The Microprogrammed Control of an Associative Processor

C.V.W.Armstrong

Thesis presented for the Degree of Doctor of Philosophy of the University of Edinburgh in the Faculty of Science, November 1976.

Summary

This thesis reports on research conducted into the microprogrammed control of associative processors. A study of the microprogrammed control of processor systems containing multiple resources acting in parallel and cooperating with each other was made. This led to the design and development of a simple associative processor using Digital Equipment Corporation Register Transfer Modules and some additional custom logic. The results and experiences gained from the implementation of the microprogrammed control unit for this associative processor led to a proposal for a more generalised microprogrammed control unit eliminating a number of drawbacks and difficulties discovered with the design approach. A number of optimization techniques were developed for control units based on this design.

Acknowledgement

I would like to thank Dr Tom Buckley, my supervisor, for his guidance and encouragement during this research. His excellent advice and helpful suggestions were very much appreciated.

CONTENTS

CHAPTER 1. INTRODUCTION 1-1
1.1. Statement of Work. 1-1
1.2. Motivation for this Work. 1-1
1.3. Results. 1-4

CHAPTER 2. ASSOCIATIVE PROCESSORS 2-1
2.1. Introduction. 2-1
2.2. A Review of Associative Processing. 2-1
2.2.1. Definitions and Related Research. 2-1
2.2.2. A Survey of Associative Processors. 2-10
2.2.2.1. Fully Parallel Systems. 2-10
2.2.2.2. Bit-Serial Systems. 2-12
2.2.2.3. Word-Serial Systems. 2-14
2.2.2.4. Block-Oriented Systems. 2-15
2.2.3. Associative Processor Technology. 2-15
2.2.3.1. Standard Integrated Circuitry. 2-17
2.2.3.2. Integrated Circuitry in Associative Processors. 2-19
2.2.3.3. Complexity of Control in Associative Processors. 2-20
2.3. Applications of Associative Processing. 2-21
2.3.1. The Various Possibilities. 2-21
2.3.2. Air Traffic Control Conflict Detection. 2-22
2.3.2.1. The Associative Processor Operations. 2-22
2.3.2.2. The System Configuration. 2-25
2.3.2.3. Survey of Air Traffic Control Approaches. 2-26
2.3.2.4. The Chosen Air Traffic Control Approach. 2-30
2.3.3. Information Retrieval Query Processing. 2-32
2.3.3.1. The Associative Processor Operations. 2-32
2.3.3.2. The System Configuration. 2-34

2.3.3.3. Survey of Information Retrieval Approaches. 2-36
2.3.3.4. The Chosen Information Retrieval Approach. 2-37
2.3.4. Initial Design of an Associative Processor. 2-38

CHAPTER 3. MICROPROGRAMMED CONTROL 3-1
3.1. Introduction. 3-1
3.2. Microprogramming Concepts. 3-2
3.2.1. Factors in Microprogramming Design. 3-5
3.2.2. Problems in Microprogramming Design. 3-6
3.2.3. Developments in Microprogramming. 3-8
3.3. MCU Design Principles. 3-10
3.3.1. Separation of Fields. 3-10
3.3.2. Branching Delays. 3-12
3.3.3. Functional Module Control. 3-14
3.4. Cellular Logic and Functional Memory. 3-17
3.4.1. Hybrid Associative Memories. 3-18
3.4.2. Cellular Logic. 3-21
3.4.3. Functional Memory and Array Logic. 3-23
3.4.4. The Design of the Microprogrammed Control Unit. 3-31
3.4.4.1. Some Aspects of the Design Approach. 3-31
3.4.4.2. General Description of the MCU 3-34
3.4.4.3. The Expected Advantages of the MCU. 3-40

CHAPTER 4. THE DESIGN OF AN ASSOCIATIVE PROCESSOR 4-1
4.1. Introduction. 4-1
4.2. Associative Processor System Design. 4-1
4.2.1. Control Flow of the AP. 4-1
4.2.2. Processing Element System Structure 4-3
4.2.3. Associative Processor Instruction Design. 4-4
4.2.4. The Sequential Processor Support. 4-6
4.3. The Air Traffic Control Application. 4-18

4.3.1. General Approach. 4-18
4.3.2. Stage 1. 4-19
4.3.3. Stage 2. 4-23
4.3.4. Stage 3. 4-35
4.3.5. Stage 4. 4-45
4.3.6. The Results. 4-47
4.4. The Information Retrieval Application. 4-49
4.4.1. The General Approach. 4-49
4.4.2. AP Operation. 4-51
4.4.3. The Results. 4-55

CHAPTER 5. THE OPERATION OF THE MICROPROGRAMMED CONTROL UNIT 5-1
5.1. Introduction. 5-1
5.2. Example of the Operation of the MCU. 5-1
5.3. Another Description of the Control Level 3 Microprogram. 5-18
5.4. Advantages Confirmed and Problems Encountered. 5-22
5.4.1. Control Level 0. 5-22
5.4.2. Control Level 1. 5-25
5.4.3. Control Level 2. 5-28
5.5. Microcode Compaction. 5-32

CHAPTER 6. MORE GENERAL APPLICATIONS OF THESE MICROPROGRAMMING TECHNIQUES 6-1
6.1. Introduction. 6-1
6.2. The Problems in the Original Design of the MCU. 6-3
6.2.1. Physical Implementation Problems. 6-3
6.2.1.1. Monophase Operation. 6-3
6.2.1.2. Residual Control. 6-4
6.2.1.3. Limited Memory Capability. 6-6
6.2.1.4. Branching Delays. 6-6
6.2.2. Design Dependent Problems. 6-7
6.2.2.1. Microprogram Sparsity. 6-7
6.2.2.2. Microprogram Description. 6-8
6.2.2.3. Microroutine Termination. 6-9
6.3. Generalised Functional Memory. 6-9

6.3.1. A Proposal for a Generalised Functional Memory Design. 6-13
6.4. A Generalised Microprogrammed Control Unit. 6-21

CHAPTER 7. MICROINSTRUCTIONS FOR MULTIPLE RESOURCE SYSTEMS 7-1
7.1. Introduction. 7-1
7.2. Microinstruction Formats. 7-1
7.2.1. Multi-Level Encoding. 7-5
7.2.2. Separation of Microinstruction Sections. 7-7
7.2.3. The Use of Activation Microorders. 7-9
7.3. The Microinstruction Format as a Focal Point. 7-11
7.4. The Assignment of Microorders to Microinstructions. 7-14
7.5. The Generality of this Approach to the Microprogrammed Control of Associative Processors. 7-22
7.6. Other Research Problems. 7-24

APPENDIX A A-1
A DESCRIPTION OF THE ASSOCIATIVE PROCESSOR IMPLEMENTATION
1. Introduction. A-1
2. Hardware Description. A-1
2.1. Control Console. A-1
2.2. Printed Circuit Board Distribution. A-3
2.3. Control Flow Diagrams. A-3
2.4. MCU Phase 1 Operations. A-17
2.5. MCU Phase 2 Operations. A-19
2.6. Mode of Interpretation Operations. A-20
2.7. PE-MCU and PE-PE Data Paths. A-22
2.8. Status Section. A-22
2.9. Miscellaneous Circuitry. A-22
2.10. Merging. A-24
3. Software Description. A-25
3.1. RESET n A-25

3.2. LDA x A-25
3.3. LSHFT n A-26
3.4. RSHFT n A-26
3.5. STA x A-26
3.6. RSTVR A-26
3.7. STZ x A-27
3.8. RESTS n A-27
3.9. TRANS A-27
3.10. RECV A-27
3.11. ZAC A-27
3.12. TEST ER,x A-28
3.13. TEST SR,x A-28
3.14. ARSHFT n A-28
3.15. READ x,y A-28
3.16. WRITE x,y A-29
3.17. ADD x A-30
3.18. SUBT x A-30
3.19. MULT x A-31
3.20. ADDD x A-31
3.21. SUBTD x A-32
3.22. SEEK x,y,a;zzzz A-32
3.23. JZSR ; xxxx A-33
4. The Application Programs. A-34
Application 1. A-35
Application 2. A-58
5. Sample Output. A-74
Application 1 lineprinter output. A-75
Application 1 typewriter output. A-81
Application 2 typewriter output. A-83

APPENDIX B B-1
FUNCTIONAL MEMORY TECHNIQUES APPLIED TO THE MICROPROGRAMMED CONTROL OF AN ASSOCIATIVE PROCESSOR (published paper)

References R-1

Declaration

---o0o---

CHAPTER 1

INTRODUCTION

1.1. Statement of Work.

This thesis reports on research conducted into the microprogrammed control of associative processors. The research can be divided into three sections :

(1) A study of the microprogrammed control of processor systems containing multiple resources acting in parallel and cooperating with each other.

(2) The design and development of a simple associative processor using Digital Equipment Corporation Register Transfer Modules and some additional custom logic. This design employed many of the strategies considered in (1).

(3) The results and experiences gained from the implementation of the microprogrammed control unit for the associative processor led to a proposal for a more generalized microprogrammed control unit eliminating a number of drawbacks and difficulties discovered with this approach. A number of optimization techniques were developed for control units based on this design.

1.2. Motivation for this Work.

The problem of controlling parallel processors and

other multiple resources is becoming more important. This is not due solely to the development of multiprocessor systems and networks. There is currently a rapid expansion in the use of Large Scale Integration (LSI) circuitry. Most of this expansion is to be found in the growth of microprocessor applications and the use of microprocessors as components in logic systems of all types. Attempts are already being made to develop processors using microprocessors as components, that is, to develop multimicroprocessor systems (Tjaden, 1976)(Dejka, 1976). The problems with controlling the overall operation of multiprocessor systems will now be found at the processor control unit level. Because of the limited knowledge of control requirements, flexibility in the provision of control operations is required with the possibility of easily changing the control unit design if necessary. A microprogrammed control unit is an obvious choice for managing such a system. Such a choice, however, can lead to microprogrammed control units of increasing complexity.

There are other trends in computer system design leading to increased complexity at the control unit level. There is, for example, a trend to reduce the complexity of software. Low level software operations are pushed down into the hardware and more high level instructions are made available to the user (Weber, 1967)(Wilner, 1972a, 1972b)(Lawson, 1975). This trend is found in current research in computer architecture where researchers are attempting to provide users in an experimental setting with instruction sets of minimum complexity and with the

availability of well-formed high level instructions, thus encouraging the writing of efficient, structured, easily communicated, maintainable and reliable programs (e.g. Sylvain and Vineberg, 1975).

A similar trend can be seen in the design of hardware. A reduction in hardware complexity is being attempted by the use of microprogramming with the added advantages of changeability and computer range compatability. Hardware is being designed as simple modular units with simple bus structures providing the interconnections between these modules (e.g. PDP11, META4, Interdata Model 85).

The microprogrammed control unit is being required more and more to assist in the reduction of complexity of both hardware and software, and this leads to increases in its own complexity on two fronts. Possible increases in complexity can be manifested by :

(1) increasingly complex microorders and the corresponding decoding of these microorders. Vertical microprogramming would require large microprograms with the complexity lying in the hardware circuitry used in the microprogrammed control unit.

(2) the masking and shifting operations associated with decoding and interpreting the instruction operation codes. Special hardware has to be provided for this or the resources for processing the user's data must be borrowed.

(3) the longer microinstructions which may be

required to manage the large number of hardware resources. Here complexity can be found in any requirements for a large number of microinstructions for each microroutine.

(4) the provision of a large number of control registers. In addition to those registers required to store the instruction and the instruction address, general purpose registers and special registers to store the status of various hardware and software components of the system are required.

A consideration of this problem of increasing complexity in the microprogrammed control unit suggests that various nonconventional approaches should be studied in the hope of developing a design providing simple short width user instructions and complex long width microinstructions, perhaps generated dynamically for each user instruction.

The investigation that followed was motivated by the hope that suitable microprogrammed control units could be designed where increased complexity was reduced by a number of simple changes in the method of microinstruction generation.

1.3. Results.

A concrete example to investigate such control problems was required and the design of an associative processor was chosen as suitable for study. In addition, two application problems were chosen to demonstrate the operation of this associative processor.

The first application was air traffic control conflict detection. Such as example is suitable for parallel processing. Associative operations can be used to select those aircraft requiring further processing.

The second application chosen was information retrieval query processing. This was a simple example and was intended to be complementary in its requirements for the associative processor. Whereas the first application required more processing at the processing element (PE) or data structure level, the second application required more processing at the microprogrammed control unit (MCU) or control structure level. Selection of those PEs satisfying some criterion was performed at the MCU level.

Chapters 2 and 3 relate to the initial design of the associative processor and its microprogrammed control. Chapter 2 reports on associative processing research and the development of the requirements for the associative processor to be built. Chapter 3 treats microprogramming and covers in particular the microprogrammed control of multiple resource systems. The design philosophy to be used in the implementation of the associative processor is developed here.

Chapters 4 and 5 report on the hardware and software implementation as it was finally developed. Chapter 4 covers the design and development of the associative processor. The overall system design and the application programs are covered in sufficient detail to demonstrate the capabilities of the processor. Chapter 5 covers the microprograms written and the operation of the microprogrammed

control unit in more detail. The advantages of this approach to microprogramming will be pointed out and a number of problems encountered will be identified.

Chapters 6 and 7 describe the findings and conclusions which were developed based on the actual building of a prototype associative processor. Chapter 6 is concerned with generalizing the experiences gained from this work and proposes a microprogrammed control unit alleviating a number of the problems found in the original design. This microprogrammed control unit is suitable for the control of a more general set of processing elements, for example, a set of elements for which the interconnection structure is less restrictive. Various optimization techniques for reducing the size of the functional memories are considered at this point. Chapter 7 considers the interaction and arrangement of microinstruction fields and seeks to develop a more systematic approach to the assignment of microorders to microinstruction fields when multilevel encoding is involved. This assignment was a major design problem for the associative processor, and a theory could be useful in the design of other types of microprogrammed control units.

Finally, the appendices include design diagrams and functional flow diagrams for the associative processor, together with the application programs and sample output. Also included is a paper reporting on this work which was presented at the Second Annual Symposium on Computer Architecture, at the University of Houston, Texas in January 1975.

CHAPTER 2

ASSOCIATIVE PROCESSORS

2.1. Introduction.

This chapter provides a brief survey of associative processing research and develops the requirements for the associative processor which was constructed.

Definitions of the terms to be used later when referring to associative processing are given here. A survey of different associative processors is given and consideration is given to the various technologies proposed and employed for them.

Among the various applications of associative processing, air traffic control and information retrieval are reviewed. The requirements for implementing programs for conflict detection and query processing are developed from this review and the initial design of the associative processor to meet these requirements is described.

2.2. A Review of Associative Processing.

2.2.1. Definitions and Related Research.

Associative memories or content-addressable memories are storage devices in which data is accessed by using a subset of the data. This subset is compared with a corresponding subset of each word in the associative

memory and the word in which a match occurs is selected for access. Fig.2.1 shows a typical associative memory. Note that the word selected by the associative operation can be used for either accessing data already stored in the word or for writing new data into the word. The subset of the data used in this selection operation is called the key.

An associative memory may have multiple responses to the key provided, that is, the key may find a match in more than one word in the associative memory. Various techniques are available to resolve the multiple-response problem if a single word is to be accessed (e.g. Foster, 1968).

Such multiple responses become very useful when each data word has some processing power associated with it. That is, the circuitry storing the data word has some logic capability. The associative operation can then select a subset of all the data words in the memory on which a series of data processing instructions are to be executed. For this purpose, each data word may be divided into a number of fields, and each data operation may operate on fields of the same data word or fields in different data words. When such processing is possible, the associative memory can be called an associative processor. An example of such an associative processor is shown in Fig.2.2.

An associative processor is a special type of single instruction stream, multiple data stream processor (Flynn, 1972a), that is, a parallel processor. The processing

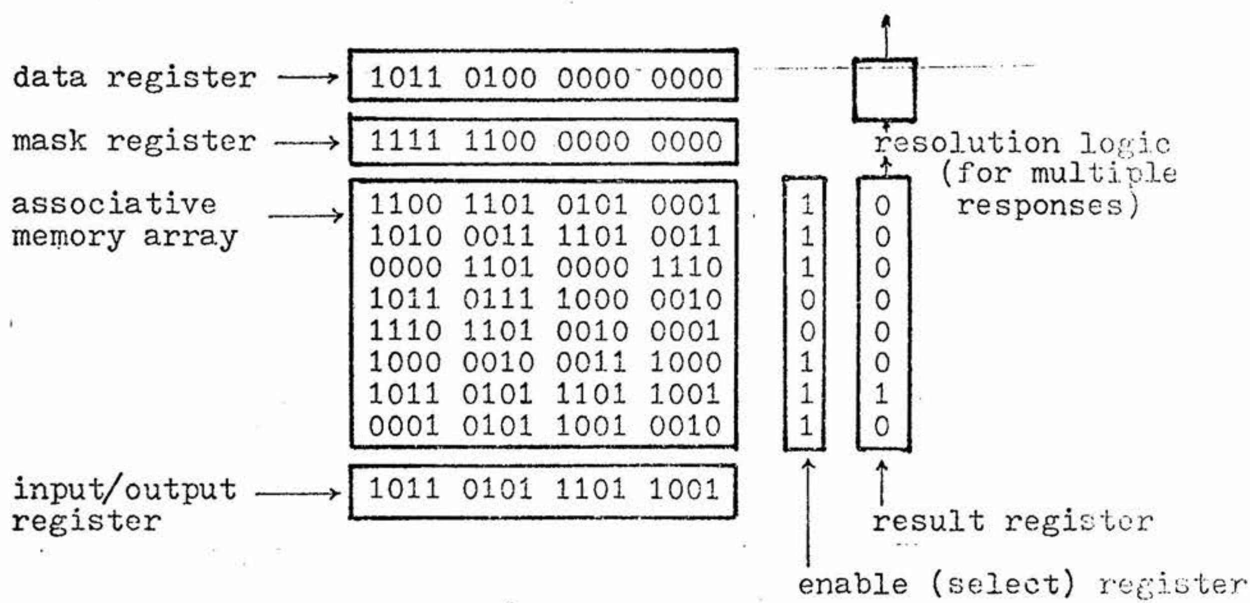


Fig.2.1. A Typical Associative Memory

Notes :
(1) The mask register specifies the subset of bits to be used in the match operation.
(2) The enable register specifies those words to be considered in the match operation.
(3) The bits set in the result register show those words which have had a match.
(4) The resolution logic selects a matching word in the associative memory if there is more than one match.
(5) If there is more than one match, the input/output register can show the ORing of the matching data words.
(6) When used for input, the input/output register's value can be written into all words in the memory with a bit set in the enable register.

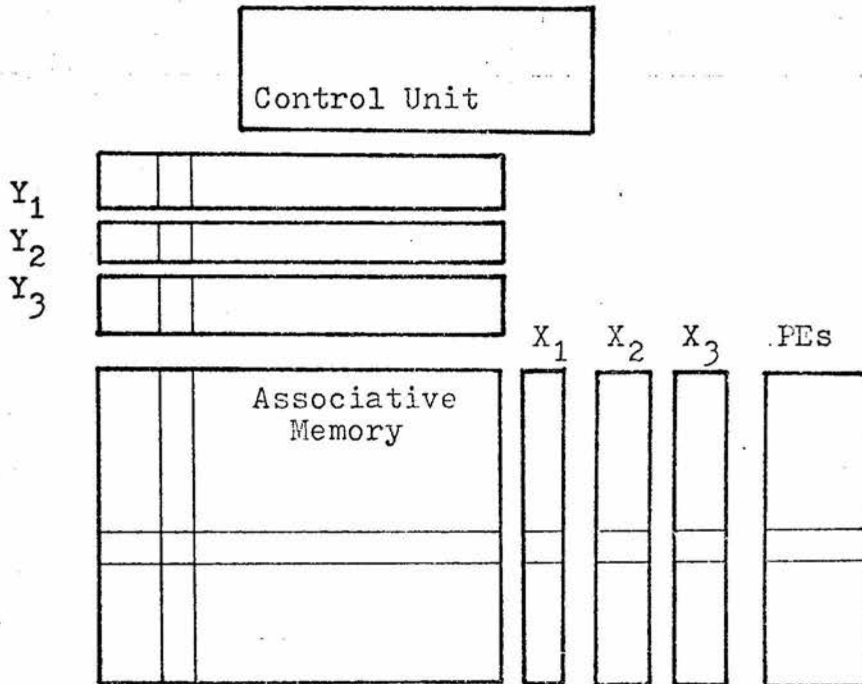


Fig.2.2. A Typical Associative Processor

Notes : (1) The X registers can be used for a number of different purposes including use as enable, match, and mask registers.
(2) The Y registers can be used for such purposes as mask registers and input and output registers.
(3) The PEs (processing elements) can have control of the X registers and control of the corresponding words of the associative memory.
(4) In addition, the PEs can have control for processing either horizontal or vertical words including the use of the Y registers.

elements (PEs) can be considered to contain one or more of the data words making up the memory array. Their complexity can vary. There have been proposals (Radoy and Lipovski, 1975) for PEs capable of selecting from a number of different instruction streams and this may be useful when the instruction stream can be broken into segments of more or less equal length. An associative processor with PEs with this capability would be a special type of multiple instruction stream multiple data stream processor. In the case of the single instruction stream, it is necessary to enable those PEs using one instruction stream segment and then disable this subset and enable another subset of PEs for the second instruction stream segment. This process is continued until all instruction stream segments for all PEs upto some common decision point have been processed. The single instruction stream version is slower but the PEs are correspondingly simpler and the instruction bandwidth is narrower. In addition, the selected subsets for instruction stream segment execution do not have to be mutually exclusive and this is particularly useful in an example such as query processing where we have a process of selecting a subset of the PEs meeting some criteria and then selecting a further subset of this subset meeting different criteria, and so on. Thus we will only consider PEs which can be enabled or disabled and which execute a single common instruction presented to the whole set of PEs.

In early attempts to exploit the coming potential of large scale integration (LSI) fabrication techniques,

much attention was given to cellular logic and logic-in-memory capabilities. Cellular logic is logic with a regular and repetitive structure (Kautz, 1971b). Such circuitry, because it is sequential, has a limited memory capability. When the storage aspects are emphasised in an implementation, the array is called a logic-in-memory array (Kautz, 1969).

Functional memory was a term used originally by Flinders, Gardner, Llewelyn and Minshull (1970) for a particular implementation of logic-in-memory arrays which could be used as a direct replacement for logic at the functional level. Such memory arrays could be a substitute for an adder or a multiplier. The "cells" making up the functional memory were particularly simple. Fig.2.3 shows the structure of functional memory in a diagrammatic form.

Functional memory has many of the attributes of associative memory. Each cell, for example, has the ability to perform a "match" operation. A signal coming down in the vertical direction, as in Fig.2.3. could be compared with an internal value stored in the cell, and, on a match, a horizontal signal could be propagated. This is an associative capability. In addition, the cell could have a "don't care" state where any horizontal signal was propagated without regard to the vertical signal.

The above was considered the select phase of the cell. In addition, there was a read phase. In this case, a horizontal signal (generated by the associative operation of the select phase) caused a vertical signal to be

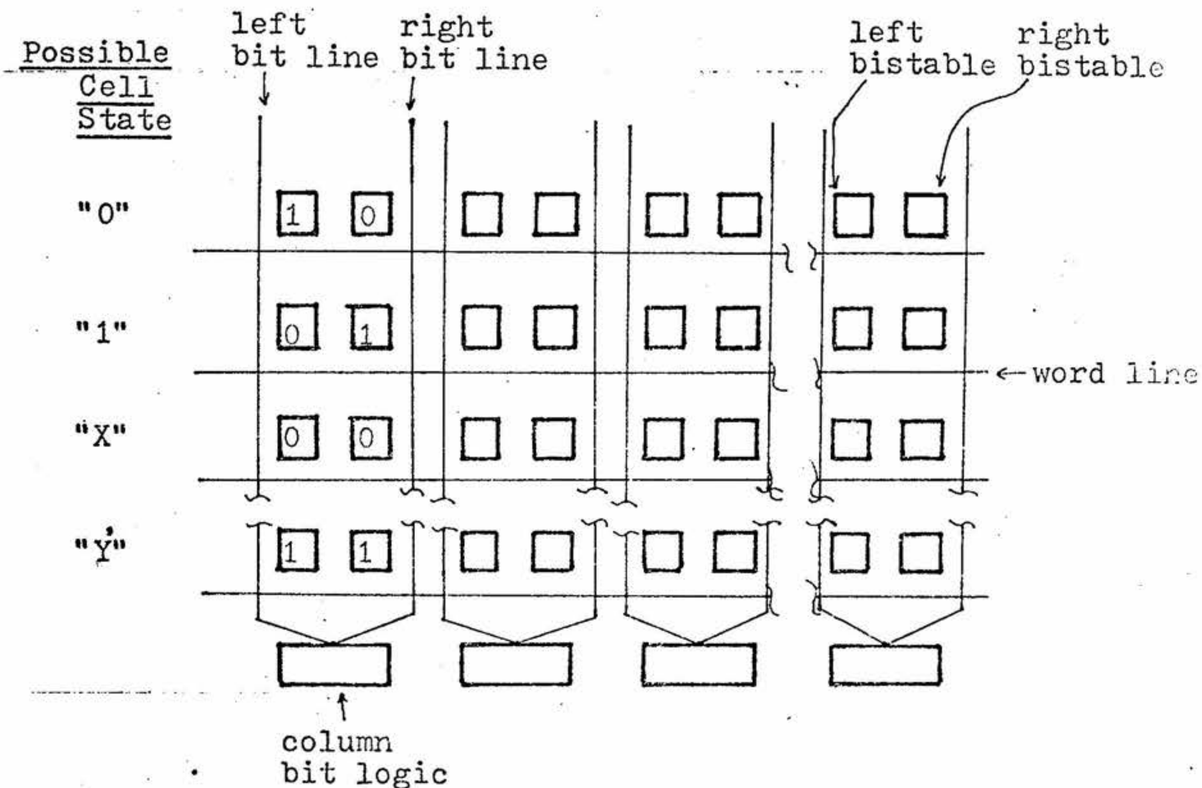


Fig.2.3. Functional Memory Array

Notes : (1) "X" is the "don't care" state and in the search phase always propagates a signal.
(2) "Y" can be used to inhibit the output of any signals.

(The above diagram is derived from Gardner, 1971, Fig.10.)

generated and propagated downwards (vertically) depending on the state of the cell.

The same cell could be used for both selecting and reading operations. This slows down access and it is more useful to partition the cells and allow one block of cells to provide the select phase and another the read phase. Such considerations have been the subject of a number of British Patents. Table 2.1 is a list of some of these patents.

For example, one of the considerations of these patents is the problem of malfunctioning cells or groups of cells. One proposal considered is the duplication of memory modules and a comparison check (Patent No. 1 265 013).

Another problem with providing functional modules is the requirement for custom logic. Array logic can suffer from the high cost of developing new integrated circuitry and the capability to use integrated circuitry which has already been developed by industry could lead to lower development costs and greater acceptance by system builders and system users.

Further developments in functional memory have pointed the way to using more conventional existing integrated circuitry. These developments are embodied in U.S. Patent No. 3,761,902 entitled "Functional Memory using Multi-State Associative Cells", No. 3,644,906 entitled "Hybrid Associative Memory" and No. 3,593,317 entitled "Partitioning Logic Operations in a Generalized Matrix System".

Patent No.	
1 127 270	Basic 3 state cell
1 186 703	Basic associative store
1 186 704	Select next good word
1 218 406/7	Interconnection of stores for processing by table look-up
1 230 834	Character recognition (Braverman)
1 265 015	Duplex processors for error checking
1 265 645	Basic operating module replacement
1 234 484	Address store
1 231 908	Search argument stores
1 233 484	Mask store. Pipelining
1 264 095	Multiplex channel
1 257 760	Multiple control stores
1 229 717	Auto sequencing store
1 265 013	Siemese basic operating module. Mini basic operating module
1 310 772	Line multiplexer
1 289 249	On demand buffers
1 281 387	2-state cells
1 233 290	Bit line functions
1 265 014	Configuration register. Character recognition
1 308 024	Graphic display
1 268 283	Connect module
1 264 096	Selector channel
1 314 140	Relocation by functional memory
1 314 486	Microprogram control system
1 334 654	Basic logic module
1 317 714	Data handling systems

Table 2.1. British Patents Related to Functional Memory

(courtesy of P.L.Gardner, IBM United Kingdom Laboratories Ltd)

A <u>hybrid associative memory</u> is a memory employing both associative and non-associative addressing. Part of the address supplied for accessing data is used to select one of a number of associative memories. The remaining part of the address field is used to perform an associative search using the selected associative memory. See Fig.2.4 for an example.

Weinberger, in papers referring to the hybrid associative memory concept (1970, 1971), has demonstrated that a continuum exists with coordinate-addressed memory at one extreme and associative memory at the other. Hybrid associative memories occupy positions between these two extremes. They are important as an attempt to overcome the high cost of large customized associative memories by using more standard techniques (such as decoding) and smaller associative arrays.

2.2.2. A Survey of Associative Processors.

An excellent survey and bibliography on associative processing has been given by Parhami (1973). In treating the architectural concepts of associative processors, the following classes are identified.

2.2.2.1. Fully Parallel Systems.

These can be divided into word-organized systems and distributed-logic systems.

Word-organized systems are systems in which each word

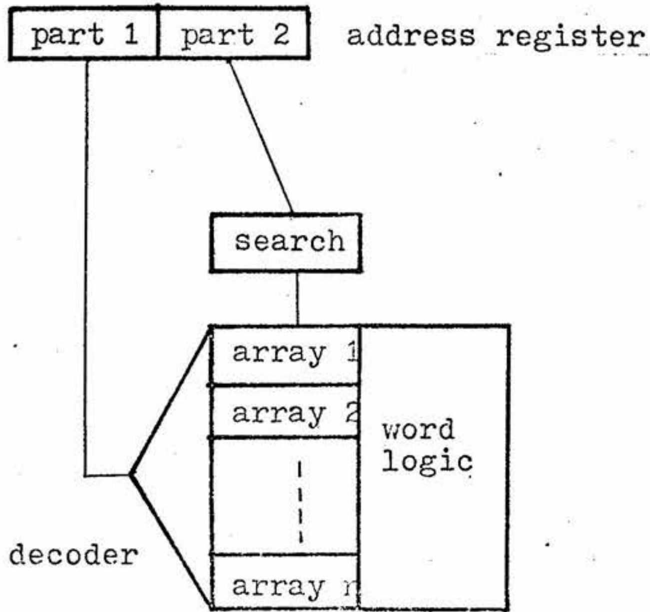


Fig.2.4. Hybrid Associative Array

Notes : (1) Part 1 of the address register is used for normal addressing using a decoder.
(2) The array selected by using part 1 of the address is associatively searched using part 2 of the address register.
(3) When the array size reduces to one word, the hybrid associative memory degenerates into a normal coordinate-addressed memory.
(4) When the decoder is eliminated, a standard associative memory remains.

(This diagram is adapted from Weinberger, 1969.)

has associated with it some processing capability. This is a very expensive implementation and has been the motivation behind research into cryogenic techniques (e.g. Rosin, 1962). The failure in developing feasible realizations of word-organized systems has prompted consideration of other organizations in which the amount of processing power associated with each word is substantially reduced.

The distributed-logic approach leads to a design based on a number of identical cells. Each cell can store data and compare this information with data presented to it, either from a control unit or a subset of the other cells. Customized logic is required to realize an array of these cells (e.g. Kautz, 1971a; Kautz and Pease, 1971).

2.2.2.2. Bit-Serial Systems.

Bit-serial systems have associated with each word, the processing capability to operate on one bit at a time. This is a reduction of the much more powerful processing capability attempted with word-organized systems. This approach has led to practical implementations of associative processors, including the Goodyear Aerospace Corporation STARAN associative processor (Rudolph, 1972; Batcher, 1972).

Fig.2.5 shows the structure of one of the STARAN associative arrays. Note that the bit processing capability for each word of the array is emulated by the operation of the control unit, using the X, Y and M registers (Batcher, 1973; Batcher, 1974). These registers are 256 bits long with one bit for each word in the memory array. Together,

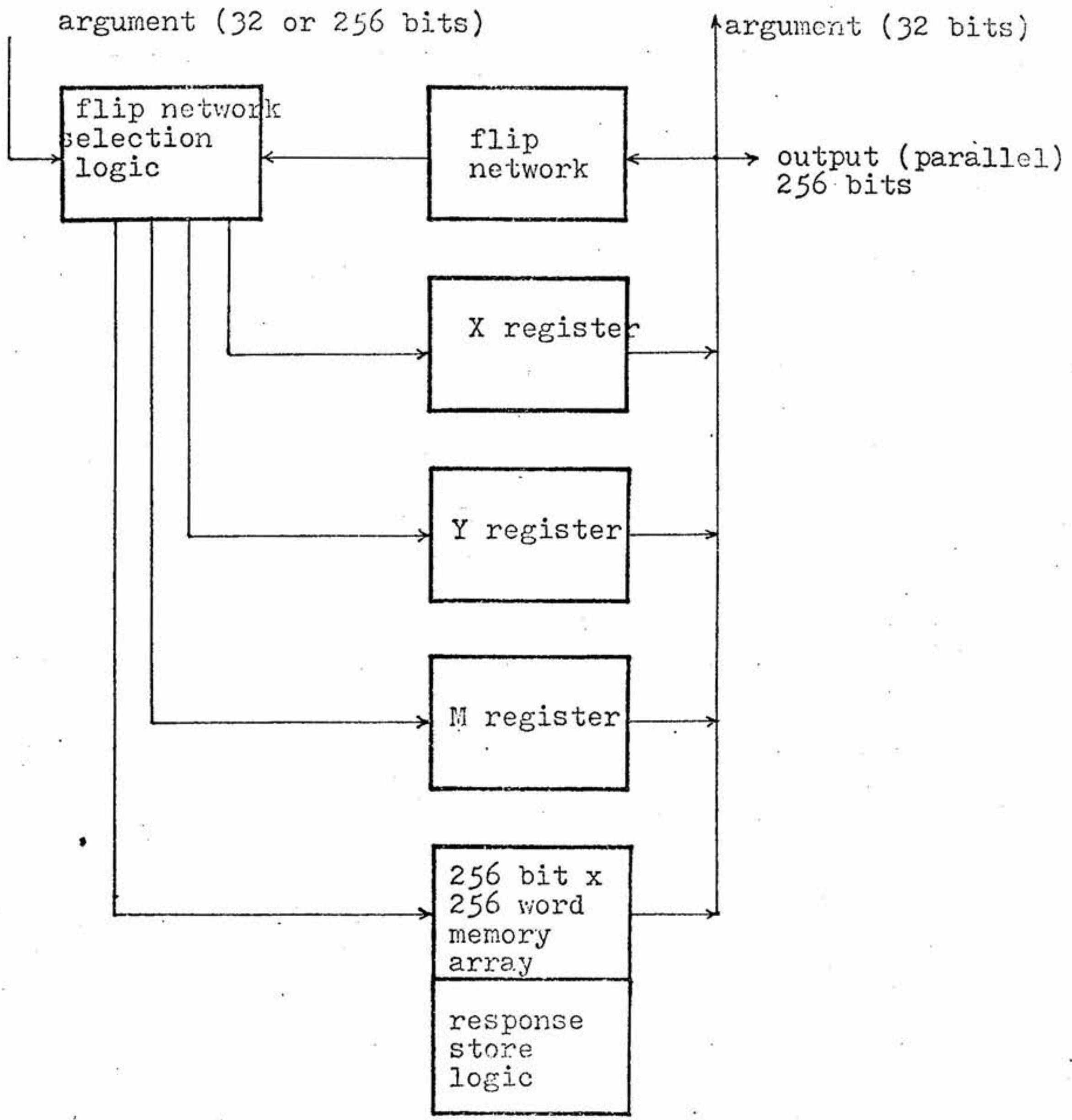


Fig.2.5. STARAN Associative Array

(See Thurber and Wald, 1975. Copyright 1975, Association for Computing Machinery. Computing Surveys, Vol.7, No.4, December 1975. Reprinting privileges were granted by permission of the Association for Computing Machinery.)

Notes : (1) The flip network is the interconnection network through which data passes before being stored in the memory array or one of the registers.

(2) Each of the registers is 256 bits long.

these registers can be considered as the processing elements associated with the memory array. Between them and the memory array is an interconnection structure allowing access to stored data in a number of different modes. These modes include the processing of bits from the same word and the same bits of a number of different words (Batcher, 1975).

This approach is particularly economical with a large word size since the cost of the processing element associated with the word can be spread over the number of bits in the word. With respect to speed, it is important to have a large number of these words. Bit-serial operation will be slow and to compensate for this, the amount of parallelism must be increased.

Between fully-parallel systems and bit-serial systems, are systems in which the processing elements act in parallel on a number of bits less than the word-length. A byte processor is one possible example.

2.2.2.3. Word-Serial Systems.

It is possible to microprogram a normal processor to perform associative operations. Parallel operations can be provided to the user, even though at the microprogram level the processor is acting upon each word in its memory sequentially. The main advantage of this technique is the reduction in the number of user instructions required to perform an operation on a number of different data items. (Dalrymple, 1975)

Various software techniques can also provide the illusion that associative hardware operations are being executed. An interpreter employing hashing techniques has performed such pseudo-associative operations (Ash and Sibley, 1968).

2.2.2.4. Block-Oriented Systems.

Block-oriented systems are based on some of the characteristics of mass-storage devices such as head-per-track disk units. The proposal is to provide some logic per head and process selected bits as they come under the heads in parallel. Such approaches attempt to exploit the high data rate of circulating memories (Slotnick, 1970; Parhami, 1972). Similar to fully-parallel systems, customized logic is required and more development work has to be done before a practical implementation is feasible.

2.2.3. Associative Processor Technology.

Early research in associative processing was very much concerned with cryogenic technology. It was hoped that in a cryogenic medium, fast economic associative cells could be developed on a large scale. Although the design ideas were fully developed (e.g. Rosin, 1962), the technology failed to live up to expectations.

Considerations on the best ways of exploiting large-scale integration fabrication technology emphasised

cellular logic as a promising approach. Kautz (1969) has pointed out that cellular logic-in-memory arrays had the advantages of :

"1) flexibility in function to reduce the required number of different types of subarrays,
2) testability,
3) fault accommodation,
4) subarray interconnectability (due to the limitations on the number of external terminals),
5) high logical performance (that is, a large processing or logical capability per chip),
6) ease of logical design,
7) low power level and high speed, and
8) ease of functional decomposition of a system into chip-size 'packages'."

The use of cellular logic has been considered for a variety of special applications, including associative processing. Kautz (1971) has considered one such associative array implemented in cellular logic, in which each cell can have one of eight possible modes of operation. In conclusion, Kautz writes :

"Architects for the next generation of large, general-purpose machines should give special attention to the best ways in which logic-in-memory arrays can be applied to capitalize upon the inherent advantages of large-scale integration. In particular, the use of these arrays allows a graceful transition between conventional and large-scale-integrated computer organizations. In addition, logic-in-memory arrays appear to offer a means whereby the drastic improvements in overall performance promised by large-scale integration may actually be achieved. This approach is also completely consistent with several widely accepted goals of advanced computer design : increased parallelism at all levels (logic circuitry, subsystems, and overall program execution), more extensive realization of software functions by means of hardware, more distributed control, and more extensive use of stored tables and other variable-data structures for paging, microprogram-control, equipment allocation, and other related operations. Logic-in-memory arrays also appear to offer some distinct advantages for the design of computers that are intended to carry out list processing as well as conventional data-processing and numerical operations."

Cellular logic has not lived up to expectations,

even though extensive proposals have been made for system organizations based on this approach. The major reason for this lack of success, despite the great optimism and excitement generated by this research, seems to be the requirement for extensive customized logic. It is essential that standard integrated circuitry should be exploited as much as possible. Functional memory as a system technology (Flinders, Gardner, Llewelyn and Minshull, 1970) which was inspired by the earlier work on cellular logic, has led to some practical implementations (Logue, Brickman, Howley, Jones and Wu, 1975). This may be due to the use of more conventional integrated circuitry.

2.2.3.1. Standard Integrated Circuitry.

We mean by "standard integrated circuitry", those integrated circuit packages that are already available from integrated circuit manufacturers, or can be expected to be available as a natural improvement of existing circuits. It is assumed that new circuits will follow rather predictable and conservative trends. Even microprocessors are a natural development of arithmetic logic unit chips and calculator chips, with their high market potential. It would be unrealistic to assume any significant innovation in circuitry which was not related to the larger markets of logic equipment.

Allen and Pearcy (1969) have considered the implications of LSI and the major problems involved. One such problem is the part number problem, where a

large number of different integrated circuits is available. Such a large number leads to higher development costs per circuit since computer equipment would require smaller numbers of different circuits rather than larger numbers of only a few types of circuit. To decrease the number of different circuits, more generalised circuits are required that can be "personalized" for a particular application.

The cost of a circuit is also related to its complexity. Complexity-reduction problems suggest that regular structures are a good organization for LSI circuitry and that the minimisation of external connections is also important. Such considerations have been motivating factors for the development of functional memory (Flinders, Gardner, Llewelyn and Minshull, 1970), which has a regular structure and also attempts to eliminate the part-number problem. Functional memory minimises the number of external connections by using the maximum amount of decoding on each chip.

The major drawback with functional memory is that it is not standard integrated circuitry but customized circuitry. Its cost could be closely related to that of some uncommitted logic arrays (ULAs) which are "personalized" in the final stages of manufacturing. There is one type of standard integrated circuitry that can satisfy the requirements of low complexity, minimal external connections, high density and regular structure, namely memory chips. Standard read-only memory chips (ROMs) and random-access memory chips (RAMs) are sufficiently general that the part-number problem is significantly

reduced. They could be easily "personalized" for a particular application by loading specific data.

Memory chips may not be as fast as logic chips but they are becoming faster. A major problem with their use is that a large amount of storage may be required to provide the same logic capability as a logic chip. Minimisation of the storage for this logic capability is a significant design problem.

To solve this problem, the types of logic capability have to be investigated. The possibilities of encoding the "logic information" to minimise the storage requirements must be considered and various interconnection structures for the memory chips must be studied. We shall come back to this topic in Chapter 6. At this point, we consider in particular how standard integrated circuitry can be used in associative processor implementation.

2.2.3.2. Integrated Circuitry in Associative Processors.

The major implementation of an associative processor, the Goodyear Aerospace Corporation STARAN, is built using standard logic circuits and a plated-wire memory. Discussion of these design choices are given by Feldman and Fulmer, (1974) and Fulmer and Meilander (1970). Another implementation using standard circuit building blocks is the Honeywell Associative Parallel Processing Ensemble, HAPPE (Marvel, 1974).

Although standard logic is now being used in both associative and parallel processors, the complexities of control require a large number of circuits to be used. In STARAN, an array of 256 associative words uses 2,500

integrated circuits whilst the circuits required to support and control this array number 6,500 (Feldman and Reimann, 1974). Another example for a parallel processor, is ILLIAC IV, where 50% of the cost of the hardware can be found in the control unit. Whereas each PE is implemented in a regular structure with 10K logic gates, the control unit is implemented with a random logic structure of 100K logic gates (Thurber and Wald, 1975).

We now consider in particular those factors contributing to complexity in the circuitry of the control section of associative processors.

