Fig. 2.6. This overview of the BOAP is, in fact, the overall system organization of the processor. But, in order not to deviate our attention away from the hardware organization of the BOAP, we shall be focusing only on the associative memory part of the BOAP, other than the system part of BOAP which will be dealt with in greater detail in the next chapter.

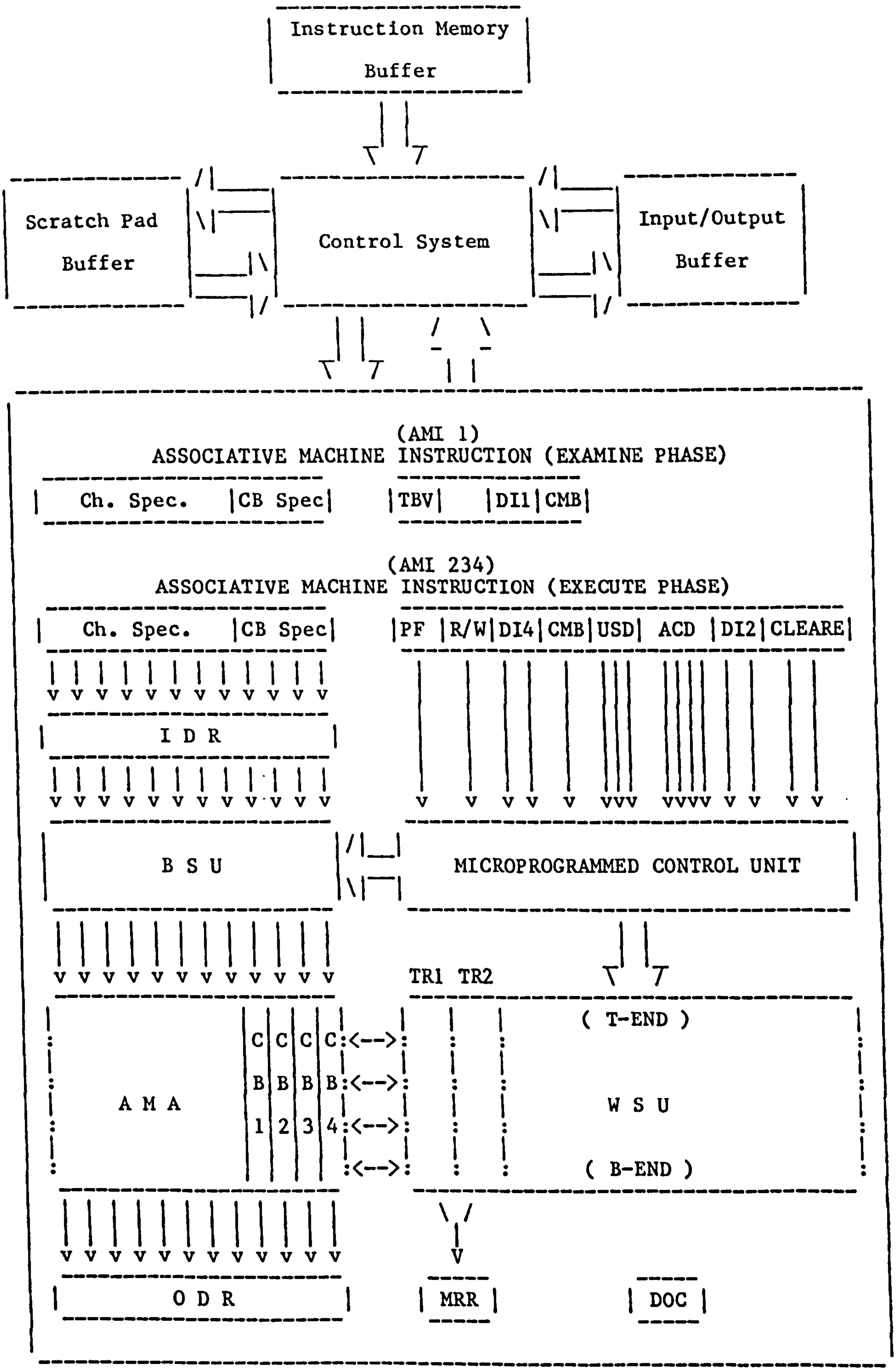


Fig. 2.6 The System Organisation of BOAP

2.3.2.1 The Associative Memory Array (AMA)

The AMA comprises a two-dimensional matrix of identical one-bit-cells, which operates as a read-write Random Access Memory (RAM) cell and contains sufficient logic to enable the selection of its content to be compared with the corresponding bit of the IDR.

Theoretically speaking, the AMA could contain any number of word-rows, which are organized in a 12 bit long format --8 bits for the character field and 4 bits for the control-bit field. Alternatively, the character field and control-bit field could also be joined together to form a 12-bits long bit-vector, in which every bit functions independently as an individual bit.

2.3.2.2 The Bit Select Unit (BSU)

The BSU interprets the control functions of IDR according to the specified Associative Machine Instruction (AMI) and tranfers the appropriate data to the AMA for SEARCH and WRITE operations. Since the BOAP has no mask register, there can be no explicit data masking during SEARCH or WRITE operations, instead characters and control-bits are represented in tertiary logic which allows implicit masking of bit-columns: conditional masking and data complementation. Bit serial processing can be achieved by masking all bits except the one of interest.

2.3.2.3 The Word Select Unit (WSU)

The WSU provides the actual programmable hardware mechanism to implement the tag manipulations and word-row(s) activations. It allows the propagation of activities between word-rows, and provides the linking between data fields to enable the representation of a wide range of data structures. The operations which the

WSU allows are as follows:

1) Activates matching or mismatching word-rows.

2) Activates adjacent word-rows and groups of word-rows of matching or mismatching word-rows.

3) Isolates a single word-row or group of word-rows for activation from a range of mapping functions.

2.3.2.4 The Microprogrammed Control Unit (MCU)

The MCU issues the micro-order (the actual hardware signals) to drive the hardware of BSU and WSU. Since every API is eventually translated into the primitive associative process (either SEARCH or ACTIVATE-READ/WRITE operation) in the form of an Associative Machine Instruction (AMI), MCU provides the microprogrammed interface between hardware and AMI.

2.3.2.5 The Match Reply Register (MMR)

The MMR indicates the presence of one or more set tags in the specified Tag Register (TR1 or TR2) to provide feedback to external control logic for conditional branch operations.

2.3.2.6 The Data Output Conflict (DOC)

The DOC is used to indicate any occurrence of multiple-responses[26] in a SEARCH operation. This will enable the isolation of matched word-rows before reading into ODR.

2.3.3 The Technology of Fabricating BOAP

Recent developments in microelectronics have revolutionized computer design, but how can the properties of VLSI be exploited to build computational structures? Our discussion at this point will focus on two aspects of computer design: the chip layout and VLSI architecture.

A) The Chip Layout Level:

The Von Neumann's design philosophy was adopted in an era of computer technologies in which wires were cheap and switching elements were expensive. However, VLSI technology has reversed this cost situation, making switching elements essentially free and leaving wires as the only expensive component. In today's technology, the area of a circuit devoted to communication between elements far exceeds the area devoted to switching elements, and the communication delays are much longer than logic delays[27]. In fact, many of the design constraints (i.e. Layout topology, Speed, Power dissipation ...etc) which constitute the characteristic of VLSI technology, could to a certain extent be relaxed by minimization of communication paths on the chip:

1) The Layout Topology:

In conventional computer design, switching theory is used as a tool by the designers to formulate logic networks with minimum number of logic gates. However, this approach is less useful in the VLSI design environment where the costs of testing, packaging and inter-connecting integrated circuits are much more important than the manufacture of the circuits themselves. If the topology of interconnection paths is not carefully controlled, the space required for them grows more than linearly as the number of logic elements to be connected is increased: bigger systems require more wires, which are on the average also longer, therefore, to interconnect twice as many randomly placed devices requires four times as much communication space[27]. Hence, it is obvious that controlling the chip layout topology is essential in the design of VLSI. If connections can be made to follow regular patterns, they can be produced by less expensive methods and can also be made to occupy less space and so be faster. Carver Mead and Lynn Conway[28] have developed a

"Chip floor plans" approach to solve wire-and-interconnection problem: A floor plan is merely a block diagram with blocks drawn to approximate scale and the routing of major buses, clocks, power, ground, and critical signal paths specified in terms of their location and the layer on which they run[29]. It is essential to avoid routing a critical signal from one corner of the chip to another, where its delay may sometimes undo all the careful optimization in other parts of the circuits. Regularized structures interact very heavily with the floor-plan strategy. A regularized structure is difficult to formally define but usually involves a functional block that uses a repeated structure to accomplish a given function. A common example is the Programmable Logic Array (PLA), which is a highly regular structure that performs an arbitrary combinatorial logic function.

2) The Speed Considerations:

Not only do longer communication paths occupy a disproportionate amount of space but also they function more slowly than short ones, due to the transmission delay in the lossy line: For a regular structure, the RC delay can be modelled as a diffusion delay in a distributed RC network in which the delay is proportional to the value of the R and C of each network element and proportional to the square of the number of elements in the network n^2RC[30]. Therefore, in order to drive a signal down a longer path, one must either build a larger driving circuit to provide for the extra power required or suffer the delays of passing the larger amounts of energy through a less powerful driver. More powerful drivers must themselves be driven and are inherently slower than small drivers. The Mead-Conway design style advocates a technique known as "wiring by cell abutment"[31] by which each cell can be interconnected by abutting it with its neighbour(s). The advantages of this approach are that it eases the design task by eliminating random wiring, uses space more efficiently by intermingling logic and buses and by eliminating the extra space absorbed in the intercell wire routing, and helps to improve performance by reducing interconnection lengths.

3) The Power Dissipation:

Before a signal path can be switched from one electrical state to another, the energy stored in the path must be removed and converted into heat. It is quite possible that the switching energy of logic elements required in a given technology and the signaling powers needed to travel down the communication paths may set a upper limit to the complexity of the system that can be build in that technology[27].

Hence, the minimization of communication paths on chip can be achieved by building very regular patterns of interconnection and partitioning processor logic accordingly[27,28]. There is already a trend toward very regular wiring patterns for integrated circuits and the interconnections among circuits[29]. This regularity is desirable not only because it makes the specification simple but also because it efficiently reclaims space for putting more switching elements on the chip.

B) The VLSI Architecture Level:

The communications on VLSI chips brings up an important point: the choice of an appropriate architecture for any computer system is very closely related to the implementation technology. Mead-Conway consider that improvement in architectural style will immediately reduce the design problem by orders of magnitude. Properly designed parallel structures that need to communicate only with their nearest neighbours will gain the most from Very-Large-Scale Integration. Precious time (and thus performance) is lost when modules that are far apart must communicate. For example, the delay in crossing a chip on polysilicon, one of the three primary interconnect layers on an NMOS chip, can be 10 to 50 times the delay of an individual gate[29]. Therefore, the architect must keep this communication bottleneck uppermost in his mind when evaluating possible structures and architectures for implementation at VLSI. The architecture of conventional Von Neumann computer suffers from two limitions[27,28]:

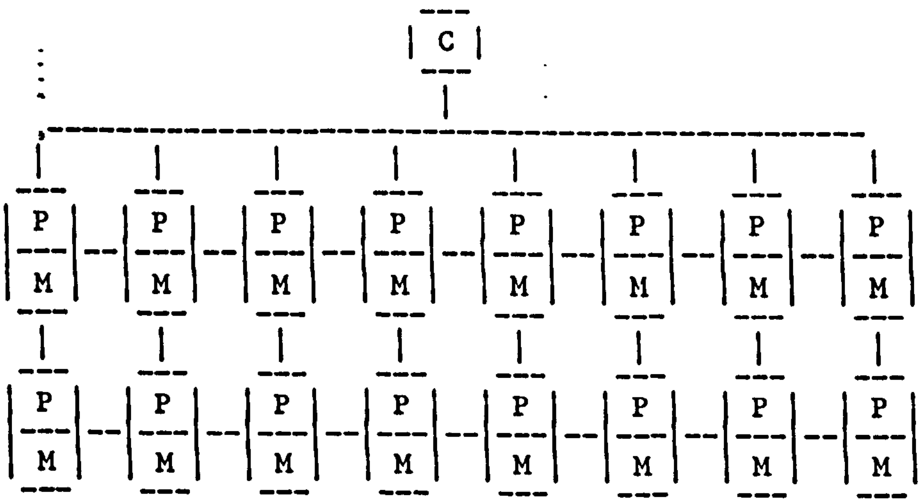
1) The conventional Von Neumann machine provides only a single processor that sequentially fetches and executes instructions; it offers very little opportunity for concurrent processing activity.

2) This SISD type processor is separated from its memory by long communication paths such as buses. The processor fetches an instruction from memory, decodes it, executes it, and repeats the cycle. Many instructions will cause additional references to memory in order to fetch operands or to store results. Therefore, the performance of such a computer depends critically on the method, and the speed of information transmission between processor and memory. As a matter of fact, this is the price we pay for using RAM as data storage. Futhermore, the locational addressing method of the Von Neumann machine, wastes access to many thousands of bits by selecting only a few bits for the CPU, and the size of address bus actually goes up in proportion with the size of computer memory: i.e. MC68000 microprocessor uses 24-bit wide of address bus for addressing 16M bytes of memory versus the 16-bit wide of MC6800 for 64K bytes of memory. This trend will continue so long as people require more and more memory space for data storage and, as a result will inevitabily lead to wider address buses, as well as more pinout problems.

Closer examination of VLSI implementation problems has, however, shown that pin limitations, rather than chip area of logical component limitations, are the major constraint of the VLSI environment[32]. Consider a chip with 16 bit data-pins and 24-bit address pins: the number of required pin connections (ignoring power, ground, and general control) for a single chip implementation is at least 40 pins. Therefore, in order to overcome the pinout limitation on chip implementation, the architecture of the conventional computer will have to be modified.

1) The SIMD Solution

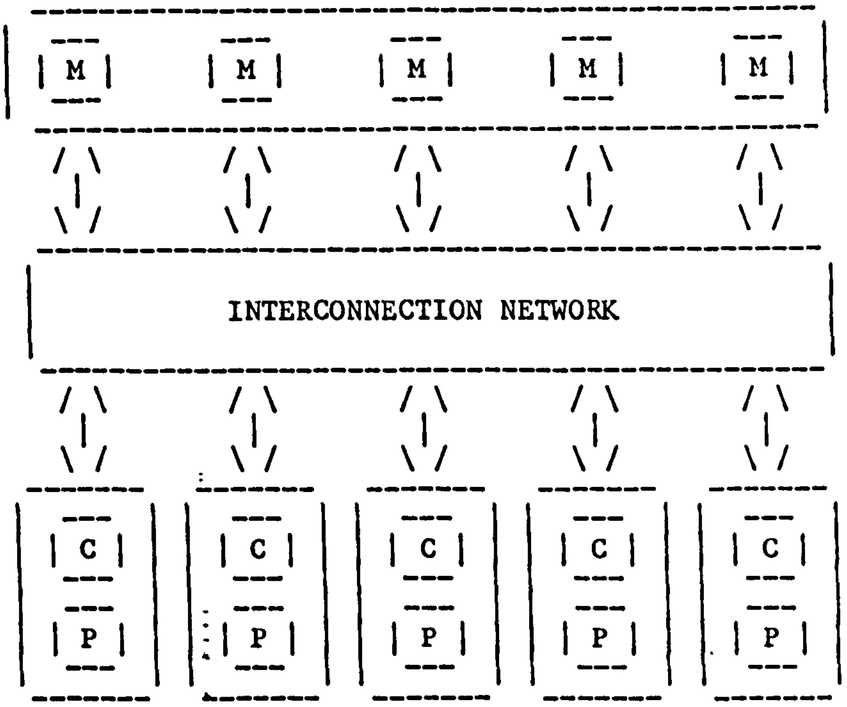
In this solution, memories (M) and processing elements (P) are brought more closely together to avoid the unnecessary movement of data on the buses which sometimes constitute up to 90% of the activities in conventional SISD machines[33].



In this hypothetical machine, many thousands of identical processing elements are brought together to bear on separate parts of a problem under the control (C) of a single instruction sequence in rigid lock-step. These are most suitable for highly regular tasks such as simulation of the weather[24], matrix arithmetic[34,35], and implementation of abstract data types.

2) The MIMD Solution

In this solution, the MIMD machine is one where separate, independent processing elements under separate, self-contained memory; ALU; and control structures, perform independent parts of the task, communicating data and instructions whenever required via a interconnection network[39].



The advent of the microprocessor has, of cause, suggested to many people the possibility of making such a system which consists of thousands of separate microprocessors working in concert on large tasks in the most flexible arrangement for parallel execution of different operations[36]. This system works best when each element can do much processing with out the need to communicate with other elements. But, bottlenecks will develop when tasks require elements to wait for the party line.

The BOAP that we proposed belongs to the category of SIMD architecture, which uses associative memory to minimize the unnecessary memory referencing in the system: Associative memories are incorporated with limited switching elements to process simple operations such as compare, read and write, which can be carried out within the associative memories with out any memory referencing. Nevertheless, like RAM, the Associative Memory Array (AMA), together with its Word Select Unit (WSU), does have an inherently regular word structure.

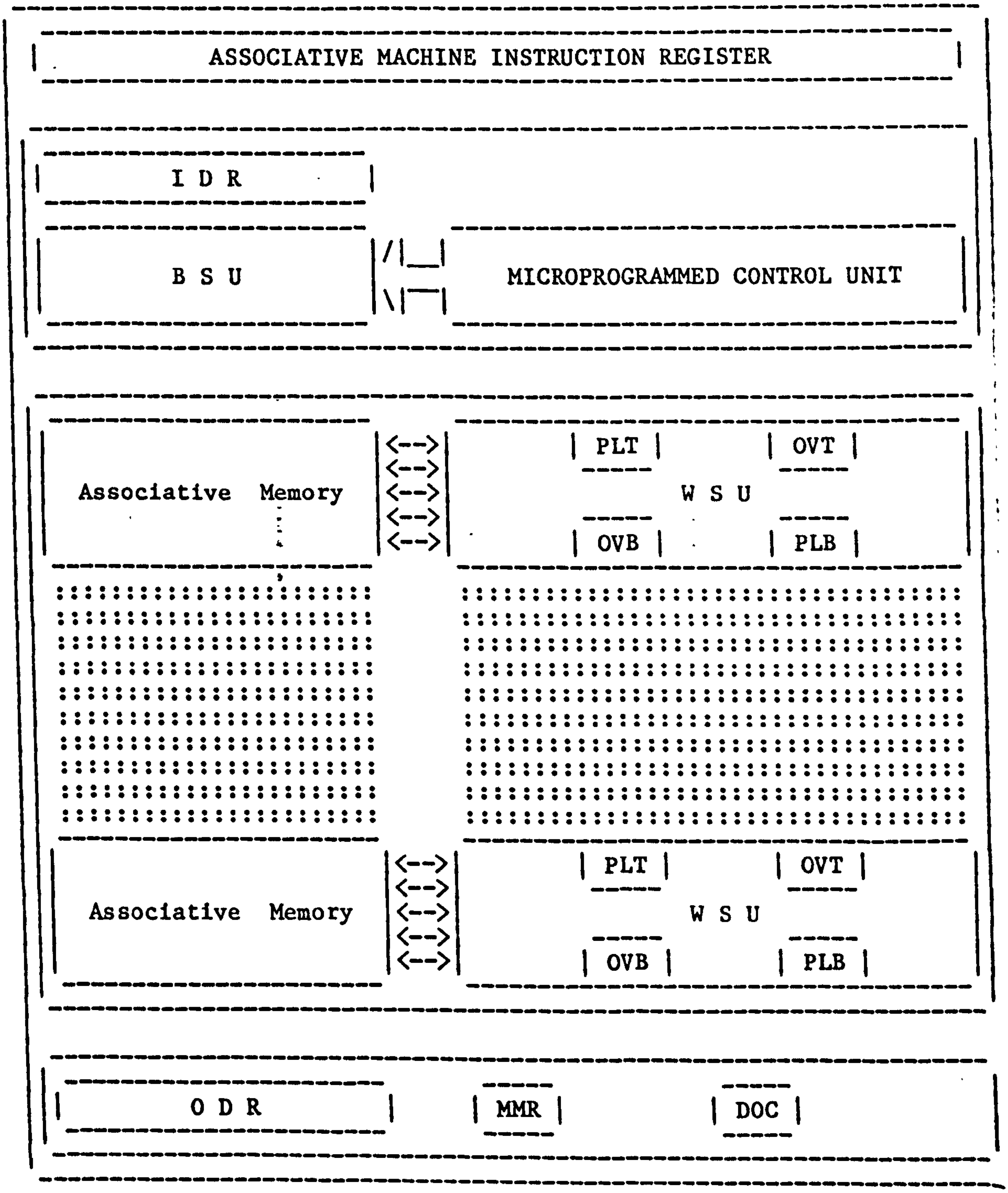


Fig. 2.7 The Chip Organization of BOAM

The proposed chip organization of Byte-Organized Associative Memory (BOAM) can be separated in four parts:

1) The Associative Machine Instruction Register (48-Bits),

2) The Bit Select Module (IDR, BSU and MCU),

3) The Memory Module (Associative Memory and WSU),

4) The Output Register Module (ODR and MMR)

In the careful examination of these four different parts of BOAM, it is revealed that only the fabrication of the memory module (AMA and WSU) has exhibited the feature of repeatability and expandability suitable for VLSI development. However, linking mechanism must be provided in order to build a workable size of AMA from this standard memory chip module, which merely consists of a limited amount of word-row memories and their WSU. The PLT and PLB are designed to serve this purpose: PLT is the propagation link at the T-END (TOP-END) of the AMA and PLB is the propagation link at the B-END (BOTTON-END) of the AMA. Both PLT and PLB can be used to allow propagation or run options to be extended to activate word-rows in adjacent memory modules, without significant loss of execution speed, by allowing the modules to execute the runs in parallel, with their propagation links PLT and PLB set according to the MRR output and/or Overflow Responses (OVT/OVB) of their adjacent memory modules. The OVT/OVB signals the propagation of activities outside the T-END/B-END (TOP END/BOTTON END) of the WSU and provide feedback to the PLT/PLB of adjacent memory modules to proceed with the propagation of word-row activation.

PTT(D) :	PLT1 = 0	PTT(U) :	PLB4 = 0
	PLT2 = OVB1		PLB3 = OVT4
	PLT3 = OVB2		PLB2 = OVT3
	PLT4 = OVB3		PLB1 = OVT2

2.4 SUMMARY

The development of associative processors is based on the search capabilities of associative memory, which is particularly suitable for non-numerical data processing. The Byte-Organized Associative Processor with it short and neat word-length has been chosen as a vehicle for the implementation of abstract data types.

In the light of the LSI/VLSI development, considerations have been given to the Memory chip organization of Byte-Organized Associative Processors. In this investigation, it is established that communications are the major problem on VLSI chip. Carver Mead and Lynn Conway[28] have developed a top-down design methodology for intermodule communication strategies. The key elements of their philosophy are:

- carefully defined chip "floor plans"
- regularized structures
- wiring by cell abutment
- Non Von Neumann architectures
- Mapping high-level functions into silicon

Futhermore, it is also revealed that in addition to its role at abstract data types in the Distributed Computer System, assocative memory offers a good solution for the ever increasing problem of data storage: it is suggested that associative memory is used to replace the conventional RAM for data storage but not program storage, as a result, remove the artificial upper limit of address space (either 16M for 24 bit address system or 64K for 16 bits address system as in the cases of M68000 or M6800 microprocessors), at the same time cut down the unwanted adddress buses and the unnecessary burden imposed on the pinout limit. These potentials plus the ever rising software and personnel costs will eventually lead to the generalization of associative processing.

CHAPTER THREE

THE DESIGN OF THE DISTRIBUTED COMPUTER SYSTEM

Distributed computing systems represent a wide variety of computer organizations, ranging from a star network to a completely decentralized computing system[37]. In all cases, the word "distributed" refers to the fact that processing logic, functions, control, data, or a combination of these of the computing system are distributed to a certain extent[38]. The characteristics of a distributed computing system are as follows:

1) There are a number of hardware processors connected together via an interconnection network.

2) The network provides data (and control) communications between the various processors and provides input and output connections for user interface.

3) Each processor has a number of functional components which can interact with each other to perform system-wise functions such as task sharing and resource sharing.

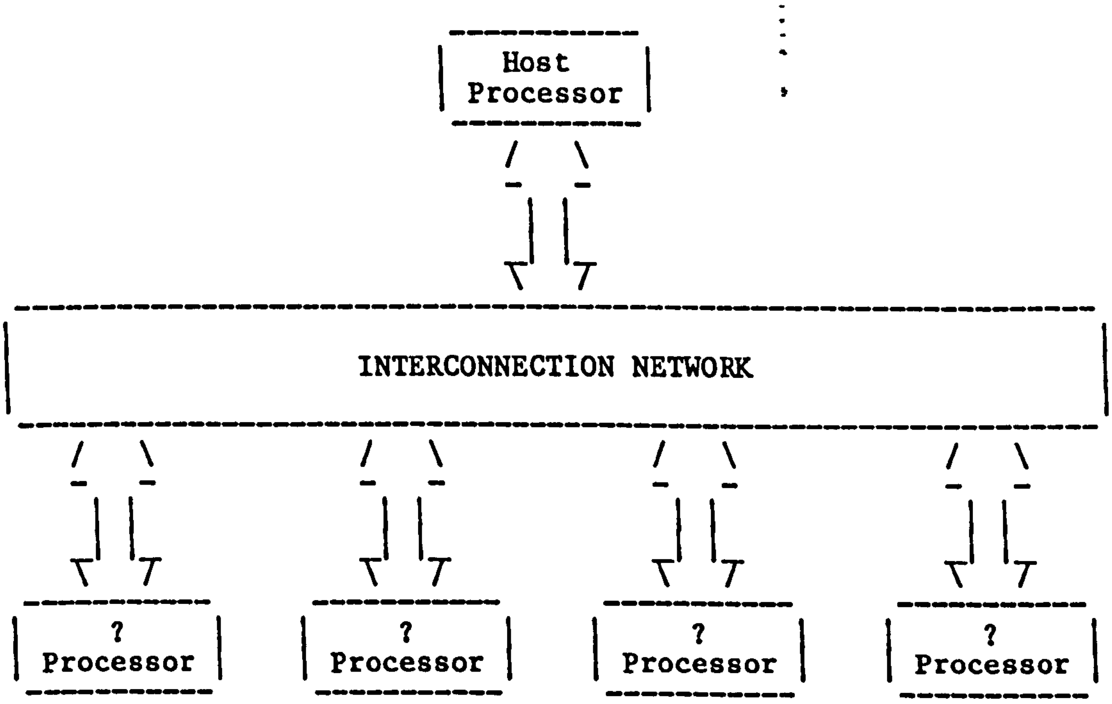


Fig. 3.1 The Organization of Distributed Computing System

Distributed computing systems exhibit extreme flexibility, reliability, survivability and modularity by virtue of the loose coupling between processors:

1) Flexibility:

With appropriate design of the network communications protocol, the total number of nodes (i.e. processors) in the network can be increased or decreased even after initial fabrication. Similarly, a sufficiently versatile computer network communications protocol allows the inclusion of nodes with a wide variety of speeds, computing capabilities, physical configurations, and so on, since the only constraint on the node design is that the interface to the communications network must obey a predetermined protocol.

2) Reliability:

In the distributed system, the individual processors may be assigned to the execution of portions of a large algorithm, followed by a merging of their partial results. Since task assignment is done via software rather than through a specially designed architecture and a fixed hardware configuration[83,84], unassigned processors could be used to achieve good reliability and fault-tolerant processing[36].

3) Survivability and Modularity:

Distributed systems are capable of resisting obsolescence, since the network communications structure may be left intact while some or all of the nodes are replaced or upgraded with newer technology representing more cost-effective or powerful performance.

A large number of topologically different network schemes have been proposed[39-45], all of which posses unique strengths and weaknesses. However, it has not yet been demonstrated through a sufficently large number of actual hardware development efforts which of the network structures is the most flexible. Nevertheless, for the detailed description of various interconnection networks, the reader could perhaps refer to Feng's survey paper[39] which covers a wide spectrum of network configurations.

In that article, Feng begins by an examination of the decisions that designers have to make in terms of operation mode, control strategy, switching methods, and network topology. He then goes on to review the major reseach efforts on the subject during the last few years, and classify them into synchronous and asynchronous models.

1) The Synchronous Models:

In the synchronous category, multiple processing elements and parallel memory modules under one control unit and linked together by an interconnection networks, can handle Single Instructions and Multiple Data streams (SIMD) processing (see the SIMD solution in page 28). Existing examples include Illiac IV[24] and Massively Parallel Processor[22].

2) The Asynchronous Models:

The asynchronous approach for concurrent processing can handle Multiple Instruction and Multiple Data stream (MIMD) processing (see the MIMD solution in page 29). Examples of the MIMD architecture include data flow machines[48-54], and reduction machines[55-60]. The multiple independently controlled processing elements are linked to a number of memory modules by an interconnention network. But, unlike the control unit in the SIMD machine, the activities are coordinated by the coordinator in the interconnention network, which implements the synchronization of processes and smooths out the execution sequence.

3.1 THE COMPUTATION ORGANIZATION OF DISTRIBUTED COMPUTER SYSTEM

The computation organization decribes the way computation progresses in the form of change(s) in the state(s) brought about by executing instructions. In here, we describe how these state changes come to take place by the rules of sequencing and the effect of instructions[46].

1) Fetch Phase : Once selected, the instructions including all necessary operands, are fetched from the memory into the processing element for possible execution, but fetching does not guarantee execution.

2) Examine Phase : each of the instructions previously fetched in the processing element is examined to see if it is executable. The rules for making this decision are different in a variety of architectures. However, if an instruction is executable, it is passed on to the next phase for execution; otherwise, the examine phase may delay the instruction or attempt to coerce arguments so as to allow execution.

3) Execute Phase : at the execute phase, the instruction is actually executed, and the result is then used to change the state(s) of the computation.

3.1.1 The Classification of Computation Organization

The mechanisms and rules which govern the fetching, examining and executing of instructions are often so unique that a clear distinction can always be drawn among different types of computation organization.

1) Control-Driven :

In the control-driven computation organization, it is best described by the well known fetch-execute control cycle of the Von Neumann architecture: once selected, the instruction and its associated operands are fetched into the processing element ready for the execute phase without being checked by an examine phase. In other words, the examine phase is redundant in control-driven environment, and the progress of computations are marked by changes of states in shared memory (global state). The advantage of control-driven computation is full control over sequencing. But a corresponding disadvantage is the burden of this imperative approach in having to specify details of how to solve a problem step by step. Futhermore, programming discipline is needed to avoid run-time errors which are harder to prevent and detect, due to the twin generalities and dangers of control-driven computation to execute data as a program.

2) Content-Driven :

The computation organization of content-driven architecture is very similar to the control-driven organization except that the examine phase is included in the three-phase computation cycle. In this computation organization, memories and processing elements are brought together to avoid the unnecessary movement of operands, to such an extent that fetching of operands is almost minimal. At the fetching phase, only various parts (examine phase and two execute phases) of the instruction are fetched from the program store into the control unit of the system. At the examine phase, a search is conducted on the content of every word-row of the memory to select the records (or data structures) concerned and to decide which execute phase of the instruction is to be executed. The result of execution is marked by state changes at all selected word-rows of the memory. The advantages of content-driven computation are minimum operand fetching, and the parallelism within the computation cycle obtained from the multiple data stream organization of SIMD architecture, which makes it the best candidate for logic programming. However, disadvantages do exist when dealing with non-structured operations such as expressions and multi-tasking.

3) Data-Driven :

In the data-driven computation organization, no explicit control is available. Instructions when fetched are passively waiting for some combination of their arguments to become available before executions can take place. Hence, the key factor governing execution is the availability of data. Conceptually, all instructions in the program are fetched into the processing elements at the beginning of the program, each instruction has a processing element allocated to it continuously, just waiting for arguments to arrive. The examine phase then implements the so called firing rule which requires all arguments (data) to be available before proceeding to execution. At the execute phase, each instruction consumes its arguments and places a result in each successor instruction. The advantage of data-driven computation is that instructions are executed as soon as their arguments are available, making way for a very high degree of implicit parallelism in the program organization. This makes it particularly suitable for processing expressions where the sequencing of the program organization is determined solely by operator precedence. However, the disadvantages are that some of the firing rules in the examine phase may be too restrictive and wasteful causing a wait for unneeded arguments. For example : content-driven type operations such as IF-THEN-ELSE operator which use only two of its three arguments may be forced to wait for all three before proceed to execute phase. In the worst case this can lead to nontermination through waiting for an unneeded argument, for example an infinite iteration.

4) Demand Driven :

In the demand-driven computation organization, an instruction is fetched into one of the processing elements only when the value they produce is needed by another already selected instruction. In the examine phase the arguments are checked to see whether execution is possible. If it is, the instruction is proceeded to the next phase for execution. Otherwise, the processing element will demand the evaluation of argument(s) until sufficient are available for execution. Logically, this demand consists of spawning one or more subcomputations to evaluate operands and waiting for them to return with a value. The execute phase in demand-driven model involves rewriting the instruction which will return with the arguments needed for the progress of the computation. Only the local state consisting of the instruction itself and those instructions that use its results are changed. The essence of demand-driven organization is that instruction sequencing is driven by the need to produce a result at the outermost level, rather than insisting on following a preset pattern. The advantage of the demand-driven computation organization is that only instructions whose result is needed are executed which make it well suited for the implementation of functional programming, especially in the cases of recursive, and iterative ("LOOP") program constructs: the demand-driven mechanism will synchronize and trigger executions amongst a group of instructions, without the imperative approach of control-driven architecture to specify the detail of sequecing. Nonetheless, it is unable to deal efficiently with expression type operations where every instruction (+, -, *, / etc.) always contributes to the final result: propagating demand from outermost level to innermost is a waste of effort, as naturally, it is the operator precedence that determine the sequencing, and every instruction must be executed.

3.1.2 The Computation Organization of the Host Processor

From the above discussion, it is clear that the control-driven model which has long been predominant, has failed in the highly parallel environment. But would the content-driven, data-driven, or demand-driven computation organizations provide the answer to the question of parallelism? Over the past few years, a number of content-driven, data-driven and demand-driven computation organizations have been proposed[47-60], however, it is by no means certain which is the best candidate to cost-effectively replace the conventional control-driven computation organization in handling the wide spectrum of computing activities. Nevertheless, it is strongly believed that none of the computation organizations that we have known to date is able to cope with the challenge of present day computational problems individually. Perhaps with the careful integration of all four kinds of computation organizations, a more efficient mechanism could be found. It is in the light of this philosophy that the Distributed Computer System is proposed.

Intensive investigations have led us to believe that there are only three kinds of computational activities existing in most of today's computer programming: Namely SISD, SIMD, and MIMD activities. Each of these is best driven by an appropriate computation organization: control-driven for SISD operations, content-driven for SIMD operations, data-driven and demand-driven for MIMD operations.

The control-driven organization is characterized by the lack of an examine stage which implies that the program has complete control over instruction sequencing. Once selected, instructions will always be executed regardless of the state of their operands. This means that there is no concern of the contents of data, no waiting or demanding for arguments. This unique characteristic, has placed the Von Neumann type control-driven computation organization in an undisputable position to be selected as the host processor of DCS, to handle activities such as program sequencing; scheduling; task allocation; and I/O control, within the system.

3.1.3 The Computation Organization of the Associative Processor

The computation organization of the associative processor is organized as a content-driven model: loading of instruction at fetch phase, a SEARCH operation at the examine phase (API 1), followed by MODIFY-READ/WRITE operation at the execute phase (API 234).

```
 -------
  Fetch      LOADING INSTRCTION
 -------
                     |
                     V
                  <API 1>

 Examine             |
                     V
                    / \
         MR=0      /   \    MR=1
                  /     \
 -------         /       \

 Execute   <API 234>  <API 234>

 -------
```

This extraordinary organization can be traced all the way back to the origins of Content Addressable Memory (CAM). Unlike RAM, CAM does not use location to address memory, instead, the content concerned is used to SEARCH (address) for required word-rows in the memory. Similarly, a SEARCH operation is also used in the associative processor to examine the content of arguments before deciding which <API 234> is to be executed. Matched keywords of the arguments are tagged in the Tag Register (TR) making ready for subsequent execution. However, it would not be very useful, if executions can only be carried out on tagged word-rows of the arguments, therefore, a tag manipulation is inserted between the SEARCH and READ/WRITE operations for the activation of word-rows to be extended beyond those tagged word-rows. This allows modification of the logical content of Tag Register to be performed, before the execution of Read/Write function. This modification comprises a two state operation: a CLEAR operation (or a CLEAR-READ/CLEAR-WRITE operation) followed by an actual tag manipulation. Hence, LOAD(fetch)-SEARCH(examine)-MODIFY-READ/WRITE(execute) forms the complete Associative Computation Cycle (ACC).

Timing beat	<API>	<API-STATE>	<AMI>
Beat 0	LOADING INSTRUCTION	API0	
Beat 1	SEARCH	API1	AMI1
Beat 2	CLEAR-READ/CLEAR-WRITE	API2	AMI234
Beat 3	Tag Manipulations	API3	
Beat 4	READ/WRITE Function	API4	

The Associative Computation Cycle (ACC) is assembled in a five beat time sequence as shown in the above table. However, in practice a four beat time sequence in a two-part instruction will be assumed, since the fetch phase of next instruction can always be pipelined with the examine-execute phases of the current instruction.

```
           |<- Beat 1 ->|<- Beat 2, Beat 3 & Beat 4 ->|
           ---------------------------------------------
          |            ---> <API234> ----              |
          |       MR=0|                  |             |
LABEL --> |-<API1> --->                   ----------->|--> NEXT
          |       MR=1|                  |             |
          |            ---> <API234> ----              |
           ---------------------------------------------
```

3.2 THE PROGRAM ORGANIZATION OF THE DISTRIBUTED COMPUTER SYSTEM

In the previous section, the computation organization of a single instruction was discussed. In this section, the scope of investigation will be broadened to consider relations between instructions: i.e. How do they communicate in terms of data mechanism? How one instruction causes the selection of another instruction in terms of control mechanism? Thus what determines the pattern of control to form the organization of a computer program? The term program organization is used here to cover the way machine code programs are represented and executed in a computer architecture[46].

1) The Control Mechanism :

The control mechanism defines the propagation of instructions, and thus the control pattern within the total program.

A) Sequential : where a single thread of control, signals an instruction to compute and passes from one instruction to another.

B) Parallel : where more than one thread of controls are actived at an instance, and protocols are also provided for the synchronizing of these threads.

C) Recursive : where control is used to signal the need for arguments, and hence, an instruction is selected for execution when one of the output arguments it generates is required by the invoking instruction. Having executed, control is returned to the invoking instruction.

2) The Data Mechanism :

The data mechanism defines the way a particular argument is used by a number of instructions.

A) By Name : where an argument is known at compile time and a separate copy is generated and placed in each accessing instruction.

B) By Value : where an argument, generated at run time, is replicated and a separate copy is placed in each accessing instruction.

C) By Reference : where an argument is commonly shared by having a reference to it stored in each accessing instruction.

The Distributed Computer System is organized in a sequential Control-Flow program organization, which has a number of common Control-Flow features:

1) The Control Mechanism :

There is a growing belief[54], shared by the author, that since a computer program is a sequence of tasks carefully put together to solve a particular problem by means of a computer system, it is by no means reasonable to suggest that parallelism could be achieved in this inherently sequential program organization. Nevertheless, in so saying, we are not discarding the possibilty of parallelism: parallel operations could be implemented as concurrent processes (procedures) within the sequential Control-Flow computer program. Here, we stress very strongly the word "sequence" in the sense that processes and state changes have to proceed in the right order before the true result can be obtained. Hence, the DCS is based on a sequential control mechanism in which a GOTO type control operator such as program counter is used to direct the flow of control, concurrent processing is achieved by the augmentation of FORK-JOIN type parallel control operators. These parallel operators allow more than one thread of control to be activated at an instance, and also provide the means for synchronizing these threads.

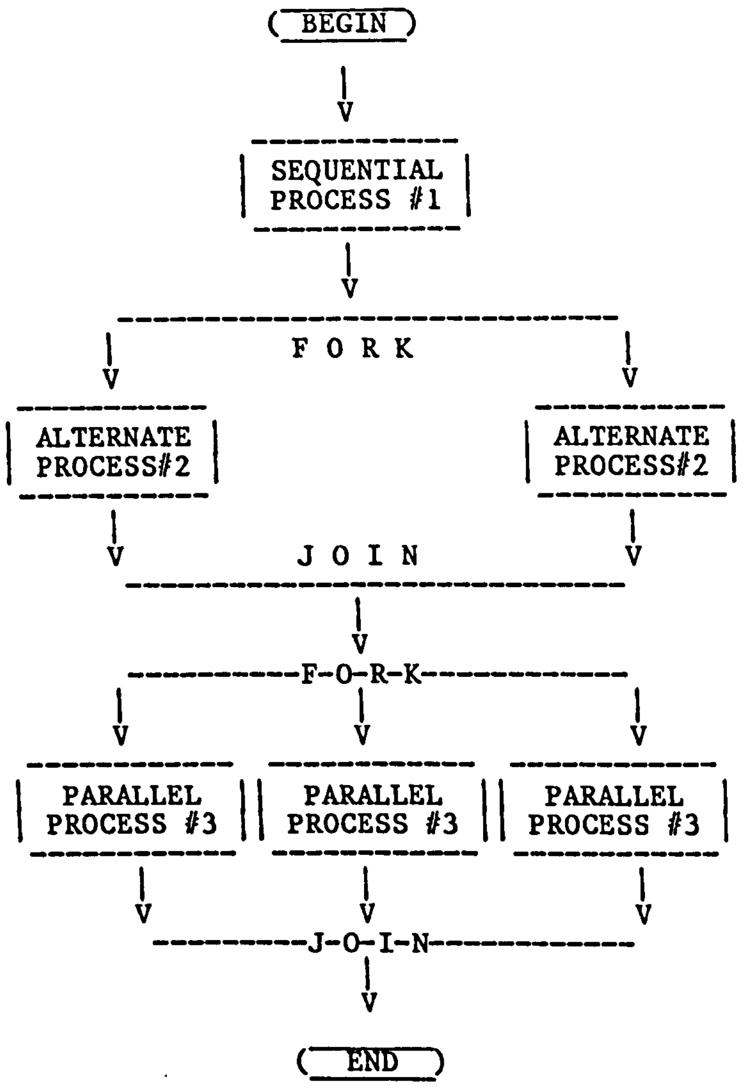


Fig. 3.2 The Control Mechanism of the DCS Program Organization

A process (or procedure)--the fundamental working element in the DCS--is a single instruction, group of instructions, or even a group of other processes, reponsible for the handling of one prescribed activity, which will then be put together with other processes (or procedures) to form a main DCS program.

2) The Data Mechanism :

The basic data mechanism amongst DCS processes is a "by-reference" mechanism, with references embedded in processes being used to access shared memory, in which the effects of changing the contents of a memory cell are immediately available to other processes. Hence, data is passed indirectly between processes via references to shared memory cells. However, within each process, it is usually the "by value" data mechanism that governs the flow of partial results directly from the producer to the consumer instruction, without reference to the shared memory. The reasons for this design are two fold:

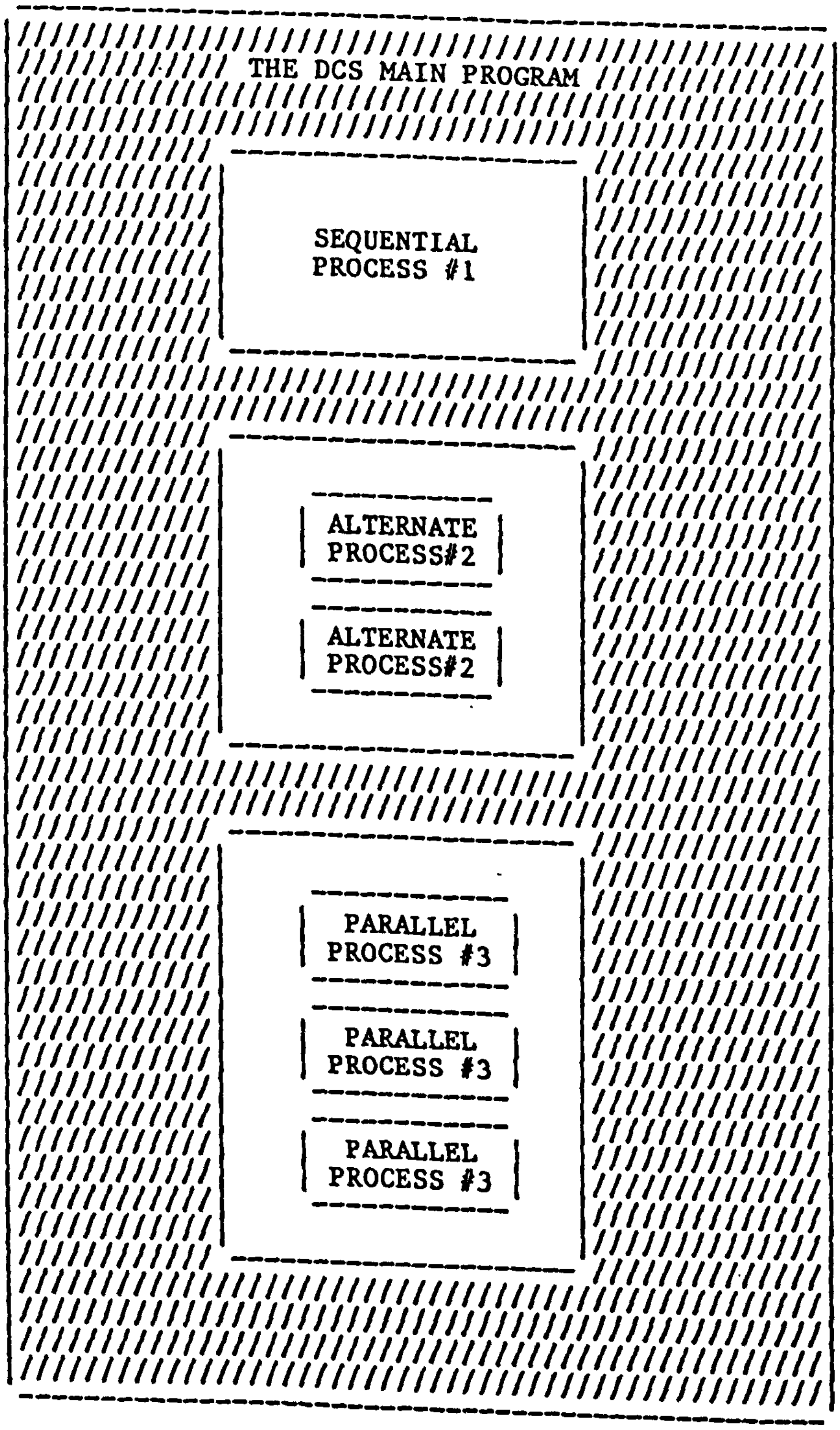
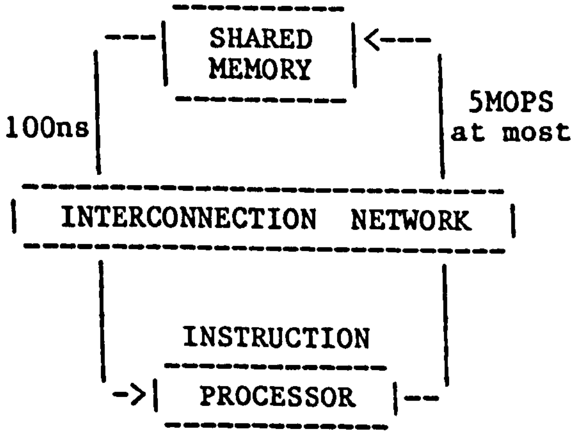


Fig. 3.3 The Data Mechanism of DCS Program Organization

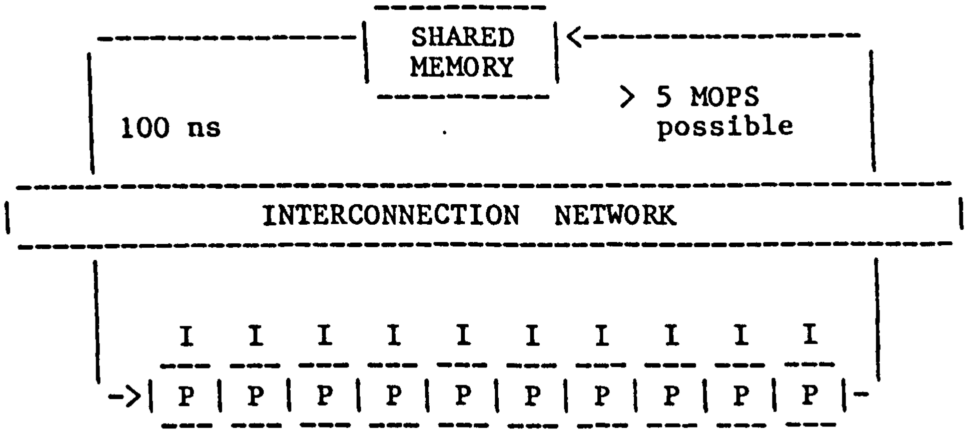
1) Architectural Advantage :

From the architectural point of view, this more modular approach of data mechanism has not only given rise to a cleaner semantics without "side effects", but it has also open the way for multiprocessing: with the "by-value" data mechanism, individual concurrent processes can function in parallel without interfere with each other in clashing for references to the shared memory. Hence, extensive concurrency can be obtained by multiprocessing many partial results in parallel, which otherwise may run into the so called "Von Neumann bottleneck": due to the "by reference" data mechanism of Von Neumann architecture, system performance is critically influenced by the I/O bandwidth of the system. Suppose that the I/O bandwidth between shared memory and processors is 10 million bytes per second. If at least one byte of operand is read from and another byte of result is written back to the memory for each instruction, the maximum rate will b 5 MOPS (Million Operation Per Second), assuming that fetch sequence and execute sequence can be pipelined.



However, this I/O problem will become especially severe when a process of large computation is involved. Orders of magnitude improvement on the throughput are possible only if multiple computations can be perform on multi-processor per I/O access.

computations can be perform on multi-processor per I/O access. Using a "by value" mechanism where data is passed directly from instruction to instruction will make this become a reality.



2) Hardware Advantage :

With the introduction of "by value" data mechanism within each process, DCS is trying to avoid long-distance or irregular wiring which arises from the global communication of "by-reference" mechanism. The only global communication is restricted at the process level. Once inside the process, a self-timing scheme is used for synchronizing neighbouring processors and passing data directly between them. This modular approach will lead to a more organized, more regular, and simpler hardware implementation, making way for the ultimate chip implementation of DCS even more closer to reality.

Looking from the system programmers' point of view, the program organization of the Distributed Computer System will look very similar to any conventional computer, except that the DCS has facilitated APIs into the conventional assembly languages, and hence the Associative Assembly Language (AAL).

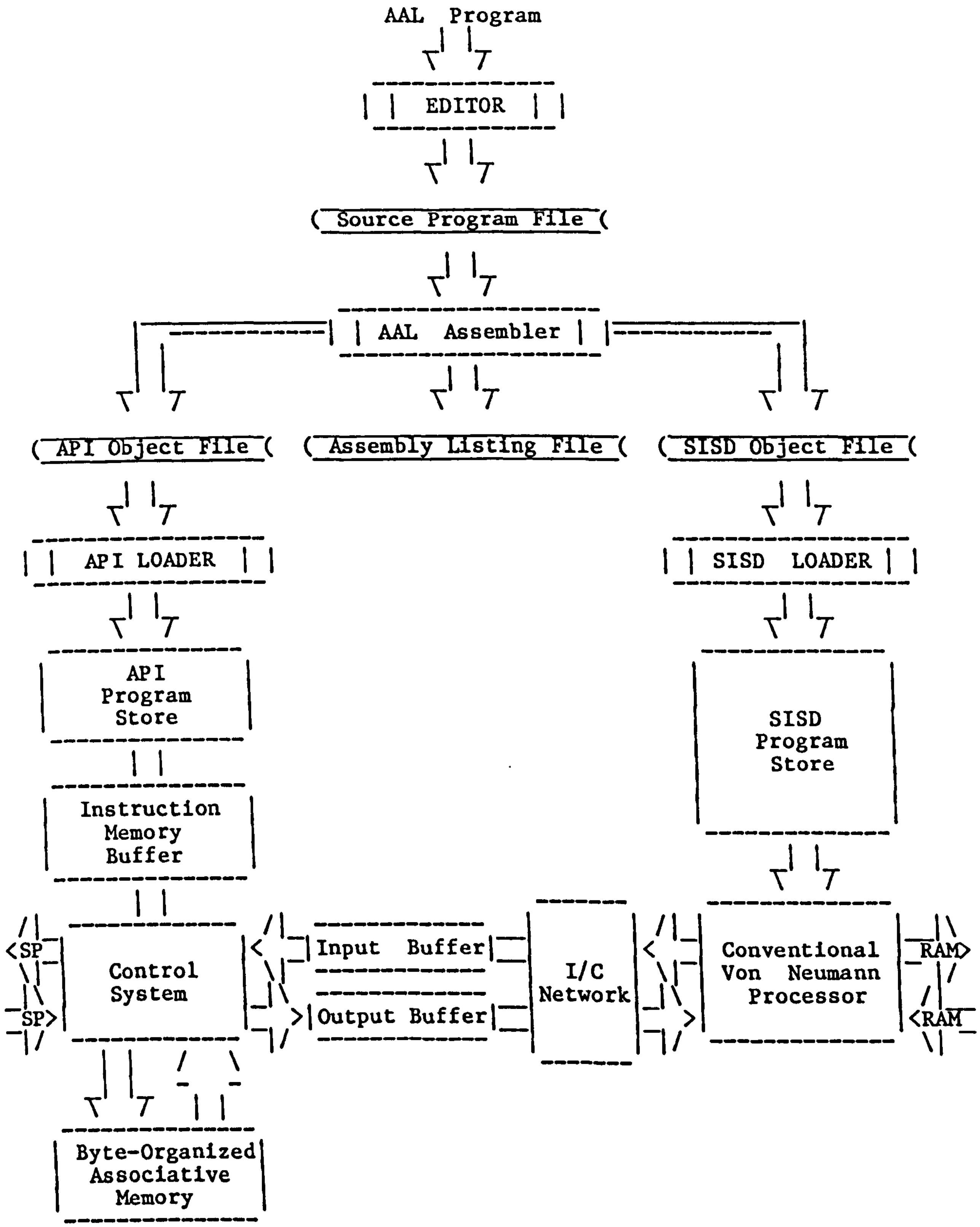


Fig. 3.4 The Program Organization of the Distributed Computer System

The AAL is the super-set of APIs and a SISD assembly language with which the programmer uses to access BOAP and the SISD processor. The choice of SISD assembly language is completely arbitrary, as the preprocessor type design[61] of AAL Assembler will be able to cope with all kind of assembly languages. For the sake of illustration, the Z-80 Assembly Language is used here as an example of an SISD assembly language.

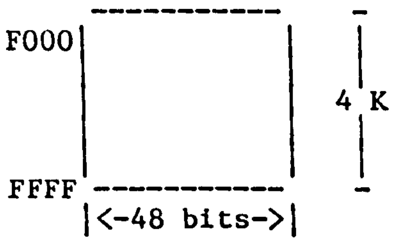
The AAL program is usually input by the programmer into the system editor before being assembled by AAL Assembler which contains a filter to separate APIs from the SISD assembly instructions. Three files are then generated after Assembling.

1) API Object File :

This is a file of APIs in the form of 48-bit machine-code, sometimes, referred to as Associative Machine Instructions (AMIs).

```
C C C C C C C C C C C C T D C|U|
H H H H H H H H B B B B B I M|S|LABEL_0 LABEL_1
7 6 5 4 3 2 1 0 1 2 3 4 V 1 B|D|

011001010101011000000000100000000000000100000010    ;API 0
011001010101100110000000100000000000001100000100    ;API 1
011001010101101000100000100000000000010000000101    ;API 2
011001010110010100001000100000011111111111111110    ;API 3
011001010110011000000010100000011111110111111100    ;API 4
011001010110100110101010100000011111100111111011    ;API 5
```

After assembling, the API object file is loaded into the API Program Store by the API Loader, ready for execution. In the DCS, the API Program Store has artificially occupied 4K memory from F000 - FFFF on the system memory map, in which the most significant 4 bits (B_{12} - B_{15}) of the 16 bits instruction address are used by the BOAP Control Unit as a signal to load APIs from API Program Store.



In the process of assembling, an Association Program Counter (APC) is used to keep track of every address of APIs. When an API is assembled, its address is given by the APC to the MACRO generated SISD LOAD instruction in place of that API in the AAL program, which when executed will trigger the BOAP Control Unit to load the corresponding API into BOAP. At the end of each operation, the flow of control is returned to the Host Processor.

```
e.g. S('T' X1XX)BMR  LABEL0,LABEL1  ;SEARCH FOR 'T',THEN
                                    ;BRANCH TO LABEL0 IF
                                    ;MATCH ELSE LABEL1.
```

will be replaced by the following SISD code

```
     LD              (0F000H),A      ;CALL UPON BOAP TO
                                     ;EXECUTE THE API
                                     ;WHICH STORED IN
                                     ;LOCATION F000.
                                     ;CONTENT OF A IS
                                     ;IRRELEVANT.
```

2) Z-80 Object File :

This is a combined file of the SISD part and the MACRO generated API replacement codes.

```
              .PROC ASSIGNMENT

       SNAME .EQU 0090H
     CONTENT .EQU 0091H
             .ORG 0100H

     ------- ; LOAD THE STRING IDENTIFIER INTO AMA
    |
    |         LD           A,(SNAME)       ; LOAD SNAME IN A
    |         LD           (0F000H),A      ; CALL UPON BOAP TO
    |                                      ; EXECUTE THE API1
    |                                      ; WHICH STORED IN
    |                                      ; LOCATION F000.
    |                                      ; CONTENT OF A IS
    |                                      ; IRRELEVANT.
    |         LD           (0F001H),A      ; EXECUTE API234 WHICH
    |                                      ; STORED IN LOC. F001
    |         DEC          A               ; DECREMENT A
    |
    | -LOOP1  LD           (0F002H),A      ; EXECUTE API1 FROM F002
    ||        LD           (0F003H),A      ; EXECUTE API234 FROM
    ||                                     ; LOCATION F003
    ||        DEC          A               ; DECREMENT A
    | ------  JR           NZ,LOOP1        ; BRANCH TO LOOP1 IF
     -------                               ; NOT ZERO

              ; SET DELIMITER FOR STRING'S IDENTIFIER
              LD           (0F004H),A      ; EXECUTE API1 FROM F004
              LD           (0F005H),A      ; EXECUTE API234 FROM
                                           ; LOCATION F005

     ------- ; LOAD THE VALUE OF STRING INTO AMA
    |         LD           A,(CONTENT)     ; LOAD CONTENT IN A
    | -LOOP2  LD           (0F006H),A      ; EXECUTE API1 FROM F006
    ||        LD           (0F007H),A      ; EXECUTE API234 FROM
    ||                                     ; LOCATION F007
    ||        DEC          A               ; DECREMENT A
    | ------  JR           NZ,LOOP2        ; BRANCH TO LOOP2 IF
     -------                               ; NOT ZERO

              ; SET DELIMITER FOR STRING'S VALUE
              LD           (0F008H),A      ; EXECUTE API1 FROM F008
              LD           (0F009H),A      ; EXECUTE API234 FROM
                                           ; LOCATION F009

              ; TERMINATE THE STRING BY SETTING OVERFLOW BYTE TO 0
              LD           (0F00AH),A      ; EXECUTE API1 FROM F00A
              LD           (0F00BH),A      ; EXECUTE API234 FROM
                                           ; LOCATION F00B

              .END
```

STOP : Next part of the program

3) AAL Program Listing :

This is the full listing of the AAL program plus error messages if any.

```
           .PROC ASSIGNEMT

     SNAME .EQU  0090H
   CONTENT .EQU  0091H
           .ORG  0100H


  ------- ; LOAD THE STRING IDENTIFIER INTO AMA
 |         LD                   A,(SNAME)
 |      -  S('?' XXXX)BMR       STOP,+1
 |     |   RSTTD(S)
 |      -  W(IQF 1XXX)
 |         DEC                  A
 |
 | -LOOP1  S('?' XXXX)BMR       STOP,+1
 ||        RSTTD(S)
 ||        W(IQF XXXX)
 ||        DEC                  A
 | ------  JR                   NZ,LOOP1
  -------


            ; SET DELIMITER FOR STRING'S IDENTIFIER
         -  S('?' XXXX)BMR       STOP,+1
        |   RSTTD(S)
         -  W('$' XXXX)


  ------- ; LOAD THE VALUE OF STRING INTO AMA
 |          LD                   A,(CONTENT)
 | -LOOP2   S('?' XXXX)BMR       STOP,+1
 ||         RSTTD(S)
 ||         W(IQF XXXX)
 ||         DEC                  A
 | ------   JR                   NZ,LOOP2
  -------


             ; SET DELIMITER FOR STRING'S VALUE
         -   S('?' XXXX)BMR       STOP,+1
        |    RSTTD(S)
         -   W('#' XXXX)


              ; TERMINATE THE STRING BY SETTING OVERFLOW BYTE TO 0
         -    S['?' XXXX]         STOP,+1
        |     RSTTD(S)
         -    W[10000000 XXXX]

              .END
```

STOP : Next part of the DCS program

3.3 THE MACHINE ORGANIZATION OF DISTRIBUTED COMPUTER SYSTEM

The term machine organization is used here to cover the way a machine's resources are configured and allocated to support a program organization. An examination of various program organizations under development reveals three basic classes of machine organization[46].

1) Centralized Machine Organization :

Centralized machine organization consists of a single processing element (P), control unit (C), and memory resource (M).

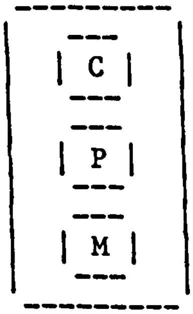


Fig. 3.5 Centralized Machine Organization

The processing element also contains a set of high-speed registers, notably Program Counter (PC), which points to the next instruction to be executed, and Instruction register (IR), which holds the instruction currently being executed. Program execution of the centralized machine organization proceeds in a SISD fashion with the PC keeping trace of the program sequencing. It views an executing program as having a single active instruction which passed execution to a specific successor instruction. This is clearly the machine organization for the familiar Von Neumann sequential control-flow Computer.

2) Packet switching Machine Organization :

Packet switching machine organization[48] consists of a circular instruction execution pipeline of resources in which processing elements and instruction memory unit are interspersed with "pools of work" interconnected by various networks. The organization views an executing program as a number of independent information packets, all of which are conceptually active, which may split or merge. Each packet that is ready to be processed is placed with similar packets in one of the pools of work. When a resource become idle, it takes a packet from its input pool, processes it , places the modified packet in an output pool, and then returns to the idle state. Parallelism is obtained either by having a number of identical resources between pools, or by replicating the circular pipelines and connecting them by the communications[46]. This feature has in fact made packet switching machine organization the favourite candidate for the implementation of data-flow program organization[48,49,51,52,53,54].

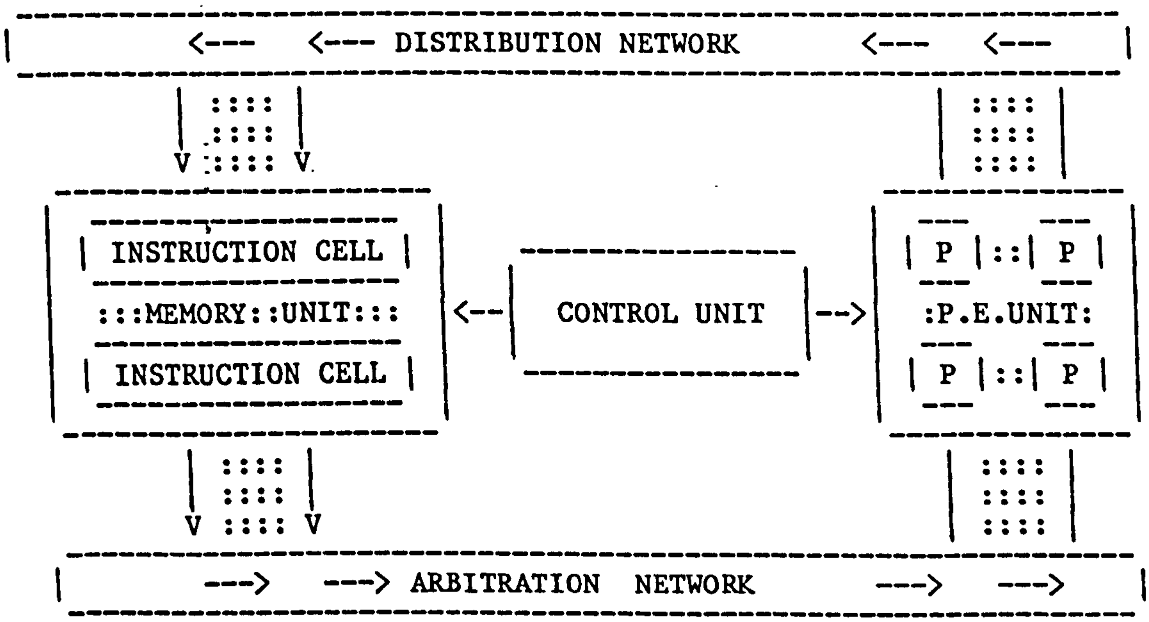


Fig. 3.6 Packet Switching Machine Organization

The Packeting Switching machine organization shown in Fig. 3.6 consists of five major units:

A) Memory Unit, consisting of Instruction Cells that hold the instructions and their operands.

B) Processing Element Unit, consisting of processing elements that perform operations on instruction packets.

C) Control Unit, Co-ordinating the transmission protocol between the Memory Unit and the Processing Element Unit.

D) Arbitration Network, delivering executable instruction packets from the Memory Unit to the Processing Element Unit.

E) Distribution Network, delivering data packets from the Processing Element Unit to Memory Unit.

3) Tree Machine Organization :

Tree machine organization consists of identical resources organized as a regularly structured hierarchy[57,58] such as a tree, as shown in Fig. 3.7.

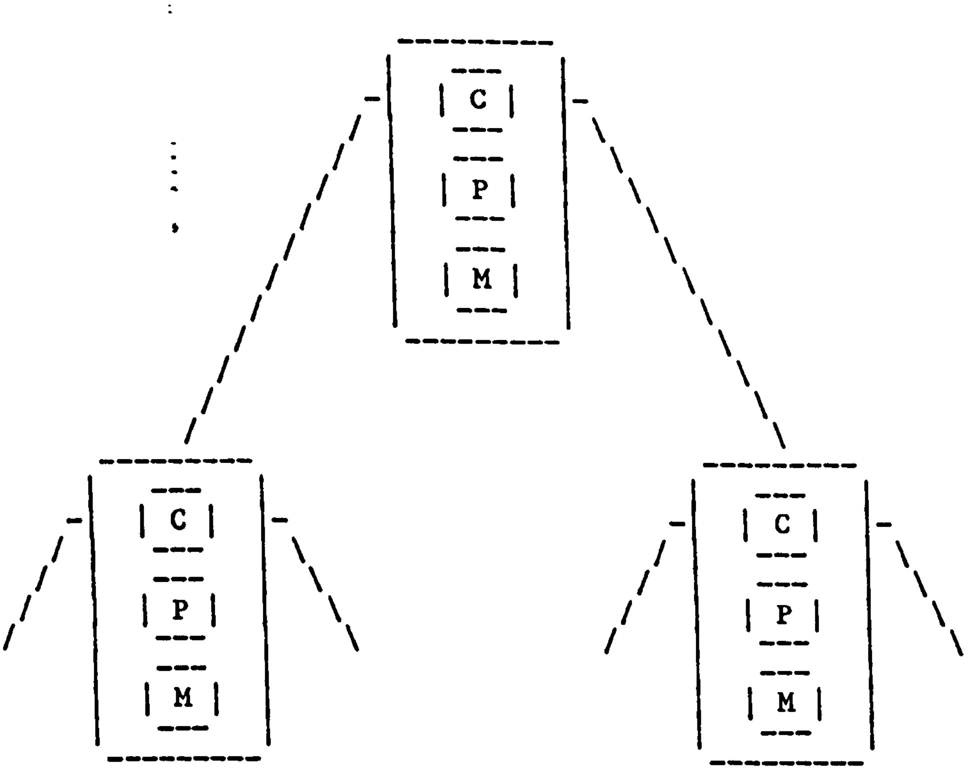
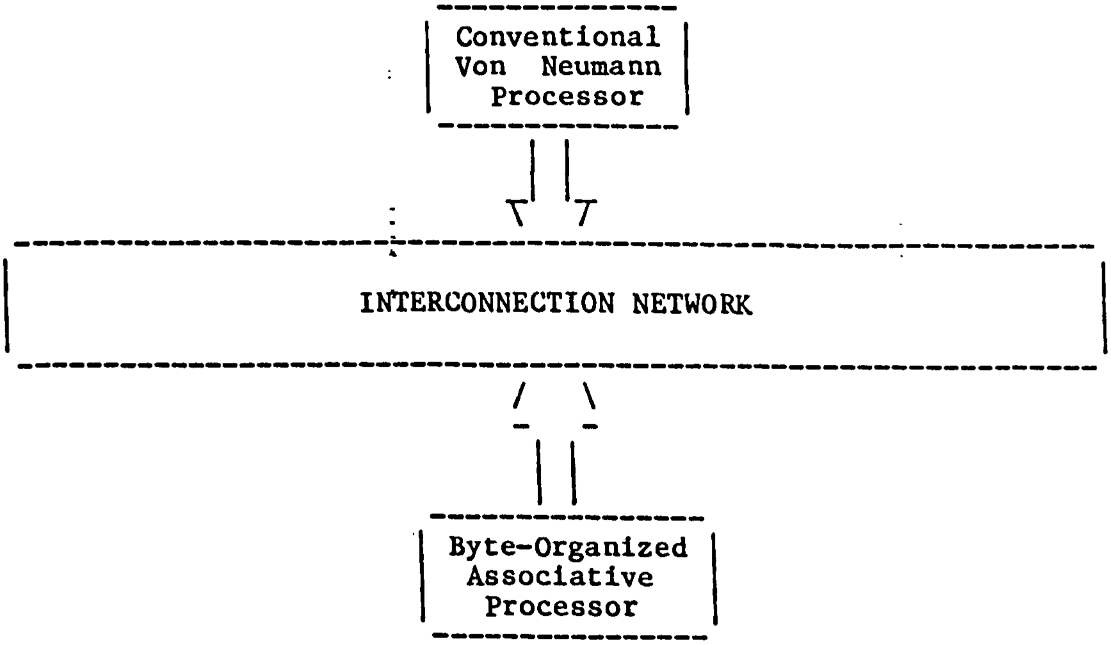


Fig. 3.7 Tree Machine Organization

Each resource contains a processing element (P), control unit (C), and memory capability (M). The organization views an executing program as consisting of one large nested expression which is then partitioned into the collection of hierarchically organized resources. Execution is by a substitution process, which traverses the program structure and successively replaces reducible expessions by others that have the same meaning until a constant expression representing the result of the program is reached. This machine organization seems most applicable to supporting the reduction form of program organization.

The Distributed Computer System is configured as centralized machine organization, with its host processor virtually a conventional Von Neumann machine, and its associative processor organized in a central control SIMD architecture.



The Von Neumann processor is used here mainly as the host of the system to co-ordinate activities such as program sequencing, scheduling, task allocation and I/O control within the network, in addition to the implemention of the conventional SISD operations. The Byte-Organized Associative Processor (BOAP), on the other hand, is integrated in the system to deal with the implementation of abstract data structures. Communications are provided via the Interconnection Network. But, due to the incompatibility of their machine-code instruction formats, functional components of both processors are stored separately in two different program stores.

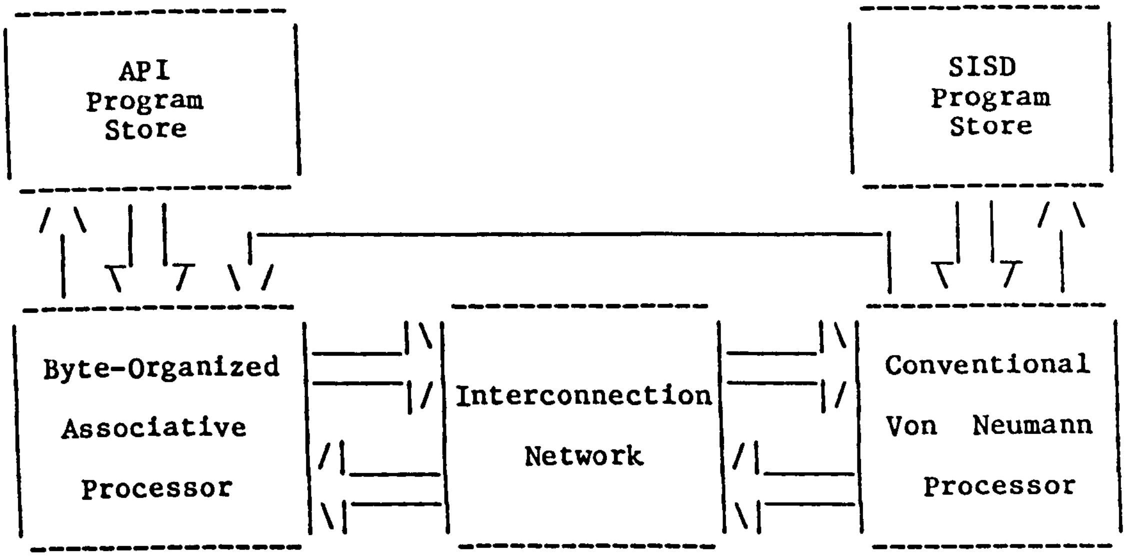

API Program Store
SISD Program Store
Byte-Organized Associative Processor
Interconnection Network
Conventional Von Neumann Processor

Fig. 3.8 The Distributed Computer System

3.3.1 The API Program Store

The API Program Store is consist of a 48-bit word AMI memory, a Associative Program Counter (APC), an Address Control Unit, a Associative Machine Instruction Address Register (AMIAR), and a Associative Machine Instruction Register (AMIR). The API program is kept in a 48-bit word AMI memory after being assembled into Associative Machine Instruction. Loading is done by the Host Processor via the Interconnection Network. Requests for an AMI comes from the BOAP Control System in the form of an AMI address, which it is then used together with the content of APC to calculate the absolute address of the AMI and load it into the AMIR.

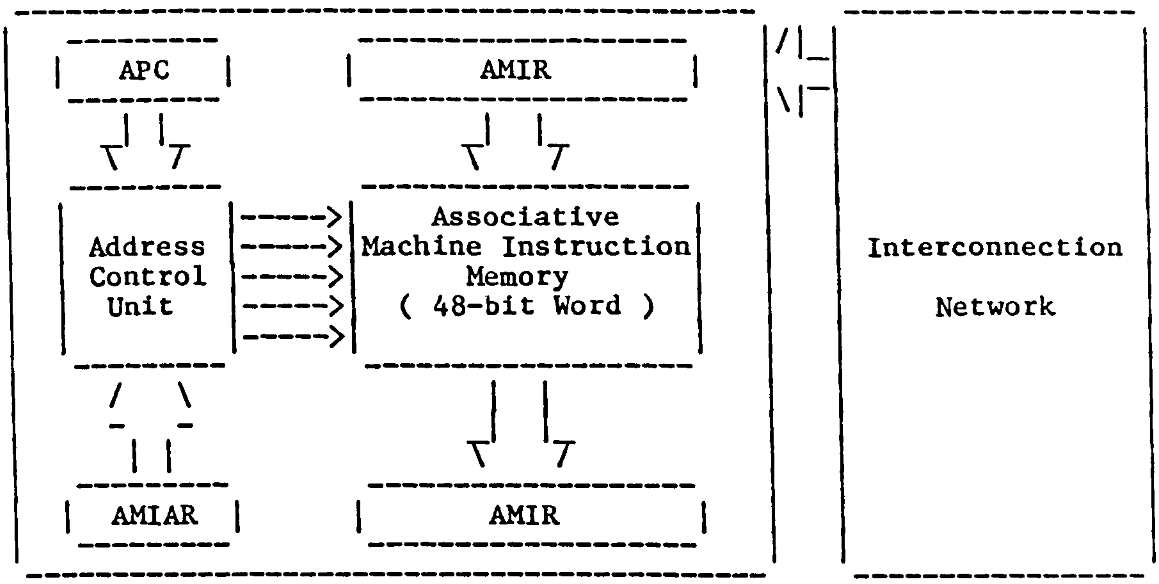
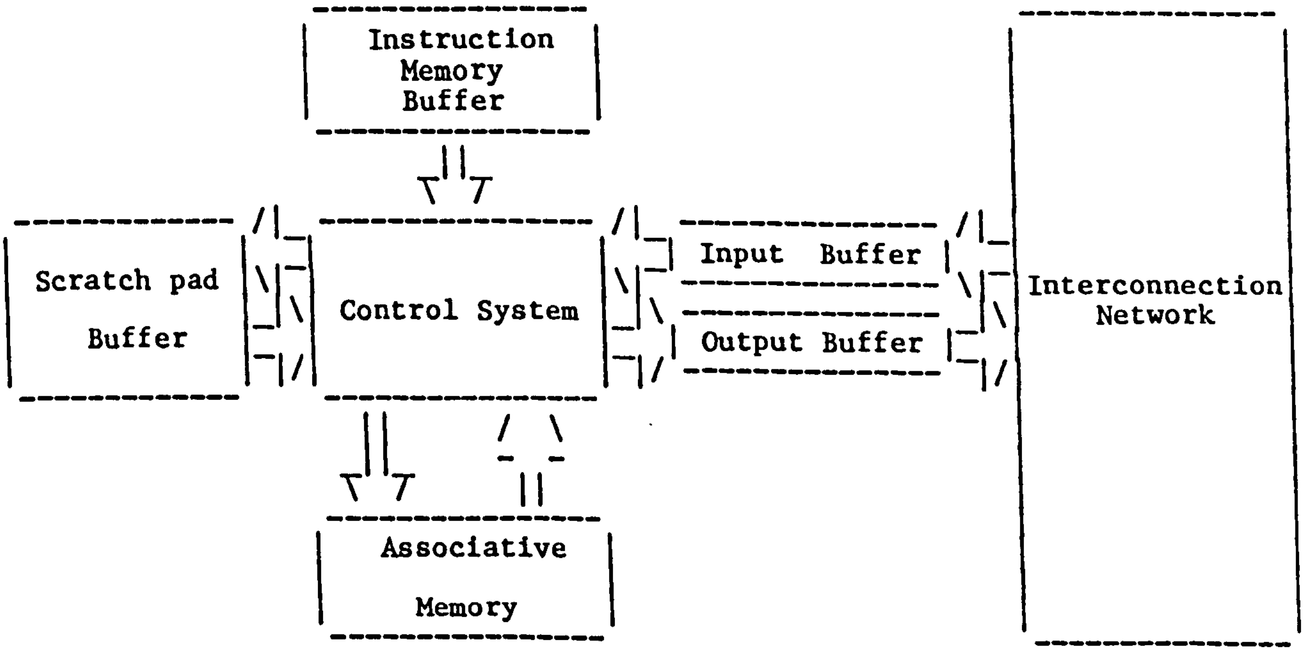


Fig. 3.9 The API Program Store

3.3.2 The Byte-Organized Associative Processor

In the reproduction of Fig. 2.6 (the system organization of BOAP), it is shown that the Control System is like the host and interconnection network within BOAP, that co-ordinates activities and data transfer within the associative processor.



3.3.2.1 The BOAP Instruction Memory Buffer

The BOAP Instruction Memory Buffer is a two register memory block that contains the next two alternative examine parts of the current AMI, to support the "Pipelining" of the Fetch and Examine-Execute Cycles.

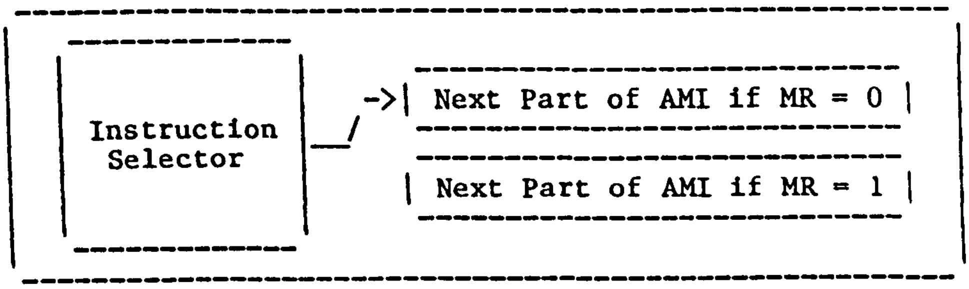


Fig. 3.10 The BOAP Instruction Memory Buffer

The <AMI1> Instruction Format

The <AMI1> of BOAP uses a 2-address instruction format[62] to address the <AMI234> parts (examine phases) of the instruction.

```
|<---------------------- AMI 1 ------------------------>|
1           25   26  27    29  30  33              41   48
---------------------------------------------------------
| Wd Spec |TBV|      |DI1| CMB |000|    Label-0     |Label-1|
---------------------------------------------------------
```

These two addresses (Label-0 and Label-1) of the current <AMI1> are sent to the Address Control Unit of the API Program Store, for the fetching of next two alternative parts (<AMI234>) of this AMI to be loaded into the BOAP Instruction Memory Buffer. The destined next <AMI234> is pending on the outcome of the Match Reply (MR): If MR = 0, the <AMI234> in the upper register will be loaded into the Control System for execution, otherwise, the <AMI234> in the lower register will be chosen.

The <AMI234> Insturction Format

The <AMI234> uses a 1-address instruction format for the <AMI234> part of the instruction to specify the addresses of its next instruction. Since only one next instruction is involved, it is fetched and loaded into both upper and lower registers of the Instruction Memory Buffer.

```
|<-------------------- AMI 234 ----------------------->|
1            25   26  27    29  30  33    37  39    41     48
----------------------------------------------------------------
| Wd Spec |PF | R/W |DI4| CMB |USD| ACD |DI2| CLEAR | Label |
----------------------------------------------------------------
```

3.3.2.2 The BOAP Input Buffer

The transfering of data between BOAP and Host Processor is provided by the Input Buffer and Output Buffer. Both buffers are structured as a 1K x 12-bit RAM, but function as a FIFO (First In First Out) queue.

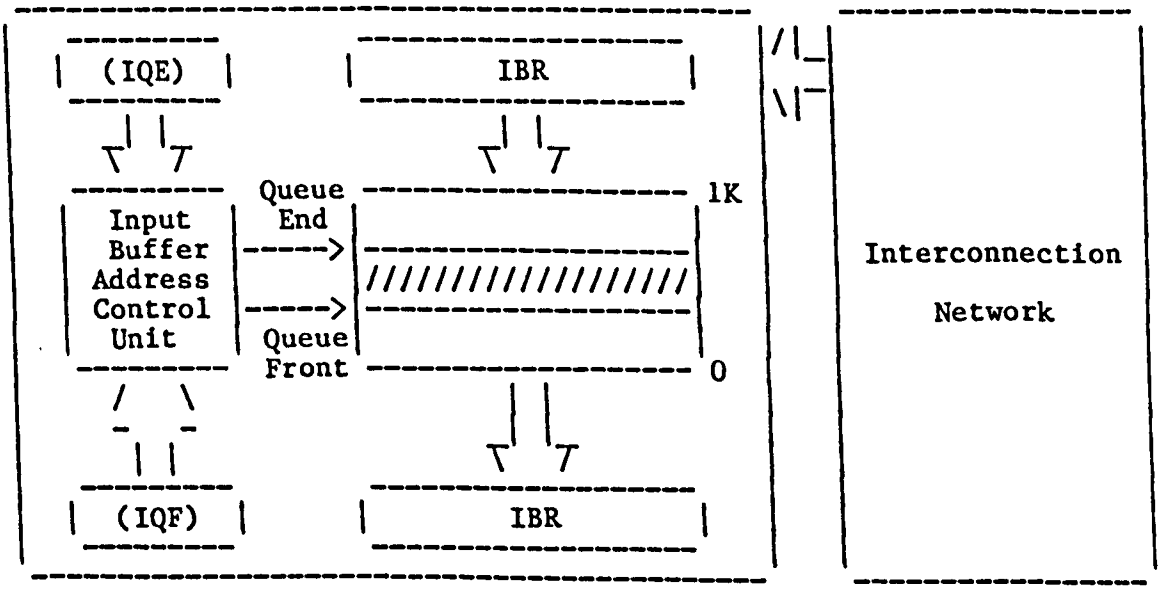


Fig. 3.11 The BOAP Input Buffer

The Input Buffer deals with the incoming data traffic from the Host Processor via Interconnection Network: data is placed at the end of the Input Buffer Queue by the Input Queue End (IQE) pointer as it comes in, and later is transfered into the Control System under the control of the Input Queue Front (IQF) pointer. Both IQE and IQF pointers are always reset back to 0, whenever they have gone beyond the upper limit of the Input Buffer.

3.3.2.3 The BOAP Output Buffer

The BOAP Output Buffer is organized similarly to the FIFO queue of the Input Buffer. However, it deals with the outgoing data traffic from the BOAP to the Host Processor via the Interconnection Network.

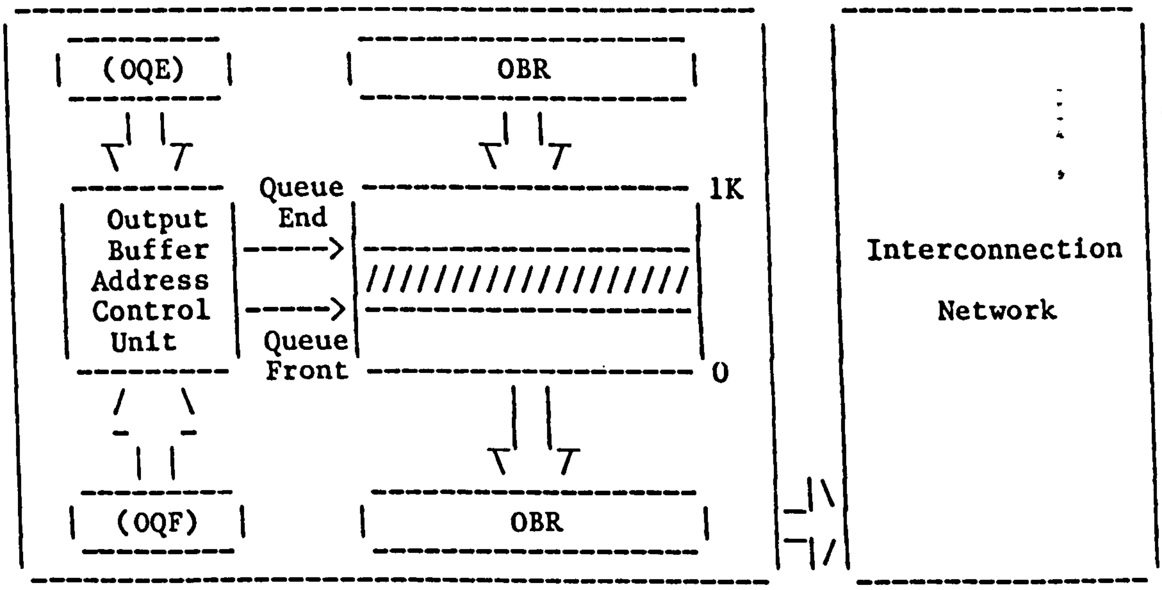


Fig. 3.12 The BOAP Output Buffer

The data from BOAP is placed at the end of the Output Buffer Queue by the Output Queue End (OQE) pointer, and then transfered to the Host Processor via the Interconnection Network by control of the Output Queue

Front (OQF) pointer. Both OQE and OQF pointers are always reset back to 0, whenever they have gone beyond the upper limit of the Output Buffer.

3.3.2.4 The BOAP Scratch Pad Buffer

The BOAP Scratch Pad Buffer is used as a working storage for data transformations and data transfer within BOAP. It is also structured as a 1K x 12-bit RAM block. Data usually comes in from the ODR of the Byte-Organized Associative Memory via the Control System, and vice versa into the IDR of BOAM.

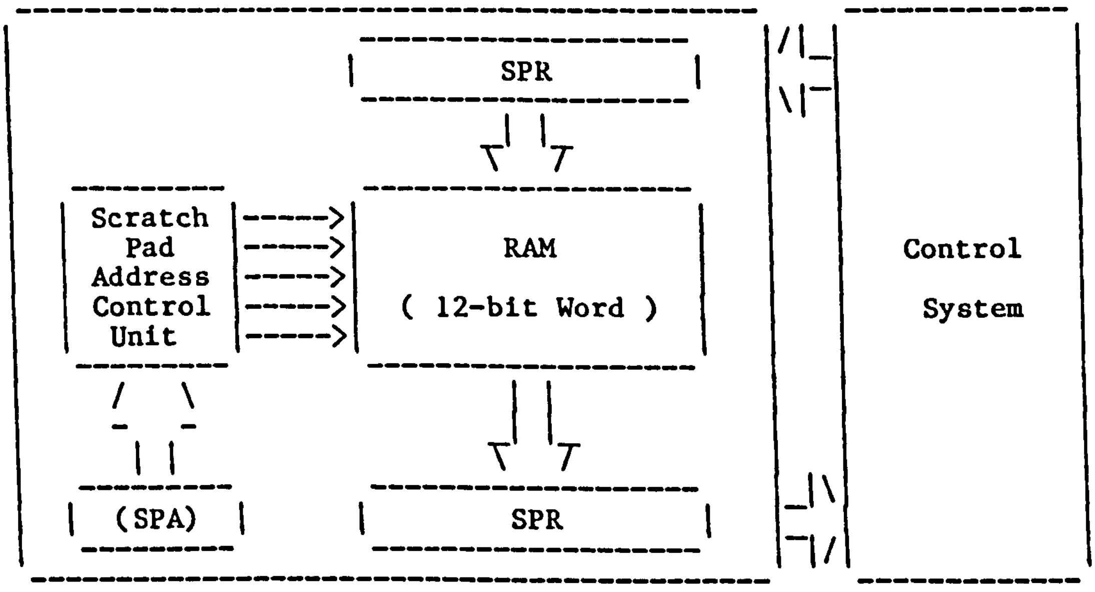


Fig. 3.13 The BOAP Scratch Pad Buffer

3.3.2.5 The BOAP Control System

The BOAP Control System is the host within the associative processor that oversees and co-ordinates activities such as sequencing; scheduling; and I/O control of Assocative Machine Instruction (AMI), within BOAP.

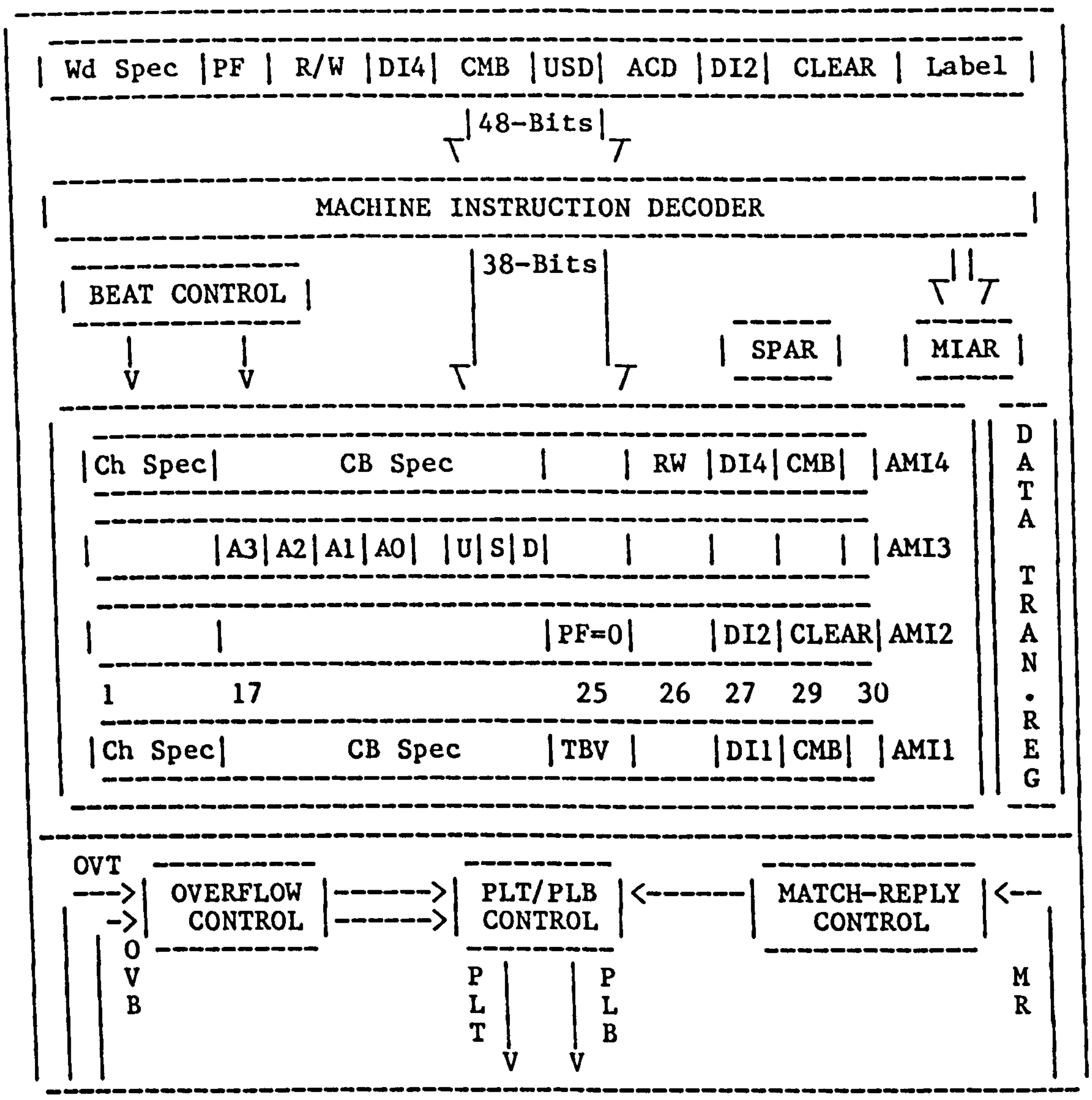

Wd Spec | PF | R/W | DI4 | CMB | USD | ACD | DI2 | CLEAR | Label
48-Bits
MACHINE INSTRUCTION DECODER
BEAT CONTROL
38-Bits
SPAR
MIAR
Ch Spec | CB Spec | | RW | DI4 | CMB | AMI4
A3 | A2 | A1 | A0 | U | S | D | AMI3
PF=0 | DI2 | CLEAR | AMI2
1 17 25 26 27 29 30
Ch Spec | CB Spec | TBV | DI1 | CMB | AMI1
DATA TRAN. REG
OVT
OVERFLOW CONTROL
PLT/PLB CONTROL
MATCH-REPLY CONTROL
OVB
PLT
PLB
MR

Fig. 3.14 The BOAP Control System

The Machine Instruction Decoder:

At the fetching phase, the <AMI1> and <AMI234> parts of AMI are loaded from the Instruction Memory Buffer into the Instruction Register of the Control System. They are then separated by the Machine Instruction Decoder into a four beat sequence and addresses of <AMI234> or address of next instruction.

The Data Transfer Register:

In BOAP, every request for data transfer among various buffers (Scratch Pad, Input and Output Buffers) has to go through the Data Transfer Register of the Control System before reaching their destination. In the case of Scratch Pad Buffer, the address of Scratch Pad Buffer is kept in the Scratch Pad Address Register (SPAR) for the SPA in Scratch Pad Buffer (Fig.3.13).

The Feed Back Control Network:

The feed back signals from the associative memory modules: Namely OVT/OVB and MR, are first of all feed into the OVERFLOW CONTROL and MATCH-REPLY CONTROL respectively, for processing before driving the PLT/PLB CONTROL to set the PLT/PLB of various memory modules.

3.4 SUMMARY

For almost the last fourty years, the principles of computer design have largely remained static, based on the model of Von Neumann computer. However, as computing moves from a sequential world into a multiprocessing environment, distributed processing has become a necessity to bring together a large number of computing elements providing either a general-purpose or a special-purpose function. They may be broadly classified as control-flow, data-flow and reduction architecture in terms of their computation organization, program organization, and machine organization.

The Distributed Computer System with its dual processor configuration is based on the control-flow architecture, which includes the host processor--a conventional Von Neumann machine, and an associative processor which operates as a content-driven SIMD architecture. Apart from this distinct feature, the only exception is the physical separation of data storage (arguments) from the control storage (program): arguments in associative memory and program in RAM respectively. The design of DCS is based on the middle-out strategy by first designing the conventional machine level of the computer system, which including its computation organization, program organization and machine organization, before using a bottom-up approach to design the assembly language and its machine instructions in the successive chapters (Chapter Four & Five). In Chapter Four, the formal definition of the Associative Assembly Language (AAL) will be presented as the means for programming the DCS, which followed by the design of its machine instructions to drive the hardware of the DCS.

CHAPTER FOUR

THE DESIGN OF THE ASSOCIATIVE ASSEMBLY LANGUAGE

Assembly languages differ in a significant respect from the conventional problem-oriented languages in that there is a one-to-one mapping between machine instructions and statements in the assembly program. In other words, assembly language is just a mirror image of its machine code instruction in symbolic form, which it is therefore machine dependent, and has access to all the features and instructions available on the target machine (host machine). However, assembly languages for different machines have sufficient resemblance to one another to allow a discussion of assembly language in general. Assembly language instructions usually have four fields:

1) Label Field :

Labels, which are used to provide symbolic names for memory addresses, are needed on the executable instructions so that the location of the instructions can be referenced.

2) Operation Field :

The operation field contains either a symbolic abbreviation for the opcode or a pseudoinstruction (which is a command to the assembler). This is usually the most distinguishable field by which the flavour of the machine is reflexed. However, the choice of an abbreviation is often a matter of taste for individual assembly language designers.

3) Operand Field :

The operand fields are used to specify the addresses or registers whereby operands can be found.

4) Comments :

The comment field provides a place for the programmers to put helpful explanations of how the program works for the benefit of other programmers as well as the author himself.

The Associative Assembly Language (AAL) which is the superset of APIs and a SISD assembly language, follows the same general pattern of other assembly languages in the design of instruction format.

```
      LOOP1 S         ('?' X1XX)BMR MR=0,MR=1 ; SEARCH FOR '?'
      MR=1  M(CLBTT)                          ; CLEAR BITS TRUE TAGS
            RSTTD(S)                          ; RESOLVE TRUE TAGS DOWN
            W         (IQF 1XXX)              ; WRITE TO ALL TAGGED
                                              ; WORD-ROWS WITH THE
                                              ; OPERAND FORM THE INPUT
                                              ; BUFFER QUEUE FRONT
            DEC       A                       ; DECREMEMT REGISTER A
            JR   NZ,             LOOP1        ; GO BACK TO LOOP1 IF > 0
      MR=0  other <API234>                    ; PROCEEDED WITH THIS
                                              ; EXECUTE PHASE IF MR = 0
  |<-LABEL->|<OPCODE>|<-OPERAND->|<-NEXT INS.->|<------ COMMENT ------>|
```

Governed by the control-flow program organization, AAL uses the automatic sequencing of a program counter for the selection of next instruction. This mechanism allows instructions stored in consecutive memory locations to be fetched, examined, and executed one after the other. The program counter can also be explicitly altered by a branch instruction in order to accomplish the flow of control to be transfered to a specified location other than the next one in the sequence. Hence, the address field for next instruction is not needed in most of the AAL instructions. However, at the computation organization level, the API uses a rather different control structure to maintain the flow of control: based on the content-driven architecture, the API needs to split its instruction logically into two separated statements in order to comply with the rule of content-driven organization: notably <API1> and <API234>.

```
                      |<-    Beat 1   ->|<- Beat 2, Beat 3, Beat 4 ->|
                      ------------------------------------------------
                      |                  -> <API 234> --             |
 Associative          |            MR=0|    Statement  |             |
 Computation      -->|--- <API 1> ---->                ------------|-->
    Cycle             |   Statement MR=1|   Statement  |             |
                      |                  -> <API 234> --             |
                      ------------------------------------------------
```

The flow of control from <API1> to whichever <API234> is governed by the outcome of SEARCH operation in <API1>, therefore, the addresses of the two alternative <API234> are explicitly included in the instruction format of the <API1> statement.

```
LOOP1 S('?' X1XX)BMR    MR=0,MR=1   ; <API1> PART OF API
 MR=1 M(CLBTT) RSTTD(S) W(IQF 1XXX) ; <API234> PART FOR MR = 1

 MR=0 an alternative <API234>       ; <API234> PART FOR MR = 0
```

Moveover, in order to improve the readability of APIs, and be consistent with the general pattern of other assembly languages, the <API234> part of API is further split into three separated lines.

```
   LOOP1 S('?' X1XX)BMR    MR=0,+1         ; SEARCH FOR '?'
         M(CLBTT)                          ; CLEAR BITS TRUE TAGS
         RSTTD(S)                          ; RESOLVE TRUE TAGS DOWN
         W(IQF 1XXX)                       ; WRITE TO ALL TAGGED
                                           ; WORD-ROWS WITH THE
                                           ; OPERAND FORM THE INPUT
                                           ; BUFFER QUEUE FRONT
         DEC               A               ; DECREMEMT REGISTER A
         JR                NZ,LOOP1        ; GO BACK TO LOOP1 IF > 0
    MR=0 an alternative <API234>           ; PROCEEDED WITH THIS
                                           ; EXECUTE PHASE IF MR = 0
|<-LABEL->|<OPCODE>|<-OPERAND->|<-NEXT INS.->|<------ COMMENT ------>|
```

Nevertheless, at the program organization level, API as a whole (the Associative Computation Cycle) is being treated a single instruction similar to all other SISD instructions, in which the flow of control is sequenced by the program counter.

```
<ASSOCIATIVE COMPUTATION CYCLE> ::= <API1 STATEMENT><API234 STATEMENT>
```

```
Associative     ---------------       -----------------
Computation -->| API1 satement |---->| API234 statement |-->
   Cycle        ---------------       -----------------
```

4.1 THE EXAMINE PHASE OF API <API1>

The examine phase of API is always the SEARCH instruction which used to locate the potential candidates within the AMA for subsequent READ/WRITE operations. The functions of <API1> are shown as follows:

1) Reset TR1 before the SEARCH operation.

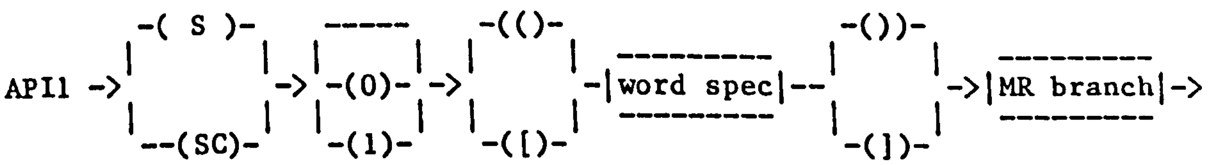
2) SEARCH<complement><DI><word spec>
Where the AMA is searched for the domain of word-rows which match the effective data of IDR, as interpreted by <complement> and <DI>.

3) For all matching word-rows set their tags in TR1.

4) Set MRR to logical '1' if one or more tags set.

5) Load the Associative Program Counter (APC) with the addresses of next two alternative parts of API (<API234>) into the BOAP Instruction Memory Buffer ready to be loaded into the BOAP Control System pending on the outcome of MRR.

```
<API1 STATEMENT> ::= <LABEL><TAB><API1><TAB>;<COMMEMT><CR>

<API1> ::= S<BSU>(<WORD SPEC>)<MR BRANCH>|
           S<BSU>[<WORD SPEC>]<MR BRANCH>
```



4.1.1 The <WORD SPEC>

In <WORD SPEC>, three different kinds of addressing schemes are used: namely Immediate Addressing, Scratchpad Addressing, and Buffer Addressing.

```
                     ---------------
            -------| immediate data |---------
           |         ---------------          |
           |       -------------------        |
word spec -->------| scratchpad address |----------->
           |       -------------------        |
           |      ---------------------------- |
            --| input buffer queue address |--
              ----------------------------
```

1) Immediate Addressing Scheme

In the Immediate Addressing Scheme, the actual data to be used for searching is embedded in the <WORD SPEC>. The BOAP supports two types of data organizations.

A) Text Symbols Mode :

Each word-row of the Associative Memory Array (AMA) is allocated to a single text symbol comprising an 8-bit character field and a 4-bit control field.

```
      7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
    -------------------------------------------------
    |   |   |   |   |   |   |   |   |   |   |   |   |
    -------------------------------------------------
    |<------Character Field-------->|<-C.B. Field ->|
    |<---- 8-bit Character Code --->|<-C.B. Field ->|
  ->|MSB|<---- 7-bit ASCII Code --->|<-C.B. Field ->|

    |<------------ 12-bit Bit Vector ------------->|
```

The choice of codes within the character field can be either one of the following:

I) 8-bit Character Code

```
S('T' X1XX)BMR     @LABEL0,@LABEL1 ; SEARCH FOR 'T'
                                   ; WITH C.B.= X1XX
```

Where T is the 8-bit character code.

II) 7-bit ASCII Code

Only Bit-0 to Bit-6 of the character field are used for the ASCII code, and remaining 7th bit (MSB) is be used as a extra Control Bit.

S(M/'k' X1XX)BMR   @LABEL0,@LABEL1

where M is the Most Significant Bit

K is any ASCII character.

B) Bit-Vector Mode :

In contrast with Text Symbols Mode, the Bit-Vector Mode organizes <WORD SPEC> into a 12-bit vector.

S[BBBBBBBBBBBB]BMR   @LABEL0,@LABEL1

Where <B> ::= X|0|1, a pair of square brackets is used here to distinguish Bit-Vectors from the Text Symbols which use instead a pair of round brackets.

```
                         ----------------------       -------
              ----(')--| 8-bit character code |--(')-|CB spec|-
             |           ----------------------     | -------   |
             |  ->(X)-      :                        ----       |
immediate    | |       |    :       ----------------      |     |
  -->-------->(0)- -(/)--(')-|7-bit ASCII code|-(')->          |->
   data      | |       |    ,     ----------------             |
             |  ->(1)-                                          |
             |                  ----------                      |
              --------------->|bit vector|----------------------
                                ----------
```

```
              ->(X)--    ->(X)--    ->(X)--    ->(X)--
             |       |  |       |  |       |  |       |
CB spec ------->(0)------->(0)------->(0)------->(0)------>
             |       |  |       |  |       |  |       |
              ->(1)--    ->(1)--    ->(1)--    ->(1)--
```

```
                                ->(X)--
                               |       |
bit vector ------------------->(0)----------------------->
                  |            |       |         |  = 12
                  |             ->(1)--          |
                  |                              |
                   <-----------------------------
```

2) Scratch-Pad Addressing Scheme

Scratch-Pad is the working area between IDR and ODR for storing intermediate data. The Scratch pad address, when in use, must start with @ to distinguish API Scratch-Pad address from SISD RAM address, and is written in one of the following forms:

A) Direct Addressing Mode

a) Numbers (between 0 to 1023)

```
S(@164 X1XX)BMR    @LABEL0,@LABEL1 ; SEARCH WITH THE
                                   ; OPERAND AT LOC.
                                   ; 164 IN S.P. AND
                                   ; SET C.B.= X1XX
```

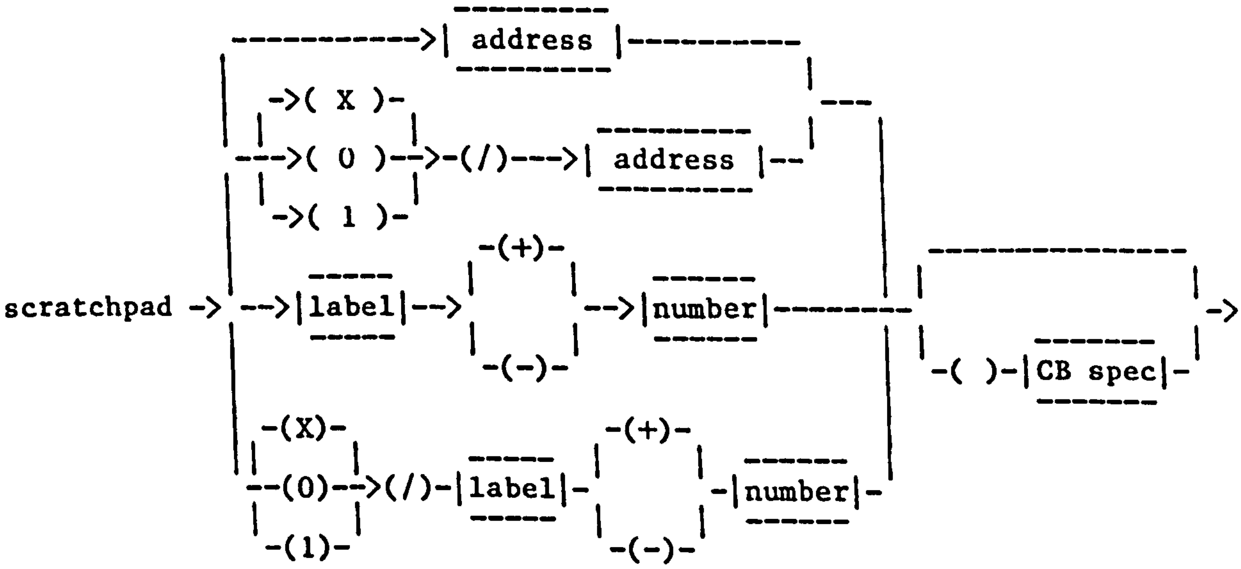
b) Symbols

```
S(@ADDR X1XX)BMR   @LABEL0,@LABEL1
S(@ADDRESS)BMR     @LABEL0,@LABEL1
```

The Control Bits setting can either be taken directly from the Scratch-Pad or set in the <API1> statement.