2.2.3.3. Complexity of Control in Associative Processors.

One of the problems that has received consideration in associative processing is the accessing and alignment of data (Lawrie, 1975). That is, before data can be processed in a PE, it may have to be accessed from other PEs. After receiving this data, realignment may be required before any data operations can proceed. The accessing and alignment of data is supported in STARAN by the scrambling and skewing of data in storage. Bits of the same 256-bit data word are distributed over all the memory words in the 256-word array. They are accessed through an XOR scrambling network (Batcher, 1975). This allows standard MSI circuitry to read the data in various storage modes, including word-slice, bit-slice and byte-slice.

Accessing of data in any multiprocessor system can

become inefficient due to contention for memory resources. Studies of memory contention in multiprocessor systems have been done by several researchers (e.g. Juliussen and Mowle, 1973; Bhandarkar and Fuller, 1974). This may be even more complex if alignment of data is required when processors have to access data in other processor modules. This problem is a major factor in designing multiprocessor systems (Searle and Freberg, 1975). Other problems derived from this include the interconnection and bus structure design problems (Thurber, 1972). The interconnection structure can also be influenced by any requirement for fault-tolerance (Parhami and Avizienis, 1974).

2.3. Applications of Associative Processing.

2.3.1. The Various Possibilities.

Thurber and Berg (1971) have considered the various applications of associative processors, and have proposed that the problems that are most suitable are those which

(1) have a large number of data sets,
(2) need to perform the same computation on a large number of these data sets,
(3) are parallel in nature,
(4) require a large amount of searching or associative operations,
(5) require the processing of data sets that are selected by some criteria of the data sets, and

(6) require the processing of data selected on the basis of previous results.

As examples of suitable applications, they cover pattern recognition and radar data processing.

The choice of air traffic control conflict detection was made because it fitted in with many of the points above. In particular, it afforded an opportunity to consider the generalised associative operation, where instead of matching a key supplied by the control unit with data in each PE, each PE supplied a key to its neighbour.

The choice of information retrieval query processing was intended to provide an application which required complementary operations of the associative processor. Rather than emphasising the parallel and associative operations at the PE level, operations at the microprogrammed control unit (MCU) level were emphasised. This choice will be considered again later.

2.3.2. Air Traffic Control Conflict Detection.

2.3.2.1. The Associative Processor Operations.

The basic system structure was developed as shown in Fig.2.6. This preliminary specification evolved from considering the resources available for building the system. The sequential processor (SP) was an Interdata Model 74. The programming language HAL70, supported by the University of Edinburgh Department of Computer Science,

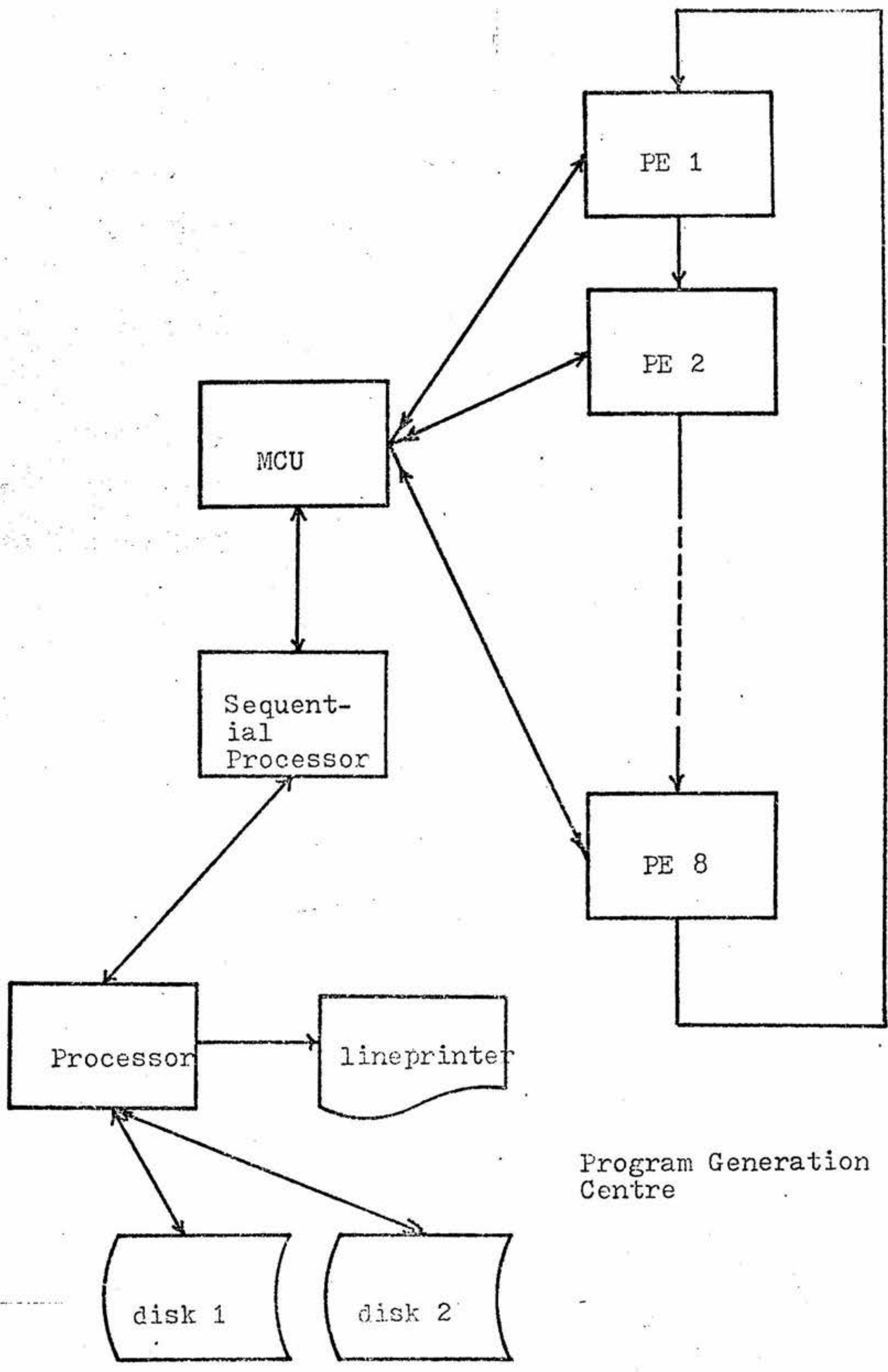


Fig.2.6. The System Configuration.

was used to program this machine. The program generation centre used a multi-user operating system called ISYS to support program development. The associative processor (AP) was built using DEC RTMs and additional custom logic (Bell and Grason, 1971). This was housed in standard cabinets used in the Department, including patch panels, control panels and power supplies.

In the early stages of design, a simulation program written in ALGOL was exercised on the DEC PDP10 computer available in the Department. The first microprogram to be developed was written at this stage. Such software simulation was demonstrated to be both time-consuming and tedious in comparison to the hardware implementation. Microprograms were easier to develop on the RTM version and provided a more graphic effect.

The choice of method to demonstrate conflict detection was heavily dependent on the operations which could be available in an associative processor, and their relative cost in implementation, complexity and speed. It was obvious that the MCU would have to assume the responsibility for a number of different tasks :

(1) control of each PE in parallel,
(2) intra-PE communication, so that data could be moved from one PE to another,
(3) sequential processor - associative processor (SP-AP) I/O control so that the small size of the AP-PE array is not obvious, and a larger "virtual" array is presumed by the programmer, and
(4) interface control between the AP and the SP for both data and instructions.

Special instructions would have to be developed to support these tasks. The usual associative processor operations were assumed. Because of the limited size of the control memory of the MCU, only those operations necessary for this application (and the second application) would be coded. In addition, only a subset of the full set of microprograms could be loaded at any one time.

In general, arithmetic operations would be required at the PE level since many computations are applied to data streams in parallel.

The standard associative operation is to have a key (ANDed with a mask) compared with all words of the PE array. The generalization (as already pointed out) is to allow each PE to generate a key that can be matched to words in other PEs. This powerful associative operation was available in the STARAN AP. It was considered particularly necessary when developing algorithms for conflict detection which employed a gross screening technique of dividing the "air space" into cells and storing a cell-map in the PE-array. Such a generalized associative operation could be used for checks on each cell and neighbouring cells which may "contain" aircraft.

2.3.2.2. The System Configuration.

Apart from the practical resources and the hardware design constraints in using Digital Equipment Corporation Register Transfer Modules, the application considered here had a significant influence on the system configuration.

Fig.2.6 has given a brief idea of the general system configuration. Limitations in AP memory space meant that the AP memory must really be provided and supported by the SP memory. In particular, the instructions for the AP would be provided by the SP support program. The PE-array, which consists of 8 PEs each storing 256 bits as 16 words of 16 bits, would be loaded and stored from the SP.

The AP would perform as a special I/O device connected by a serial link to the SP. There would be separate channels for input to the AP and output from the AP, with a 265 Kbaud serial interface for each channel (Lindsay, 1972; Lindsay, 1974). The SP would provide data and instructions to the AP and could continue with other activity until such point when it was required to wait for data or status from the AP. It would be possible for both SP and AP to be active at the same time.

2.3.2.3. Survey of Air Traffic Control Approaches.

Example 1.

Goodyear's air traffic control application was divided up into the following sections :

beacon tracking
radar tracking
conflict detection
conflict resolution
terrain avoidance
automatic voice advisory

digital display processing

Consideration of various strategies for handling conflict detection determines the various roles that the SP and the AP should play. The AP was found in this case to be most suitable for the bulk of the processing required for conflict detection (Rudolph, 1972). An initial evaluation of STARAN involved a two-dimensional air space, and this seemed an appropriate simplification to follow in this study. A preliminary review of STARAN's capabilities has stressed the importance of filtering out the pairs of aircraft that can not possibly be on collision courses (Dietzler, 1972). In this case, however, the parallel algorithms for predicting an aircraft's position and potential conflict area seemed to have been stressed.

Example 2.

Downs (1973) describes a simple example of conflict detection, where operations are performed for each pair of aircraft. Let

D_r = relative distance at initial time

S_r = relative speed at initial time

T = lookahead time

and MD = miss distance

Then $MD = D_r - TS_r$

and ($MD <$ some criterion) implies a conflict danger.

Straightforward associative processing is used, comparing each aircraft with all others. The pairs of aircraft meeting the criterion are output.

This method assumes that :

(1) the SP is faster than the AP in the final stages of conflict detection. Further more detailed processing may be required to resolve an aircraft's predicted position once it is known that it may be on a collision path.

(2) the AP is performing very efficient screening for potential conflicts.

Example 3.

The screening process that can be used with Example 2 (Downs, 1973) can be considered as follows. The air space is divided into "boxes" or "cells". It is then determined whether any pairs of aircraft are in the same cell or are in neighbouring cells. That is, each cell is compared "with itself" and "with its neighbours". It suffices to check a cell with those cells which are to the north, north-west, north-east and east. See Fig.2.7.

It is necessary to consider how easy it would be to map such an algorithm into the system to be developed. One cell to a PE would be very wasteful, since many cells may be empty. Alternatively, more than one cell could be stored in a PE. If two cells are stored in each PE, checks between them, as part of this screening process, could be easily made. If each PE could communicate with a neighbour, then all of the comparisons shown in Fig.2.7 could be carried out. For such an approach, the two cells in each PE could be considered to be in the horizontal or east-west orientation and those in adjacent PEs to be in the vertical or north-south orientation.

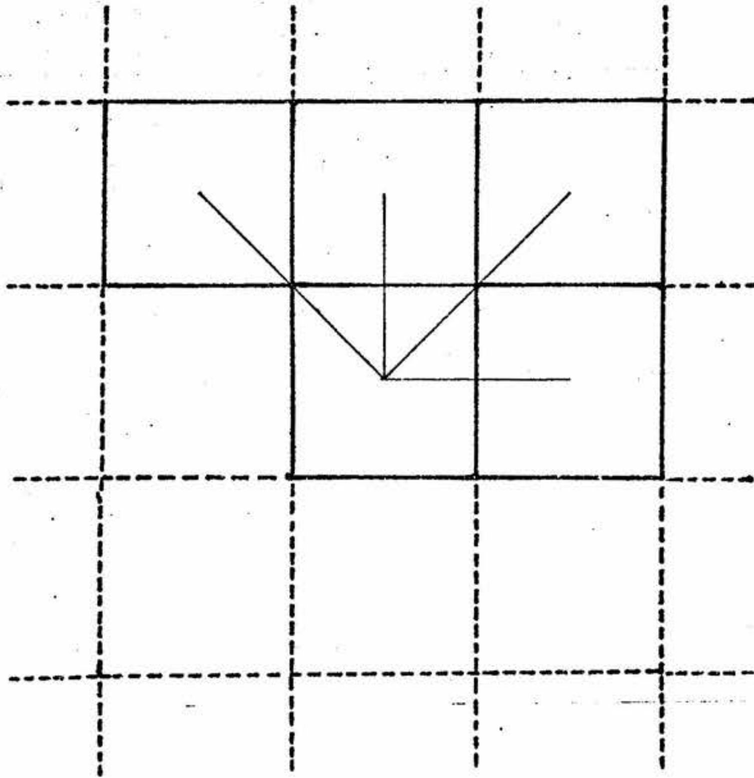


Fig.2.7. Cell Comparisons for Filtering of Aircraft Pairs

Example 4.

Downs (1973) also mentions as a possible approach, the sliding correlation algorithm. The aircraft are sorted in the PEs into increasing range order.

Copy the state estimates and I.D.'s of each aircraft and simultaneously pass them to the next adjacent processing element. Each element checks the two aircraft stored there for a possible conflict and flags each of them if a conflict occurs. The process continues until no checks are required for the distance in the real-time given for processing.

Note that the ordering of the aircraft allows an implicit screening process - aircraft are only checked against those of a similar range. Downs suggests that in comparison to the previous example, this approach has a smaller data management overhead and less selectivity.

If we assume that the final conflict detection algorithm is very complicated and time-consuming, and is to be applied to pairs of aircraft, then the most suitable preliminary role for the AP is to provide a very fast screening process of great selectivity and low data management overhead.

2.3.2.4. The Chosen Air Traffic Control Approach.

It is not necessarily true that although only a small AP could be considered, only simple solutions to the air traffic control problem could be used. It was hoped that an algorithm could be provided that could be easily expanded for larger APs, that is, APs with a much larger number of PEs.

The approach chosen was divided into four sections :

(1) A prediction is made of the position of each aircraft in the two-dimensional airspace, based on the assumption that the velocity of the aircraft remains constant over the prediction interval. The assignment of aircraft to cells is made and an 8x8 array of cells is used for this operation. This choice was dictated by the number of PEs which were going to be used.

(2) The filtering process is then performed and the SP receives the identification numbers of those pairs of aircraft that may be on collision paths. The cells are compared with their neighbours as shown in Fig.2.7. It is possible for a cell to have none, one or more than one aircraft assigned to it. It was decided to allow a maximum of four aircraft per cell. The occurrence of more than that number of aircraft in a cell would suggest that the cell size was too large and the scale should be changed.

(3) The main algorithm for predicting collisions would be applied at this point to those pairs of aircraft that had been selected by the previous operation. This algorithm would be adapted to the available operations that could be provided by the AP and could be replaced by more sophisticated algorithms if more memory space was available in the SP and AP. The AP would be operating as a parallel processor for these pairs of aircraft.

(4) An updating operation on each aircraft's position would be performed. This section would replace and simulate the real-time tracking and position algorithms

that would be used in a real environment. After this, the program returns to (1) and the whole sequence is repeated.

2.3.3. Information Retrieval Query Processing.

2.3.3.1. The Associative Processor Operations.

Minshull and Murphy (1972) have demonstrated the use of associative processing for search operations in which considerable logical processing can take place before the information is extracted. An AP could handle a dictionary of keys (all situated at the same level of the storage hierarchy) whilst the information linked to these keys is available elsewhere (e.g. in the SP). (Examples of data base applications are given in Berra, 1974; Dugan, Green, Minker and Shindle, 1966.)

This application was chosen because it emphasised complementary operations to those of the air traffic control conflict detection application. Table 2.2 shows the operations of the AP and the emphasis provided by each application. Since the main point of this study was a consideration of the microprogrammed control of associative processors, a second application which demonstrated the generality of any control approach was more appropriate.

Minshull and Murphy (1972) note that the query operation can be provided by microprogrammed control of an AP. It seems possible to reduce the number of search operations by using operations on the various control registers and flags associated with the memory part of

Operations	First Application Air Traffic Control Conflict Detection	Second Application Information Retrieval Query Processing
branches and control at MCU		X
loads and stores		X
parallel moves in PEs	X	
parallel arithmetic in PEs	X	
parallel associative operations in PEs	X	
searching with MCU key		X
interPE communication	X	
I/O operations	X	

(X - emphasis)

Table 2.2. The Emphasis on AP Operations.

the PEs. As Love and Savitt (1967) noted :

> The memory and its associated logic should provide the capability to either automate or greatly facilitate the selection of values of an X which is specified by the conjunction of two or more Control Structure relations. Specifically, the time spent processing values which meet the criterion of being in one specified relation but not another must be minimised.

2.3.3.2. The System Configuration.

The same system configuration as that employed in the first application would be used here. We note only certain additional aspects that are important for this application.

The PEs would be mainly used for storing a data structure. If the data structure became too large to be stored completely in the PEs, then it would have to be stored in the SP memory and be brought across to the PEs in sections. The software organization is shown in Fig.2.8, which shows whether a particular software operation would be contained in the SP or the AP. The SP would interpret the command string and call a number of SP querying routines. A set of instructions would be passed across to the AP, for use in processing the data structure.

In a multi-user environment, queuing of requests to,the information querying routines would be used, and the information output for a particular query could be tagged with data on the particular user station that it should be sent to. Simultaneous operation of both AP and SP could then be maintained.

We now consider various approaches to this problem.

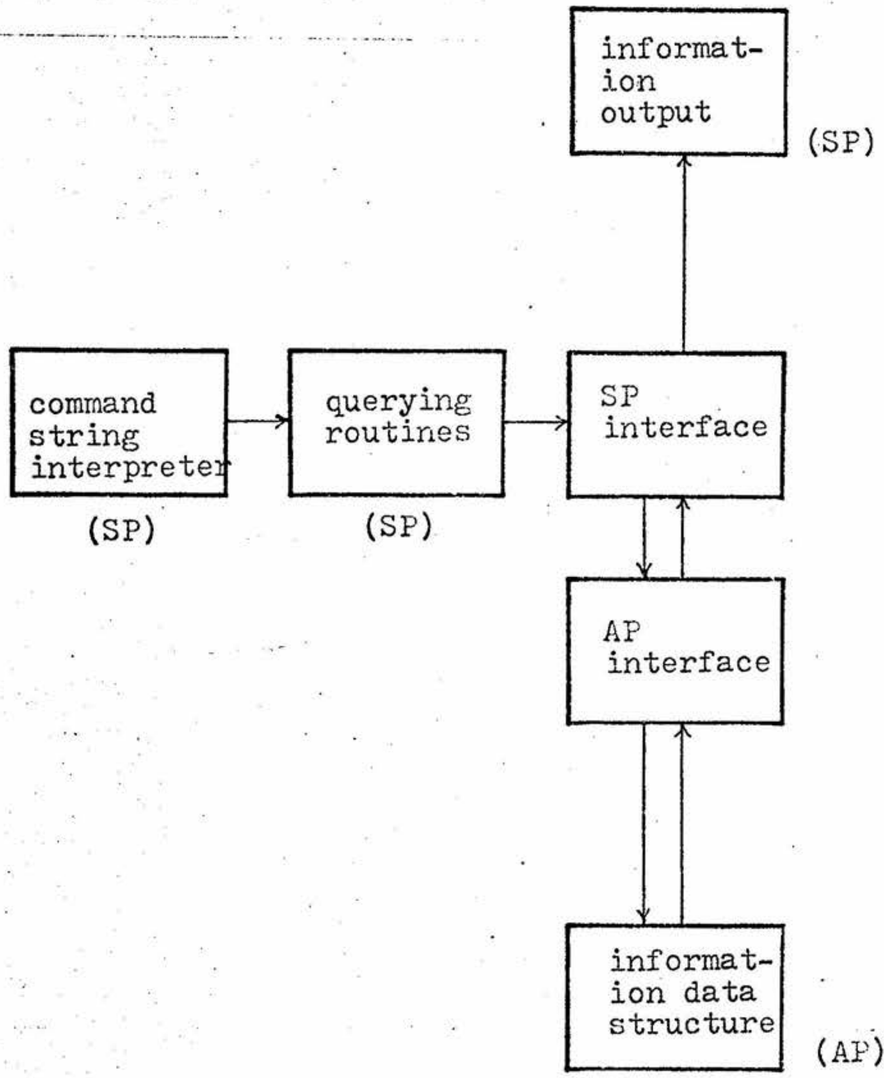


Fig.2.8. The Software Organization for the Second Application

2.3.3.3. Survey of Information Retrieval Approaches.

Example 1.

Minshull and Murphy's approach (1972) gives some idea of the querying activity desired. They consider relations set up with attribute, object and value elements. Their querying operation is the simple one of solving the equation :

$$A(O) = V$$

where one or more of A (attribute), O (object), V (value) is an unknown quantity.

Microprogramming could provide this simple operation as well as much more complicated operations derived from this simple operation. The main drawback of this approach is that a considerable number of associative searches may be required. It seems possible to minimise the number of such associative operations by making full use of the processing capabilities of the MCU on the AP status information.

Example 2.

Sibley, Taylor and Gordon (1968) have developed a similar approach. However, they simulate the operation of an associative processor on a conventional machine by using a program called TRAMP (Timeshared Relational Associative Memory Program). A hash-coded scheme is used to achieve this. These techniques were applied to graphic systems communication. Use of a sequential machine is suitable when only one relation is required at a time.

Example 3.

Love and Savitt (1967) have described ASP as a language suitable for information retrieval operations. It seems to provide a powerful generalization to the type of approach which was considered by Murphy and Minshull (1972). The method allows the query mechanism to be used for creating and updating the data base. It was decided to concentrate on just the basic query operation.

2.3.3.4. The Chosen Information Retrieval Approach.

Since the number of PEs would be limited, the data structure would be stored in the SP and sent across when necessary. It would be sufficient to demonstrate the operation of the system employing a small data structure since the operation of the MCU would not depend on this size.

For simplicity, Minshull and Murphy's approach (1972) would be used for querying. For example, a query such as :

A(*) = V

would be interpreted as requiring a list of all those objects whose attribute A had a value of V.

The command string interpreter and the query routines would assemble a routine of AP instructions and transmit them to the AP. The AP would output back to the SP a list of all entries in the data base that had been tagged as satisfying this relation.

This approach together with that for the first application led to a general idea of what the design of the

AP should be like in more detail. This is considered further in the next section.

2.3.4. Initial Design of an Associative Processor.

The system configuration has already been illustrated in Fig.2.6. The microprogrammed control unit design is covered in Chapter 3. It remains to consider the design of the processing elements and their interconnection to the MCU and to each other.

Based on the Register Transfer Modules available and the operations required by the two applications, the data structure for the PE as shown in Fig.2.9 was developed. Microprogrammed control of this PE would be determined by the control lines to each of these modules.

Note that the PE provides for communication with the MCU and neighbouring PEs by incorporating a general purpose interface into the design. The input section merges the signals from the output section of the PE "above" this PE (see Fig.2.6) and the output section of a general purpose interface that is part of the MCU. Similarly, the output section for this PE is connected to the input section of the PE "below" it and also to the general purpose interface of the MCU. In addition, there is a special network that can output one bit of status information to the MCU. The eight PEs each output one bit and this provides an eight-bit status word that the MCU can use for control operations.

We can now consider some relevant aspects of micro-

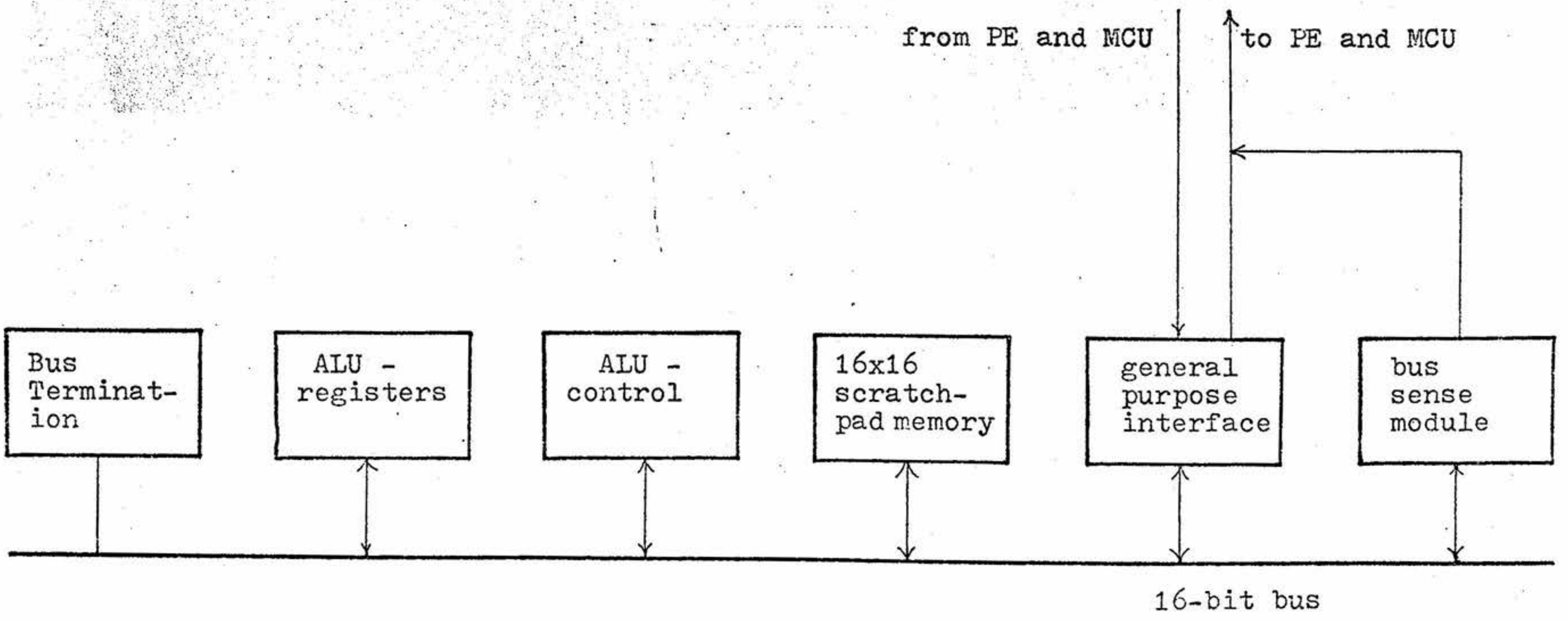


Fig.2.9. The Data Structure of the Processing Element

programming and the design of the MCU in more detail.

CHAPTER 3

MICROPROGRAMMED CONTROL

3.1. Introduction.

This chapter covers the microprogrammed control of relevant computer systems. Firstly, the concepts of interest are introduced. The major factors in microprogramming design work and the reasons for their use are treated and developments to date are noted.

The principles to be followed in the design of microprogrammed control units are considered in further detail, with particular reference to multiple resource systems. The treatment of associative processors is a special case of such systems. Current trends in hardware and software are also important in developing these principles.

Research in cellular logic theory and the applications of functional memory are particularly important to this study and the work of a number of researchers in this field is surveyed.

The design of the microprogrammed control unit of the associative processor is then covered. Apart from the general description, the design philosophy used and the advantages gained from using this approach are also considered.

3.2. Microprogramming Concepts.

A number of concepts have come to be considered important in the design of microprogrammed control units. See Davies (1972) and Flynn and Rosin (1971). We now consider the more relevant of these concepts.

The basic unit of microprogramming is called a microorder or microoperation. This can be considered as an indivisible processor operation which is executed in an indivisible period of time from the microprogrammer's point-of-view. Depending on the hardware implementation, the operation can be low-level and influence the operation of a small number of logic gates, or at the other end of the spectrum, it can be high-level and initiate quite a complex operation.

A number of such microorders are grouped together to make up the microinstruction. This is the basic unit retrieved from the control store of the microprogrammed control unit. This control store can be implemented in a variety of logic types. In general, these logic types can be classified as read-only memory (ROM), random-access memory (RAM) or shift-memory (SM).

Dynamic microprogramming refers to the ability to change the contents of the control memory while the processor is active, by using instructions to load microprograms from another level of storage.

A number of general characteristics can be used in describing the microinstruction used by an MCU. They include the following :

The vertical/horizontal characteristic.

The microinstruction can contain a few microorders or a large number of microorders. Since microinstructions with a small number of microorders will tend to require a smaller number of bits, the microinstruction is called vertical. On the other hand, microinstructions with a large number of microorders will tend to be longer (of the order of 64 bits) and thus are called horizontal. The larger number of microorders means that many operations can be executed concurrently whereas vertical microinstructions operate in a more sequential manner.

The encoding characteristic.

A number of microorders can be mutually exclusive, either because they are logically incompatible or they are precluded in hardware from occurring concurrently. In this case, they can be encoded in a microinstruction field and thus reduce the number of bits required to specify desired microorders during operation. There is a tendency for vertical microinstructions to employ maximum encoding and horizontal microinstructions to employ minimal encoding. There is no direct logical connection for this but the narrower wordlength for vertical microprogramming motivates the hardware designer to encode various operations at the functional level.

The serial/parallel characteristic.

Serial operation implies sequential operation of the microprogrammed control unit. (See Agrawala and

Rauscher, 1976, pp.76-79.) Parallel operation implies that the microprogrammed control unit performs some functions in parallel, namely, it executes one microinstruction whilst fetching the next.

The monophase/polyphase characteristic.

Monophase operation of the MCU denotes operation in one single clock cycle. Polyphase operation refers to execution of the microinstruction in a number of clock cycles which can be considered as minor cycles making up one major cycle. The number of minor cycles required for the execution of a microinstruction may vary depending on the complexity of the microorders in that microinstruction. (See Agrawala and Rauscher, 1976, pp.79-80.) In this case, some convenient multiple of the minor clock cycles is chosen to denote a major clock cycle.

Residual control.

All the information necessary for the execution of a microinstruction may not be contained in it. Previous microinstructions may set up other control registers to be activated by the microorders of this microinstruction. Such control registers constitute residual control.

We now consider the main factors in microprogramming design and the reasons for their use, with particular reference to the concepts outlined above.

3.2.1. Factors in Microprogramming Design.

Although technology is important, the major factor in the design of microprogrammed control units seems to be the intended application. This intended application is the control of the data processing resources of the processor, and the design decisions taken in determining what data processing resources to provide are a crucial factor. In most microprogrammed processors, the MCU is totally integrated with these data processing resources and its design is difficult to separate out from that of the whole processor for the purposes of study. The Interdata Models 70, 74, 80 and 85 and the Hewlett-Packard HP21MX are good examples of this problem.

In addition, the way that the microprogram is going to be written plays an important part in determining the microorders. Some of the microorders are determined by the bus structure and the functional units that must be controlled, but others are determined by such factors as a particular way of implementing a multiplication algorithm or implementing the branching and testing for an application such as memory management. The impression is given in some cases that the hardware functions have been added to the MCU whilst the microprograms were being written.

The bus structure plays an important part in the complexity of the microinstruction format. Those processors with a large number of busses and/or a sophisticated interconnection structure can allow greater parallelism

in the use of processor resources. Consequently, the microinstruction format tends to be more horizontal. Vertical microinstruction formats tend to be associated with simple bussing structures.

We now consider the problems associated with these factors in microprogramming design.

3.2.2. Problems in Microprogramming Design.

In the debugging stage of the development of the STARAN microprograms for conflict detection, the lack of appropriate tools was evident. Dietzler (1972) found it difficult to determine the complexity of programming STARAN. However, he concluded that programming and microprogramming the AP for conflict detection required 18 man months for the Knoxville air traffic control contract whereas the use of a conventional processor at the Atlanta air traffic control contract centre required only 9 man months. These results are significant since they are related to a microprogrammed special processor with an unconventional instruction set and unconventional algorithms. In such an environment, the tools for efficient development are seldom available and this can detract from any possible advantage obtained by using the unconventional design.

The fact that writeable control store was used reduced the time spent in microprogram design, but Dietzler (1972) considers that the microprogrammed control led to machine instructions of a high level, and that

the same microprogram would not be suitable for more general applications. Thus, the capability to dynamically load microprograms is a very useful feature for any processor with special operational features.

As an example of this feature, Snyder (1975) has described the HP2100S microprogrammed minicomputer and how it was microprogrammed to provide the HP5451B Fourier Analyzer. One of the important points about this machine is that it uses a form of "virtual microcode" as he calls it. That is, an illegal opcode trap causes assembly language system routines, using a DMA channel, to load the writeable control store and "cause the no-longer illegal opcode to be reexecuted". This dynamic writeable control store feature is invisible to the user. Special PROMs (programmable ROMs) assist in providing for parameter passing, the illegal opcode trapping, pseudo-interrupts and the pseudo-DMA. The loading of the writeable control store (WCS) takes approximately 1-2 milliseconds. The rate of loading is every 10 to 20 milliseconds. This approach has been found useful in those applications where optimization of critical areas can lead to significant improvements in performance.

Thomas (1974) in his consideration of the execution of microinstructions from main memory, also assumes an illegal opcode fault being exploited to switch to interpretation.

Another of the problems in microprogramming design is determining the format of the microinstruction. This is especially true for horizontal microprogramming. The fields

to be encoded can be determined to some degree by the microorders to be sent to particular functional modules. Interpretation may also require the use of residual control, which can also provide a form of conditional interpretation. The residual control registers can also provide residual data for various operations. Such interpretation also influences the format of the microinstruction.

Another problem is the minimisation of the length of microprograms. One programming feature that can assist is the use of subroutines. Microprogram subroutines require additional circuitry in the MCU and usually only a limited capability is allowed. For example, the Hewlett-Packard HP21MX provides a save register associated with the control store that can be used for this purpose.

The minimisation of the microinstruction width and hence the width of the control store has also been studied. Jayasri and Basu (1976) have considered this problem for the case where the microprogram is already defined and the task is to map this microprogram into the control store, and encode and format the microorders. They apply linear programming techniques to solve this problem.

3.2.3. Developments in Microprogramming.

Jones and Merwin (1974) have noted that the development of the theoretical foundations of microprogramming has been slow. They note :

Only a few papers have even attempted to define the central problems of microprogramming and microprogrammable processors, and there has been little work linking microprogramming to automata theory, combinatorics, or computer architecture.

Much of the current developments in microprogramming is a consolidation of previous work. In general, there tends to be a preoccupation with those techniques and system tools which make microprogramming at the user level easier to do. In particular, such tools are very important for making microprogramming of special processors more cost-effective.

Topics covered in recent papers have included :

(1) user microprogramming of a machine with writeable control store, particularly to allow optimized operation of critical program sections.

(2) microprogramming tools, including simulators, high-level microprogramming languages, microdiagnostics, error control and the system specifications of microprogrammed processors.

(3) the optimization of microprograms and the correctness and equivalence of microprograms.

(4) new microprogrammed control units and processor designs, and their applications. Multimicroprocessor designs are being considered although no new control techniques are evident.

There does not seem to be any new work on the concept of control of functional modules, particularly with reference to instruction or microinstruction format. Such a theory would be very useful in the design of many special purpose systems.

3.3. MCU Design Principles.

We describe here those principles to be followed in the design of microprogrammed control units which seem most useful. The general case of multiple resource systems is treated here and the special case of associative processors will be treated later.

3.3.1. Separation of Fields.

Even with the decreasing cost of semiconductor memory, one of the major considerations of microprogramming is the size of the microprogram and consequently the size of the control store.

Flinders, Gardner, Llewelyn and Minshull (1970) in considering a possible new type of control store design, emphasised the minimisation of the tables representing switching functions. They considered the possibilities of sharing words between different functions.

One of the important factors in determining the size of a control store is the field combination problem (Gardner, 1971). Consider two fields F_1 and F_2, where F_1 can have n_1 possible values and F_2 can have n_2 possible values. If these fields are provided by the same word in control memory, there are n_1*n_2 possible combinations. If they could be provided by two separate control words, separated by either time or space, then the number of possible combinations is reduced to n_1+n_2. Although not all combinations may be used in microinstructions, the

subset of microinstructions determines subsets of values for each field making up that microinstruction, and a summation rather that a multiplication greatly reduces the number of microinstructions that have to be provided.

Emit fields that can hold a data value for some duration provide a limited separation of fields, but the ultimate solution is to have the microinstruction physically split into separate sections when stored in control memory modules. That is, the microinstruction is provided by different control words accessed from separate memory modules.

A consideration of this design principle leads to a number of potential problems :

(1) It is not clear what is the best way to control the generation of these different fields that make up the microinstuction. (The technique chosen for the associative processor will be one solution to this design problem.)

(2) It is most unlikely that the number of separate sections of the microinstruction correspond to the number of fields required, although the choice for encoding fields would take place after choosing the number of separate sections and would thus be influenced by it. In general, there would be many more fields than the separate sections of the control store. These fields have to be allocated to control memory sections so that the generation of microinstructions is optimal in some way. That is, we would like useful field combinations from the microprogrammer's

point of view.

(3) How do we measure and compare design alternatives if we are allowed the freedom of assigning encoded fields to separate sections of control store? Ideally, the same microroutine can be encoded according to different microprogramming strategies. There is much subjectivity involved in each such programming exercise and it can be difficult to judge which microroutine is more efficient in the use of resources and easier to understand. However, it is possible to determine whether the minimal number of microinstructions for a particular simple operation has been used. The number of bits of control memory can also be compared for different microroutines if we can estimate the hardware resources saved by using a particular firmware feature.

These considerations should be kept in mind in developing microprograms. Separation of the microinstruction into independent sections generated by different control memory modules does seem a very useful way of alleviating the field combination problem.

3.3.2. Branching Delays.

Gardner (1971) has stated that :

> Data flow conditions are of two types : those that give only minor changes in direction (e.g. the differences that exist between ADD, SUBTRACT and COMPARE) and those that give gross changes in direction (e.g. differences between, say, ADD and EDIT). Minor changes should be kept in the data flow; only gross conditions should be allowed to return to the control store.

Branching delays can have a significant effect on the execution speed of a processor. Whenever straight line code is being executed, instruction fetches can be pipelined to different banks of memory in parallel with the execution of the current instruction. The use of micro-programming can hopefully reduce the number of decisions at the user instruction level, by incorporating minor decisions into the microcode whenever possible.

It would be preferable if decisions at the microcode level could be made without any inherent delay, in parallel with other microoperations and using processor status information provided by previous microinstructions. The user instruction level should find no noticeable effect of the decisions being made at the microinstruction level.

The principle of using status provided by micro-instructions executed before the current microinstruction can be extended to the user instruction level. If the processor status on which decisions are made is available at least one instruction before it is used in a decision-type instruction, the instruction fetch section of the control unit can independently modify the instruction stream, in parallel with the activities of the instruction execution section.

With associative processors, the possibilities of exploiting such a principle is further enhanced. The number of loops is drastically reduced since iteration can in many cases be accommodated in the parallel operation. With careful programming, all decisions should be made on status provided well before it is required.

3.3.3. Functional Module Control.

Gardner (1971) has also stated :

> The system designer should attempt to identify those areas of control which, though concurrent, are architecturally independent of each other and provide a separate source of control for each independent area for the duration of their independence.

This principle is related to the field combination principle and leads to a reduction in the number of control lines to functional modules. It can also lead to a simplification of the microinstruction format for the control of these modules.

Such an approach presupposes that the processor is divided up in some way into a number of different sections that can be identified as separate modules. With associative processors, the functional modules are fairly easily identified. The processing elements are identical and the control field for their control can be provided by a single section of the microinstruction. Additional functional modules can be associated with I/O and the control of the MCU itself.

An established trend seems to be the use of more semi-independent functional modules, and microprocessors have been considered as possible general purpose functional modules. They have in many ways taken over from the earlier concepts of universal logic modules provided by various organizations of cellular logic. The success of this concept is nevertheless dependent on the same sort of problems, such as the problem of controlling a number of such modules communicating with each other.

Although each module may have a separate control unit associated with it, overall control of a network of such modules must still be provided.

Various approaches to this problem have included the following :

(1) The control function is spread over all the modules, by suitably designing the architecture of each module and the means for controlling and communicating with its neighbours. The design of the AN/UYK-17(XB-1)(V) is an example of this approach (Rauscher, 1974).

(2) A very powerful central control unit is provided and the control capability at the processing element level is very limited. The ILLIAC IV computer was designed with this approach in mind (Barnes, Brown, Kato, Luck, Slotnick and Stokes, 1968). Each PE could be selectively enabled or disabled from executing instructions, and the only parallelism allowed was in all active PEs obeying the same instructions.

A particular architecture of a processor may require a mixture of the above approaches, with little autonomous control for some modules and extensive autonomous control for others. The control of a processor could come to resemble that of a simple computer network.

The normal method for managing the control of a number of functional modules is to issue microorders from a central control module. This method reduces the intercommunication between functional modules to data paths and some status information.

As an interesting example, the AN/UYK-17(XB-1)(V)

processor (Rauscher, 1974) has already been mentioned. This is an example where microinstructions are used to control a number of functional modules. Considerable parallelism is involved. A functional module such as the signal processing arithmetic unit of this processor is self-contained. Once its operation is initiated by the microprogrammed control unit, it controls its own operation and when execution is complete, signals termination to the microprogrammed control unit by means of an interrupt. Thus, each functional module provides the same asynchronous control as that of register transfer modules (Bell, Grason and Newell, 1972) or macromodules (Clark, 1967).

The signal processing arithmetic unit is in turn composed of a number of functional modules which function independently and in parallel. Thus we have a hierarchically structured system with autonomous control of functional modules and extensive parallelism.

The microinstruction used to control such a system is 160 bits long and composed of 63 fields. The allocation of encoded fields of the microinstruction show the influence of a careful microprogram development effort. Many microorders may seem quite arbitrary until they are considered in the context of signal processing and the requirements for processing finite series and generating Fourier transforms.

Jayasri and Basu (1976) have considered methods for reducing the width of control store by suitably encoding the microorders into groups of fields. The difficulty with this approach may be that it presupposes

that the modules to be controlled are designed after the microprograms have been established. If the existence of the functional modules is presumed, then the principle used here is to assign encoded fields of the microinstruction for these modules and to consider the microprograms and the microprogrammed control unit at a later stage of the design cycle. Such as approach may not lead to the minimization of the control store obtainable by their use of linear programming.

Cellular logic is an extreme case for providing functional modules where the control operation is distributed over the collection of cells by suitable design of the cell and interconnection structure between cells. We now treat those aspects of cellular logic and functional memory research that are particularly relevant to microprogrammed control.

3.4. Cellular Logic and Functional Memory.

The motivation for the survey of research in cellular logic and functional memory is to see how developments in these areas of switching theory are relevant to the design of new kinds of microprogrammed control units, particularly those designs that follow the principles outlined above.

Hybrid associative memories have already been described. These memories can provide functions suitable for the control of functional modules.

3.4.1. Hybrid Associative Memories.

An associative memory is an ideal device for providing a microprogrammed control unit, because the extra logic required to support this "control store" is reduced to a minimum. Fig.3.1 shows an example of such an MCU. The major disadvantage is that a large amount of memory may be required and this is very costly for memories of this type.

Reduction of the required size can be achieved by a number of techniques including encoding the data in some way and providing extra logic to support this operation. The design of the associative memory can also be altered in some way to reduce its size and cost.

Weinberger (1971) in his development of hybrid associative memories considered four-state cells, with the four states of the cell being called :

1 "don't care" state
2 0 state
3 1 state
4 permanent mismatch (disable)

These four state cells were used to create small associative arrays. A number of such arrays would be available on a chip and could be selected by standard decoding. The address field that is used to access this memory consists of a section that is used for coordinate addressing and a section that is used for associative addressing. Weinberger suggested using one field to select a particular switching function and the other to provide

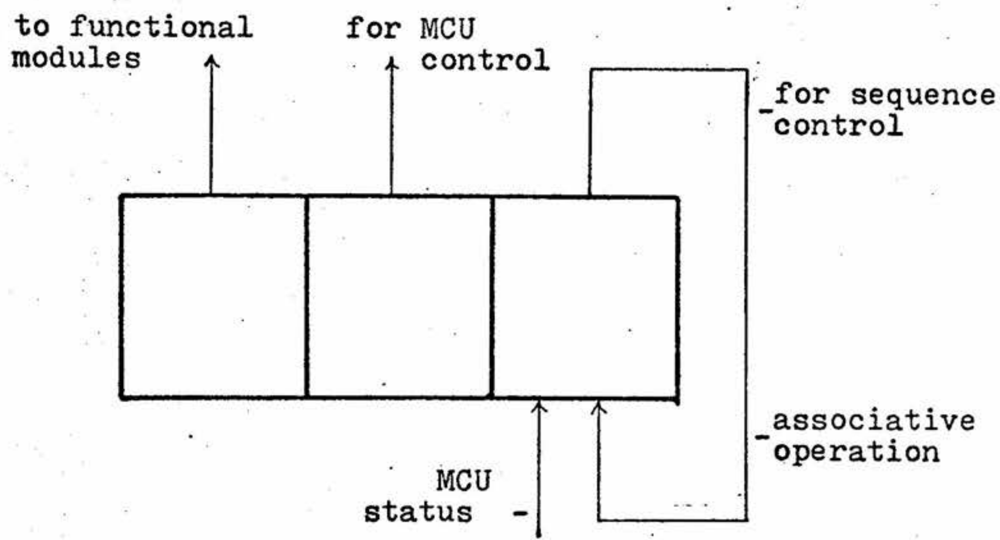


Fig.3.1. An Associative Memory MCU

the input values for this switching function. The coordinate addressing field is assumed appropriate for selecting the function, but although easier to visualise and simulate, it is not necessary to have the switching function fields correspond to the implementation fields.

Weinberger (1971) asserts that

> ... the hybrid organization has a significant advantage over a fully associative memory in reducing the number of connections that must be made to a chip for an array of any particular size. Furthermore, most data can be arranged in hybrid form and the performance of a hybrid memory can substantially equal the performance of a fully associative memory. Thus, the hybrid organization is useful with specifically designed arrays as well as arrays designed for nonassociative use. (U.S. Patent 3,644,906, p.5)

Each multivalued switching function would have an optimum partition for the decoded part of the function and the associative part. A number of such functions would require considerable calculation to provide an optimum solution to this partitioning problem.

If the switching functions were unknown, as in the case of a dynamic medium which is supposed to provide logic capability or control the issue of microinstructions, the optimum partition can only be designed with reference to other engineering considerations such as the pin-circuit ratio, the power dissipation, the circuit area and packaging. Additional choice would be based on criteria such as the ease of conceptually organizing the switching and control functions in this new environment.

Fleisher, Weinberger and Winkler (1970) have considered this problem of providing logic functions using memory with decoders available on each chip, without treating any associative aspects. To quote from the abstract of

their patent (No. 3,593,317) :

> An improved method and means to implement a logic function F of N variables by partitioning the logic operation in a plurality of generalized logic matrices. It is first mathematically demonstrated that a function F of N variables may be expanded into subfunctions of a lesser number of variables. These subfunctions may be logically implemented individually and then logically combined so as to produce the desired function of N variables with a concomitant savings in logic circuitry over that required if the functions were directly implemented. The means used to implement the logic function F are a plurality of generalized logic matrices, each of which comprises a plurality of logic gates arranged in columns and rows, an input decoder for accepting the input variables, and a storage register for varying the functions generated at the output of the matrix. These matrices are arranged in cascade so that, as the function F is constructed from the several subfunctions, additional variables are inserted at each matrix stage until the function F of N variables is fully generated.

The use of matrices of logic gates together with the use of decoders implemented on the same chips is related to Weinberger's work on hybrid associative memories. Again, although it is possible to partition a logic function and assign logic matrices to generate each part of this function, there is some difficulty in applying this same approach to the generation of microinstructions.

3.4.2. Cellular Logic.

Mukhopadhyay and Stone (1971) have given the following definition :

> A cellular array consists of a 1-, 2-, or 3-dimensional iterative arrangement of similar or identical cells with a uniform interconnection pattern on the cells.

For a given arbitrary switching function, the realizability of particular organizations of cells to provide this switching function and the bounds on the

largest number of cells required have been studied.

One such organization has been the single-rail cascade (Mukhopadhyay and Stone, 1971) consisting of cascades of 2-input 1-output cells. By increasing the number of interconnections between cells, we get two-rail cascades. Two-dimensional arrays include organizations called cutpoint arrays, NOR arrays, NOR-NAND arrays and majority-gate arrays. The minimization of cellular arrays has also been considered.

Programmable cellular arrays have been considered by Kautz (1971) where it is possible to change the operation of each cell by placing it in a number of different states. The "state" of the cell can be determined by storage of some data within it, by input lines to the cell dedicated to providing state information, or by a combination of both these techniques.

In addition to treating the use of programmable cellular arrays to provide general switching functions, Kautz (1971) has also considered various special-purpose arrays including

(1) threshold arrays,

(2) sorting arrays,

(3) associative memories,

(4) coding arrays,

and (5) interconnection arrays.

One of the major problems with the use of such special-purpose modules in processors has been the difficulty of controlling them. Some form of microprogrammed control has been suggested and consideration has been given

to providing control by using arrays of these cells in similar organizations. Functional memory has been suggested as a suitable medium for all the functional modules making up the processor, including the control unit (Flinders, Gardner, Llewelyn and Minshull, 1970).

3.4.3. Functional Memory and Array Logic.

Fleisher and Maissel (1975) refer to "array logic" as the "use of memory-like structures for performing logic". They suggest that only the use of semiconductor memory technology can provide the circuitry that will receive extended usage. The hybrid associative memory treated above is a form of array logic in which the memory properties are emphasised over the logic properties of the circuitry.

Similar to Gardner (1971), they have studied an AND array followed by an OR array for generating their switching functions. Their array logic uses one-input decoders and Boolean expressions are found to be easily translated into table entries. The two-input decoder case is also considered. They are concerned with bit counts and show that the minimum number of bits required to implement a function of 16 variables is the same for both two-input and one-input decoders, because 2 one-bit decoders provide the same number of output lines as 1 two-input decoder.

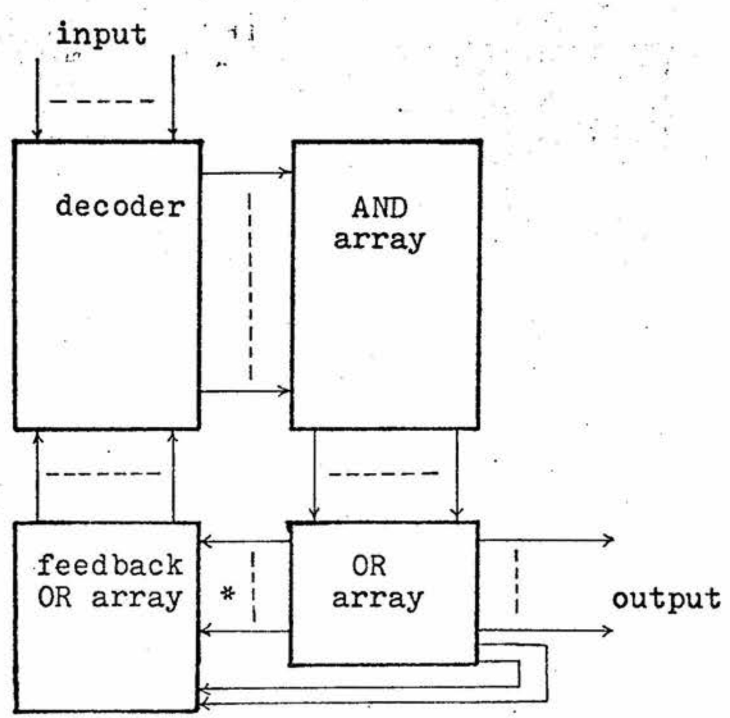
The choice of partitioning in implementing switching functions is related to the problem of partitioning the

operations in microprogrammed control unit design. The main motivation is again the minimization of the arrays. Important factors in providing this minimization include the choice of correct output phase, the sharing of output columns of the arrays, the splitting of variables (so that they are used again in different sections of the arrays) and the maximum use of minterm "DON'T CARE"s. Kaurnaugh diagrams have an easy correspondence to the 2-input decoder arrays and the minimization of such diagrams has an analogy to the minimization of these arrays.

Fleisher and Maissel (1975) have considered that MNOS and CMOS provide an adequate technology for a physical implementation. The decoders can also be provided by the use of an OR-array in the extra rectangular array space available, as shown in Fig.3.2.

Logue, Brickman, Howley, Jones and Wu (1975) have considered the design of a control unit using programmable logic arrays (PLAs). Their design is unique in that it uses similar functional memory techniques. Fig.3.3 shows the configuration of AND and OR arrays which implement feedback control for the generation of microinstructions. It should be noted that they attempted to implement an associative technique for addressing.

One of the concepts that they introduce is that of a macro, which they equate with a building-block of the PLA which performs one specific function. The operation of a macro can be identified as a state of the PLA and the PLA is sent from one state to another. A state assignment procedure is required and they give a hierarchical control



* = a register is used here to prevent race conditions

Fig.3.2. Functional Memory Unit with Built-In Feedback

(from Fleisher and Maissel, 1975)

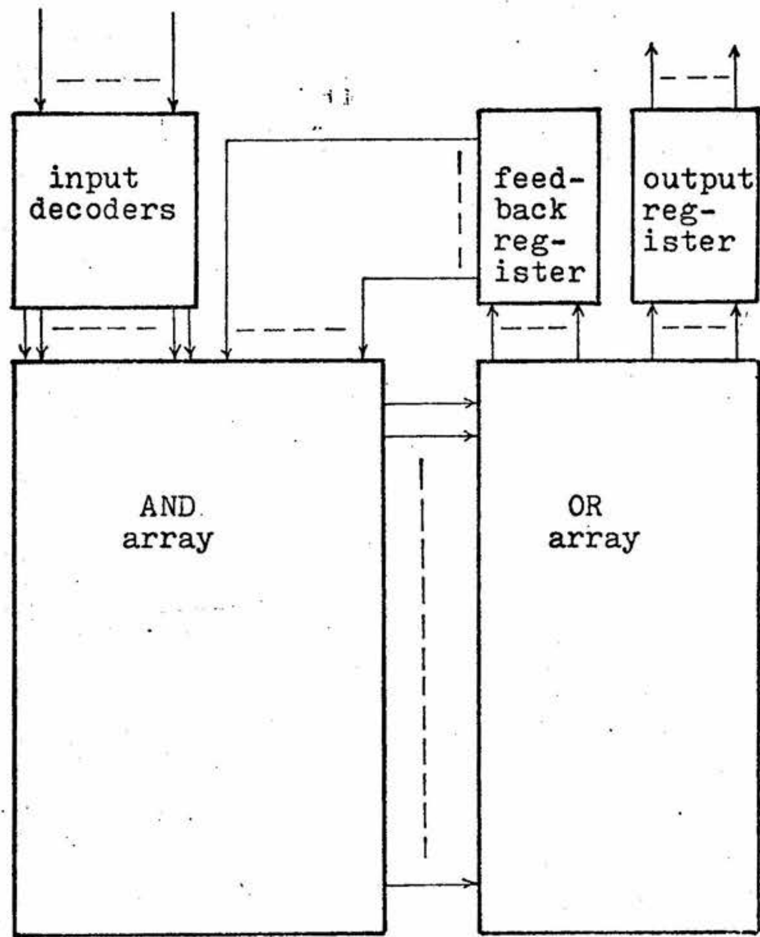


Fig.3.3. Functional Block Diagram of Terminal Control Unit

(from Logue, Brickman, Howley, Jones and Wu, 1975)

scheme which could be useful. As shown in Fig.3.4, PLA 2 performs the normal feedback operation. However, at the end of a macro operation, an acknowledgement signal is generated and PLA 1 uses this to request the next macro operation and to control the state of the system in general.

The physical implementation of this control unit called the Terminal Control Unit, was preceded by a design phase and a simulation phase. The language APL was found to match closely the simulation requirements for developing programs for the PLAs.

Logue et al (1975) give am example of a microprogram written for their functional memory, demonstrating the linking of macros and the ability to encode and decode control information. The sparseness of the arrays is a characteristic of this form of microprogramming. In fact, such sparseness can be found in more conventional micro-programming as that used in the AN/UYK-17(XB-1)(V) signal processor (Rauscher, 1974).

It was found that the packaging of the PLAs and supporting circuitry would have halved the number of printed circuit boards required. Each PLA replaced on the average of 250 logical circuits, 48% being combinational and 52% being sequential.

It is interesting to note that one of the advantages that the PLAs were considered to have was the integration of both the control and data paths of the processor. The alternate approach which was taken here has been to separate out these paths and provide a control unit which

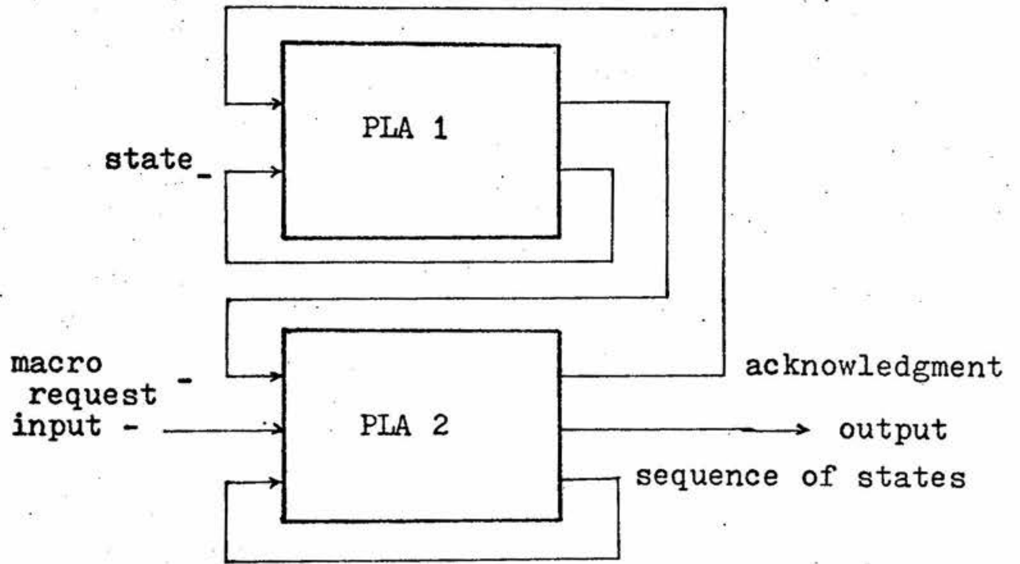


Fig.3.4. Hierarchical Control Scheme

(from Logue, Brickman, Howley, Jones and Wu, 1975)

incorporates all the processing associated with the control path.

Jones (1975) has mentioned how AND-OR arrays have many of the characteristics of associative arrays. Among the advantages for combinational logic are :

(1) Race conditions can be eliminated by using a register to store the resultant logic values.

(2) The speed for generating logic values is dependent only on the propagation delay of the arrays and is independent of the complexity of the logic function.

(3) The implementation of certain logic macros can lead in conventional cases to topological problems. These are non-existent in array implementations.

Jones (1975) has studied various enhancements to array logic. Providing sequential logic by the use of feedback registers is mentioned. He suggests using JK- or T-type flipflops to provide incrementing and decrementing macros. The disadvantage of this approach is that more complicated circuitry is required than if simpler SR-type flipflops are used. The incrementing and decrementing facilities that his circuits have can be provided by using "counter fields" in the functional memories as will be illustrated in the description of the MCU which was designed.

Jones also considers the capability of array logic of this type to provide a data switching facility. As shown in Fig.3.5, an input field can be switched to one of a number of different output fields (with a possible change of value, i.e. code conversion) by the use of a

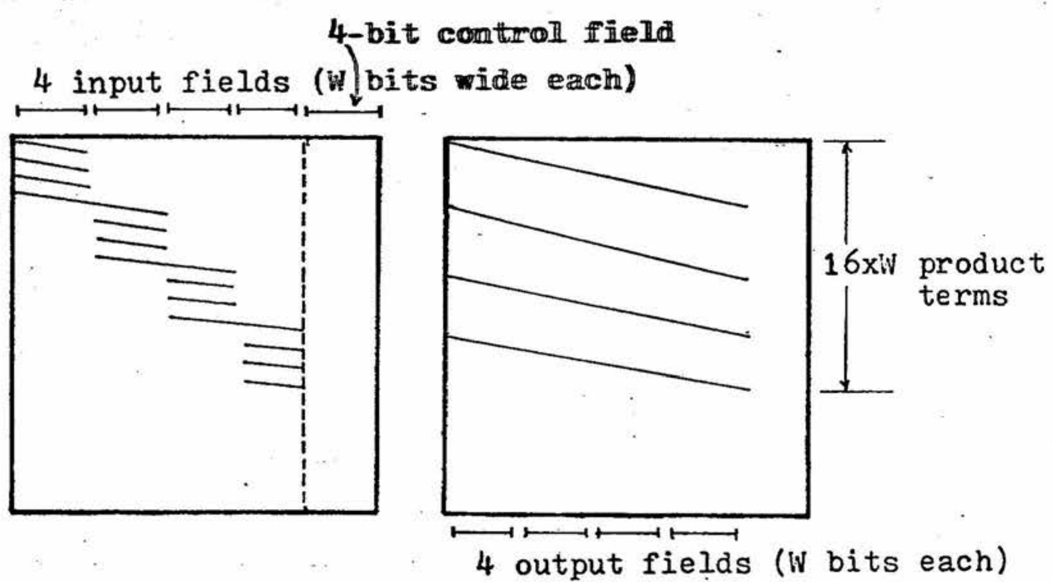


Fig.3.5. Data Path Switching Capability

(from Jones, 1975)

control field.

3.4.4. The Design of the Microprogrammed Control Unit.

Much of the design philosophy has already been treated above. In this section, other aspects of the design approach not previously covered are mentioned. The general description of the MCU is given and some of the advantages of this approach are also described.

3.4.4.1. Some Aspects of the Design Approach.

The instruction stream provided by the user has a wordlength of 16 bits. The microinstruction wordlength is 64 bits long and a variable number of such microinstructions is required to accomplish the operation of the user instruction. The microinstruction subfield sent to each module of the processor should constitute a "subinstruction" in the format ready for direct execution by the microprocessors or other logic associated with that module. The MCU circuitry also constitutes such a module for the purposes of control. Such "subinstructions" are encoded to minimise the bandwidth of the MCU with due regard for the requirements of each module. In the case of the DEC RTMs, the appropriate microorders are determined by the control lines for each module. In some cases where microorders are mutually exclusive, encoding may be used. For example, one RTM may provide only one of a number of operations at any one time. No potential parallelism in

the use of the modules should be prohibited by the design of the MCU.

The size of the functional memories means that only a subset of the possible logic macros can be loaded at one time. Each subset is called a control level. A field of the user instruction determines which control level should be loaded. Dynamic loading takes place on a mismatch between the control level required and the existing control level of the MCU.

The approach to dynamic microprogramming in this implementation was similar to that of Thomas (1974) and Snyder (1975), with one difference in design philosophy. In both of these cases, dynamic microprogramming was used for special user instructions. In this implementation, dynamic microprogramming was applicable to every instruction. Obviously, the loading of microprograms or the changing of the control level of the processor is an activity which should be minimised because of the overhead involved. An attempt at such a minimisation was made by grouping the microroutines into classes in which there was a greater probability that after one instruction had been executed, the next instruction would require a microroutine in the same class. Such a grouping tended to be characterised by the level of complexity of control of the instruction, and hence the term "control level" was used to describe each such class of microroutines.

Note that dynamic microprogramming should not be confused with user microprogramming, which is not supported in this design. User microprogramming is the technique of

allowing the users to provide microprograms to be used in tailoring the computer system to their own application programs. The user provides both his program and microprogram which is to be used to augment the existing microroutines in the processor. Various architectures have been considered allowing user microprogramming including one such architecture where the user microprogram remains in main memory (Thomas, 1974).

User microprogramming has its serious disadvantages. Lehman (1975) has pointed out the dangers of allowing users to freely change the system architecture. He points out the difficulty of maintaining error-free and reliable system software in such an environment. This danger leads to policies such as IBM's, where no support is given to users who attempt to use the dynamic microprogramming capabilities of System 360/370 processors.

After studying the two applications, a general determination of the appropriate instructions was made. The microorders available and the microinstructions required were also considered. As mentioned before, the choice of microorders was very much determined by the register transfer modules and the available control lines. Thus the design process needed to be both top-down and bottom-up. It is interesting to note the experiences of Anagnostopoulos, Michel, Sockut, Stabler and van Dam (1973) on the close interrelationship between computer architecture and instruction sets. In particular, they note the restriction placed on the design of instruction sets due to the host machine architecture. Microorder

selection is similarly constrained by the machine architecture.

The choice of microinstruction format and the fetch-execute interpretative cycle has some degree of freedom since the microprogrammed control unit uses a different set of resources than that of the computation or data processing section of the processor. This was an approach used in the associative processor and is the opposite case from that of the Interdata Models 74 and 70, for example, where the microprogrammed control unit integrates resources with the data processing section of the processor.

The fact that the design of the machine architecture, including the microprogrammed control unit, the microinstruction format, the microprograms and the application programs, all can take place concurrently, emphasises one of the important advantages of microprogramming - namely the postponing of many design decisions about the instruction set to a later stage of the design process when the requirements are more clearly known. Such an approach to the design of a processor was used in developing the Hewlett-Packard 2100S computer (Snyder, 1975) and can be seen in the design of many other processors.

3.4.4.2. General Description of the MCU.

Fig.3.6 shows the basic data structure of the MCU. A standard RTM bus is used to connect the components together. The set of functional memory units in the centre

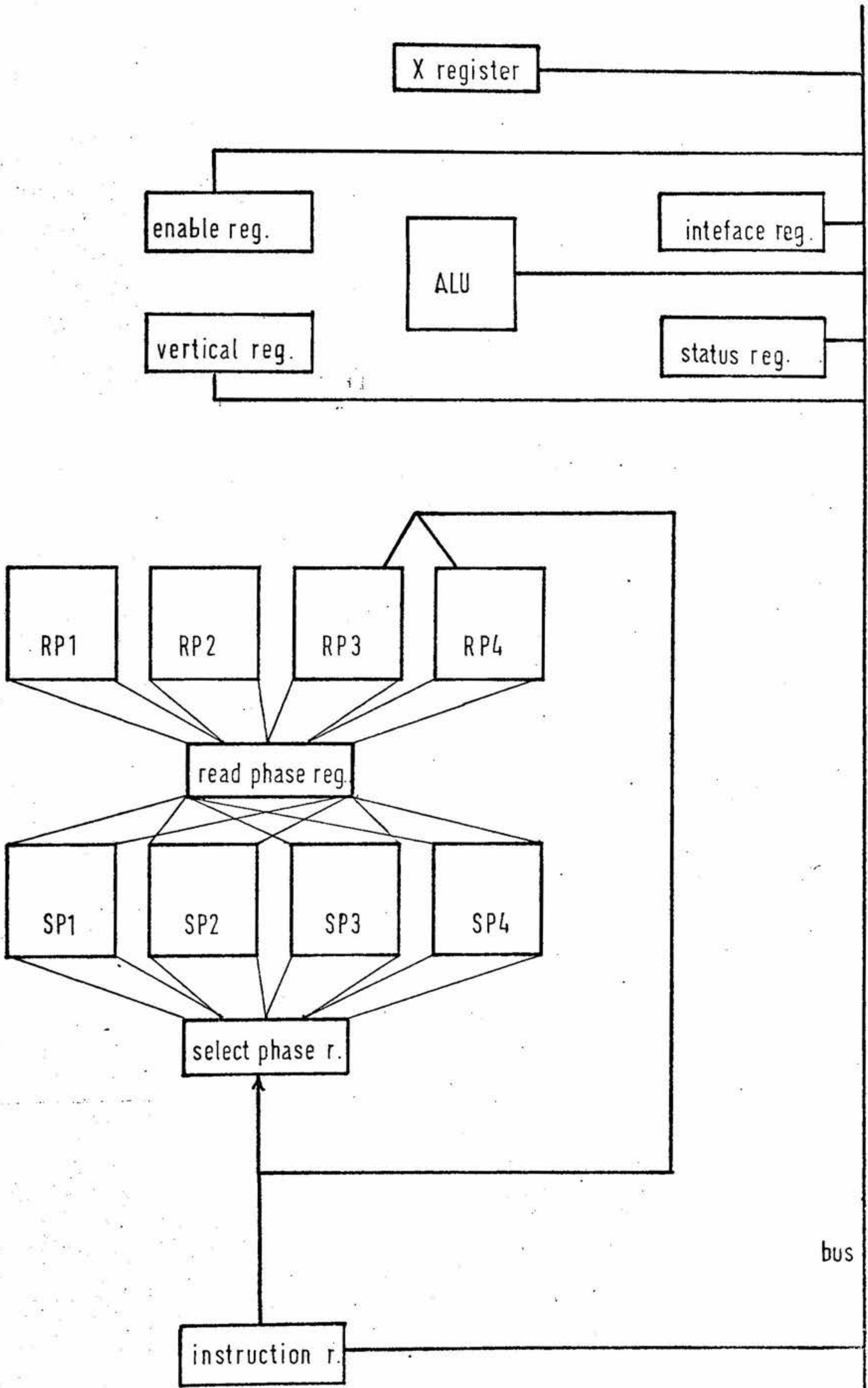


Fig.3.6. Data Structure of the MCU

of the diagram has its own interconnection structure separate from this bus.

Four registers are part of the MCU which are specifically for use in controlling the operations of the associative processor. These are :

(1) the enable register - a bit set in this register corresponds to the enabling of one of the PEs.

(2) the vertical register - provides a useful flag for each PE, which can be used in search and I/O operations.

(3) the interface register - used for I/O between the MCU and the PEs. Both data and status I/O operations with the PEs use this register.

(4) the status register - provides another flag for each PE, primarily for status information. Such status information can, in turn, be transmitted to the SP.

These registers and their associated microorders represent the separation of the control section of the associative processor from the data processing section provided by the PEs.

Two other registers provide for the input of user instructions into the system. The instruction register receives its data from the SP. The X register (eXtra register) provides a useful means of storing one of the four-bit fields of the instruction for further data manipulation. It constitutes local storage for the MCU which is unobtainable from the functional memories because of their small size.

The functional memory section is the final and central section of the MCU, and it is shown in Fig.3.6 in a more functional form than the rest of the diagram. Its data operations are as follows :

(1) The select phase register breaks the 16-bit word into 4 4-bit fields. Each 4-bit field is interpreted as an address for one of the select phase functional memories (SP1-4).

(2) A word is selected from each select phase functional memory. These are 16x16 scratchpad memories. The bus is used at this point to provide an ORing function on the data from each SP functional memory and the result of this operation is stored in the read phase register.

(3) The read phase register, similar to the select phase register, breaks up the data word into four addresses for the read phase functional memories (RP1-4).

(4) The words read out of the read phase functional memories constitute a 64-bit data word and this provides the microinstruction for the MCU.

The choice of the terms "select phase" and "read phase" was motivated by the attempt to apply an array logic implementation philosophy to the generation of the microinstruction data word. The major difference was that standard RAM or ROM-type circuitry was being employed. "Select phase" was meant to suggest the AND operation of array logic, performing essentially a match operation, and "read phase" was meant to suggest the OR operation.

Fig.3.7 tries to show the relationship between the standard array logic techniques considered above and this

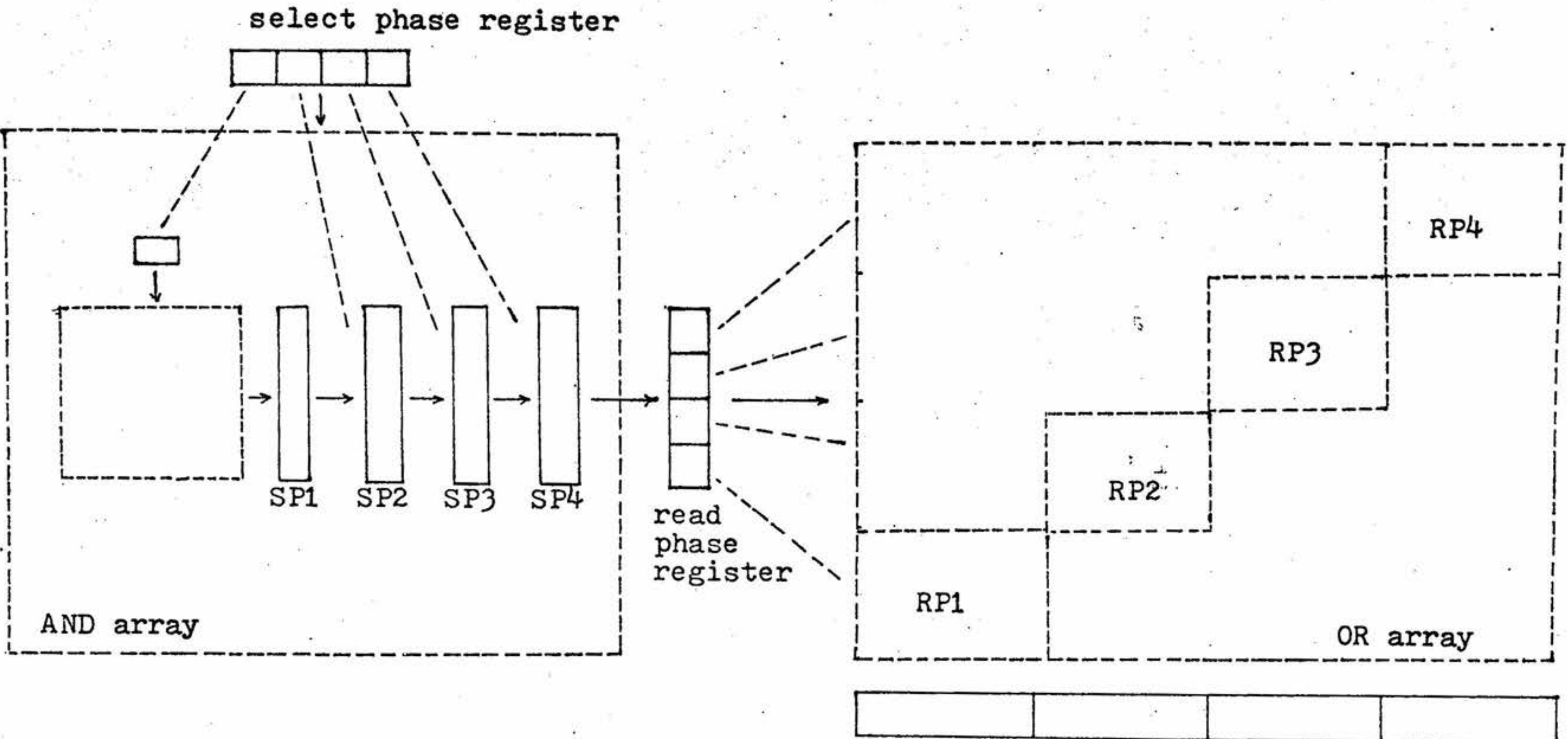


Fig.3.7. Array Logic Viewpoint of Functional Memory Implementation

(note : SP2, SP3 and SP4 have an abbrieviated representation)

RAM implementation. Note in particular that one major feature of this design is that the input has a small bandwidth and the output has a much larger bandwidth (namely, four times the input bandwidth). Another characteristic is that the data stored in the arrays has to bear an intuitive relationship to representations of conventional data structures and operations. With this in mind, we can note the following :

The decoding of 4 bits to 16 lines is a generalization of the 1 bit, 2 line functional memory case. The 16 bit output word can represent the match-nomatch operation of 16 rows of "cells". Each "cell" has a multistate property and any one of a possible 16 input states can generate a "match" with this cell. The ORing of the 4 words from the select phase functional memories represents 4 columns of these multistate cells.

Thus, in the case of the select phase operation, a genuine associative operation of a quite powerful nature is alluded to. The output bits which are loaded into the read phase register represent the match bits of this operation.

The normal array logic case provides for inversion at this point, so that the ORing of mismatch bits is converted into an ANDing of match bits. This is not done here because the next decoding operation can provide for this implicitly by the arrangement of the data in the read phase functional memories.

The "OR-array" is required to generate a 64-bit word. A standard array logic approach would thus lead to

16x64 bits being used to provide this microinstruction. A bit set in the input word (read phase register) would cause a 64-bit word to be ORed into the "microinstruction register". Only 16 standard microinstructions and their ORed combinations could be provided however and this would be a serious limitation to the generation of control signals.

The separation of fields technique was thus employed to provide an extremely flexible means of storing "parts" of microinstructions in this array. This separation is made by generating 16-bit subsections of the microinstruction by decoding 4-bit fields of the read phase register. Such an approach still leads to a full utilization of the same number of bits = 4x16x16. As shown in Fig.3.7, the effect is similar to that of an OR-array of four times the present size, for the same number of microinstructions.

3.4.4.3. The Expected Advantages of the MCU.

Finally, we summarise below, drawing from the previous discussions of microprogramming, a number of the expected advantages of applying the above design techniques to microprogrammed control. Note again that the practical implementation has had to compromise some of these ideas which could have been more fully implemented if the constraints on the circuitry were relaxed.

(1) This technique provides horizontal microinstructions with an optimum encoding of microorders.
(2) Parallel operation of multiple resources is allowed. The practical implementation may not always allow

this parallel operation due to the number of busses employed.

(3) Monophase operation is provided. Again, polyphase operation may be required in some parts of the practical implementation due to the single bus structure of the MCU.

(4) Residual control storage requirements are substantially eliminated and provided implicitly by the MCU.

(5) The separation of the control process from the data process is achieved.

(6) The necessity of designing part of the MCU whilst microprograms or microorders are being considered is substantially reduced.

(7) The capability for dynamically loading microprograms is provided.

(8) The length of the microprograms to support user instructions is minimised.

(9) An implicit incrementing and decrementing capability is provided in sequencing through microinstructions.

(10) The separation of fields of the microinstruction is provided with demonstrable advantages.

(11) Branching delays in instruction execution are reduced.

(12) Separate sources of control for functional modules are provided.

(13) The approach to designing the MCU allows an orderly design sequence, namely functional modules, micro-programmed control unit, microorders and then micro-programs.

(14) The minimal amount of support logic for the MCU is

required.

(15) The small amount of memory required for microprograms allows compact functional memories of minimum size.

(16) Partitioning problems are simplified since all partitioning both of hardware and software is on the basis of function.

The following chapters will provide comments on whether or not these expected advantages were realized.

CHAPTER 4

THE DESIGN OF AN ASSOCIATIVE PROCESSOR

4.1. Introduction.

This chapter covers in more detail, the design and development of the associative processor. The overall system design is first covered, in sufficient detail to demonstrate the capabilities of the processor. Then the applications and the AP application programs are described.

4.2. Associative Processor System Design.

Appendix A gives a more detailed description of the control flow of the associative processor microprogrammed control unit. The requirements of the various microorders used directly influenced this control flow, as did the way in which the functional memories were used. These micro-orders were in turn directly influenced in their choice, by the design of the PEs and the requirements for controlling a number of them in parallel.

4.2.1. Control Flow of the AP.

First of all, some means of performing I/O between the AP and the SP was required. A series of RTM control

modules were used to provide this input and output.

Another control flow "module" is associated with the loading of the functional memories. Upto sixteen different microprograms can be stored in the SP. Each can be loaded dynamically into the MCU. This takes place automatically on a user instruction fault and the microprogram storage area in the SP can be considered as virtual microcode for the associative processor, which can only hold one microprogram at a time.

The major control flow operation is called the interpretation "module". It is associated with both the fetch and execute phases of instruction interpretation. In addition to reading in the instruction to internal MCU storage, instruction faults are resolved and the microinstruction interpretation cycle using the functional memories is set up. Here, a number of operations which ideally should take place simultaneously, are treated sequentially due to the requirements of the interconnection structure and the single-bus MCU.

In addition to sending out the appropriate part of the microinstruction to the PE control lines, MCU microorders have to be executed. Again, tests have to be made of each bit sequentially, which could have been implemented in parallel in a more ideal machine.

Finally, the mode of interpretation field is checked, and this determines the final stage in generating the next microinstruction address for the Select Phase Register. An escape mechanism must be provided for stopping this minor interpretation cycle and returning to receive the

next user instruction, and this is provided for at this point.

4.2.2. Processing Element System Structure.

Each PE has the following RTMs :

(1) Bus Sense Module (M7304)

(2) ALU - registers (M7301)

(3) ALU - control (M7300)

(4) 16x16 Scratchpad Memory (M7318)

(5) General Purpose Interface (M7311)

(6) Bus Termination Module.

(1) and (6) provide the single-bus structure of the PE. (2) and (3) provide the processing power and (4) provides the general storage. The B register provided by (2) is considered to be primarily for the microprogrammer's use whilst the user programmer has general use of the memory provided by (4) and the A register (accumulator) provided by (2). (5) is the means for sending and receiving information to and from the MCU and the other PEs.

Also associated with each PE are Enable Buffers which allow the reception of control signals by only that PE, if an enable signal is being sent from the Enable Register of the MCU. Ideally, rather than using an Enable Register, one flag module should be associated with each PE. However, the difficulties of controlling this flag module from the MCU required that more direct control be provided.

Similarly, the Vertical Register and the Status Register

in the MCU could be considered as the grouping of one flag for each of the PEs.

The Bus Sense Module provides four status signals. A special status section for each PE, under the control of a microorder, merges this information into one bit which is sent back to the MCU as one of the bits of the 16-bit MCU input word. Actually, such positional information is not required if flag modules were used, and the one-bit status information could be merged with that of the other PEs to provide a one-bit status message sent back to the MCU. This is because only general information about a block of PEs is required in the SP such as that at least one of the PEs satisfies some condition. Such information can change the control flow in an AP program by conditional branches. Specific information about PE status is more easily handled at the local level.

A diagram of the data structure of the PE has already been given in Fig.2.9 of Chapter 2.

4.2.3. Associative Processor Instruction Design.

The PEs can provide a number of arithmetic processing capabilities for operations that take place in parallel. In addition, associative operations are implemented which perform comparisons in each of the PEs. These comparisons are between words in each PE and the MCU, or between words of different PEs. The results of these associative operations can remain within the PE, or be sent back to the AP, or be sent back to the SP. Thus, the SP can use

this information for control operations or the AP can use this information for setting up any of the PE-type registers.

Instructions also have to be provided for input to the PEs. Since in general, all the PEs would be filled with data from the SP for each processing sequence, it was felt that a general input instruction would be useful for filling the same number of words in each of a contiguous number of PEs. Each PE would be successively enabled by the Enable Register, and then a series of microinstructions would transfer data from the SP to the AP and then from the AP to the PE.

Similarly, output would require a general output instruction which output a series of words from each of a series of PEs. In this case, however, the PEs involved are not necessarily contiguous but would most probably be selected by some associative criteria. In order to select which PEs should have output performed for them, the Vertical Register was used. To simplify SP programming and AP microprogramming and use the same microinstructions as those for input, it was necessary to step through all the PEs (by shifting a 1-bit through the Enable Register) and use the corresponding bit value of Vertical Register to inhibit the output of that PE if necessary. A better method would have been to have skipped down to the next PE with a bit set in the Vertical Register. The complexity of the logic circuitry for this operation precluded its implementation.

In any case, this is one area of the design of an AP, where custom logic is extremely useful. The Goodyear

Aerospace Corporation STARAN processor performed the resolution of a multiple-response, which is what is required here, by performing a match operation for the minimum key in each PE. This is probably even slower then any software control of Enable Registers. Such problems of multiple responses have been noted in the early application of associative memories to the provision of fast mapping of virtual addresses to real addresses (Aspinall, Kinniment and Edwards, 1968). Such techniques as those proposed by Foster (1968) can be applied to the logic level, to select one response out of a number of responses.

4.2.4. The Sequential Processor Support.

The sequential processor gives considerable support to the determination of user instruction sequence and data input and output for the application programs. We cover at this point, the program that runs in the SP and show how this support is achieved.

The program that is run in the SP for the first application is given in Appendix A. It contains within it, the support routines for the AP. A summary flowchart and a more detailed flowchart of these routines are given in Fig.4.1.