B) Relative Addressing : Expressions

```
S(@ADDR+1 X1XX)BMR   @LABEL0,@LABEL1 ; SEARCH WITH THE
                                     ; OPERAND AT ONE
                                     ; AFTER @ADDR IN SP
```



```
                              -------
                    ------>| label |------
                    |         -------        |
address ---------->                          |----->
                    |         --------       |
                    --( @ )-->| number |--
                              --------
```

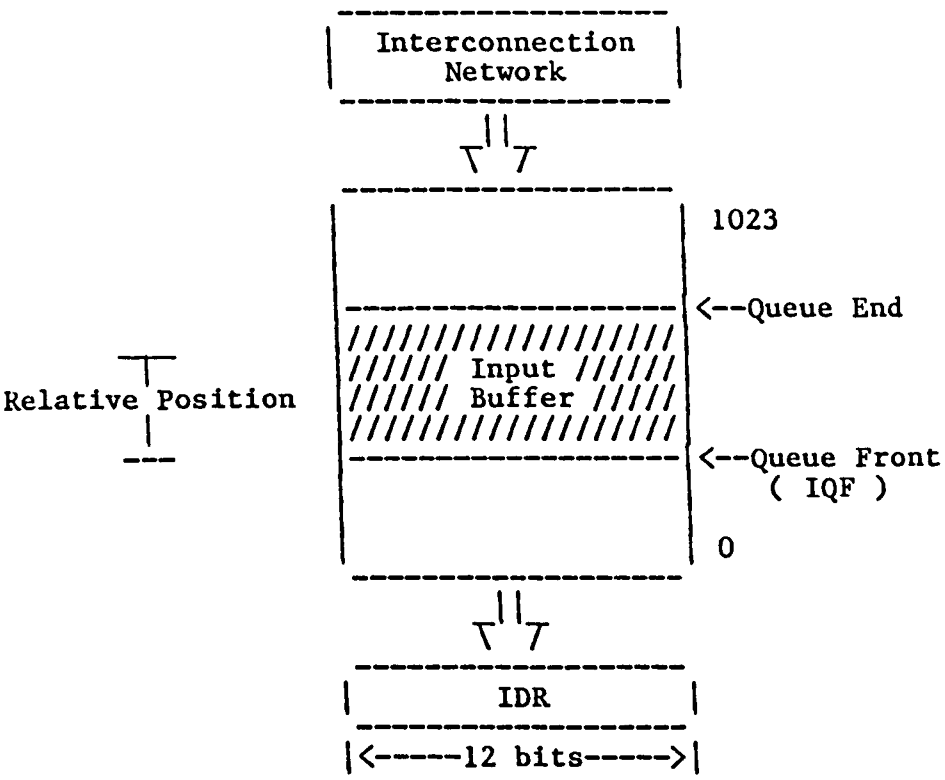
```
                            -------
number ------------->| digit |-------------->
                 |          -------          |
                 |   0 <= digit <= 1023      |  < 5
                 |                           |
                 <----------------------------
```

```
                                -------
                  -----------| empty |-----------
                  |             -------             |
label ----->                                        |----->
                  |            --------             |
                  --( @ )------| Ch.Spec |---------
                             |     --------     | < 6
                             <-----------------
```

```
                       --------------------
               --->| 8-bit character code |---
               |       --------------------      |
Ch.spec --->                                     |--->
               |       -----------------         |
               ----->| 7-bit ASCII code |-----
                       -----------------
```

3) Buffer Adressing Scheme

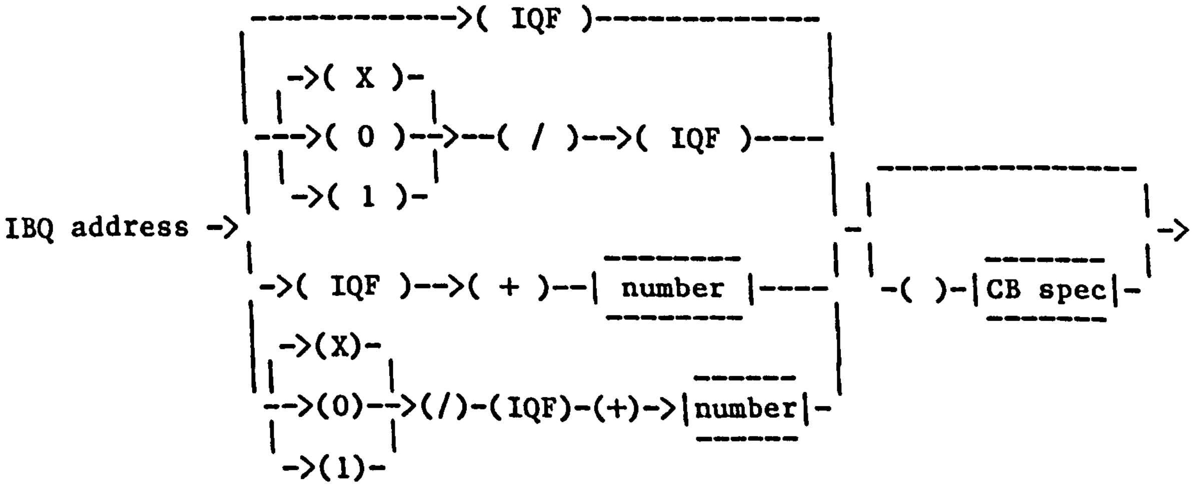
In <API1>, only Input Buffer addressing is used for buffering input data from the Host processor via an Interconnection Network. The Input Buffer functions as a word-organized FIFO (First In First Out) queue: incoming data is placed at the end of the queue, and outgoing data is taken from the beginning of the queue pointed at by the IQF (Input-Buffer Queue Front).



```
S(IQF 1XXX)BMR       @LABEL0,@LABEL1  ; SEARCH WITH OPERAND
                                      ; AT THE QUEUE FRONT OF
                                      ; INPUT BUFFER, AND SET
                                      ; CONTROL BITS = 1XXX

S(IQF)BMR            @LABEL0,@LABEL1  ; SEARCH WITH SAME
                                      ; OPERAND AND C.B.
                                      ; SETTING FROM IQF
```

Or alternatively, a relative addressing with reference from the queue front can be used.

```
S(IQF+3 XX1X)BMR    @LABEL0,@LABEL1
S(IQF+3)BMR         @LABEL0,@LABEL1
```



4.1.2 The <BSU>

The <BSU> defines the functions of the Bit Select Unit: namely, Data Masking and Data Complementing.

1) The data masking

The BOAP has no Mask Register. Hence, there can be no explicit data masking during SEARCH or WRITE operations. Instead, two modes of implicit data masking are provided.

A) Unconditional Data Masking

Each bit position in the DIR can be loaded with the tertiary datum <D>, where <D> ::= X|0|1, and X implies that the corresponding bit-column is to be masked during SEARCH or WRITE operations.

B) Conditional Data Masking

In addition to the Unconditional Data Masking, bit-columns of AMA can be masked according to the state of Data Identity <DI>.

Content of IDR	DI = 0	DI = 1	DI = X
0	0	Masked	0
1	Masked	0	1
X	X	X	X

With the inclusion of this data masking, one could select and mask on either '0' or '1' within IDR.

Mnemonic	Function
S	Search with the true content of IDR
S0	Search with <DI> = 0
S1	Search with <DI> = 1

2) The data complementing

Data complementing is used for the selection of effective data for SEARCH or WRITE operation by the true or complemented content of IDR.

Content of IDR	CMW = 0	CMW = 1
0	0	1
1	1	0
X	X	X

Mnemonic	Function
SC	Search with the complemented content of IDR

To sum up the data organization of BOAP; we could view the Data Masking as a filter, and Data Complementing as an invertor of some kind. With the combination of both, a comprehensive variety of data transformations can be achieved.

	Effective data for SEARCH/WRITE					
	CMW = 0			CMW = 1		
Content of IDR	DI=0	DI=1	DI=X	DI=0	DI=1	DI=X
0	0	X	0	1	X	1
1	X	1	1	X	0	0
X	X	X	X	X	X	X

Mnemonic	Function
SC0	Search with complemented content of IDR, subject to <DI> = 0
SC1	Search with complemented content of IDR, subject to <DI> = 1

	7	6	5	4	3	2	1	0	CB1	CB2	CB3	CB4
I D R	0	0	0	0	1	1	1	1	X	X	X	X
MASK (DI)	0	0	1	1	0	0	1	1	0	0	1	1
COMPLEMENT	0	1	0	1	0	1	0	1	0	1	0	1
	v	v	v	v	v	v	v	v	v	v	v	v
EFFECTIVE DATA	0	1	X	X	X	X	1	0	X	X	X	X

In Bit-Vector Mode, the Data Masking and Data Complementing are applied to all 12 bits, whereas only the Control Bit Field is effected in the case of Text Mode.

4.1.3 The <MR branch>

With the two possible alternative <API234>, the <API1> adopts a 2-address instruction format.

```
     1    S('T' X1XX)BMR    @LABEL0,@LABEL1  ; SEARCH FOR 'T'
     2    S(IQF X1XX)BMR    @16,@17          ; SEARCH IN AMA WITH
                                             ; THE <WORD SPEC>
                                             ; CURRENTLY STORED
                                             ; AT THE TOP OF THE
                                             ; INPUT BUFFER. THEN
                                             ; EXECUTE <API234> AT
                                             ; ADDRESS 16 IF MR = 0,
                                             ; ELSE <API234> AT
                                             ; ADDRESS 17 IF MR = 1
     3    S(IQF X1XX)BMR    +1,+2
 |<Label>|<Opcode, Operand>|<- Next Inst. ->|<--- Comments --->|
                              Addresses
```

Syntactically, <MR branch> may be either partially or totally omitted.

A) <MR branch> totally omitted:

```
     S('T' X1XX)                   ; DEFAULT IS +1,+2
```

An empty label will be interpreted by the assembler as branch to the next address location immediately after the current one for MR = 0, and the second immediately adddress location for MR = 1.

```
          empty implies    BMR    APC+1,APC+2
```

Where APC is the Associative Program Counter.

B) <MR branch> partially omitted:

```
    S('T' X1XX)BMR  @LABEL0              ; DEFAULT IS @LABEL0,+1
    S('T' X1XX)BMR           ,@LABEL1    ; DEFAULT IS +1,@LABEL1
```

Besides the unique feature of this addressing format, <API1> uses Relative Addressing to select two of its <API234> statements which must be either symbolic labels or displacement numbers, and are written in one of the following forms:

A) Displacement Numbers (between -128 to +127)

```
        S(@164 X1XX)BMR      +10,+12    ; RELATIVE BRANCH TO 10
                                        ; INST. AHEAD IF MR = 0,
                                        ; OR BRANCH TO 12 INST.
                                        ; AHEAD IF MR = 1
```

B) Symbolic Labels

```
        S(@ADDR X1XX)BMR  @LABEL0,@LABEL1 ; BRANCH TO @LABEL0
                                          ; IF MR = 0, OR
                                          ; BRANCH TO @LABEL1
                                          ; IF MR = 1
```

The Symbolic Label must start with @ to distinguish API statement addresses from SISD instruction addresses.

```
                           -------
             ------------>| empty |-----------
            |              -------            |
MR branch -->                                  -->
            |          ------      ------     |
             ->(BMR)->|label0|->(,)->|label1|-
                       ------      ------


                        -(+)-
           -------     |     |   ------------------
label0 --> ->| label |-       -| relative address |->
           -------     |     |   ------------------
                        -(-)-


                        --------
label1 -------->| label0 |-------->
                        --------


                                   --------
relative address -->----------| offset |---------------->
                     |         --------          | < 4
                     | -128 <= offset <= +127 |
                     <-----------------------
```

4.2 THE EXECUTE PHASE OF API <API234>

The <API234> is the combination of API2, API3 and API4, which has two variants of these sequences to allow for domain modification before or after function executions.

1) <u>The Pre-Function Associative Computation Cycle</u>

A) Non Group-Run Associative Computation Cycle

Beat 0 : Fetch phase (LOAD instruction)

Beat 1 : Examine phase (SEARCH operation)

Beat 2 : Execute phase 1--Domain modification (Clear option)

Beat 3 : Execute phase 2--Domain modification (Tag manipulation)

Beat 4 : Execute phase 3--Function execution (READ/WRITE oper.)

B) Group-Run Associative Computation Cycle

Beat 0 : Fetch phase (LOAD instruction)

Beat 1 : Examine phase (SEARCH for TR1)

Beat 2 : Execute phase 1--Domain modification (SEARCH for TR2)

Beat 3 : Execute phase 2--Domain modification (Group-Run)

Beat 4 : Execute phase 3--Function execution (Restricted)
(READ/WRITE)

2) <u>The Post-function Associative Processing Cycle</u>

Beat 0 : Fetch phase (LOAD instruction)

Beat 1 : Examine phase (SEARCH operation)

Beat 2 : Execute phase 1--Domain modification (Clear option)
Function execution 1 (READ/WRITE)

Beat 3 : Execute phase 2--Domain modification (Tag manipulation)

Beat 4 : Execute phase 3--Function exection 2 (Update operation)

<API234 STATEMENT> ::=<LABEL><TAB><API234>;<COMMENT><CR>

```
                 -------------------------
              ->| pre-function modification |--
             |   -------------------------     |   ------
API 234 -->                                     ->|branch|-->
             |   --------------------------    |   ------
              ->| post-function modification |-
                 --------------------------
```

4.2.1 The Pre-Function Non Group-Run <API234>

In the Pre-Function Non-Group-Run <API234>, it consists of

CLEAR OPTION (<API2>)
TAG MANIPULATION (<API3>)
READ/WRITE FUNCTION (<API4>)

```
                                ------   ------   ------
pre-function modification -->|mod. 2|->|mod. 3|->|fun. 4|-->
                                ------   ------   ------
```

API2 : Clear Options

The CLEAR options are designed to clear the bit-columns of tagged word-rows which are related to the content of IDR used during beat 1 SEARCH operation.

```
<MOD. 2> ::= <EMPTY>|
             M(<CL>)<TAB>;<COMMENT><CR>|
             M<DI>(<CL>)<TAB>;<COMMENT><CR>

    <CL> ::= CLAB|
             CLBTT|
             CLBCT
```

In these CLEAR operations, bit-columns are selected by the <DI2> and use the content of IDR left over from beat 1 SEARCH operation. The following table shows how the BSU enables the selected bit-columns for the subsequent CLEAR operation.

	Logical content of IDR	
<DI2>	0	1
0	e	-
1	-	e
X	e	e

e = enable

Whereas word-rows are activated for the specified CLEAR operations in according to the option of <CL> chosen and the logical content of TR1, as shown in the following table.

		Logical content of TR1	
Specifications	<CL>	0	1
No Clear	-	-	-
Clear bits on true tags	CLBTT	-	a
Clear bits on complemented tags	CLBCT	a	-
Clear all bit	CLAB	a	a

a = activation

1) Text Symbols

In the case of Text Symbols, only Control-Bits will be cleared: The Control-Bits are used as markers to mark positions within a record or field. Propagation of these markers will then be used to chain up a number of fields or records, but physically, markers can not be propagated from one word-row to the others, instead the markers of present stage are cleared before the new markers of adjacent word-rows can be created.

```
S('T' X1XX)BMR +1,+2
M(CLBTT)                      ; CLEAR BITS TRUE TAGS
PTT(U)
W(* 0XXX)BRN   @NEXT
```

Since CB2 alone was used during beat 1 SEARCH, only the CB2 column will be enabled for CLEAR operation.

```
              Ch. spec.     CB spec.
         -------------------------------
    IDR  |          T        |X|1|X|X|
         -------------------------------
                              | | | |
         -------------------------------
    BSU  | | | | | | | | | | |e|e|e|e|
         -------------------------------
                              | | | |
                              v v v v      TR1    WSU
         -------------------------------   ---    ---
         |                   | |0| | |   | 1 |  | a |
         |                   |C| |C|C|   | 0 |  |   |
         |                   | |0| | |   | 1 |  | a |
         |        A M A      |B| |B|B|   | 0 |  |   |
         |                   | | | | |   | 0 |  |   |
         |                   |1| |3|4|   | 0 |  |   |
         |                   | |0| | |   | 1 |  | a |
         -------------------------------   ---    ---
```

2) Bit-Vectors

For Bit-Vectors, the CLEAR operations will affect every bit which has been selected in IDR during beat 1 SEARCH operation.

```
S[XXXX1100 XXXX]BMR   +1,+2
M(CLBCT)                          ; CLEAR BITS COMPLEMENT TAGS
PTT(U)
W[XXXX1111 1XXX]BRN   @NEXT
```

```
              Ch. spec.        CB spec.
         ---------------------------------
    IDR  |X|X|X|X|1|1|0|0|  |X|X|X|X|
         ---------------------------------
          | | | | | | | |    | | | |
         ---------------------------------
    BSU  |e|e|e|e|e|e|e|e|  |e|e|e|e|
         ---------------------------------
          | | | | | | | |    | | | |
          v v v v v v v v    v v v v     TR1    WSU
         ---------------------------------   ---    ---
         |0|1|0|1|0|1|0|0  | | | | |    | 1 |  |   |
         |1|1|1|1|0|0|1|0  | | | | |    | 0 |  | a |
         |1|0|1|0|0|0|1|1  | | | | |    | 0 |  | a |
    AMA  |0|0|0|0|0|0|0|1  | | | | |    | 0 |  | a |
         |0|0|1|1|0|0|0|0  | | | | |    | 0 |  | a |
         |0|1|0|1|0|0|1|1  | | | | |    | 0 |  | a |
         |1|1|1|1|1|0|0|0  | | | | |    | 1 |  |   |
         ---------------------------------   ---    ---
```

```
                                -------
m    ---------------------->| empty |----------------------
o   |                           -------                     |
d-->                -----     -(CLBTT)-                      -->
    |              |     |   |         |    -------   ----  |
2    ->(M)->  -(0)--(()---(CLBCT)-->();)->|comment|->| cr |--
               |     |   |         |    -------   ----
                -(1)-     -(CLAB )-
```

API3 : Tag Manipulations

The tag manipulations of this <API234> provide programmer-control over the mapping between the tags in the TR1 and the word-rows which will then be activated for function executions.

```
<MOD. 3> ::= <TAB><PROPAGATE OPTIONS><TAB>;<COMMENT><CR>|
             <TAB><RUN OPTIONS><TAGS>;<COMMENT><CR>

<PROPAGATE OPTIONS> ::= <PROPAGATE TAGS>(<DIRECTION>)

<PROPAGATE TAGS> ::= PTT|
                     PCT|
                     RSTTU|
                     RSTTD|
                     RSCTU|
                     RSCTD

<PROPAGATE TAGS> ::= D|S|SD|U|UD|US|USD

<RUN OPTIONS> ::= <RUN TAGS>(<DIRECTION>)

<RUN TAGS> ::= EIR|
               MOR
```

There are two types of tag manipulations available for Non-Group Run Pre-Function <API234>.

1) PROPAGATE TAGS ACTIVATION

A) Propagate Options :

The propagate options select the tagged word-rows with PTT (Propagate True Tags) or untagged word-rows with PCT (Propagate Complement Tags) for subsequent word-row activations. The propagation <Direction> allows the activations to be extended beyond the contents of the Tag Register (TR). The following table shows how word-rows could be activated for each <Direction> selected when a tag is set in word-row n of the TR1.

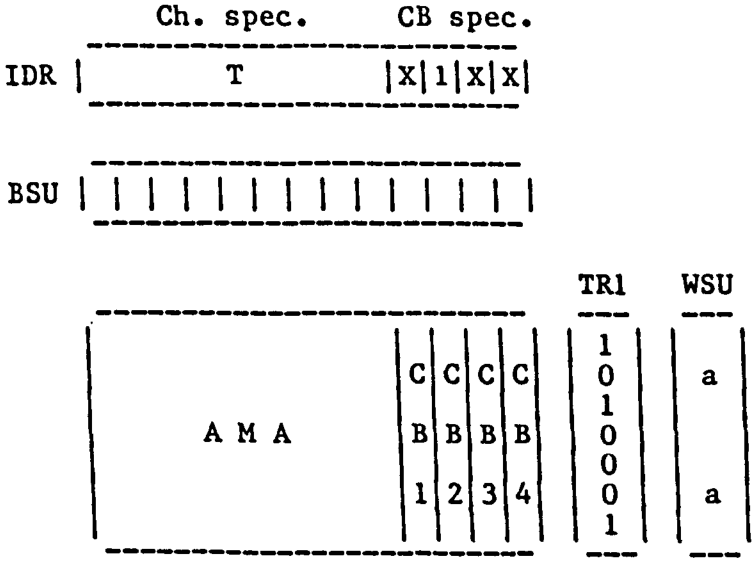
<Direction>			Activated word-row		
<U>	<S>	<D>	n-1	n	n+1
-	-	-	-	-	-
-	-	D	-	-	a
-	S	-	-	a	-
-	S	D	-	a	a
U	-	-	a	-	-
U	-	D	a	-	a
U	S	-	a	a	-
U	S	D	a	a	a

U = Up (B end --> T end)

D = Down (T end --> B end)

S = Straight a head (only TT or CT word-rows)

```
S('T' X1XX)BMR +1,+2
M(CLBTT)
PTT(U)                         ; PROPAGATE TRUE TAGS UP
W(* 0XXX)BRN   @NEXT
```



* overflow at T-end OVT = 1

```
S('T' X1XX)BMR +1,+2
M(CLBTT)
PCT(D)                    ; PROPAGATE COMPLEMENT TAGS DOWN
W(* 0XXX)BRN   @NEXT
```

```
            Ch. spec.      CB spec.
        -------------------------
   IDR |        T         |X|1|X|X|
        -------------------------

        -------------------------
   BSU | | | | | | | | | | | | | |
        -------------------------

                                    TR1    WSU
        -------------------------   ---    ---
       |                 | | | | | | 1 |  |   |
       |                 |C|C|C|C| | 0 |  |   |
       |                 | | | | | | 1 |  | a |
       |     A M A       |B|B|B|B| | 0 |  |   |
       |                 | | | | | | 0 |  | a |
       |                 |1|2|3|4| | 0 |  | a |
       |                 | | | | | | 1 |  | a |
        -------------------------   ---    ---
```

B) Resolve Options :

The <Resolve Tags> are used to isolate a particular word-row from the other selected word-rows: it inhibits all but the first (either from the T-end or B-end) activated word-row for function execution.

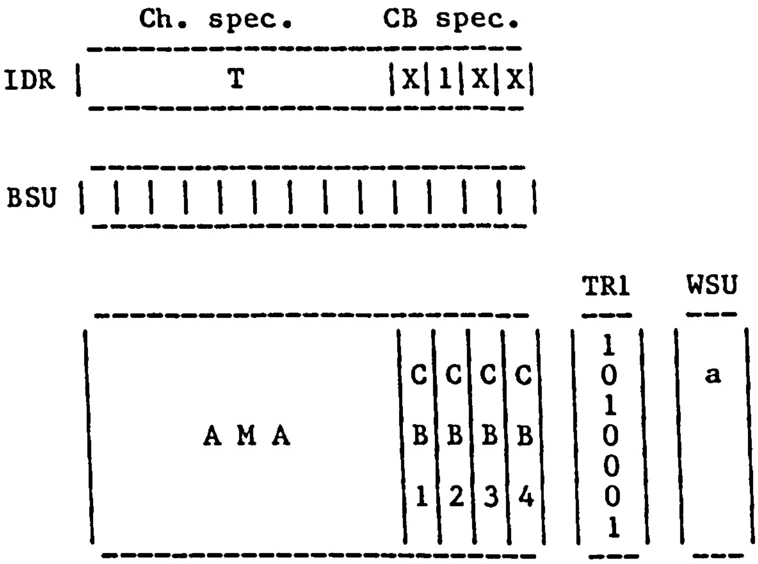
a) Resolve True Tags Up : RSTTU

```
S('T' X1XX)BMR +1,+2
M(CLBTT)
RSTTU(S)                  ; RESOLVE TRUE TAGS UP (S)
W(* 0XXX)BRN   @NEXT
```

```
            Ch. spec.      CB spec.
        -------------------------
   IDR |        T         |X|1|X|X|
        -------------------------

        -------------------------
   BSU | | | | | | | | | | | | | |
        -------------------------

                                    TR1    WSU
        -------------------------   ---    ---
       |                 | | | | | | 1 |  |   |
       |                 |C|C|C|C| | 0 |  |   |
       |                 | | | | | | 1 |  |   |
       |     A M A       |B|B|B|B| | 0 |  |   |
       |                 | | | | | | 0 |  |   |
       |                 |1|2|3|4| | 0 |  |   |
       |                 | | | | | | 1 |  | a |
        -------------------------   ---    ---
```

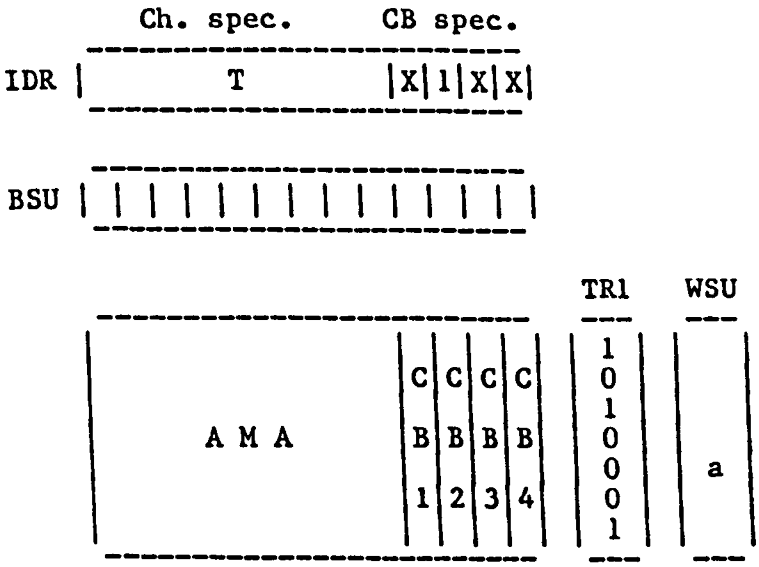
b) Resolve True Tags Down : RSTTD

```
S('T' X1XX)BMR +1,+2
M(CLBTT)
RSTTD(D)                    ; RESOLVE TRUE TAGS DOWN (D)
W(* 0XXX)BRN   @NEXT
```



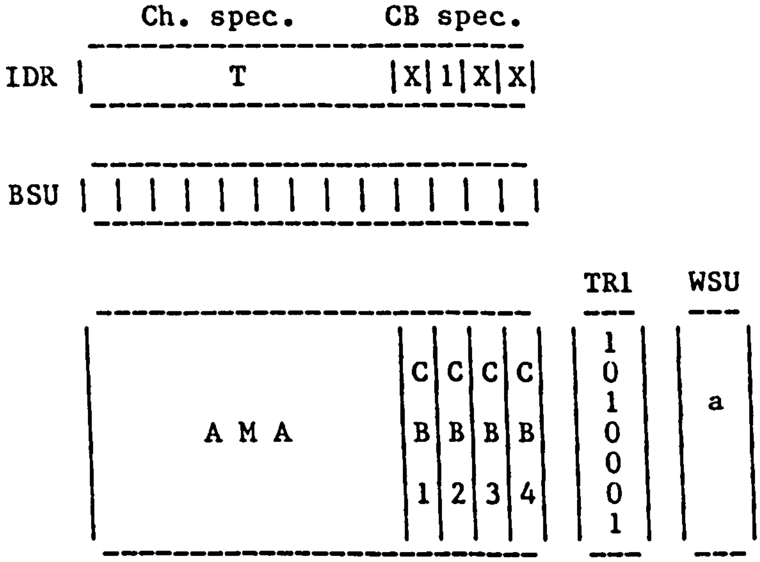
c) Resolve Complement Tags Up : RSCTU

```
S('T' X1XX)BMR +1,+2
M(CLBTT)
RSCTU(U)                    ; RESOLVE COMPLEMENT TAGS UP(U)
W(* 0XXX)BRN   @NEXT
```



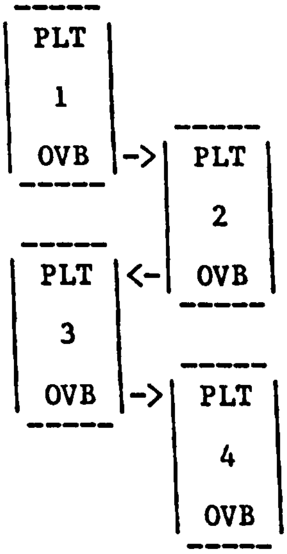
d) Resolve Complement Tags Down : RSCTD

```
S('T' X1XX)BMR +1,+2
M(CLBTT)
RSCTD(D)                   ; RESOLVE COMP. TAGS DOWN (D)
W(* 0XXX)BRN   @NEXT
```



Since the actitvation of adjacent word-rows are allowed, the Overflow Bits <OVT> and <OVB> would be set if the selected propagation mode propagates out of either T-end or B-end of a chip module. This could be used as a means to propagate activations over a number of chip modules, if more than a single chip were used.

PTT(D) : PLT1 = 0
PLT2 = OVB1
PLT3 = OVB2
PLT4 = OVB3

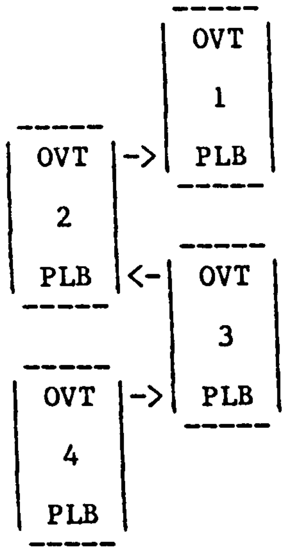


PTT(U) : PLB4 = 0

PLB3 = OVT4

PLB2 = OVT3

PLB1 = OVT2



2) Run Tags Options :

The run options allow each set tag of the TR1 to activate word-rows in an adjacent block of word-rows in the direction specified by <Direction>.

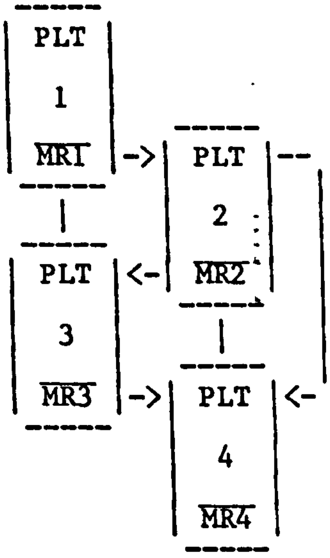
A) End In Run : EIR

An EIR activates all word-rows from either T-end or B-end to the first tagged word-row, as indicated below.

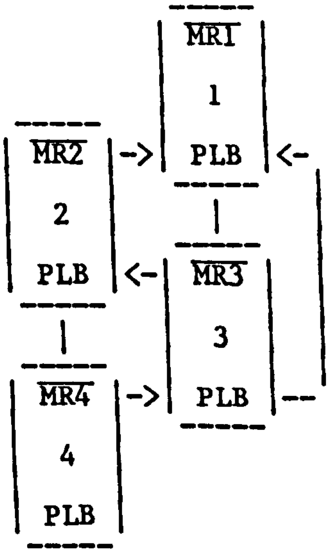
<Direction>			Logical Content of TR1																							
<U>	<S>	<D>	T	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	B
0	0	0																								
0	0	1		a	a	a	a	a																		
0	1	0						a						a							a					
0	1	1		a	a	a	a	a						a							a					
1	0	0																			a	a	a	a	a	
1	0	1		a	a	a	a	a													a	a	a	a	a	
1	1	0						a						a							a	a	a	a	a	
1	1	1		a	a	a	a	a						a							a	a	a	a	a	

The activation networks for up (B-end -> T-end) and down (T-end -> B-end) are implemented independently such that an up run and down run may proceed in parallel. Resolve operation is not necessary, as EIR usually only activates a block of word-rows at any one time: EIR(U) will activate only the first group up from the B-end, and EIR(D) will activate only the first group down from the T-end etc. An EIR is initialized by setting <PLT> = 1 for EIR(D) or <PLB> = 1 for EIR(U). It can proceed over a number of chip modules without significant loss of the execution speed by allowing the modules to execute the runs in parallel with their <PLT> or <PLB> set according to the <MR> outputs in Beat 1.

$$\begin{aligned} \mathrm{EIR(D)} :\ & \mathrm{PLT}\ 1 = 1 \\ & \mathrm{PLT}\ 2 = \overline{\mathrm{MR}\ 1} \\ & \mathrm{PLT}\ 3 = \overline{\mathrm{MR}\ 1} + \overline{\mathrm{MR}\ 2} \\ & \mathrm{PLT}\ 4 = \overline{\mathrm{MR}\ 1} + \overline{\mathrm{MR}\ 2} + \overline{\mathrm{MR}\ 3} \end{aligned}$$



$$\begin{aligned} \mathrm{EIR(U)} :\ & \mathrm{PLB}\ 4 = 1 \\ & \mathrm{PLB}\ 3 = \overline{\mathrm{MR}\ 4} \\ & \mathrm{PLB}\ 2 = \overline{\mathrm{MR}\ 4} + \overline{\mathrm{MR}\ 3} \\ & \mathrm{PLB}\ 1 = \overline{\mathrm{MR}\ 4} + \overline{\mathrm{MR}\ 3} + \overline{\mathrm{MR}\ 2} \end{aligned}$$



If no tag is set in the TR1, then <OVT> or/and <OVB> will be set when the selected run option causes a word-row to be activated beyond the T-end or B-end of the modules.

B) Middle Out Run : MOR

A MOR activates all word-rows from (but not including except when <S> is set) the first word-row which has been tagged in TR1 to (and beyond) the T-end or B-end.

<Direction>			Logical Content of TR1																							
<U>	<S>	<D>	T	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	B
0	0	0																								
0	0	1							a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	
0	1	0						a					a							a						
0	1	1						a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	
1	0	0		a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a							
1	0	1		a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	
1	1	0		a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a						
1	1	1		a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	

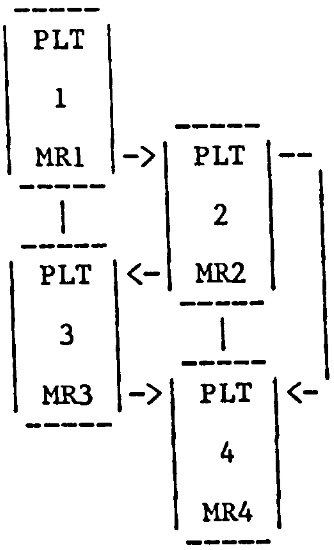
Similar to EIR, the up-run and down-run of MOR may proceed in parallel. Resolve is almost impossible as MOR is a continued run across the modules, except for the case of MOR(S). If resolving MOR(S) were proved to be necessary, then perhaps it might be better to use the Resolve Tags options. MOR can proceed over a number of chip modules, without significant loss of speed, by allowing the modules to execute the runs in parallel, with their <PLT> or <PLB> set according to the <MR> outputs in Beat 1.

MOR(D) : PLT 1 = 0

PLT 2 = MR 1

PLT 3 = MR 1 + MR 2

PLT 4 = MR 1 + MR 2 + MR 3

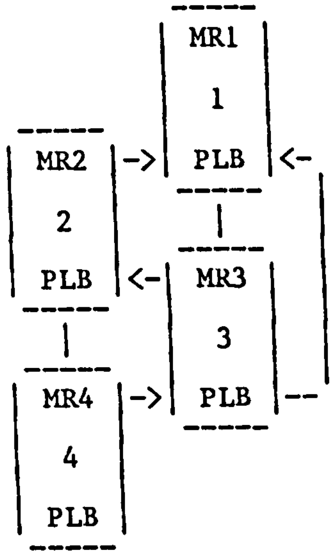


MOR(U) : PLB 4 = 0

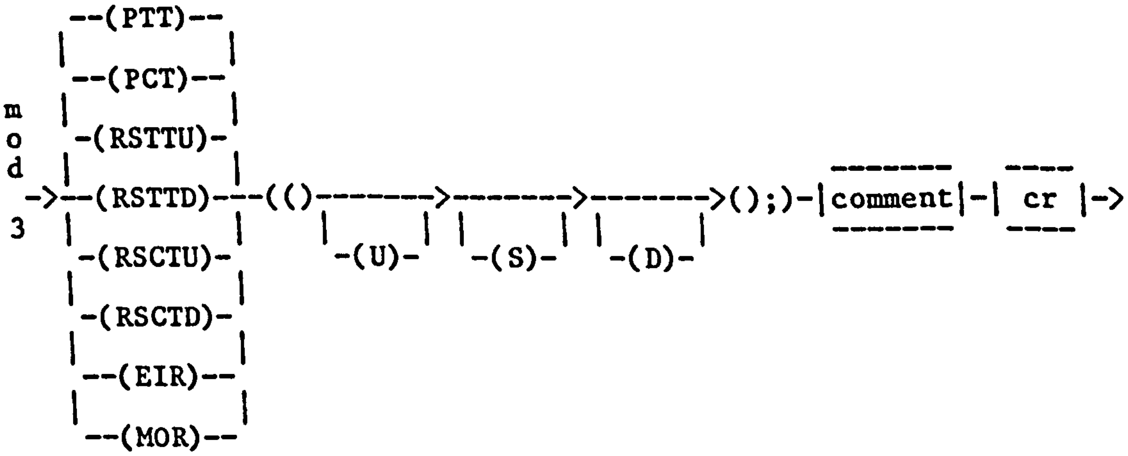
PLB 3 = MR 4

PLB 2 = MR 4 + MR 3

PLB 1 = MR 4 + MR 3 + MR 2



The <OVT> will be set for MOR(U), MOR(US), MOR(UD), MOR(USD), and <OVB> will be set for MOR(D), MOR(SD), MOR(UD), MOR(USD). As any MOR will tend to activate beyond T-end or/and B-end.



API4 : FUNCTION EXECUTION

Function execution is the ultimate goal of the Associative Computation Cycle which is either to read from or write to the tagged word-rows of the AMA.

```
                ------------     --------
function 4  -->| read/write |->| branch |-->
                ------------     --------
```

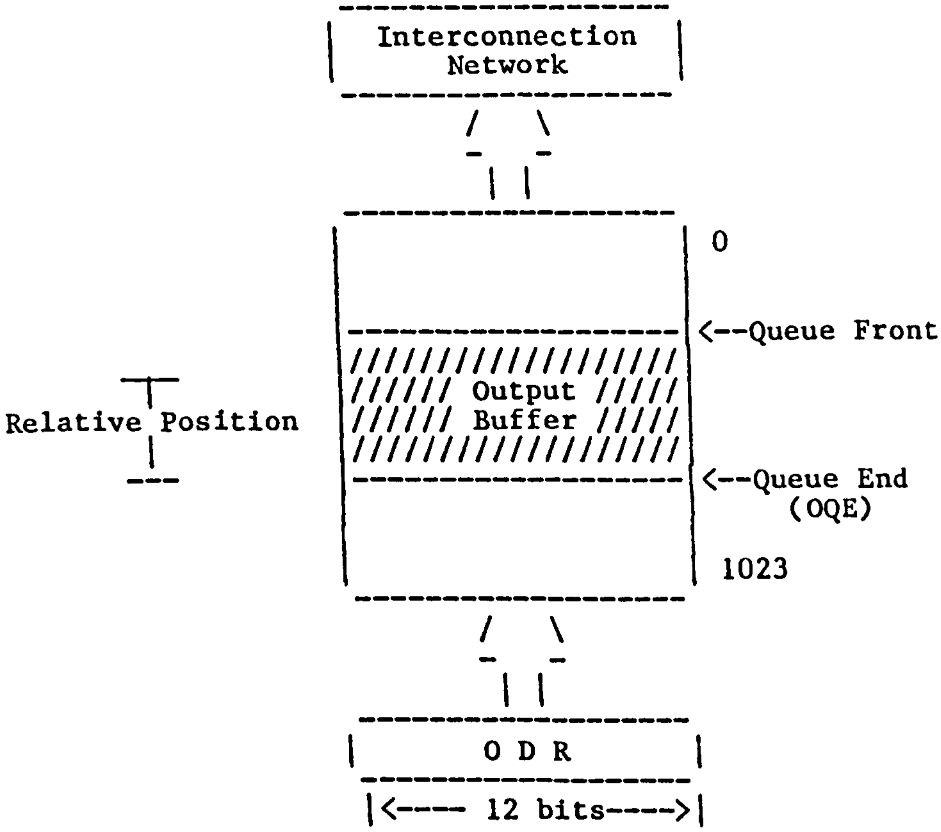
1) The Read/Write Operation :

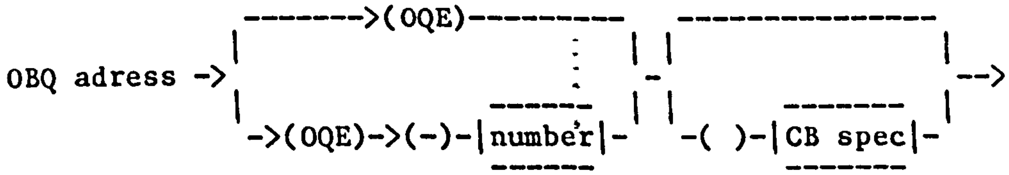
```
<READ/WRITE> ::= W<BSU>(<WORD SPEC>)|
                 W<BSU>[<WORD SPEC>]|
                 R<BSU>(<SCRATCHPAD ADDRESS>)|
                 R<BSU>(<OUTPUT BUFFER QUEUE ADDRESS>)
```

In API4, the <WORD SPEC> for WRITE function is actually the same as <WORD SPEC> in API1. However, the READ function has a slightly different <WORD SPEC>, namely Immediate Addressing, Scratchpad Addressing, and Buffer Addressing (Output Buffer Addressing instead of Input Buffer Addressing).

```
                             --------------------
                   ------| scratchpad address |-------
                  |          --------------------      |
READ word spec -->                                      -->
                  |   ----------------------------     |
                   --| output buffer queue address |--
                      ----------------------------
```

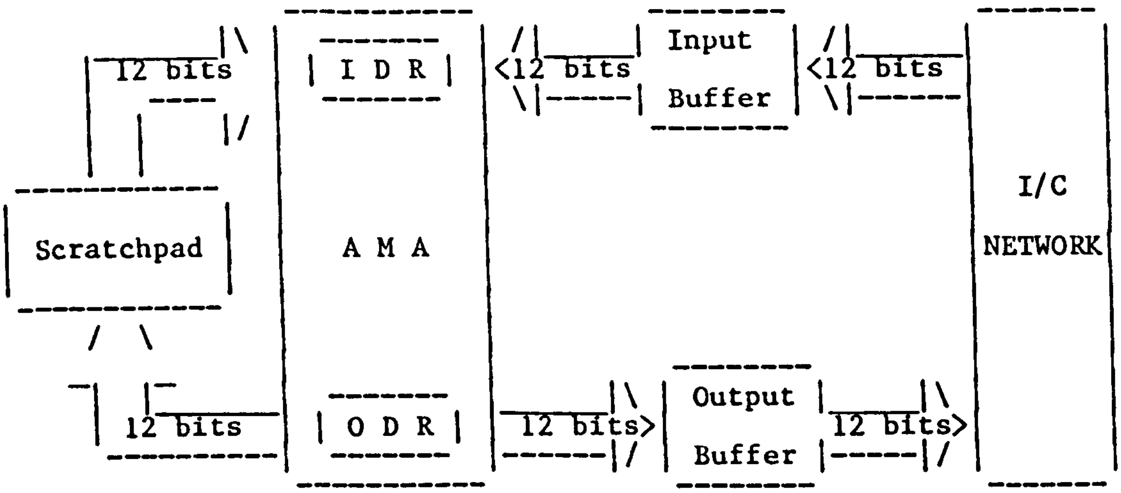
The structure of Output Buffer is very similar to the Input Buffer except that data is read in from ODR and output to the Interconnection Network, and in the case of relative addressing, it referred to the queue end (OQE) instead of queue front (IQF).





```
R(OQE 1XXX)       ; READ TAGGED WORD-ROW TO OQE WITH
                  ; C.B. = 1XXX

R(OQE)            ; READ TAGGED WORD-ROW TO OQE WITH
                  ; ON CHANGE IN C.B. FIELD

R(OQE-2 X1XX)     ; READ TAGGED WORD-ROW TO TWO LOC.
                  ; BEHIND OQE WITH C.B. = X1XX

R(OQE-2)          ; READ TAGGED WORD-ROW TO TWO LOC.
                  ; BEHIND OQE WITH NO CHANGE IN C.B.
```

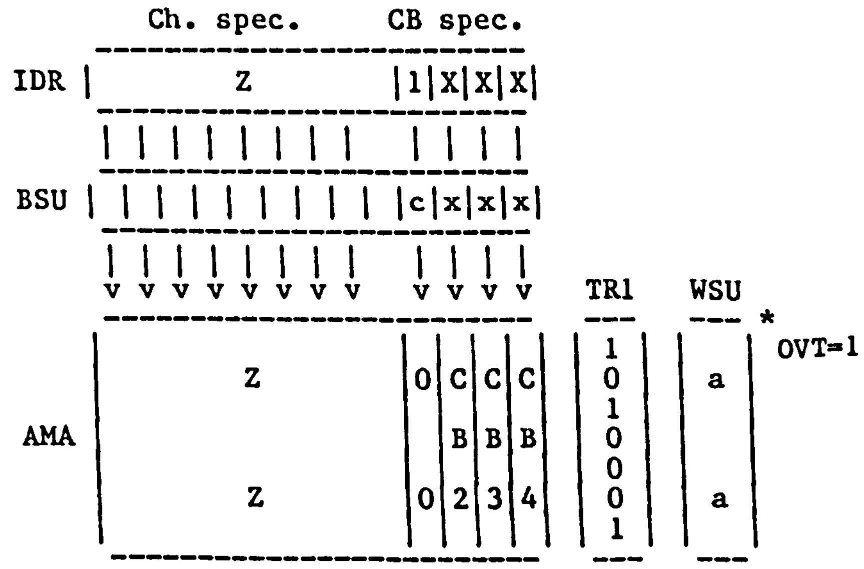
The flow of data within the BOAP could be viewed as shown in the diagram as follows:



A) The WRITE Operation :

The WRITE operation will update all activated word-rows with the effective content of the IDR as interpreted by the BSU subject to the <Complement> and <DI4>.

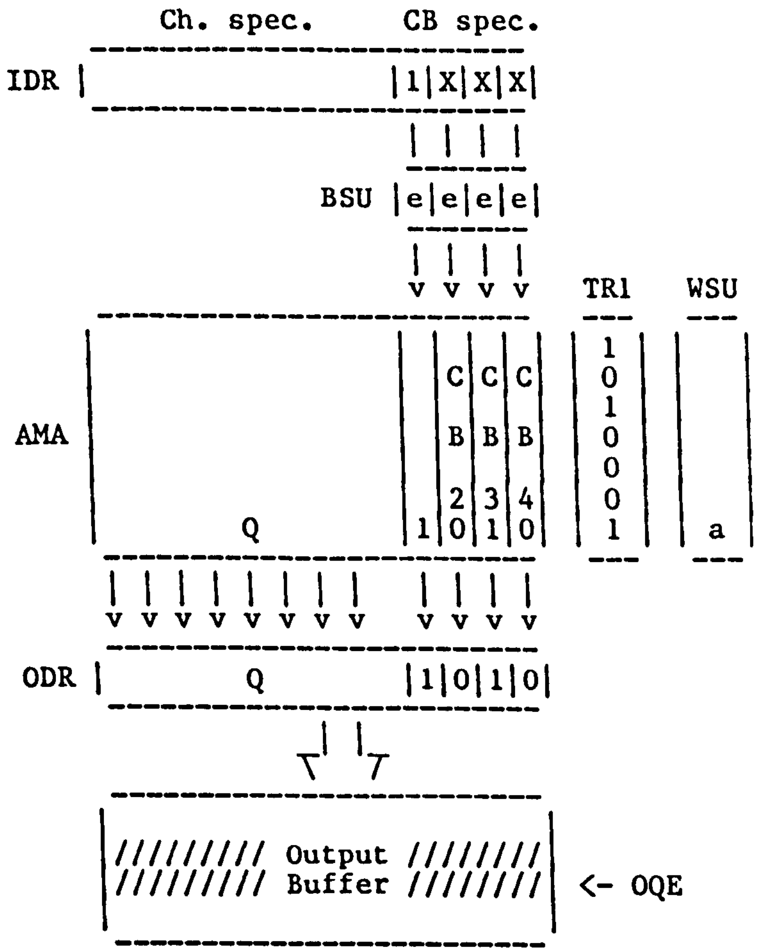
```
S('T' X1XX)BMR +1,+2
M(CLBTT)
PTT(U)
WC1('Z' 1XXX)BRN   @NEXT       ; WRITE TO ALL TAGGED WORD-
                               ; ROWS WITH 'Z' AND 1XXX
                               ; SUBJECTED TO COMP. MASKING
                               ; AND DI = 1
```

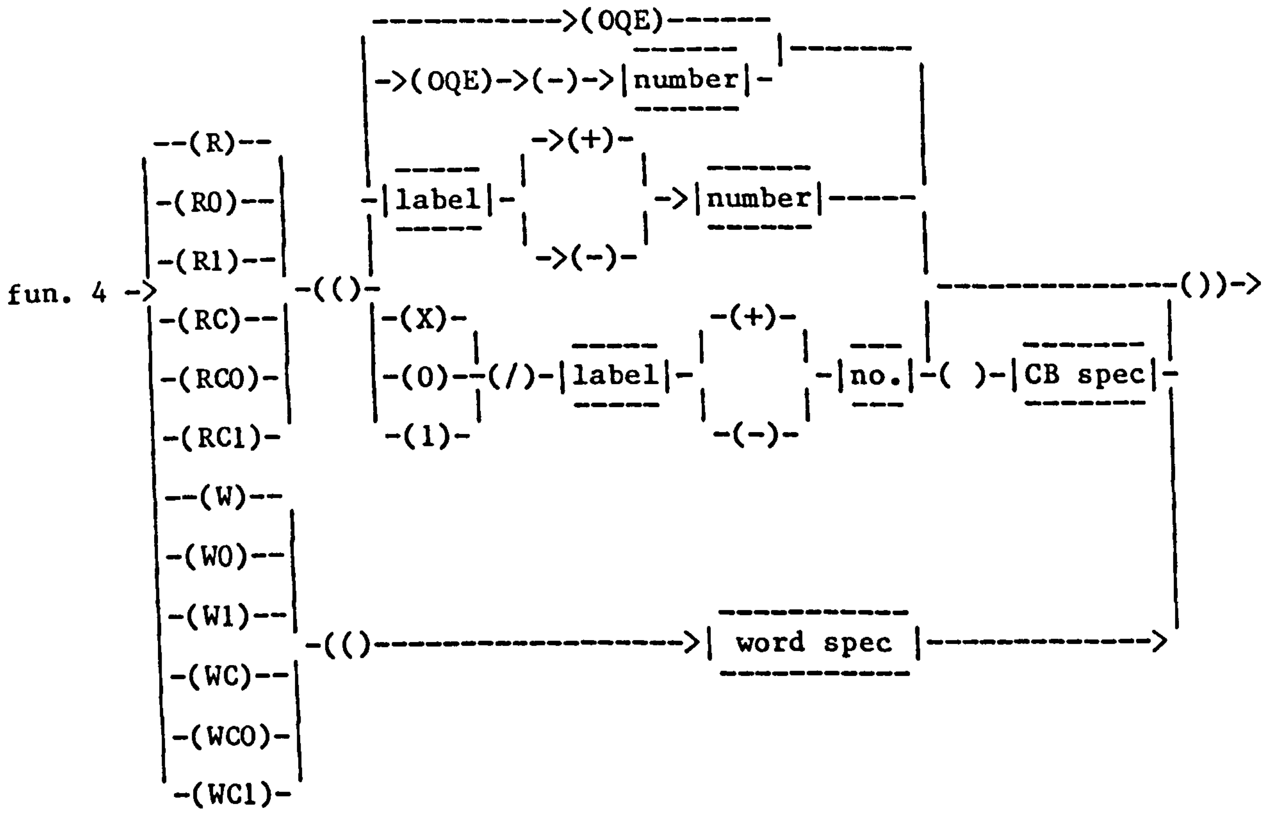


B) The READ Operation :

The READ operation will update the Control Bits of activated word-row with the effective data content of IDR, and will simultaneously, read the content of the activated word-row (included both Ch.spec. and CB spec.) to the ODR, and then to the Output Buffer.

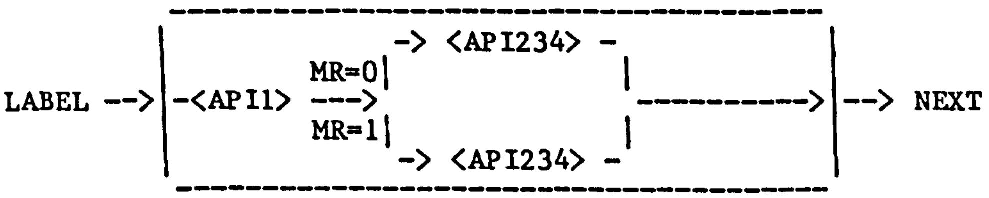
```
S('T' X1XX)BMR +1,+2
M(CLBTT)
RSTTU(S)
R(OQE 1XXX)BRN   @NEXT    ; READ CONTENT OF TAGGED WORD-
                          ; ROW TO OUTPUT BUFFER
```





2) The Branch Operation :

In a complete Associative Computation Cycle, two types of instruction addressing formats are used: one for <API1> and the other for <API234>.



Because of the possibility of two <API234> pending on the outcome of Match reply (MR), <API1> uses a 2-address format. As <API234> leads to the completion of the Associative Computation Cycle, only a one-address fomat is needed to select the next instruction. Similar to <MR branch>, relative address is used for the selection of next instruction, in this case, either another Associative Computation Cycle or return to the flow of control to the Host Processor:

A) The Selection of another ACC :

The address of the next ACC will, in this case, has to be explicitly included in the <API234> format.

```
W('G' X1XX)BRN  +1          ; SELECT THE NEXT API IN THE
                            ; CONSECUTIVE LOCATION OF API
                            ; PROGRAM STORE
                            ; APC = APC + 1

R(OQE 1XXX)BRN  @NSTEP      ; SELECT THE NEXT API LABELED
                            ; @NSTEP IN THE API PROGRAM
                            ; STORE
```

B) Return the Flow of Control to the Host :

The return of control is signified by totally omitting the next instruction address.

```
W('G' X1XX)             ; RETURN CONTROL TO THE HOST
```

```
                          --------
               -------->| empty |--------
              |           --------        |
branch  -->                                -->
              |            ---------      |
               -->( BRN )--->| label0 |--
                            ---------
```

4.3.2 The Group-Run Pre-Function <API234>

The Group Run operation activates all word-rows between Tags of TR1 to Tags of TR2. Hence a second search is needed in Beat 2 to set the tags for TR2 before the Group Run operation can actually take place.

```
                   -----------   -------------   ------------
group run -->|GR search 2|-->|GR operation 3|-->|GR function 4|-->
                   -----------   -------------   ------------
```

API2 : SEARCH Operation for TR2

GRS : Group Run Search

GRSC : Group Run Search with Complement Tags

<GR SEARCH 2> ::= GRS<BSU>(<WORD SPEC>)<TAB>;<COMMENT><CR>

When Group Run is specified, <API 2> is such that :

1) Tag Register TR2 will be reset.

2) CLEAR options will be inhibited, as the tags in TR1 will be needed for Group Run operation in Beat 3.

3) A second search operation will be initialized in which the AMA will be searched for the domain of word-rows which match the effective data content of IDR as interpreted by the <complement> and <DI>. But unlike the Beat 1 SEARCH operation, all matching word-rows will be tagged in TR2 instead of TR1.