In the program for the first application, there is a section labelled PROG. Here is found the instructions for the AP. They are given in hexadecimal and are followed by a comment which is the AP "assembler" version of the

Fig.4.1. Overview of the Support Program for the AP

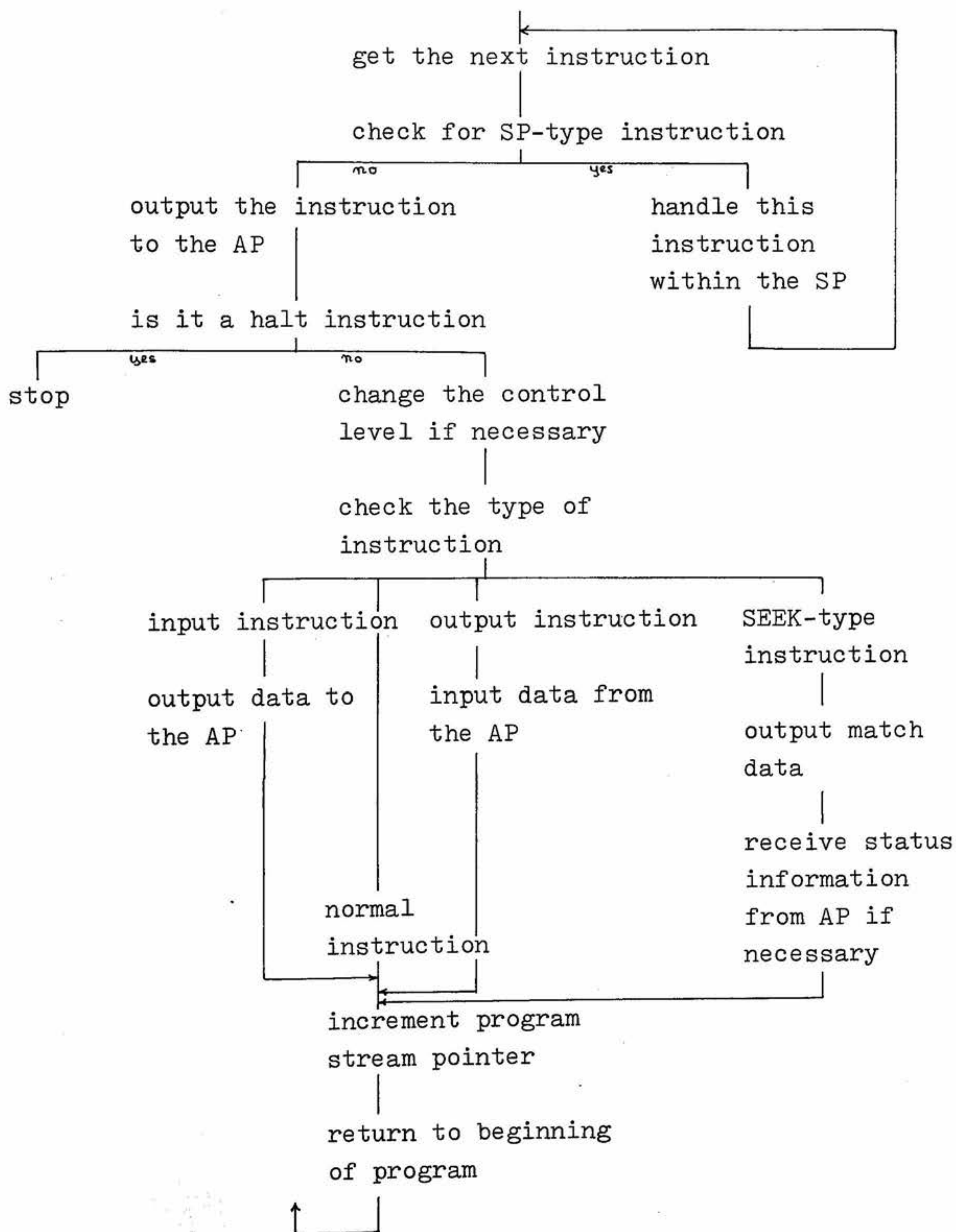


Fig.4.1. (cont.) The Support Program for the AP

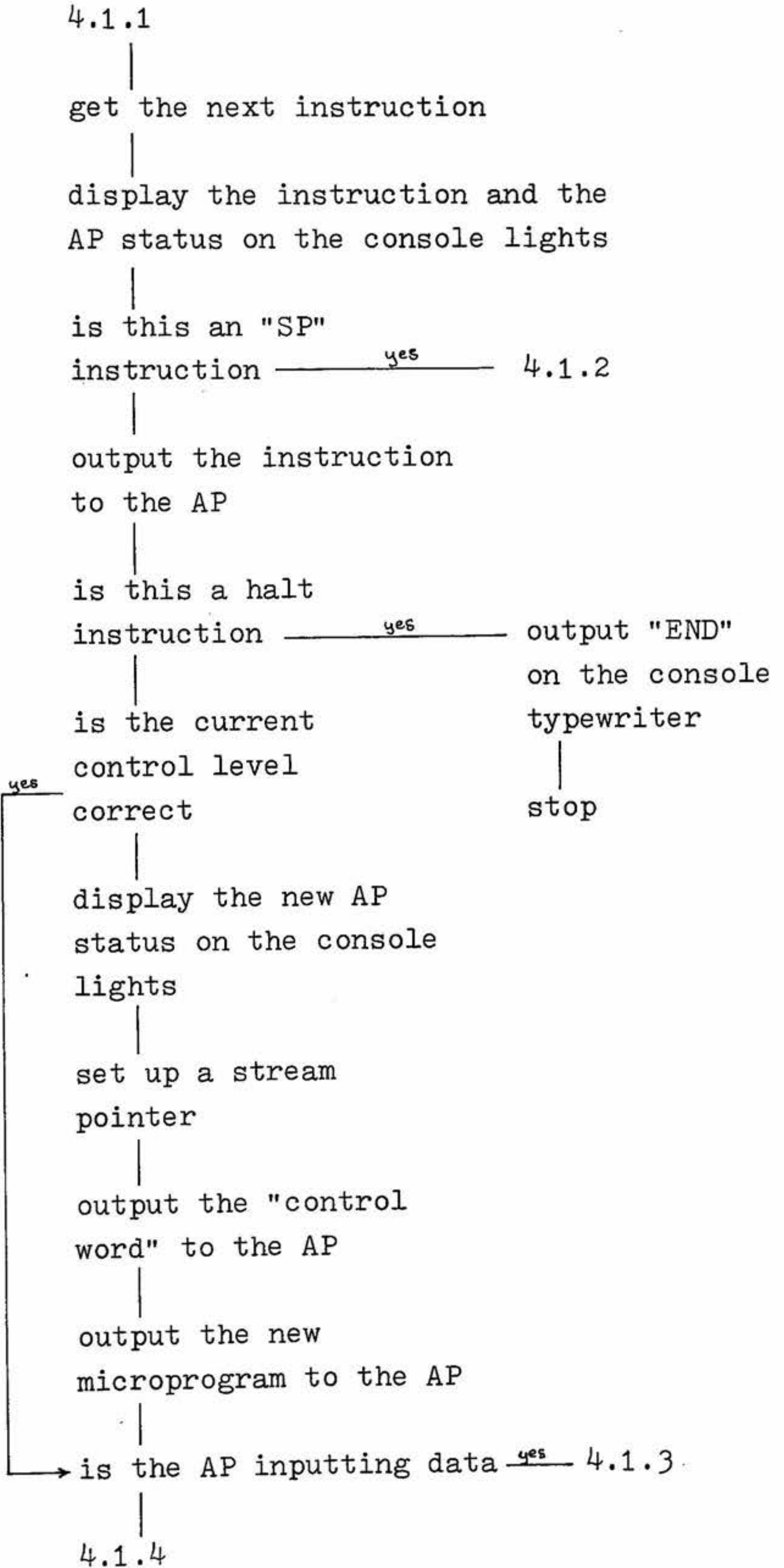


Fig.4.1. (cont.) The Support Program for the AP

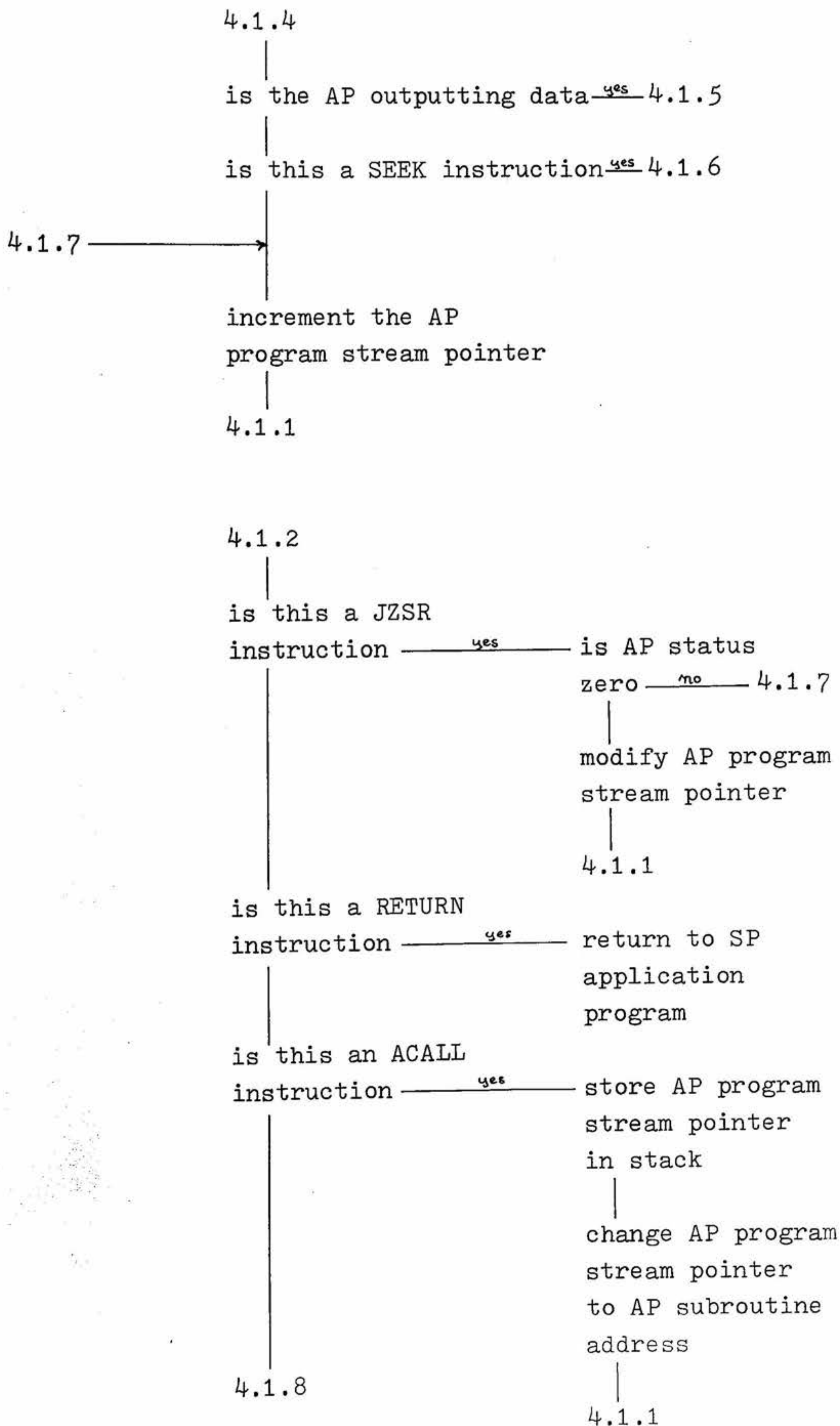
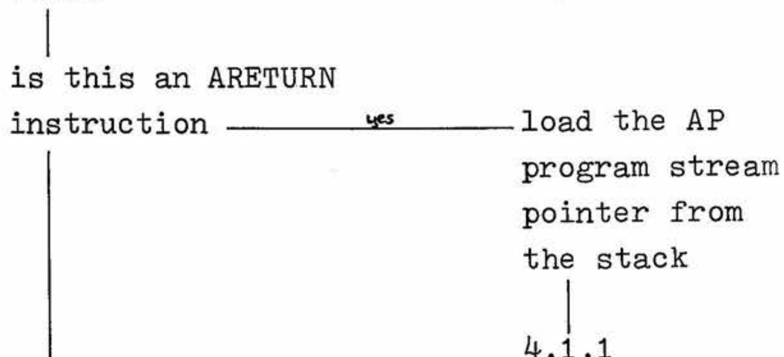
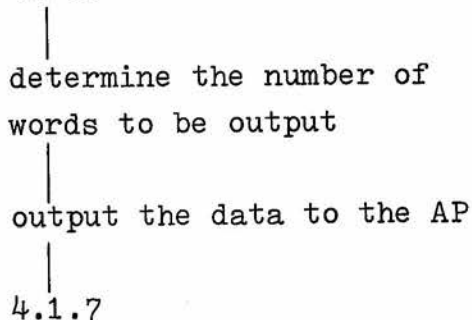


Fig.4.1. (cont.) The Support Program for the AP





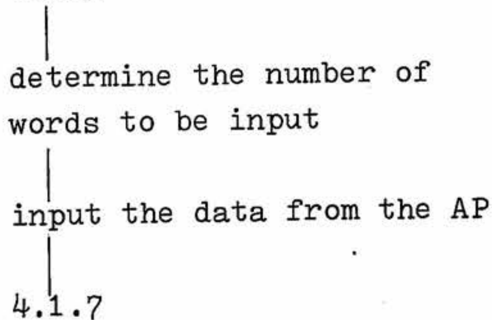


Fig.4.1. (cont.) The Support Program for the AP

4.1.6

output the data for the SEEK operation to the AP

is a new AP status to be returned — no → 4.1.7

input new status from the AP

display AP status on console lights

4.1.7

instruction. Other comments may also follow. Sections of AP code may also be labelled and AP instructions may reference these labels for CALLs and JZSRs and other appropriate instructions. HAL, the Interdata assembler language that was used, assigns the appropriate values for these references.

The AP support program's main task is to fetch the AP instructions and to transmit them to the AP. The instructions are considered as a stream and there is a program stream pointer used for accessing them (PSTRM).

First, the next instruction is fetched. It is displayed on the first row of the console lights and followed by the SP "status word" on the second row of lights. A check is made to see whether this AP instruction can be executed solely within the SP. Such an instruction would have "8" in the fourth 4-bit field of the user instruction, the control field. The program jumps to a special routine for handling this instruction if this is the case.

Otherwise, the instruction is transmitted to the AP along the transmitter link. It is then checked to see whether it was a null instruction (the halt instruction). If so, the SP must also halt.

At the completion of the instruction fetch cycle, checks are made to see whether the SP must do any further processing for the AP. If not, the program stream pointer is incremented and the instruction interpretation routine commences again.

The microprograms are given in a section of the program labelled FM (for functional memories). Of the

sixteen microprograms anticipated, only four were required for the two applications considered. Control level 8 is provided as a pseudo-control level when only SP operation is required for an AP operation (a modification of the program stream pointer, for example). A possible 16 microprograms could be contained in the SP and ideally these would be stored in the MCU if there was enough memory available.

The current microprogram loaded in the AP is given in the last 4-bit field of the SP "status word". After the new instruction has been output to the AP, a check is made to see whether a new microprogram must be loaded for this instruction to be executed. This is done by comparing the 4-bit field of the status of the AP against the same 4-bit field of the new instruction, which gives the number of the microprogram required for execution. If they are the same, then nothing need be done. If they differ, then the following operations occur. First, the SP "status word" is updated and displayed on the second row of console lights. Then a stream pointer to the microprogram to be loaded is set up. A control word which is very useful in the MCU for loading the functional memories is then transmitted to the AP. Using the stream pointer, the appropriate microprogram is output along the link. In turn, the AP does a check of the new instruction to see whether it must input a new microprogram. Its status register is also updated if necessary. Note that the same word of each scratchpad memory is loaded before going on to the next word of the scratchpad memories. That is,

the scratchpad memories are loaded "row-wise". This was found to be more efficient from a hardware point of view than loading each scratchpad memory completely one after the other.

We have already mentioned that after sending out the user instruction, the SP checks for any other operations required for assisting the AP. A check is made to see whether a READ or WRITE instruction has been sent to the AP. If so, the SP must output the required data or read in the data words provided by the AP. In this case, the number of words involved in the operation is determined. The output stream pointer or the input stream pointer is used for these operations and must have been set up by the application program in the SP before running the AP. The operation of the AP must stop if these pointers are to be changed in any way.

Note, in particular, that an input of X'FFFF' is ignored. There are a number of reasons for this :

(1) It must be possible for the write operation from the AP to be selective. That is, of all the PEs, it must be possible to select a subset from which output is to occur. Ideally, it should be possible to suppress the appropriate output signals in this serial operation. A parallel I/O operation would only require the selective enabling or disabling of the appropriate PEs. The amount of unavailable circuitry required together with the fact that negative logic was being employed which would allow X'FFFF' to be a suitable "mask" prompted this approach.

(2) The applications being considered were relatively non-

numeric in their output from the AP to the SP, and X'FFFF' was a value that would not be employed as a non-numeric value. However, numeric calculations did crop up in the first application and this approach had its disadvantages. (3) This is an area where custom logic may indeed be necessary, namely as mentioned before, the resolution of multiple responses. Special microprogramming techniques would not be necessary.

A more appropriate approach may have been to have some special and unique non-numeric code or better still to suppress the I/O operation at the serial transmitter stage. The ability to skip over the PEs to be ignored was difficult to achieve with this implementation, especially with the simplification that the SP was expecting a fixed number of data words input in a WRITE operation.

We now consider other special operations that the SP must perform. Certain SEEK instructions affect the values in the Enable Register, the Vertical Register or the Status Register in the AP. Other SEEK operations affect the Status Register of the SP. The SP must determine whether the operation output to the AP was a SEEK instruction. If this is so, the data word for the test must also be output and this is the next word in the instruction stream. A check is then made to see whether a new status value is to be returned. If so, the new value is received and displayed on the console lights.

Certain AP instructions deal with the modification of the instruction stream and can be handled completely within the SP. The SP checks the current instruction for

this case and branches to a routine for handling this. These special instructions are as follows :

(1) The instruction JZSR means jump if the SP Status Register is zero. Otherwise, continue with the next instruction.

(2) The instruction RETURN means return to normal SP operation. The AP ceases to operate unitl the normal SP application code includes a call to the AP support program again.

(3) Normal AP code can contain subroutines and ACALL and ARETURN are available to enter and exit from such subroutines. A 10 word instruction address stack is available in the SP and the return values of the AP program stream pointer are stored here on calling a subroutine.

The operation of the SP is completely independent of the AP. Normal programs can be obeyed and when necessary, the program stream pointer and the input and output stream pointers can be set up and the SP operation switched to supporting the AP. In this mode, the SP is operating as the "fetch" processor for the AP, supplying instructions, microprograms and data and receiving data and status information.

The serial link transmitter to the AP can be transmitting a byte of information whilst storing the next byte for transmission. The SP proceeds with instruction fetching and possible interpretation as far as possible until forced to wait for the operation of this transmitter or for input to occur.

Thus, whilst one instruction is being transmitted to the AP :

(1) the next one can be available,

(2) the appropriate microprogram can be determined and set up for output,

(3) the appropriate input or output operation can be set up,

(4) the serial receiver can be set up to receive a status word, and

(5) all program stream modifications can be completed and the next AP instruction ready for output by the time the AP has completed the current instruction execution.

The SP and AP are therefore to a certain extent working in parallel. All delay associated with program modification is eliminated since this takes place on information supplied well before required. This is not to imply that the Interdata processor is not delayed in testing the appropriate locations of its memory but that the AP receives a constant stream of instructions and data and does not have to wait longer than determined by the physical requirements of the serial transmitter and receiver. Detailed tests of this were not carried out, but the speed of the AP processor did seem to be limited by the speed of the serial links. The AP speed was affected substantially when these serial links where speeded up in their operation.

In addition to the consoles of both the AP and SP, the user interfaces to the system via a console typewriter and a lineprinter. All input from the console typewriter and output to both console typewriter and lineprinter are handled at the interrupt level without any noticeable delay

to normal AP and SP operation.

4.3. The Air Traffic Control Application.

This application was chosen because it is one of the classical problems where parallelism in processing has indicated the usefulness of programming an associative processor. This application was the simulation of a two-dimensional air space with a number of aircraft in it, where the AP was to provide a capability for air traffic control conflict detection. Avoidance or conflict resolution was not considered.

This was the first application to be considered and so had a more critical effect on the design and instruction set development than the second application. It was from a determination of the operations of the stages of conflict detection that the required operations of the AP became clear.

4.3.1. General Approach.

The general approach that was followed was to divide the conflict detection problem into three stages. First, a general prediction is made of the position of each aircraft in 64 seconds time. Secondly, a filtering is done of those pairs of aircraft which are in close proximity to each other. The final stage is a more complex prediction test on these selected pairs of aircraft.

The first and last stages are mostly the application

of algorithms on parallel streams of data. The second stage performs a general associative operation to provide a filtering or screening process.

4.3.2. Stage 1.

We now cover the programming of the application in more detail than described in Chapter 2.

Fig.4.2 provides a flowchart of the first part of the SP operation for this stage. Appendix A gives the program used and example output from both the lineprinter and the console typewriter. The output stream pointer is set up to point to the aircraft data to be processed. The input stream pointer is set up to the location in core memory from which the result of the AP calculation is to be stored. The program stream pointer is set to point to the start of the AP instructions for processing in this stage. Finally, the AP operation is initiated.

The data for this operation is in DATA BLOCK 1. Each aircraft is represented by 5 words in scratchpad memory :

(1) word 0 - the aircraft identification number.

(2) words 1 and 2 - the position of the aircraft.

(3) words 3 and 4 - the velocity of the aircraft.

Note that the simulated air space is two-dimensional for simplicity. The position and velocity are given in arbitrary units and adjusted so that the display produced by the lineprinter looked realistic.

An aircraft identification number of zero signifies

Fig.4.2. SP Program for Stage 1 of Air Traffic Control Application (first part)

set up SP output stream pointer to air space DATA BLOCK 1

set up SP input stream pointer to air space DATA BLOCK 2

set up AP program stream pointer to AP program for stage 1

commence AP support operation

the end of the data stream for the AP. Note that enough data must be provided (even if zero) to fill all the PEs each time a read instruction is executed.

The data output by the AP which is used in turn to generate the input data for the next stage of the operation is given by the AP in DATA BLOCK 2. Each aircraft is represented by 5 words :

(1) word 0 - the aircraft identification number.

(2) words 1 and 2 - the aircraft position for the lineprinter.

(3) words 3 and 4 - the aircraft box position.

An 8 x 8 box array is being considered and after the prediction calculation has been made, each aircraft is allocated to a box in this box array. In addition, the AP provides the unpredicted position of each aircraft in a 64 x 64 element array for the lineprinter, using the second and third words of this 5-word block of data for each aircraft.

The position values are 15-bit integers. The velocity values are 6-bit integers and thus a consideration of velocities from -64 units/sec to 63 units/sec is allowed.

We now treat the operations that take place in the AP. Fig.4.3 shows the processing involved. First, the Enable and Vertical Registers are reset. Then five words of data are read into each PE. All PEs are enabled for the following parallel operations. Calculations are performed for allocating the x-coordinate box position. Then the operations for the y-coordinate are done. This is followed by the operations for determining the aircraft position in

Fig.4.3. AP Program for Stage 1 of Air Traffic Control Application

4.3.1

reset enable register for I/O

reset vertical register

input air space DATA BLOCK 1

switch to parallel operation
(all PEs enabled)

calculate box position :
X = ((x' lshft 6)+x) rshft 12
Y = ((y' lshft 6)+y) rshft 12

calculate aircraft position
for lineprinter :
LX = x rshft 9
LY = y rshft 9

associative search for aircraft
identification no. of 0
set SP status register

reset enable register for I/O

output air space DATA BLOCK 2

is SP status register zero —— yes —— 4.3.1

return to normal
SP operation

a 64 x 64 element array for the lineprinter. The test for the end of the data stream is then performed. Finally, the Enable Register is reset for an I/O operation and the computed data is written out to the SP. A test is then made within the SP to check for the end of the data stream and if not then the program loops back to the beginning. Otherwise, SP application program execution commences again.

The SP operation continues as shown in Fig.4.4. The SP activates the lineprinter which asynchronously writes out the aircraft position using a chart. The SP continues by assigning aircraft to boxes using the data supplied by the AP. Four words of storage are given for each box in the array. Each word is initially zero. Each aircraft identification number is placed in the appropriate "box" using the first of the zeroed words available. If an attempt is made to place an aircraft identification number in a "box" already containing four identification numbers, overflow has occurred and operation must cease. This condition did not occur with the large number of aircraft considered in the air space. The data generated forms DATA BLOCK 3 and contains the "box" information column by column.

4.3.3. Stage 2.

Again the input and output stream pointers are set up and operation of the AP is continued. The program stream pointer is already set up to the required value.

The AP operation at this point is shown in Fig.4.5.

Fig.4.4. SP Program for Stage 1 of Air Traffic Control Application (second part)

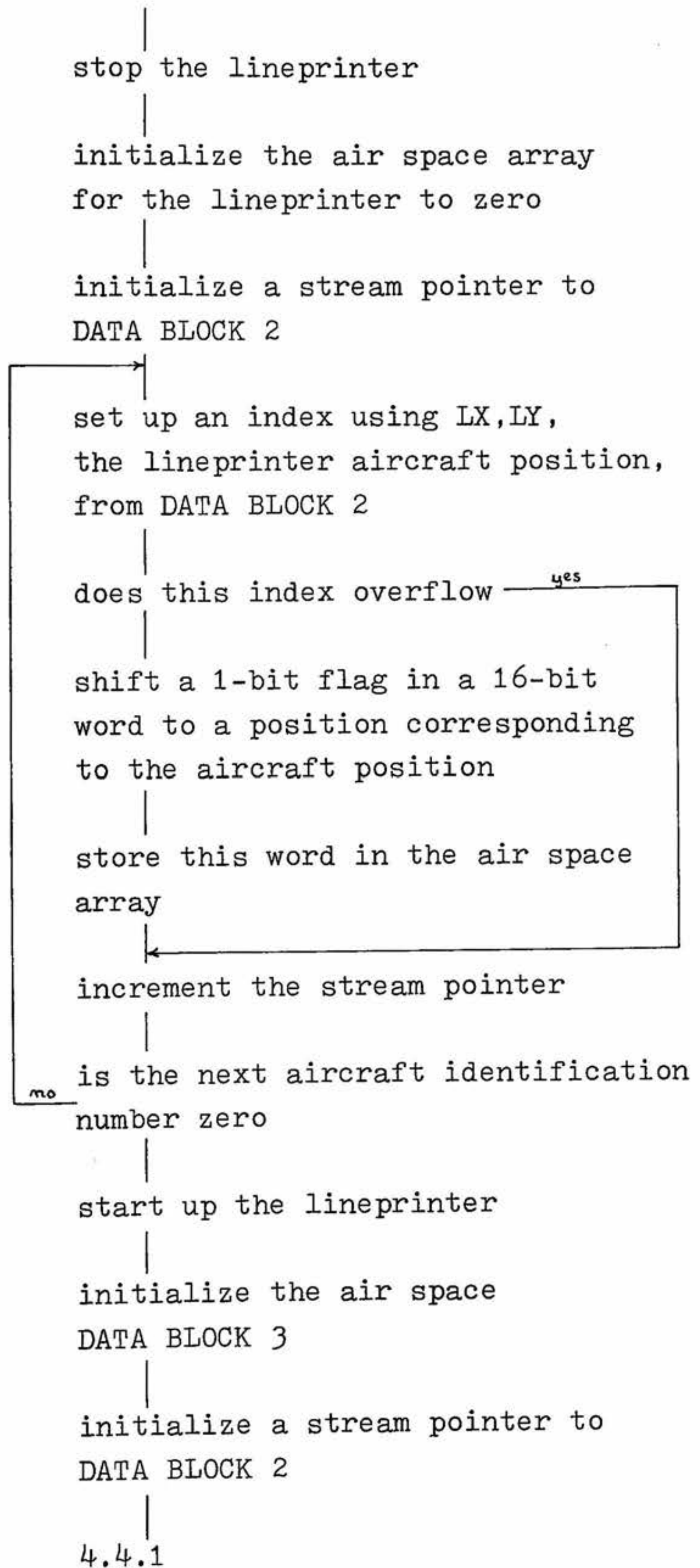


Fig.4.4. (cont.) SP Program for Stage 1 of Air Traffic Control Application (second part)

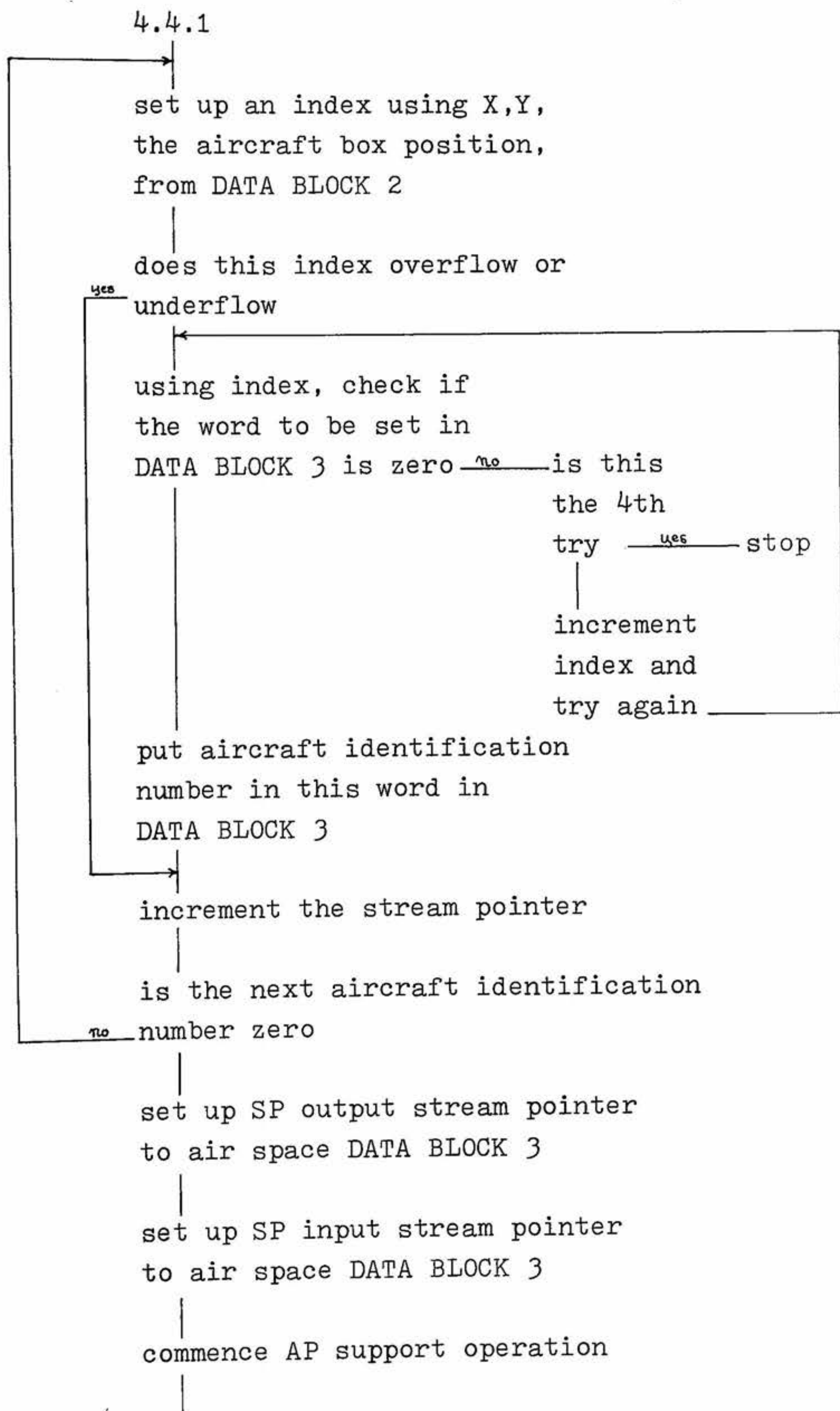


Fig.4.5. AP Program for Stage 2 of Air Traffic Control Application

```
   |
reset enable register for I/O
   |
input air space column 1
from DATA BLOCK 3
   |
switch to parallel operation
(all PEs enabled)
   |
transpose data :
 word 8 = word 0
 word 9 = word 1
 word 10 = word 2
 word 11 = word 3
   |
call ASSOC (see Fig.4.6.)
- columns 1 and 2 checked
   |
call TRANSP (see Fig.4.7.)
   |
call ASSOC
- columns 2 and 3 checked
   |
call TRANSP
   |
call ASSOC
- columns 3 and 4 checked
   |
call TRANSP
   |
call ASSOC
- columns 4 and 5 checked
   |
call TRANSP
   |
call ASSOC
- columns 5 and 6 checked
   |
4.5.1
```

Fig.4.5. (cont.) AP Program for Stage 2 of Air Traffic Control Application

```
4.5.1
   |
call TRANSP
   |
call ASSOC
- columns 6 and 7
   |
call TRANSP
   |
call ASSOC
- columns 7 and 8
   |
call TRANSP
   |
load 9th "column" of zeroes :
 word 4 = 0
 word 5 = 0
 word 6 = 0
 word 7 = 0
   |
call ASSOC1 (see Fig.4.6.)
- columns 8 and "9" checked
   |
return to normal
SP operation
```

First of all, the Enable Register is reset and the "boxes" are read into the PEs, one box per PE. Thus, each PE receives four data words, representing upto four aircraft identification numbers. The first four words of each PE are used, and the first "column" of boxes are the initial input.

The Enable Register is reset to enable parallel operation. The next few instructions place the data read into the first four words of each PE into the third four words of each PE. The rest of the program consists of calls to the routines ASSOC and TRANSP, described in Fig.4.6 and 4.7, respectively.

ASSOC reads in the next column of boxes and places them in the second four words of each PE. A number of associative operations are then performed. Checks are made for non-zero words representing aircraft. The routine TRANSP is called at the end of each such operation. It simply moves the columns along from the second four words to the third four words. The end of the box array required special treatment. Instead of the reading of a ninth column, a dummy column of zeroes is set up. This is an efficient use of the code since a number of associative tests must still be made on the eighth column alone and the routine ASSOC (or part of it) can still be used.

ASSOC as mentioned sets up the next column in the second four words of each PE. The set up is omitted the last time by calling ASSOC1 instead, which is also shown in Fig.4.6.

There are four basic tests :

Fig.4.6. The AP Routines ASSOC and ASSOC1

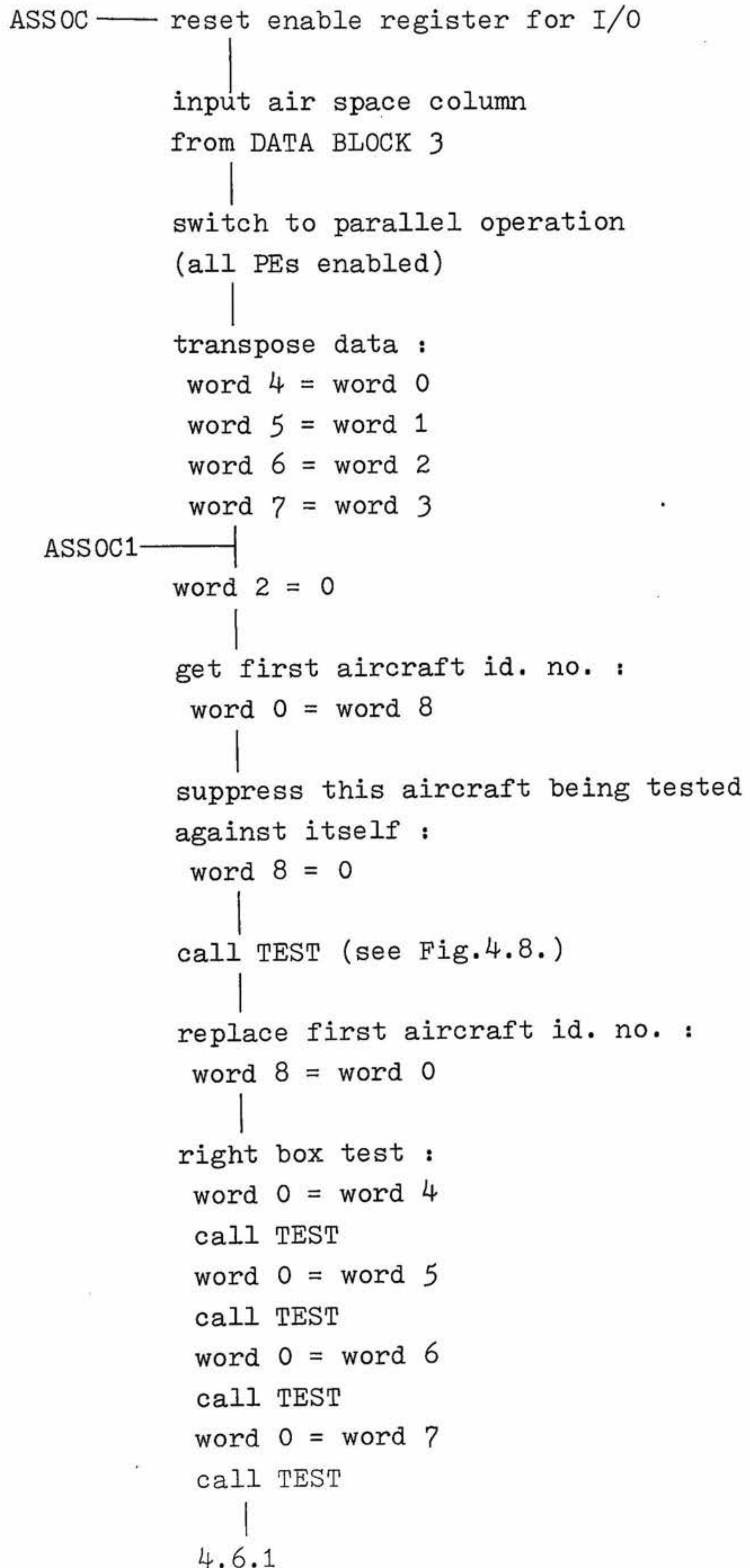


Fig.4.6. (cont.) The AP Routines ASSOC and ASSOC1

```
4.6.1
  |
top box test :
 - using the routine TRNS (see Fig.4.9.)
 word 0 = word 8 from neighbouring PE
 call TEST
 call SPOT (see Fig.4.10.)
 word 0 = word 9 from neighbouring PE
 call TEST
 call SPOT
 word 0 = word 10 from neighbouring PE
 call TEST
 call SPOT
 word 0 = word 11 from neighbouring PE
 call TEST
 call SPOT
  |
top-right box test :
 - using the routine TRNS
 word 0 = word 4 from neighbouring PE
 call TEST
 word 0 = word 5 from neighbouring PE
 call TEST
 word 0 = word 6 from neighbouring PE
 call TEST
 word 0 = word 7 from neighbouring PE
 call TEST
  |
return
```

Fig.4.7. The AP Routine TRANSP

```
TRANSP——transpose data :
         word 8 = word 4
         word 9 = word 5
         word 10 = word 6
         word 11 = word 7
         |
        return
```

Fig.4.8. The AP Routine TEST

```
TEST ———test aircraft id. no. in word 0
        for nonzero value and set AP
        status register
           |
        check the current air space box
        against this test :
         word 1 = word 8
         call T (see Fig.4.11.)
         word 1 = word 9
         call T
         word 1 = word 10
         call T
         word 1 = word 11
         call T
           |
        return
```

(1) A check is made for aircraft in the same box.

(2) A check is made for aircraft in a box and aircraft in the box to the right.

(3) A check is made for aircraft in a box and aircraft in the box above.

(4) A check is made for aircraft in a box and aircraft in the box to the right of the box above.

For each of these tests, a word to be checked against the words already in the third set of four words is put into word 0 of the scratchpad memories. The routine TEST, shown in Fig.4.8, is then called. Some of the words to be placed in word 0 may have to come from a neighbouring PE, and in this case, the routine TRNS, shown in Fig.4.9, is used to transmit the appropriate data. The routine SPOT, shown in Fig.4.10, makes use of the data from the box "above" to check against the "right" box. This takes care of all adjacencies. Note that the SEEK instruction is used to initialise the output buffers which may otherwise cause interference. This is necessary because of the limited interconnection structure available.

The routine TEST should now be considered. This routine loads word 1 of the scratchpad memory from the "first" column data (the third set of four words) and calls the routine T after each such load (see Fig.4.11 for a description of the routine T). SPOT is similar but takes its data from the "second" column (the second set of four words). At the start of TEST, a check is made in all PEs to see if word 0 is non-zero and if so the Status Register is set up accordingly. SPOT does not

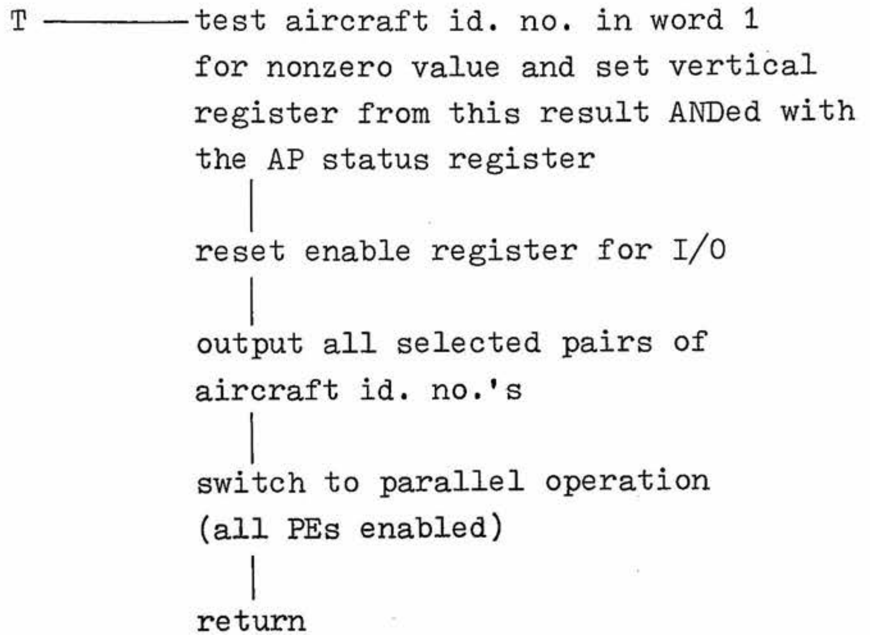
Fig.4.9. The AP Routine TRNS

```
TRNS ——— output the PE accumulator
         (set up the output buffer register)
            |
         input to the PE accumulator
            |
         clear the accumulator of the first
         PE - input not required :
          enable only the first PE
          set the accumulator to zero
            |
         switch to parallel operation
         (all PEs enabled)
            |
         return
```

Fig.4.10. The AP Routine SPOT

```
SPOT ——— check the "right" box against
         the current (word 0) test :
          word 1 = word 4
          call T (see Fig.4.11.)
          word 1 = word 5
          call T
          word 1 = word 6
          call T
          word 1 = word 7
          call T
            |
         return
```

Fig.4.11. The AP Routine T



have to do this since it only has to be done once for each aircraft identification number loaded into word 0 and TEST has already done it.

The routine T then goes ahead and does a similar check on the second word (word 1) and if also non-zero, sets the Vertical Register from this result ANDed with the Status Register. The next operation is the output operation which outputs those pairs of aircraft identification numbers which have been selected. This output forms DATA BLOCK 4.

4.3.4. Stage 3.

Further processing of the data output from Stage 2 by the AP is required in the SP. The data for input to the AP must be prepared in DATA BLOCK 5. Each PE is to receive the information for a pair of aircraft as selected by the second stage of AP operation. This data will be stored in the first 11 words of the scratchpad memory of each PE as follows :

(1) word 0 - the aircraft no. 1 identification number.
(2) words 1 and 2 - the first aircraft's position.
(3) words 3 and 4 - the first aircraft's velocity.
(4) word 5 - the aircraft no. 2 identification number.
(5) words 6 and 7 - the second aircraft's position.
(6) words 8 and 9 - the second aircraft's velocity.
(7) word 10 - the radius constant R.

Then, the AP program stream pointer must be set to point to the program for this stage. Fig.4.12 gives the SP operations.

Fig.4.12. SP Program for Stage 3 of Air Traffic Control Application (first part)

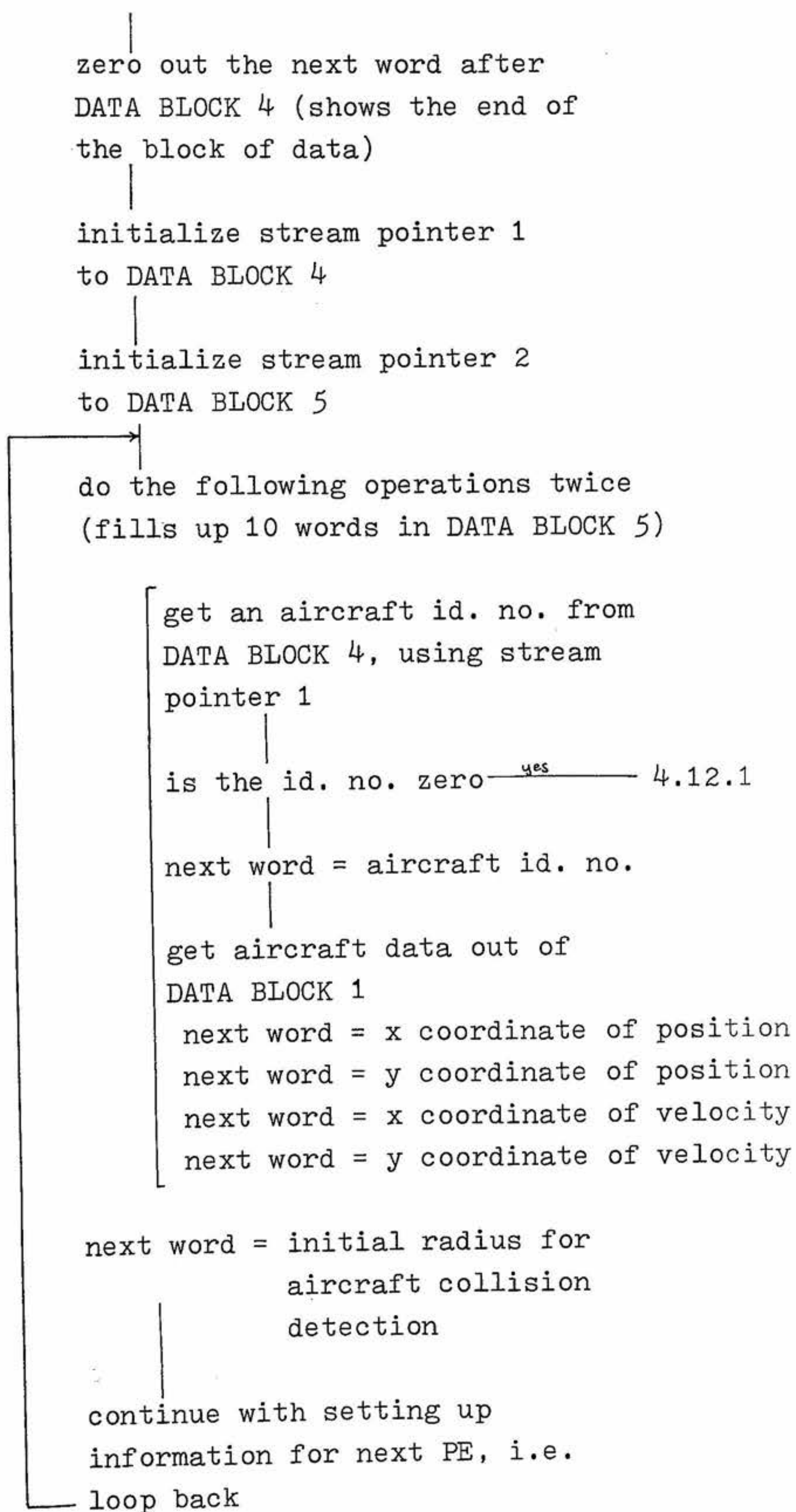


Fig.4.12. (cont.) SP Program for Stage 3 of Air Traffic Control Application (first part)

4.12.1

zero out the next word in DATA BLOCK 5 (shows the end of the block of data)

zero out any extra space in DATA BLOCK 5 for an integral block of PEs

set up SP output stream pointer to air space DATA BLOCK 5

set up SP input stream pointer to air space DATA BLOCK 6

set up AP program stream pointer to AP program for stage 3

commence AP support operation

Next, we treat the AP program for Stage 3. See Fig.4.13 for a flowchart of this program. First of all, the AP calculates two radius values for the aircraft, called R_1 and R_2. These are proportional to the velocity of the aircraft, and are given by using the formula :

$$R_n = |(x'_n - y'_n)\text{lshft } 4| + R$$

where x'_n = the horizontal (x-coordinate) velocity

y'_n = the vertical (y-coordinate) velocity

and R = a fixed radius constant supplied by the SP to each PE.

These radii define two possibly intersecting semicircles in front of the two aircraft, as shown in Fig.4.13. If they do intersect, the pair of aircraft are considered to be on potential collision courses and their aircraft identification numbers are returned to the SP.

In order to determine this, the next section of the AP program calculates r where r is the maximum distance between the two aircraft at which these two semicircles intersect. The program then checks whether the inequality

$$(x_1-x_2)^2+(y_1-y_2)^2-r^2 \leqslant 0$$

is true, where

x_1, y_1 = the coordinates of the first aircraft's position, and

x_2, y_2 = the coordinates of the second aircraft's position.

In this case, the pair of aircraft identification numbers are marked for output by setting the Vertical Register appropriately.

Most of the activity in this stage is parallel operation.

Fig.4.13. AP Program for Stage 3 of Air Traffic Control Application

4.13.1

reset enable register for I/O

input aircraft information
from DATA BLOCK 5
(information on a pair of aircraft
for each PE)

switch to parallel operation
(all PEs enabled)

word 13 (r) = 0

calculate R_1

word 11 = $(x_1'+y_1')$ lshft 4

word 11 = |word 11| :
enable all PEs with Accumulator < 0
Acc = Acc - 2*word 11
switch to parallel operation

word 11 = |word 11| + R

calculate R_2

word 12 = $(x_2'+y_2')$ lshft 4

word 12 = |word 12| :
enable all PEs with Acc < 0
Acc = Acc - 2*word 12
switch to parallel operation

word 12 = |word 12| + R

4.13.2

Fig.4.13. (cont.) AP Program for Stage 3 of Air Traffic Control Application

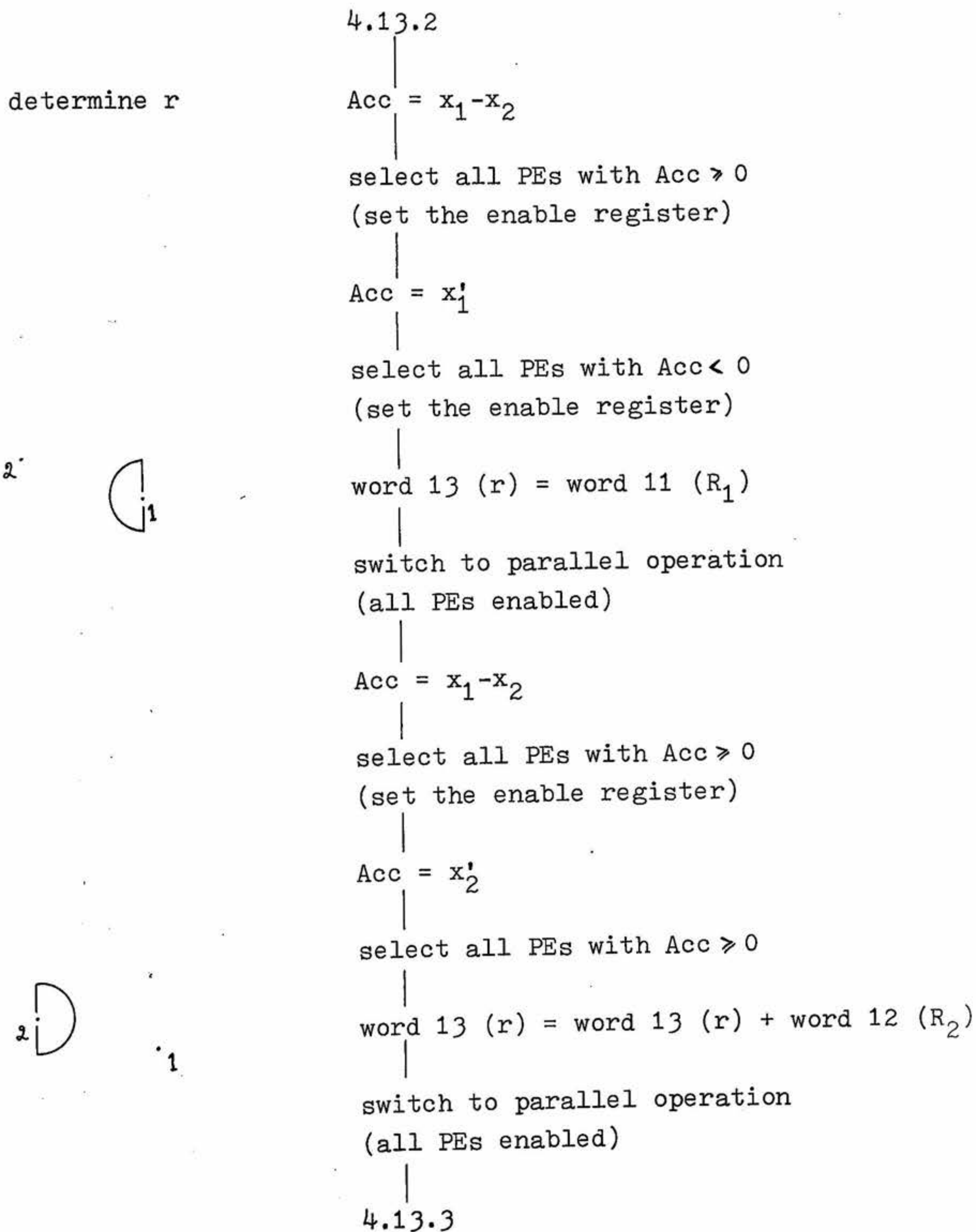


Fig.4.13. (cont.) AP Program for Stage 3 of Air Traffic Control Application

4.13.3

Acc = $x_1 - x_2$

select all PEs with Acc < 0

Acc = x_1'

select all PEs with Acc ≥ 0

1 D · 2

word 13 (r) = word 11 (R_1)

switch to parallel operation
(all PEs enabled)

Acc = $x_1 - x_2$

select all PEs with Acc < 0

Acc = x_2'

select all PEs with Acc < 0

1 · (2

word 13 (r) = word 13 (r) + word 12 (R_2)

switch to parallel operation

determine r^2

word 12 = word 13

Acc = 0

words 10,11 = (word 12 (r))^2

4.13.4

Fig.4.13. (cont.) AP Program for Stage 3 of Air Traffic Control Application

4.13.4

determine $(x_1-x_2)^2$ — word 8 = x_1-x_2

word 8 = |word 8|

word 9 = word 8

Acc = 0

words 8,9 = (word 8)^2

determine $(y_1-y_2)^2$ — word 6 = y_1-y_2

word 6 = |word 6|

word 7 = word 6

Acc = 0

words 6,7 = (word 6)^2

$(x_1-x_2)^2+(y_1-y_2)^2-r^2$ — words 3,4 = words 8,9 + words 6,7

words 3,4 = words 3,4 - words 10,11

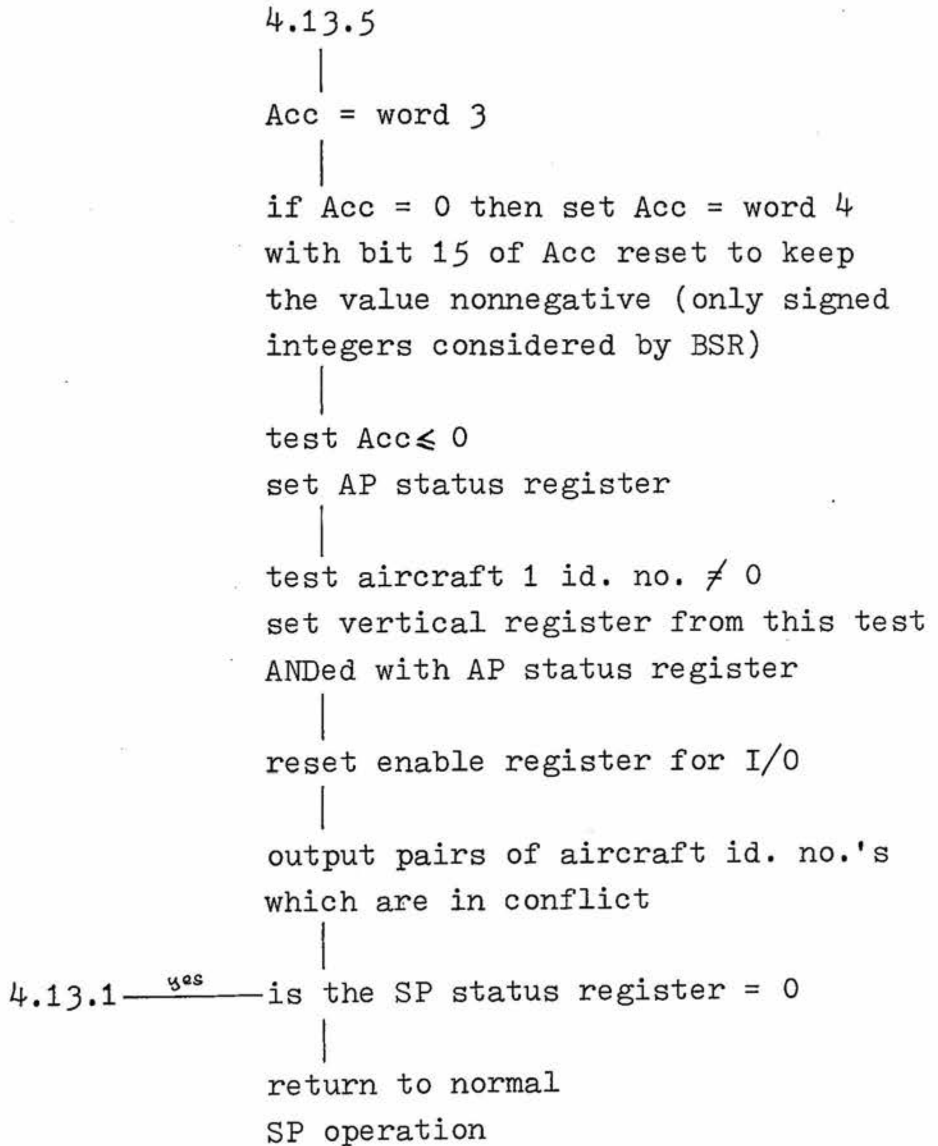
final conflict output operation — reset vertical register

test for end of DATA BLOCK 5
(zero aircraft id. no.)
- set SP status register

word 1 = word 5 (aircraft 2 id. no.)

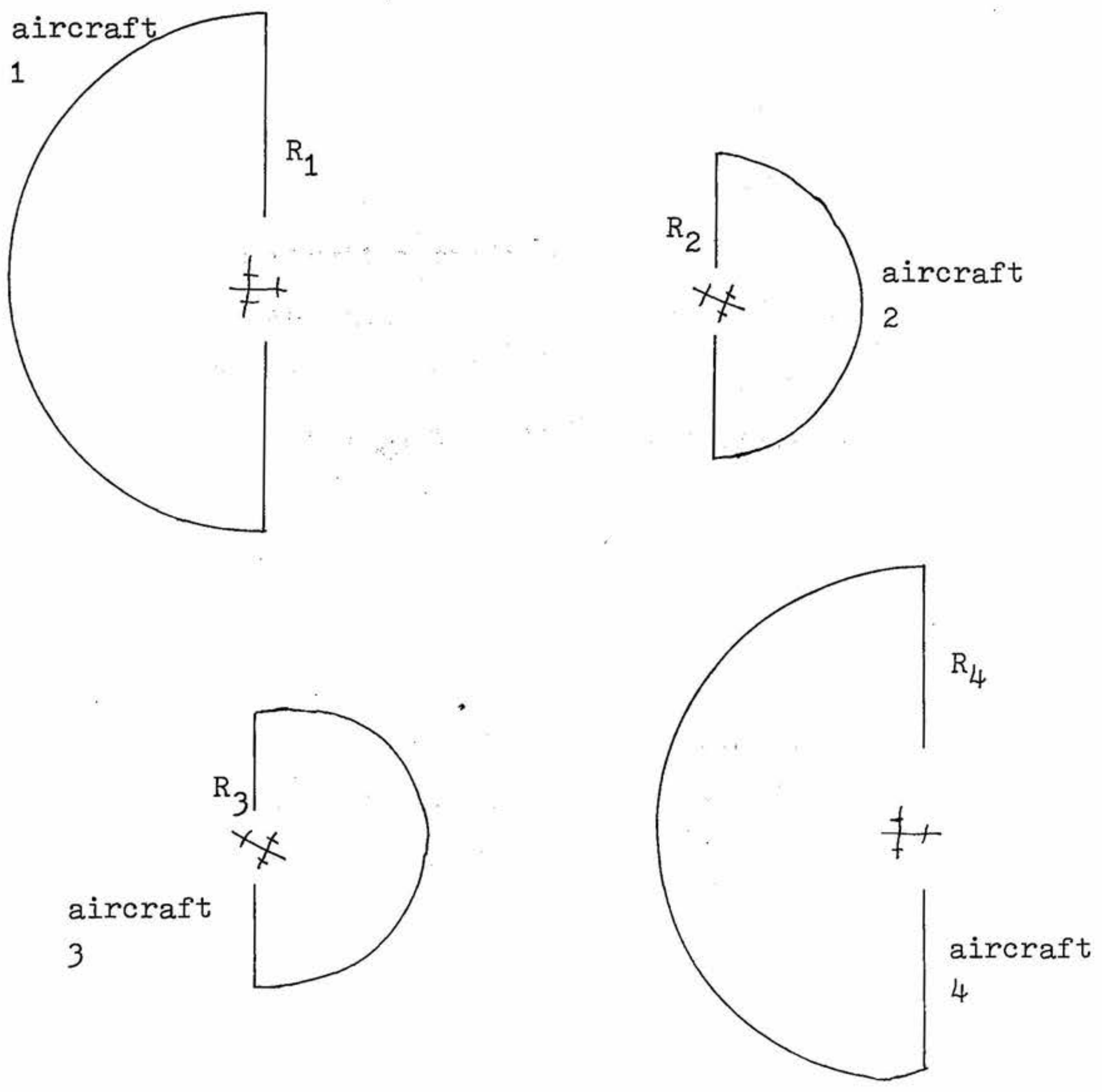
4.13.5

Fig.4.13. (cont.) <u>AP Program for Stage 3 of Air Traffic Control Application</u>



An explanation of the diagrams to the left of the flowchart in the section for determining the value r, is given on the next page.

Fig.4.13. (cont.) AP Program for Stage 3 of Air Traffic Control Application



The semicircles are lefthand or righthand depending on whether the aircraft are going to the left or right, respectively. R_n is proportional to the aircraft velocity. The distance r between two aircraft is determined from R_n of the semicircles encompassing those aircraft if they lie between the two aircraft. For example :

for aircrafts 1 and 2, $r = 0$
for aircrafts 1 and 4, $r = R_4$
for aircrafts 3 and 2, $r = R_3$
for aircrafts 3 and 4, $r = R_3+R_4$

However, there is a section in this stage when the value r is being determined, where tests have to be made on whether values are positive or negative and this leads to the use of a binary decision tree. The availability of a stack of flags representing a stack of Enable Registers would have been very useful at this point.

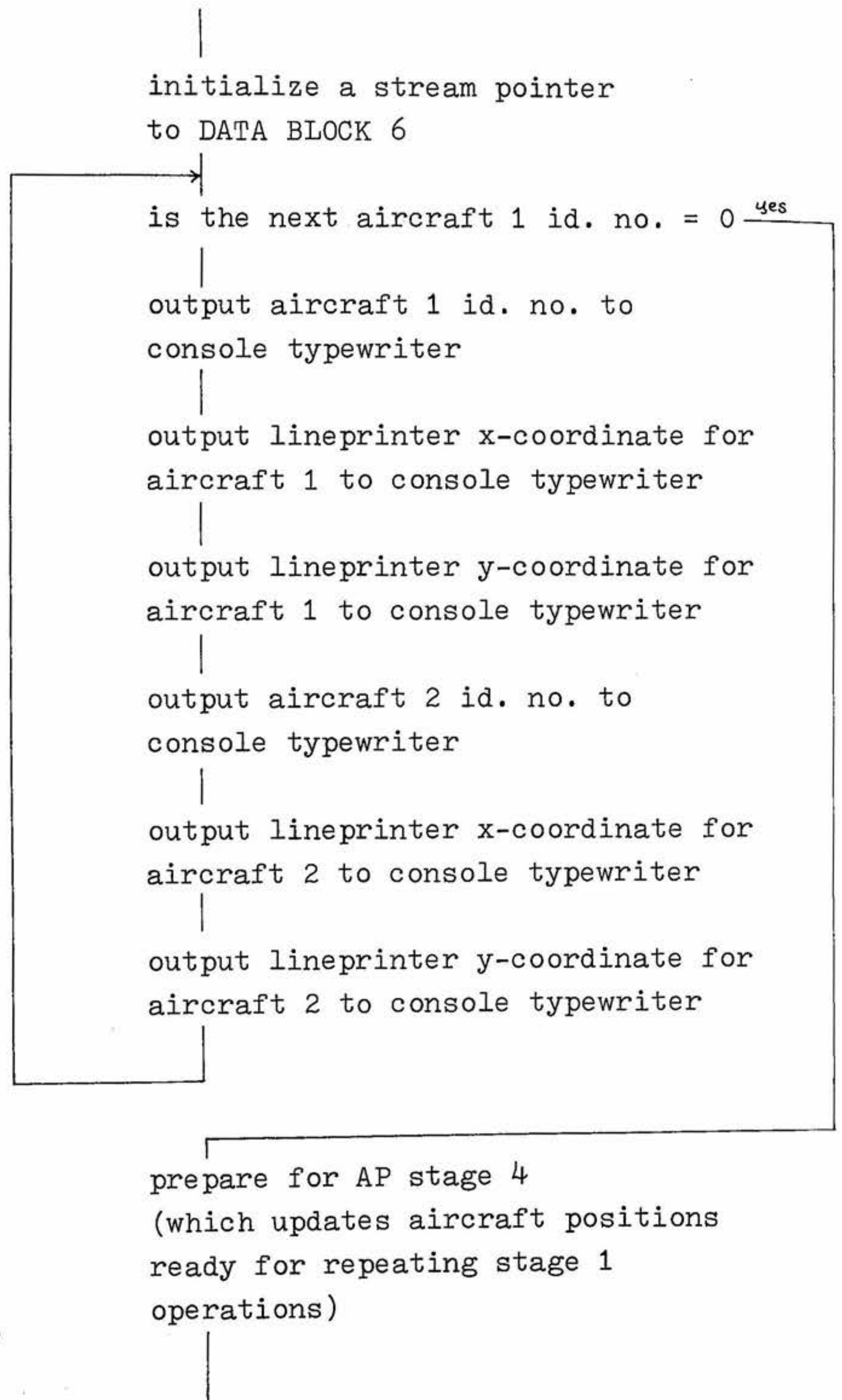
Again, the end of the data stream is determined when a zero aircraft identification number is encountered. The SP continues with the data output from the AP to perform output to the console typewriter. See Fig.4.14 for a description of this part of the SP operation.

4.3.5. Stage 4.

There is a short fourth stage where for the basis of this simulation, the position of each aircraft is changed. This is covered only briefly here. This change is determined by the aircraft velocity. When an aircraft disappears outside the area of simulation, a new aircraft with the same identification number is introduced on the opposite side of the air space. Thus the number of aircraft in the air space remains constant. New aircraft can be entered at any time by appending the appropriate values to the end of DATA BLOCK 1 for Stage 1 of this operation.

The SP during this brief stage is typing out the information on the pairs of aircraft considered to be dangerously close to each other, output being to the console typewriter.

Fig.4.14. SP Program for Stage 3 of Air Traffic Control Application (second part)



4.3.6. The Results.

In Appendix A, a sample copy of the lineprinter output is given. Chart-type coordinates are given although no unit values are shown. Each aircraft is represented by an asterisk. No identification is given on the chart and instead, this must be determined by reference to the table printed out on the console typewriter. This table is also shown in Appendix A and is interpreted as follows. The first number in the line is the first aircraft identification number. This is followed by its x-coordinate and y-coordinate position as shown on the chart from the lineprinter. This is followed by similar values for the second aircraft of the pair considered to be in conflict. All these values are given in hexadecimal. Note that the chart has 64 x 64 positions with markings at every eighth position.

At the end of each pass of the simulation run, a space is shown on the typewriter printout.

The activity of the AP is broken up into self-contained sections with AP programs acting on data blocks provided by the SP, and in turn generating new data blocks to be stored in the SP.

This method of operation allows :

(1) a well controlled interaction between the AP and SP. Although the application is I/O-bound in this particular hardware realization, this would not happen if the number of PEs was much larger, say 1K, and thus all the data blocks could be stored within the AP and accessed in parallel.

(2) a minimisation of the amount of AP to SP interaction, and in particular the modification of the instruction stream. In a large machine, with the AP program stored in a primary memory of the AP, this is still an advantage as the instruction stream remains mainly constant and decisions are more and more the activity of the microinstructions.

The use of a zero aircraft identification number to represent the end of a data stream means that :
(1) the control of the AP from the SP can be kept to a minimum. The AP can do its own tests to determine action quicker than having the SP determine this alone, and
(2) placing counts in the data block is unnecessary and so no preprocessing of the data block is required.

The stand-alone operation of this program makes the assumption that there is no change in the velocity or acceleration of any aircraft. Changes could be made in the program to accommodate acceleration and unpredicted changes in velocity. However, the application as it stood in its simplicity, provided a most helpful basis in determining the instructions to be used.

The instruction set does not attempt to be complete and indeed there is space for many more instructions. Nevertheless, it gives a sample of the many possibilities of this type of microprogramming. Whenever a particular instruction was required, it was possible to find a suitable microroutine to provide it with this microprogrammed control unit. In particular, the major associative instruction SEEK and its use in setting various registers such as the Vertical Register, was dictated primarily by Stage 2 of this

application.

4.4. The Information Retrieval Application.

The second application was the provision of a simple information retrieval query analysis and processing operation. A brief description has already been given in Chapter 2 and here, we expand on that description with respect to the associative processor operations and how they are determined.

4.4.1. General Approach.

The first application concentrated on two types of application techniques :
(1) parallel (arithmetic) operations making use of the number of PEs available, and
(2) associative operations making use of comparison operations.

It was felt that a second application should concentrate more on operations within the microprogrammed control unit and the processing of data at this level. Information retrieval query analysis would require such processing of status information returned from each PE and thus seemed appropriate. The approach that was finally taken was motivated by the research of Ash and Sibley (1968).

Whereas, they considered an associative data structure which was "simulated" by hash encoding techniques, the associative capabilities of the PEs were used here. Their

question-answering system used the basic concept of the associative triple :

$$A(O) = V$$

representing

(Attribute) of (Object) equals (Value)

If we let A, O and V stand for constants and x, y and z stand for variables, then a number of queries can be represented by :

(1) $A(O) = V$

(2) $A(O) = x$

(3) $A(x) = V$

(4) $A(x) = y$

(5) $x(O) = V$

(6) $x(O) = y$

(7) $x(y) = V$

(8) $x(y) = z$

We can solve the above equations for x, y and z. Note that (1) is either true or false, whilst (8) requires that all triples be printed out. The remaining queries require subsets of this set to be printed. Their application in combinations allowed the possibility of sophisticated deductive operations, using the example of a data base of family relationships. In this example, a similar small data base of family relationships was used and only the processing of these simple queries was treated.

An analysis of the problem showed that only one extra operation was required in the AP. This was to AND the result of the condition code test with the Vertical Register and replace the result back in the Vertical Register. Thus,

the Vertical Register was storing intermediate results.

4.4.2. AP Operation.

The SP program basically has to act as an assembler and develop the set of AP operations to process the query. Fig.4.15 shows the activity of the SP in support of this operation. First, the SP program performs a number of initialising operations. Then, we wait until a query has been entered. The next operations set up an array called STRING. The three parameters of an associative triple are placed in this data area. Constants are allocated six characters and are padded with spaces if necessary. Variables are replaced by "*" and the rest of the notation of the query, that is "(", ")", and " ", is stripped away and ignored. Appendix A shows an example of the interaction.

Next, the contents of the array STRING are checked and the instructions to be used by the AP are set up in a special program area for the AP. This program is intended to fill the PEs from the data base, perform the necessary associative operations, output any associative triples that satisfy the tests, and loop back to read the next set of triples into the PEs. Thus the speed of operation is inversely proportional to the size of the data base, and proportional to the number of PEs available.

Any results from the operations of the AP are output with an appropriate format. A simple data base is contained within the program. This consists of family relationships.

Fig.4.15. <u>SP Program for Information Retrieval Application</u>

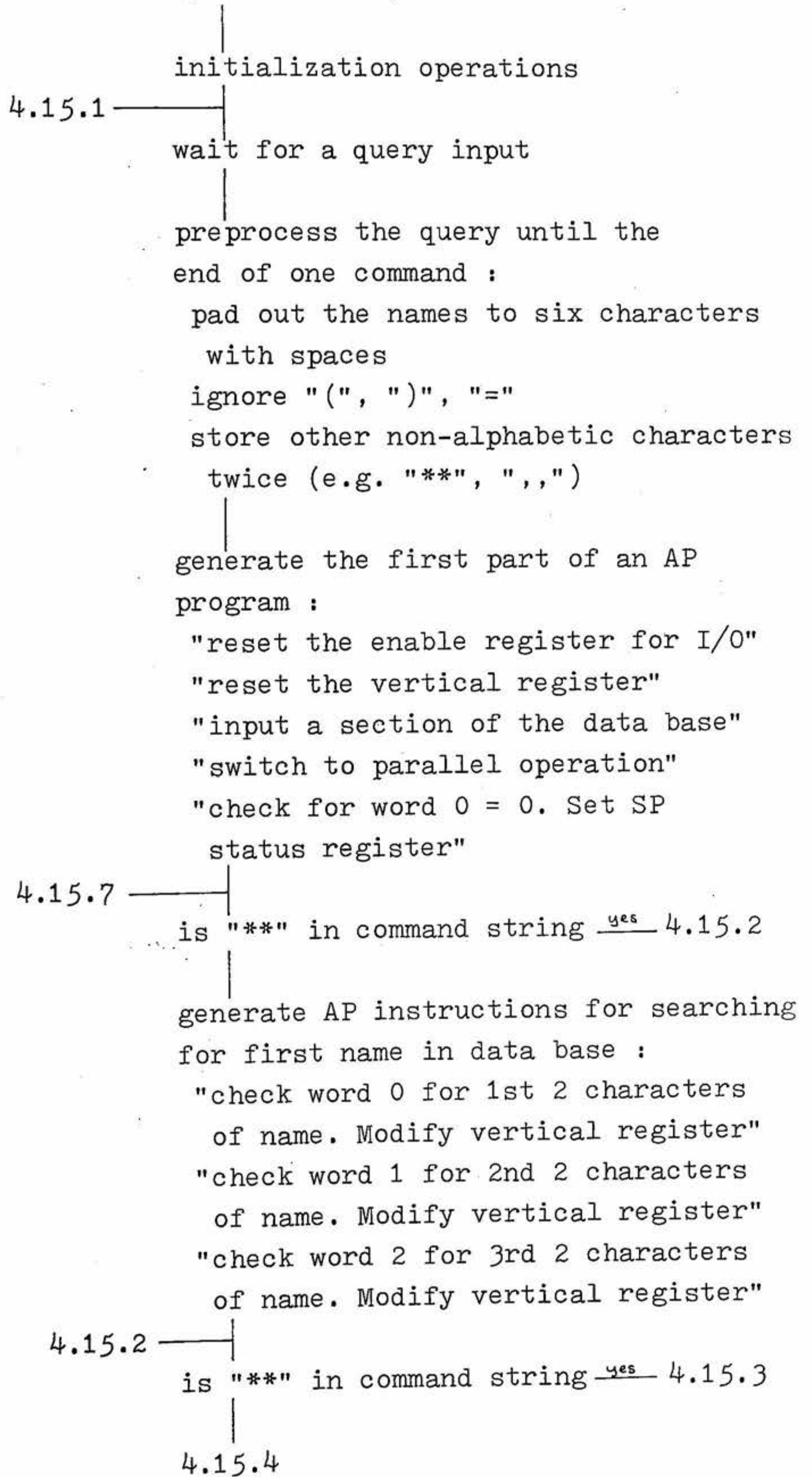


Fig.4.15. (cont.) <u>SP Program for Information Retrieval Application</u>

```
          4.15.4
            |
          generate AP instructions for searching
          for second name in data base :
           "check word 3 for 1st 2 characters
            of name. Modify vertical register"
           "check word 4 for 2nd 2 characters
            of name. Modify vertical register"
           "check word 5 for 3rd 2 characters
            of name. Modify vertical register"
4.15.3 -----|
          is "**" in command string --yes-- 4.15.5
            |
          generate AP instructions for searching
          for third name in data base :
           "check word 6 for 1st 2 characters
            of name. Modify vertical register"
           "check word 7 for 2nd 2 characters
            of name. Modify vertical register"
           "check word 8 for 3rd 2 characters
            of name. Modify vertical register"
4.15.5 -----|
          generate output instructions :
           "reset enable register for I/O"
           "output data base items"
            |
          is ",," in command string --no-- 4.15.6
            |
          generate instructions to repeat
          search operation for second query :
           "switch to parallel operation"
           "reset vertical register"
            |
          4.15.7
```

Fig.4.15. (cont.) SP Program for Information Retrieval Application

4.15.6

generate instructions for continuing
with next section of data base :
"return to beginning of AP program
if SP status register = 0"
"return to normal SP operation"

set up the AP program stream pointer
to the beginning of the generated
program

set the SP output stream pointer
to the beginning of the data base

set the SP input stream pointer
to the answer buffer

commence AP support operation

print out the data in the
answer buffer

4.15.1

Both the object and the value in each associative triple would be the name of a person.

The generated AP program is quite straightforward and its operation can be deduced from the SP program description.

4.4.3. The Results.

A query of the form

*(*) = *

essentially causes the whole of the data base to be printed out. An asterisk is used in place of the variables x, y, or z. Note that a query of the form

sister(jack) = jane

would be printed out again if true or would be replied with only a carriage return/line feed if false.

The main operation available in this application was the ANDing of the results of various comparison tests. The effect of ORing is only available by the SP separating out the "minterms" of the query expression and sending each minterm separately to the AP. The addition of a microorder to perform this ORing within the MCU would be a useful improvement. However, this would require that each minterm could be developed separately and ORed to the value already held in the Vertical Register. Another register would be required and at this late stage of the design process, the availability of transfer register modules and other associated control modules suggested that it should be left out.

The deductive capabilities which were possible in TRAMP (Ash and Sibley, 1968) are best provided by a pre-processor at the SP stage of the operation. Each query however complex can be broken down into a sum of minterms expression which simplifies the activity of the AP and provides fast response.

CHAPTER 5

THE OPERATION OF THE MICROPROGRAMMED CONTROL UNIT

5.1. Introduction.

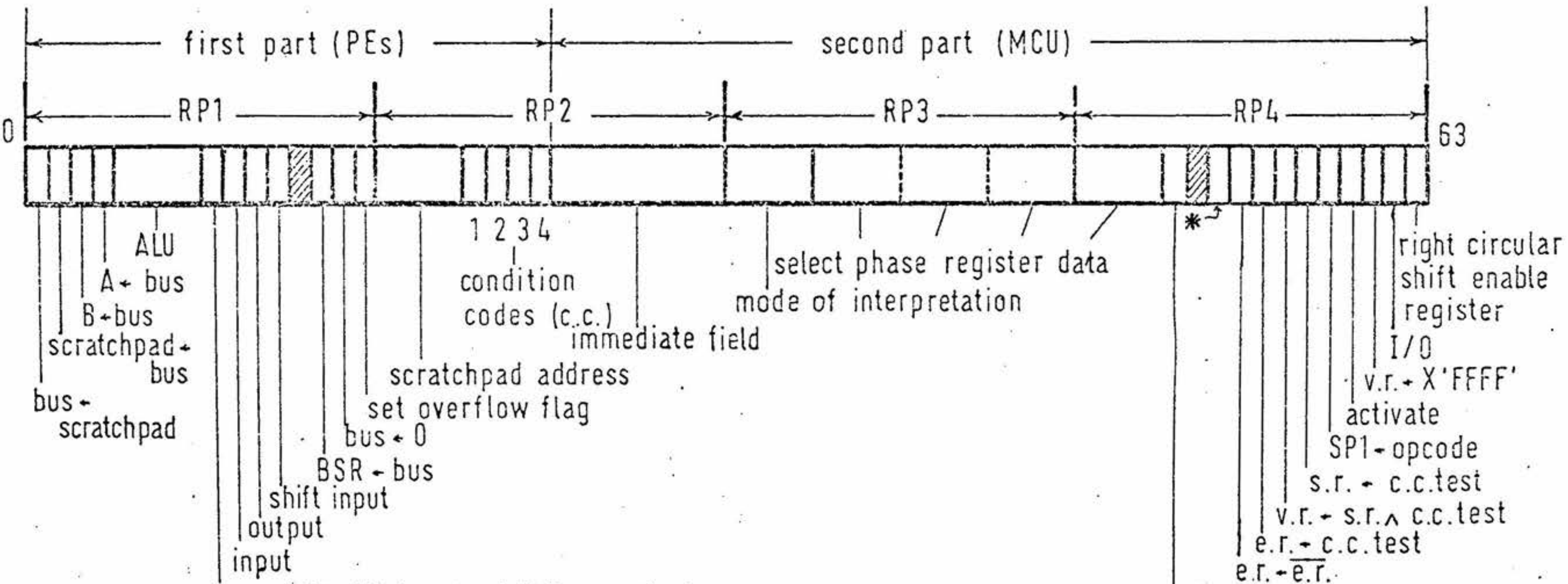
This chapter covers the microprograms written and the operation of the microprogrammed control unit in more detail. An example of associative instruction execution is given to make the operation clearer. Appendix B also refers to another series of operations and describes how they were microprogrammed. Because of the difficulty in describing microprograms, the associative instruction execution is described in another form to make it clearer.

The advantages that were found to be confirmed using this approach to microprogramming will be pointed out. A number of the problems encountered will also be identified. The last section gives some idea of the microcode compaction achieved.

5.2. Example of the Operation of the MCU.

A diagram showing the format of the 64-bit microinstruction is given in Fig.5.1. A summary of the microorders is given in Table 5.1, which shows their bit position and encoding. In the diagram, the numbering of the bits is given from left to right. This convention is also followed in the table although the DEC RTM convention is to number the bits of the scratchpad memories from right to left.

The diagram showing again the structure of the MCU



condition codes :

1 BSR ≠ 0
2 BSR > 0
3 BSR < 0
4 overflow flag

ALU

0 no operation
1 A XOR B
2 A OR B
3 A AND B
4 A - B
5 A + B
6 A - 1
7 NOT B
8 NOT A
9 A
10 B
11 A + 1
12 leftshift A

* = vertical register ← vertical register AND condition code test

Fig.5.1. The Microinstruction Format

Table 5.1. Microorders in the Microinstruction

Read Phase Functional Memory 1 - RP1

Bit No.	Microorder	Comment
0	bus ← scratchpad memory	The scratchpad address is set up by another field in RP2.
1	scratchpad memory ← bus	Same again for the address.
2	B ← bus	
3	A ← bus	
4-7	0 no operation 1 A XOR B 2 A OR B 3 A AND B 4 A - B 5 A + B 6 A - 1 7 NOT B 8 NOT A 9 A 10 B 11 A + 1 12 leftshift A	A and B are the registers of the ALU. The result of this operation is placed on the PE bus. Decoding of this field is done at the MCU. Ideally, this decoding should be done at the PE level. Other mutually exclusive operations could be added here.
8	bus ← rightshift(result of ALU operation)	This only has an effect if there has been an ALU operation executed.
9	input	The data determined by the PE data communication links is ORed onto the bus of the PE.
10	output	The PE output register is loaded. Other PEs and the MCU may receive this data in combination with other data. This is due to the structure of the data communication links.
11	input for shift operations	If set, 0 is shifted in. Otherwise, 1 is shifted in.
12	unused	
13	BSR ← bus	BSR is the Bus Sense Register.
14	bus ← 0	
15	set overflow flag	This flag is set if enabled by this microorder and an overflow condition has occurred.

Table 5.1. (cont.) <u>Microorders in the Microinstruction</u>

Read Phase Functional Memory 2 - RP2

Bit No.	Microorder	Comment
0-3	PE scratchpad memory address	This is decoded by the MCU and becomes effective on a scratchpad memory microoperation.
4	condition code - BSR $\neq$ 0	The choice of condition codes is dictated by the design of the Bus Sense Module. The microorder bit set means that if the condition is true then a 1 bit is ORed onto the bus line associated with that PE, to the MCU.
5	condition code - BSR $>$ 0	
6	condition code - BSR $<$ 0	
7	condition code - overflow flag set	
8-15	immediate field	This 8-bit field is used for loading registers in the MCU.

Read Phase Functional Memory 3 - RP3

Bit No.	Microorder	Comment
0-3	mode of interpretation field	Specifies any special handling of the micro-instruction generation cycle.
4-7	next SP1 address field	These fields become part of the new Select Phase Register data.
8-11	next SP2 address field	
12-15	next SP3 address field	

Read Phase Functional Memory 4 - RP4

Bit No.	Microorder	Comment
0-3	next SP4 address field	This field becomes part of the new Select Phase Register data.
4	enable register ← immediate field	This could be expanded to a field allowing a number of set initial states for the Enable Register. The immediate field could then be eliminated.
5	unused	

(continued on the next page)

Table 5.1. (cont.) Microorders in the Microinstruction.

Read Phase Functional Memory 4 - RP4 (continued)

Bit No.	Microorder	Comment
6	vertical register ← vertical register AND condition code test	Because of the special role of the Vertical Register in AP-SP I/O, this operation allows further selection and combinations of selections of PEs for I/O.
7	enable register ← ¬enable register	This allows a switch from all PEs currently active to all those which were previously deactivated.
8	enable register ← condition code test	The effect is to deactivate those PEs which fail the test.
9	vertical register ← status register AND condition code test	Note that only DEC RTM bits 0 to 7 of the Vertical Register are used.
10	status register ← condition code test	The result of the PE condition code test is placed in the higher byte of the Status Register, with one bit corresponding to each PE's status.
11	load opcode of user instruction into SP1 address field (in SPR)	This is an ORing operation which allows the user instruction to affect the choice of microinstructions throughout microroutine operation.
12	activate	This bit has special significance for one of the modes of interpretation. (See Appendix A.)
13	reset vertical register to all ones	The Vertical Register has a special significance for I/O operations to the SP.
14	I/O operation	If a PE I/O operation is taking place simultaneously, then on output from the PE, output to the SP takes place; otherwise, on input to the PE, input from the SP takes place.
15	right shift circular enable register	The Enable Register is considered to be 8 bits long. The bit shifted out from bit 0 of the register is shifted into bit 7.

and giving some idea of its operation is given in Fig.5.2. Its operations have been explained before and are summarised on the diagram.

It is difficult to consider a simple method for describing the microroutines that were developed. The unorthodox method of generating microinstructions and the fact that on an invocation of a microroutine a variable number of different microinstructions may be generated eliminated a simple mapping of user instructions to sets of microinstructions. This problem is compounded by the fact that microorders in the microprogram are scattered throughout functional memories and one microroutine's microorders may overlap with another's. Nevertheless, one description is given in this section, and another more graphic description is given in the next section.

It was decided not to concentrate on bit patterns in this description but to consider the functional relationships between the various registers. All the possible programming techniques available in this implementation have not been fully exploited. The fact that algorithmic descriptions of the microroutines are not particularly useful implies that many standard approaches to programming may not be the best to use. Descriptions of this type of control must be chosen with care and allow the greatest possible amount of flexibility since if illchosen they can restrict the possibilities of exploring different programming techniques that are in any way different from conventional techniques.