4) MR is set if one or more tags are set in TR2.

```
e.g.   S('T' X1XX)BMR +1,+2
       GRS('$' XXX1)              ; GROUP-RUN SEARCH FOR '$'
       GRN(U)
       W(0000)BRN      @NEXT
```

```
                  Ch. spec.     CB spec.
             -------------------------
       IDR  |          $        |X|X|X|1|
             -------------------------
             | | | | | | | |   | | | |
             -------------------------
       BSU  | | | | | | | | |  |e|e|e|e|
             -------------------------
             | | | | | | | |   | | | |
             v v v v v v v v   v v v v     TR1    TR2    WSU
             -------------------------     ---    ---    ---
            |         T         | |1| |0|  | 1 |  | 0 |  |   |
            |         3         |C| |C| |  | 0 |  | 0 |  |   |
            |         T         | |1| |1|  | 1 |  | 0 |  |   |
       AMA  |         4         |B| |B| |  | 0 |  | 0 |  |   |
            |         $         | | | |1|  | 0 |  | 1 |  |   |
            |         #         |1| |3| |  | 0 |  | 0 |  |   |
            |         T         | |1| |0|  | 1 |  | 0 |  |   |
             -------------------------     ---    ---    ---
```

```
 G
 R

 s     ->(GRS)--    -----
 e    |         | |      |     ---------       -------    ----
 a ->             ---(0)-->(()-|word spec|-();)-|comment|-| cr |->
 r    |         | |      |     ---------       -------    ----
 c     ->(GRSC)-    -(1)-
 h

 2
```

API3 : Group Run Operations

GRN : Group Run

RSFGU : Resolve First Group Up

RSFGD : Resolve First Group Down

RSGSU : Resolve Group Start Up

RSGSD : Resolve Group Start Down

RSFGSU : Resolve First Group Start Up

RSFGSD : Resolve First Group Start Down

<GR OP.3> ::= <TAB><GR OPTION>(<DIRECTION>)<TAB>;<COMM.><CR>

A) The Group Run :

The Group Run activates all word-rows from (but not including except when S = 1) those word-rows having a tag set in TR1 to (and including) the first occurrences tagged word-rows in TR2 as indicated below.

<Direction>			Contents of Tags Registers TR1 and TR2																							
<U>	<S>	<D>	TR1	T	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	B
			TR2		0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	1	
0	0	0																								
0	0	1								a	a						a	a	a	a				a	a	
0	1	0							a							a							a			
0	1	1							a	a	a					a	a	a	a	a			a	a	a	
1	0	0				a	a	a						a	a					a	a	a				
1	0	1				a	a	a		a	a			a	a		a	a	a	a	a	a		a	a	
1	1	0				a	a	a	a					a	a	a				a	a	a	a			
1	1	1				a	a	a	a	a	a			a	a	a	a	a	a	a	a	a	a	a	a	

In BOAP, Group Run can proceed over a number of chip modules, by allowing the modules to execute the run in parallel in two phases. In the first phase, a group run is performed inside each module, culminating in a set of output signals from the modules indicating the position at which the group run is to be continued. In the second phase, these signals (MR in beat 2 and PLT or PLB in beat 3) are picked up and used to link up adjacent modules and allow the group run to proceed to completion.

GRN(D) :

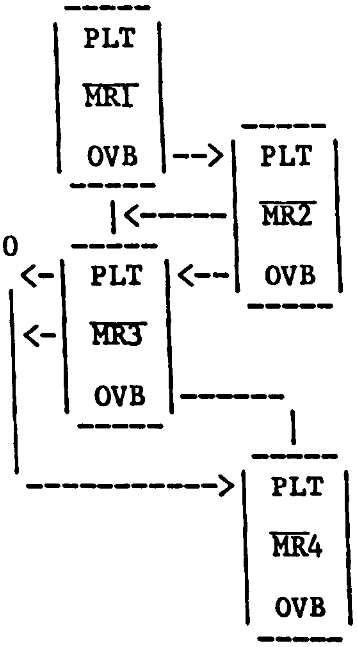
Phase 1 : PLT 1 = PLT 2 = PLT 3 = PLT 4 = 0

Phase 2 : PLT 1 = 0

PLT 2 = OVB 1

PLT 3 = OVB 2 + $\overline{\text{MR 2}}$ * PLT 2

PLT 4 = OVB 3 + $\overline{\text{MR 3}}$ * PLT 3



GRN(U) :

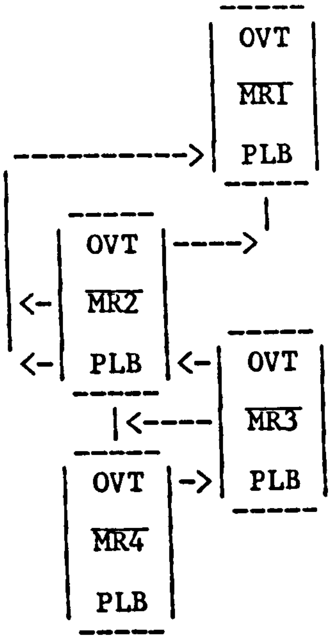
Phase 1 : PLB 4 = PLB 3 = PLB 2 = PLB 1 = 0

Phase 2 : PLB 4 = 0

PLB 3 = OVT 4

PLB 2 = OVT 3 + $\overline{\text{MR 3}}$ * PLB 3

PLB 1 = OVT 2 + $\overline{\text{MR 2}}$ * PLB 2



Hence the total time for a inter-module group run is the time taken for a group run within a module plus the propagation delays of module linking logic and an EIR (End In Run).

B) Resolve First Group Up (RSFGU)

The "Resolve First Group Up" comprise a group run followed by a resolve group option, executed in the specified direction (U or D), which inhibits all but the first group of word-rows from the B-end for function execution.

<Direction>			Contents of Tags Registers TR1 and TR2																							
<U>	<S>	<D>	TR1	T	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	B
			TR2		0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	1	
0	0	0																								
0	0	1																						a	a	
0	1	0																					a			
0	1	1																					a	a	a	
1	0	0																	a	a	a					
1	0	1																						a	a	
1	1	0																	a	a	a	a				
1	1	1												a	a	a	a	a	a	a	a	a	a	a	a	

C) Resolve First Group Down (RSFGD)

The "Resolve First Group Down" is a resolve group run option, which inhibits all but the first group of word-rows from the T-end for function execution.

<Direction>	Contents of Tags Registers TR1 and TR2																							
<U> <S> <D>	TR1	T	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	B
	TR2		0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	1	
0 0 0																								
0 0 1								a	a															
0 1 0							a																	
0 1 1							a	a	a															
1 0 0				a	a	a																		
1 0 1				a	a	a																		
1 1 0				a	a	a	a																	
1 1 1				a	a	a	a	a	a															

D) Resolve Group Start Up (RSGSU)

The "Resolve Group Start Up" is a resolve group run option, which inhibits all but the first word-rows from the B-end of every activated group for function execution.

<Direction>	Contents of Tags Registers TR1 and TR2																							
<U> <S> <D>	TR1	T	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	B
	TR2		0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	1	
0 0 0																								
0 0 1									a									a					a	
0 1 0							a							a							a			
0 1 1									a									a					a	
1 0 0						a							a							a				
1 0 1						a			a				a							a			a	
1 1 0							a							a							a			
1 1 1									a														a	

E) Resolve Group Start Down (RSGSD)

The "Resolve Group Start Down" is a resolve group run option, which inhibits all but the first word-rows from the T-end of every activated group for function execution.

<Direction>			Contents of Tags Registers TR1 and TR2																							
<U>	<S>	<D>	TR1	T	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	B
			TR2		0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	1	
0	0	0																								
0	0	1								a							a							a		
0	1	0							a							a							a			
0	1	1							a							a							a			
1	0	0				a								a						a						
1	0	1				a				a				a			a							a		
1	1	0				a								a						a						
1	1	1				a								a												

F) Resolve First Group Start Up (RSFGSU)

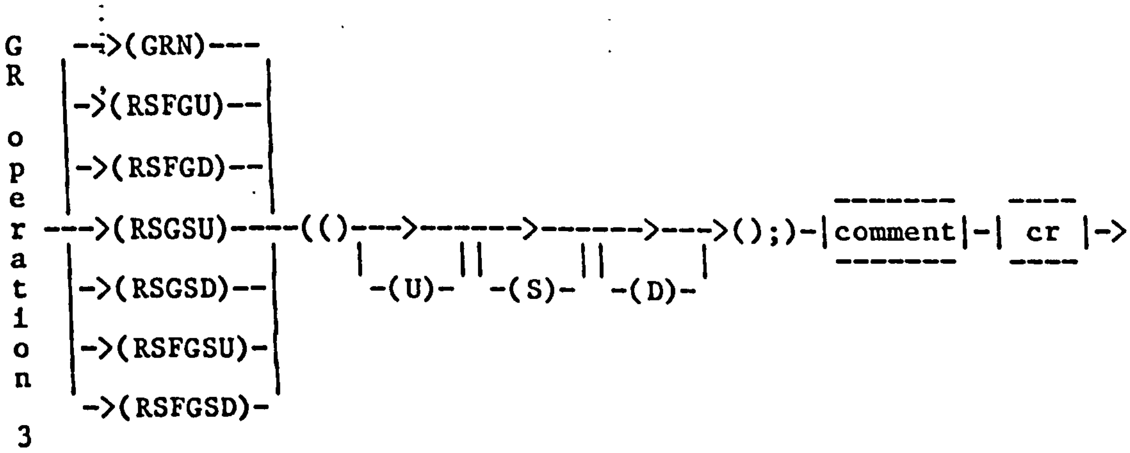
The "Resolve First Group Start Up" is a resolve group run option, which inhibits all but the first word-row of the first activated group from the B-end for function execution.

<Direction>			Contents of Tags Registers TR1 and TR2																							
<U>	<S>	<D>	TR1	T	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	B
			TR2		0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	1	
0	0	0																								
0	0	1																							a	
0	1	0																					a			
0	1	1																							a	
1	0	0																				a				
1	0	1																							a	
1	1	0																					a			
1	1	1																							a	

G) Resolve First Group Start Down (RSFGSD)

The "Resolve First Group Start Down" is a resolve group run option, which inhibits all but the first row of the activated group from the T-end for function execution.

<Direction>	Contents of Tags Registers TR1 and TR2																							
<U> <S> <D>	TR1	T	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	B
	TR2		0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	1	
0 0 0																								
0 0 1								a																
0 1 0							a																	
0 1 1							a																	
1 0 0				a																				
1 0 1				a																				
1 1 0				a																				
1 1 1				a																				



<u>API4 : Group Run Functions</u>

```
<GR FUNCTION 4> ::= <TAB>W(<CB4 SPEC>)|
                    <TAB>R(<CB4 SPEC>)

<CB4 SPEC> ::= <BINARY><BINARY><BINARY><BINARY>

<BINARY> ::= 0|1
```

In the Distributed Computer System, every <API234> is assembled into a 48-bit Associative Machine Instruction (AMI) which to a certain extent has imposed restrictions on the amount of information that we might wish to carry. This is certainly true in the case of <GR Function 4>, in which the READ/WRITE operation is only possible in the Control-Bit Field due to the fact that there are only 4 bits out of the 48-bit AMI remain unused after API2 and API3. Therefore, in order to read from or write to the Character Field of AMA, at least one more Associative Computation Cycle is needed: the first Associative Computation Cycle (the Group-Run Associative Computation Cycle) to activate and mark the Control-Bits of those appropriate word-rows, and the second Associative Computation Cycle does the actual writing to, or resolving and reading from the <ch. spec.> of those activated word-rows.

```
e.g.   S('T' X1XX)BMR +1,+2
       GRS('$' XXX1)
       GRN(U)
     : W(0001)BRN    @NEXT    ; WRITE <CB4 SPEC> = 0001
```

```
                  Ch. spec.     CB spec.
            ------------------------------
      IDR  |                    |0|0|0|1|
            ------------------------------
                                 | | | |
                                 -------
                          BSU   |e|e|e|e|
                                 -------
                                 | | | |
                                 v v v v     TR1    TR2    WSU
            ------------------------------   ---    ---    ---
           |          T         |0|0|0|1|   | 1 |  | 0 |  | a |
           |          3         |0|0|0|1|   | 0 |  | 0 |  | a |
           |          T         | | | | |   | 1 |  | 0 |  |   |
      AMA  |          4         | | | | |   | 0 |  | 0 |  |   |
           |          $         |0|0|0|1|   | 0 |  | 1 |  | a |
           |          #         |0|0|0|1|   | 0 |  | 0 |  | a |
           |          T         | | | | |   | 1 |  | 0 |  |   |
            ------------------------------   ---    ---    ---
```

Generally speaking, the READ operation is not permitted in this Associative Computation Cycle, except in the case of <RSFGSU> and <RSFGSD>, by which only one word-row will be activated, thus avoiding the problem of reading "multiple-responses"[26]. In these cases, the function will update the word-row with the effective data content of IDR, and simultaneously, read the content of this word-row into ODR.

```
e.g.    S('T' X1XX)BMR +1,+2
        GRS('$' XXX1)
        RSFGSU(U)
        R(1000)BRN     @NEXT    ; READ THE TAGGED WORD-ROW TO
                                ; OQE, AND UPADTE CB4 = 1000
```

```
                  Ch. spec.     CB spec.
             -------------------------
       IDR  |                |1|0|0|0|
             -------------------------
                              | | | |
                              -------
                         BSU |e|e|e|e|
                              -------
                              | | | |
                              v v v v    TR1    TR2    WSU
             -------------------------   ---    ---    ---
            |        T       | | | | |  | 1 |  | 0 |  |   |
           :|        3       | | | | |  | 0 |  | 0 |  |   |
            |        T       | | | | |  | 1 |  | 0 |  |   |
       AMA  |        4       | | | | |  | 0 |  | 0 |  |   |
            |        $       | | | | |  | 0 |  | 1 |  |   |
            |        #       |1|0|0|0|  | 0 |  | 0 |  | a |
           :|        T       | | | | |  | 1 |  | 0 |  |   |
             -------------------------   ---    ---    ---
           ,| | | | | | | |   | | | |
            v v v v v v v v   v v v v
             -------------------------
       ODR  |        #        |1|0|0|0|
             -------------------------
                     |   |
                     v   v
             -------------------------
            |///////// Output /////////|
            |///////// Buffer /////////|  <- OQE
            |                          |
             -------------------------
```

Hence, as far as <RSFGSU> and <RSFGSD> are concerned, they could both have WRITE and READ operations in API4.

```
               -(W)-          -(0)-    -(0)-    -(0)-    -(0)-
              |     |        |     |  |     |  |     |  |     |
GR fun. 4 ->          ->(()-         -        -        -         -())->
              |     |        |     |  |     |  |     |  |     |
               -(R)-          -(1)-    -(1)-    -(1)-    -(1)-
```

4.2.3 The Post-Function <API234>

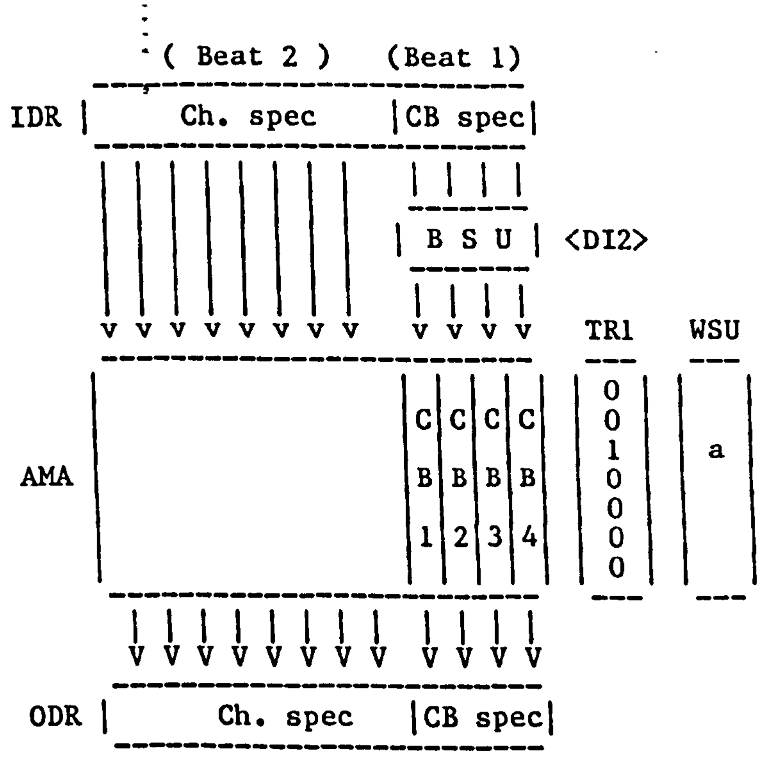
In the Post-Function Associative Computation Cycle, the READ/WRITE function is executed before Beat 3 Tag Manipulations, therefore, word-rows must be activated in Beat 2 before the CLEAR and READ/WRITE function.

```
post-function    ----------   -----   --------
               ->|mod.-fun.2|-|mod.3|-|update 4|-->
 modification    ----------   -----   --------
```

The Post-Function Computation Processing Cycle can be viewed as the combination of two restricted READ/WRITE cycles:

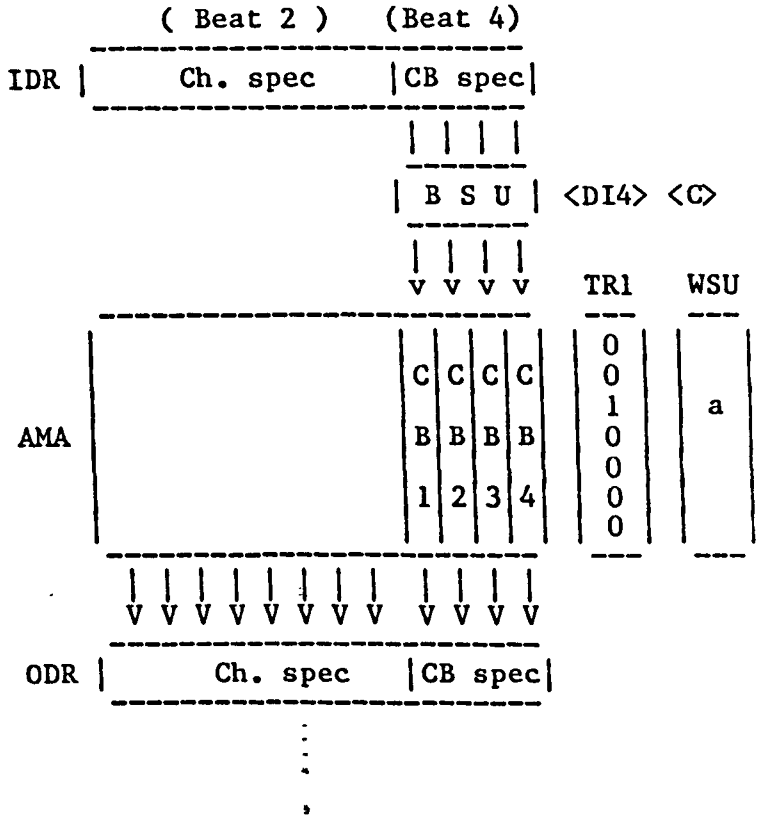
Cycle 1 : (Beat 1 --> Beat 2)

Searches for <word spec.> in Beat 1, then activates them for CLEAR operation in their Control-Bit Fields, and simultaneously executes a READ/WRITE function on the Character Field.



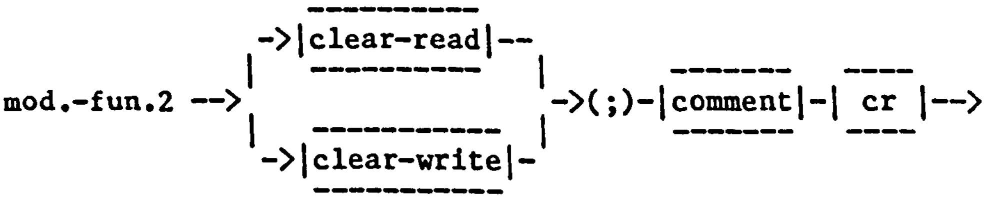
Cycle 2 : (Beat 3 --> Beat 4)

Activates word-rows according to the content of TR1 in Beat 3, then updates their Control-Bit Fields with the <CB spec.> of IDR.



API2 : CLEAR and READ/WRITE Operations

The CLEAR options in the Post-Function <API234> are exactly the same as those in Pre-Function <API234>, except that while executing the CLEAR operation on Control Bit Field, a read or write is carried out on the Character Field.



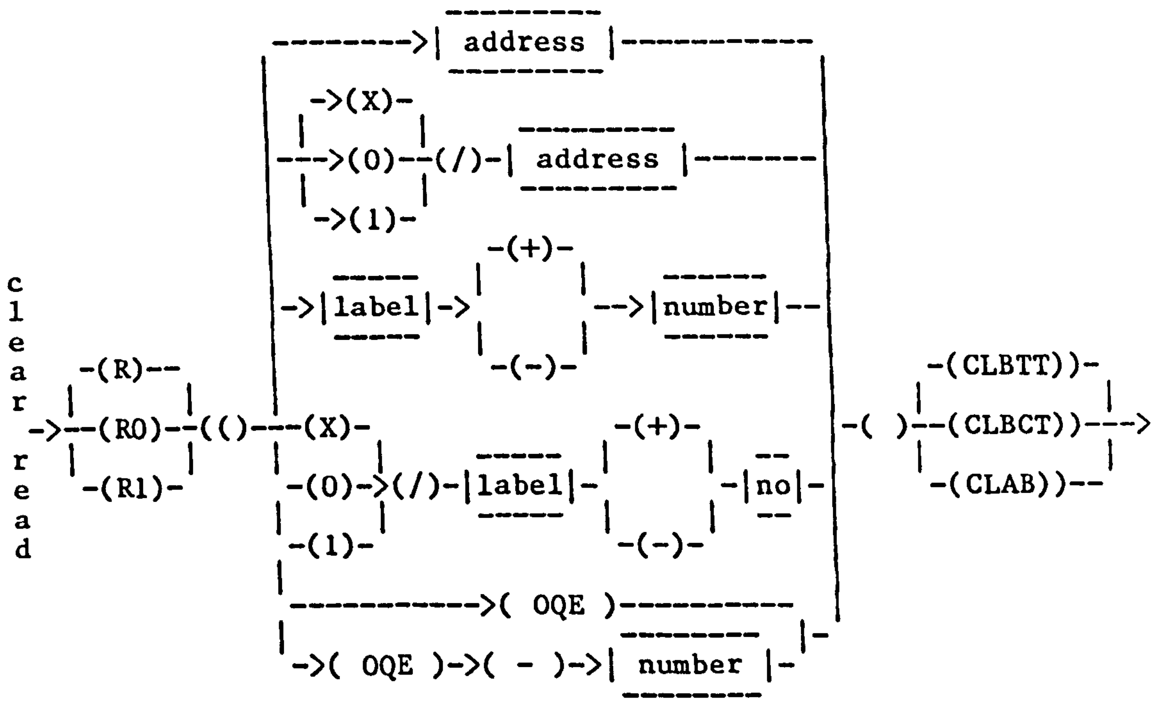
A) CLEAR_READ Operations :

It executes the specified CLEAR option on the Control-Bit Field of the activated word-rows, with the <CB spec> of the IDR remaining from Beat 1 and selected by <DI2>. Simultaneously, it also transfers the contents of the word-row (which activated for the CLEAR operation) to the ODR, before loading it into the Output Buffer.

```
<CLEAR_READ> ::=R<DI>(<SP LOCATION><$><CL>)|
                R<DI>(<OB LOCATION><$><CL>)
```

```
S('T' X1XX)BMR +1,+2
R(OQE CLBTT) .        ; READ TAGGED WORD-ROW TO OQE, AND CLBTT
PTT(U)
U(0XXX)BRN      @NEXT
```

```
              ( Beat 2 )   (Beat 1)
          ---------------------------
   IDR   |                 : |X|1|X|X|
          ---------------------------
                           ,  | | | |
                              -------
                       BSU  |e|e|e|e|  <DI2>
                              -------
                              | | | |
                              V V V V    TR1     WSU
          ---------------------------    ---     ---
         |                 | | | | |   |  0  |  |   |
         |                 | | | | |   |  0  |  |   |
   AMA   |        B        |1|0|0|1|   |  1  |  | a |
         |                 | | | | |   |  0  |  |   |
         |                 | | | | |   |  0  |  |   |
         |                 | | | | |   |  0  |  |   |
         |                 | | | | |   |  0  |  |   |
          ---------------------------    ---     ---
           | | | | | | | |   | | | |
           V V V V V V V V   V V V V
          ----------------------------
   ODR   |        B          |1|0|0|1|
          ----------------------------
                   \|   |/
          ----------------------------
         |///////// Output /////////|
         |///////// Buffer /////////|  <- OQE
         |                          |
          ----------------------------
```

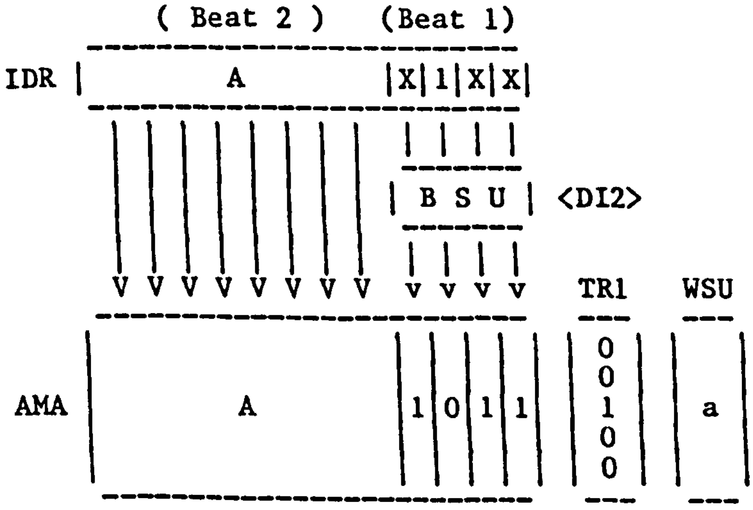


B) CLEAR_WRITE Operations :

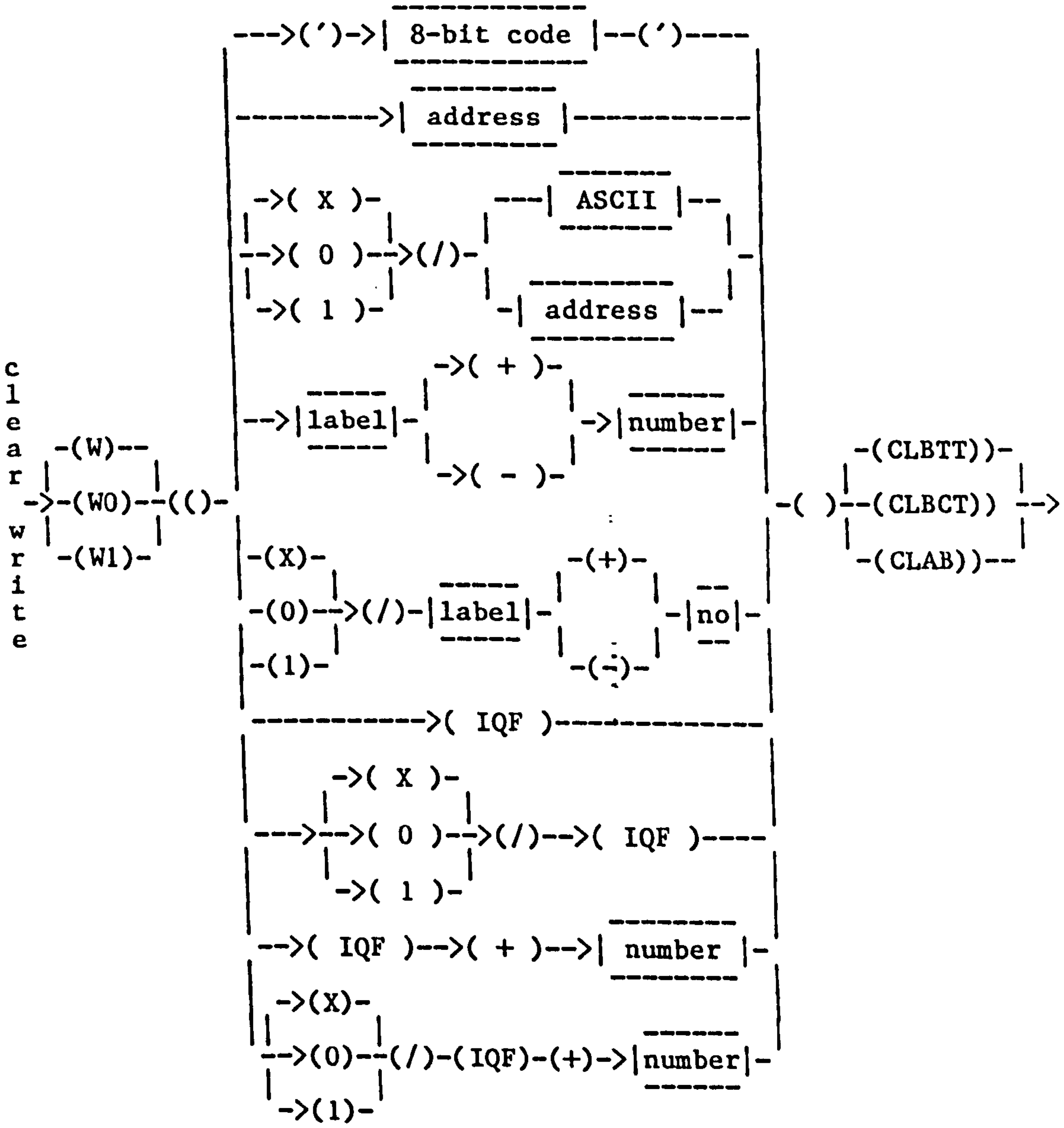
It executes the specified CLEAR option on the Control Bit Field of the activated word-rows, with the <CB spec> of IDR remaining from Beat 1 and selected by the <DI2>. Simultaneously, it also updates all word-rows (which are activated for CLEAR operation) with the <ch.spec> of IDR.

```
<CLEAR_WRITE FUNCTION> ::= W<DI>('<8-BIT CODE>'<$><CL>)|
                           W<DI>(M/'<ASCII>'<$><CL>|
                           W<DI>(<SP LOCATION><$><CL>)|
                           W<DI>(<IB LOCATION><$><CL>)
```

```
S('T' X1XX)BMR +1,+2
W('A' CLBTT)             ; WRITE 'A' TO ALL TAGGED ROWS & CLBTT
PTT(U)
U(0XXX)BRN     @NEXT
```



The CLEAR_READ or CLEAR_WRITE Operations on <Word Spec> in the Post-Function Associative Computation Cycle is split mostly into <Ch Spec> and <CB Spec>. Hence, Post-Function <API234> can apply to Text Mode only.



API3 : Tag Manipulations

The Post-Function's tag manipulations are exactly same as the set used in Pre-Function <API234>.

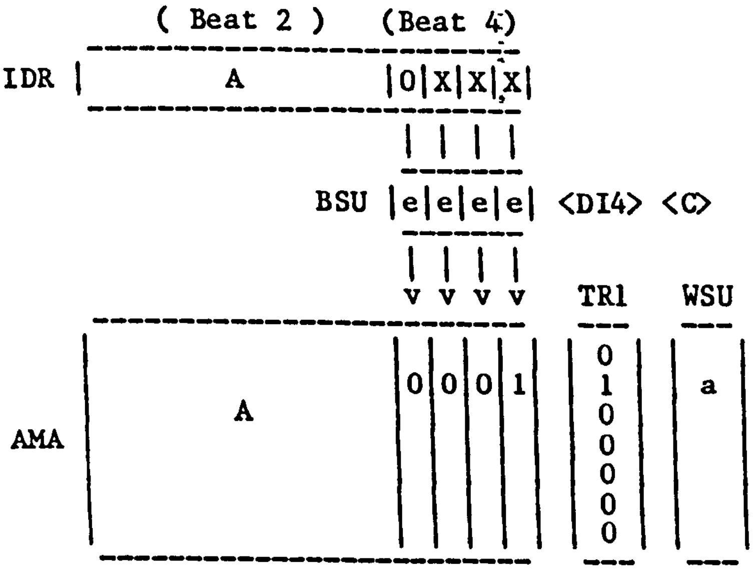
API4 : Update Operation

```
              ----------    --------
API4 -->| update 4 |-->| branch |-->
              ----------    --------
```

The Post-Function <API234> has a somewhat restricted function execution which is a restricted write function on Control-Bit Field only. Hence, the symbol U (Update) is used in place of R (READ) or W (WRITE).

<UPDATE 4> ::= <TAB>U<BSU>(<CB SPEC>)

```
e.g.    S('T' X1XX)BMR +1,+2
        W('A' CLBTT)
        PTT(U)
        U(0XXX)BRN      @NEXT     ; UPDATE ALL TAGGED WORD-ROWS
                                  ; WITH 0XXX
```



```
              ->(U)--     ------
             |        |  |      |
update 4 --> |         ----->(0)--->(()->|CB spec|-(()-->
             |        |  |      |
              ->(UC)-'    ->(1)-'
```

4.3 SUMMARY

The choice of an instruction format is a crucial decision in the system design of a computer, and predetermines to a certain extent the resultant structure of the machine in the top-down design strategy, but is restricted by the structure of the machine in the case of the bottom-up approach. The design of the Associative Processing Instruction has been strongly influenced by the latter case in that a two-part instruction format was adopted to process the Associative Computation Cycle.

The Associative Computation Cycle is organized in three different types of sequencing:

1) The Pre-Function Non-Group-Run Associative Computation Cycle:
Fetch -> Search -> Clear -> Tag-Manipulation -> Read/Write

2) The Pre-Function Group-Run Associative Computation Cycle:
Fetch -> Search(TR1) -> Search(TR2) -> Group-Run -> Read/Write

3) The Post-Function Associative Computation Cycle:
Fetch -> Clear-Read/Write -> Tag-Manipulation -> C.B.Update

In this chapter, the full definition of the Associative Assembly Language (AAL), which comprise both SISD and SIMD facilities, has been presented. The API is really a symbolic form of the Associative Machine Instruction (AMI) which can then be used to drive the microprogrammed associative processor. However, the AAL is designed to provide for people to write program for DCS in a form that is not as unpleasant as the AMI. Programs written in AAL are first translated into a AMI file and a Z-80 file before they can be executed by our Distributed Computer System.

CHAPTER FIVE

THE DESIGN OF ASSOCIATIVE MACHINE INSTRUCTIONS

When designing the Associative Machine Instruction set, the following design criteria for instruction format has been adopted:

1) Short Instruction Format :

First, and the most important, short instructions are better than long instructions:

A) cheaper hardware cost,

B) simpler hardware configuration.

However, this criteria should be carefully applied in order not to achieved shorter instruction format at the expense of longer fetching time.

2) Convenient Word Length :

It is highly desirable for the word length of machine instruction to be an integral multiple of its bus bandwidth. If the data bus between the host processor and BOAP is 8 bits, the word length should be 8-bits, or 16-bits, or 24-bits and so on; otherwise the efficiency of I/O transfer will be in doubt.

3) Short Address Field :

Address field, regardless of whichever it is for (either instruction or operand addressing), has often been considered as a piece of unproductive information within the instruction format. As a result, many attempts have been made to remove it from the instruction format whenever desirable: the stack machine architecture was designed to remove the operand address field, similarly, the program counter scheme was adopted in order to remove the necessity of next instruction's address. However, the computation organization of AMI makes it necessary to preserve both the operand address field and instruction address field. Therefore, the only saving that could be achieved in address fields is to reduce the length of addresses.

In examining the control structure of the Associative Computation Cycle, it has become apparent that the optimum machine instruction format is the combination of two 48-bit codes: one for <AMI1> (Examine Phase of AMI) and other for <AMI234> (Execute Phase of AMI).

```
|<----------------------- AMI 1 ------------------------->|
 1          25   26  27    29  30  33               41   48
 ---------------------------------------------------------
| Wd Spec |TBV|       |DI1| CMB |000|     Label-0    |Label-1|
 ---------------------------------------------------------
```

```
|<----------------------- AMI 234 ----------------------->|
 1          25   26  27    29  30  33    37  39      41   48
 ---------------------------------------------------------
| Wd Spec |PF | R/W |DI4| CMB |USD| ACD |DI2| CLEAR | Label |
 ---------------------------------------------------------
```

1) The Instruction Format :

This is the shortest possible word length for the AMI to accommodate all essential information:

A) AMI1 needs 46 bits to hold information for the Examine Phase

a) Word Spec. requires a 24 bit code

b) Text/Bit-Vector selection requires a 1 bit code

c) Data Identity requires a 2 bits code

d) Data Complementing requires a 1 bit code

f) Two <AMI234> addresses requires 8 bits each

g) SEARCH operation requires a 3-bit opcode

B) AMI234 needs 48 bits to hold information for the Execute Phase

a) Word Spec. requires a 24 bit code

b) Post/Pre-Function selection requires a 1 bit code

c) READ/WRITE selection requires a 1 bit code

d) Data Identity (Beat 2) and Data Identity (Beat 4) require 2 bits each

f) Data Complementing requires a 1 bit code

g) Tag Manipulation requires 4-bit opcode and 3-bit direction code

h) CLEAR operation requires a 2 bit code

i) Next AMI address requires an 8 bit address.

2) The Word Length :

The 48-bit AMI format is an integral of the 8-bit data bus.

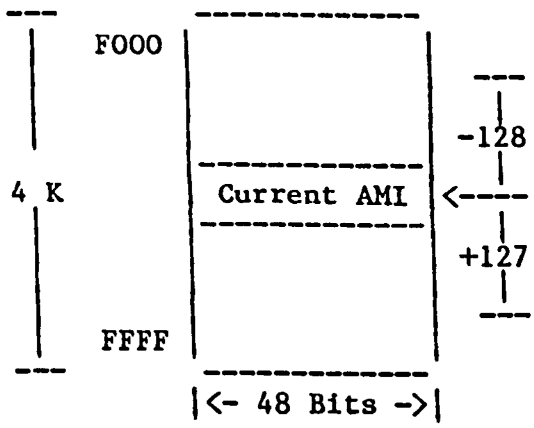
3) The Address Fields :

A) The Operand Addresses:

The immediate addressing mode of AMI needs 24 bits for its 12 bits of tertiary data (8 bits for Ch. Spec., and 4 bits for Control-Bits). This requirement, to a great extent, has set the minimimum length for operand addresses. Nonetheless, the other operand addressing schemes such as Scratch-Pad; Input and Output Buffers modes, also need 24 bits for operand addressing.

B) The Instructure Addresses:

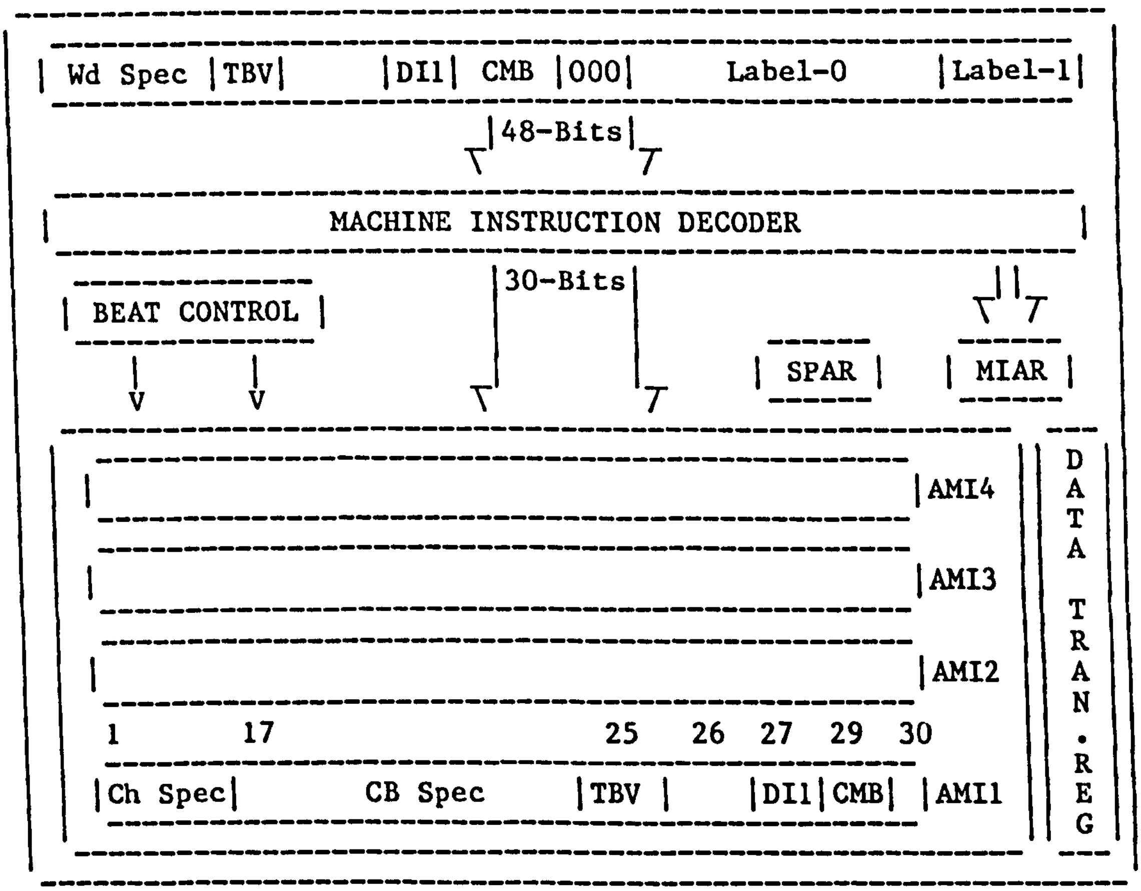
On the other hand, the minimization of instruction addressing is restricted by the size of API Program Store (4k words), and the requirement to branch to all necessary locations. However, in order to keep the AMI format as an integral of the 8-bit data bus, a 8-bit relative instruction address is used to address -128 to +127 locations from the current AMI location within the API Program Store.



In the Distributed Computer System, programs written in AAL are translated by the one-pass AAL assembler, which in turn, will generate the AMI file to drive the hardware of the Distributed Computer System. The details of the AAL assembler are presented in the Appendix C.

5.1 THE EXAMINE PHASE OF AMI <AMI1>

The <AMI1> is the object machine instruction of the <API1> generated by the Associative Assembler. During the Fetch Phase of the Associative Computation Cycle, the <AMI1> is loaded from the Instruction Memory Buffer into the Instruction Register of the Control System, which is then separated by the Machine Instruction Decoder into a 30-bit long operational code and two 8-bit <AMI234> addresses for fetching the alternative execute phase of AMI into Instruction Memory Buffer during the Examine Phase.



The <AMI1> uses the 48-bit instruction format, in which is divided into three different fields.

```
|<-------------------- AMI 1 ------------------------>|
 1           25    26   27      29   30   33                      41    48
-----------------------------------------------------------------------
| Wd Spec |TBV|        |DI1| CMB |000|      Label-0        |Label-1|
-----------------------------------------------------------------------
|<OPERAND>|<------- OPCODE ------>|<-- AMI234 ADDRESSES -->|
```

1) The Operand Field :

The operand field provides the information about operand movement from a source (i.e. from the instruction itself, Input Buffer or Scratch-Pad) to the destination (the IDR).

2) The Opcode Field :

The opcode field provides the information about state transformations. In <AMI1>, it includes three part of opcodes:

A) the Text/Bit-Vector selection (Bit-25)

B) the bit select functions, namely Data Identity (DI1 in Bit-27 & Bit-28) and Data Complementing (CMB in Bit-29)

C) the SEARCH operation (Bit-30 -> Bit-32)

3) The <AMI234> Addresses :

The <AMI234> addresses provide the branching information of the execute parts of AMI, pending on the outcome of the SEARCH operation.

The <AMI1> part of Associative Machine Instruction is distinguishable from the <AMI234> part of AMI in its SEARCH operation, which is indicated by the code 000 in Bit-30 to Bit-32.

5.1.1 The <AMI1> Word Spec.

In <AMI1> Word Spec, the data organization is indicated by the TBV bit (Bit-25) of the <AMI1>.

A) The Text Symbols :

```
1            25   26  27     29  30  33                  41    48
-----------------------------------------------------------------
| Wd Spec  | 1 |      |DI1| CMB |000|      Label-0      |Label-1|
-----------------------------------------------------------------
    24     TBV   1     2     1   3         8                8
|<OPERAND>|<------- OPCODE ------>|<--  AMI234 ADDRESSES -->|
```

B) The Bit-Vector :

```
1            25   26  27     29  30  33                  41    48
-----------------------------------------------------------------
| Wd Spec  | 0 |      |DI1| CMB |000|      Label-0      |Label-1|
-----------------------------------------------------------------
    24     TBV   1     2     1   3         8                8
|<OPERAND>|<------- OPCODE ------>|<--  AMI234 ADDRESSES -->|
```

The three kinds of <AMI1> addressing schemes are shown as follows:

1) The Immediate Addressing Mode :

A tertiary datum format <B> is used to represent the three level logic where <B> ::= X|0|1 ::= 00|01|10

```
  7    6    5    4    3    2    1    0   CB1  CB2  CB3  CB4
 ------------------------------------------------------------
| B  | B  | B  | B  | B  | B  | B  | B  | B  | B  | B  | B  |
 ------------------------------------------------------------
|<------ Character Field ------>|<- C.B. Field->|
|<---- 8-Bit Character Code --->|<- C.B. Field->|
|MSB|<--- 7-Bit ASCII Code ---->|<- C.B. Field->|
|<------------ 12-bit Bit Vector -------------->|
```

2) The Scratch-Pad Addressing Mode :

The Immediate Addressing Mode uses only three combinations of the two-bit code to represent the tertiary datum: namely 00 for X, 01 for 0 and 10 for 1. The fourth combination (11) is therefore, used here to indicate non-immediate addressing modes, notably Scratch-pad, Input and Output Buffer addressing modes.

```
 7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
  2   4   6   8  10  12  14  16  18  20  22  24
 -----------------------------------------------
|?|?|1|1|1|0|?|?|?|?|?|?|?|?|?|?| B | B | B | B |
 -----------------------------------------------
|<------ Addressing Mode ------>|<- C.B. Field->|
|<M>|< >|< >|<--- SP Address--->|
  S    |   |
  B    |    --> Addressing Mode
       |
        --> Non-Immediate Mode
```

Bit 3 & 4 of the <AMI1> Word Spec. is filled with the code 11 to represent non-immediate addressing mode. Bit 5 & 6 is filled with the code 10 to mean Scratch-Pad addressing.

Bit-5	Bit-6	Non-Immediate Addressing Modes
0	0	Output Buffer Addressing
0	1	Input Buffer Addressing
1	0	Scratch-Pad Addressing
1	1	Not defined

The field from Bit-7 to Bit-16 is used to address the 1K Scratch-Pad Buffer.

3) The Input Buffer Addressing :

The Input Buffer Addressing when used, its Bit 5 & 6 of the <AMI1> Word Spec. is filled with the code 01 to indicate Input Buffer Addressing Mode, and from Bit-7 to Bit-16 is used for the address of 1K Input Buffer.

```
 7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
  2   4   6   8  10  12  14  16  18  20  22  24
------------------------------------------------
|?|?|1|1|0|1|?|?|?|?|?|?|?|?|?|?| B | B | B | B |
------------------------------------------------
|<------ Addressing Mode ------>|<- C.B. Field->|
|<M>|< >|< >|<--- IB Address--->|
  S   |   |
  B   |    --> Addressing Mode
      |
       --> Non-Immediate Mode
```

5.1.2 The Bit Select Functions :

The Bit Select Functions of <AMI1> are defined by the <BSU> of <API1> : Data Masking and Data Complementing.

```
|<------------------------ AMI 1 ------------------------->|
 1          25   26   27    29  30  33               41    48
-------------------------------------------------------------
| Wd Spec |TBV|      | ? |   ?   |000|    Label-0     |Label-1|
-------------------------------------------------------------
    24      1    1   DI1   CMB   3         8            8
|<OPERAND>|<------- OPCODE ------>|<--  AMI234 ADDRESSES -->|
```

1) The Data Masking :

Apart from the use of tertiary data format for the unconditional data masking, <AMI1> uses a two bit Data Identity code (DI1) to represent conditional data masking.

Bit-27	Bit-28	Data Identity <DI1>
0	0	<DI1> = X
0	1	<DI1> = 0
1	0	<DI1> = 1
1	1	Not defined

2) The Data Complementing :

A) The selection of true data content of IDR.

```
 1          25    26   27     29   30   33                       41     48
---------------------------------------------------------------------------
| Wd Spec  |TBV|       |DI1|  0   |000|      Label-0           |Label-1|
---------------------------------------------------------------------------
     24      1     1     2   CMB    3           8                   8
```

B) The selection of complemented data content of IDR.

```
 1          25    26   27     29   30   33                       41     48
---------------------------------------------------------------------------
| Wd Spec  |TBV|       |DI1|  1   |000|      Label-0           |Label-1|
---------------------------------------------------------------------------
     24      1     1     2   CMB    3           8                   8
```

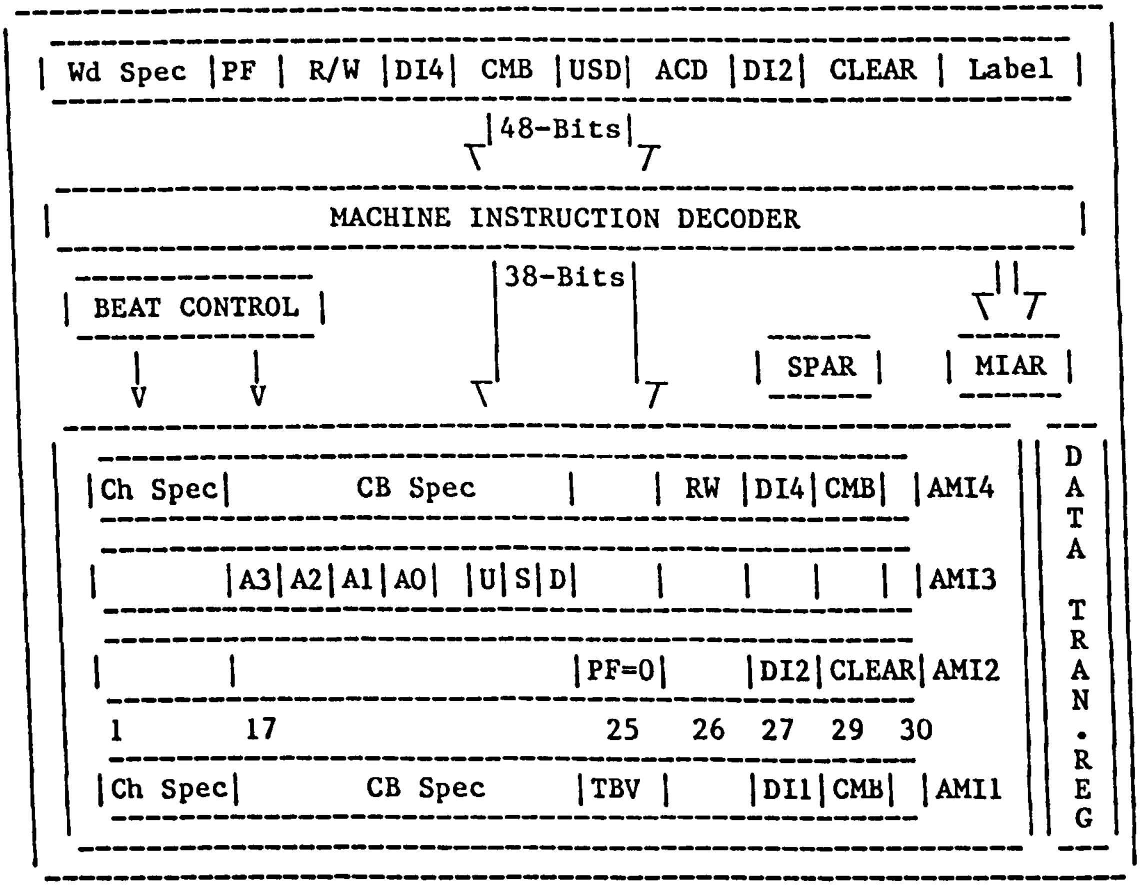
5.1.3 The <AMI234> Addresses :

The <MR branch> of <API1> specifies the addresses of two alternative <API234>'s, notably the <AMI234> addresses which have a range of between -128 to +127.

```
 1          25    26   27     29   30   33                       41     48
---------------------------------------------------------------------------
| Wd Spec  |TBV|       |DI1| CMB  |000|          ?             |   ?    |
---------------------------------------------------------------------------
     24      1     1     2    1     3        Label-0            Label-1
|<OPERAND>|<-------- OPCODE ------->|<--  AMI234 ADDRESSES  -->|
```

5.2 THE EXECUTE PHASE OF AMI <AMI234>

The <AMI234> is the object machine instruction of <API234> which governs the execute phase of the Associative Computation Cycle. During the Examine Phase of the Associative Computation Cycle (<AMI1>), the <AMI234> parts of AMI are loaded from the API Program Store into the Instruction Memory Buffer ready for the Execute Phase of AMI. The appropriate <AMI234> (pending on the outcome of MR) will then be loaded into the BOAP Control System for the Machine Instruction Decoder to separate it into a 8-bit next instruction address and a 38-bit operational code, which in trun, will be assembled into a three beat execute sequence: AMI2 (Beat-2), AMI3 (Beat-3) and AMI (Beat-4).



Wd Spec | PF | R/W | DI4 | CMB | USD | ACD | DI2 | CLEAR | Label
48-Bits
MACHINE INSTRUCTION DECODER
BEAT CONTROL
38-Bits
SPAR
MIAR
Ch Spec | CB Spec | | RW | DI4 | CMB | AMI4
A3 | A2 | A1 | A0 | U | S | D | AMI3
PF=0 | DI2 | CLEAR | AMI2
1 17 25 26 27 29 30
Ch Spec | CB Spec | TBV | DI1 | CMB | AMI1
DATA TRAN. REG

The <AMI234> uses the 48-bit instruction format similar to the <AMI1> which is also divided into three fields, but with different lengths of Opcode field and Next Instruction Address field.

```
|<-------------------- AMI 234 ------------------------->|
 1          25   26  27    29  30  33    37  39     41    48
 ------------------------------------------------------------
| Wd Spec |PF | R/W |DI4| CMB |USD| ACD |DI2| CLEAR | Label |
 ------------------------------------------------------------
|<OPERAND>|<--------------- OPCODE --------------->|- NEXT-|
                                                     INS.
                                                   ADDRESS
```

1) The Operand Field :

The Operand Field provides the information about operand movement from a source (i.e. from the instruction itself, Input Buffer or Scratch-pad) to the destination (either for the tagged word-rows in AMA or via ODR to the Output Buffer).

2) The Opcode Field :

The Opcode Field provides the information about tag manipulations and READ/WRITE functions. In <AMI234>, it includes eight opcode subfields:

A) the selection of Pre/Post Function ACC (Bit-25)

B) the READ/WRITE selection (Bit-26)

C) the bit select functions for Beat-four AMI (<AMI4>), namely Data Identity (DI4 in Bit-27 & Bit-28) and Data Complementing (CMB in Bit-29)

D) the Tag Manipulation code (from Bit-33 to Bit-36), plus a 3-bit direction code (from Bit-30 to Bit-32)

E) the setting of Beat-two Data Identity (DI2 in Bit-37 & Bit-38)

F) the selection of CLEAR operation (Bit-39 to Bit-40)

3) The Next Instruction Addresses :

Since the execution of <AMI234> leads to the completion of the Associative Computation Cycle, it is therefore suggested that the next instruction could either be another Associative Computation Cycle or an order to return the flow of control to the Host processor:

A) When the content of the Next Instruction Address field is equal to zero, it signifies "return the flow of control to the Host processor", and proceed with the next SISD instruction in the Program.

B) However, a non-zero next instruction address will signify the selection of another ACC, which can be anywhere in the range of -128 to +127 from the current location within the API Program Store.