The functional approach is given below. Since the

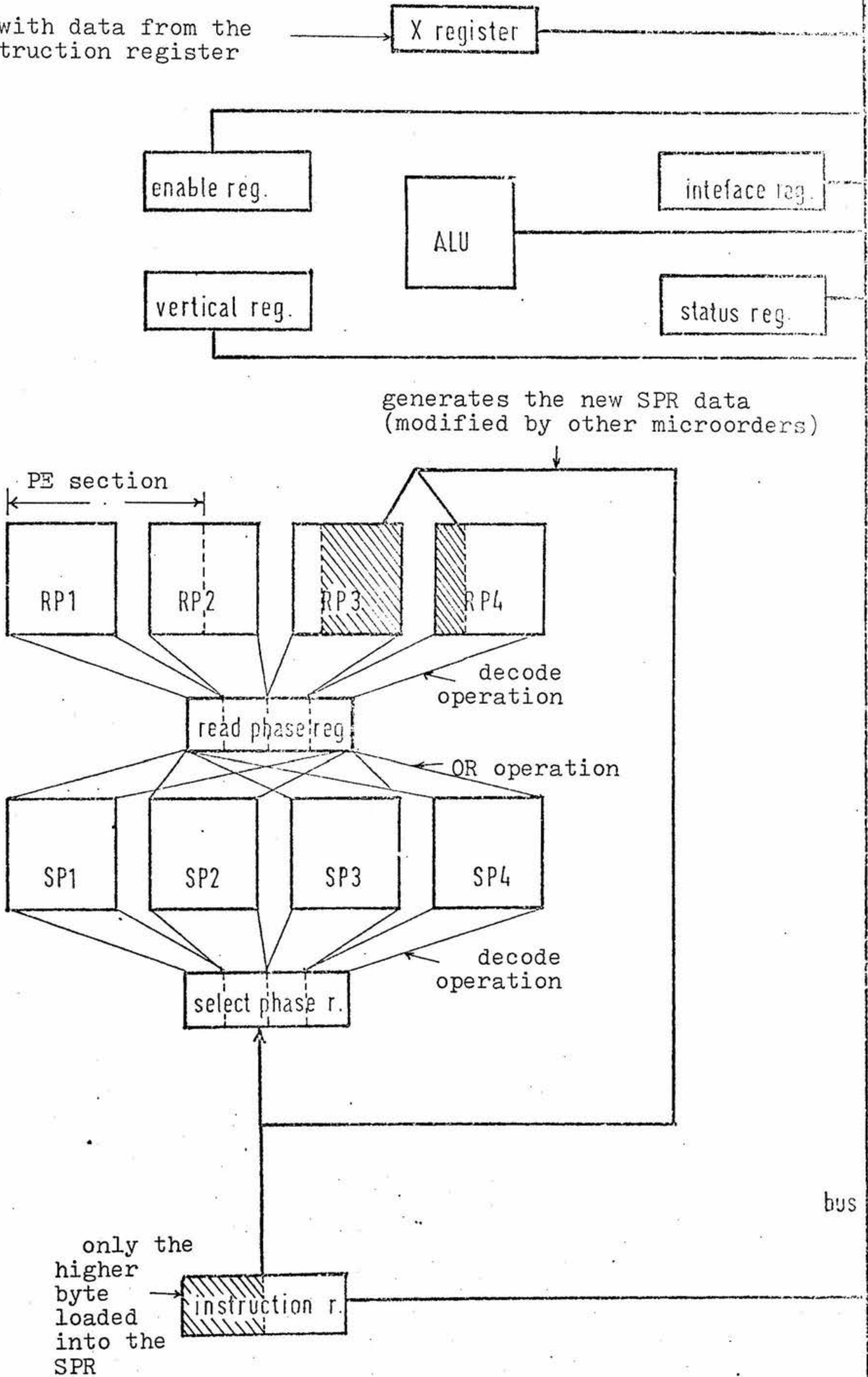


Fig.5.2. The MCU System Structure

select and read phase registers are divided up into four fields during microinstruction generation, they can be described at any instant in time by their contents as follows :

(SPR) = abcd

(RPR) = a'b'c'd'

where a, b, c, d, a', b', c' and d' represent four-bit fields and () represents "the contents of".

There is a mapping :

(SPR) → (RPR)

which takes a value abcd in the SPR to a new value a'b'c'd' in the RPR. That is, the read phase register data is determined by the select phase register data.

Similarly, there is a mapping :

(RPR) → (microinstruction register)

which takes a value in the RPR to a new microinstruction value which can be considered to exist in a microinstruction register, although this term is used loosely to refer to the logic which stores or is influenced by the values generated by the read phase functional memories.

In all descriptions of microprogramming to follow, reference should be made to Fig.5.2 which shows the microinstruction generation operation in a summarised form. Reference should also be made to the microinstruction format in Fig.5.1.

Since a microinstruction bit pattern is not very self-explanatory, the microinstruction will be given in the descriptions to follow as a list of the microorders which are to be activated. For example, a possible list

representing a microinstruction could be :

```
A ← scratchpad(x)
vertical register ← X'FFFF'
(condition code test - BSR ≠ 0)
```

These microorders are considered to be taking place in parallel. Microorders which are initiated but do not have any effect, although generated, are parenthesized. For example, the last microorder in the example above has no real effect since the condition code test results are not used in any way. The parentheses are given as a guide to these types of operations. They represent a form of microcode compaction. Rather than having two separate microinstruction sections stored in a functional memory for different operations, they can be stored in the same microinstruction section if there is no conflict. A condition code test, for example, only becomes significant when other microorders are present in different sections of the microinstruction. When such condition code tests are necessary, the same compacted microinstruction section can be used, thus saving space. Further discussion of such microcode compaction is given in Chapter 7.

The state of the MCU is given by the data in the SPR, the RPR, the MCU ALU registers A and B, and the other PE-type registers such as the Vertical Register and the Enable Register. The instruction register and the X register are also included. The ALU registers are not significant since no microorders ever refer to them or use their values explicitly. They are used implicitly by the MCU microorders. The next state of the MCU is generated by the current microinstruction together with the current state of the MCU. Note

that the mode of interpretation of the microinstruction plays an important part in generating this next state of the MCU.

The approach used to describe all microinstruction-use is based on a table providing information on the SPR data, the RPR data, the microinstruction, the next SPR data and the mode of interpretation. The changes in the other state parameters are given as comments when they are not mentioned in the microinstruction description. Lower-case alphabetic characters represent 4-bit fields and are replaced by hexadecimal digits if their values are known. Groups of microorders are assumed to occur in parallel and verbal descriptions are freely used to convey the meanings of these operations.

We now describe in detail the microprogram for control level 3, which provides the control of the generalized associative operation SEEK, a description of which appears in Appendix A.

The user instruction for a SEEK operation can be represented by

xya3 ; $z_1z_2z_3z_4$

where x = the type of comparison, i.e. which condition codes must be checked

y = the scratchpad location for the comparison

a = the action to be performed with the result of the comparison

3 = the control level for the generalized associative operation

and $z_1z_2z_3z_4$ = the 16-bit comparison data. This is the

external data provided by the SP with which the comparison is to take place.

The first word of this pair of words is loaded into the instruction register of the MCU. The second is transmitted by the SP to the AP MCU, and the microprogram for this operation has the necessary microorders for accepting it from the SP and handling it appropriately. The format of the user instruction has been referred to before, but it is summarised in Fig.5.3, which shows what happens to each section of the instruction in the MCU.

The arithmetic operation which is used to set the Bus Sense Module condition codes for this instruction is :

BSR ← scratchpad(y) - z

The microprogram execution in the general case is described below. The "SPR" column gives the contents of the SPR which generated the current microinstruction. The "RPR" column gives the corresponding RPR data. After the description of the microinstruction, the "next SPR" column gives the contents which will be loaded into the SPR in the next cycle of the microinstruction interpretation operation. The "mode" column gives the number of the mode of interpretation specified by the current microinstruction, which directly influences the generation of the "next SPR" data.

Note that the "a3" section of the user instruction is not used to load the SPR. Field 4 of the user instruction is always used to determine which microprogram to use and field 3 is always loaded into the X register of the MCU, to be used at the discretion of the microroutine in execution.

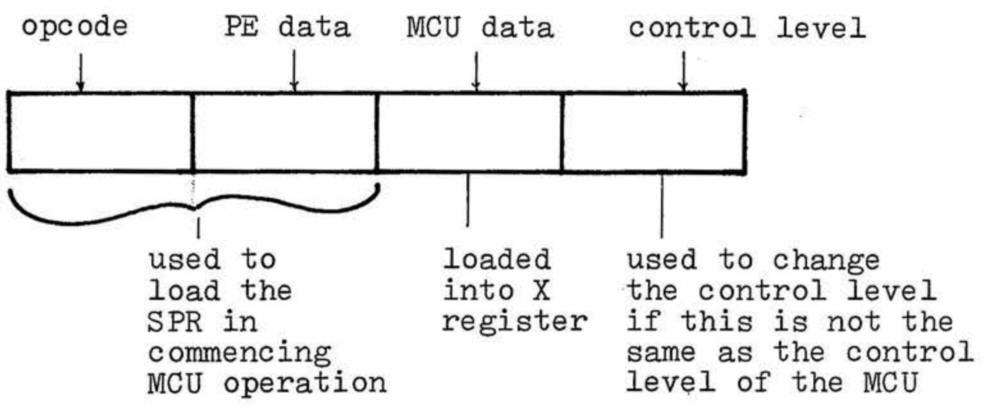


Fig.5.3. The User Instruction Format

Referring to Fig.5.1, Fig.5.2 and the microprogram for control level 3 which is given in Fig.5.4, microprogram execution commences as follows :

SPR	RPR	Microinstruction	next SPR	mode
xy00	0yx0	A ← scratchpad(y) (condition test(y))	0x01	1

(note that the X register has been implicitly set up)

Continuing with the next microinstruction :

SPR	RPR	Microinstruction	next SPR	mode
0x01	1x02	B ← input($z_1z_2z_3z_4$) reset overflow flag (condition test(x)) MCU performs input of $z_1z_2z_3z_4$ load opcode x into SP1 field of SPR	x002	1

Here we see the use of the opcode to affect the loading of the SPR. This is done to prepare x for use in a later microinstruction which will involve shifting the position of x to the appropriate part of the 16-bit word.

SPR	RPR	Microinstruction	next SPR	mode
x002	20x3	BSR ← A - B set overflow flag if necessary (condition test(F)) activate bit set	0x0a	1

Here we see the use of an activate bit (RP4 bit 12) to allow a different set of microinstructions to be generated depending upon a field of the user instruction and thus allowing implied decisions without noticeably affecting the speed of microinstruction generation from the PE point of view. Refer to Appendix A for a description of the use of this activate bit with mode 1.

Further generation of microinstructions depends on the value of "a".

Case 1. a = 3

SPR	RPR	Microinstruction	next SPR	mode
0x03	FxF1	output status information condition test(x) microroutine stop MCU performs output	0000	2

RP1	RP2	RP3	RP4
9000	0F00	1000	1000
2041	1100	1010	0002
0405	2200	1020	2012
0060	3300	1030	0008
0000	4400	1040	0020
0000	5500	1050	0040
0000	6600	1060	0000
0000	7700	1070	0200
0000	8800	1080	0000
0000	9900	1090	0000
0000	AA00	10A0	0000
0000	BB00	10B0	0000
0000	CC00	10C0	0000
0000	DD00	10D0	0000
0000	EE00	10E0	0000
0020	FF00	2000	0000

SP1	SP2	SP3	SP4
0000	0000	0000	0000
0010	0100	0000	1002
0020	0200	0000	2003
0030	0300	0000	F0F1
0040	0400	0000	F0F4
0050	0500	0000	F0F5
0060	0600	0000	F0FF
0070	0700	0000	F0F7
0080	0800	0000	0000
0090	0900	0000	0000
00A0	0A00	0000	0000
00B0	0B00	0000	0000
00C0	0C00	0000	0000
00D0	0D00	0000	0000
00E0	0E00	0000	0000
30F1	0F00	0000	0000

Fig.5.4. Microprogram for Control Level 3

```
SPR   RPR   Microinstruction                   next SPR   mode

Case 2. a = 4

0x04  FxF4  output status information  0000       2
            condition test(x)
            microroutine stop
            status register ← condition code test
(note that the AP Status Register is changed)

Case 3. a = 5

0x05  FxF5  output status information  0000       2
            condition test(x)
            microroutine stop
            vertical register ← status register AND
               condition code test
(note that the Vertical Register has been changed)

Case 4. a = 6

0x06  FxFF  output status information  0000       2
            condition test(x)
            microroutine stop

Case 5. a = 7

0x07  FxF7  output status information  0000       2
            condition test(x)
            microroutine stop
            vertical register ← vertical register AND
               condition code test
(note that the Vertical Register has been changed)
```

When x = 0, the condition test(x) is equivalent to condition test(F), not condition test(0). That is, this is a test for BSR ≠ 0 or overflow. This is because a condition test(0) is not very useful. When x = F, no comparison takes place and instead "z" is returned in the higher byte of the SP Status Register. Thus, this allows a forced loading of the Status Register by the AP. "a" must be 3 for this to work. In this case, microinstruction generation commences as follows :

```
Fy00  3yF1  set PE output from PE input 0000       2
            (condition test(y))
            microroutine stop
            MCU inputs z and outputs to SP status register
```

The MCU I/O microorder has a double meaning in this case, both input and output taking place. The fact that condition test(0) is useless has been exploited by allowing the x = 0 case to use condition test(F) and the x = F case is free for this other purpose. This arrangement was also chosen to allow RP3(F) (the word number given in parenthesis) to be free and provide the "escape" operation X'2000', i.e. the mode 2 microroutine stop.

A number of comments can now be made based on the microprogram for this control level.

This SEEK operation can perform any simple comparison test on a particular scratchpad location of all PEs enabled, using the comparison data supplied by the SP. The status information can be used in a number of different ways. Although the third field is used in a limited way, that is with values a = 3 to a = 7, eight other values are free to have an effect, namely a = 8 to a = F. Thus the third field could for example :

(1) provide additional information on how the data is to be used. Double word comparisons could be made, for instance.

(2) provide additional information for control. Additional associative operations such as allowing masking could be incorporated.

Note that in the a = 6 case, the SEEK instruction semantics are employed in order to preset the value of the MCU-PE output buffer. This was a useful addition because the limitations of the interconnection structure between the MCU and the PEs required an operation to reset the MCU output buffer.

The SP3 functional memory is not used in this microprogram. This is due to the appearance of the SP2 field and the SP3 field (part of the select phase register data) in the same word of the RP3 functional memory microinstruction. Whenever there is this connectivity, exploitation of a field independent of another is very difficult.

SP4 and RP4 are used in a more or less sequential manner. Branching and the conclusion of the microroutine are implicit due to the particular field values of the SPR and RPR. More operations could easily be accommodated here. SP2 is only used for preserving the value of one of the fields (the second 4-bit field) from the SPR to the RPR. SP1 swaps the first field of the SPR to the third field of the RPR. Only the case of X'Fxyz' in the SPR gets special treatment. RP1 and RP2 provide the various microorders required for controlling the PEs. Some overlap of microorders occurs. That is, different microinstructions are generated using the same values of RP1, a demonstration of microcode compaction. RP3 swaps the value of the third field of the RPR to the second field of the next SPR. Only one value gets special treatment (namely 'F') and this cannot be generated by any SPR value since it has a special meaning and requires special processing.

The general associative operation is as short as it can possibly be. The four microinstructions generated are the absolute minimum to perform this operation. In one case, the generation reduces to only one microinstruction and this could be considered as dynamic optimization of the microroutine length. In another case, namely a = 6,

the only reason for the operation is to reset the MCU-PE output buffer with some value, and redundant microinstructions (three in fact) are used.

Note that a certain amount of decision making, which in a conventional microprogrammed control unit would require a number of non-productive conditional branch instructions, are handled implicitly by the normal interpretive procedure of this MCU. In addition, we see in this microprogram the use of microorders which are not actually executed. That is, they are generated but do not become operative except by activation by another microorder. Examples of this are :

(1) the activation of I/O depending on a bit set in the PE part of the microinstruction and a bit set in the MCU part,
(2) the activate bit in the MCU part of the microinstruction, which has special significance in Mode 1 interpretation,
(3) condition code tests which are only significant during certain I/O operations, and
(4) scratchpad addresses which only have significance during scratchpad operations.

The concept of activation seems to be an important one in dealing with microinstruction independence, compression and mutual exclusiveness, and will be discussed again later in Chapter 7.

5.3. Another Description of the Control Level 3 Microprogram.

We give here another description of the control level 3 microprogram, based on the flow-type diagram given

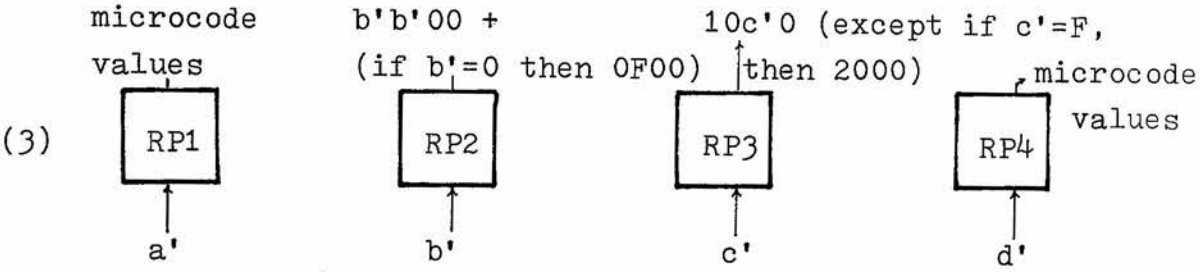
in Fig.5.5.

There are four operations constituting the micro-instruction generation of interest here :

(1) The decode operation for the SP functional memories. A 4-bit value is used to generate a 16-bit value in four cases - SP1 to SP4. This mapping can be simple, as when the 4-bit value is placed in some position of the 16-bit word generated or bears some obvious functional relation to the input value that can be described by a formula. In some cases, a formula can suffice, and in others, this formula must be supplemented by exception conditions. Such exceptions can be described in an ALGOL-like language as is shown in Fig.5.5, for the SP1 functional memory. In other cases, it is too difficult to describe the output of a FM by means of a formula. This is usual for the section of the functional memories that contain either the actual microcode values or represent the vagaries of the sequencing operation. Such is the case for SP4, which in addition has values that "blanket" the values of the other SP functional memories during the ORing operation of the next stage. Such microcode values are better described by referring to the actual microcode or the description in the previous section.

(2) The second operation of interest here is the ORing operation where the results of the various SP functional memories are combined together. Such an operation can be easily expressed as a formula in terms of the original input values with the addition of the microcode values of the SP functional memories which are not expressed as a

(4) new SPR = oc'0d_1' where d_1' is the first 4-bits generated by RP4
mode 1 used except for c' = F when mode 2 used



---

(2) (RPR =) 00a0 + (if a=F then 3001) + 0b00 + microcode values
= 0ab0 + (if a=F then 3001) + microcode values

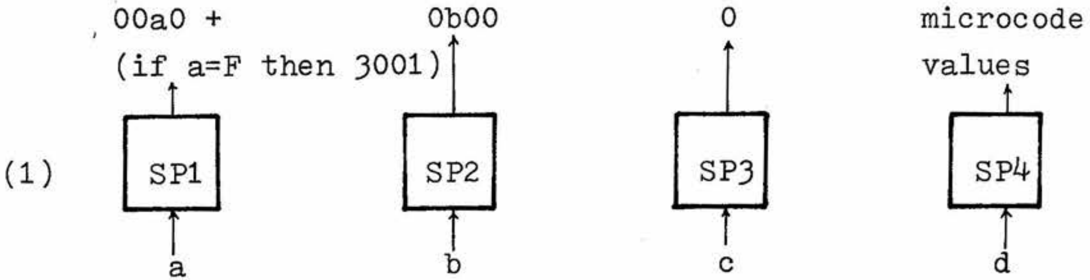


Fig.5.5. Microprogram for Control Level 3

('+' is the OR operation)

formula. Usually, there is only one exception which requires consideration such as in this example. Even in the case where more than one SP functional memory provides microcode values of this sort, they are usually combined in an organised way which makes the value of only one such functional memory significant.

In general, the values of the functional memories do not interact to a significant degree. Thus it is possible to separate out the contributions that each SP functional memory make to the ORed value. However, in this case, a "blanketing" operation is also used, where a "3" in the first field (generated by SP1) can be changed by the value from SP4. Similarly, the value "00a0" generated by SP1 can be subsumed by the value "F0Fz" generated by SP4.

(3) A similar decode operation description can be used for the RP functional memories. Microcode values for the PEs and for the MCU are easier explained by reference to the microprogram and the microinstruction format diagram in Fig.5.1. Nevertheless, some sections of the microinstruction still lend themselves to a formula-type description as shown in Fig.5.5.

(4) Of special interest is the generation of the new data for the SPR. Again, this can be given simply as an expression with only one field (the last 4-bit field) dependent on the special microcode values of RP4. This provides the line of continuity between RP operation and the microcode values of SP4 for SP operation.

5.4. Advantages Confirmed and Problems Encountered.

We treat here observations made in the microprogramming of the other control levels for the MCU. The microprograms are not given in any detail but an attempt is made to point out the advantages which were confirmed and note the problems which were encountered.

5.4.1. Control Level 0.

This control level consists of a number of rather independent elementary house-keeping instructions together with many that are related to only one microinstruction. They provide support for :

RESET	RSTVR	ZAC
LDA	STZ	TEST ER,x
LSHFT	RESTS	TEST SR,x
RSHFT	TRANS	ARSHFT
STA	RECV	

Appendix A contains a description of the above instructions. The microprogram for them is given in Fig.5.6.

SP1 is nearly full. SP2 is fully utilised. SP3 and SP4 are under-utilized. Many instructions generate only one microinstruction, and this leads to the under-utilization. RP1 is three-quarters full. Twenty-one microinstructions are generated by this microprogram and thus this represents a compression of two-thirds of the unoptimized size. RP2 is fully utilized. In addition, without any further conflict, the table of immediate field entries provided for the use

RP1	RP2	RP3	RP4
0000	00FF	0001	1800
9000	1101	0010	2000
1C10	2280	0002	0800
1990	3300	0003	0004
4900	4400	0000	0080
2002	5500	0000	0020
0720	6600	0000	0000
4002	7700	0000	0100
0920	8800	0000	0000
1040	9900	0000	0000
1002	AA00	0000	0000
1980	BB00	0000	0000
0000	CC00	0000	0000
0000	DD00	0000	0000
0000	EE00	0000	0000
0000	FF00	0000	0000

SP1	SP2	SP3	SP4
0000	0000	0000	0000
5F00	0110	0000	6001
10FF	0200	3037	00F2
20FF	0300	0FF0	0000
30FF	0400	0000	0000
40FF	0500	0000	0000
00F3	0600	0000	0000
70FF	0700	0000	0000
00F0	0800	0000	0000
80FF	0900	0000	0000
90FF	0A00	0000	0000
A0FF	0B00	0000	0000
00F4	0C00	0000	0000
00F5	0D00	0000	0000
B224	0E00	0000	0000
0000	0F00	0000	0000

Fig.5.6. Microprogram for Control Level 0

of the RESET instructions can be increased by 12 entries. RP3 is underutilized, again due to the lack of multi-microinstruction generation for user instructions. RP4 is used in some cases for sequential operations. Most of its underutilization, however, is due to microinstruction field compression. Thus, these fields point out an advantage of this approach, namely field compression, whilst pointing out at the same time, a disadvantage, namely the fixed size of the functional memories. This problem will be considered again in Chapter 6.

The microprogram space for this control level is nearly full as far as adding instructions is concerned. Only one more instruction can be accommodated in this microprogram.

Some waste of words is caused by Mode 0 interpretation and it seems better in retrospect to use another type of operation as a stop command, possibly with the use of an "activate" bit.

As was shown in the example of the previous section, some fields have an effect on the system and since the results of this effect are not used in any way, they are considered to be inactive.

The immediate fields represented in the RP2 functional memory form a table of entries. The microprogram shows the limited use of table addressing in the operation of the RESET instruction. Obviously, more use could have been made of this feature. Table addressing could be considered as a special case of the operation of this type of MCU. In fact, RESTS could be given a larger table with upto 16

table entries.

5.4.2. Control Level 1.

This level has already been covered to some extent by the paper given in Appendix B. The instructions supported at this level are :

READ

WRITE

The microprogram for this control level is given in Fig.5.7.

SP1 has enough space to contain another 14 instructions. Those developed will have to be compatible with the READ and WRITE instructions. SP2 has the special function of forming the complement of the x field supplied in the user instruction. (See Appendix A for a description of the user instructions READ and WRITE.) When x = 0, then only one microinstruction is required per PE, and X'20F0' is used in the SPR to stop microroutine interpretation, in preparation for testing the X register. It also has a special use with the WRITE instruction, when x = 0 represents simply a resetting of PE output buffers. SP3 has the special function of incrementing through the scratchpad locations of a PE. It takes the number provided by the last field of RP3 and increments it for addressing RP3 again. Since SP3 is started at word (F-x), it increments until x words have been processed. Its starting at word (F-x) is determined by SP2 as mentioned before. SP4 has the function of addressing the scratchpad memories of PEs as well as providing the same value to RP4. It increments through the scratchpad

RP1	RP2	RP3	RP4
0000	0000	00F0	0013
8020	1000	00F1	1012
4040	2000	00F2	2012
0720	3000	00F3	3012
0000	4000	00F4	4012
0000	5000	00F5	5012
0000	6000	00F6	6012
0000	7000	00F7	7012
0000	8000	00F8	8012
0000	9000	00F9	9012
0000	A000	00FA	A012
0000	B000	00FB	B012
0000	C000	00FC	C012
0000	D000	00FD	D012
0000	E000	00FE	E012
0000	F000	0000	F012

SP1	SP2	SP3	SP4
0000	20F0	0010	0101
2000	00E0	0020	0202
1000	00D0	0030	0303
0000	00C0	0040	0404
0000	00B0	0050	0505
0000	00A0	0060	0606
0000	0090	0070	0707
0000	0080	0080	0808
0000	0070	0090	0909
0000	0060	00A0	0A0A
0000	0050	00B0	0B0B
0000	0040	00C0	0C0C
0000	0030	00D0	0D0D
0000	0020	00E0	0E0E
0000	0010	20F0	0F0F
0000	0000	20FF	0000

Fig.5.7. Microprogram for Control Level 1

locations. Only three words of RP1 have been used, and thus, since there are words free in SP1, more user instructions can be accommodated. RP2 is used solely to provide the address of the scratchpad location to be used and this depends on which word is addressed. RP3 and RP4 preserve in the third and fourth fields of the SPR, the values of the fields used to address them. RP4(0) which is used the first time through the microroutine has the additional function of right shifting circular the Enable Register.

Note that the shortest possible number of microinstructions are used. That is, each microinstruction generated is performing one of the I/O operations required in the execution of the READ or WRITE instruction.

We have here an example of two fields interacting. Specifically, the first field generated by the Select Phase operation can be influenced both by the user opcode and by SP2(0), SP3(E) or SP3(F). SP2(0) is used if the x value in the user instruction is zero. Such a value has no effect on a READ instruction due to the placement of the microorders, but does have an effect on a WRITE, where X'FFFF' is output as the last word from the selected PE before proceeding to the next PE. This has the effect of resetting the output buffer of the PE so that it cannot affect the output operations of the next PE. The reset operation is required because of the particular interconnection structure used.

5.4.3. Control Level 2.

This microprogram provides a number of arithmetic operations on all PEs enabled. These are the instructions that are supported :

ADD	ADDD
SUBT	SUBTD
MULT	

The microprogram is given in Fig.5.8.

Note that x = F is not allowed for ADDD and SUBTD (where the user instruction is X'ax02'). This is because the last word of RP3 has a special significance, namely to stop execution. This is no problem generally but a special microorder (in RP4) could have been used. Use of Mode 2 interpretation is no help in this case. The problem with x = F is that the value x cannot be "remembered" the way it usually is. However, since this value is used immediately by ADD and SUBT, these routines can make use of x = F. Thus, a better means of stopping microroutine execution would alleviate this problem. Note however that MULT never uses x = F since scratchpad(F) has a special significance for intermediate results.

Much use is made in this microprogram of "overriding" where SPn gives one address of an RPm word and another SPn' either modifies it (e.g. even addresses are made odd in MULT) or supersedes it completely (e.g. SPn' sometimes provides an 'F' to generate the last word of RPm). This is a useful technique which could have been further exploited by allowing different SPn to contribute sections to the

RP1	RP2	RP3	RP4
A000	0100	0001	1000
1501	1100	0010	2080
2002	2100	0020	3000
1401	3100	0030	4000
4A91	4100	0040	5000
1591	5100	0050	6000
4A80	6100	0060	B080
1991	7100	0070	8800
1B00	8100	0080	9080
1600	9100	0090	A100
0000	A100	00A0	0800
0000	B100	00B0	7100
0000	C100	00C0	0800
0000	D100	00D0	C080
0000	E100	00E0	D800
0000	F1FF	0000	0000

SP1	SP2	SP3	SP4
0000	0000	0000	0000
0002	0110	0000	10F0
0103	0220	0000	FF0C
8001	0330	0000	30F0
900D	0440	0000	4104
0000	0550	0000	0005
0000	0660	0000	5006
0000	0770	0000	0F07
0000	0880	0000	6F08
0000	0990	0000	4F09
0000	0AA0	0000	2FFA
0000	0BB0	0000	700B
0000	0CC0	0000	FF0E
0000	0DD0	0000	0002
0000	0EE0	0000	0000
0000	0F00	0000	0000

Fig.5.8. Microprogram for Control Level 2

address field of an RPm. Conditional execution depending on various condition codes is a possible application.

All of the level 2 instructions make use of Mode 0 interpretation. This means that field 3 of the user instruction has a somewhat limited use, mainly to give a repeat count for the execution of the microroutine. Some use has been made of this in the air traffic control application program, where a negative number has its sign changed by using X'1x12', which means subtract x from the accummulator twice, where the PE accummulator already contains x from the test of negativity. The best alternative would have been to use another mode of operation so that the contents of the X register are used to determine the destination of an arithmetic operation. This unfortunately was not possible due to the module and wiring limitations on the prototype.

Level 2 is an example of a microprogram where extensive multiple use of RP fields was achieved, both due to microorders being used where they had no effect and microorders being used by several microroutines in several places.

The arithmetic and logic unit B register was used as a work register. In some cases, its initial value was assumed. However, in crucial cases, a RESET instruction is normally used and this operation can perform the resetting of the B register required.

Decisions are made in the case of the MULT, ADDD and SUBTD instructions which lead to the setting of the Enable Register. In the user programming for the air traffic control application, further setting and resetting operations

were required. This prompted the consideration of stacked Enable Register values which could lead to more efficient decision tree-type processing using PEs, and a reduction in the number of such setting operations would ensue. This will be referred to again in Chapter 6.

The microinstructions could be executed in any sequence from the microprogram. The microprogram for control level 2 is on as systematic a basis as it could but is influenced by the sequence in which microroutines were written for it, namely ADD, SUBT, MULT, ADDD and SUBTD. Later microroutines tended to use available space in any order.

Again it should be noted that the microinstruction is assumed to have all microorders executed in parallel, although at a lower level, due partly to the implementation using DEC RTMs, sequential execution is taking place. PE microorders are indeed executed in parallel since their enable lines are all activated at the same time by the MCU. However, note that the MCU microorders, as described in Appendix A, are divided into Phase 1 and Phase 2. The reason for this division is due mostly to the use of DEC RTMs and the fact that slightly more complex microorders are being executed in the MCU. It would be assumed in a customized MCU that its speed of operation was faster than the PEs. In addition, master-slave registers would be used in all interaction between PEs and MCU. With these two conditions, microorders could really be considered to be in parallel. MCU microorders would be at a lower level in a customized logic version of the MCU. The constraint of a single bus structure for all data manipulations at the MCU

level would have been removed.

The execution of the ADDD and SUBTD microroutines leads naturally into the microcode for the ADD and SUBT microroutines. Thus microroutine space is saved. Note that there is a microorder that allows part of the user instruction to continue to influence the generation of the SPR contents. Thus, after executing common code, a branch to more specific microcode could be made. Unfortunately, this was demonstrated only in the case of the associative operations but could have been used in some high-level application-oriented microroutines for new control levels.

In some cases, more microinstruction sections would have reduced the duplication of various sections of a microinstruction in different words of RPn. Obviously, there is a tradeoff between the number of microinstruction sections, in this case four, and the decoding circuitry required.

5.5. Microcode Compaction.

When a microinstruction section, that is, the section of the microinstruction generated by one RP functional memory, is used to form different microinstructions, a form of microcode compaction is taking place. Table 5.2 gives an idea of the number of different microinstructions that can be generated in each control level. This is not the total size of all the microroutines since each microinstruction could be used by different microroutines.

The second column gives the number of microinstructions

	No. of valid microinstructions	No. of microinstructions without control data structure
Control level 0	231	21
Control level 1	8256	6
Control level 2	2417	17
Control level 3	293	8

Table 5.2. The Number of Different Microinstructions

when the control data structure has been accounted for. That is, microinstructions are only counted once if they only differ from the others not counted because of the data representing counters, complementers, storage and the various possible mappings used during interpretation.

Relating this to the number of user instructions supported, some idea of the power in generating appropriate microinstructions can be gained from these figures.

CHAPTER 6

MORE GENERAL APPLICATIONS OF THESE MICROPROGRAMMING TECHNIQUES

6.1. Introduction.

The microprogramming principles that were discussed before in Chapter 3 have been maintained as much as possible. In particular, the separation of fields concept was given a first demonstration in a practical microprogramming environment. The resultant compaction of microcode was clearly demonstrated.

It can be argued that with the falling prices in memory circuits, reduction in the size of the microprogram is no longer such a critical problem. However, a large microprogram represents increased complexity, in both hardware and software. The method used here has an element of dynamic control and optimization of microroutine length which is seldom achieved in conventional design. The small size of the microprogram assists in achieving these advantages.

Many of the advantages of the microprogramming techniques applied to this design can be carried over to other processor designs. Such demonstrated advantages as the following have a more general application :

(1) Horizontal microinstructions can be provided with an efficient encoding of microorders, based primarily on the requirements of each functional module to be controlled.

(2) The parallel operation of multiple resources can be

sustained. Separate sources of control can be catered for, with autonomous control units for each partition of functional modules.

(3) The separation of the control structure and the data structure of the processor can be accomplished. Such operations on the control data structure as the incrementing, decrementing and complementing of data items can be provided implicitly.

(4) The MCU design can be finalised before treating the microorders and the microprograms. An orderly design sequence can be used.

(5) The length of the microprograms can be dynamically minimised during a particular microroutine's operation.

(6) A simplified partitioning of the hardware and software can be promoted even in the structure of the MCU and other control units.

The above are some of the advantages that were expected before the design of the associative processor had been fully developed. It is suggested from the experience of designing the MCU and the AP, that in general such advantages are greatly enhanced by allowing an implicit processing capability in the storage structure that generates the microinstructions.

Functional memory was discovered to be incapable of efficiently storing large amounts of data, in particular, control structure data, which in more normal circumstances would have been stored in residual control and data registers. A memory element (ME) under the control of the MCU could provide a unifying approach to the problem of storing the

major portion of the control structure data. The separation of the control structure from the data processing structure could still be maintained with such an innovation.

This chapter considers the applicability of these results to the design of other multiple resource processor systems. The next section discusses the problems encountered in the original design. A more generalised form of functional memory reducing one of its major drawbacks, namely sparcity, is described. A proposal is then made for a new type of microprogrammed control unit employing this generalised functional memory.

6.2. The Problems in the Original Design of the MCU.

We treat first of all those problems which originated from the physical implementation which was used, and could have been foreseen. After this, we treat those problems which were design dependent.

6.2.1. Physical Implementation Problems.

6.2.1.1. Monophase Operation.

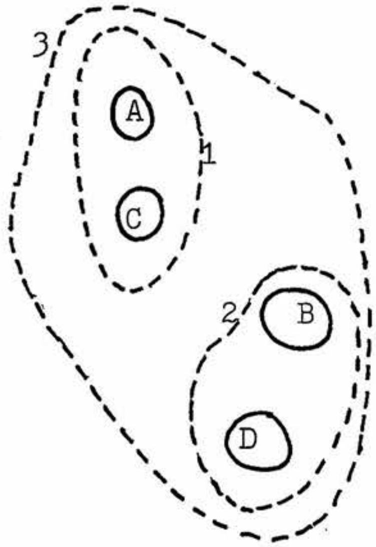
Monophase operation was not available in the realization developed. This was primarily because of the single-bus structure of the MCU implementation. The complexity of the MCU microorders tended to be increased in order to achieve with this single-bus structure all that was required to be done whilst the PEs were being exercised.

The removal of the single-bus restriction would mean that the pre- and post-dependence of microorders in the same microinstruction could be reduced to parallel dependence. More use could be made of activation bits.

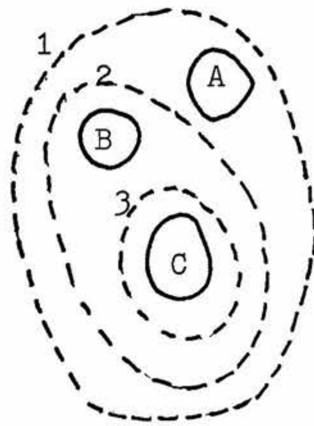
6.2.1.2. Residual Control.

The residual control storage requirements need to be reduced and this would have required circuitry which was unavailable. The final remnants of the data processing structure must be removed and this includes the Enable Register, the Vertical Register and the AP Status Register.

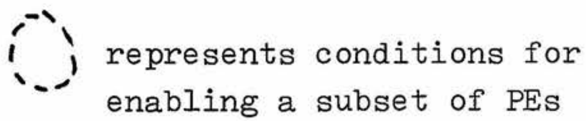
It has already been noted that there were difficulties in controlling flag modules satisfactorily in the PEs. Such operations as shifting the value of the Enable Register would have led to a more complex interconnection structure which had sufficient constraints already. There are advantages in having some stacked flags capability in the PEs. Such a stacking capability of activity bits was used in PEPE. Other operations such as shifting activity bits should also be maintained. Such flags would reduce the control data structure requirements of the MCU and the decision logic for controlling PEs. Tree-search-type activation and combinations of conditions as illustrated in Fig.6.1 would be facilitated. As was shown in the AP program for Stage 3 of the Air Traffic Control application, repeated testing and enabling operations were required to get over the problem of not having such stacked flag operations. Operations may be required on a subset of PEs, with operations then being required for a further subset of the previous subset.



Conditions 1,2,3
Processes A,B,C,D
Ordering of operations
A,C,B,D



Conditions 1,2,3
Processes A,B,C
Ordering of operations
A,B,C



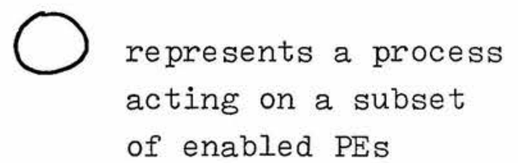


Fig.6.1. Tree Search-type Activation and Condition Combinations

6.2.1.3. Limited Memory Capability.

Dynamically loaded microprograms served a useful purpose in an implementation with very limited memory capabilities. However, this increased the traffic load across the serial link from the SP. It should be possible to reduce or eliminate the requirements for any dynamic loading to very infrequent cases such as diagnostic or special instruction cases.

The use of the control levels to provide some partitioning of the microprogram software was found to be a useful concept to accommodate this problem of limited memory. Such use could be incorporated into the operation of larger functional memories by concatenating the control level field with each separate field of the SPR before any lookup of the SP functional memories. Similarly, the control level field could be concatenated with each separated field of the RPR before looking up the RP functional memories.

6.2.1.4. Branching Delays.

A prediction was made that this approach would lead to a reduction in the branching delays associated with internal MCU control. The main reason that such delay was not apparent was that the time for the generation of the next microinstruction depended mainly on the mode of interpretation and not on whether any type of branch was being taken. The branching delays were in a sense being disguised by the mode of interpretation operation.

In the ideal case, with monophase operation, the next microinstruction generation would take place at the same time as the rest of the microinstruction execution and branching operations would have no effect unless they took longer than the time to execute all of the other microorders in parallel.

With an asynchronous pipelined operation, the microinstruction can be generated whilst the previous microinstruction is still being executed. Each functional module would signal the completion of its operation and the new microinstruction would only start to take effect when all significant completion signals from the various functional modules had been received.

6.2.2. Design Dependent Problems.

6.2.2.1. Microprogram Sparsity.

Sparseness of the microprogram layout is one problem which will be considered more fully in the description of the generalised functional memory. It should be noted that this is a common problem with MCUs providing horizontal microinstructions. For example, Fuller and Mathew (1976) note the sparseness of the microprogram of the PDP11/40. The AN/UYK-17(XB-1)(V) is another example of such sparcity (Rauscher, 1974). Logue et al (1975) demonstrated sparsity in the microprograms for their PLAs.

One of the features of horizontal microprogramming, is that most bit patterns or combinations are not meaningful

(Casaglia, 1976). This is one of the reasons why bit patterns of the functional memories in this design are relatively sparse. The description of generalised functional memory is one attempt to reduce this sparcity by eliminating regular "blank spaces". We need to consider those aspects of microinstruction bit semantics which contribute to low utilization of functional memory space.

6.2.2.2. Microprogram Description.

There was a problem in visualising and describing both switching functions and microroutines in the new environment provided by the functional memories. This problem is not critical since user microprogramming was neither catered for nor encouraged with this approach. Nevertheless, in documentation, it is necessary to give some idea of the action of a microroutine. Such an attempt at documentation to convey this unconventional approach was made in the last chapter. A formal approach may lead to a solidifying of microroutine techniques before all the potential has been realized. This technique towards microprogramming inevitably depends on all the "tricks" and "bit-twiddling" that can be exploited. This is part of the nature of microprogramming where the programmer strives to exploit to the fullest, the advantages of the hardware at his disposal. Unconventional optimization techniques may be used and even these may be difficult to document fully, especially with the detailed timing knowledge required of the microprogrammer.

6.2.2.3. Microroutine Termination.

The stop operation of the microroutine, where the microinstruction signals that the interpretation has ended and that the MCU should proceed with the next user instruction was implemented in two ways :

(1) With mode 0, the RP3 section of the microinstruction had to be zero and the X register had to be zero.

(2) Mode 2 operation always represented termination.

The first type of stop was required because of the special use of the X register to allow an instruction to be repeated with some variation of its components. The explicit stop, however, seemed to provide the more powerful mechanism and could have been used much more.

There is difficulty in choosing the most appropriate way of terminating a microroutine and perhaps a combination of mode of interpretation and activate bit should have been used in each case.

6.3. Generalised Functional Memory.

It is useful to have some measure of the utilization of the functional memories for the various control levels. How do we tell whether a bit is serving a useful purpose in a functional memory? First of all, bits can be grouped together in 4-bit fields, because of the way in which the decoders use the data from the functional memories. Each bit has a significance for a particular 4-bit decoder.

However, these bits can only be accessed 16 bits at a time. If any one of the 4-bit fields of a word is being used, the other fields may not serve any useful purpose, but being part of the word, they are considered to be utilized.