5.2.1 The Pre-Function Non Group-Run <AMI234>

The selection bit (Bit-25) of Pre/Post Function ACC is set to 0 to indicate Pre-Function ACC.

```
|<------- The Pre-Function Non Group-Run <AMI234> -------->|
 1         25   26  27    29  30  33    37  39     41     48
-----------------------------------------------------------
| Wd Spec | 0 | R/W |DI4| CMB |USD| ACD |DI2| CLEAR | Label |
-----------------------------------------------------------
    24    PF    1    2    1    3    4    2     2       8
|<OPERAND>|<-------------- OPCODE --------------->|- NEXT-|
                                                    INS.
                                                   ADDRESS
      :
      :
      ,
```

AMI2 : Clear Options:

The CLEAR options are indicated in <AMI234> format in a two bit code (Bit-39 & Bit-40).

```
 1         25   26  27    29  30  33    37  39     41     48
-----------------------------------------------------------
| Wd Spec | 0 | R/W |DI4| CMB |USD| ACD |DI2|   ?   | Label |
-----------------------------------------------------------
    24    PF    1    2    1    3    4    2   CLEAR     8
```

Bit-39	Bit-40	CLEAR Options
0	0	No Clear
0	1	CLBTT
1	0	CLBCT
1	1	CLAB

AMI3 : Tag Manipulations:

The Non Group-Run Tag Manipulations of the <AMI234> provide the mechanism to activate word-rows for READ/FUNCTION operation:

```
 1           25   26  27     29  30  33     37  39      41     48
 --------------------------------------------------------------
| Wd Spec | 0 | R/W |DI4| CMB | ? |  ?   |DI2| CLEAR | Label |
 --------------------------------------------------------------
     24     PF    1     2    1   USD  ACD   2     2        8
```

There are a total of nine Non Group-Run Tag Manipulation codes which together with the seven Group-Run codes, make up 16 tag activation codes.

Bit-33	Bit-34	Bit-35	Bit-36	Tag Manipulations
0	0	0	0	No Operation
0	0	0	1	PTT
0	0	1	0	PCT
0	0	1	1	RSTTU
0	1	0	0	RSTTD
0	1	0	1	RSCTU
0	1	1	0	RSCTD
0	1	1	1	EIR
1	0	0	0	MOR

In addition to the above activation codes, three bits of complementary codes are used to indicate propagation direction:

Bit-30	Bit-31	Bit-32	Activation Direction
0	0	0	
0	0	1	D
0	1	0	S
0	1	1	SD
1	0	0	U
1	0	1	U D
1	1	0	US
1	1	1	USD

AMI4 : READ/WRITE Operation :

Associated with the READ/WRITE function, there are four pieces of information:

```
1             25   26  27    29  30  33     37  39      41      48
-------------------------------------------------------------------
|    ?     :  | 0 |  ?   | ? |  ?   |USD| ACD |DI2| CLEAR | Label |
-------------------------------------------------------------------
  Wd Spec     PF   R/W   DI4  CMB    3    4     2     2       8
```

1) The <AMI234> Word Spec. :

The <AMI234> Word Spec. has three kinds of addressing schemes similar to <AMI1> Word Spec.:

A) The Immediate Addressing Mode

```
 7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
-------------------------------------------------
| B | B | B | B | B | B | B | B | B | B | B | B |
-------------------------------------------------
|<------ Character Field ------>|<- C.B. Field->|
|<--- 8-Bit Character Code ---->|<- C.B. Field->|
|MSB|<--- 7-Bit ASCII Code ---->|<- C.B. Field->|
|<----------- 12-bit Bit Vector --------------->|
```

B) The Scratch-Pad Addressing Mode

```
 7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
  2   4   6   8   10  12  14  16  18  20  22  24
 -----------------------------------------------
|?|?|1|1|1|0|?|?|?|?|?|?|?|?|?|?| B | B | B | B |
 -----------------------------------------------
|<------ Addressing Mode ------->|<- C.B. Field->|
|<M>|< >|< >|<--- SP Address--->|
  S   |   |
  B   |    --> Addressing Mode
      |
       --> Non-Immediate Mode
```

C) The Input Buffer Addressing

```
 7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
  2   4   6   8   10  12  14  16  18  20  22  24
 -----------------------------------------------
|?|?|1|1|0|1|?|?|?|?|?|?|?|?|?|?| B | B | B | B |
 -----------------------------------------------
|<------ Addressing Mode ------>|<- C.B. Field->|
|<M>|< >|< >|<--- IB Address--->|
  S   |   |
  B   |    --> Addressing Mode
      |
       --> Non-Immediate Mode
```

D) The Output Buffer Addressing

For the Output Buffer Addressing, the Bit 5 & 6 of the <AMI234> Word Spec. is filled with the code 00 to indicate Output Buffer Addressing Mode, and from Bit-7 to Bit-16 is used for the address of 1K Output Buffer.

```
 7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
  2   4   6   8   10  12  14  16  18  20  22  24
 -----------------------------------------------
|?|?|1|1|0|0|?|?|?|?|?|?|?|?|?|?| B | B | B | B |
 -----------------------------------------------
|<------ Addressing Mode ------>|<- C.B. Field->|
|<M>|< >|< >|<--- OB Address--->|
  S   |   |
  B   |    --> Addressing Mode
      |
       --> Non-Immediate Mode
```

2) The <AMI234> Bit Select Functions :

The Bit Select Functions of <AMI234> are defined by the <BSU> of <API234> : Data Masking and Data Complementing.

```
1           25    26   27     29   30   33     37   39      41      48
----------------------------------------------------------------------
| Wd Spec | 0 | R/W | ? |   ?   |USD| ACD |DI2| CLEAR | Label |
----------------------------------------------------------------------
     24     PF     1    DI4   CMB    3     4     2      2        8
```

A) The Data Masking :

The <AMI234> uses a two bit Data Identity code <DI4> to represent conditional data masking.

Bit-27	Bit-28	Data Identity <DI4>
0	0	<DI4> = X
0	1	<DI4> = 0
1	0	<DI4> = 1
1	1	Not defined

B) The Data Complementing :

a) The selection of true data content of IDR.

```
1           25    26   27     29   30   33     37   39      41      48
----------------------------------------------------------------------
| Wd Spec | 0 | R/W |DI4|   0   |USD| ACD |DI2| CLEAR | Label |
----------------------------------------------------------------------
     24     PF     1     2    CMB    3     4     2      2        8
```

b) The selection of complemented data content of IDR.

```
1           25    26   27     29   30   33     37   39      41      48
----------------------------------------------------------------------
| Wd Spec | 0 | R/W |DI4|   1   |USD| ACD |DI2| CLEAR | Label |
----------------------------------------------------------------------
     24     PF     1     2    CMB    3     4     2      2        8
```

3) The READ/WRITE Function :

The READ/WRITE Function is indicated by Bit-26

A) The Selection of READ Function

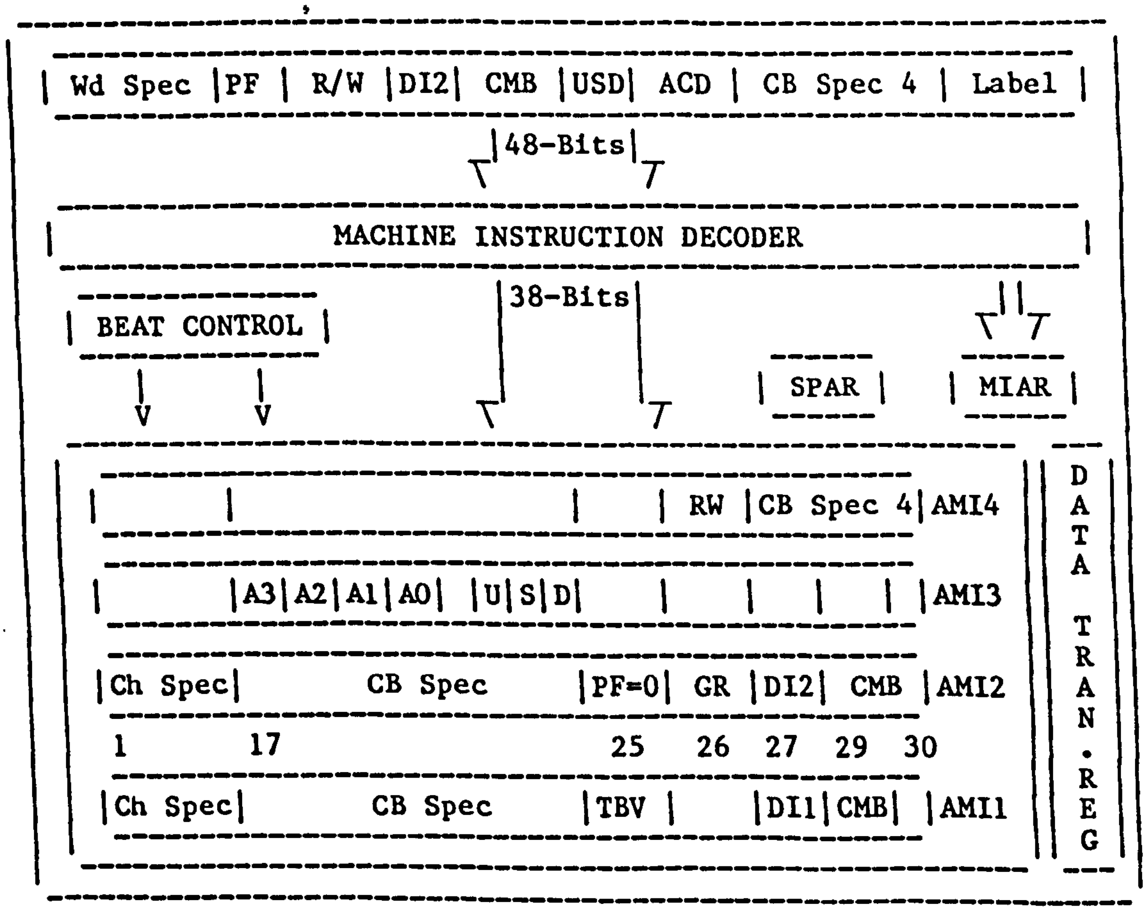
```
1          25   26  27     29  30  33    37  39      41      48
----------------------------------------------------------------
| Wd Spec | 0 |  1   |DI4| CMB |USD| ACD |DI2| CLEAR | Label |
----------------------------------------------------------------
     24     PF  R/W   2    1    3    4    2     2        8
```

B) The Selection of WRITE Function

```
1          25   26  27     29  30  33    37  39      41      48
----------------------------------------------------------------
| Wd Spec | 0 |  0   |DI4| CMB |USD| ACD |DI2| CLEAR | Label |
----------------------------------------------------------------
     24     PF  R/W   2    1    3    4    2     2        8
```

5.2.2 The Pre-Function Grounp-Run <AMI234>

The decoding of the Pre-Function Group-Run <AMI234> sequence follows a very similar fashion to the Pre-Function <AMI234>, except with differences in the field allocation within instruction format.


| Wd Spec |PF | R/W |DI2| CMB |USD| ACD | CB Spec 4 | Label |
|48-Bits|
MACHINE INSTRUCTION DECODER
BEAT CONTROL
38-Bits
SPAR
MIAR
| | | | RW |CB Spec 4|AMI4
| |A3|A2|A1|A0| |U|S|D| | | | | |AMI3
|Ch Spec| CB Spec |PF=0| GR |DI2| CMB |AMI2
1 17 25 26 27 29 30
|Ch Spec| CB Spec |TBV | |DI1|CMB| |AMI1
DATA TRAN.REG

Comparing with the Non Group-Run <AMI234>, the Pre-Function Group-Run <AMI234> uses a slightly different AMI format.

```
|<---------- The Pre-Function Group-Run <AMI234> ---------->|

 1          25   26  27    29  30  33    37         41     48
 -----------------------------------------------------------
| Wd Spec | 0 | R/W |DI2| CMB |USD| ACD | CB Spec 4 | Label |
 -----------------------------------------------------------
    24     PF    1    2    1    3    4        4         8
|<OPERAND>|<--------------- OPCODE ---------------->|- NEXT-|
                                                      INS.
                                                     ADDRESS
```

AMI2 : SEARCH Operation for TR2

Three pieces of informations are involved in the <AMI2> SEARCH as similar to <AMI1> SEARCH, namely <Word Spec>, DI2 and CMB.

```
 1          25   26  27    29  30  33    37         41     48
 -----------------------------------------------------------
|    ?     | 0 | R/W | ? |  ?  |USD| ACD | CB Spec 4 | Label |
 -----------------------------------------------------------
  Wd Spec   PF    1   DI2  CMB   3    4        4         8
```

1) The <AMI2> Word Spec. :

The <AMI2> Word Spec is actually the same as the <AMI1>.

A) The Immediate Addressing Mode :

```
  7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
 -----------------------------------------------
| B | B | B | B | B | B | B | B | B | B | B | B |
 -----------------------------------------------
|<------ Character Field ------>|<- C.B. Field->|
|<---- 8-Bit Character Code --->|<- C.B. Field->|
|MSB|<--- 7-Bit ASCII Code ---->|<- C.B. Field->|
|<----------- 12-bit Bit Vector --------------->|
```

B) The Scratch-Pad Addressing Mode :

```
   7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
    2   4   6   8   10  12  14  16  18  20  22  24
 ------------------------------------------------
|?|?|1|1|1|0|?|?|?|?|?|?|?|?|?|?| B | B | B | B |
 ------------------------------------------------
|<------ Addressing Mode ------>|<- C.B. Field->|
|<M>|< >|< >|<--- SP Address--->|
  S    |   |
  B    |    --> Addressing Mode
       |
        --> Non-Immediate Mode
```

C) The Input Buffer Addressing

```
  7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
   2   4   6   8  10  12  14  16  18  20  22  24
 -------------------------------------------------
|?|?|1|1|0|1|?|?|?|?|?|?|?|?|?|?| B | B | B | B |
 -------------------------------------------------
|<------ Addressing Mode ------>|<- C.B. Field->|
|<M>|< >|< >|<--- IB Address--->|
  S   |   |
  B   |   --> Addressing Mode
      |
      --> Non-Immediate Mode
```

2) The <AMI2> Bit Select Functions

The <AMI2> Bit Select Functions are defined by the <BSU> of <API2> : Data Masking and Data Complementing.

```
 1          25   26   27    29   30  33    37          41     48
 ---------------------------------------------------------------
| Wd Spec | 0 | R/W | ? |   ?   |USD| ACD | CB Spec 4 | Label |
 ---------------------------------------------------------------
     24     PF    1   DI2   CMB   3    4        4         8
```

A) The Data Masking :

Bit-27	Bit-28	Data Identity <DI2>
0	0	<DI2> = X
0	1	<DI2> = 0
1	0	<DI2> = 1
1	1	Not defined

B) The Data Complementing :

a) The selection of true data content of IDR.

```
 1          25   26   27    29   30  33    37          41     48
 ---------------------------------------------------------------
| Wd Spec | 0 | R/W |DI2|   0   |USD| ACD | CB Spec 4 | Label |
 ---------------------------------------------------------------
     24     PF    1    2    CMB   3    4        4         8
```

b) The selection of complemented data content of IDR.

```
1          25   26   27     29   30   33    37            41     48
-----------------------------------------------------------------------
| Wd Spec | 0 | R/W |DI2|   1   |USD| ACD | CB Spec 4 | Label |
-----------------------------------------------------------------------
    24      PF    1    2    CMB    3    4         4          8
```

AMI3 : Group-Run Operation

The Group-Run operations of the <AMI234> provides the information for the activation of word-rows for READ/WRITE Function:

```
1          25   26   27     29   30   33    37            41     48
-----------------------------------------------------------------------
| Wd Spec | 0 | R/W |DI2| CMB | ? |  ?  | CB Spec 4 | Label |
-----------------------------------------------------------------------
    24·     PF    1    2     1   USD   ACD        4          8
```

There are a total of seven Group-Run operational codes which together with the nine Non Group-Run codes, make up 16 tag activation codes.

Bit-33	Bit-34	Bit-35	Bit-36	Tag Manipulations
1	0	0	1	GRN
1	0	1	0	RSGSU
1	0	1	1	RSGSD
1	1	0	0	RSGSU
1	1	0	1	RSFGD
1	1	1	0	RSFGSU
1	1	1	1	RSFGSD

In addition to the above activation codes, three bits of complementary codes are used to indicate propagation direction, as similar to the Non-Group Tag Manipulation.

Bit-30	Bit-31	Bit-32	Activation Direction
0	0	0	
0	0	1	D
0	1	0	S
0	1	1	SD
1	0	0	U
1	0	1	U D
1	1	0	US
1	1	1	USD

AMI4 : Restricted READ/WRITE OPERATION

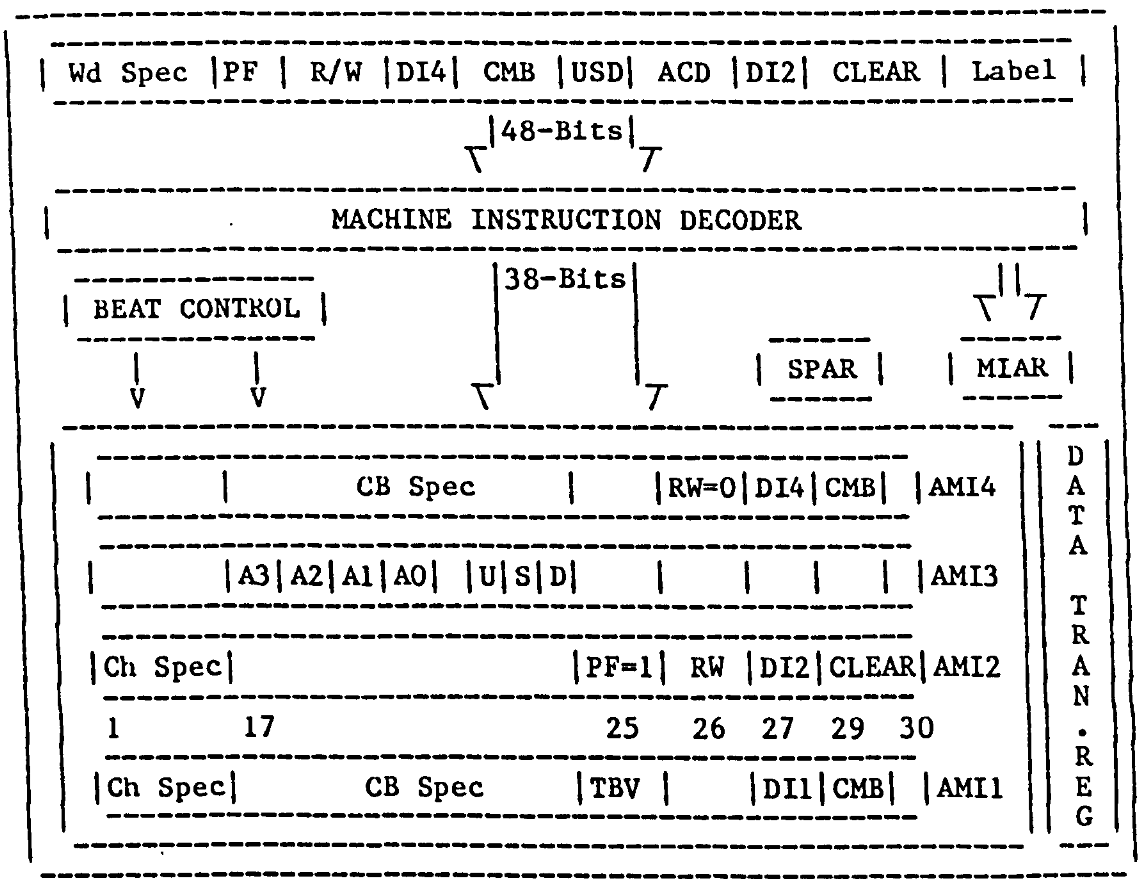
In beat-4 of Group-Run <AMI234>, only 4 bits of the 48-bit instruction format remain unused. This has significantly reduced the scope of activity to the Control-Bit field only.

```
1             25: 26   27    29  30  33    37          41     48
| Wd Spec | 0 |' R/W |DI2| CMB |USD| ACD |      ?      | Label |
    24       PF    1     2    1    3    4   CB Spec 4      8
```

Hence, only 4 bits of binary codes are available in this <AMI4> with no conditional masking or data complementing.

5.2.3 The Post-Function <AMI234>

In the Post-Function <AMI234>, the data transformations are always split into <CH Spec> and <CB Spec>, as a result, this characteristic is also reflected in the field allocation within the instruction format.



Although the execute sequence of the Post-Function <AMI234> is different from the Pre-Function <AMI234>: the READ/WRITE operation is executed before Beat 3 Tag Manipulation functions, its machine instruction format actually looks the same as the Pre-Function Non Group-Run <AMI234> format.

```
|<------------- The Post-Function <AMI234> ------------->|
1          25   26  27    29  30  33    37  39     41     48
-----------------------------------------------------------
| Wd Spec | 1 | R/W |DI4| CMB |USD| ACD |DI2| CLEAR | Label |
-----------------------------------------------------------
    24     PF   1    2    1    3    4    2     2       8
|<OPERAND>|<------------- OPCODE --------------->|- NEXT-|
                                                    INS.
                                                  ADDRESS
```

In the Post-Function <AMI234>, the selection bit (Bit-25) is set to 1 to indicate Post-Function ACC.

AMI2 : Clear and READ/WRITE Options :

The CLEAR options on the Control-Bit field are indicated in <AMI234> format in a two bit code (Bit-39 & Bit-40).

```
 1    17    25   26  27    29  30  33    37  39      41     48
 --------------------------------------------------------------
| ? | CB | 1 |  ?  |DI4|  ?  |USD| ACD | ? |    ?    | Label |
 --------------------------------------------------------------
 Ch   8    PF  R/W   2   CMB   3    4   DI2  CLEAR      8
```

Bit-39	Bit-40	CLEAR Options
0	0	No Clear
0	1	CLBTT
1	0	CLBCT
1	1	CLAB

Associated with the CLEAR operations is the READ/WRITE function on the Character field of all activated word-rows.

```
 1    17    25   26  27    29  30  33    37  39      41     48
 --------------------------------------------------------------
| ? | CB | 1 |  ?  |DI4|  ?  |USD| ACD | ? | CLEAR | Label |
 --------------------------------------------------------------
 Ch   8    PF  R/W   2   CMB   3    4   DI2    2        8
```

1) The Post-Function <AMI2> Character Spec. :

The <AMI2> Character Spec. uses three kinds of addressing schemes similar to <AMI1> Word Spec., except that the READ/WRITE operation on Control-Bit field is not effected until Beat 4.

A) The Immediate Addressing Mode

```
 7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
 -----------------------------------------------
| B | B | B | B | B | B | B | B |   |   |   |   |
 -----------------------------------------------
|<------ Character Field ------>|<- C.B. Field->|
|<---- 8-Bit Character Code --->|<- C.B. Field->|
|MSB|<--- 7-Bit ASCII Code ---->|<- C.B. Field->|
```

B) The Scratch-Pad Addressing Mode

```
 7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
  2   4   6   8   10  12  14  16  18  20  22  24
 -----------------------------------------------
|?|?|1|1|1|0|?|?|?|?|?|?|?|?|?|?|   |   |   |   |
 -----------------------------------------------
|<------ Addressing Mode ------>|<- C.B. Field->|
|<M>|< >|< >|<--- SP Address--->|
  S   |   |
  B   |    --> Addressing Mode
      |
       --> Non-Immediate Mode
```

C) The Input Buffer Addressing

```
 7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
  2   4   6   8   10  12  14  16  18  20  22  24
 -----------------------------------------------
|?|?|1|1|0|1|?|?|?|?|?|?|?|?|?|?|   |   |   |   |
 -----------------------------------------------
|<------ Addressing Mode ------>|<- C.B. Field->|
|<M>|< >|< >|<--- IB Address--->|
  S   |   |
  B   |    -->Addressing Mode
      |
       --> Non-Immediate Mode
```

D) The Output Buffer Addressing

```
 7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
  2   4   6   8   10  12  14  16  18  20  22  24
 -----------------------------------------------
|?|?|1|1|0|0|?|?|?|?|?|?|?|?|?|?|   |   |   |   |
 -----------------------------------------------
|<------ Addressing Mode ------>|<- C.B. Field->|
|<M>|< >|< >|<--- OB Address--->|
  S   |   |
  B   |    --> Addressing Mode
      |
       --> Non-Immediate Mode
```

2) The Post-Function <AMI2> Conditional Data Masking :

The Conditional Data Masking of Post-Function <AMI2> uses Bit-37 and Bit-38 for Data Identity selection (DI2).

```
1     17    25    26   27     29   30   33    37   39      41      48
-----------------------------------------------------------------------
| Ch | CB | 1 | R/W |DI4| CMB |USD| ACD | ? | CLEAR | Label |
-----------------------------------------------------------------------
  16    8    PF    1     2     ?     3     4    DI2     2        8
```

Bit-37	Bit-38	Data Identity <DI2>
0	0	<DI2> = X
0	1	<DI2> = 0
1	0	<DI2> = 1
1	1	Not defined

3) The READ/WRITE Function :

The READ/WRITE Function is indicated in Bit-26

A) The Selection of READ Function

```
 1    17    25    26  27     29  30  33     37  39      41     48
--------------------------------------------------------------------
| Ch | CB | 1 |   1   |DI4| CMB |USD| ACD |DI2| CLEAR | Label |
--------------------------------------------------------------------
  16    8   PF   R/W   2     1    3    4    2      2        8
```

B) The Selection of WRITE Function

```
 1    17    25    26  27     29  30  33     37  39      41     48
--------------------------------------------------------------------
| Ch | CB | 1 |   0   |DI4| CMB |USD| ACD |DI2| CLEAR | Label |
--------------------------------------------------------------------
  16    8   PF   R/W   2     1    3    4    2      2        8
```

AMI3 : Tag Manipulations:

The Post-Function <AMI234> also uses the same Tag Manipulations codes as the Pre-Function Non Group-Run <AMI234>.

```
 1    17    25    26  27     29  30  33     37  39      41     48
--------------------------------------------------------------------
| Ch | CB | 1 | R/W |DI4| CMB | ? |  ?  |DI2| CLEAR | Label |
--------------------------------------------------------------------
  16    8   PF    1    2     1   USD  ACD   2      2        8
```

Bit-33	Bit-34	Bit-35	Bit-36	Tag Manipulations
0	0	0	0	No Operation
0	0	0	1	PTT
0	0	1	0	PCT
0	0	1	1	RSTTU
0	1	0	0	RSTTD
0	1	0	1	RSCTU
0	1	1	0	RSCTD
0	1	1	1	EIR
1	0	0	0	MOR

Bit-30	Bit-31	Bit-32	Activation Direction
0	0	0	
0	0	1	D
0	1	0	S
0	1	1	SD
1	0	0	U
1	0	1	U D
1	1	0	US
1	1	1	USD

<u>AMI4 : Update Operation:</u>

The Post-Function <AMI4> has a very restricted WRITE function which operates on Control-Bits only, since 37 bits out of the 48-bit AMI format have been used during Beat-2 and Beat-3.

1	17	25	26	27	29	30	33	37	39	41	48
Ch	?	1	R/W	?	?	USD	ACD	DI2	CLEAR	Label	
16	8	PF	R/W	DI4	CMB	3	4	2	2	8	

1) The Post-Function <AMI4> Control Bit Spec. :

The Post-Function <AMI4> Control-Bit Spec. has only one addressing scheme: Immediate Addressing.

```
 7   6   5   4   3   2   1   0  CB1 CB2 CB3 CB4
-----------------------------------------------
|   |   |   |   |   |   |   |   | B | B | B | B |
-----------------------------------------------
|<------ Character Field ------>|<- C.B. Field->|
|MSB|<--- 7-Bit ASCII Code ---->|<- C.B. Field->|
```

2) The Post-Function <AMI4> Bit Select Functions :

The Post-Function <AMI4> uses both Conditional Data Masking and Data Complementing.

```
1     17    25   26  27    29  30  33    37  39     41     48
-------------------------------------------------------------
| Ch | CB | 0 | R/W | ? |  ?  |USD| ACD |DI2| CLEAR | Label |
-------------------------------------------------------------
  16    8   PF    1   DI4  CMB  3    4    2      2       8
```

A) Conditional Data Masking :

Bit-27	Bit-28	Data Identity <DI4>
0	0	<DI4> = X
0	1	<DI4> = 0
1	0	<DI4> = 1
1	1	Not defined

B) The Data Complementing :

a) The selection of true data content of IDR.

```
1     17    25   26  27    29  30  33    37  39     41     48
-------------------------------------------------------------
| Ch | CB | 1 | R/W |DI4|  0  |USD| ACD |DI2| CLEAR | Label |
-------------------------------------------------------------
  16    8   PF    1    2   CMB  3    4    2      2       8
```

b) The selection of complemented data content of IDR.

```
1     17    25   26  27    29  30  33    37  39     41     48
-------------------------------------------------------------
| Ch | CB | 1 | R/W |DI4|  1  |USD| ACD |DI2| CLEAR | Label |
-------------------------------------------------------------
  16    8   PF    1    2   CMB  3    4    2      2       8
```

5.3 SUMMARY

Being a mirror instruction of its API counterpart, the instruction format of AMI is predetermined by the two-part instruction structure of the Associative Computation Cycle. However, in the design of the AMI format, freedom of movement is still possible in the following areas:

1) The Instruction Length:

A 48-bit machine instruction format has been adopted as the shortest possible instruction length that is an integral multiple of the 8-bit data bus.

2) The Instruction Address Fields:

In this 48-bit format, the instruction address field is constrained to a 8-bit relative address (-128 to +127).

	<AMI1>	<AMI234>	
Field allocation		(Non Group-Run) & Post-Function	(Group-Run)
1 - 16	Ch Spec	Ch Spec	Ch Spec2
17 - 24	CB Spec	CB Spec	CB Spec2
25	TBV	PF	PF
26		R/W	R/W
27 - 28	DI1	DI4	DI2
29	CMB	CMB	CMB
30 - 32	000 (For AMI1)	DIRN (USD)	DIRN (USD)
33 - 36	Label-0[1 - 4]	ACD (<=1000)	ACD (>=1001)
37 - 38	Label-0[5 - 6]	DI2	CB Spec4 (1-2)
39 - 40	Label-0[7 - 8]	CL	CB Spec4 (3-4)
41 - 48	Label-1[1 - 8]	Label-X[1 - 8]	Label-X[1 - 8]

The translation of API is performed by the AAL Assembler, which as a result, will generate a file of 48-bit long AMIs as an object program to be run on the Distributed Computer System. The loading of AMI file is done by the API Loader (Fig. 3.4) into the API Program Store, ready to be fetched for execution. In the process of fetching, either <AMI1> or <AMI234> parts of the AMI will then be loaded into the BOAP for instruction decoding, which in turn, will be broken into a 4-beat sequence to drive the microprogrammed associative processor. The detailed simulation of BOAP is presented in the Appendix D. However, the following table shows the detailed breakdown of field allocations for all three kinds of Associative

Computation Cycles, in terms of beat sequence : Beat-1 for AMI1, Beat-2 for AMI2, so on and so for.

	ASSOCIATIVE COMPUTATION CYCLE											
	PRE-FUNCTION								POST-FUNCTION			
	NGRN				GRN							
	1	2	3	4	1	2	3	4	1	2	3	4
1.	CH1x			CH1x	CH1x	CH1x			CH1x	CH1x		
2.	CH1y			CH1y	CH1y	CH1y			CH1y	CH1y		
3.	CH2x			CH2x	CH2x	CH2x			CH2x	CH2x		
4.	CH2y			CH2y	CH2y	CH2y			CH2y	CH2y		
5.	CH3x			CH3x	CH3x	CH3x			CH3x	CH3x		
6.	CH3y			CH3y	CH3y	CH3y			CH3y	CH3y		
7.	CH4x			CH4x	CH4x	CH4x			CH4x	CH4x		
8.	CH4y			CH4y	CH4y	CH4y			CH4y	CH4y		
9.	CH5x			CH5x	CH5x	CH5x			CH5x	CH5x		
10.	CH5y			CH5y	CH5y	CH5y			CH5y	CH5y		
11.	CH6x			CH6x	CH6x	CH6x			CH6x	CH6x		
12.	CH6y			CH6y	CH6y	CH6y			CH6y	CH6y		
13.	CH7x			CH7x	CH7x	CH7x			CH7x	CH7x		
14.	CH7y			CH7y	CH7y	CH7y			CH7y	CH7y		
15.	CH8x			CH8x	CH8x	CH8x			CH8x	CH8x		
16.	CH8y			CH8y	CH8y	CH8y			CH8y	CH8y		
17.	CB1x		A3	CB1x	CB1x	CB1x	A3		CB1x		A3	CB1x
18.	CB1y		A2	CB1y	CB1y	CB1y	A2		CB1y		A2	CB1y
19.	CB2x		A1	CB2x	CB2x	CB2x	A1		CB2x		A1	CB2x
20.	CB2y		A0	CB2y	CB2y	CB2y	A0		CB2y		A0	CB2y
21.	CB3x			CB3x	CB3x	CB3x			CB3x			CB3x
22.	CB3y		U	CB3y	CB3y	CB3y	U		CB3y		U	CB3y
23.	CB4x		S	CB4x	CB4x	CB4x	S		CB4x		S	CB4x
24.	CB4y		D	CB4y	CB4y	CB4y	D		CB4y		D	CB4y
25.	TBV	PF=0			TBV	PF=0			TBV	PF=1		
26.				RW		(GR)		RW		RW		(RW=0)
27.	DI1x	DI2x		DI4x	DI1x	DI2x		CB1	DI1x	DI2x		DI4x
28.	DI1y	DI2y		DI4y	DI1y	DI2y		CB2	DI1y	DI2y		DI4y
29.	CMB	CLx		CMB	CMB	CMB		CB3	CMB	CLx		CMB
30.		CLy						CB4		CLy		

CHAPTER SIX

THE STRING PROCESSING ALGORITHMS

Like any model, a computer program is an abstraction from reality, from the relevant qualities and properties of the phenomenon being modelled. The computerization of problem solving usually involves three stages of abstractions:

1) Abstraction from the original problem to a design specification feasible of being implemented on the computer.

2) Abstraction from a design specification to a program written in a particular programming language in terms of a collection of data objects, operations, and representation schemes. Two kinds of abstractions useful during the construction of programs are procedural and data abstractions.

A) The Procedure Abstraction :

Procedure abstraction is better known as a subroutine or a function, and has been used in computer programming for a long time. The purpose of procedure abstraction is to permit the use of operations (algorithms) without specifying the details of implementation. What this involves is distinguishing between the use and implementation of objects in programming, which are referred to as the specification and the implementation phases. Over the years, most programmers realize that the degree of complexity which the human mind can cope with, at any one time, is considerably less than that embodied in much of the software that one might wish to build[63]. One way to overcome this limitation is by means of top-down design[64], which is known to some people as "structured programming"[65] or "modularity"[66,67], that organizes computer programs in hierarchical structured procedures regardless of their detailed implementations in the first instance, and then builds them up in the later stage by stepwise refinement in the direction of instructions and predicates available in the programming language[68].

B) The Data Abstraction :

The top-down design approach for the abstraction of operations achieved by procedure abstraction, can be extended to cover the structuring of data. The purpose of data abstraction is to permit the use of data objects without specifying the detailed structure of the data, and again it can be organized in two phases: the specification and the implementation phases. Normally, programs operate on data structures, which are aggregates of information with important structural relationships. These data structures might be a vector, a matrix, a list, a tree, a graph, or almost any structure that can be built upon on the primitive data types and other existing user-defined data structures.

3) Abstraction from the computer program to the underlying hardwares that support it. This is the abstraction of hardware onto the virtual machine of a particular language: implemenation of instructions and data types of that language which could be executed by the hardware.

Generally, it is these three levels of abstractions that constitute the life cycle of a program construction. In the development of programs by stepwise refinement[68], the programmer is encouraged to postpone the decision on data representation until after he has designed his algorithm and has expressed it as an "abstract" program operating on "abstract" data. He then chooses for the abstract data in some convenient and efficient concrete representation in the store of a computer; and finally programs the primitive operations required by his abstract program designed during the specification phase in terms of this concrete representation. In other words, the success of an algorithm depends almost always on the choice of a suitable data representation in the light of the ease in which this representation allows the necessary operations to be expressed.

However, a commom difficulty in program design lies in the unfortunate fact that at the stage where decisions about data representations have to be made, it often is still difficult to foresee the details of the necessary instructions operating on the data, and often quite impossible to estimate the advantages of one possible representation over another, due to the shortage of powerful built-in data types in the language for complex modelling. This situation has become even more horrifying in the case of artificial intelligence where the data structures involved are so complex and the size of their knowledge bases are so large that a combinational explosion has resulted. Hence, it is in the light of bridging this gap between built-in data types and user-defined data structures that has motivated us in a search for the means to support powerful and well established data structures such as string, list, tree, set etc., at the programming language level thus enhancing the existing primitive data types.

Before coming to the definition of data types, it is perhaps necessary to discuss the meaning of the abstraction in computer science. It has been used in at least two ways which are distinct but related[69]:

1) The Abstract Model :

This is the meaning common to most of science: "abstraction" covers the creation of a model, usually a mathematical model, to describe certain behaviour or characteristics of a object, as opposed to the real object as a whole.

2) The Abstract Machine :

The second meaning is closely related to the first, but projected onto the computer science perspective. It refers to process of generalizing, so that certain detailed features can be ignored at the higher levels. There are many examples in computer science, in particular, finite state machine models of hardware, procedure and data abstractions of computer programs.

In programming--especially in high-level programming languages--the concept of a data type is referred to as:

1) abstraction of data representations from hardware storage

2) operations applicable to objects in the data abstraction

However, it is the semantics of these operations as the definition of the data type that is of greatest importance. Cliff Jones has gone even further in saying "data types are characterized by their operations alone"[70], as what one wants to do with data types is to manipulate them, and the essential information about the operations is their inter-relationships. In other words, the data type itself is like a black box and its representation or implementation is of no concern to its user.

The difference between a data structure, which is an interconnection of the various data elements, and a data type, which separates the specification and the implementation of a data structure, is a matter of the organization of the contents of a data structure. This point can also be understood in terms of the external and internal behaviour of a data structure. In a data structure, there is no concept of the black box, every part is visible to all users who could write their own software to manipulate any part of the data structure. On the other hand, users never have any direct access to the implementation part of the data type. Instead, they are forced to access them indirectly through a set of predefined procedures. There are two kinds of data types: the built-in data type which the language supports as a primitive, or a user-defined data type which is sometimes referred to as an abstract data type[71]. Nevertheless, only recent languages, such as CLU and ADA , have provided facilities to define and enforce the implementation of abstract data types.

Physically, the ultimate components in the construction of data objects are bits. Higher level data structures are then constructed using bits as basic building blocks, eventually to be mapped upon the memory structure of the machine. This is the process referred to as finite mapping[72], which in the mathematician's sense is a function defined within a finite range of arguments of type A which maps each argument onto a value from type B.

$$A \dashrightarrow B$$

Conceptually, the finite mapping is similar to the symbol table in a compiler, but instead of mapping identifiers onto its decode (type, address, etc.), finite mapping maps defined ranges of data structures onto the hardware memory structure. Theoretically, a one bit memory can be mapped onto a Boolean type data, but in order to represent a greater variety of data structures, a larger collection of bits is necessary, and the first meaningful structure one can build is to collect 8 bits to form a byte. Using bytes one can now represent character structures in the memory hardware, and then a string of characters. It is from here as a starting point that, we will begin the investigation of the construction of data structures on the Byte-Organized Associative Processor.

The idea of "pattern" type structure for strings integrated with a powerful pattern matching system originated from the string processing language SNOBOL4. Recognition of this fact has led us to abstract most of the SNOBOL4 constructs into nine string algorithms for investgation into efficient implementation of string structures on BOAP. However, we shall only be concerned with the underlying concepts common to such system rather than remaining completely faithful to a particular language. Our approach in this chapter is to develop only the specification of string patterns and their use in structuring collections of strings. Nonetheless, the detailed implementation phase of these algorithms is presented separately in Appendix E.

A character string, or string for short, is a sequence of zero or more characters which can be mapped directly on a byte organized memory structure. The string algorithms mainly involve a SEARCH plus some kind of structuring operations[72]:

1) Sequence :

The first and simplest structuring method is called the sequence. A sequence consists of zero or more components of data, arranged in some meaningful order. In mathematical notation a sequence is frequently denoted by an asterisk, as follows

$$\text{STRING} = \text{Character}^{*},$$

A sequence corresponds to the iterative program structure in procedure abstraction, using the WHILE statement:

```
WHILE condition true DO loop
```

The computation of this structure consists of a sequence of zero or more computations of the program component loop; the sequence is not bounded in advance, but its length on any given occasion must be finite.

2) Discriminated Union :

The next simple structuring method is the discriminated union, which specifies that a choice is to be made from a selection of alternative structures. In the simplest case, the alternatives are just indicators of some condition such as SEARCH

```
SEARCH(object) = SUCCESS | FAILURE
```

This states that the SEARCH operation will proceed with either the sequence SUCCESS or the sequence FAILURE pending the outcome of SEARCH on those data structures. A more complicated example is as follows

```
ANY(PATTERN_1 | PATTERN_2 | PATTERN_3) = SUCCESS | FAILURE
```

This discriminated union differs from the mathematical union of sets, and is closely analogous to the conditional or case construction of program control structure.

```
IF condition true THEN statement_1 ELSE statement_2

CASE identifier OF
  option_1 : function_1;
  option_2 : function_2;
           :
           :
END; {CASE}
```

3) Direct Product :

The third major data structing methods is known by mathematicians as direct or Cartesian product, which involves a compound operation. For example, REPLACE operation can be defined as

$$\text{REPLACE} = \text{DELETE} \times \text{INSERT}$$

By this definition, it is stated that each operation of the type REPLACE is a structure with exactly two components, a DELETE operation followed by a INSERT operation. The close analogy of the direct product in program control structures is program by composition (the compound statement). For example a procedure which composed of a number of disjoint statements.

A formal string data organization on BOAP is shown in Fig. 6.1, which included three different fields:

1) The Identifier of the String

A string identifier is the label used by progrmmers to locate the string of characters, it uses a label name terminated by a $ sign.

2) The Value of the String

The second field contains the value of the string which is terminated by a # sign.

3) The Link_Name of the String

The Link_Name field is used to signify the continuation of the string which is due to the REPLACE or CONCATENATE operation. A Link_Name usually has a value of 0 (or [10000000] which means end of this string), otherwise, it provides the linking identifier for the next part of the string. For example, the String_Name_1 which has a value of 'abc' is to be linked up (or concatinated) with the other string 'de' which has the Link_Name of [10000001] as its identifier. In other words, String_Name_1 actually has a value of 'abcde'.

```
          [ 1 | 0 0 0 0 0 0 1 ]
            ^             ^
            |             |
    Marker set to 1   The actual Link_Name
    to mean this is
    a Link_Name
```

Since only 7 bits are used by the ASCII code in the character field, the Most Significant Bit (MSB) is used, in this case, as a marker to distinguish Link_Names from the other string names.

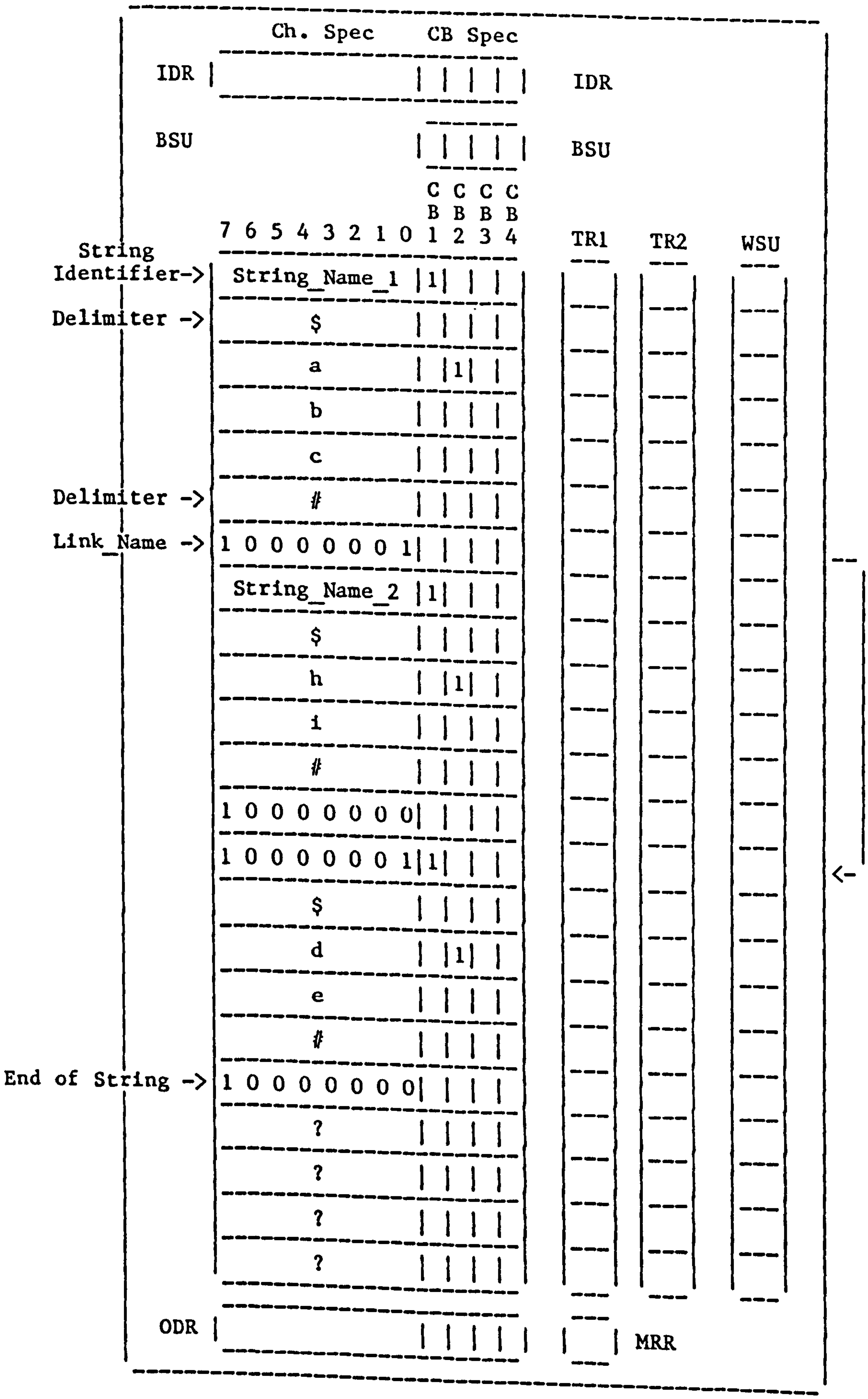
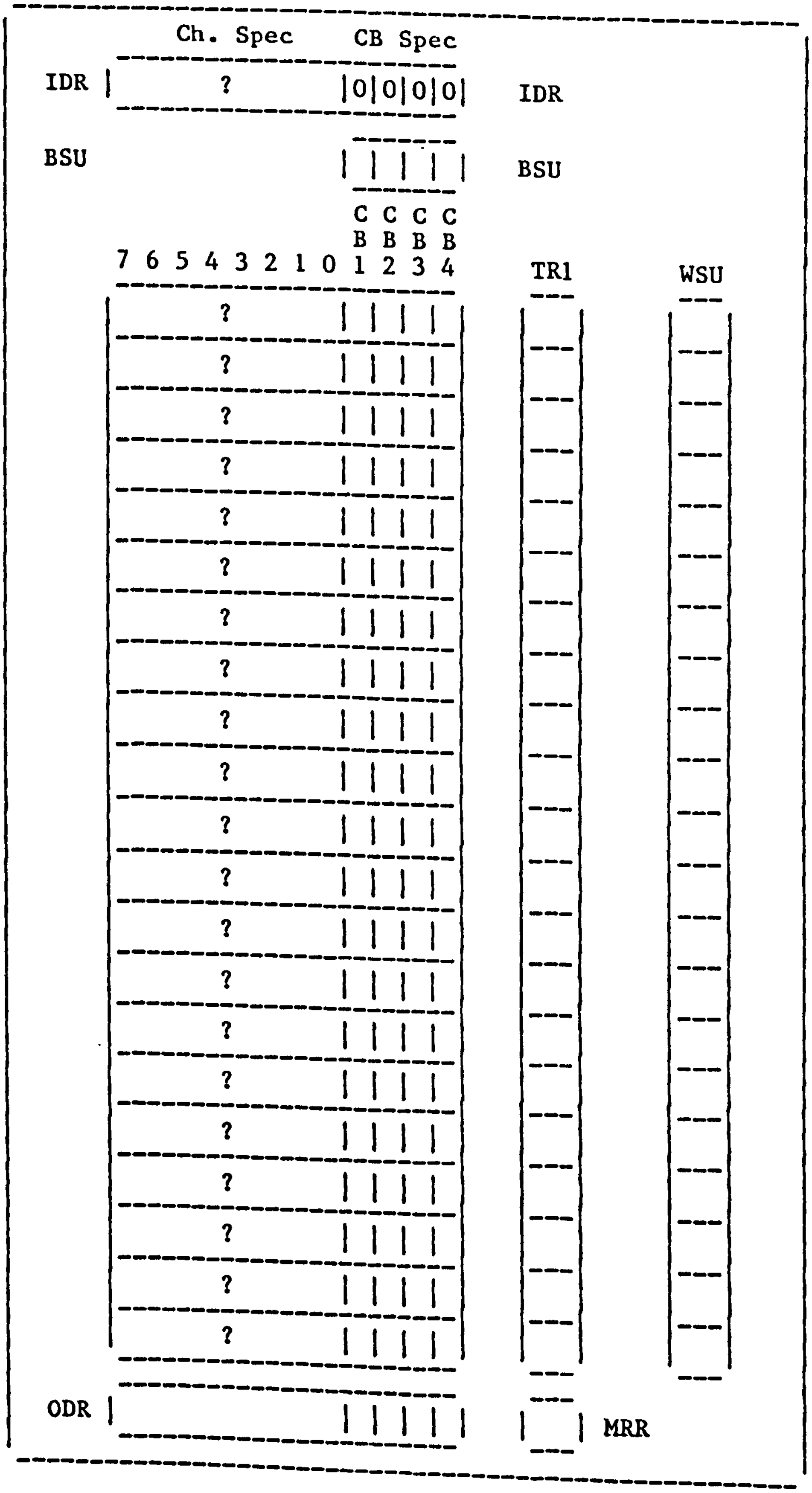


Fig. 6.1 The Organization of String Data on BOAP

The Associative Memory Array must be initialized before any processing can actually take place. It is done by being filling in with '?' at every word-row.



6.1 THE ALGORITHM : ASSIGN

The ASSIGN algorithm uses the Cartesian product for its data structuring method.

ASSIGN = String_Identifier x String_Value x Link_Name

By this definition, it states that each content of the type ASSIGN is a structure with exactly three components: a String_Identifier, a String_Value and a Link_Name. However, syntactically, it has the form

```
Variable := Value;

Stringl := 'abc';
```

The assignment statement may be said to have the following meaning: "Let Variable have the given Value". In the later section, we will generalize this assignment statement to include expression such as concatenation.

```
Variable := Expression;

Variable := Valuel + Value2  (Concatenation);

Stringl := String2 + String3;
```

```
         Ch. Spec        CB Spec
IDR |1 0 0 0 0 0 0 0|X|X|X|X|     IDR

BSU                 | | | | |     BSU

                     C C C C
                     B B B B
     7 6 5 4 3 2 1 0 1 2 3 4      TR1        WSU
    |       S       |1| | | |    |   |      |   |
    |       t       | | | | |    |   |      |   |
    |       r       | | | | |    |   |      |   |
    |       i       | | | | |    |   |      |   |
    |       n       | | | | |    |   |      |   |
    |       g       | | | | |    |   |      |   |
    |       l       | | | | |    |   |      |   |
    |       $       | | | | |    |   |      |   |
    |       a       | | | | |    |   |      |   |
    |       b       | | | | |    |   |      |   |
    |       c       | | | | |    |   |      |   |
    |       #       | | | | |    |   |      |   |
    |1 0 0 0 0 0 0 0| | | | |    | 1 |      | * |
    |       ?       | | | | |    | 1 |      |   |
    |       ?       | | | | |    | 1 |      |   |
    |       ?       | | | | |    | 1 |      |   |
    |       ?       | | | | |    | 1 |      |   |
    |       ?       | | | | |    | 1 |      |   |
    |       ?       | | | | |    | 1 |      |   |
    |       ?       | | | | |    | 1 |      |   |
    |       ?       | | | | |    | 1 |      |   |
    |       ?       | | | | |    | 1 |      |   |
    |       ?       | | | | |    | 1 |      |   |
    |       ?       | | | | |    | 1 |      |   |

ODR |               | | | | |    | 1 | MRR
```

6.2 SEARCH (SUCCESS AND FAILURE) :

The string processing algorithms are characterized by searching through strings of characters followed by state transformation on the chosen string. Therefore, the whole of string processing is centered round the SEARCH operations. The notions of SEARCH (success or failure) are shown as follows:

```
IF 'Pattern' IN Variable
  THEN Function_1
  ELSE Function_2;
```

To illustrate this, let us consider to search through a string for a pattern 'err'.

```
Stringl := 'ferry'

IF 'err' IN Stringl
  THEN Function_1
  ELSE Function_2;
```

In this example, the program control sequence will go to execute Function_1, as the condition which it is searching for is satisfied. The SEARCH algorithm forms the precondition of "Discrimination Union" in which a choice is to be made from a selection of alternative structures pending the outcome of some condition. Frequently, it is necessary to know whether a pattern matches with its origin at the first character of the reference string, if so, it is known to be Anchored at the beginning of the reference string. Sometimes, the outcome of the pattern matching will be quite different subject to different settings of Anchored Mode. Anchored Mode defines the marker within the reference string, and all subsequent processings will start from the marker onward to the first character (or the last character) of the reference string. If Anchor is not set, then the default setting is the first character of the reference string. However, the Anchored Mode will be discussed in more detail in Section 6.7.

	Ch. Spec 7 6 5 4 3 2 1 0	CB Spec CB1	 CB2	 CB3	 CB4		TR1	TR2	WSU
IDR		X	X	1	X	IDR			
BSU						BSU			
	S	1							
	t								
	r								
	i								
	n								
	g								
	l								
	$								
	f		1						
	e				1				
	r								
	r						1		*
	y			1					
	#								
	1 0 0 0 0 0 0 0								
	?								
	?								
	?								
	?								
	?								
	?								
	?								
	?								
	?								
	?								
ODR							1	MRR	

6.3 THE ALGORITHM : REPLACE

The REPLACE algorithm is a string processing function that uses the Cartesian Product structuring method

REPLACE = DELETE x INSERT

By this definition, it means that every computation evoked by this string function REPLACE always consists of two disjoint parts; the function DELETE followed by function INSERT.

The string function REPLACE has the following syntax which means taking a substring out of the reference string, as determined by the pattern, and replacing it by the object_string.

```
REPLACE 'Subject_String' BY 'Object_String' IN Variable

Stringl := 'ferry';

REPLACE 'err' BY 'uzz' IN Stringl;
```

will cause the reference string(Stringl) to be scanned for the subject string ('err'), and replace it by the object string ('uzz').

```
----------------------------------------------------------------------------
|            Ch. Spec     CB Spec                                          |
|       ---------------------------                                        |
| IDR  |        ?         |X|1|0|0|   IDR                                  |
|       ---------------------------                                        |
|                          ---------                                       |
| BSU                     | | | | |   BSU                                  |
|                          ---------                                       |
|                          C C C C                                         |
|                          B B B B                                         |
|       7 6 5 4 3 2 1 0 1 2 3 4      TR1       TR2        WSU              |
|       --------------------------   ---       ---        ---              |
|      |         S        |1| | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         t        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         r        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         i        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         n        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         g        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         l        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         $        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         f        | |1| |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | |1| |  | |   |     | 1 |      | * |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | |1| |  | |   |     |   |      | * |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | |1| |  | | 1 |     |   |      | * |            |
|       --------------------------   ---       ---        ---              |
|      |         y        | | |1|  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         #        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |1 0 0 0 0 0 0 0| | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|      |         ?        | | | |  | |   |     |   |      |   |            |
|       --------------------------   ---       ---        ---              |
|       --------------------------   ---                                   |
| ODR  |                  | | | | |  | 1 |  MRR                            |
|       --------------------------   ---                                   |
----------------------------------------------------------------------------
```

	Ch. Spec	CB Spec						
IDR	z	X	0	X	X	IDR		
BSU						BSU		
	7 6 5 4 3 2 1 0	CB1	CB2	CB3	CB4	TR1	TR2	WSU
	S	1						
	t							
	r							
	i							
	n							
	g							
	l							
	$							
	f		1					
	u							
	z							
	z					1		*
	y			1				
	#							
	1 0 0 0 0 0 0 0							
	?							
	?							
	?							
	?							
	?							
	?							
	?							
	?							
	?							
ODR						1	MRR	

Furthermore, any string could be replaced in this way by any other string; without the pre-condition that both strings must have the same length. For instance, if we execute

REPLACE 'err' BY 'l' IN Stingl;

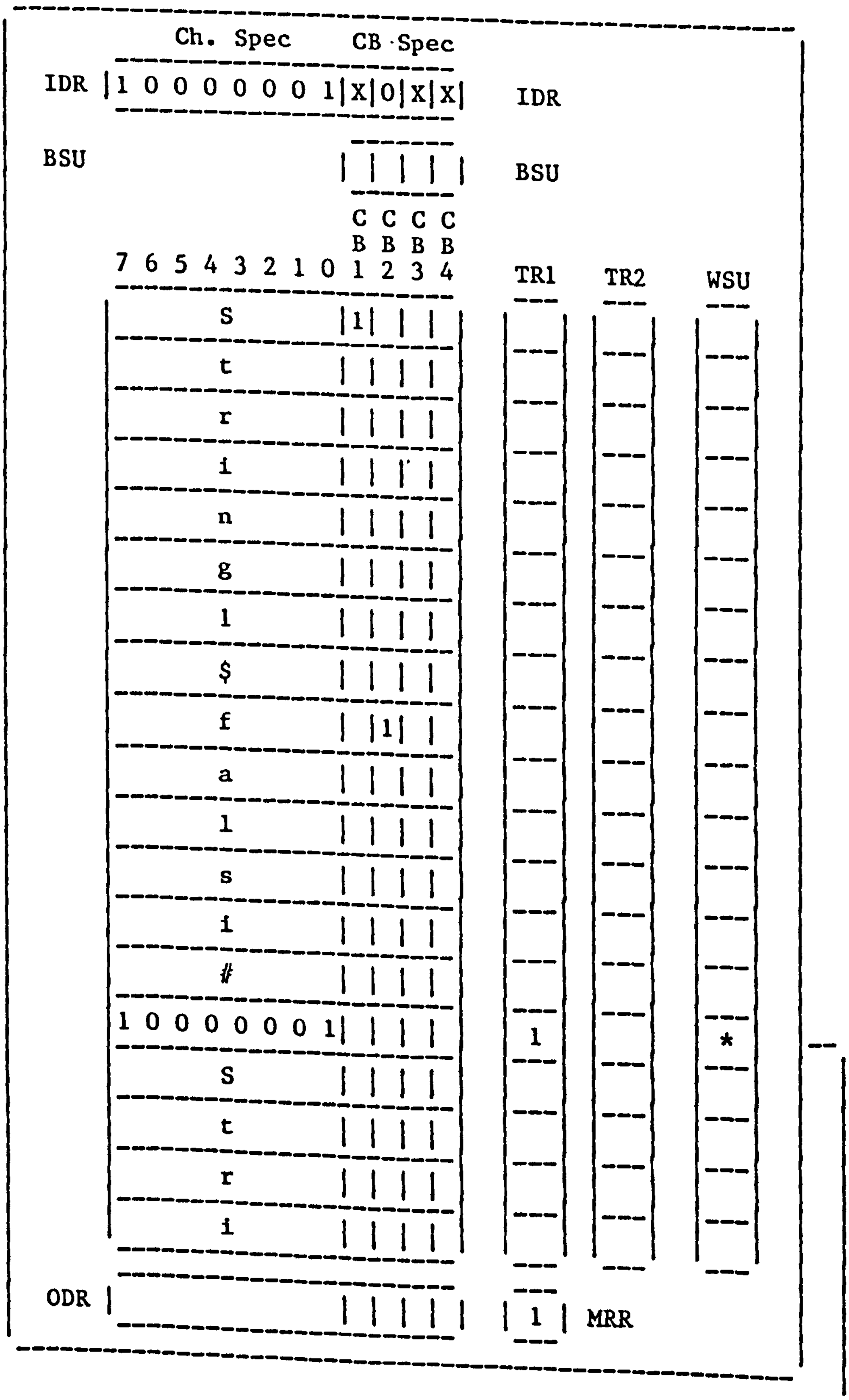
then the new value of Stringl will be 'fly'.

```
            Ch. Spec        CB Spec

IDR  |0 0 0 0 0 0 0 0|0|0|0|0|      IDR

BSU                  | | | | |      BSU

                      C C C C
                      B B B B
      7 6 5 4 3 2 1 0 1 2 3 4       TR1      TR2      WSU

     |      S        |1| | | |      |   |    |   |    |   |
     |      t        | | | | |      |   |    |   |    |   |
     |      r        | | | | |      |   |    |   |    |   |
     |      i        | | | | |      |   |    |   |    |   |
     |      n        | | | | |      |   |    |   |    |   |
     |      g        | | | | |      |   |    |   |    |   |
     |      l        | | | | |      |   |    |   |    |   |
     |      $        | | | | |      |   |    |   |    |   |
     |      f        | | |1| |      |   |    |   |    |   |
     |      l        | | | | |      |   |    |   |    |   |
     |      y        | | | | |      |   |    |   |    |   |
     |      #        | | | | |      |   |    |   |    |   |
     |1 0 0 0 0 0 0 0| | | | |      |   |    |   |    |   |
     |0 0 0 0 0 0 0 0| | | | |      | 1 |    |   |    | * |
     |0 0 0 0 0 0 0 0| | | | |      | 1 |    |   |    | * |
     |      S        |1| | | |      |   |    |   |    |   |
     |      t        | | | | |      |   |    |   |    |   |

ODR  |               | | | | |      | 1 |  MRR
```

But, if we execute

REPLACE 'err' BY 'alsit' IN Stringl;

then the new value of Stringl will be 'falsity'.



```
                 Ch. Spec      CB Spec
          ----------------------------
     IDR |1 0 0 0 0 0 0 0|0|0|0|0|        IDR
          ----------------------------

                              --------
     BSU                     | | | | |    BSU
                              --------
                              C C C C
                              B B B B
          7 6 5 4 3 2 1 0    1 2 3 4      TR1    TR2    WSU
          ----------------------------    ---    ---    ---
         |1 0 0 0 0 0 0 1|1| . | |  |    |   |  |   |  |   |   <-
          ----------------------------    ---    ---    ---
         |        $      | | | |    |    |   |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        t      | | | |    |    |   |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        y      | | | |    |    |   |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        #      | | | |    |    |   |  |   |  |   |
          ----------------------------    ---    ---    ---
End of String ->|1 0 0 0 0 0 0 0| | | |  | 1 |  |   |  | * |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
         |        ?      | | | |    |    | 1 |  |   |  |   |
          ----------------------------    ---    ---    ---
          ----------------------------    ---
     ODR |               | | | | |       | 1 | MRR
          ----------------------------    ---
```

Since the two component parts of the function REPLACE are disjoint, either one of these functions can be bypassed. For instance, using a null string as a subject_string will turn the function REPLACE into a DELETE function. Similarly, a null object_string will turn the function REPLACE into the function INSERT.

However, what happens if the pattern (subject_string) occurs more than once in the given reference string? Suppose that the value of String2 is 'I TOOK A VACATION WITH MY CAT', and we now execute

```
REPLACE 'CAT' WITH 'DOG' IN String2;
```

the new value of String2 will not be

'I TOOK A VACATION WITH MY DOG'

but rather

'I TOOK A VADOGION WITH MY DOG'

This is because all searching in BOAP is done quite differently from the conventional SISD processor; every word_row in AMA is searched in parallel for the pattern concerned, and all matched word_rows are then tagged and enabled for subsequent reading/writing, yield, 'VADOGION' and 'DOG'. For certain kinds of applications, this may be hazardous, but with a more carefully thought out algorithm one could easily get round this problem by resolving them into groups and then updating (or read) them one by one.

```
REPLACE 'CAT' BY 'DOG' WITH RESOLVE(LEFT)
```

will give us

```
'I TOOK A VACATION WITH MY DOG'
```

and by executing

```
REPLACE 'CAT' BY 'DOG' WITH RESOLVE(RIGHT)
```

will instead give us

```
'I TOOK A VADOGION WITH MY CAT'
```

```
REPLACE 'Subject_String' BY 'Object_String' WITH RESOLVE(Direction)
```

A "number factors" can also be included with Direction option, for Replace operation, to isolate a particular substring with reference from either the left or right hand end of the reference string,

```
REPLACE 'CAT' BY 'DOG' WITH RESOLVE(2LEFT)
```

will have the same effect as

```
REPLACE 'CAT' BY 'DOG' WITH RESOLVE(RIGHT)
```

since CAT is the second substring from the left that matched the pattern. In addition, perhaps a verification routine could also be included to interact with the user, of which those word_rows are to be updated (or read). However, this is already outside the scope of this chapter.

6.4 THE ALGORITHM : CONCATENATE

The algorithm Concatenate is the basic operation for combining two strings to form a third. In other words, it is an 'ADD' operator in the context of string processing. It uses the Cartesian Product to structure the data.

CONCATENATE = String1 x String2

This states that the type "Concatenate" is a structure with two components, and its associated function is to add two strings together. The following statement illustrates the format of an expression involving concatenation.

```
String3 := String1 + String2;
```

If (String1 = 'very ') and (String2 = 'good'), then the content of the concatenated string3 will be 'very good';

```
              Ch. Spec        CB Spec
         ---------------------------
IDR     |                  |X|X|X|0|      IDR
         ---------------------------

                            -------
BSU                        | | | | |      BSU
                            -------
                            C C C C
                            B B B B
         7 6 5 4 3 2 1 0    1 2 3 4       TR1      TR2      WSU
         -------------------------------  ---      ---      ---
        |        S         |1|  | | |    |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        t         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        r         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        i         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        n         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        g         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        l         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        $         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        v         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        e         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        r         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        y         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |                  | | | | |     | 1 |    |   |    | * |
         -------------------------------  ---      ---      ---
        |        #         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |1 0 0 0 0 0 0 0   | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        ?         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        ?         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        ?         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        ?         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        ?         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        ?         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        ?         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        ?         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---
        |        ?         | | | | |     |   |    |   |    |   |
         -------------------------------  ---      ---      ---

         -------------------------------  ---
ODR     |                  |X|X|X|0|     | 1 |  MRR
         -------------------------------  ---
```

```
 -----------------------------------------------------------------
|
|              Ch. Spec       CB Spec
|         -----------------------------
| IDR    |                   |X|X|X|X|     IDR
|         -----------------------------
|                             ---------
| BSU                        | | | | |     BSU
|                             ---------
|                             C C C C
|                             B B B B
|         7 6 5 4 3 2 1 0     1 2 3 4      TR1      TR2       WSU
|         -----------------------------    ---      ---       ---
|        |        S           |1| | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        t           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        r           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        i           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        n           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        g           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        2           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        $           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        g           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        o           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        o           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        d           | | | |  |  | 1 |    |   |     | * |
|         -----------------------------    ---      ---       ---
|        |        #           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |1 0 0 0 0 0 0 0     | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        ?           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        ?           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        ?           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        ?           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        ?           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        ?           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        ?           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        ?           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        ?           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|        |        ?           | | | |  |  |   |    |   |     |   |
|         -----------------------------    ---      ---       ---
|         -----------------------------    ---
| ODR    |        d          |X|X|X|0|    | 1 |  MRR
|         -----------------------------    ---
 -----------------------------------------------------------------
```

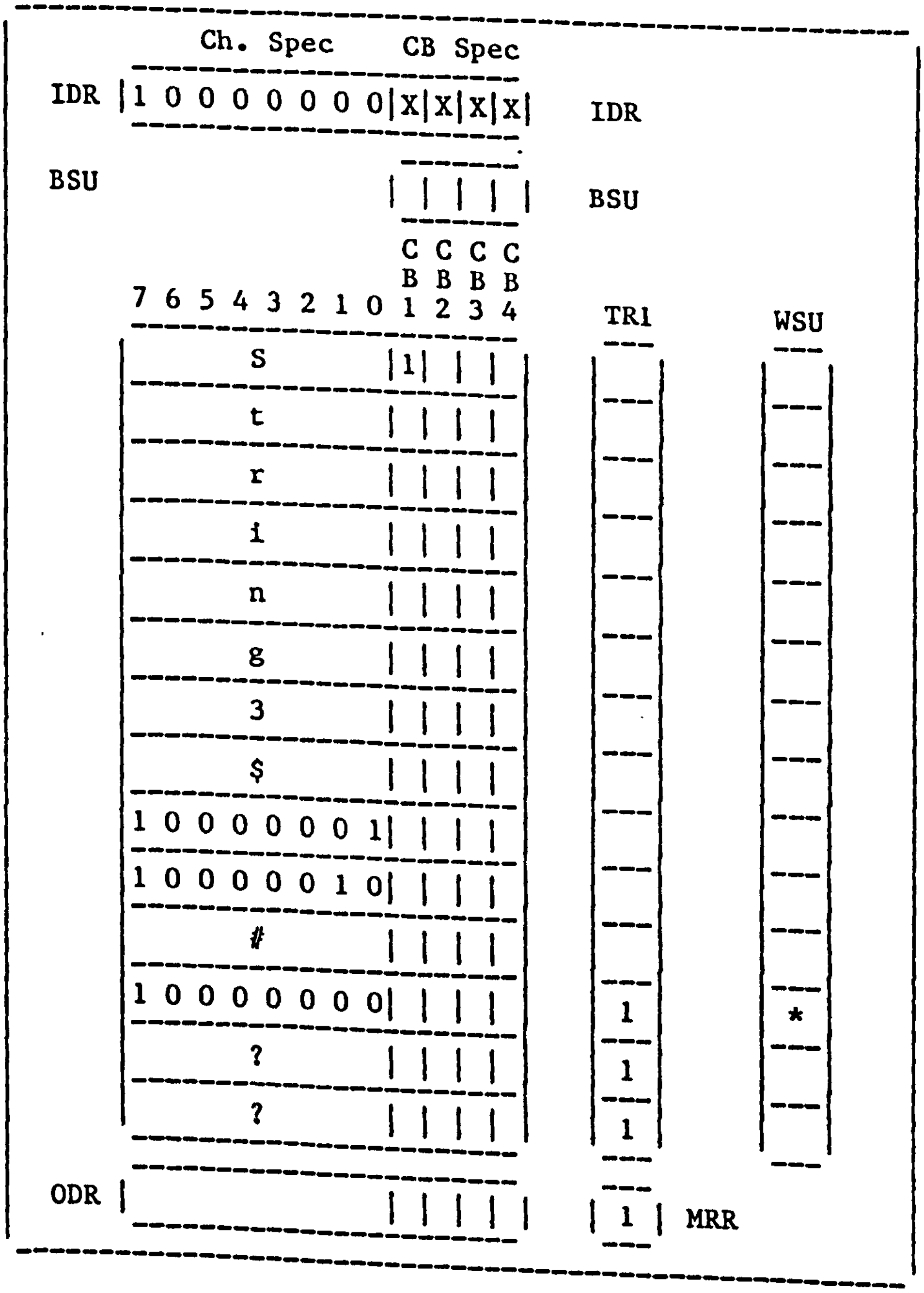
	Ch. Spec	CB Spec						
IDR	1 0 0 0 0 0 0 0	X	X	X	X	IDR		
BSU						BSU		
	7 6 5 4 3 2 1 0	CB1	CB2	CB3	CB4	TR1	TR2	WSU
	S	1						
	t							
	r							
	i							
	n							
	g							
	3							
	$							
	v							
	e							
	r							
	y							
	g							
	o							
	o							
	d							
	#							
	1 0 0 0 0 0 0 0					1		*
	?					1		
	?					1		
	?					1		
	?					1		
	?					1		
ODR						1	MRR	

6.5 THE ALGORITHM : UNION

The algorithm UNION is again a Cartesian Product that brings together a collection of two or more string components under one string identifier.

UNION = String1 x String2

It is the "union of sets" in the logical sense: a union of strings contains the values of all its member strings.



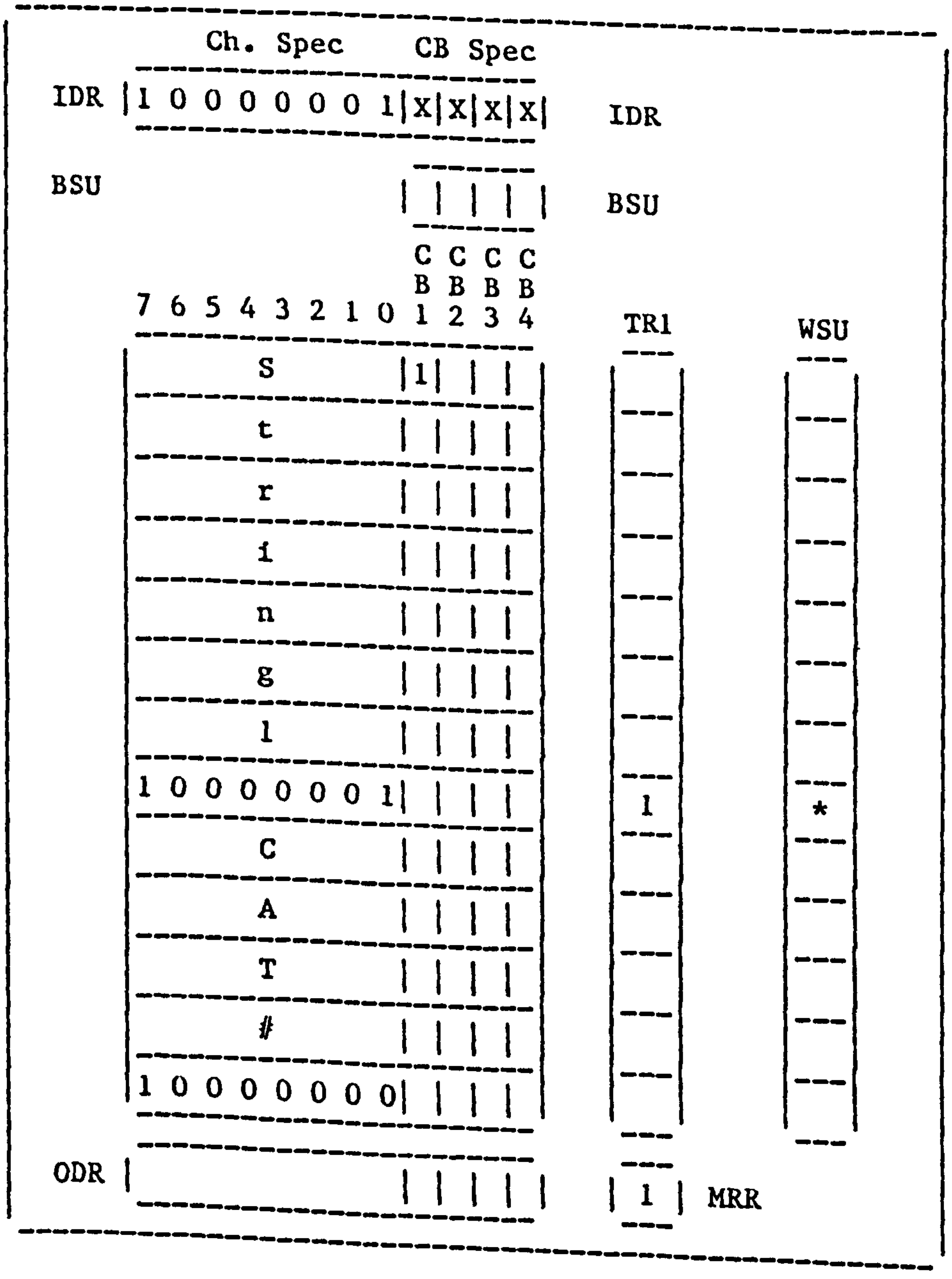
```
          Ch. Spec        CB Spec
IDR  |1 0 0 0 0 0 0 0|X|X|X|X|      IDR

BSU                  | | | | |      BSU

                      C C C C
                      B B B B
      7 6 5 4 3 2 1 0 1 2 3 4       TR1        WSU
     |       S       |1| | |       |   |      |   |
     |       t       | | | |       |   |      |   |
     |       r       | | | |       |   |      |   |
     |       i       | | | |       |   |      |   |
     |       n       | | | |       |   |      |   |
     |       g       | | | |       |   |      |   |
     |       3       | | | |       |   |      |   |
     |       $       | | | |       |   |      |   |
     |1 0 0 0 0 0 0 1| | | |       |   |      |   |
     |1 0 0 0 0 0 1 0| | | |       |   |      |   |
     |       #       | | | |       |   |      |   |
     |1 0 0 0 0 0 0 0| | | |       | 1 |      | * |
     |       ?       | | | |       | 1 |      |   |
     |       ?       | | | |       | 1 |      |   |

ODR  |               | | | | |     | 1 | MRR
```

In general, UNION statements can be expressed two ways.

1) UNION by Name :

String3 := String1 | String2;

By this definition, it causes the building of a new structure String3 and assigns to it the values of both member strings, String1 and String2. Since the assignment of String3 is carried out by means of the strings' identifiers rather than the strings' values, it is referred to as "UNION by Name".

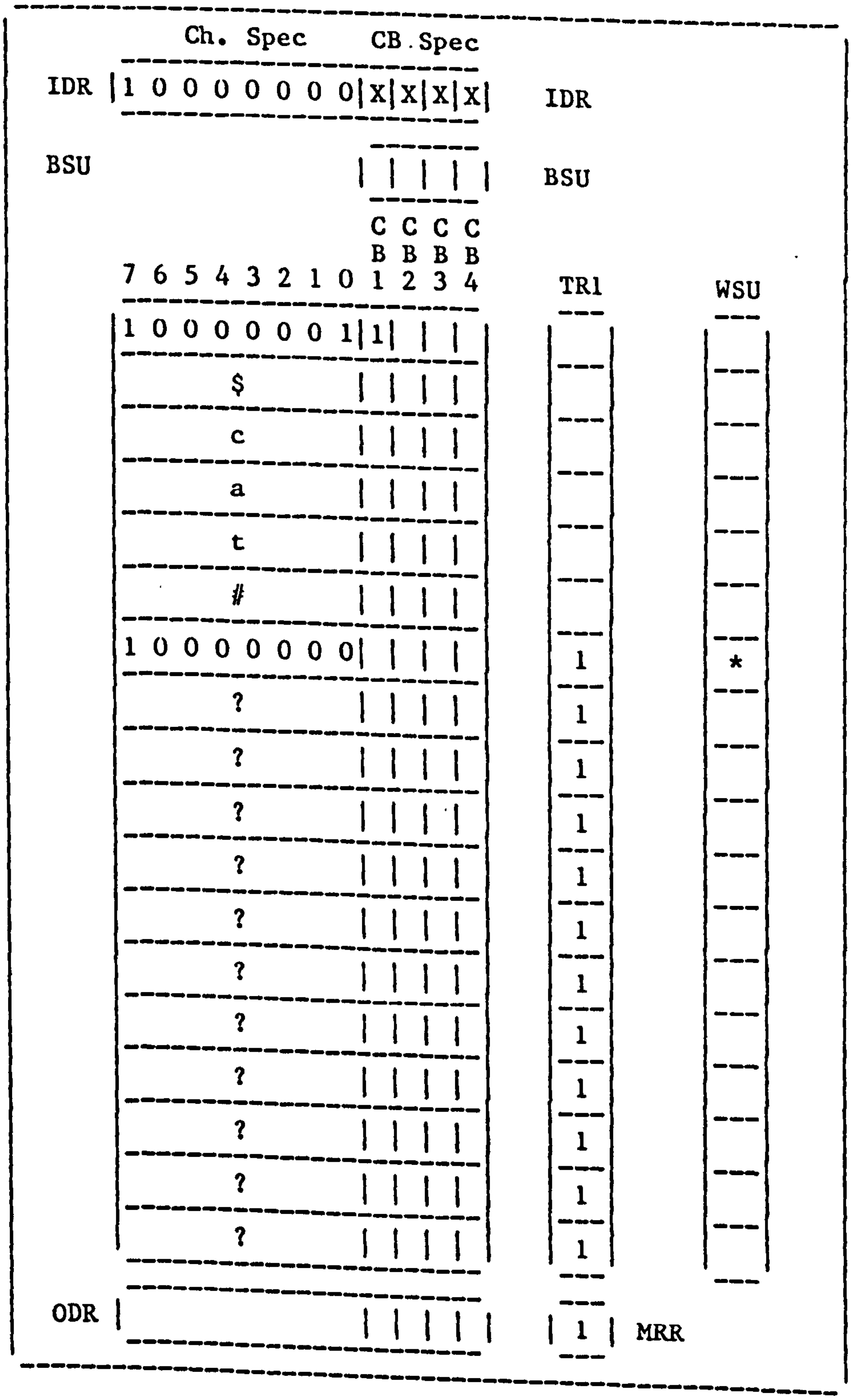


```
-----------------------------------------------------------------
|          Ch. Spec      CB Spec                                 |
|       ------------------------                                 |
| IDR  |1 0 0 0 0 0 1 0|X|X|X|X|   IDR                           |
|       ------------------------                                 |
|                         -------                                |
| BSU                    | | | | |   BSU                         |
|                         -------                                |
|                        C C C C                                 |
|                        B B B B                                 |
|       7 6 5 4 3 2 1 0  1 2 3 4     TR1        WSU              |
|       ------------------------     ---        ---              |
|      |       S         |1| | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       t         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       r         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       i         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       n         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       g         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       2         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |1 0 0 0 0 0 1 0| | | |   | 1 |      | * |             |
|       ------------------------     ---        ---              |
|      |       D         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       O         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       G         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       #         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |1 0 0 0 0 0 0 0| | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       ?         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       ?         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       ?         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       ?         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       ?         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       ?         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       ?         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       ?         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       ?         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       ?         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|      |       ?         | | | |   |   |      |   |             |
|       ------------------------     ---        ---              |
|       ------------------------     ---                         |
| ODR  |                | | | | |   | 1 | MRR                     |
|       ------------------------     ---                         |
-----------------------------------------------------------------
```

2) UNION by Value :

In this case, the values are used as the union of String3.

String3 := 'cat' | 'dog';



```
--------------------------------------------------------------------
|          Ch. Spec      CB Spec                                   |
|       ---------------------------                                |
| IDR  |1 0 0 0 0 0 0 0|X|X|X|X|     IDR                           |
|       ---------------------------                                |
|                        ---------                                 |
| BSU                   | | | | |    BSU                           |
|                        ---------                                 |
|                        C C C C                                   |
|                        B B B B                                   |
|       7 6 5 4 3 2 1 0  1 2 3 4      TR1          WSU             |
|       --------------------------    ---          ---             |
|      |1 0 0 0 0 0 1 0|1| | |   |   |   |        |   |            |
|       --------------------------    ---          ---             |
|      |       $        | | | |  |   |   |        |   |            |
|       --------------------------    ---          ---             |
|      |       d        | | | |  |   |   |        |   |            |
|       --------------------------    ---          ---             |
|      |       o        | | | |  |   |   |        |   |            |
|       --------------------------    ---          ---             |
|      |       g        | | | |  |   |   |        |   |            |
|       --------------------------    ---          ---             |
|      |       #        | | | |  |   |   |        |   |            |
|       --------------------------    ---          ---             |
|      |1 0 0 0 0 0 0 0| | | |   |   | 1 |        | * |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|      |       ?        | | | |  |   | 1 |        |   |            |
|       --------------------------    ---          ---             |
|       --------------------------    ---                          |
| ODR  |                | | | | |   | 1 | MRR                      |
|       --------------------------    ---                          |
--------------------------------------------------------------------
```

6.6 THE ALGORITHM : ANY & NOTANY

The algorithm ANY & NOTANY are organized in a "Discriminated Union" structure, which specifies a choice to be made from a selection of alternative structures such as a UNION structure.

ANY(UNION) = Success | Failure

Suppose that, instead of searching through the word 'ferry' to find if it contains the pattern 'err', we want to know of whether it contains any vowel. Obviously, we could do five separate searches using SEARCH statement; one each for a, for e, for i, for o and for u. However, this process would be inefficient, instead we can use the ANY statement as shown in the following

```
                    Stringl := 'ferry';

          IF ANY('a' | 'e' | 'i' | 'o' | 'u') IN Stringl
            THEN Function_1
            ELSE Function_2;
```

This states that "If Stringl contains any character of 'a'|'e'|'i'|'o'|'u', then execute Function_1 else Function_2."

ANY(string) and NOTANY(string) are primitive pattern matching functions whose arguments are pattern structures that match single characters or strings. ANY will match any character or string appearing in its argument, whereas, NOTANY will match any character or string not appearing in its argument.

ANY and NOTANY are the operations of the UNION string structure: with the UNION statement collecting the set of values of the alternative strings, the ANY or NOTANY statement takes these values to examine whether or not any of these values exist in the reference string concerned. The argument of ANY may be any string either in the form of a string value or string identifier, which is similar to the case of the UNION structure (section 6.5). Hence we will call the first 'ANY by Value', and the second 'ANY by Name'.

1) ANY by Value :

```
IF ANY('Patterns') IN String_Variable
  THEN Function_1
  ELSE Function_2;
```

```
IF NOTANY('Patterns') IN String_Variable
  THEN Function_1
  ELSE Function_2;
```

```
'Patterns' := 'Pattern_1' | 'Pattern_2' | 'Pattern_3';
```

ANY or NOTANY can be further extended to match a sequence of characters or strings instead of terminating the search operation after the first match.

```
'Patterns' := 'Pattern_1' -> 'Pattern_2 -> 'Pattern_3';
```

For instance, we would like to look for any strings that consists of 'f' 'e' and 'y' in that order, without worrying about other possible characters in between, we could write

```
ANY('f' -> 'e' -> 'y') IN String1
  THEN Function_1
  ELSE Function_2;
```

Now, if String1 were 'ferry', the flow of control would certainly switch to Function_1. Arguments of ANY and NOTANY must be non-null strings when pattern matching is performed.

1) ANY by Name :

Since the pattern structure for ANY('a'|'e'|'i'|'o'|'u') matches any vowel, and the pattern for NOTANY('a'|'e'|'i'|'o'|'u') matches any character that is not a vowel, it could equally be legitimate to use ANY(Vowel) or NOTANY(Vowel) to mean the same operation, if Vowel := 'a'|'e'|'i'|'o'|'u'.

```
Vowels := 'a' | 'e' | 'i' | 'o' | 'u';


         IF ANY(Vowels) IN Stringl
           THEN Function_1
           ELSE Function_2;
```

The syntax of ANY by Name is shown as follows

```
         IF ANY(Variables) IN String_Variable
           THEN Function_1
           ELSE Function_2;


         IF NOTANY(Variables) IN String_Variable
           THEN Function_1
           ELSE Function_2;
```

Bearing in mind that

```
<Variables> ::= <Stringl> | <String2>;

<Variables> ::= <Stringl> -> <String2>;
```

The ANY and NOTANY which we have just described is the superset of their counterparts in SNOBOL4; they are in fact, the combination of ANY, NOTANY, SPAN, BREAK, ARB and FENCE. This is due to the changes of hardware; as a matter of fact, we have found that it is very straightforward to remove the restrictions of the SNOBOL4 instructions mentioned above, and merge them together to form the ANY/NOTANY statements.

6.7 THE ALGORITHM : POSITION

The algorithm POSITION is a string function to position either the cursor or anchor within the reference string. Position, in this sense, is perhaps best thought of as occuring between the characters of a string. In the string 'ferry', "position 2" occurs between the e and the first r. If we are at position 2, then r is the next character.

```
f e r r y
   ^
   |
```

1) POSITION CURSOR :

The function POSITION (N) refers to "POSITION CURSOR AT N" within the reference string.

```
POSITION (Number) WITHIN Variable;
```

will cause a marker (cursor) to be set at the position specified by Number within the reference string (Variable). For readers who are familiar with a screen editor, the function POSITION is nothing more than a cursor positioning instruction that moves the cursor around the reference string. For the case of the string "ferry".

```
POSITION (2) WITHIN "ferry";
```

will set the cursor at position 2 with reference from the first character of the reference string.

```
f e r r y
   ^
   |
```

The Number here is a absolute number referred from the beginning of the reference string and has a default implication of "from left to right". This is sometimes referred to in SNOBOL4 as Anchored mode, which means anchored at the first character of the reference string for any subsequent pattern matching. However, the definition of Number can be extended to include Relative Number which takes reference from either the anchor or the cursor, set by the previous POSITION instruction, at any position within the reference string.

A) POSITION CURSOR With Reference from the Cursor

A pair of diamond brackets is used here to imply "with reference from the Cursor". If the Cursor is not preset by any previous POSITION instruction, then this <number> will be the same as the absolute number which refers from the beginning of the reference string.

```
        POSITION <Number> WITHIN Variable;
           POSITION <2> WITHIN "ferry";
```

Since this positioning operation is performed with reference from the previous setting of the cursor, it will therefore, cause the cursor to be positioned beyond the second r.

```
f e r r y
       ^
       |
```

B) POSITION CURSOR With Reference from the Anchor

A pair of square brackets, in this case, is used to mean refer from the Anchor position. If the Anchor is not preset by any previous POSITION instruction, then this <number> will take reference from the beginning of the reference string.

```
POSITION [Number] WITHIN Variable;
   POSITION [2] WITHIN "ferry";
```

In this case, since no Anchor has been set so far, the positioning will take reference from the first character of the reference string.

```
f e r r y
   ^
   |
```

Extension of the definition of POSITION is possible by diverging from conventional "from left to right" to "from right to left".

```
RPOSITION (Number) WITHIN Variable;
RPOSITION [Number] WITHIN Varaible;
RPOSITION <Number> WITHIN Variable;
.
.
.
```

Hence, the position in the reference string that is before the first character is POSITION(0), whereas, the rightmost position to the right of the last character is RPOSITION(0).

2) POSITION ANCHOR :

Apart from the positioning of the cursor within the reference string, the anchor can also be positioned in accordance to a similar set of POSITION instructions.

```
POSITION_ANCHOR (Number) WITHIN Variable;
POSITION_ANCHOR [Number] WITHIN Variable,
POSITION_ANCHOR <Number> WITHIN Variable;
RPOSITION_ANCHOR (Number) WITHIN Variable;
RPOSITION_ANCHOR [Number] WITHIN Variable;
RPOSITION_ANCHOR <Number> WITHIN Variable;
```

In the previous section, the content of substring was used as a means to locate a position within the reference string, whereby processing could take place. In this section, we will introduce another way of marking the beginning of substring (HEAD of block) and the end of substring (TAIL of block) by locating their positions within the reference string regardless of its content. The function POSITION_HEAD (N) refers to "position the beginning of substring at N" within the reference string, and POSITION_TAIL (N) refers to "position the end of substring at N within the reference string".

3) POSITION HEAD :

The function "POSITION_HEAD" will cause the "HEAD MARKER" to be placed at the position specified by Number within the reference string (Variable).

```
POSITION_HEAD (Number) WITHIN Variable;
POSITION_HEAD [Number] WITHIN Variable;
POSITION_HEAD <Number> WITHIN Variable;
RPOSITION_HEAD (Number) WITHIN Variable;
RPOSITION_HEAD [Number] WITHIN Variable;
RPOSITION_HEAD <Number> WITHIN Variable;
```

4) POSITION_TAIL :

Whereas the function "POSITION_TAIL" will cause the "TAIL MARKER" to be placed within reference string in according to Number.

```
POSITION_TAIL (Number) WITHIN Variable;
POSITION_TAIL [Number] WITHIN Variable;
POSITION_TAIL <Number> WITHIN Variable;
RPOSITION_TAIL (Number) WITHIN Variable;
RPOSITION_TAIL [Number] WITHIN Variable;
RPOSITION_TAIL <Number> WITHIN Variable;
```

6.8 THE ALGORITHM : REMAINDER

The algorithm REMAINDER is an assignment statement by which the remainder of the value of the reference string, which is marked by markers, is assigned to a string identifier.

1) REMAINDER_ANCHOR :

```
Variable := REMAINDER_ANCHOR,
```

will assign the value bounded by the Anchor and the end or beginning of the reference string, to the string identifier. The directional convention was preset by the POSITION statement. In other words, any remainder statement must be preceded by a POSITION statement. For instance,

```
         :
POSITION_ANCHOR (6) WITHIN "transfer";
   Temp_String := REMAINDER_ANCHOR;
         .
         ,
```

will assign "fer" as the value of Temp_String, whereas

```
RPOSITION_ANCHOR (3) WITHIN "ferry";
  Temp_String := REMAINDER_ANCHOR;
```

will assign "fer" as the value of Temp_String.

2) REMAINDER_CURSOR :

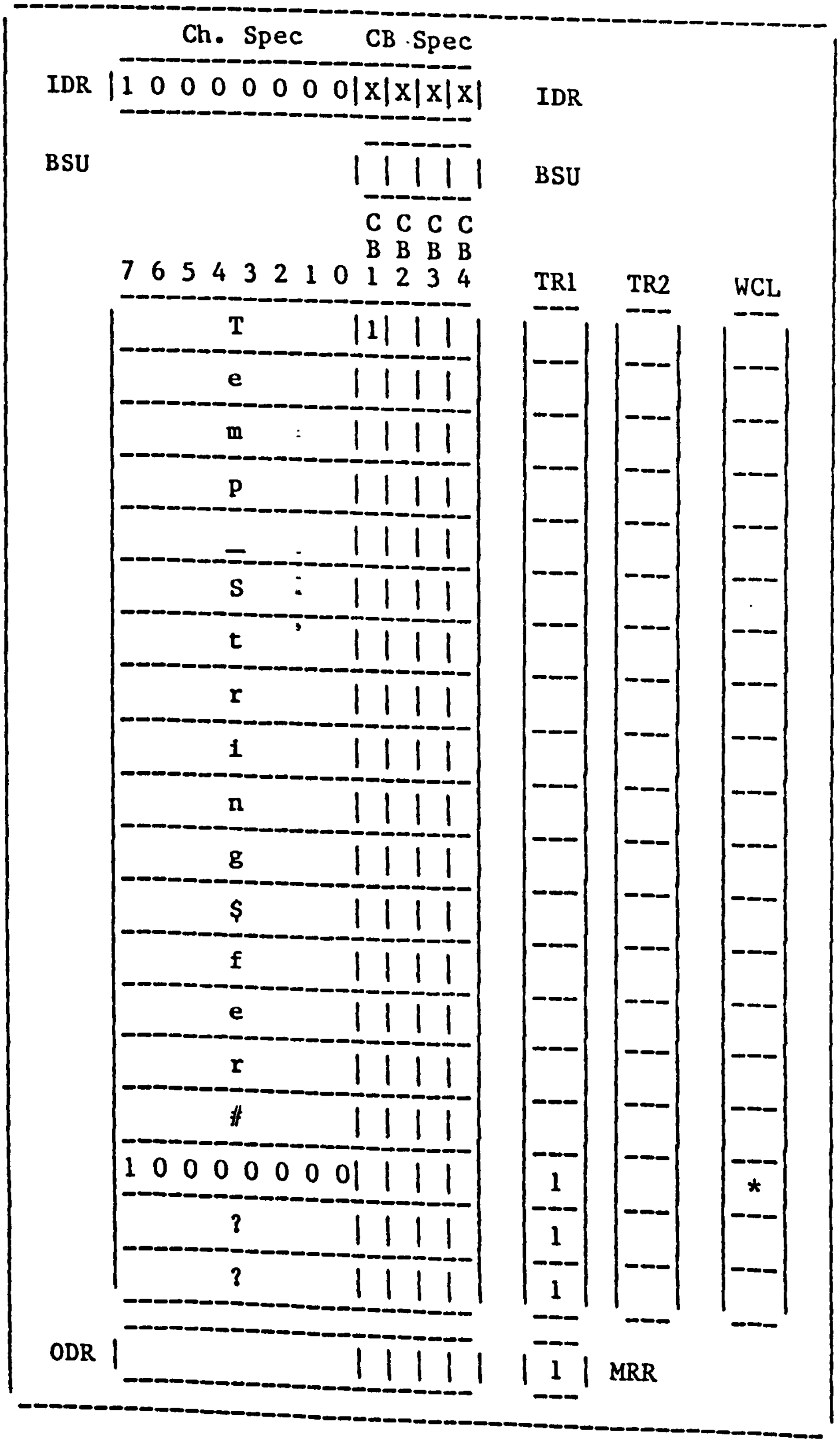
REMAINDER could also use the cursor as the marker, instead of the anchor.

```
Variable := REMAINDER_CURSOR;
```

3) REMAINDER_BLOCK :

For the case of the value bounded by the head and tail markers, within the reference string, it takes the following form:

Variable := REMAINDER_BLOCK;



6.9 THE ALGORITHM : LENGTH

The algorithm LENGTH is a string processing function, which has a single string argument and returns as value an integer that is the length of the string.

```
Example :   String := 'ferry';
            WRITELN('The number of characters is ',LENGTH(String1));
```

will print:

```
             The number of characters is 5
```

6.10 SUMMARY :

The evolution of programming languages has seen a steady development in the use of data types, various kinds of data structures are used in all areas of computer science. Compilers use stacks, symbol tables, and parse trees; operating systems maintain lists of processes and files, and employ memory management schemes that use lists or tables of available space; programs in artifical intelligence use stacks, queue sets, search trees, tables and graphs; and database systems use strings, lists, trees, rings and tables. As a matter of fact, part of the art of programming is the art of organizing data representations. In examining the specification of contemporary computer programs of substantial size, one often finds that they tend to contain layers of separate representations that span the gap from the naked machine upwards to the problem domain. For example, in a data base system, one may include name field, address field, room_number field and fields for internal/external phone numbers, at the problem domain level. However, at the intermediate levels one might use strings, tables, lists, trees, queues and other data types to support the problem domain, while at the lowest levels, one might find that these data structures are constructed from bits, bytes, and serially arranged sequences of machine words. Some authors[68,72] refer to the layers in such a representation cascade as levels of abstraction. This concept of mapping offers a very abstract way for the programmer to specify the system in the early stages of his design, without being confused by the details of the representation and implementaion of his data. However, all data structures or data types are ultimately mapped on to the physical organization of hardware storage such as byte. Higher level data objects are then constructed using the byte as a basic building block and the first meaningful structure, which one can build, is the ordered collection of bits to form a bit string.

Our purpose in this chapter is to deal in some detail, and in a fairly formal manner, with the semantics of the string patterns at the specification level and their use in structuring collections of strings. Orginally, the SNOBOL4 language integrated the idea of a "pattern" in string algorithms with a very powerful pattern matching system.

Recognition of this fact has led us to abstract most of the string constructs from SNOBOL4 for the investigation of the implementation of string algorithms on the Byte-Organized Associative Processor: examining the mapping of string objects onto the memory structure of BOAP, and investigating how string functions can be efficiently implemented on DCS. Hopefully, this detailed exploration of string structure implementation will provide us with the insight and confidence we need to do a superficial but convincing coverage of other data structures on DCS.

In the next chapter, we will expand from the basis of string algorithms to look into the structural organization of other data structures, and examine their mapping onto BOAP: a single linked list can be formed out of a string with a single pointer; similarly a binary tree can be formed from a linked list with double pointers...etc. However, due to the triviality and length that it may involve, the detailed implementation phase of their associated functions will not be discussed.

CHAPTER SEVEN

THE OTHER DATA STRUCTURES

In Chapter Six, only small and relatively simple structures (i.e Strings) have been dealt with. However, the String structure has its serious shortcoming of being bounded, and has a maximum allocation of whatever number of bytes the compiler designer may decide. Although an array structure can often be used to collect together a number of individual strings, the storage allocation still remains static at compile time, so much so, that this has become very inflexible if the program has to deal with dynamic data structures which may grow and shrink at run time. However, for a great many applications, a suitable compromise can be found by employing the so called "linked-memory philosophy"[73]:

> If there isn't room for the information here, let's put it somewhere else and plant a link to it.

As a result, since the late fifies, a great deal of effort has gone into the development of dynamic memory management. In a dynamic memory allocation environment a data structure is a block of information with one, two or more pointers by which next records can be found. However, we will start with the data structures with only one pointer.

```
 ------------
|            |
| Data Block |-------> Next Data Block
|            |
 ------------
```

1) Dynamic Memory Management using Indirect Addressing:

This is a very simple dynamic memory management system widely used by FORTRAN type programming languages for simulating pointer type records. It is basically a fixed size array of addresses pointing to the beginning of each record concerned, where a fixed length of address is used in the system, this is, in fact, a very convenient way of organizing dynamic data structures in those First Generation high-level programming languages such as FORTRAN. Nonetheless, the array still remains static.

```
                                             --------------------
                               ------->| Head of the Record |
                              |        --------------------
                              |        | : : : : : : : : : |
 ---------------              |        --------------------
| Record Address |------------      --->| Head of the Record |
 ---------------                  |    --------------------
| Record Address |----------------     | : : : : : : : : : |
 ---------------                       | : : : : : : : : : |
| : : : : : : :  |                     --------------------
| : : : : : : :  |           -------->| Head of the Record |
 ---------------             |         --------------------
| Record Address |-----------     --->| Head of the Record |
 ---------------                  |    --------------------
| Record Address |----------------     | : : : : : : : : : |
 ---------------                       | : : : : : : : : : |
                                       | : : : : : : : : : |
                                        --------------------
```

2) Dynamic Memory Management using a Linked-List:

The Linked-List approach is a more modern way of organizing dynamic memory which is supported by most of the Second Generation programming languages such as PASCAL, C, BCPL ... etc. A List is defined (recursively) as a finite sequence of zero or more atoms/Lists. Here, an "atom" is an undefined concept referring to elements from any universe of objects as may be desired, so long as it is possssible to distinguish an atom from a List. Nonetheless, in the implementation phase, Lists are organized as ordered collections of an arbitrary number of elements which can be accessed by pointers.

```
 ------------------             ------------------
| Data Block | PTR |---> : : : : | Data Block | PTR |---> NULL
 ------------------             ------------------
```

The Linked-List provides a much more flexible and efficient scheme organising dynamic memory management for data expansion and data manipulation : for example, INSERT and DELETE operations are simplified to just altering pointers.

7.1 THE SINGLY LINKED-LIST DATA STRUCTURES :

The Singly Linked-List is sometimes referred to as a linear list[73], or one-way linked list[74], which is a set of N nodes whose structural properties essentially involve only the linear (one-dimensional) relative positions of the nodes. The Singly Linked-List data structures, like all other linked-list data structures, use RECURSION for data structuring.

List = Atom | List x List;

In other words, a list is either an atom (defined elsewhere) or an ordered pair, whose first and second components are themselves lists. RECURSION is a structuring mechanism that can be used to defined aggregates whose size can grow arbitrarily and whose structure can have arbitrary complexity. As opposed to SEQUENCING, it allows the programmer to create arbitrary access paths for the selection of components. Data objects of recursive type are implemented by use of pointers. Each component specified as belonging to the recursive type is represented by a location containing a pointer to the data object, rather than the data object itself. In the case of a Singly Linked_List Data Structure, the formal list data organization on BOAP is shown in Fig. 7.1, which includes five different parts:

1) The Identifier of the List

Similar to string identifier, list identifier is the label used by programmers to locate the list by name.

2) The Link_Name of the List

But in the usual case, the Link_Name is used instead to chain from one list to the other.

3) The Data Objects of the List

The is the part of the list where the actual data can be found. In theory, data objects may be of any size.

4) The delimiter of Data Object Fields

The delimiter # is used in the list structure to terminate the data object fields.

5) The Pointer of the List

This is the part where the Link_Name of the next list is kept.

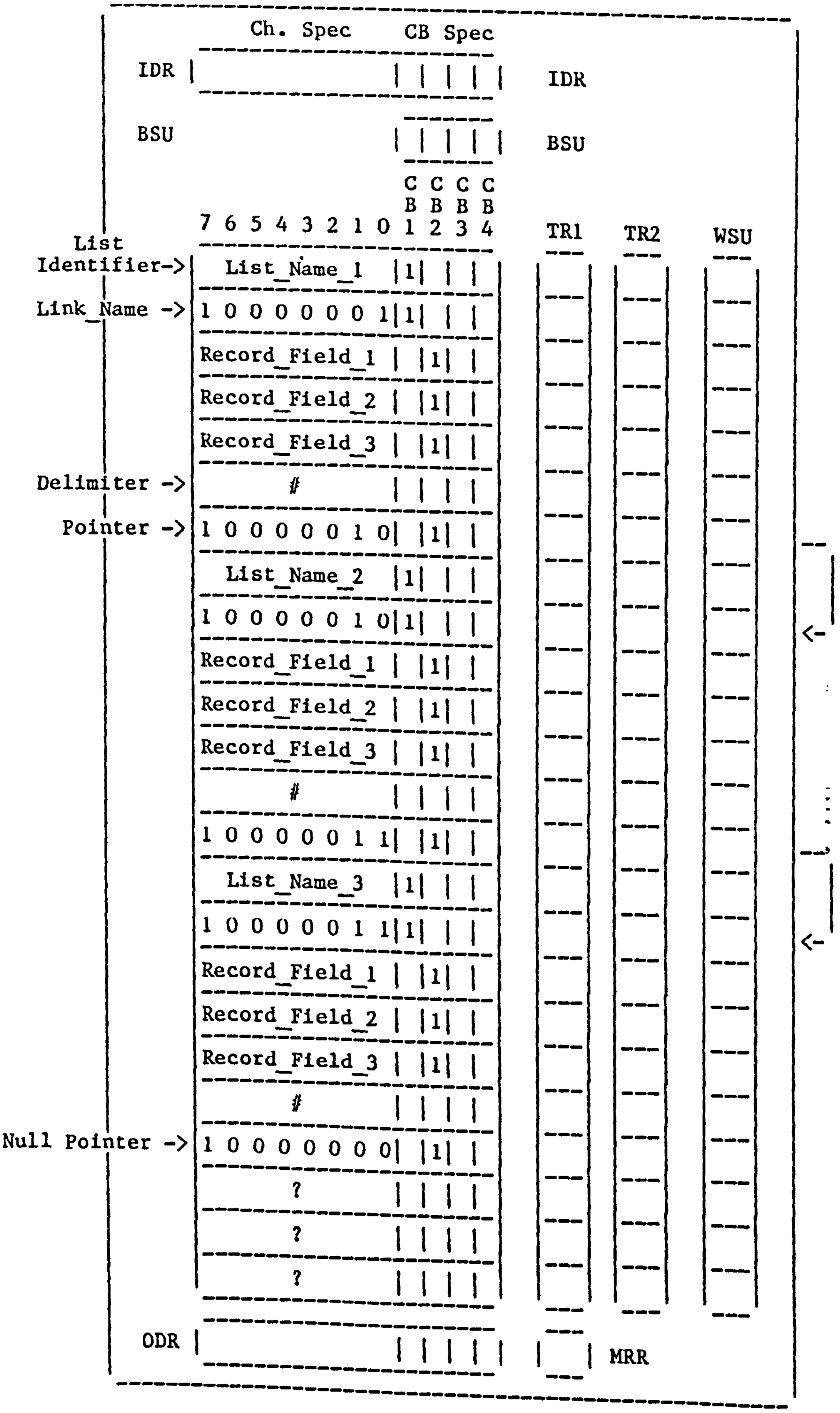


Fig. 7.1 The Data Organization of a Singly Linked-List Data Structure

The data organization of a Singly Linked-List only provides the declaration part of the data structure, and the operational part still has to be defined. The operations we might want to perform on a singly linked-list are as follows:

1) Gain access to the Kth node of the list to examine and/or to change the contents of its fields.

2) Insert a new node just before the Kth node.

3) Delete the Kth node.

4) Combine two or more Singly Linked-Lists into a single list.

5) Split a Singly Linked-List into two or more lists.

6) Make a copy of a Singly Linked-List.

7) Determine the number of nodes in a list.

8) Sort the nodes of the list into ascending/decending order based on certain fields of the nodes.

9) Search the list for the occurrence of a node with a particular value in some field.

A computer application rarely calls for all nine of the above operations in the full generality, therefore, we may distingish between types of Singly Linked-Lists depending on the principal operations to be performed.

7.1.1 The Stack Structure:

A Stack is a Singly Linked-List for which all insertions and deletions (and usually all accesses) are made at one end of the list.

```
 BOTTOM                                                     TOP
 --------     --------     --------     --------     --------
| Data |  |->| Data |  |->| Data |  |->| Data |  |->| Data |  |->
 --------     --------     --------     --------     --------
                                                        ^
                                                        |
                                                  Insert/Delete
```

Stacks arise quite frequently in practice, and people have given them a number of names: push-down list, Last-In-First-Out (LIFO) list, and even yo-yo list! Stacks are particularly useful for implementing nested structures, like procedure calls, Reverse Polish and recursive algorithms.

7.1.2 The Queue Structure:

A Queue is a Singly Linked-List for which all insertions are made at one end of the list; all deletions (and usually all accesses) are made at the other end.

```
 QUEUE END                                                 QUEUE FRONT
 --------     --------     --------     --------     --------
| Data |  |->| Data |  |->| Data |  |->| Data |  |->| Data |  |->
 --------     --------     --------     --------     --------
    ^                                                    ^
    |                                                    |
  Insert                                               Delete
```

Queues are sometimes called circular stores or First-In-First-Out (FIFO) lists. With the Queue structure, data blocks are entered at the end of the queue and are removed when they ultimately reach the front of the queue.

7.1.3 The Circular List Structure:

A circularly-linked list (Briefly : a Circular List) has the property that its last node links back to the first instead of to NULL. It is then possible to access all of the list starting at any given point for either insertion or deletion.

```
 -------------------------------------------------------------------
|    --------     --------     --------     --------     --------    |
 ->| Data |  |->| Data |  |->| Data |  |->| Data |  |->| Data |  |->
    --------     --------     --------     --------     --------
```

In view of the circular symmetry, there is no NULL link to signal the end of list. Consequently, a special and recognizable node is put into the list, as a convenient stopping place. This special node is called the list head, and in applications, it is often found that it is quite convenient to insist that every circular list have exactly one node which is its list head.

7.1.4 The Dynamic Array Structure:

In the absence of special information about the expansional patterns of array, the static array scheme has to define the maximum possible storage to cater for the worst case, which often means inefficient memory management. The Dynamic Array scheme which sometimes known as the extendible array[75], offers a better solution by defining a one dimensional array: an open ended array.

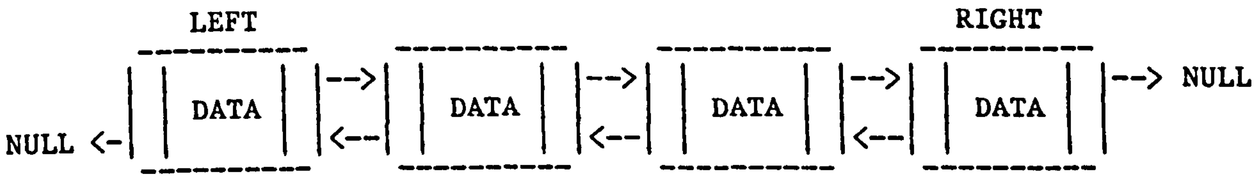
```
 --------      --------     --------     --------
| Base |  |--->| Data |  |->| Data |  |->| Data |  |-> NULL
 --------      --------     --------     --------
                                            ^
                                            |
                                          Insert
```

Usually, insertions are done in front of the list, but, other conventions could always be adopted. Deletions are done more efficiently than in the static arrray scheme by just manipulating pointers.