One of the presumed advantages of this technique of microprogramming is that microcode compaction is achieved. Thus, the same word should not be replicated twice in a functional memory. This extends to a word of zero bits. More than one such word represents unutilized bits of the functional memory. Another factor, namely the microinstruction format, can influence the utilization of bits. A microinstruction field that is underutilized generates a corresponding underutilization of the bits in the read phase functional memories dedicated to providing this microinstruction field with data. Some underutilization is nevertheless inevitable with horizontal microprogramming with its minimal encoding of arbitrary microorders.

With this in mind, Table 6.1 provides some figures on utilization of the functional memories. In some cases, a complete FM was not used, as shown in the "FMs in Use" column. The utilization of words can be equated with bit utilization because of the previous discussion. All figures are given in percentages.

It is interesting to note that there are cases where the column utilization was significantly higher than the bit utilization and other cases where the bit utilization was higher than the column utilization. Nevertheless, the word and bit utilization figure seems to be more appropriate

Control Level	FMs in Use	4-bit Columns in Use	Word and Bit Utilization
0	100	84	61
1	100	56	80
2	88	69	73
3	88	59	67

note : numbers are percentages

Table 6.1. Utilization of the Functional Memories

as a statistic.

An additional factor that does not show in such a table is a measure of the logic capability that the functional memories represent. How much logic capability can be provided by a particular size of store? Donath (1974) using a hierarchical structuring of a computer system develops a figure of 8.5 memory cells having the same logic capability as one elementary computer circuit. He mentions empirical results communicated to him by Weinberger, of ratios of 5:1 to 30:1 for the logic capability of memory cells. Rossman (1976) in his review of Donath (1974) mentions a figure of 30:1 having been obtained in the PLA system developed by Logue et al (1975).

This logic capability is not only provided by the functional memories but must be supported by the decoders and the interconnection structure that form an integral part of its use. This logic capability is also related to the complexity of these decoders and the extent of the interconnection structure connecting the parts of the system together.

Taking all these factors into consideration, it is reasonable that if the amount of support logic for the functional memories is to decrease per number of bits, the size of the functional memories must be drastically increased from 16 words per functional memory to perhaps 256 words per functional memory. Such memories are both economical and readily available. They could probably hold all the microprograms required for the majority of applications and reduce considerably the requirement for

dynamic loading.

The results of the MCU design suggest that the design of any generalised functional memory with a larger size should be based on the following factors :

(1) The number of unused words in the functional memories should be minimised. In other words, it should be possible for an arbitrary microroutine to use any of the unused words left over from the other microroutines. In addition, the number of unused columns should be minimised. Here, a column may be a 4-bit field in the same place in each word or any other suitable grouping of bits, usually dictated by the decoding circuitry outside the functional memory.

(2) An asymmetric layout of the microroutine should be possible. Frequently in the microroutines written, there was an asymmetric use of the FMs, in some cases leaving unused a complete FM which was available. Such asymmetric use of FMs should be expected.

(3) In the MCU design, it was sometimes found that a useful combination of fields to make up a new SPR or RPR was difficult to achieve, usually because of the somewhat arbitrary layout of the next SPR fields in the microinstruction. The layout of the microinstruction bears an influence on the generation which should be reduced as much as possible.

6.3.1. A Proposal for a Generalised Functional Memory Design.

Fig.6.2 gives an indication of one way of developing a more generalised functional memory for the select phase

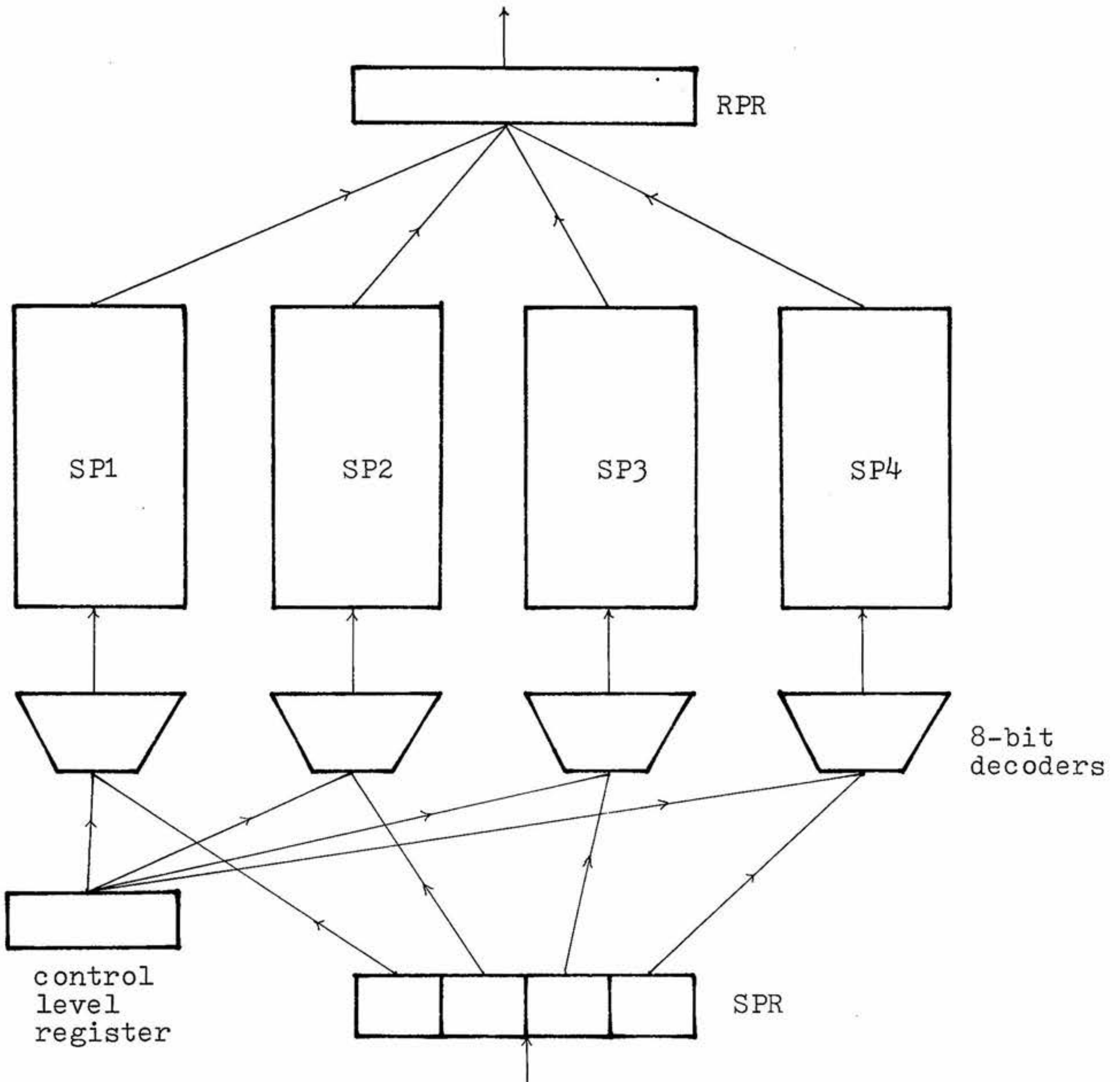


Fig.6.2. Generalised Functional Memory - SP Section

operation. The FMs are 256 x 16 bit memory units, addressed by 8-bit decoders. As before, the output from these FMs is ORed together to give the data for the Read Phase Register. This ORing operation has proved to be a most useful operation. The FMs are addressed by separate 4-bit fields of the data in the Select Phase Register as before. The major addition for this stage is a Control Level Register of 8 bits. The value of this register is ORed to the data from the SPR for each of the 8-bit decoders. (The SPR data is ORed to the least significant 4 bits of the control level.) Thus the additional delay in addressing the FMs is minimal.

The use of such an additional register is as follows :

(1) There is no need to dynamically load microprograms. Each FM can contain the microroutines for its partition of all control level microprograms.

(2) The number of unused words can be minimised across all four Select Phase FMs. Using a control level which is a multiple of 16 gives the same layout of microprograms as before, but with careful use of the values for the SPR, and control levels which are multiples of 4 and 8, data can be arranged in the FMs at 4-word and 8-word boundaries instead of 16-word boundaries as was required originally. Greater utilization of FM space is achieved.

(3) Possible extension of the control level idea, leads to a control level register for each of the FMs. Rather than a common control level register being ORed with data for each 8-bit decoder, each decoder has its own control level register which ORes an 8-bit value to any address

supplied to it. This allows a more asymmetric use of unused words of the memory. Greater code compaction can be achieved at the cost of requiring some means of loading these registers individually. In addition, the asymmetric use of FMs as previously hoped for can now be achieved. Setting a control level register to an "all-ones" value can switch of the use of that FM which then outputs some constant value.

Thus, simple changes in the supporting logic for a set of functional memories can lead to very useful advantages that reduce some of the difficulties encountered in this application to associative processing. We now describe a proposal for the Read Phase section of the generalised functional memories as shown in Fig.6.3.

Again, as for the Select Phase section, a control level register provides greater flexibility in addressing the Read Phase FMs. This can be a single register or a set of four registers, each having an influence over one of the FMs. The same advantages in reduced sparsity apply.

The FMs are again of the larger 256 x 16 bit size, with 8-bit decoders. Again, a 64-bit microinstruction is generated. In addition, there are two interconnection structures, one placed after the Read Phase Register and one after the generation of the microinstruction.

These interconnection structures are controlled by 8-bit interconnection data registers. For each of the four fields making up the RPR, a 2-bit field of this IDR, specifies to which read phase FM this field should apply. It is possible for two or more fields of the RPR to be

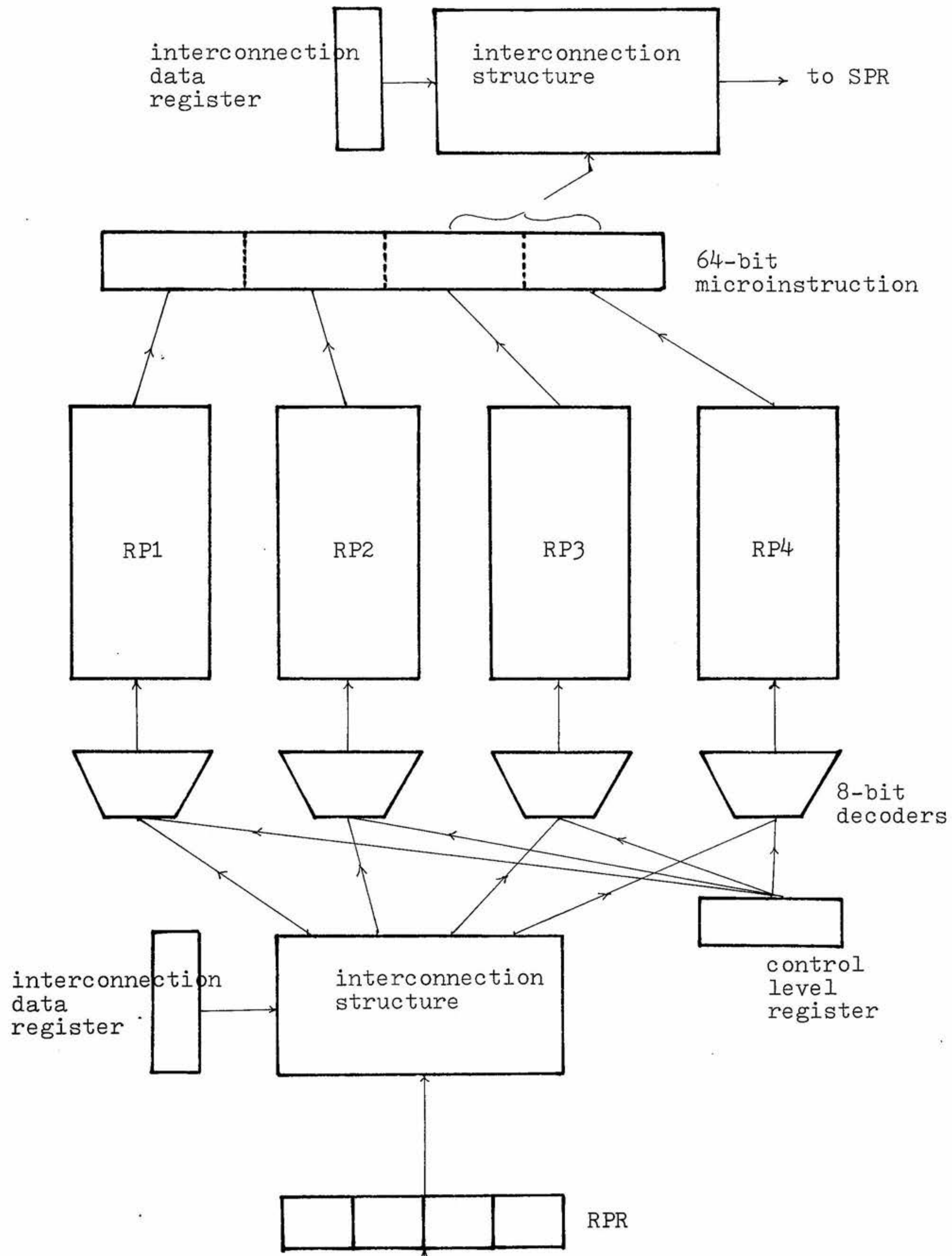


Fig.6.3. Generalised Functional Memory - RP Section

destined for the same FM, and in this case the data in these two fields of the RPR are to be ORed together.

Similarly, the 16-bits of the data which originally would have been loaded into the Select Phase Register first go through an interconnection network which performs a similar mapping based on the bits set in an 8-bit interconnection data register.

The operation of the interconnection structure used after the microinstruction has been generated does not unduly influence the speed of microinstruction generation. This is because these activities are in parallel with the operation, both decoding and execution, of all the other microinstruction fields by the other functional modules of the processor. The interconnection structure after the Read Phase Register does have an effect on this speed of generation in the case where microinstruction generation proceeds only when the previous microinstruction has been completely executed. If asynchronous and parallel operation of the various parts of the MCU takes place which is the process encouraged here, then the effect of an additional stage to the speed of microinstruction generation is minimised.

This interconnection structure does not require any special custom logic. Indeed, it provides another example of the use of simple memory arrays to provide useful logic capabilities. Fig.6.4 shows how four 64-word ROMs can provide this interconnection structure. The 16-bit data word to be permuted is distributed together with bits from the interconnection data register to the input terminals

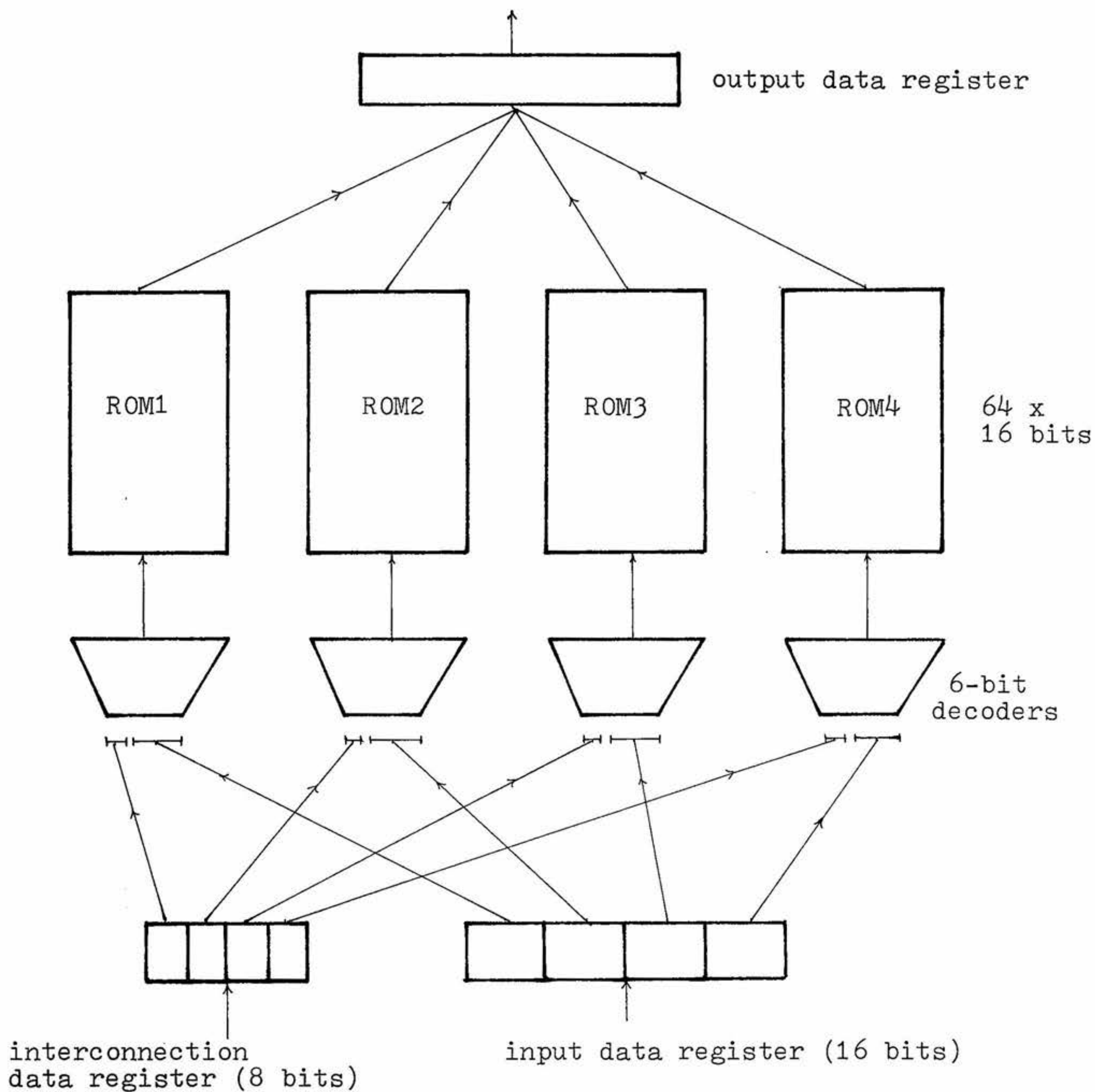


Fig.6.4. Interconnection Structure Provided by ROMs

of these ROMs. The ORed output from these ROMs provides the required rearrangement.

The major advantage of the use of such structures is that greater use of 4-bit columns of the FMs can now be provided. Rather than being fixed as to the arrangement of data in columns, greater flexibility is now available and data can be allocated with consideration given to minimization of empty columns in FMs. There is another advantage with this scheme. The microprograms written never required more than 16 microinstruction to be generated before returning to an earlier section of the interpretation cycle. This was fortuitous, since it would have been difficult to sequence through more than 16 microinstructions. The promise of being able to rearrange columns of a FM on output means that the other columns can also play a part in a sequencing of microinstructions for the same microroutine.

The use of these interconnection structures also extends an important and useful facility of the FMs which was used extensively. The ORing operation took place between FMs and a field in one FM could have a direct influence on that of another FM. The interconnection structure extends this useful operation to fields taken from the same word of an FM. Depending on the value in the interconnection data register, the data in two fields of an input data register, possibly provided by the same FM, can be combined to give a single field in the output data register.

The interconnection structure placed after the

microinstruction generation section of the MCU can assist in two ways. Firstly, microorders are encoded and assigned to fields, which in turn have to be laid out to form the basic format of the microinstruction. The number and size of the fields are related to the amount of decoding required and are influenced by the principle of providing encoded fields most suitable for the independent operation of separate functional modules. Probably, fields for the next SPR are drawn from different Read Phase FMs and may need to be regrouped and rearranged before loading the SPR. Its use may lead to a greater freedom in assigning the other fields making up the microinstruction. Secondly, the principle of separating out the microinstruction into a number of sections is a somewhat arbitrary one, based on an intuitive idea for the breakdown of the microinstruction into subsections. In addition, it remains fixed and it seems impractical to provide the separation into a variable number of variable length sections. The use of such an interconnection structure, or indeed a number of them, may go some way to provide the advantages of a more variable separation of fields and variable field lengths. Such an interconnection structure can be considered as a standard module making up the MCU, and a number of them could be provided for the use of all sections of the microinstruction.

6.4. A Generalised Microprogrammed Control Unit.

Based on the experiences gained from this design, which was initially specifically concerned with the micro-

programmed control of associative processors, it would be interesting to consider a more generalised microprogrammed control unit. That is, the MCU would be suitable for controlling a more general set of PEs, a set of PEs for which the interconnection structure was less restrictive. The microinstruction generated would have fields for controlling groups of these PEs, and fields would also be allocated for those substructures of the interconnection structure which required active control.

It has been pointed out that PEs in the design could have been replaced by microprocessors. Borgerson (1976) among others, has pointed out the advantages of using multiprocessor systems of this sort to implement single-instruction stream computers. Fig.6.5 gives an example of a simple configuration of an MCU controlling a set of PEs with a rather loose interconnection structure. Each PE can address any other PE directly. Contention for the use of the PE interconnection structure can be reduced by providing a time-division-multiplex switch as the basis of this interconnection structure. Such a structure can also be duplicated or triplicated to reduce contention even further. The PEs are specifically considered to be of the order of complexity of microprocessors.

Berg and Johnson (1970) have studied the use in the Advanced Avionics Digital Computer System for the U.S. Naval Air Systems Command, of an associative memory to provide a task assignment algorithm based on dynamic priority, for a similar system of PEs. Such an approach seems appropriate for an MCU to perform for any multiple

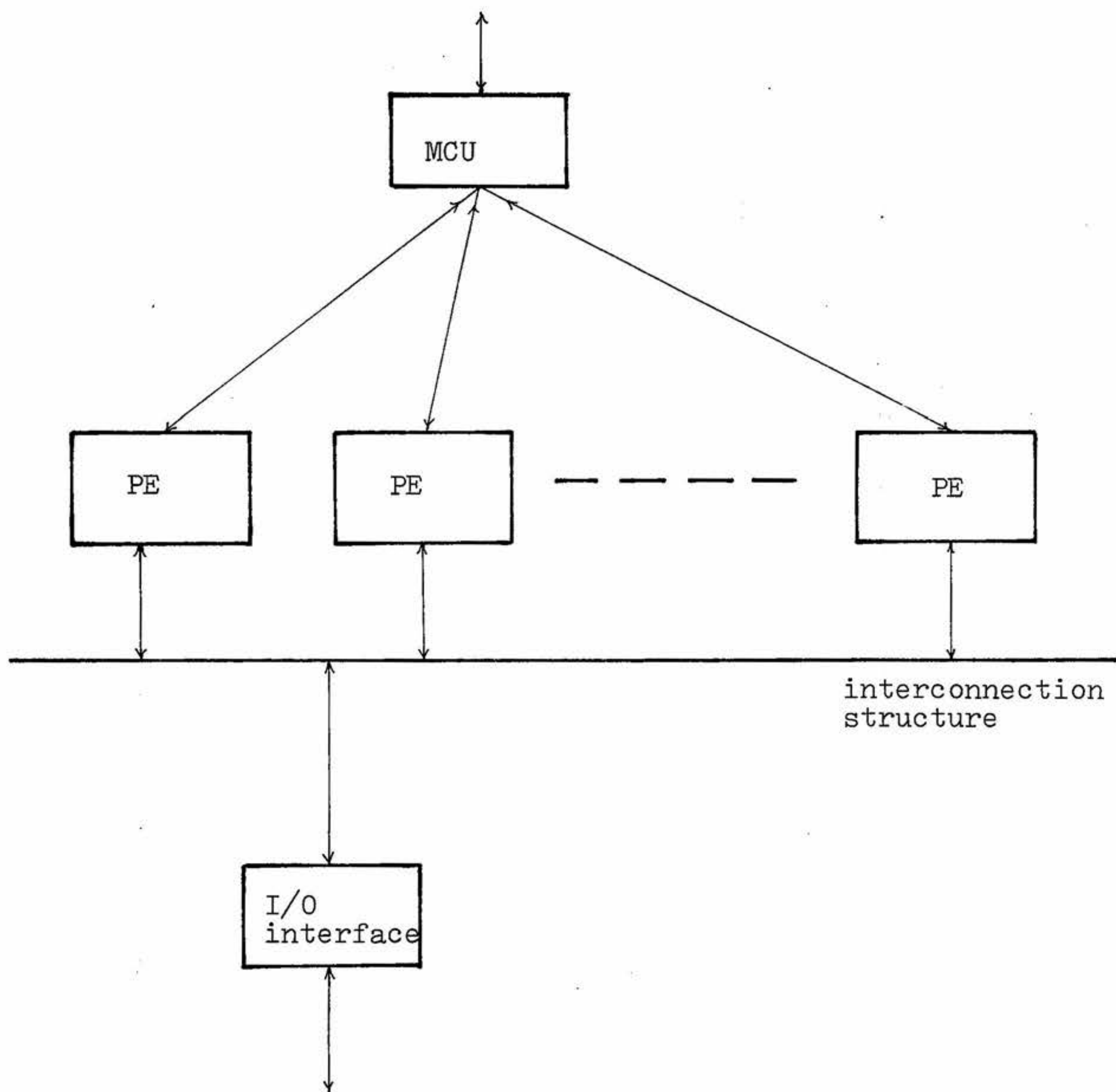


Fig.6.5. A Generalised MCU System Configuration

resource system. It is especially appropriate with MCUs which have a demonstrated design for providing a limited but useful computation capability such as has been proven here.

The generalised MCU is understood to employ the generalised functional memory techniques discussed in the previous section. In addition, a number of additions and modifications to the supporting logic are suggested here that would further enhance the power for controlling multiple resources.

A functional module can be provided within the MCU for the use in storing the control data structure - that is, the information which is specifically related to the control structure. This is basically a memory element since the processing of this data is incorporated in the way that the functional memories work.

Such a functional module can store information for the following features :

(1) microroutine subroutine calls - A stack can be provided for storing data for the SPR, and other residual information for parameters and context. The operation of storing this information on calling a microsubroutine, by using a common SPR data entry, could be initiated by an activate bit in the microinstruction.

(2) mode of interpretation design - Usually, a particular mode of interpretation may use information stored away as part of the control data information. For example, with mode 0 in the design developed for associative processing, information was stored in the X register.

(3) execute microinstruction - In some cases, it may be useful to store a microinstruction in the memory element and transmit it by using an activate bit, in the previous microinstruction.

(4) dynamic scheduling of resources - Similar to Berg and Johnson (1970), it may be possible to store the status of critical functional modules as part of the residual data. Such critical information would contain the status of the various sections of the interconnection network connecting the PEs together. Information on the status of various sections of the MCU could also be contained. This information is useful for deciding which resources to use and when to load diagnostic programs.

(5) structuring of the interconnection network - The data in the memory element can have a direct effect on the interconnections and the activation of various sections of the interconnection structure. This operation would assist the dynamic reorganization of resources.

The microinstruction format for this design could look something like that in Fig.6.6. The fields in this example are as follows :

(1) the microprocessor instruction field - for a parallel processor, one field for controlling PE operation is available for all those PEs which are enabled.

(2) mode of operation field - this has similar operations to the ones used in the original design.

(3) select phase register data - this is provided with a more symmetric layout, i.e. two fields provided by one FM and two fields provided by another.

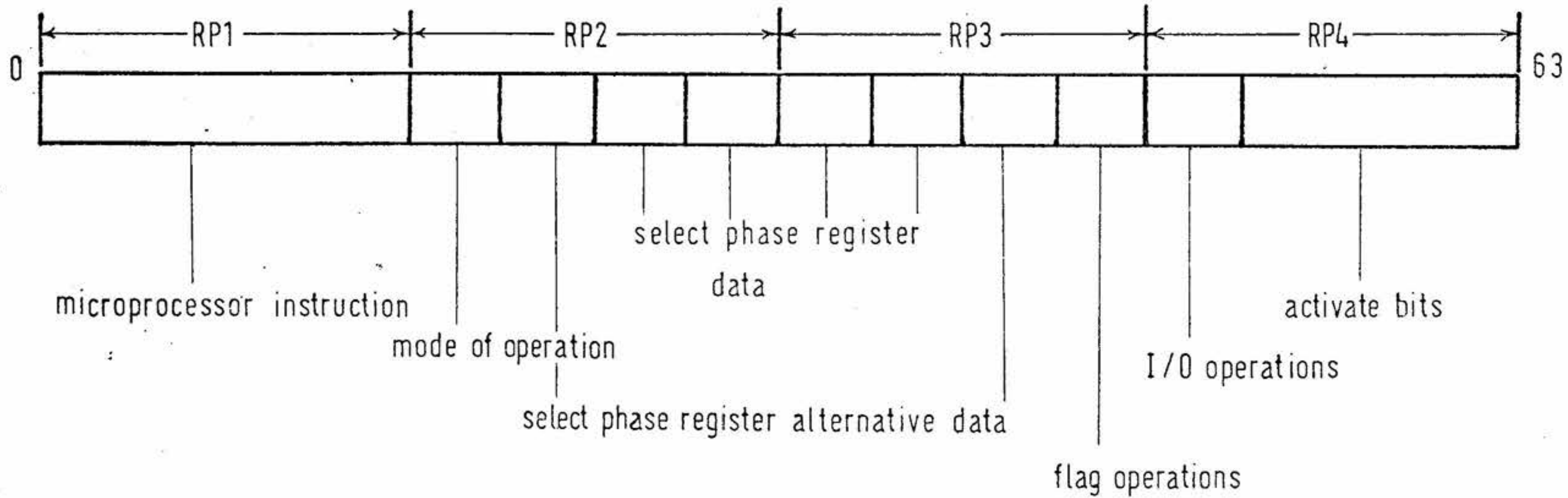


Fig.6.6. Microinstruction Format for Generalised MCU

(4) select phase register alternative data - this is useful for decisions and selecting microroutine subroutines.
(5) flag operations - these are operation on the set of flags provided within all PEs. They are separated out from the other PE microprocessor operations, because in general these operations act on PEs whether they have been enabled or not. If these flags are used, the processor becomes ostensibly an associative processor.
(6) I/O operation field - this is basically concerned with the interconnection structure and those aspects of its control which are not implicit in either the control of the PEs or that of the other sections of the MCU.
(7) activate bit field - this field is concerned with the control of the other sections of the MCU. Among the operations which are controlled are the following :

(a) the interpretation cycle - a "stop" type command specifies that interpretation of the next user instruction is to commence.

(b) testing operations based on data in the ME (memory element associated with the MCU). No special test field as was in the original design is used here, as all relevant status that could be used in decisions is available in this ME. This also provides the capability for dynamic resource allocation within the MCU.

(c) control of the use of the interconnection data registers.

(d) control of the control level registers - such control reduces the main interpretation cycle's

operation to the basic minimum and simplifies the MCU design even further. It would no longer be necessary to reserve a field in the user instruction for control level information.

With this microinstruction format and the design changes implied here, it should not be necessary to have either an arithmetic-logic unit or a bus structure as part of the MCU. We will consider the influence of microorders on microinstruction formats in the next chapter.

CHAPTER 7

MICROINSTRUCTIONS FOR MULTIPLE RESOURCE SYSTEMS

7.1. Introduction.

The previous chapter has demonstrated the strong influence that the structure of the microinstruction format can have on the design of the microprogrammed control unit. In this chapter on microinstructions for multiple resource systems, a general discussion is given on the interaction and arrangement of microinstruction fields. A more systematic approach to the assignment of microorders to microinstructions when multilevel encoding is involved is attempted. The applicability of these techniques to other types of microprogrammed control units is considered and the chapter ends with a description of how this topic is related to other research problems.

7.2. Microinstruction Formats.

Agerwala (1976) in discussing the optimization of both the control store and the execution time of microprograms, has stated :

> Much of the effort on optimization has been devoted to obtaining the absolute minimum solutions rather than "good engineering reductions." Whether the reduction is being performed with respect to the word dimension, the bit dimension, or the number of states, the optimum solution is obtained by techniques that use exhaustive enumeration. These techniques require a great deal of effort and there is no guarantee that any significant reduction can ultimately be realized. ... It is doubtful that an optimum solution can be justified even if the microcode produced is critical and frequently executed.

Bit dimension reduction is closely related to the design of microinstruction formats. Agerwala (1976) surveys research on partitioning the microorders for a microprogram into sets of nonconflicting microorders, each of which can constitute a microinstruction. An an example of this, Dasgupta and Tartar (1976) are interested in the automatic identification and maximization of microparallelism, i.e. parallelism of resources at the microprogram level. They start with a set of microorders in sequence and determine a partition into microinstructions. Their basic definitions and concepts assume a conventional polyphase MCU with an architecture for the data structure similar to conventional processors. Their objective is to provide output which represents the minimum number of microinstructions to perform a set of microorders which represent a straight-line microprogram. Polyphase operation assumes that microorders require variable units of time to execute. Tsuchiya and Gonzalez (1976), studying operations that occur in unit time, produce an optimization which is near minimum time, using graph models and resource requirement matrices. In treating horizontal microprogram optimization, they are concerned with the concurrency of microoperations requiring some parallelism detection technique, and some method of allocating microprogram resources requiring a resource allocation scheme.

Both of these examples, and most other attempts at optimization of horizontal microprograms suffer from what seems a fundamental defect. After the microprogram and microorders are determined, the microorders are set

out and partitioned. The microinstruction format is then determined and only then can the microprogrammed control unit be designed in detail. (Since a conventional control store implementation is assumed, the design of the microprogrammed control unit is essentially the decoding and sequencing combinational network together with the storage of the control data structure.)

From the engineering viewpoint, it would seem more realistic to determine the control data structure and the data structure for the user's requirements. Then, after the choice of the functional modules and the interconnection networks, the microorders can be determined. Suitably encoding these microorders to give the microinstruction format is the next stage, and the approach suggested here is to determine this encoding more than anything else on the control requirements for these functional modules. Especially with the use of LSI components, these requirements may already be in a highly encoded form.

The alternative design approach has serious drawbacks when it is recalled that one of the reasons for using microprogrammed control is the degree of flexibility in changing the control operations either to enhance or correct the original design. A microinstruction design which has been minimised for one set of microorders (probably representing a straight-line microprogram), may be quite unsuitable for a different microprogram written at a later stage of the design process when the microprogrammed control unit has already been built.

Agerwala (1976) has referred to this engineering

aspect in mentioning that the cost of various common circuitry such as ROMs require the use of standard modules of 8 or 16 bits. With this in mind, it is not very practical to use a long and tedious enumeration procedure to reduce a microinstruction of 15 bits to one of 9 bits, if 8-bit ROMs mean that 2 ROMs have to be used in either case. Indeed, 16 bits would probably allow greater flexibility in control and are the cost-effective choice.

Fuller and Mathew (1976) have mentioned changing the format of the microinstruction to allow a reduction in microinstruction space. Such a large number of possibilities exist that only more heuristic grounds can be used for minimization. Intuition plays an important part in the choice of microorders, microinstruction format, microprograms and user instructions. Intuition considers such qualities as flexibility, i.e. what happens when a user instruction must be added, a microprogram changed, a microorder added, or a functional module added as a special enhancement.

Agerwala (1976) in surveying heuristic reduction of microprograms treats phrase structured languages, microinstruction structured languages and points out the importance of residual control. Patterson (1976) using a machine-dependent high level language for microprogramming found that intuitive approaches such as manual optimization provided significant advantages over automatic optimization.

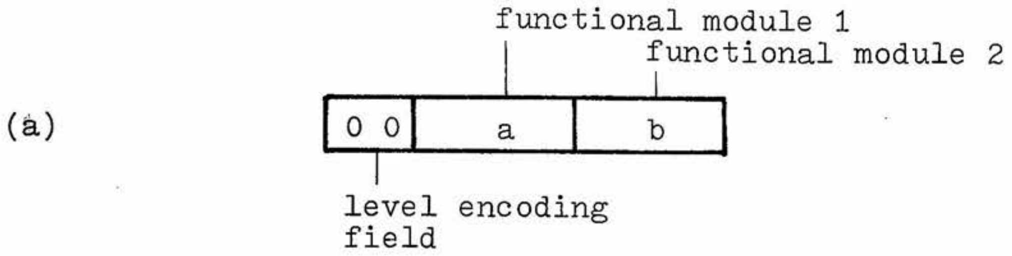
Intuition in the choice of microinstruction fields and the encoding of microorders seems a reasonable approach leading to understandable control systems. The structure of the functional modules to be controlled play an important

part in determining the encoding of microinstruction fields. We can now consider the other factors that influence the design of the microinstruction format.

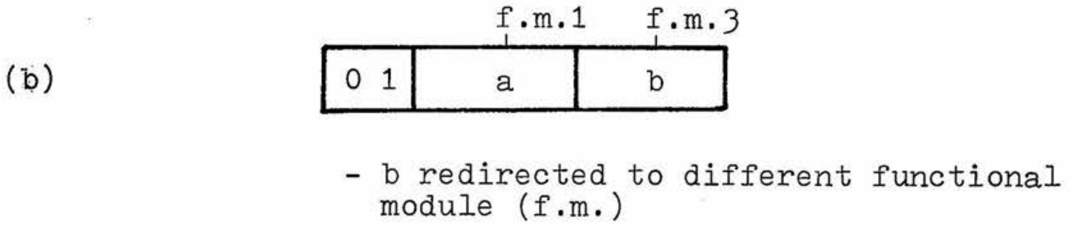
7.2.1. Multi-Level Encoding.

The mutual exclusion of microorders, that is, the fact that two or more microorders could not take place at the same time, was one of the first reasons for encoding a microinstruction field. Similarly, the operation of two or more functional modules may be prohibited from occurring in parallel. For example, two adders may be allowed to work in parallel but a multiplier which uses some of the resources within these adders may be restricted to operating only when no addition operations are in progress. Such mutual exclusion considerations for functional modules suggests a further encoding of the microinstruction fields representing these operations. Rather than one-level encoding, we have what is called multi-level encoding of the microinstruction, and examples of this are shown in Fig.7.1. Special fields have to be provided to specify which of a number of different interpretations the other fields may have, as shown in Fig.7.1(a) and (b). In addition, a different microinstruction format altogether can be specified by this field as shown in Fig.7.1(c). Further levels of encoding can be employed as shown in Fig.7.1(d), where the first field specifies a format that includes other level encoding fields. An example of a horizontal microprogrammed processor with considerable diversity in

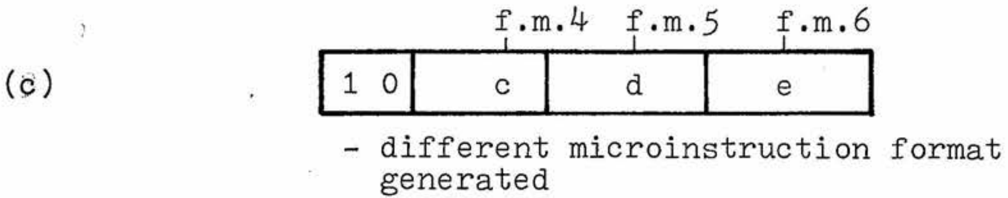
(a)



(b)



(c)



(d)

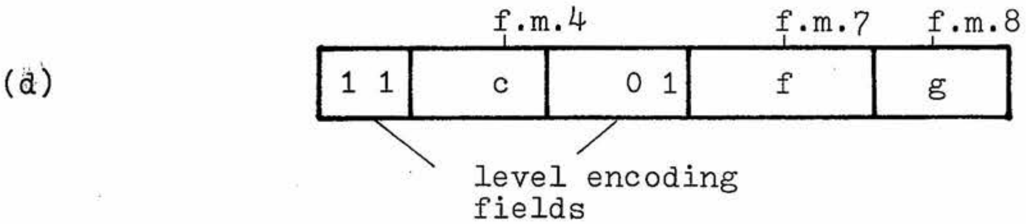


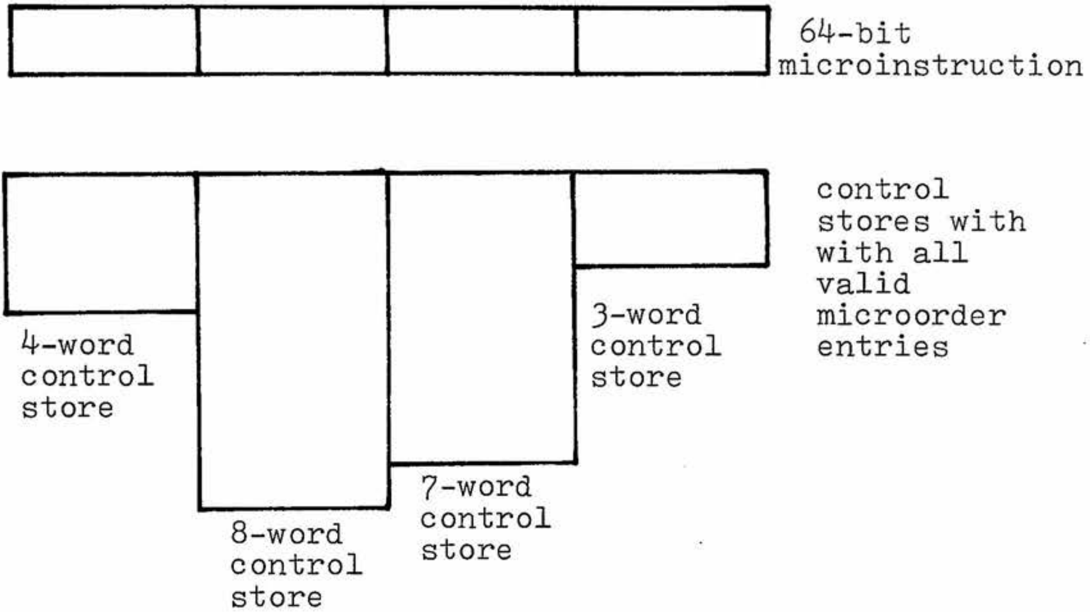
Fig.7.1. <u>Multi-Level Encoding - Examples</u>

the format of the microinstruction is the Varian 73 with its 64-bit microinstruction. (See Agrawala and Rauscher, 1976, pp.239-241.)

7.2.2. Separation of Microinstruction Sections.

Although there is no longer such a critical need to minimize the size of the control store due to the plummeting costs of ROMs and RAMs and the larger size of this circuitry, the large software costs of developing and maintaining microprograms suggests attempts to minimize the size of the microprograms to a manageable level. In addition, the microinstruction format should be kept as simple as possible in order to reduce costly customized control logic. In order to minimize the amount of microprogram space, certain techniques can be followed in the design of the microinstruction format. One such techinque which was followed closely in this work is the separation of microinstruction sections.

The microinstruction can be split up into sections and different control stores can provide each microinstruction section. Such an approach leads to possibly substantial savings in microprogram space because a microinstruction section can be used by a number of different microinstructions. Fig.7.2 shown an example of the potential saving for a group of microinstructions. The division of the microinstruction format into sections and the placement of the microorders and microorder fields within these sections can lead to combinational possibilities approaching the maximum allowed as shown in Fig.7.2. This is most likely if each section



all possible valid microinstructions require 4 x 8 x 7 x 3 words
= 672 x 16 bits
= 10752 bits

all possible valid microinstructions with conventional control store require 672 x 64 bits
= 43008 bits

saving in control store space = 75%

Fig.7.2. Microinstruction Sections - An Example

of the microinstruction references totally independent functional modules or groups of modules. However there are other possibilities and it is suggested that intuitive considerations leading to combinations that are easy to visualise and thus microprogram are most suitable. The next section describes one such possibility.