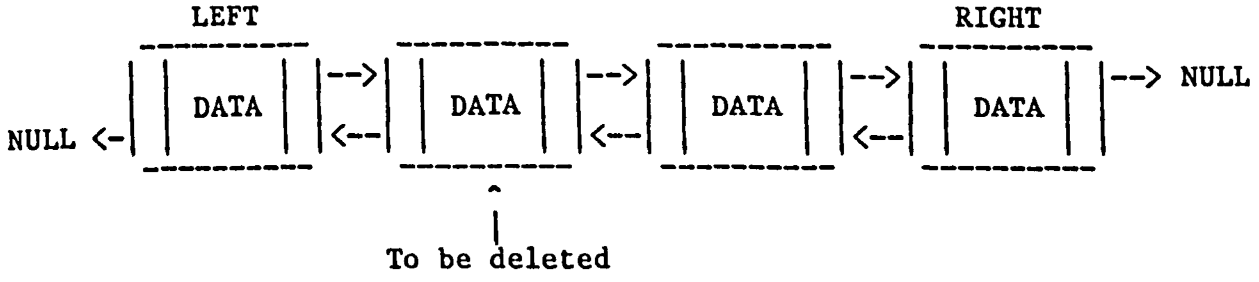
Apart from Stack, Queue, Circular List and Dynamic Array, there are many other ways in which operations on Singly-Linked Lists can be definded. Nevertheless, there is one common feature that each of them shares: all functions must only operate on the one-dimensional lists.

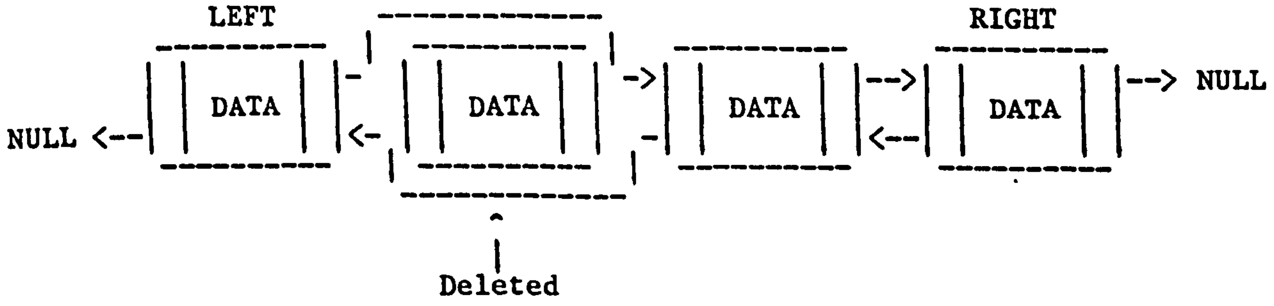
7.2 THE DOUBLY LINKED-LIST DATA STRUCTURES :

For even greater flexibility in the manipulation of Linked-Lists, we can include two links in each node, pointing to the items on either side of that node.



In the Doubly Linked-Lists, Manipulation of data items becomes much easier: in the Singly Linked-List, we cannot perform a deletion without knowing which node precedes it in the chain, since the preceding node needs to have its link altered when the unwanted node is deleted. However, in the Doubly Linked-List, data blocks are chained together with a two-way pointer, by which deletion or insertion can be done very easily.





The formal Doubly Linked-List data organization on BOAP is shown in Fig. 7.2, which is virtually the same as the Singly Linked-List data structure, except the inclusion of one extra pointer field.

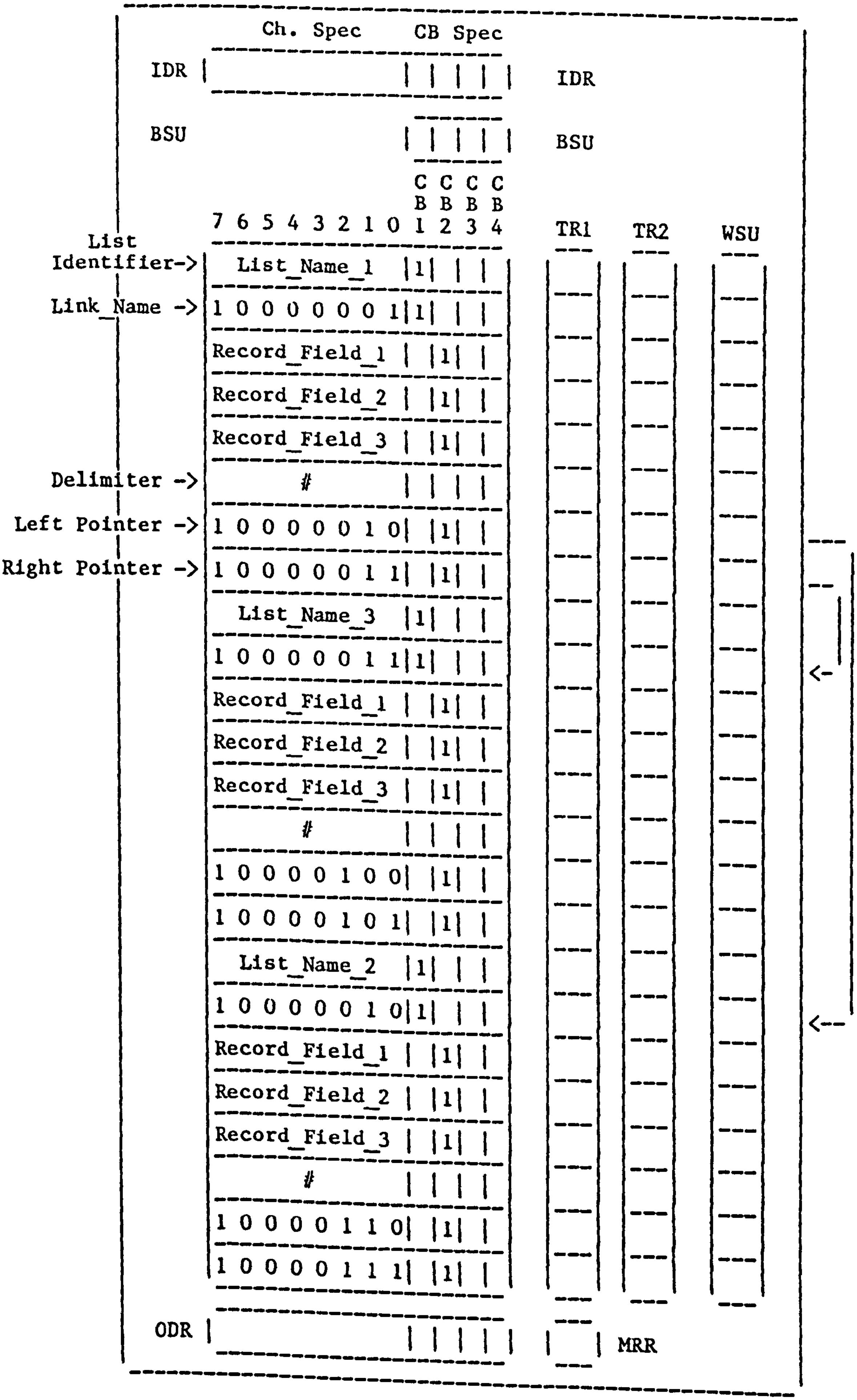
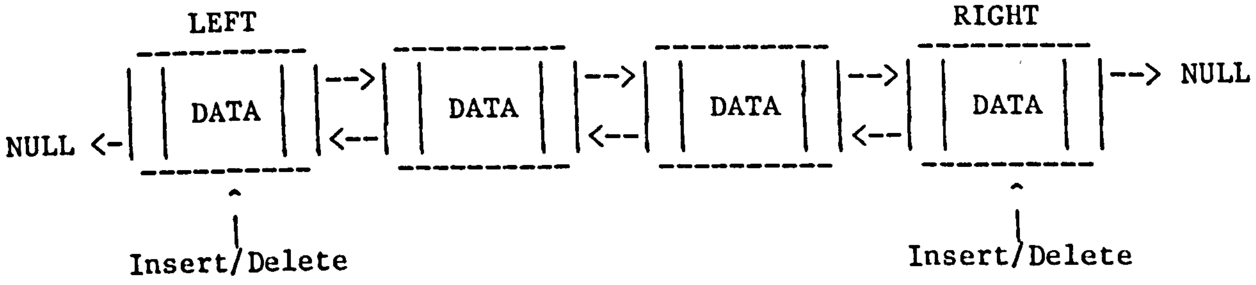


Fig. 7.2 The Data Organization of a Doubly Linked-List Data Structure

The operational parts of a Doubly Linked-List are shown as follows:

7.2.1 The Deque Structure:

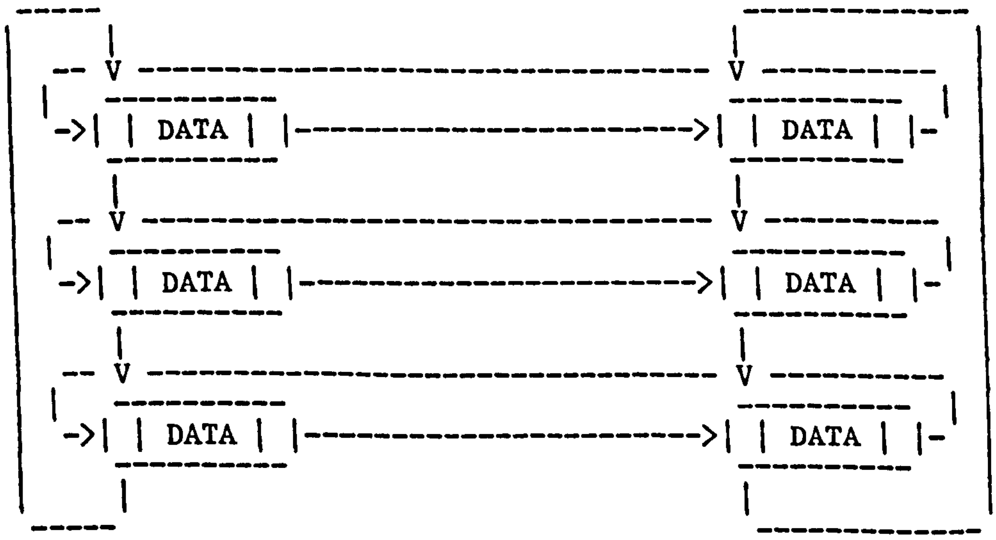
A Deque ("double-ended queue") is a linear list for which all insertions and deletions (and usually all accesses) are made at the ends of the list.



A deque is therefore more general than a Stack or a Queue; it has some properties in common with a deck of cards, which in a way can also further distinguish between output-restricted or input-restricted deques, in which insertions or deletions are allowed to take place at only one end respectively.

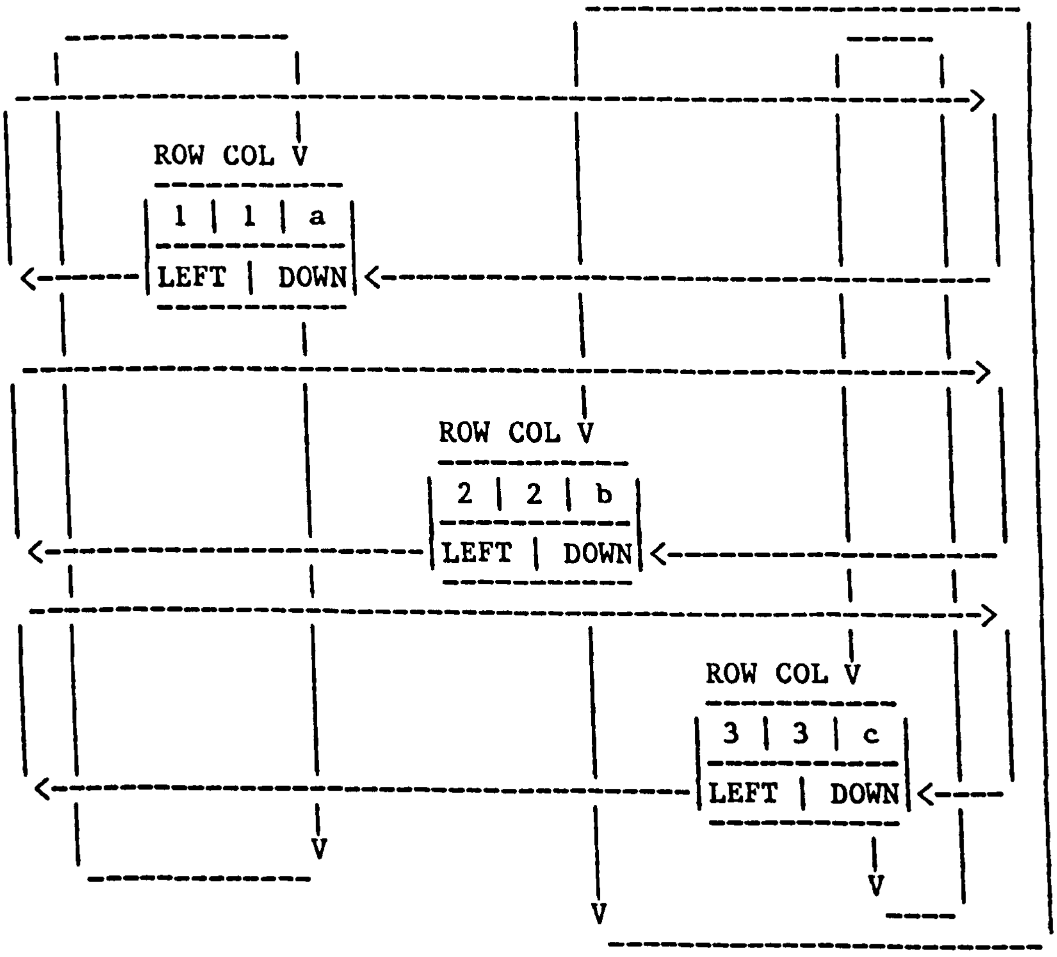
7.2.2 The Ring Structure:

If orthogonal Circular Lists are used, we have what is called a Ring Structure.



Ring Structures have proved to be quite flexible in a number of applications. The proper choice of representation depends, as always, on the type of insertions, deletions, and traversals that are needed in the algorithms that manipulate these structures. For instance, in the representation of sparse matrices (matrices of large order in which most of the elements are zero), the goal is to operate on these matrices as though the entire matrix were present, but ignore the zero entries in order to save memory. For example, the matrix

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}$$



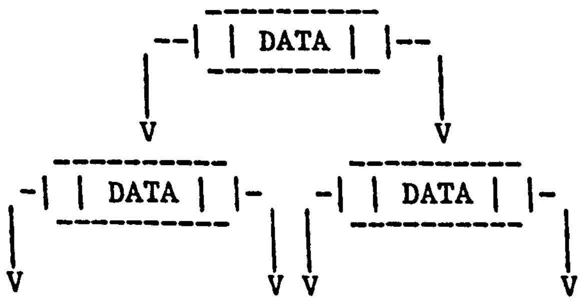
7.2.3 The Binary Tree Structure:

Tree structures have been the object of extensive mathematical investigations for many years, long before the advent of computers, and many interesting facts have been discovered about them. Generally speaking, tree structure means a"branching" relationship between nodes, much like that found in the trees of nature. Let us define a tree formally as a finite set T of one or more nodes such that

A) There is one specially designated node called the root of the tree.

B) The remaining nodes (excluding the root) are partitioned into $m \geq 0$ disjoint sets $T_1, \ldots, T_m$, and each of these sets in turn is a subtree of the root.

The simplest form of tree structure is a Binary Tree, which is an important type of tree, in the sense that a Binary Tree is not a special case of an ordinary tree, but it is another concept entirely (although we will see many relations between ordinary tree and Binary Tree). For example, conventionally, general trees are conveniently representable as Binary Trees, many trees that arise in applications are themselves inherently binary. A Binary Tree is defined as a finite set of nodes that is either empty, or consists of a root together with two binary trees. This definition suggests a natural way to represent binary tree with a Doubly Linked-List structure.



This simple and natural memory representation accounts for the special importance of Binary Tree structure, by which any Binary Tree can be contructed with the recursive data structure. There are many algorithms for the manipulation of Binary Tree structures, and one idea that occurs repeatedly in these algorithms is the notion of traversing or "walking through" a tree. A complete traversal of the tree gives us a sequence of movements of the nodes. Three principal ways may be used to traverse a Binary Tree:

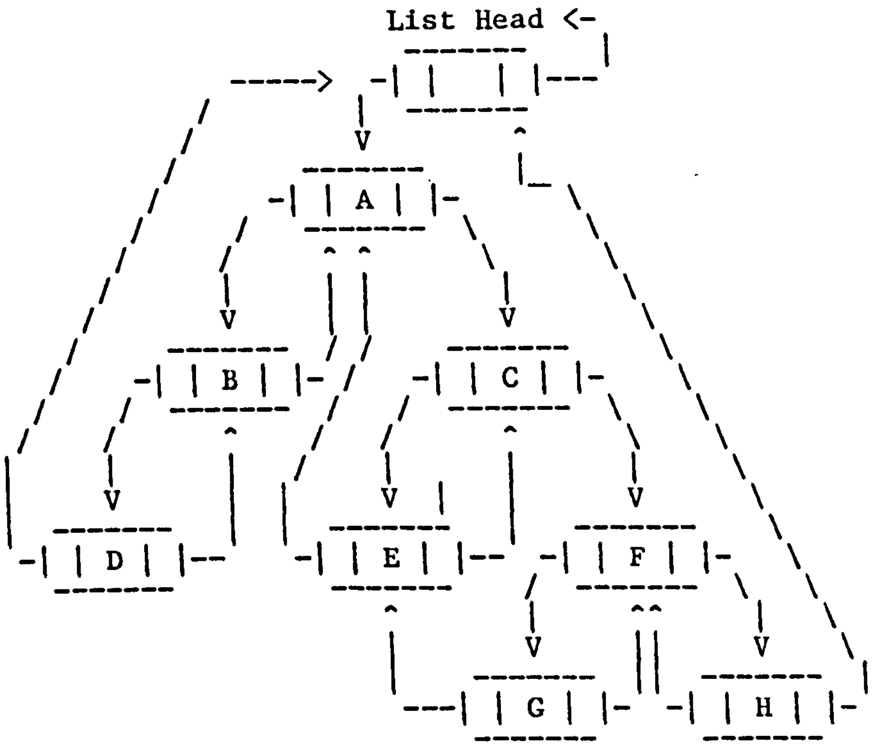
1) The Preorder Traversal
Visit the root
Traverse the left subtree
Traverse the right subtree

2) The Inorder Traversal
Traverse the left subtree
Visit the root
Traverse the right subtree

3) The Postorder Traversal
Traverse the left subtree
Traverse the right subtree
Visit the root

These three ways of arranging the nodes of a Binary Tree into a sequence are extremely important, as they are intimately connected with most of the computer methods dealing with trees. In many applications of Binary Tree, there is more symmetry between the meanings of the left subtrees and right subtrees, and in such cases, the Inorder is used, which puts the root in the middle, is essentially symmetric between left and right.

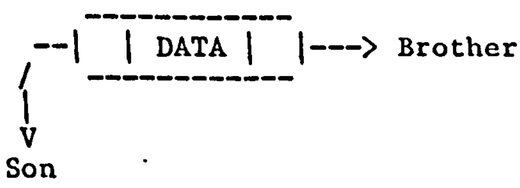
There is an important alternative representation of Binary Tree to replace the NULL links (to terminal links) by "threads" to other parts of the tree, as an aid to traversing the tree.

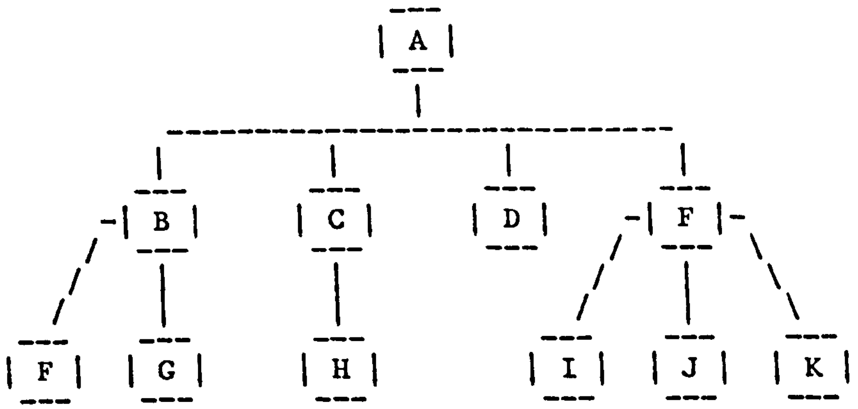


The great advantage of threaded trees is that the traversal algorithms become simpler. So a threaded Binary Tree is decidedly superior to an unthreaded one, with respect to traversal. However, these advantages are sometimes offset by the slightly increased time needed to insert and delete nodes in a threaded tree.

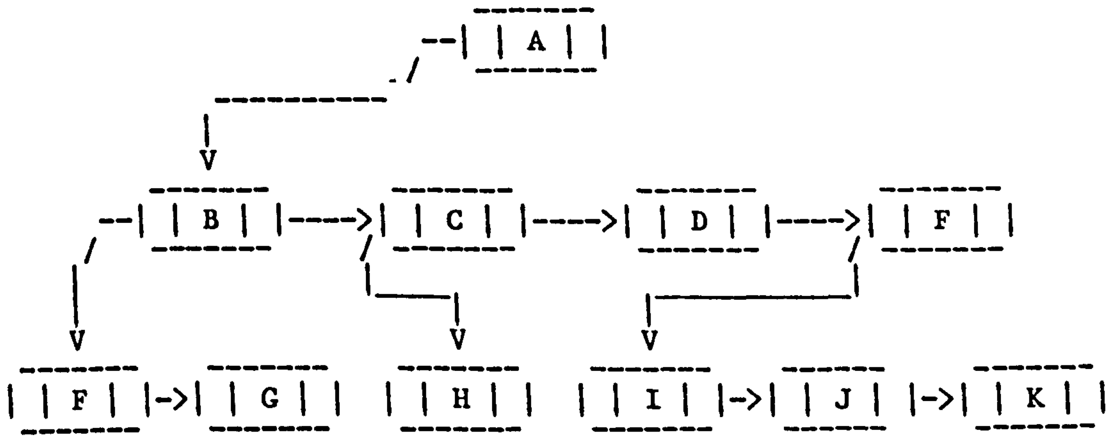
7.2.4 The Binary Tree Representation of Ordinary Tree Structure:

The problem in implementing ordinary trees is the nodes may have a different number of children and the maximum number of children may be much larger than the minimum or may be unknown prior to the generation of the tree. One solution to this problem is to use a dynamic tree structure implemented by Binary Tree, in which the left pointer points to its first (leftmost) son, and the right pointer points to its brothers in the same generation.





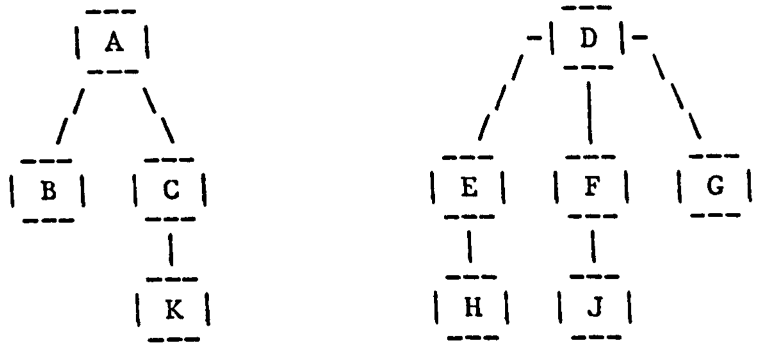
For example, the above ordinary tree could be mapped into a form of Binary Tree as shown in the following.



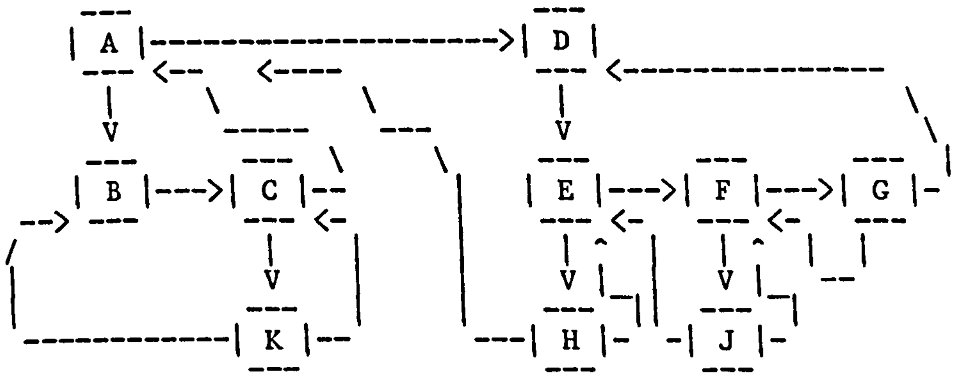
With this method of dynamic tree structuring, all types of ordinary trees could be implemented by Binary Tree.

7.2.5 The Binary Tree Representation of Forest Structure:

A Forest is a set (usually an ordered set) of zero or more disjoint trees, or in other words, the nodes of a tree excluding the root form a Forest. There is a natural way to represent any forest as a Binary Tree. Consider the following Forest of two trees:



The corresponding Binary Tree representation is obtained by linking the roots(fathers) of each family, and removing all vertical links except from a father to his son:



The above transformation gives the natural correspondence between Forests and Binary Trees. Note that right thread links go from the rightmost son of a family to the father. The ideas about traversal expressed in the previous section can be recast in terms of Forests. However, there is no simple analog of the "Inorder" sequence, since there is no obvious place to insert a root among its descendants; but "Preorder" and "Postorder" carry over in an obvious manner. Given any nonempty Forest, the two basic ways to traverse it may be defined as follows:

1) The Preorder Traversal

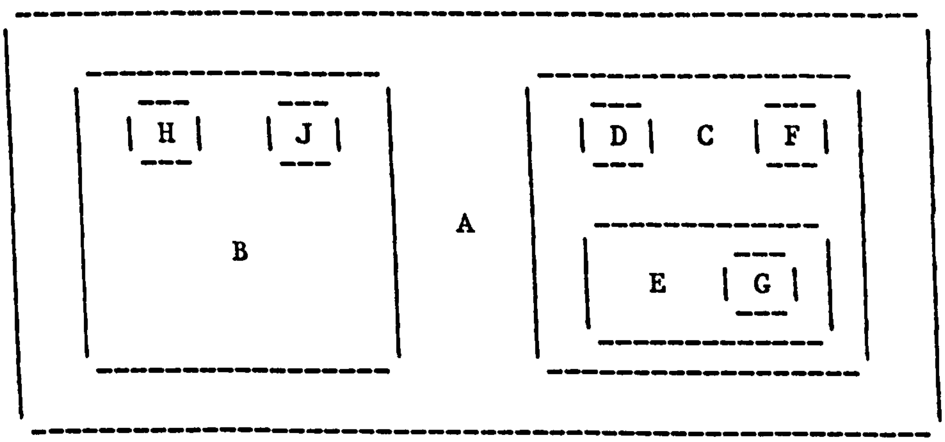
Visit the root of the first tree;
traverse the subtrees of the first tree (in Preorder);
traverse the remaining tress (in Preorder).

2) The Postorder Traversal

Traverse the subtrees of the first tree (in Postorder);
visit the root of the first tree;
traverse the remaining tress (in Postorder).

One of the most important application of Forest is the implementation of Set structures. Sets differ from Trees in that the members of a Set must be distinct, a condition not necessarily imposed on Tree structures. For example, in a special case of a nested set, which is a collection of sets in which any pair of sets is either disjoint or one contains the other:

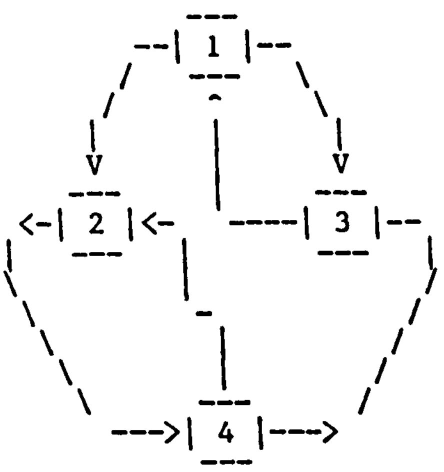
(A (B (H) (J))(C (D) (E (G)) (F)))



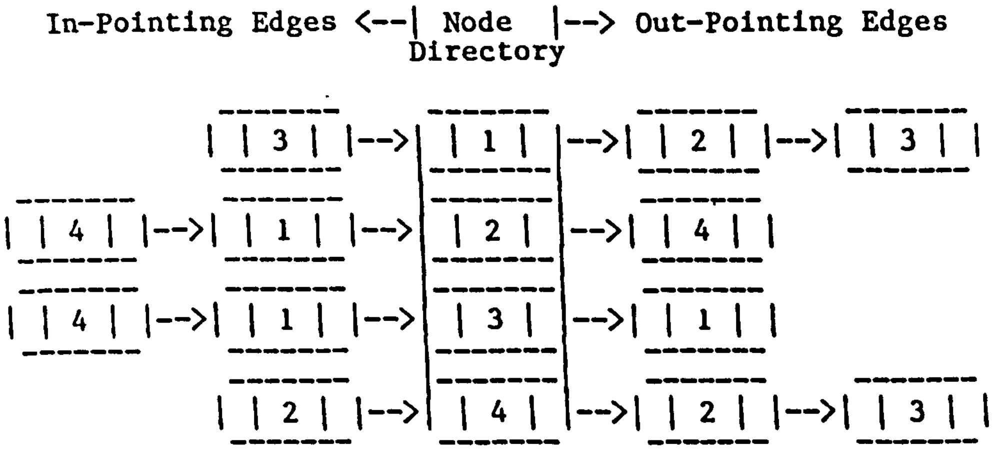
7.2.6 The Connected Graph Structure:

Intuitively, a Graph is a data structure used to represent relationships among objects. It is generally defined to be a set of points (called vertices) together with a set of lines (called edges) joining certain pairs of distinct vertices. There is at most one edge joining any pair of vertices, and two vertices are called adjacent if there is an edge joining them. By this definition, it is obvious that trees, in general, belong to a class of Graph structures with hierarchical relationships among items of data, which are sometimes referred to as connected Graphs without cycles.

A graph is connected if there is path between any vertices of the graph. Therefore, Circular List and Ring structures can both be considered as connected graphs. However, there is a good deal more significance if the direction of each edge is taken into account in the interpretation of a graph, and in this case we have what is called a "direct graph" or "digraph". The main feature of a direct graph from the modelling standpoint is that it indicates precedence constraints. If two nodes X and Y are linked by an arc from X to Y that this implies an activity to happen at node X must precede those of node Y. This constraint can be extended to be related to time, for example when X and Y are the Fetch and Execute sequences of the Control-Flow architecture. This key idea has found numerous applications in computer science, such as task scheduling, Semaphore, state graph diagram, program flowcharts, ... etc. However, it is not the purpose of this section to review them all exhaustively, instead we will concentrate on the Linked-List representation of Graphs. This is done by providing for each vertex two "vertex-edge lists" of its adjacent vertices: one for in-pointing edges, the other for out-pointing edges.



For instance, the above graph can be represented in the following Doubly Linked-List.



As a matter of fact, this Doubly Linked-List representation is the mapping of Graph Connectivity Matrix[76].

$$\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\
1 & --- & \text{Out} & \text{Both} & --- \\
2 & \text{In} & --- & --- & \text{Both} \\
3 & \text{Both} & --- & --- & \text{In} \\
4 & --- & \text{Both} & \text{Out} & ---
\end{array}$$

7.3 THE MULTILINKED-LIST DATA STRUCTURES :

A Multilinked-List data structure involves nodes with several linked fields in each node, not one or two as in most of our previous examples.

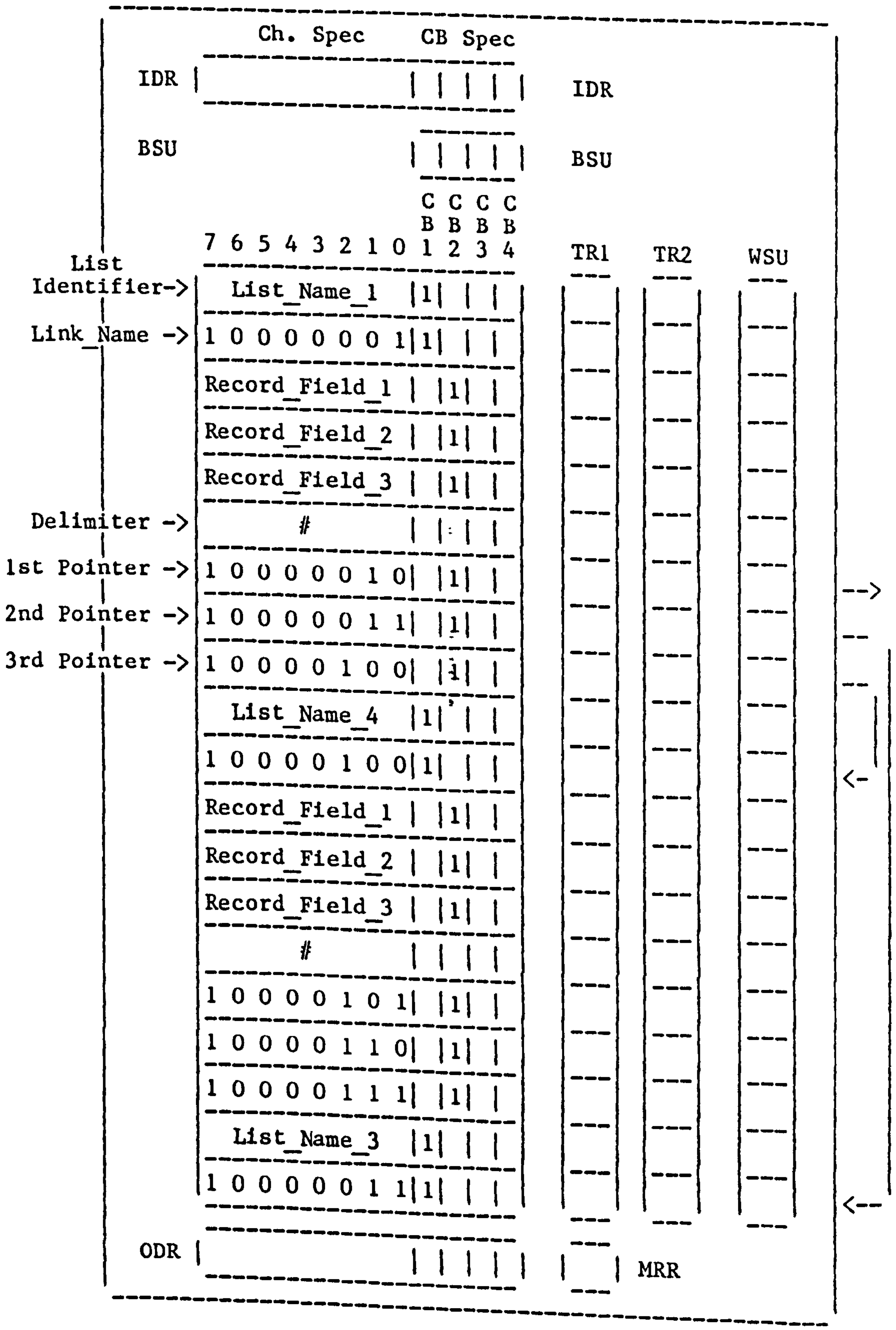
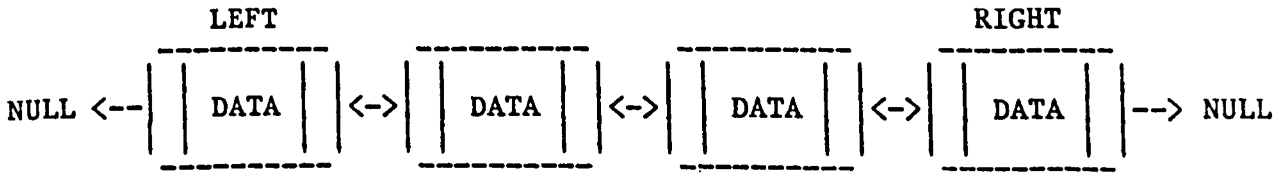


Fig. 7.3 The Data Organization of a Triply Linked-List Data Structure

Theoretically speaking, the number of pointers in the Multilinked-List structures can be increased to as many as desired. Nevertheless, in view of the implementation of most of the known data structures, they can all be accomplished by two pointer Doubly Linked-Lists. Because of its associative properties, all pointers on BOAP are implemented as bidirectional pointers, instead of unidirectional, as in conventional implementations. Hence, some of the Doubly Linked-List data structures previously discussed could in fact be implemented in Singly Linked-List on BOAP, i.e. Deque could be implemented in a Singly Linked-List shown as follows:



The following Associative Computation Cycle shows how a "father" link can be traced when necessary.

```
S[1XXXXXXX X1X1]BMR    +1,+2   ; SEARCH FOR PTR
GRS[1XXXXXXX 1XXX]             ; SEARCH FOR FATHER
RSGSD(U)                       ; ACTIVATE FATHER PTR
R(1010)BRN             @NEXT   ; MARK FATHER PTR
```

```
----------------------------------------------------------------
|
|              Ch. Spec     CB Spec
|            -------------------------
|   IDR |                   |1|0|1|0|     IDR
|            -------------------------
|                            -------
|   BSU                     | | | | |     BSU
|                            -------
|                            C C C C
|                            B B B B
|            7 6 5 4 3 2 1 0 1 2 3 4     TR1     TR2     WSU
List         -----------------------     ---     ---     ---
Identifier->|  List_Name_1  |1| | |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
Link_Name ->|1 0 0 0 0 0 0 1|1| |1|    |   |   | 1 |   | * |
|            -----------------------    ---     ---     ---
|           |Record_Field_1 | |1| |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |Record_Field_2 | |1| |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |Record_Field_3 | |1| |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
Delimiter ->|       #       | | | |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
Pointer ->  |1 0 0 0 0 0 1 0| |1| |1   | 1 |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |  List_Name_2  |1| | |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |1 0 0 0 0 0 1 0|1| | |    |   |   | 1 |   |   |
|            -----------------------    ---     ---     ---
|           |Record_Field_1 | |1| |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |Record_Field_2 | |1| |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |Record_Field_3 | |1| |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |       #       | | | |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |1 0 0 0 0 0 1 1| |1| |    |   |   | 1 |   |   |
|            -----------------------    ---     ---     ---
|           |  List_Name_3  |1| | |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |1 0 0 0 0 0 1 1|1| | |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |Record_Field_1 | |1| |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |Record_Field_2 | |1| |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|           |Record_Field_3 | |1| |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
Delimiter ->|       #       | | | |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
Pointer ->  |1 0 0 0 0 0 0 0| |1| |    |   |   |   |   |   |
|            -----------------------    ---     ---     ---
|            -----------------------    ---
|   ODR |                   | | | | |  | 1 | MRR
|            -----------------------    ---
----------------------------------------------------------------
```

7.4 SUMMARY:

Data type encapsulation is now a widely accepted method of program development and structuring[66,78,79]. Indeed, it is one of our chief programming paradigms[80]:

> one is encouraged to write programs as algorithms operating on abstract data types, then gradually refine the data representation, applying this method recursively, until a concrete representation is found[68].

At each stage, the abstract data being operated upon present themselves as the objects of certain operations, which may be observed to behave in particular ways but whose precise internal structure is hidden.

In this chapter, models of data structures have been developed based on linked-list structures, which in effect, create a mapping between abstract data types and the memory organization of BOAP. Briefly, every data type has a structure after refinement and all typed variables have a structure corresponding to that type. Nevertheless, we have not yet mentioned anything about the relational operations to be performed on these data structures, and how data items communicate with each other? It is well known that the concept and the use of relations are very important in data structures[4,81,82,83], so much so that, one can approach various data in a unified way via relations[77]. This unified approach seems especially attractive when data to be handled are diverse and heterogeneous[84].

1) UNION(R_1,R_2, ..., R_k)

2) INDEXED_UNION(I) : UNION(RELATION(I))

3) INTERSECTION(R_1,R_2, ..., R_k)

4) INDEXED_INTERSECTION(I) : INTERSECTION(RELATION(I))

5) SYMMETRIC_DIFFERENCE(R_1,R_2) : exclusive OR operation

6) RELATIVE_COMPLEMENT : $R_1 - R_2$

7) COMPLEMENT : $\overline{R}$

8) CARTESIAN PRODUCT(S_1, S_2) : $S_1 \times S_2$

9) INVERSE : R^{-1}

10) CARDINALITY(S) : the number of members in the set S

11) PROJECTION(L,R) : projection of R by L

12) PERMUTATION(A,B) : permutation of B by A

13) RANK(R) : rank of R

14) RELATION(E_1, E_2, ..., E_k) :

15) SUBSET(A,B) : true if A is a subset of B, false otherwise

16) EQUAL(A,B) : true if A = b, false otherwise

17) EQUIVALENT(A,B) : true if $|A| = |B|$, false otherwise

18) DISJOINT(A,B) : true if UNION(A,B) = 0, false otherwise

The relational data structure described above is general enough to handle operations of String, List, Tree, Set, Graph and all other data structures that we have discussed in this chapter. In BOAP, relations between data items are established by first searching for the data concerned, examining the Data Fields and/or Pointer Fields for the identities of data items, before building a relationship between them by means of marking Control Bits (communication links). There are many of these examples in Chapter Six's algorithms: the String UNION algorithm to form a union of set, the String ANY algorithm to check for set membership.

Finally, the point that we want to make here is that, the material which we present so far is not just "yet another way of implementing data structures", but in contrast, it is an expedition to a better and more efficient way of organizing; representing; accessing and manipulating data, so much so that the burden of searching and sorting[85], which has been predominating the field of data structuring for so long, can be eliminated.

CHAPTER EIGHT

CONCLUSION

Having contructed the Distributed Computer System, we now have a string processor, a list processor, a tree processor, a set and a graph processor within the computer system, to oversee the implementation of data structures and mathematical dictions.

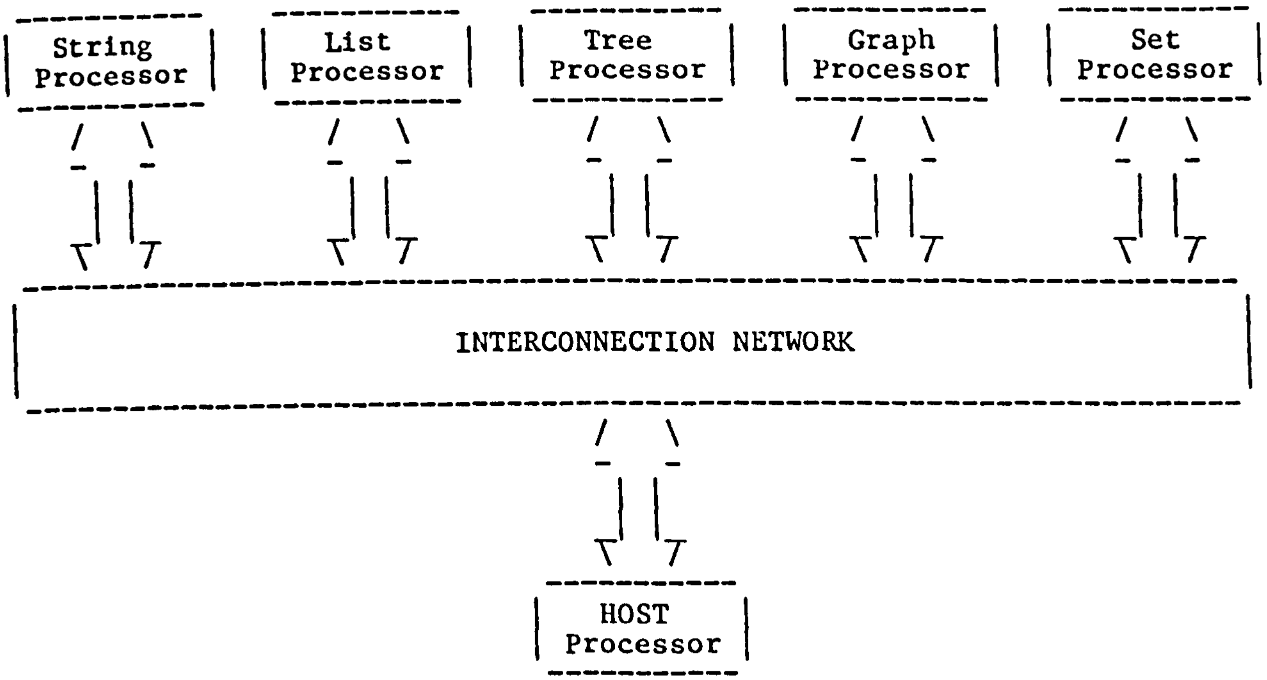
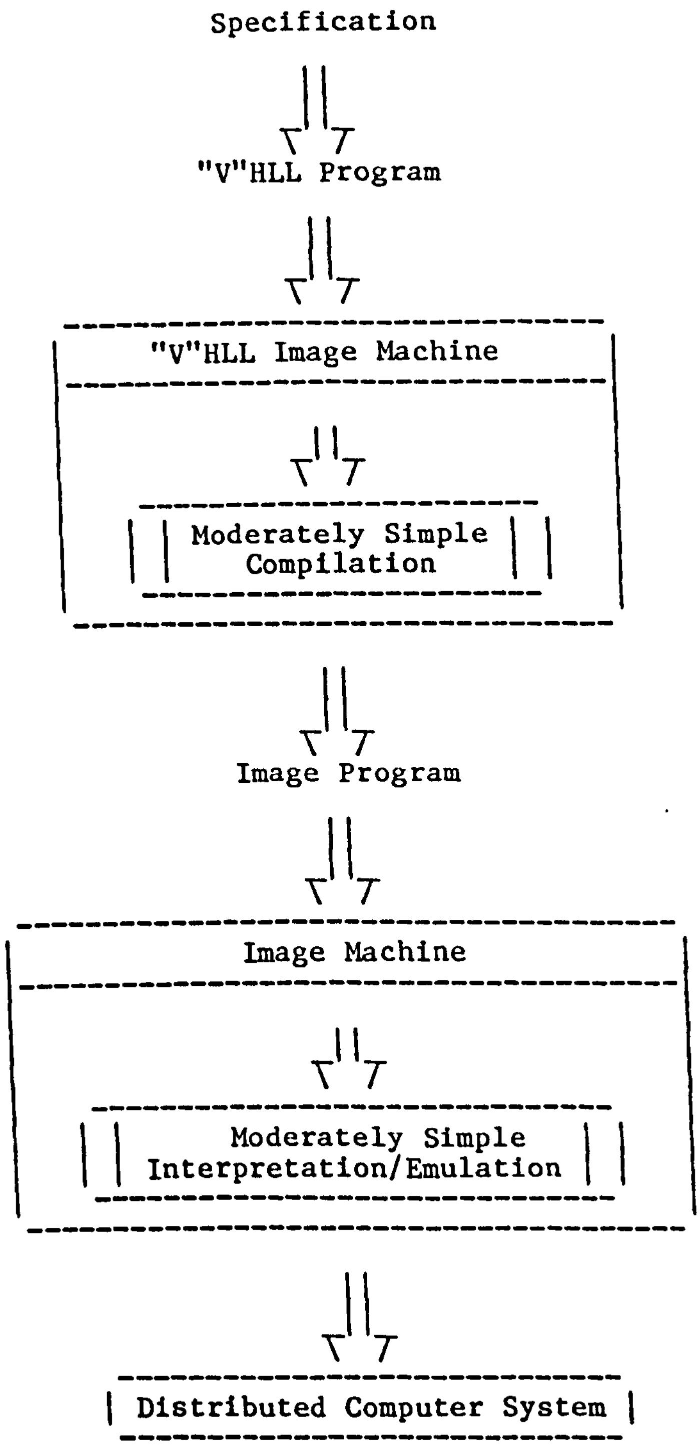


Fig. 8.1 The Model of the Distributed Computer System

In this transformation, the burden of program coding, the size and overhead of the compilation/interpretation process have been greatly reduced. Hence, the machine has now not only been equipped with various useful data structures and powerful mathematical dictions at the HLL level, but is actually running much more quickly and efficiently with a smaller compiler/interpreter at the system translation level, and the wishful thinking of Rex Rice[86,87] can now really come true. The Rice's Symbol IIR computer architecture was proposed in 1966 at Fairchild's research facility in Palo Alto, California, as a "blue print" to build a HLL machine. Although this machine was built and delivered to Iowa State University in 1971, the termination of funding and the hardware failures have forced it to be permanently decommissioned in 1978. Nevertheless, it had taught us a great deal about building HLL machines. Summing up the experience of the Symbol computer, and with the aids of present VLSI technology, we could now build a even more powerful machine based on the model of our Distributed Computer System.


Specification
"V"HLL Program
"V"HLL Image Machine
Moderately Simple Compilation
Image Program
Image Machine
Moderately Simple Interpretation/Emulation
Distributed Computer System

With the Distributed Computer System (DCS), we bring together two very distinct processors, each performs what it is best capable of doing: The Von Neumann machine (SISD processor) for SISD operations, and the BOAP (SIMD processor) for SIMD operations. Nevertheless, this level of specialization could be further extended, by bringing in more special purpose processors into the DCS.

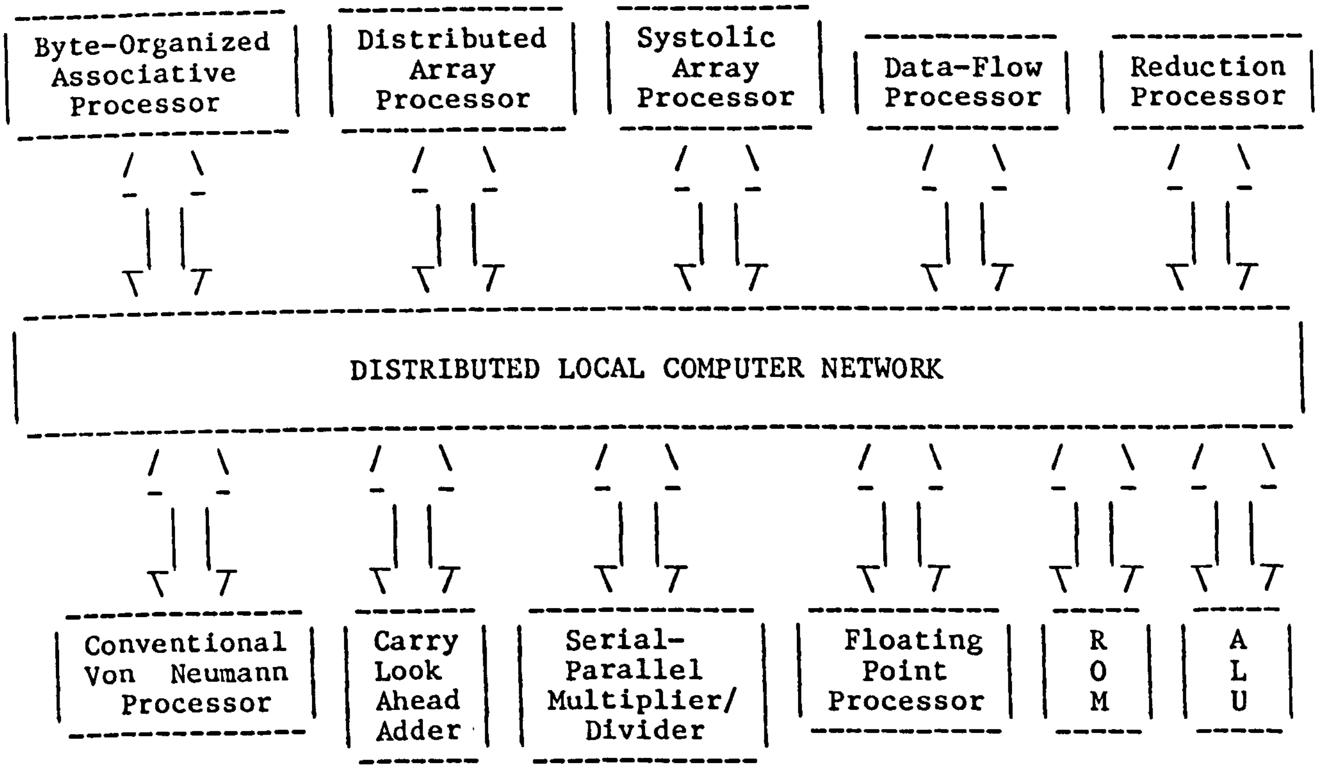

Byte-Organized Associative Processor
Distributed Array Processor
Systolic Array Processor
Data-Flow Processor
Reduction Processor
DISTRIBUTED LOCAL COMPUTER NETWORK
Conventional Von Neumann Processor
Carry Look Ahead Adder
Serial-Parallel Multiplier/ Divider
Floating Point Processor
R O M
A L U

Fig. 8.2 The Extended Distributed Computer System

In this extended DCS, the Von Neumann processor is still remains the Host of the system, to oversee the program sequencing, scheduling, task allocation, I/O control and system reconfiguration[88,89], and as many special purpose SISD processors as necessary are integrated into the system for implementations of arithmeic and logic operations: the carry-look-ahead adder[90], the serial-parallel multiplier/divider[91] and the floating-point processor[92] for arithmetic processings; the look-up-table in ROM[91] for trigonometric functions; the single chip ALU[90] for logic operations. On the other hand, the byte-organized associative processor (BOAP) is used here to facilitate all SIMD non-numerical processing, and the ICL Distributed Array Processor[93], or Kung's Systolic Array[94] type of hardwares are used for implementation of those SIMD "number crunching" operations such as matrix operations, radar image processing, FFT and signal processing etc. Data-flow[46-54,95] and reduction[55-60] types of

MIMD processors can also be included for the implementation of mathematical expressions and recursive operations respectively. All processors are connected together via the interconnection network[39-45,96-99], which provides data (and control) communications between the various processors.

So what is the next step forward after this extended Distributed Computer System? Apart from the consolidation and expansion of the existing system, we think that it is perhaps the time to start thinking about a more higher level machine on which natural languages could be used to program the machine. For many years, Jean Sammet[100] has been preaching the virtues of allowing ordinary individuals to communicate with a computer in their own natural language (which is simply meant to be the language native to the group that using it, e.g., English, French, German, and which also contains scientific notation wherever it is appropriate). One of the primary advantages of this concept is to make it easier for any non-computer minded person to communicate with a computer to get his/her task done.

Work on natural languages has been at the center of A. I. research into the ways in which concepts can be represented and cognitive processes organized. Since language is vital to our thought, any theories concerning memory or reasoning are strongly intertwined with the attempt to understand how language works. It is believed that a deep understanding of the context is vital to all uses of language. Applied to machine translation, this means that before one can translate material about a subject, one must first have a program that "understands" the subject. However, in writing a program for understanding natural languages, one is faced with all the problems of artificial intelligence, problems of coping with huge amounts of knowledge, problems of finding ways to represent and describe complex cognitive structures, as well as problem of finding an appropriate structure in a gigantic space of possibilities. Among the areas in which research on the application of natural language understanding systems[101] is currently active are machine translation[102], information retrieval [103,104], and interactive interfaces to computer systems[105,106,107].

Like all the other programming languages, natural language would have to go through a translation process, similar to the compilation/interpretation process of the HLLs, except that it would have to be strictly interpretive:

1) Lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Pragmatics Analysis
5) Code Generation

In the Syntax Analysis, the parsing problem consists of finding the structure of an input string, based on a given grammar. This is a common problem on the analysis of natural language and HLLs. Whereas the designers of HLLs hope to aviod ambiguity, the designers of natural languages must accept it. The grammatical component used most is a context-free grammar augmented by conditions, constraints, restriction, or transformations, and the result is determined by how the context-free system is augmented[108]:

> A parse tree is produced according to a formal grammar expressed as an Augmented Transition Network (ATN). The ATN, which is a general representation of a phrase-structure grammar, retains some of the simplicity of a finite-state machine but is extended to context-free power by allowing recursion. It is further enhanced by allowing the use of registers, arbitrary conditions and actions. Parsers for ATN grammars can incorporate advances made in the general theory of context-free parsing.

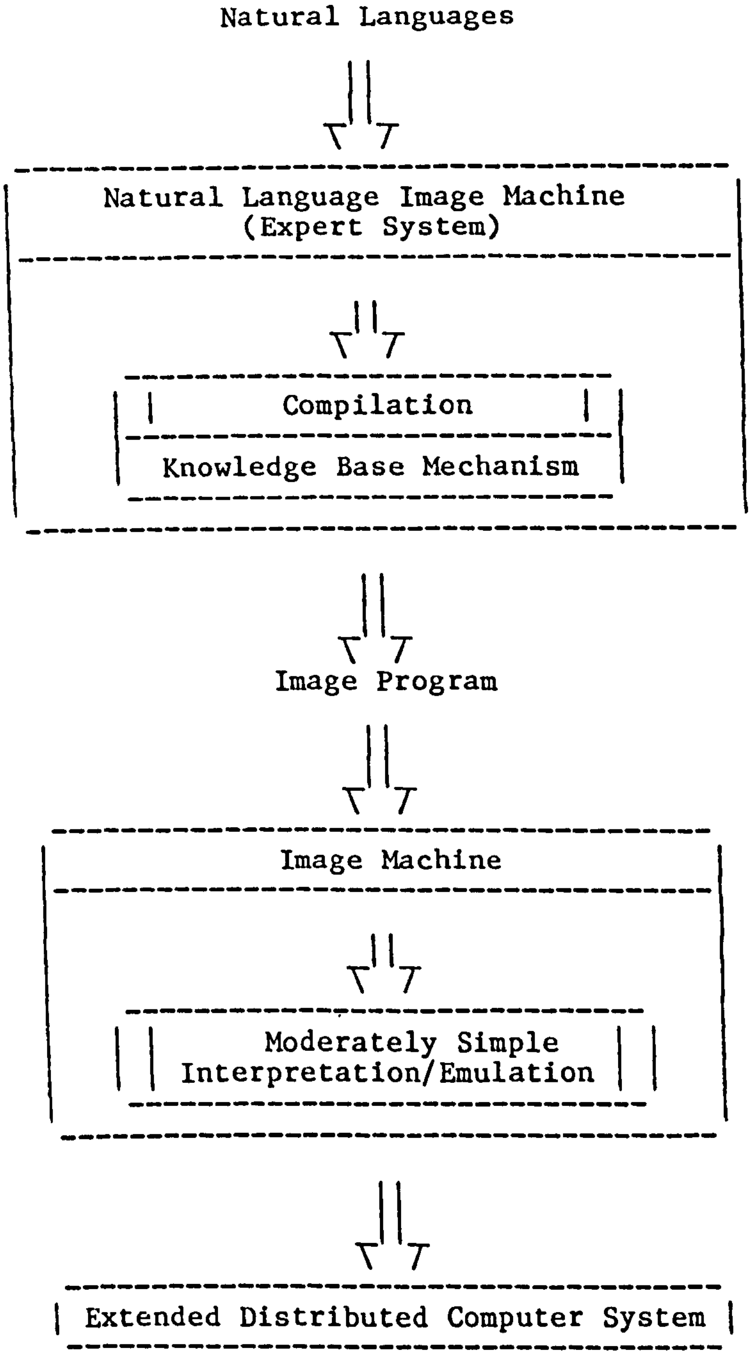
On the whole, semantics interpretation has had the greatest impact of understanding on natural language. In order to represent the meaning of words and sentences, it is necessary to have a formalism for representing facts, concepts, and ideas. Work in the semantics of natural language has followed two general lines: using formal logic and developing new representations. Several standard techniques exist for representing knowledge in natural language systems, namely, semantic networks[109,110], procedural semantics[108], and frames[111]. Based on these systematic set of representations, problem-solving and reasoning could then proceed by means of formal logic. In all these systems the existence of prototype frames makes possible the use of "expectations" in analysis. When an ambiguous or underspecified phrase or sentence occurs, it can be compared with a description of what would be expected, based on the prototype; if there is a plausible fit to the expectation, assumptions can be made as to what was meant. Researchers are currently involved in developing tools to cope with the complexities of these data structures and control--"VHLL" programming constructs. This would enable programmers to concentrate on the complexities more closely connected to the structure of language and thought, rather than the details of programming constructs.

Pragmatics is the study of the use of language in context which some people refer to as "common-sense". For example, when you talk to someone, you have a prior understanding that you and he/she have much in common. You share a large body of what might be called common-sense knowledge of the human world--physical objects, events, thoughts, motivations. In asking a question, stating a desire, or giving information, you include just enough detail for the other person to be able to understand what you are saying. Moreover, information about the communication itself, as well as its context in a conversation, are vital to understanding of what is being said. These is the subjects of fuzzy logic and inexact reasoning. Anyway, this aspect of language is one that is just beginning to be dealt with in current systems. Although most large systems in the past had specialized ways of dealing with a subset of pragmatic problems, there is as yet no theoretical approach. However, as people look to interactive system for teaching and explanation, it seems likely that this will be the major focus of research in the 1980s.

Summing-up the above disscussion on natural language, it has become apparent that "problem solving" is the key issue in the natural language processing. By problem solving, we mean a large corpus of basic ideas having to do with the pocesses of deduction, inference, planning, "commom sense" reasoning, and theorem-proving, ideas that have been applied in programs for understanding natural languages, information retrieval, automatic programmming, robotics, scene analysis, game-playing, and mathematical theorem-proving. Here we examine some ideas concerning problem solving.

The problem solver has two requirements that are logically indepentent. One defines the allowable configurations for the class of problem (representation), while the other defines the solution for a problem of that class (reasoning). Problem-solving methods are characterized by searching through a state, or situation space or through a space of alternatives. A solution of a sequence of state transitions from an initial state or states given in the problem specification, to a final, or goal state. A solution sequence is any succession of states such that the transition is consistent with the problem specification and the operators provided by the method. The term search emphasizes the teleological nature of the solution sequence; it need not involve much trial-and-error seaching, although some searching is as inevitable in machine problem-solving as in human problem-solving.

Research in natural languages is usually conducted by building large-scale systems, by intensively studying subproblems and algorithms, and by formally analyzing these systems. The state of the art is exemplified by the large-scale systems. These systems have become the context for developing and exploring algorithms, as well as for additional research. In them are evident the subproblems of designing representations of knowleuge, developing organized bodies of linguistic knowledge, and designing algorithms for processing natural languages. By formally analyzing mathematical models of natural languages, it will become possible to study the power of and limitations on various approaches. Therefore, it is apparent that in such a system, it needs not only a knowledge of the structure of the language, but a body of "world knowledge" about the domain discussed in the language. Thus a comprehensive, language understanding system presupposes an extensive reasoning system, one with a base of commom-sense and domain-specific knowledge.



In the fall of 1981, Japan had called a international conference on Fifth Generation Computer Systems[112], which has sent a great pulse of excitement across the whole computing community. These "fifth generation" plans are centred on knowledge-based systems, which embody the specialised knowledge and experience of a human expert, so much so that one could simply "talk" to the machines to tell them what to do. Although a lot of ideas presented in the conference are not new to us; some of them could even be traced back a few decades ago, the whole world was impressed by the determination and the schedule for the realisation of this radical plans: it was reported that the Japan's Ministry of International Trade and Industry (MITI) had set up a institute for new generation computer technology, and would be spending 20 million pounds over the first three years on the project. Together with government contributions later in the programme, and with those from big companies such as Fujitsu, Hitachi and NEC, the total outlay over 10 years could amount to between 500 million and 1000 million pounds. But, it is not just the funding that is impressive, in the two years leading to the conference, the Japanese had spent 100 man-years in identifying their research priorities--before setting out this ten year programme, which set targets for key technology and software advances, merging hardware and software to an unprecedented degree. Since the plan was published and discussed in the FGCS conference at Tokyo in October, 1981, Western governments and industry have been taking this programme very seriously.

Semantically, the "Fifth Generation Computer System" (FGCS) is a very misleading term as the aim of the programme is to produce a radically new family of the computers (A. I. machines) of the 1990s. But, in our opinion, the A. I. machine will not be the only type of machine in the next generation of computer systems, other new generation systems such as weather forecasting machines, air traffic control systems, VLSI development tools ... etc may themselves not be related to A. I., yet could be classified under the fifth generation computer system. Traditionally, the term "generation" is used to describe the advance of computer technology[113]. As stated by Bell and Newell, "The generations are best definded solely in terms of logic technology"[114]:

1) The First Generation is that of vacuum tubes (1945 - 58)

2) The Second Generation is that of transistors (1958 - 66)

3) The Third Generation is that of ICs (1966 - 72)

4) The Fourth Generation is that of LSI circuits (1972 - 82)
The LSI circuitry is a integrated subsystem on a chip[28]. The Intel 4004 chip set was the first commercially available microprocessor that marked the beginning of the Fourth Generation computers[115].

5) The Fifth Generation is that of VLSI circuits (1982 - ?)
The VLSI is a complete digital system on a chip[116].

Therefore, we think that it may be more appropriate to refer to the Japanese FGCS as expert systems. Nevertheless, we do agree that A. I. would be the dominant force of the next generation of computer systems.

Basically, the areas of research and development targeted in the Japanese FGCS are as follows:

1) The Hardware Level :
The "System 5G" is proposed--a VLSI CAD system, on which any design from basic VLSI architecture to mask pattern in a uniform manner can be performed[117]. The personal logic programming station with LISP and PROLOG will be served as the standard inference terminal in the System 5G.

2) The Architecture Level :
Since the FGCS are designed as Knowledge Information Processing System (KIPS) which realize a very high level and flexible man-machine interface based on generalized or specialized knowledge data, abstract data type, relational algebra, and database support mechanisms, have to be integrated into the systems[118-121]. In these respects, the Japanese had concluded that the data-flow machine and database machines are the most promising candidates for the basic architecture of the KIPS[122].

3) The HLL Level :

The Japanese had adopted PROLOG as the starting point [123-128] and working towards the definition of the 5G-Kernel Language[129], or "a core language" for short, to serve as a nucleus of the software systems and a fundemental specification for the architecture of FGCS[119]. This proposed language will be a type of logic programming language designed on the basis of a simple inference like a syllogism in logic. It is expected to incorporate the capability to specify parallel processing and to express more advanced functions pertaining to knowledge or meta-inference mechanisms.

4) The Natural Language Level :

The expectation for the FGCS is as Karaisu has stated[130]:

> "... non professional without training can handle the new machines. This must be placed at the first position."
>
> :

With this requirement, natural language processing, speech processing, and image processing are three fundamental categories of research into intelligent man-machine interfaces[131].

5) The Expert System Level :

The expert systems are the final goal of the FGCS plan, all points mentioned so far are the foundation lain down for the ultimate building of expert systems. Hence, the heart of the FGCS project is to develop a methodology for building knowledge information processing systems which will provide people with the intelligent agents[132,133]--field of A. I. currently starting to yield significant commercial results in expert systems.

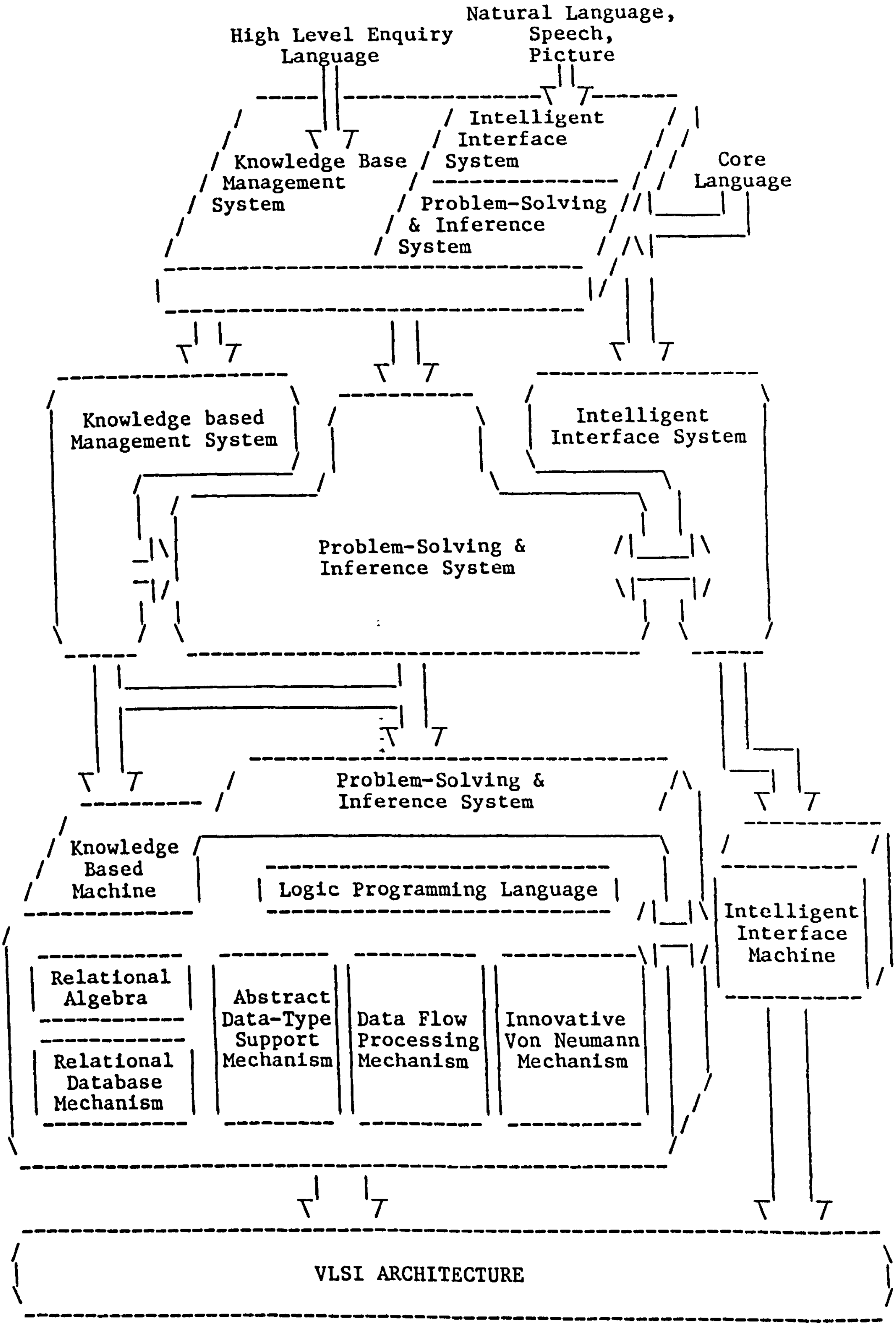
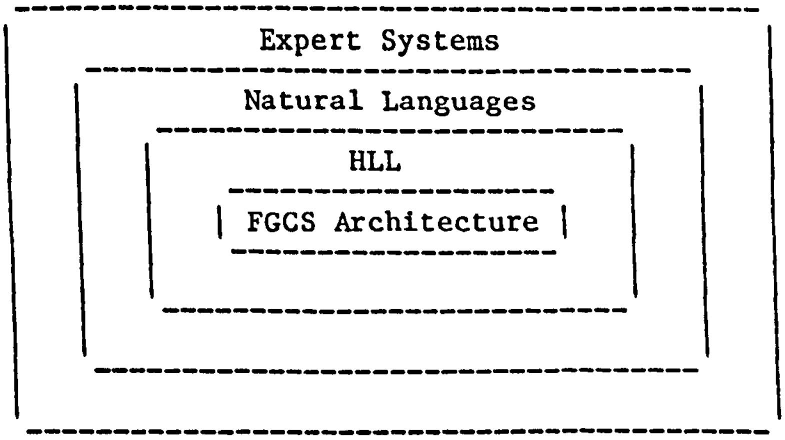

High Level Enquiry Language
Natural Language, Speech, Picture
Knowledge Base Management System
Intelligent Interface System
Problem-Solving & Inference System
Core Language
Knowledge based Management System
Problem-Solving & Inference System
Intelligent Interface System
Problem-Solving & Inference System
Knowledge Based Machine
Logic Programming Language
Relational Algebra
Relational Database Mechanism
Abstract Data-Type Support Mechanism
Data Flow Processing Mechanism
Innovative Von Neumann Mechanism
Intelligent Interface Machine
VLSI ARCHITECTURE

Fig. 8.3 The Japanese Fifth Generation Computer Systems

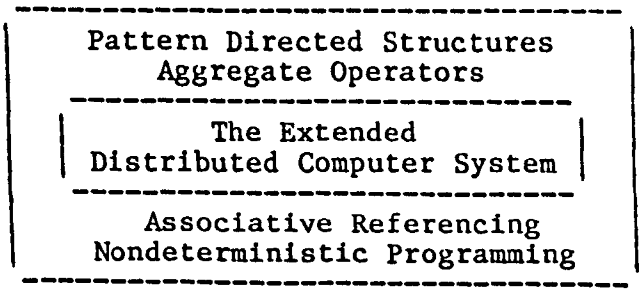
The Japanese FGCS plan is divided into three stages and at the end of each stage there appears to be a short term marketing opportunity to develop products. For example, the first stage examines mechanisms of inference machines, including the use of data-flow machines for symbol processing; the second stage aims to build a sequential inference machine, at the end of which the machine would be integrated with a knowledge base machine; the third and final stage would build an integrated prototype of an expert system. In this plan, the Japanese has certainly presented to the world their views of FGCS, but the author does not think that they have taken the right approach for the realisation of their program. Our criticisms are as follows:

1) The First Stage of the FGCS Plan

By analysing the five different level of activities, it has become apparent that interrelations do exist between them: on the one hand; the FGCS Architecture has to be equipped with facilities to support HLL and expert systems, yet, on the other hand, the specification of programming contructs are difficult to establish without a host machine for natural language and expert systems to develop ideas.



However, we have concluded that the host machine of FGCS has to be designed and built in the first stage of the program. In other words, the architecture and HLL are the two levels of activities that ought to be dealt with first before all other activities. Using the top down approach, it is not difficult to estabish the programming constructs needed to support the processing of natural languages and expert systems. This information could then be used as a guidance to draw up the specifications for the architecture and HLL of the host machine. In this respect, we think that the Japanese have made a very serious mistake, by wrongly identifying the data-flow machine (a MIMD machine) as the solution for symbol processing, abstract data types, relational algebra, and data base support mechanisms, which are mostly of SIMD type architecture. Undoubtedly, the data-flow type MIMD machines will be needed in FGCS for the processing of arithmetic expressions and implementation of multi-tasking type operations, but it is certainly not the answer to SIMD type programming constructs. In our opinion, the Extended Distributed Computer System, with its undisputable capability both as the symbol processing and database machines, is a more promising candidate for the first stage of a FGCS program. This point is also supported by Edward Feigenbaum[134], who in his statement in the Tokyo conference pointed out that the fifth generation computer systems would be primarily symbolic manipulation systems.


Pattern Directed Structures
Aggregate Operators

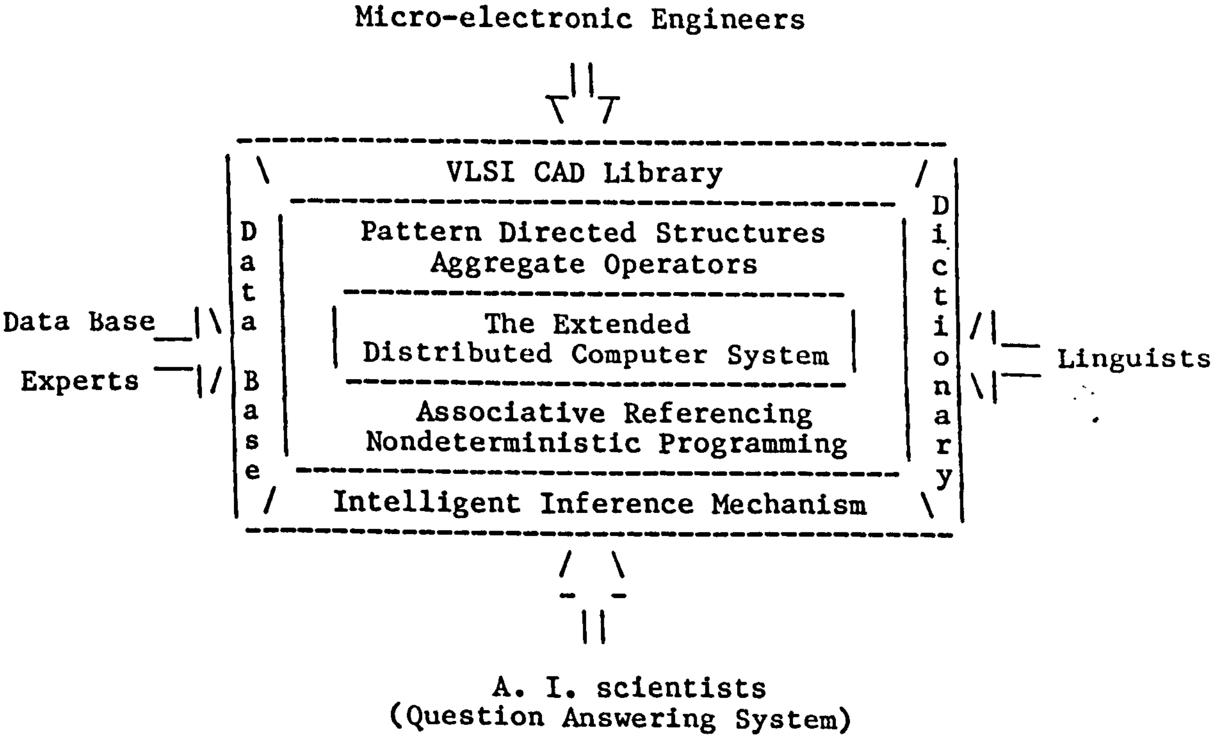
The Extended
Distributed Computer System

Associative Referencing
Nondeterministic Programming

Futhermore, although PROLOG is a very powerful database and A. I. language, it is still far from being as a "universal programming language", therefore, if the Jananese proceed with PROLOG as a standard of their "FGCS", substantial improvement will have to be added onto the language before it could be the standard language of the FGCS. It may be better to accept a more popular and well-defined language, such as PASCAL or C, as a starting point, and enhance it with the "VHLL" programming constructs[135]. However, it is essential to adopt an extensible modular aproach[136] towards the language design of FGCS, by firstly designing the core language and a set of well-known data structures as the standard language of the first stage of the program, and any enhancement in the later stages will then just be an extension of this standard language. In other words, the new architecture in the later stages should provide for the upward compatibility that programms written in the standard language should run on these new machines with minimum changes.

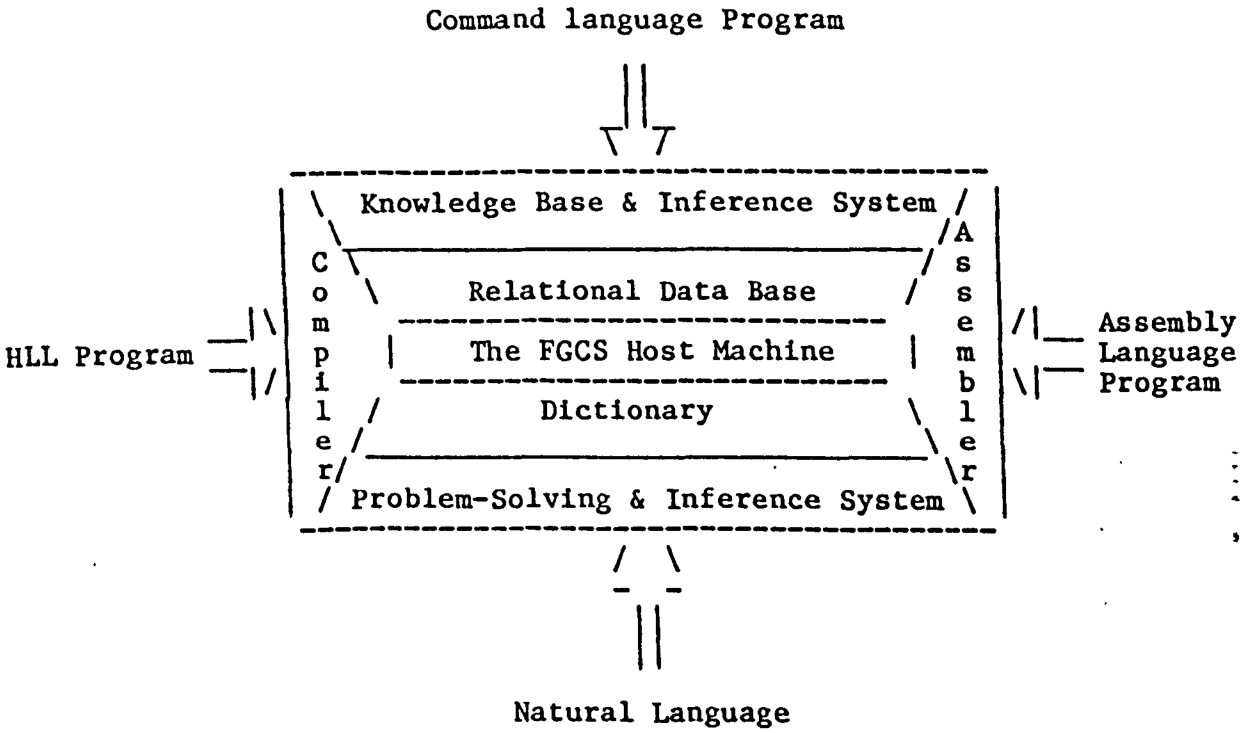
2) The Second Stage of FGCS Plan:

Having constructed the FGCS host machine (Extended DCS), the micro-electronic engineers, the A. I. scientists, the linguists, and the database experts can then move on the host machine to develop the System 5G, question answering sytems, natual language translating machines, and data base machines.


Micro-electronic Engineers
VLSI CAD Library
Data Base
Pattern Directed Structures
Aggregate Operators
The Extended
Distributed Computer System
Associative Referencing
Nondeterministic Programming
Dictionary
Intelligent Inference Mechanism
Data Base Experts
Linguists
A. I. scientists
(Question Answering System)

3) The Third Stage of FGCS Plan:

This is the final integration stage, in which all different parts of FGCS subsystems are connected together onto an interconnection network to form a expert system--the Japanese FGCS. (Phil Treleaven's analysis[137] has stated that the FGCS will represent a unification of research into VLSI processors and into distributed processing which will allow replicated general-purpose computing elements, as well as special-purpose computing elements, to be integrated into a network.)



Despite all these differences, we think the Japanese has certainly made a very significant contribution toward the development of FGCS. At the very least, they have set the world computing targets for the rest of the decade and beyond. The natural of these targets and the timing of their announcement omen the dawn of the second computer revolution and the new round of races for supremacy is already beginning in earnest. However, our choice is simple: whether to develop our own plans to escape from the Von Neumann architecture or to be prepared to accept the Japanese domination in the 1990s.

BIBLIOGRAPHY

1. S. J. Schwartz, "On Programming: An Interim Report on the SETL Project--Installment I: Generalities," Computer Science Dept., Courant Institute of Mathematical Sciences, New York University (1973).

2. E. F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus," Report RJ 893, IBM Research Lab., San Jose, Calif., (July 1971).

3. E. F. Codd, "Relational Completeness of Data Base Sublanguage," Report RJ 987, IBM Research Lab., San Jose, Calif., (March 1972).

4. E. F. Codd, "A Relational Model of Data for Large Shared Data Bank," CACM, Vol. 13, No. 6, (June 1970).

5. A. A. Markov, "Theory of Algorithms," Akad. Nauk, USSR, 1954 (English edition OTS-USDC 1961).

6. K. E. Iverson, "A Programming Language," New York, Wiley, (1962).

7. R. E. Griswold, J. F. Poage and I. P. Polonsky, "The SNOBOL4 Programming Language," Prentice-Hall (1968), Englewood Cliffs. N. J.

8. W. F. Clocksin and C. S. Mellish, "Programming in Prolog," Springer-Verlag, Germany, (1981).

9. J. McCarthy., et al., "LISP 1.5 Programmer's Manual," Cambridge, Mass, MIT Press, (1962).

10. M. J. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Trans. Computer, Vol. C-21, No.9, pp.948-960, (September 1972).

11. R. W. Floyd, "Nondeterministic Algorithms," JACM, Vol. 14, (October 1967).

12. S. Golomb and L. Baumert, "Backtrace Programming," JACM, Vol. 12, No. 4, (october 1965).

13. B. M. Leavenworth and J. Sammet, "An Overview of Nonprocedureal Languages," Proc. Symp. of Very-High-Level Languages, ACM SIGPLAN Notices, Vol. 9, No. 4, pp.1-12, ACM, New York (April 1974).

14. K. Krishna et al., "HLL Architectures: Pitfalls and Predilections," Proc. of 9th Annual Symp. on Computer Architecture, Austin, Texas, USA, 26-29 April 1982, pp.18-23, (1982).

15. A. A. Hanlon, "Content-addressable and Associative Memory Systems: A Survey," IEEE Trans. Computers EC-15, pp.509-521, (August 1966).

16. B. Parhami, "Associative Memories and Processors: An Overview and Selected Bibliography," Proc. IEEE Vol.61, pp.722-730, (June 1973).

17. J. Minker, "An Overview of Associative or Content-addressable Memory Systems and a KWIC Index to the Literature," Computing Reviews Vol.12, No.10, pp.453-504, (October 1971).

18. K. J. Thurber and L. D. Wald, "Associative and Parallel Processors," Computing Surveys Vol.7, No.4, pp.215-255, (December 1975).

19. S. S. Yau and H. S. Fung, "Associative Processors Architecture--A Survey," ACM Computing Surveys, Vol.9, No.1, pp.3-27, (march 1977).

20. T. DiGiacinto, "Airborne Associative Processor (ASPRO)," in Proc. AIAA Comput. in Aerosp. III Conf.,pp.202-205, (October 1981).

21. J. M. Surprise, "Airborne Associative Processor (ASPRO)," in Proc. of 1981 International Conf. on Parallel Processing, IEEE, pp.129-130 (August 1981).

22. K. E. Batcher, "Design of a Massively Parallel Processor," IEEE Trans. Computer, Vol. C-29, pp.836-840, (September 1980).

23. R. M. Lea, "An Associative Parallel Processor for efficient and flexible file-searching," Proceedings International Symposium on Technology for Selective Dissemination of Information, 1976 IEEE, New York, pp.73-78, (1976).

24. G. H. Barnes, "The Illiac IV Computer," IEEE Trans. Computer, Vol. C-17, NO.8, pp.746-757, (August 1968).

25. D. L. Slotnick, et al, "The Solomon Computer," AFIPS Conf. Proc., FJCC, Vol.22, pp.97-107, (1962).

26. M. H. Lewin, "Retrieval of Order Lists From a Content-Addressed Memory," RCA Review, Vol. 23, pp.215-229, (1962).

27. I. E. Sutherland and C. A. Mead, "Microelectronics and Computer Science," Scientific Amercian, Vol.237, No.9, pp.210-228, (September 1977).

28. C. A. Mead and L. Conway, "Introduction to VLSI System," Addision-Wesley Publishing Company (1980).

29. D. G. Fairbairn, "VLSI: A New Frontier for System Designers," IEEE Computer, Vol. 15, No. 1, pp.87-96, (January 1981).

30. P. Penfield and J. Rubinstein, "Signal Delay in RC Tree Networks," Proc. Second Caltech Conf. Very Large Scale Integration, (1981).

31. W. Lattin, "The Challenge of Microprocessor Design in the 80's," Proc. Calthec Conf. Very Large Scale Integration, (1979).

32. M. A. Franklin, D. F. Wann, and W. J. Thomas, "Pin Limitations and Partitioning of VLSI Interconnection Newtworks," IEEE Trans. Computer, Vol. C-31, No. 11, pp.1109-1116, (November 1982).

33. D. K. Hsiao, "Data Base Computer," Advances in Computers, Vol.19, pp.1-59, Academic Press (1980).

34. H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," Sparse Matrix Proc. 1978, Society for Industrial and Applied Mathematics, pp.256-282, (1979).

35. H. M. Ahmed, J. Delosme and M. Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," IEEE Computer, Vol. 15, No. 1, pp.65-82, (January,1981).

36. L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," IEEE Computer, Vol. 15, No. 1, pp.47-56, (January 1982).

37. D. J. Farber et al., "The Distributed Computing System," in Dig. COMPCON'73, pp.31-34, (February 1973).

38. S. S. Yau, C. C. Yang, and S. M. Shatz, "An Approach to Distributed Computing System Software Design," IEEE Trans. Software Eng., Vol. SE-7, No.4, pp.427-436, (July,1981).

39. T. Y. Feng, "A Survey of Interconnection Network," IEEE Computer, Vol. 14, No. 12, pp.12-27, (December,1981).

40. G. H. Barnes and S. F. Lundstrom, "Design and Validation of a Connection Network for Many-Processor Multiprocessor systems," IEEE Computer, Vol. 14, No. 12, pp.30-41, (December,1981).

41. D. M. Dias and J. R. Jump, "Packet Switching Interconnection Networks for Modular systems," IEEE Computer, Vol. 14, No. 12, pp.43-53, (December 1981).

42. P. Y. Chen, D. H. Lawrie, D. A. Padua, and P. C. Yew, "Interconnection Networks Using Shuffles," IEEE Computer, Vol. 14, No. 12, pp.55-64, (December 1981).

43. H. J. Siegel and R. J. McMillen, "The Multistage Cube: A Versatile Interconnection Network," IEEE Computer, Vol. 14, No. 12, pp.65-76, (December 1981).

44. E. E. Swartlander and B. K. Gilbert, "Supersystems: Technology and Architecture," IEEE Trans. Computer, Vol. C-31, No.5,pp.399-409, (May 1982).

45. K. J. Thurber and H. A. Freeman, "Local Computer Networks," IEEE Computer Society Press, (1981).

46. P. G. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," ACM Computing Surveys, Vol.14, No. 1, pp.93-143, (March 1982).

47. K. E. Batcher, "Bit-Serial Parallel Processing Systems," Vol. C-31, No. 5, pp.377-384, (May 1982).

48. J. B. Dennis, "The Varieties of Data Flow Computers," Proc. 1st Int. Conf. on Distributed Computing Systems, pp.430-439, (October 1979).

49. M. Cornish, "The TI Data Flow Architecture: The Power of Concurrency for Avionics," in Proc. 3rd Conf. Digital Avionics Systems, IEEE, New York, pp.19-25, (November 1979).

50. A. L. Davis, "The Architecture and System Method of DDM1: A Recurisively Structure Data Driven Machine," in Proc. 5th Annu. Symp. Computer Architecture, ACM, New York, pp.210-215, (April 1978).

51. K. V. Arvind and K. Pingali, "A Processing Element for a Large Multiprocessor DataFlow Machine," in Proc. Int. Conf. Circuits and Computers, IEEE, New York, (October 1980).

52. I. Watson and J. Gurd, "A Prototype Data Flow Computer with Token Labeling," in Proc. Nat. Computer Conf. Vol. 48, AFIPS Press, pp.623-628, (June 1979).

53. D. Comte and N. Hifdi, "LAU Multiprocessor: Mircofunctional Description and Technology Choices," in Proc. 1st European Conf. Parallel and Distributed Processing, Toulouse, France, pp.8-15 (February 1979).

54. P. G. Treleaven, R. P. Hopkins and P. W. Rautenbach, "Combining Data Flow and Control Flow Computing," Comput. J. Vol. 25, No. 2, pp.207-217, (May 1982).

55. W. E. Kluge and H. Schlutter, "An Architecture for the Direct Execution of Reduction Languages," in Proc. Int. Workshop High-Level Language Computer Architecture, pp.174-180, (May 1980).

56. P. G. Treleaven and G. F. Mole, "A Multi-Processor Reduction Machine for User-Defined Reduction Languages," in Proc 7th Int. Symp. Computer Architecture, IEEE, New York, pp.121-130, (May 1980).

57. G. A. MaGo, "A Cellular Computer Architecture for Functional Programming," in Proc. IEEE COMPCON 80, IEEE, New York, pp.179-187, (February 1980).

58. R. M. Keller, et al., "A Loosely Coupled Applicative Multiprocessing System," in Proc. Nat. Computer Conf., AFIPS Press, pp.861-870, (1978).

59. D. A. Turner, "A New Implementation Technique for Applicative Language," Soft. Pract. Exper. No. 9, pp.31-49, (January 1979).

60. T. J. W. Clarke, P. J. S. Glandstone, C. D. Maclean and A. C. Norman, "SKIM--The S, K, I Reduction Machine," in Proc. LISP-80 Conf., pp.128-135, (Augest 1980).

61. I. H. Witten and Y. H. Ng, "An Ideographic Language Front End Processor for Accessing English Computer Systems," Comput. J. Vol. 24, No. 1 pp.62-70, (February 1981).

62. A. S. Tanenbaum, "Structured Computer Organization," Prentice-Hall, (1976).

63. G. A. Miller, "The Magical Number Seven, Plus or Minus two: Some Limits on Our Capacity for Processing Information," Pschological Review, Vol. 63, pp.81-97, (1956).

64. E. W. Dijkstra, "Notes on Structured Programming," in Structured Programming, (ed. by O. J. Dahl), Academic Press, New York, pp.1-82, (1972).

65. R. C. Linger, H. Mills, and B. Witt, "Structured Programming: Theory and Practice, Addison-Wesley, (1979).

66. D. Parnes, "On the Criteria to be used in Decomposing Systems onto Modules," Comm. ACM, Vol. 15, No. 2, pp.1053-1058, (December 1972).

67. D. Parnes, "On the Design and Development of Program Families," IEEE Trans. Software Eng., Vol. SE-2, No.1, pp.1-9, (March 1976).

68. N. Wirth, "Program Development by Stepwise Refinement," Comm. ACM, Vol. 14, No. 4, pp.221-227, (April 1971).

69. J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An Initial Algebra Approach to the Speciification, Correctness, and Implementation of Abstract Data Types," in Current Trends in Programming Methodology, Vol. 4, (ed. by R. T. Yeh), Prentice-Hall, Englewood Cliffs, N. J. pp.80-149, (1978).

70. C. B. Jones, "Software Development--A Rigorous Approach," Prentice-Hall (1980), Englewood Cliffs, N. J.

71. C. Ghezzi and M. Jazayeri, "Programming Language Concepts," John Wiley & Sons, Inc., (1982).

72. C. A. R. Hoare, "Data Structures," in Current Trends in Programming Methodology, Vol. 4, (ed. by R. T. Yeh), Prentice-Hall, pp.1-11, (1978), Englewood Cliffs, N. J. .

73. D. E. Kunth, "The Art of Computer Programming Vol. 1 : Fundamental Algorithms," Addison-Wesley, Reading, Mass, (1968).

74. F. L. Bauer and H. Wossner, "Algorithmic Language and Program Development," ed. by D. Gries, Springer-Verlag, (1982).

75. A. L. Rosenberg, "Storage Mappings for Extendible Arrays," in Current Trends in Programming Methodology, Vol. 4, (ed. by R. T. Yeh), Prentice-Hall, pp.1-11, (1978), Englewood Cliffs, N. J.

76. J. L. Baer, "Graph Models in Programming Systems," Vol. 3, (ed. by K. M. Chandy and R. T. Yeh), Prentice-Hall, pp.168-231, (1978), Englewood Cliffs, N. J.

77. S. Y. Bnag and R. T. Yeh, "Notes on Relational Data Structures," in Current Trends in Programming Methodology, Vol. 4, (ed. by R. T. Yeh), Prentice-Hall, pp.241-262, (1978), Englewood Cliffs, N. J.

78. C. A. R. Hoare, "Notes on Data Structuring" in Structured Programming, (ed. by O. J. Dahl), Academic Press, New York, pp.83-174, (1972).

79. O. J. Dahl and C. A. R. Hoare, "Hierarchial Program Structures," (ed. O. J. Dahl), Academic Press, New York, pp.175-220, (1972).

80. R. W. Floyd, "The Paradigms of Programming," Comm. ACM, Vol. 22, No. 8 pp.455-460, (Augest 1979).

81. K. S. Fu, "Linguistic Approach to pattern Recognition," in Applied Computation Theory (ed. by R. T. Yeh), Prentice-Hall, Englewood Cliffs, N. J., (1974).

82. B. Raphael, "SIR: A Computer Program for Semantic Information Retrieval," in Semantic Information Processing (ed. by M Minsky), MIT Press, Cambridge, Mass, (1968).

83. W. C. McGee, "File Structures for Generalized Data Management," Proc. IEIP Congress, North Holland, Amsterdam, (1969).

84. R. T. Yeh, "Generalized Pair Algebra with Applications to Automata Theory," JACM, Vol 15, No. 2, pp.304-316, (1968).

85. D. E. Kunth, "The Art of Computer Programming Vol. 3 : Sorting and Searching," Addison-Wesley, Reading, Mass, (1973).

86. R. Rice and W. R. Smith, "SYMBOL--A Major Departure from Classic Software Dominated Von Neumann Computing System," AFIPS Conf. Proc., SJCC, Vol. 38, pp.575-587 (1971).

87. W. R. Smith et al., "SYMBOL--A Large Experimetal System Exploring Major Hardare Replacement of Software," AFIPS Conf. Proc., SJCC, Vol. 38, pp.601-616, (1971).

88. C. R. Vick, S. P. Kartashev, and S. I. Kartashev, "Adaptable Architectures for Supersystems", IEEE Computer Volume 13, No.11, pp.17-35 (November 1980).

89. S. I. Kartashev and S. P. Kartashev, "A Multicomputer System with Software Reconfiguration of the Architecture," Proc. Conf. Computer Performance, ACM-SIGMETRICS/CMG VIII Washingston D.C., pp.271-286 (1977).

90. Texas Instruments, Inc., " The TTL Data Book for Design Engineers", 2nd. Ed. Dallas, Texas, (1976).

91. Monolithic Memories, Inc., "Bipolar LSI Data Book", Sunnyvale, CA, (July 1978).

92. Adanced Micro Devices, Inc., "The AM2900 Family Data Book", Sunnyvale, CA, (1976).

93. R. W. Gostick, "Software and Hardware Technology for the ICL Distributed Array Processor", The Australian Computer Journal, Volume 13, No.1, pp.1-6 (February 1981).

94. H. T. Kung, "Why Systolic Architectures?", IEEE Computer, volume 15, No.1, pp.37-46 (January 1982).

95. J. B. Dennis, "Data Flow Supercomputer", IEEE Computer, Volume 13, No.11, pp.48-56 (November 1980).

96. M. T. Liu, "Distributed Loop Computer Network," Advances in Computers, Vol 17, pp.163-221, (1978).

97. J. F. Shoch, Y. K. Dalal, D. D. Redell and R. C. Crane, "Evolution of the Ethernet Local Computer Network," IEEE Computer, Vol. 15, No. 8, pp.10-27, (August 1982).

98. A. Hopper, "Local Area Computer Communication Networks", PhD dissertation, University of Cambridge, (April 1978).

99. M. V. Wilkes and D. J. Wheeler, "The Cambridge Digital Communication Ring", Proceeding pf the LACN Symposium, pp.47-60 (May 1979).

100. J. E. Sammet, "An Overview of High-Level Languages," Advances in Computers, Academic Press, Vol. 20, pp.199-259, (1981).

101 T. R. Addis, "Machine Understanding of Natural Language," Int. J. Man-Mach. Stud. (GB), Vol. 9, No. 2, pp.223-231, (March 1977).

102 M. Nagao, and J. Tsujii, "Some Topics of Language Processing for the Purpose of machine Translation," Research Reports in Japan, pp.310-334 (November 1981).

103. W. J. Plath, "REQUEST: A Natural Language Question-Answering System," IBM J. Res. Develop., Vol. 20, pp.326-335, (1976).

104. D. L. Waltz, "An English Language Question Answering System for a Large Relational Database," Comm. ACM 21, pp.526-539 (1978).

105. W. A. Woods, "Progress in Natural Language Understanding: An Application to Lunar Geology," AFIPS Coference Proceedings, Vol. 42, (1973).

106. J. S. Brown and R. Burton, "Multiple Representations of Knowledge for Tutorial Reasoning," pp.311-349 in Representation and Understanding, ed. D. G. Bobrow and A, M. Collins, Academic Press, New York (1975).

107. J. R. Carbonell and A. M. Collins, "Natural Semantics in Artificial Intelligence, "Amer. J. Computational Linguistics, Vol. 1, No. 3, (1974).

108. W. A. Woods, "Semantics and Quantification in Natural Language Question Answering," Advances in Computer, Vol. 19, (1978).

109. R. Fikes and H. Hendrix, "A Network-based Knowledge Representation and its Natural Deduction System," Proceeding of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, pp.235-246, (1977).

110. Roger D. Schank and Kenneth M. Colby (eds.), "Computer Models of Thought and Language," Freeman (1973).

111. M. Minsky, "A Framwork for Representing Knowledge," in "The Psychology of Computer Vision," edited by P. H. Winston, McGraw-Hill, New York, (1975).

112. "Proceedings of International Conference on Fifth Generation Computer Systems," Tokyo, (October 1981).

113. J. P. Hayes, "Computer Architecture and Organization," McGraw-Hill Computer Science Series, (1978).

114. C. G. Bell and A. Newell, "Computer Structures : Reading and Examples," McGraw-Hill, New York, (1971).

115. J. B. Peatman, "Microcomputer-Based Design," McGraw-Hill, New York, (1977).

116. R. C. Johnson, "32-bit Microprocessors inherit mainframe features," Electronics, pp.138-141, (February 1981).

117. K. Sakamura et al., "VLSI and System Architecture-- The Development of System 5G," Proceedings of FGCS (see reference 112).

118. K. Fuchi, "Aiming for Knowledge Information Processing System," Proceeding of FGCS (see reference 112).

119. H. Aiso, "Fifth Generation Computer Architecture," Proceedings of FGCS (see reference 112).

120. S. Uchida et al., "New Architecture for Inference Mechanisms," Proceedings of FGCS (see reference 112).

121. M. Amamiya et al., "New Architecture for Knowledge Base Mechanisms," Proceedings of FGCS (see reference 112).

122. Hi. Tanaka et al., "The Preliminary Research on Data Flow Machine and Data Base Machine as the Basic Architecture of Fifth Generation Computer," Proceedings of FGCS (see reference 112).

123. H. Tanaka and Y. Matsumoto, "PROLOG and Natural Language Processing," Research Reports in Japan, pp.57-63, (November 1981).

124. T. Yokoi, "PROLOG and Data-Flow Computer Mechanisms," Research Reports in Japan, pp.64-71, (November 1981).

125. M. Suwa and H. Tanaka, "A PROLOG Based Production System," Research Reports in Japan, pp.72-78, (November 1981).

126. K. Furukawa, "Problem Solving with PROLOG," Research Reports in Japan, pp.79-83, (November 1981).

127. S. Uchida and T. Higuchi, "Logic Simulation in PROLOG," Research Reports in Japan, pp.84-90, (November 1981).

128. K. Nitta and Koichi Furukawa, "Description of the PROLOG Interpreter by a Concurrent Programming Language, Research Reports in Japan, pp.91-96, (November 1981).

129. T. Yokoi et al., "Logic Programming and a Dedicated High-performance Personal Computer," Proceedings of FGCS (see reference 112).

130. H. Karaisu, "What is Required of the Fifth Generation Computer--Social Needs and its Impact," Proceedings of FGCS (see reference 112).

131. Ho. Tanaka et al., "Intelligent Man-Machine Interface," Proceedings of FGCS (see reference 112).

132. K. Furukawa et al., "Problem Solving and Inference Mechanisms," Proceedings of FGCS (see reference 112).

133. M. Suwa et al., "Knowledge Base Mechanisms," Proceedings of FGCS (see reference 112).

134. E. A. Feigenbaum, "Innovation in Symbol Manipulation in the Fifth Generation Computer Systems," (see reference 112).

135. W. Bibel, "Logical Program Synthesis," Proceedings of FGCS (see reference 112).

136. C. Prenner, "Modern Extensible Languages," 7th Annual Symposium on Computer Science and Statistics, Oct. 18-19, 1973, pp380-388, (1973)

137. P. Treleaven, "Fifth Generation Computer Architecture Analysis," Proceedings of FGCS (see reference 112).

GLOSSARY

GLOSSARY

AAL : Associative Assembly Language.

ACC : Associative Computation Cycle.

ACD : Tag Manipulation code.

ALU : Arithmetic and Logic Unit.

AMA : Associative Memory Array.

AMI : Associative Machine Instruction.

AMI1 : The machine instruction of API1.

AMI234 : The machine instruction of API234.

AMIAR : Associative Machine Instruction Address Register.

AMIR : Associative Machine Instruction Register.

APC : Associative Program Counter.

API : Associative Processing Instruction.

API1 : The examine phase of API in beat 1.

API234 : The execute phase of API which occurring in beat 2, beat 3 and beat 4.

Architecture: A program representation that can be interpreted. Strictly speaking, it is the instruction set and I/O connection capabilities. Hence, the architecture of a machine is the "blue print" used to build it.

Adaptable Architecture : An architecture which able to adjusts to computed algorithms by mean of software.

GLOSSARY

Control-Flow Architecture : The control-flow architecture has a control-driven computation organization, which is characterized by the lack of an examine phase: instructions are arbitrarily selected, and once selected they are immediately executed. This implies that the program has complete control control over instruction sequencing.

Data-Flow Architecture : An architecture which implement the data-flow principles inherent in modern program structure by allowimg each instruction to be executed as soon as its operands arrive.

Reduction Architecture : A reduction architecture has a demand-driven computation organization, and is characterized by an outermost computation rule coupled with the ability to coerce arguments at the examine phase.

SISD Architecture : The single instruction, single data stream organization.

SIMD Architecture : The single instruction, multiple data stream organization.

MIMD Architecture : The multiple instruction, multiple single data stream organization.

ATN : Augmented Transition Network.

B-end : Bottom end.

Binary Tree : A binary tree is defined as a finite set of nodes that is either empty, or consists of a root together with two binary trees.

BOAM : Byte-Organized Associative Memory.

BOAP : Byte-Organized Associative Processor.

BSU : Bit Select Unit.

GLOSSARY

CAD : Computer Aided Design.

CAM : Content Addressable Memory.

CB1 : Control Bit 1.

CB2 : Control Bit 2.

CB3 : Control Bit 3.

CB4 : Control Bit 4.

Circular List: A circularly-linked list has the property that its last node links back to the first instead of to NULL.

CLAB : Clear All Bits.

CLBCT : Clear Bits on Complemented Tags.

CLBTT : Clear Bits on True Tags.

CMOS : Complementary Metal Oxide Semiconductor.

CMB : Data Complementing Bit.

Command : A function which given a particular state, determines the next state.

Dynamic Array: An open ended one dimensional array.

DCS : Distributed Computer System.

Deque : A deque a double-ended queue in which all insertions and deletions are made at the ends of the list.

DI1 : Data Identity at beat 1.

DI2 : Data Identity at beat 2.

DI4 : Data Identity at beat 4.

DOC : Data Output Conflict.

EIR : End In Run.

GLOSSARY

FGCS : Japanese Fifth Generation Computer System.

FIFO : First In First Out queue.

Forest : A forest is a set (usually an ordered set) of zero or more disjoint trees, or in other words, the nodes of a tree excluding the root form a forest.

Floor Plan : A chip floor plan is merely a block diagram with blocks drawn to approximate scale and the routing of major buses, clocks, power, ground, and critical signal paths specified in terms of their location and the layer on which they run.

Graph : A graph is a set of points (called vertices) together with a set of lines (called edges) joining certain pairs of distinct vertices.

GRN : Group Run.

GRS : Group Run Search at beat 2.

GRSC : Group Run Search with Complement tags.

HLL : High Level Language.

IBR : Input Buffer Register.

IDR : Input Data Register.

Instructions : The set of all image commands, which represents the architecture of the image machine, They are sometimes referred to as image instructions.

IQE : Input Queue End (pointer).

IQF : Input Queue Front (pointer).

IR : Instruction Register.

LIFO : Last-In-First-Out.

GLOSSARY

List : List is defined (recursively) as a finite sequence of zero or more atoms/Lists which can be accessed by means of pointer.

LSI : Large Scale Integration.

Machine : A set of commands and a storage which is exactly the range and domain of the commands, together with a mechanism that causes the state transitions determined by the commands. If the mechanism is itself a machíne (i.e., has commands, storage, and mechanism), the original machine is called the image machine and the mechanism is called the host machine.

Machine Organization : A implementation of the machine, shown by in the form of block diagram.

Machine Realization : The actual hardware interconnection and construction of the machine with a given technology.

MCU : Microprogrammed Control Unit.

Microinstructions : The commands comprising the host machine.

MOPS : Million Operation Per Second.

MOR : Middle Out Run.

MR : Match Reply.

MRR : Match Reply Register.

NMOS : N-Channel Metal Oxide Semiconductor.

Nonprocedural Languages : A language is nonprocedural to the degree that it shortens the distance between fomulating and solving some significant classes of programming problems.

OBR : Output Buffer Register.

ODR : Output Data Register.

GLOSSARY

OQE :	Output Queue End (pointer).
OQF :	Output Queue Front (pointer).
OVB :	Overflow responses at the Bottom-end.
OVT :	Overflow responses at the Top-end.
PC :	Program Counter.
PEs :	Processing Elements.
PF :	Pre/Post-Function selection bit.
PCT :	Propagate Complement Tags.
PLB :	Propagation Link at the Bottom-end.
PLT :	Propagation Link at the Top-end.
PLA :	Programmable Logic Array.
Process :	A sequence of commands and an initial state.
PTT :	Propagate True Tags.
Queue :	A queue is a Singly Linked-List for which all insertions are made at one end of the list; all deletion (and usually all accesses) are made at other end.
RAM :	Random Addressed Memory.
Ring :	A orthogonal circular list.
RSCTD :	Resolve Complement Tags Down.
RSCTU :	Resolve Complement Tags Up.
RSFGD :	Resolve First Group Down.
RSFGU :	Resolve First Group Up.
RSFGSD :	Resolve First Group Start Down.
RSFGSU :	Resolve First Group Start Up.

GLOSSARY

RSGSD : Resolve Group Start Down.

RSGSU : Resolve Group Start Up.

RSTTD : Resolve True Tags Down.

RSTTU : Resolve True Tags Up.

R/W : Read/Write selection bit.

Set : A set is a collection into a whole of definite distinct objects of our intuition or of our thought, with some common property as directed from N-tuple. The objects are called elements (members) of the set.

SP : Scratch Pad Buffer.

SPA : Scratch Pad Address.

SPAR : Scratch Pad Address Register.

Sparse Matrix: A matrix of large order in which most of the elements are zero.

SPR : Scratch Pad Register.

Stack : A stack is a Singly Linked-List for which all insertions and deletion (usually all accesses) are made at one end of the list.

State : A particular configuration of storage.

State Transition : A change in the storage configuation.

String : A sequence of zero or more characters.

TBV : Text or Bit-Vector selection bit.

T-end : Top end.

TR1 : Tag Register 1.

TR2 : Tag Register 2.

Tree :	A connected directed graph which is free of cycles.
USD :	Direction code for Tag Manipulation.
VLSI :	Very Large Scale Integration.
VHLL :	Very High Level Language.
WSU :	Word Select Unit.