7.2.3. The Use of Activation Microorders.

The concept of activation was demonstrated in the design of the microinstruction for this associative processor implementation. It represents another way of looking at a number of properties of microorders in a microinstruction. Firstly, multilevel encoding can be represented by a structure diagram showing the level for each field which is specifically used for encoding the other microorders. Activation as a concept suggests that these bits should be looked at individually as activating some MCU operation, namely, in this case, the type of encoding for the other bits. A two-bit encode field is thus one activation bit acting on another activation bit which in turn has a direct effect on the interpretation of other microorder fields. Fig.7.3 shows this concept in a diagrammatic form. The use of activation as a concept may lead to a more orderly approach to the organization of multilevel encoded fields. It may also lead to simpler hardware for decoding these fields.

Activation as a design aim in microinstruction format design is the opposite approach to that of considering

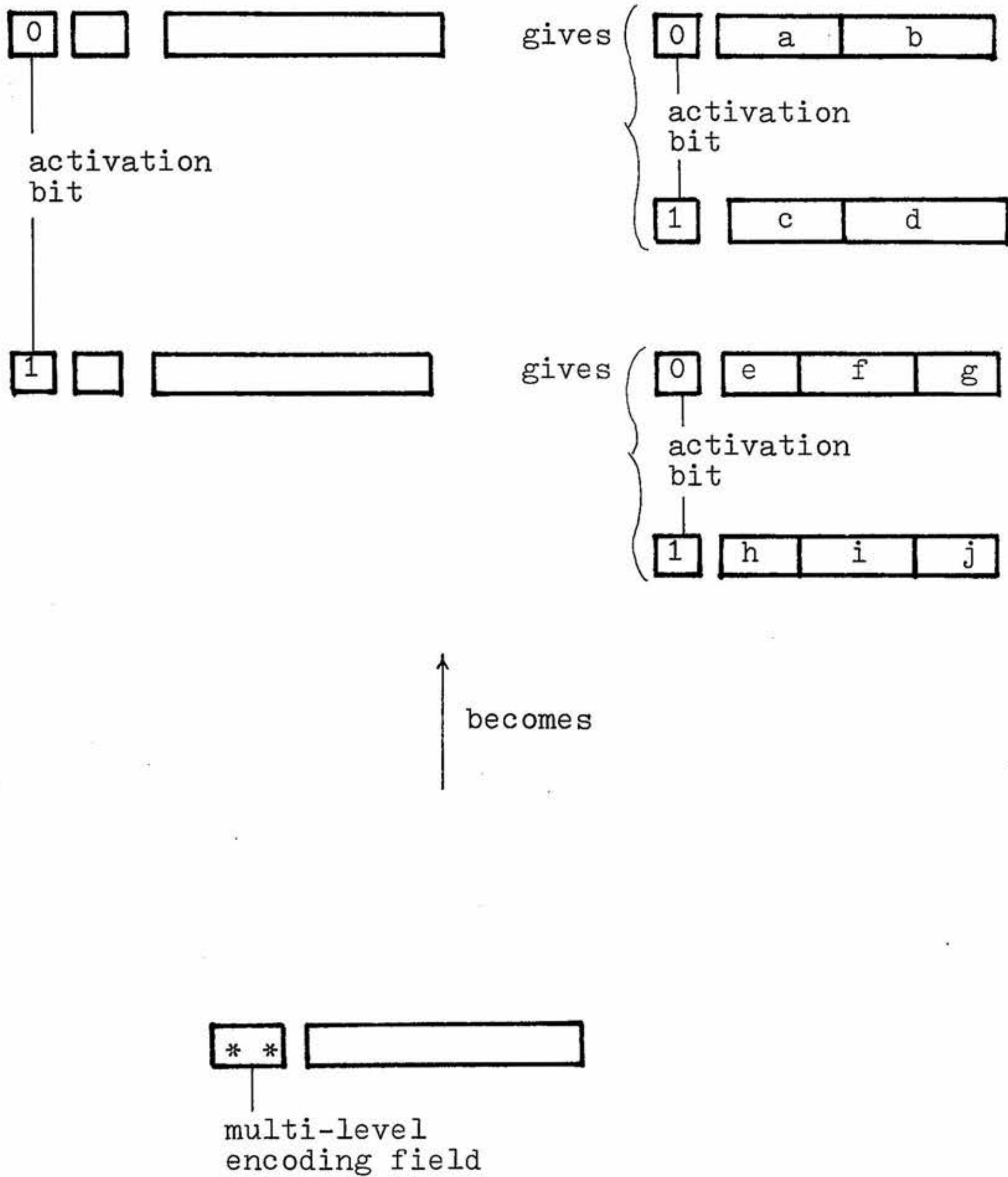


Fig.7.3. Activation Representing Multi-Level Encoding

conflicts and interference between microorders. The design process of studying microprograms and microorders first, performing a partition of microorders into microinstructions, etc., leads to an attempt to reduce the consequences of interference between microorders. In the opposite direction, considering microorders, the microinstruction formats, and then the microprograms, leads to an interest in activation and the creation and exploitation of such microorders which "interfere" with others.

This is not only an intuitively clear idea for control of microoperation but has important consequences in the area of microprogram compaction. Just as the use of separated sections of the microinstruction leads to compaction, where the same field is used by many different microinstructions, we find compaction to follow from the use of activation bits in relation to these separated sections. Fig.7.4 gives examples of this compaction. It only becomes possible with a suitable separation of the microinstruction into sections.

This idea proved useful in the AP design and manifested itself prominently when microorders were generated which had no real effect. Whenever this happened, it represented compaction of the microcode, because other microinstructions could use these sections and exploit these microorder fields.

7.3. The Microinstruction Format as a Focal Point.

The design of the microinstruction format is closely

(a)

- microorder b activates field a (i.e. causes an effect from this field's operation)

(b)

- a different microorder c activates field a

(c)

- microorder d has an effect, modified by microorder c

(d)

- another microorder e has an effect, modified by the same microorder c

Fig.7.4. Activation Microorders Leading to Compaction

linked with the design of the MCU and the whole processor. It can provide the focal point for all of the processor design. For example, Berndt (1970) developed the concept of functional microprogramming for the design of control systems which look like they are microprogrammed controlled. That is, microprogramming as a concept and the development of a functional microprogram can lead to a detailed status specification of the complete processor and is able to assist in the partitioning of resources.

The microinstruction format can assist in the specification of residual control. Foster and Gonter (1971) have proposed a method of allowing compression of the opcode field. After a particular opcode, only a small set of other opcodes are allowed explicitly, although there is an escape mechanism for any of the other opcodes. This leads to a reduction of the size of the field containing the opcode in the user instruction. Thus the present opcode which forms a part of the residual control data is utilized in decoding the next instruction. The control data structure which contains all the information about the current instruction should play a strong part in the design of the microorders and fields in the microinstruction.

Fuller, Lesser, Bell and Kaman (1976) suggest that one of the factors determining the structure of microprogrammed processors is the task of emulation. Microinstruction formats have to be adapted to the primary task of emulation and interpretation. Referring to the trend to more generalized emulation, they treat the interpretation of the IBM7090 emulation on the IBM360/65. This required the use of the

IBM360/65 microcode together with the IBM360 machine code.

This example also points up the difference between actions which are done solely for the sake of interpretation control and information (mapping actions) and those which actually cause the interpreted program to be executed (execution actions).

They also refer to the SAAB FCPU which provided "separate asynchronous processing elements for each type of action". The task of interpretation requires a number of microorders in addition to any for simply specifying the next address. It is suggested here that a simple mapping scheme based on the use of functional memories is suitable. However, the generation of the address for the use in this mapping scheme may need to be subject to modification (modes of interpretation) and the design of the microinstruction format may facilitate this.

7.4. The Assignment of Microorders to Microinstructions.

We suggest here a possible approach to handling the assignment of microorders. First of all, the system to be controlled is considered to be easily represented as a hierarchical structure of functional modules, each of which is possibly controlled by an independent control unit. The structure is recursive in the sense that any necessary design of the microinstructions for each subsidiary functional module is a recursive evaluation of the assignment method suggested here. The microinstructions for the higher levels are determined in part from that of the microinstructions for the lower levels.

Fig.7.5 gives an example of such a structured

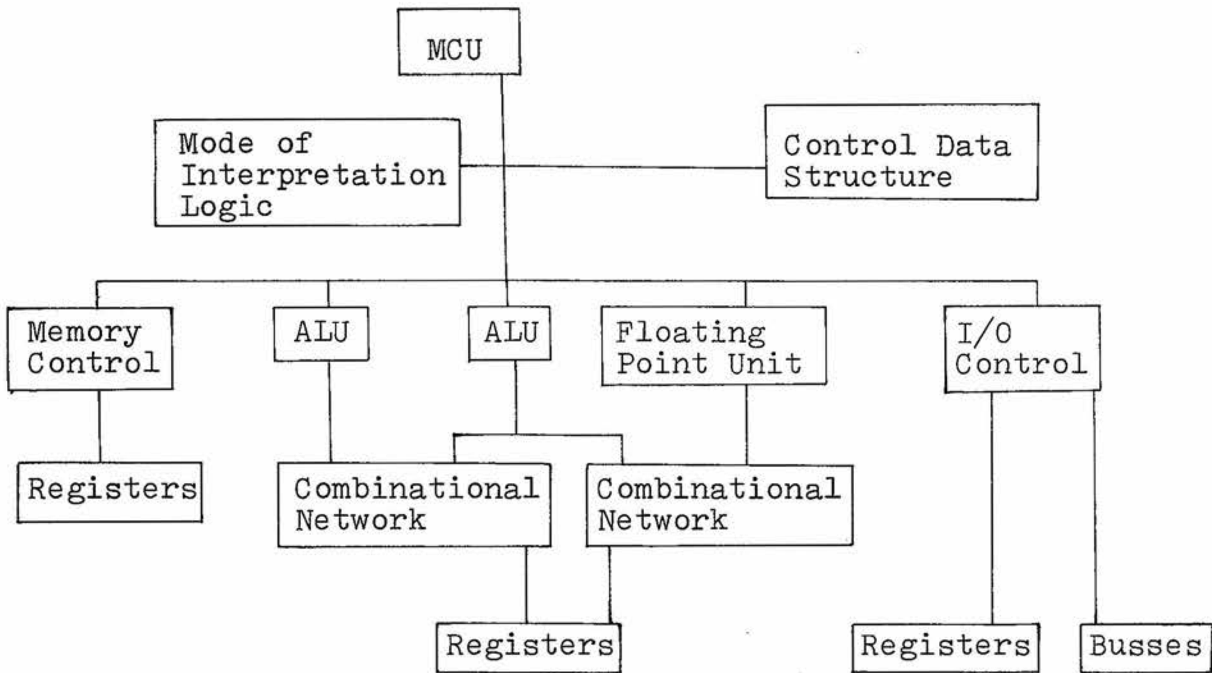


Fig.7.5. Horizontal Microprogrammed Processor Organization

functional module organization. Names of possible units are given here for a processor which demonstrates a fair amount of parallelism but which is nevertheless a more or less conventional machine which provides a non-parallel architecture to the user.

Fig.7.6 is a flow diagram with the suggested approach to the assignment for microorders. This procedure could be used at the highest level and used recursively for the functional modules at the lower levels if these in turn have control units which are microprogrammed. As Fig.7.6 shows, an arbitrary functional module to be controlled is first chosen. It is then determined whether this module is independent from the other functional modules or not. That is, certain microorders for this module may be precluded by the microorders being executed by a different functional module. If this functional module is indeed independent, we check if this functional module has its own control unit, however rudimentary. If not, it is only necessary to set out the microorders for this functional module, encoding these microorders if the operations are mutually exclusive. If this module has a control unit then the whole procedure is started again for this control unit to generate a microinstruction which is incorporated as a field in the microinstruction for the major functional module.

If the module is not independent of the others, in the first case no other activate bits will have been assigned. Thus, we continue by assigning activate bits for this operation. That is, we assign an activate bit which

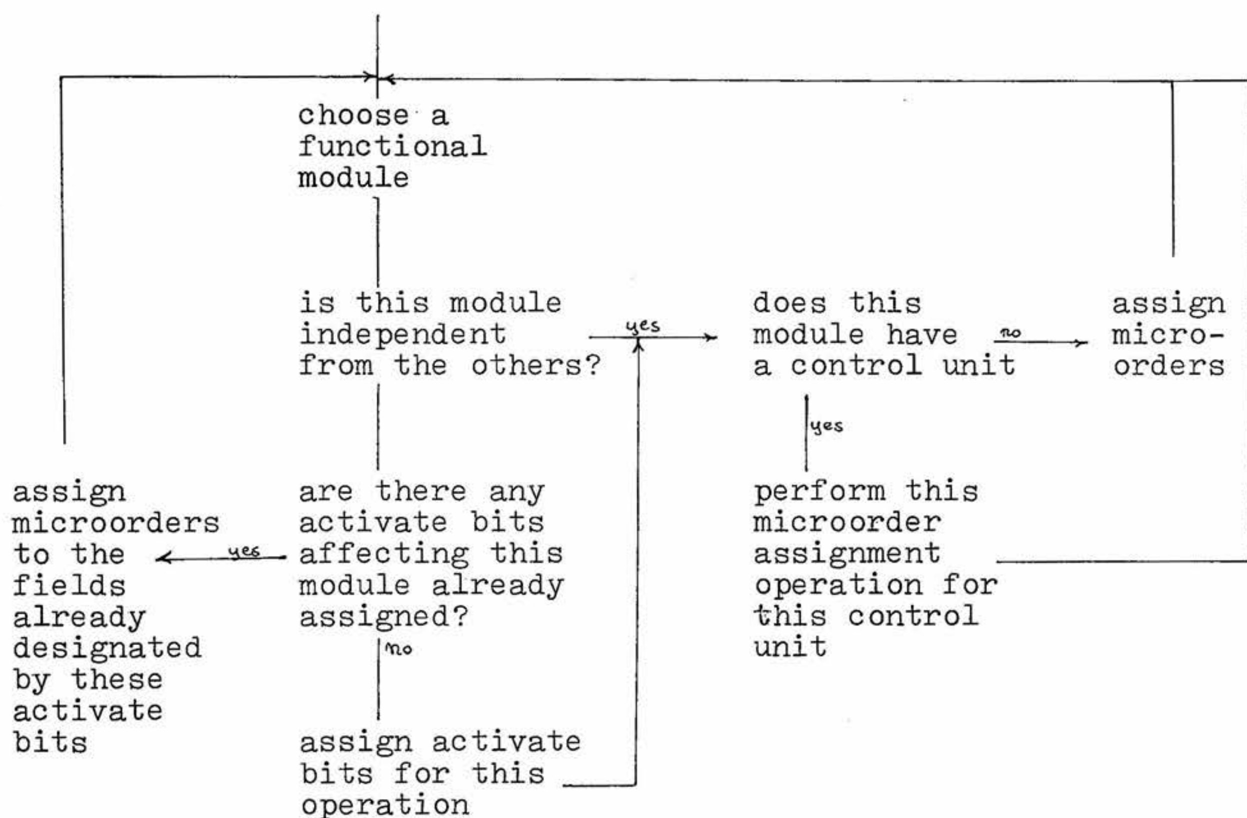


Fig.7.6. Flow Diagram for Microorder Assignment

indicates if set that this functional module is to have control of the rest of the microinstruction field to be assigned. After this activate bit is assigned, we can return and check if this functional module has a control unit and proceed from there as before.

If, on the other hand, activate bits have already been assigned, we consider the case where the activate bit is not set, representing the case when the other functional module is not active. In this case, we should attempt to use the space already assigned for the microorders for the non-active functional modules. In doing so, it may be necessary to increase the field size for this common microinstruction space. On the other hand, the field space may be larger than necessary and in this case, the fields required should be placed in ways that can capitalise on the common decoding and other combinational circuitry.

After having treated this functional module, we continue with the next one until all have been done. Two of the functional modules are the control data space and the modes of interpretation modules. If it is possible to treat these systematically in the same way as all the others, we are left with the requirement for only a simple MCU of the level of complexity which has been studied here. It may be necessary to consider the mapping and interpretative function separate from the execution function and with the demonstrated advantages of separate microinstruction sections, a variation of the functional memory techniques applied in the case of the associative processor may be a practical solution.

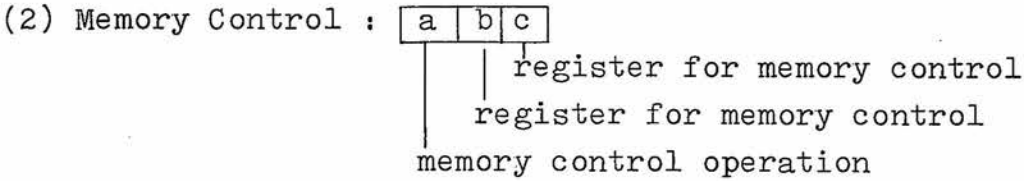
Intuition plays a part in the choice of which functional modules to start with and it is possible that different choices may lead to different microinstruction formats. This method generates a multilevel encoding which corresponds to the hierarchical structure of the system components. The choice of functional modules could be critical when they are not independent. Then, after a recursive generation of the same procedure a further choice is made of a non-independent functional module and the choice can lead to a particular encoding of the activate bits. Recursion leads to a fully encoded set of activate bits whenever there is a dependence between functional modules. Thus, this is one way of handling implicitly the multi-level encoding which is usually a part of horizontal microprogrammed machines.

Fig.7.7 is an example of the use of this procedure leading to a microinstruction format for the hierarchical processor organization shown in Fig.7.5. Its development should be self-explanatory.

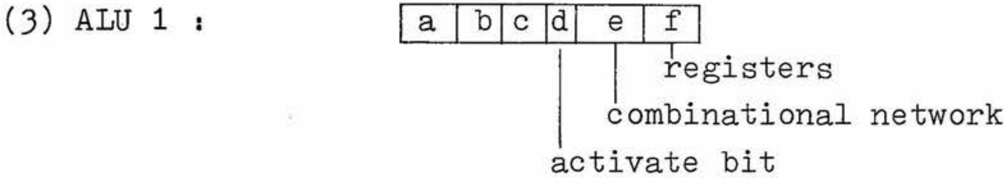
After the microinstruction format has been developed, it is then in the interest of microcode compaction to split it up into sections. Before this splitting-up operation, it should be rearranged to maximise the compaction which will ensue. An example of this operation, based on the micro-instruction format developed in Fig.7.7 is given in Fig.7.8. When there is a large number of encoded activation bits, it may be possible to consider ways of maximising the micro-code compaction by spreading out these activation encoded fields over the largest number of separate sections.

(1) MCU : nothing assigned yet

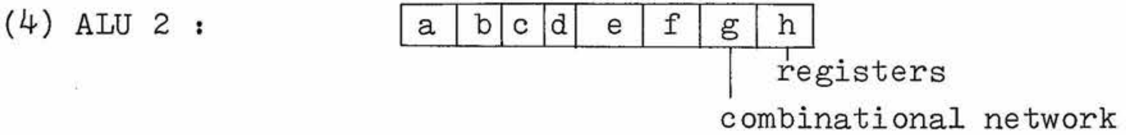
(2) Memory Control :



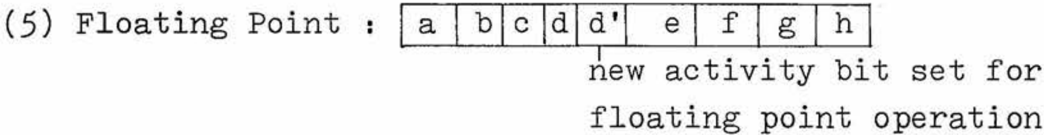
(3) ALU 1 :



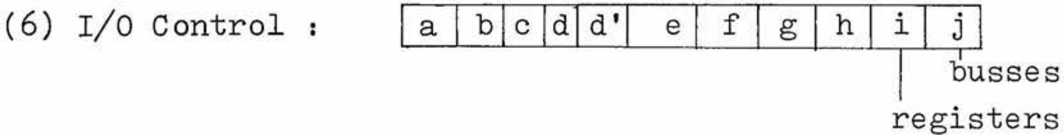
(4) ALU 2 :



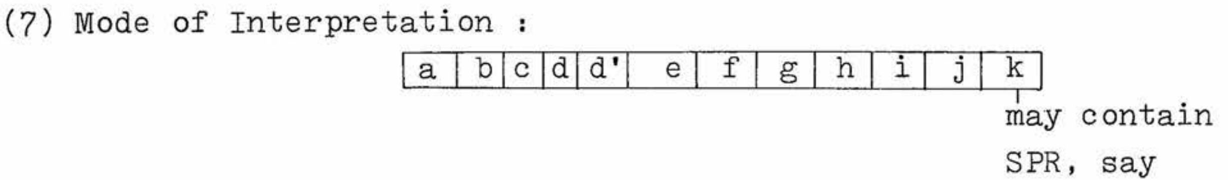
(5) Floating Point :



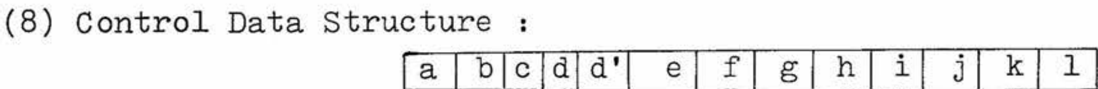
(6) I/O Control :



(7) Mode of Interpretation :



(8) Control Data Structure :



Note implicit multilevel encoding for ALU/floating point units :

10 ALU 1 operation
00 ALU 2 operation
01 floating point unit operation
11 ALU 1 and floating point unit operation

Fig.7.7. Example of Microinstruction Format Generation

(1) determine the section size : say 16-bit sections

(2) 1st section :

a	b	c	e	f

(3) add 2nd section :

a	b	c	e	f		g	h	i	j

(4) add 3rd section :

a	b	c	e	f		g	h	i	j		k	d	d'	l_1

(5) add 4th section :

a	b	c	e	f		g	h	i	j		k	d	d'	l_1		l_2	spares

Fig.7.8. Rearranging the Microinstruction Format

Note that the activation bits d and d' have been separated from the fields that they relate to.

The major objective is always to provide the microinstruction format which is intuitively easy to grasp, as this leads to simpler microprogramming and cheaper development costs.

7.5. The Generality of this Approach to the Microprogrammed Control of Associative Processors.

The initial design of the microprogrammed control unit was based on the need to control an associative processor. It quickly became apparent that this was more than anything else a parallel processor with special types of operations supporting associativity. The PE interconnection structure and the other control registers used with the PEs were also specifically for associative operations. Nevertheless, the techniques developed for controlling the PEs seem applicable to other sorts of parallel processors, whether or not they incorporate PEs which are operating simultaneously on parallel data streams.

The concept of activation and the assignment of microorder fields to separated sections of the microinstruction seems to lead to a valid approach to operations which are normally multiencoded, where there is any degree of parallelism in the data processing or where a computing resource is shared between a number of functional modules. It is particularly suitable for a hierarchically structured configuration.

This method leads to increased compaction of microcode, reduced cost of control storage, and good pin/circuit ratios. The use of similar techniques for the mapping function,

dynamically generating each microinstruction, leads to economies in the facilities required for the MCU. The code in these select phase functional memories does not represent microcode but the mapping operation. It is possible that a smaller number of them may be adequate. Thus at one extreme only one functional memory may be used for generating the microinstruction. Each entry in this functional memory represents one unique microinstruction and each microinstruction to be used in sequence requires an entry in this functional memory. Only very simple microprograms could be accommodated with such a scheme. The mode of interpretation field assumes responsibility for all variations of the next microinstruction address. Every possible microinstruction requires an entry in this mapping or select phase functional memory when it has to be invoked.

By using more than one such functional memory, some of the advantages enjoyed by the execution section of the MCU, namely the code compaction which exists in separate sections of the control store, are found in the select phase. Thus, microinstructions can be represented by fields contributed by separate control stores providing the mapping function. In addition, various computational facilities become available through the use of this mapping function. The number of select phase sections need not correspond to the number of read phase sections, but if they do so, this leads to a scheme which is easy for the microprogrammer to work with. Standard formats mean that the microprogrammer needs to remember and use only a few general principles.

It is an interesting hypothesis to consider whether

PLAs are only useful in replacing information which is characterised by sparseness or regularity. PLAs may only be useful for horizontal and minimally encoded instructions. (Note Fuller and Mathew, 1976, who make similar observations on replacing main memory with PLAs.) A functional memory may also be useful where regularity and sparseness are characteristics, but it has the advantages that standard memory circuits are used, and by compacting the microcode, this sparseness can be reduced by exploiting the regularity of the code.

We now consider further developments from this research.

7.6. Other Research Problems.

Fuller and Mathew (1976) have implied the splitting up in the use of microinstruction fields by using PLAs and note that the ordering of microinstructions is not related to sequencing since explicit addressing of the next microinstruction is presumed in their design. Current trends in research in microprogrammed control suggests the increased importance of looking at new ways of controlling parallel processing elements, especially with the use of the new larger ROMs and RAMs and the increasingly sophisticated microprocessors coming on the market.

It would be interesting to design in detail the generalised microprogrammed control unit proposed in Chapter 6 using the generalised functional memory approach and the ROM-type interconnection networks. Such an implement-

ation requires standard logic as DEC RTMs and similar macromodular units provide too many constraints on the system to be feasible building blocks. It has been demonstrated that the amount of custom MSI logic with this design is minimal and thus only a small number of such circuits is required.

Applications in parallel processing are probably the easiest to consider, to justify the large investment in new system organizations, microprograms and user programs. Hanlon (1966) in his survey of associative memories and processors has given a long list of possible applications which are suitable for exploitation by some means of parallel processing. A choice of one of these can lead to a design whose cost is justified.

Error detection and tolerance is an important area in microprogrammed control. With a large number of identical processing elements and other functional modules the occurrence of faults can usually be easily accommodated by removing the offending module from active status and perhaps replacing it by one of a small number of redundant modules. The microprogrammed control unit can easily detect such malfunctions and with the help of microdiagnostics reconfigure the system organization (e.g. Parhami and Avizienis, 1974).

Error detection and tolerance is much more difficult at the microprogrammed control unit stage. This problem was given serious consideration in the first research on functional memory (Flinders et al, 1969b), concerning the design of error detecting circuitry of particular use with

associative storage systems. They proposed duplicated storage arrays with compared output signals. They were also concerned with rendering a failed array in a duplexed associative storage system inactive (Flinders et al, 1969a). Flinders et al (1971) have described the use of functional memories in a framework where error detection is important. Similarly, Kautz (1967) was interested in the detection of faults in cellular arrays.

This is one area where there may not be the same flexibility in detection and avoidance of errors. It is usually costly to duplicate all of the circuitry in a microprogrammed control unit. The possibilities of much redundancy is not likely to be easily reconfigured because of the specific operations of each section of the micro-programmed control unit. Special circuitry to handle error detection and tolerance would lead to extra complexity. In a design such as this where complexity is at a minimum, such extra complexity would form a high ratio to the cost of the original circuitry.

It would be an interesting research problem to study how it would be possible to structure the microinstruction format to facilitate error detection and recovery. It may very well be possible for a particular control unit design to use this format together with the data encoded in the functional memories to provide an implicit detection and correction facility which would be dynamic and thus lead to very fast restructuring of the control unit in such a way as to minimize the impact of errors during microprocessing.

Such techniques have already been mentioned for the

case of the data processing structure. Here, however, extra circuitry would be required to provide control signals showing the status of any malfunctioning modules in the control unit. The microprograms would have to be correspondingly larger to handle all possible defects. Nevertheless, this would not require larger functional memory units as the use of dynamic loading of microprograms has been adequately demonstrated as an efficient way to increase the effective size of the microprogram.

APPENDIX A

A DESCRIPTION OF THE ASSOCIATIVE PROCESSOR IMPLEMENTATION

This appendix contains :

(1) a hardware description of the AP

(2) a software description of the AP

(3) the application program listings

(4) sample output

APPENDIX A
A DESCRIPTION OF THE ASSOCIATIVE PROCESSOR IMPLEMENTATION

1. Introduction.

A description is given here of the associative processor as constructed using DEC RTMs and other circuitry. The hardware specification is covered first. The use of the lights and switches of the control console, the printed circuit boards used in each panel, control flow diagrams and other wiring details are given here.

This is followed by the software description which categorizes all the instructions available and describes their use. The application program listings are next and these are followed with sample output from the execution of both these programs.

2. Hardware Description.

2.1. Control Console.

The use of the various lights and switches of the control console are detailed in Table A-1.

The control switches are self-explanatory. The PAUSE switch, however, should be noted. While stepping through the interpretation of an instruction, a large number of steps may be necessary if a new microprogram has to be loaded. This inconvenience can be eliminated by setting the PAUSE switch and unsetting the AUTO/MANUAL switch. The processor will run until just after the new microprogram if needed has been loaded. Single-step operation can be resumed by setting the AUTO/MANUAL switch and unsetting the PAUSE switch. Reference should be made to the flow control diagrams for a further explanation of this operation.

Lights

Row 1 (row of 8 control lights)

1	stop
2	character received over link
3	character transmitted over link
4	device unavailable flag, for link transmitter

Row 2 (upper row of 16 lights)

microprogrammed control unit bus sense register

Row 3 (lower row of 16 lights)

selector 5 - processing element bus sense register
(any PE can be chosen by changing the plugs)

selector 7 - transmitter data buffer
(lower 8 lights used only)

Control Switches

1	pause
4	auto/manual
6	start
7	single-step
8	power clear

Table A-1. Lights and Control Switches

2.2. Printed Circuit Board Distribution.

The printed circuit board distribution is given in Table A-2. Column A refers to the upper row of the plugboard panel and column B to the lower. A1-16, B1-16, etc, refer to the various slots in the panel, going from left to right. These references were used extensively in the control flow diagrams during development but are omitted in this appendix for clarity.

In some cases, more Register Transfer Modules were used then would be necessary if all circuits available on each module were utilized. This was in order to cut down on the number of long leads running across two or more panels.

2.3. Control Flow Diagrams.

The control flow diagrams are given at this point. They are mostly self-explanatory. The following wiring details are omitted in these diagrams for clarity but they are described here because they played an important part in keeping a careful track of all wiring throughout the system.

A connection is described as follows :

an/bcm

where a = the panel specification (A is at the top, E at the bottom)

n = the slot number, 1 to 16, running from left to right

b = A, the upper row of the panel slots; B, the lower row

cm = as specified as a pin connection in the DEC PDP16 Handbook, e.g. J1.

Thus C15/BF1 refers to connection F1 in the lower 15th slot of the third panel from the top of the cabinet.

Control modules are usually represented by a control block in the flow diagrams. The location of the control module is given by a three-character descriptor at the

Table A-2. Printed Circuit Board Distribution

A - an A21 panel

	A	B
A1	4-input merge	4-input merge
A2	4-input merge	4-input merge
A3	4-input merge	2-input merge
A4	4-input merge	2-input merge
A5	two-way branch module	evoke module
A6	eight-way branch module	evoke module
A7	no-op module	evoke module
A8	two-way branch module	evoke module
A9	flag module	evoke module
A10	two-way branch module	evoke module
A11	evoke module	evoke module
A12	evoke module	evoke module
A13	-	4-input merge
A14	4-to-16 decoder	-
A15	4-to-16 decoder	-
A16	4-to-16 decoder	-

B - an A20 panel

B1	bus termination module
B2	transfer register (RP1)
B3	transfer register (RP2)
B4	transfer register (RP3)
B5	transfer register (RP4)
B6	general purpose interface (MCU - PE)
B7	scratch-pad memory (SP1)
B8	scratch-pad memory (SP2)
B9	scratch-pad memory (SP3)
B10	scratch-pad memory (SP4)
B11	scratch-pad memory (RP1)
B12	scratch-pad memory (RP2)
B13	scratch-pad memory (RP3)
B14	scratch-pad memory (RP4)
B15	transfer register (for RPR)
B16	transfer register (for SPR)

Table A-2. Printed Circuit Board Distribution (cont.)

C - an A20 panel

C1	bus sense module
C2	general purpose arithmetic - registers
C3	general purpose arithmetic - control
C4	byte register (instruction register)
C5	transfer register (enable register)
C6	byte register (status register)
C7	byte register (constant register)
C8	transfer register (X register)
C9	byte register (vertical register) (The high order byte of the register is mapped onto the low order byte of the bus and the low order byte of the register is mapped onto the high order byte of the bus)
C10 to C13	-
C14	general purpose interface (AP - SP)
C15	general purpose interface (AP - SP)
C16	bus termination module

D - an A21 panel

	A	B
D1	inverter module	4-to-16 decoder
D2	4-input merge	4-input merge
D3	4-input merge	4-input merge
D4	4-input merge	4-input merge
D5	evoke module	two-way branch module
D6	evoke module	subroutine module
D7	evoke module	subroutine module
D8	evoke module	two-way branch module
D9	evoke module	evoke module
D10	evoke module	evoke module
D11	evoke module	evoke module
D12	evoke module	evoke module
D13	4-input merge	4-input merge
D14	4-input merge	link clock
D15	link transmitter (AP to SP communication)	
D16	link receiver (SP to AP communication)	

Table A-2. Printed Circuit Board Distribution (cont.)

	A	B
E - an A21 panel		
E4	4-input merge	-
E5	evoke module	two-way branch

----

lower right side of the control block :

anb

where a = the panel specification.

Connections to more than one point are given as follows :

an/bcm,an/bcm

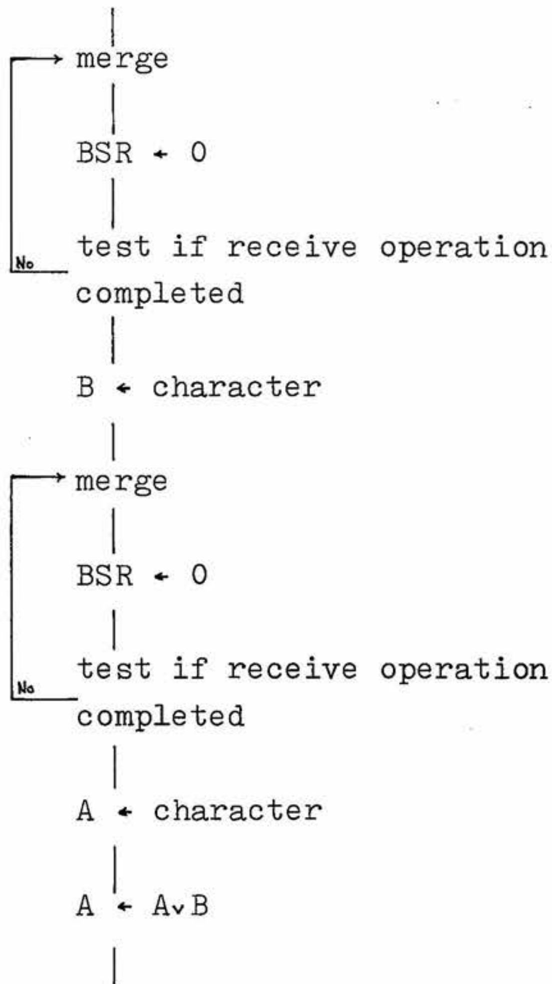
or an/bcm,bcm

In many cases, there may not be enough input terminals to a particular data module. In these cases, the number of input terminals is increased by using 4-input merge modules. This was shown on the wiring diagrams by giving the desired input terminal and then in brackets, the actual input terminal of the 4-input merge module and its output terminal which would be connected to the desired input terminal. E.g.

D2A/P1 (through D2B/M1 to S1)

Section 2.4 follows after the control flow diagrams.

Fig. A-1. Control Flow Diagram - input operation



Notes :

(1) This control "module" is used as a subroutine by the other control flow diagrams.

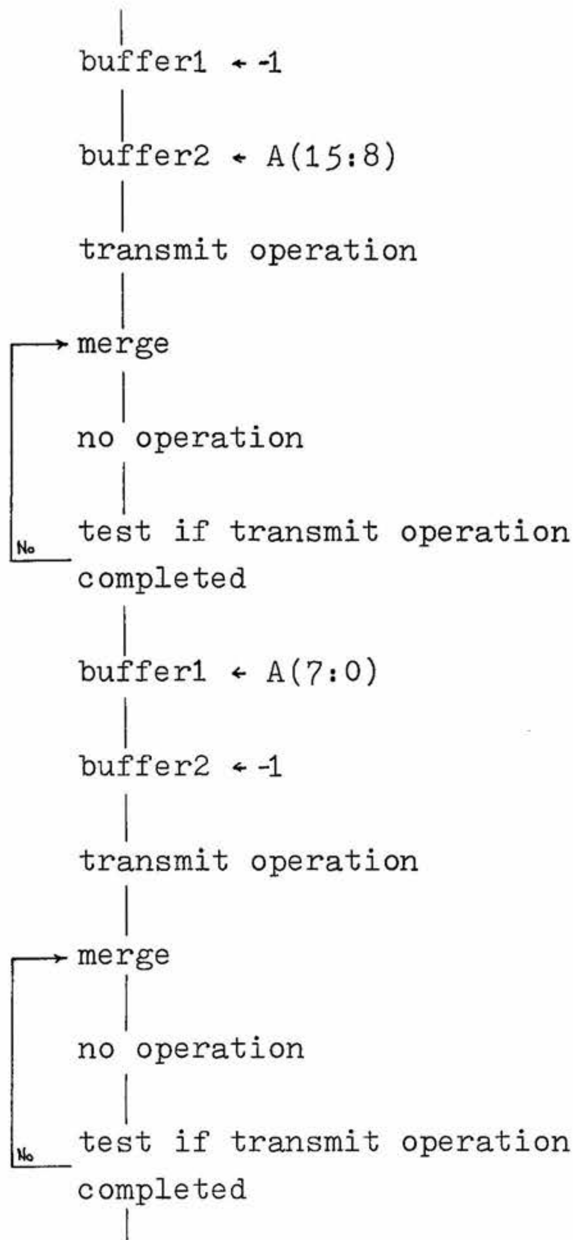
(2) The less significant byte of the 16-bit word is accepted first over the serial link. The more significant byte follows.

(3) B receives the first byte in the lower 8 bits and A receives the second byte in the higher 8 bits.

(4) After calling this subroutine, the value in A can be used directly, or moved to another destination register.

(5) This subroutine can be called by a bit set in the microinstruction. It is also used to read in an instruction and to load the microprogram.

Fig. A-2. Control Flow Diagram - output operation



Notes :

(1) The more significant byte is transmitted first and followed by the less significant byte.

(2) This module is called by a bit set in the microinstruction. It does not form part of the interpretation cycle.

Fig. A-3. Control Flow Diagram - load the functional memories

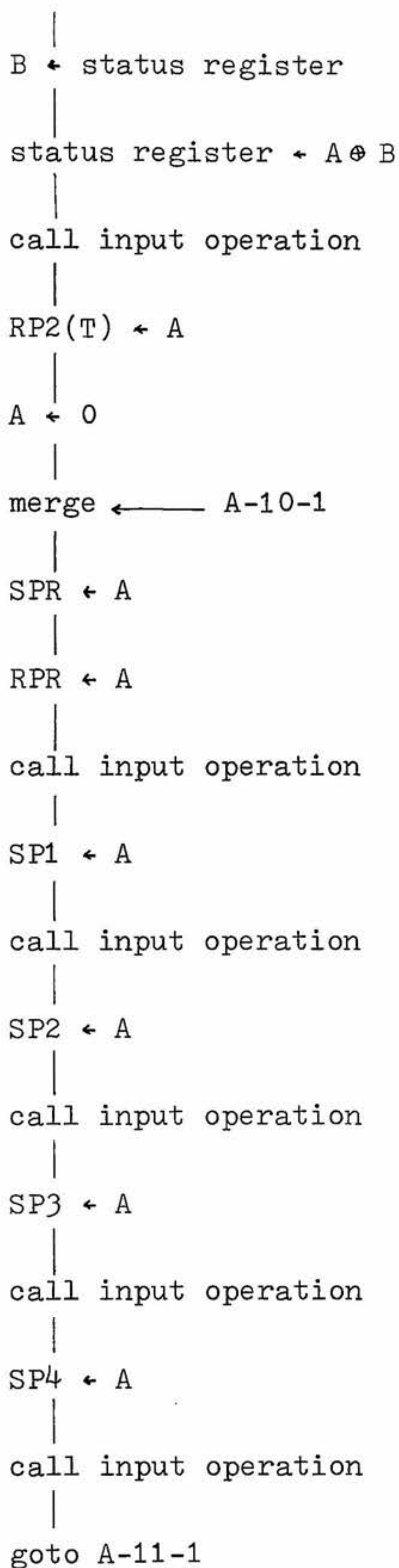
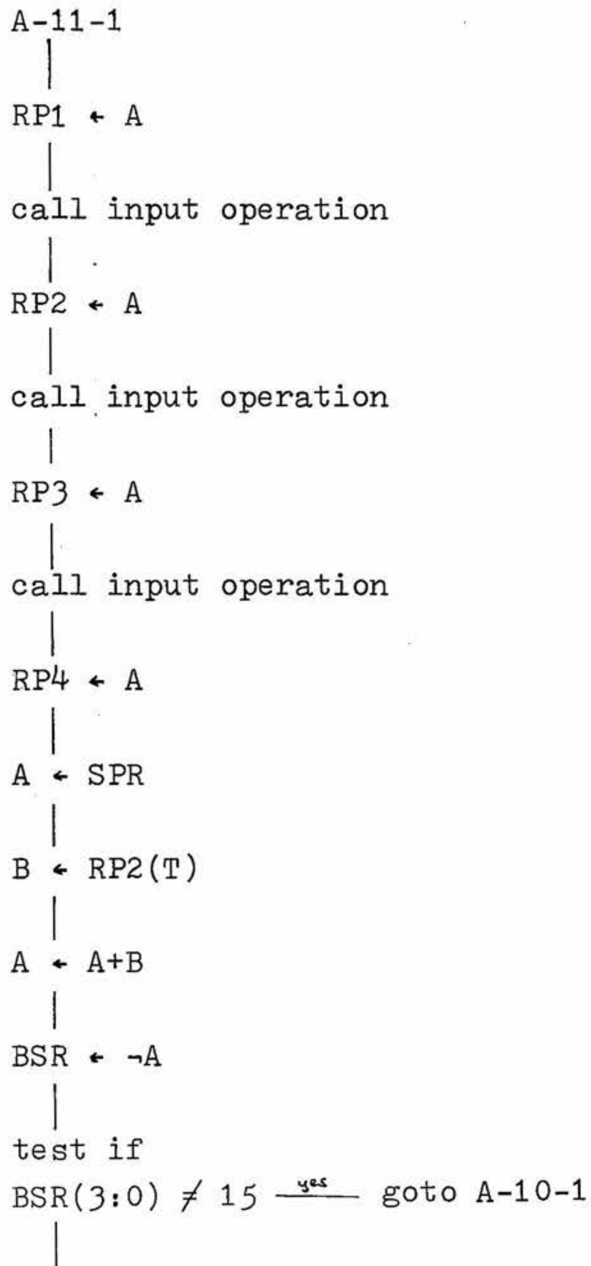


Fig. A-3 (cont.) Control Flow Diagram - load the functional memories



Notes :

(1) The input operation subroutine is used a number of times and each "row" of the microprogram is loaded consecutively.

(2) The status register of the MCU is updated to show the new control level of the AP.

(3) This module is activated only if the new instruction to be executed requires a different microprogram than the current loaded microprogram.

Fig. A-4. Control Flow Diagram - main interpretation cycle

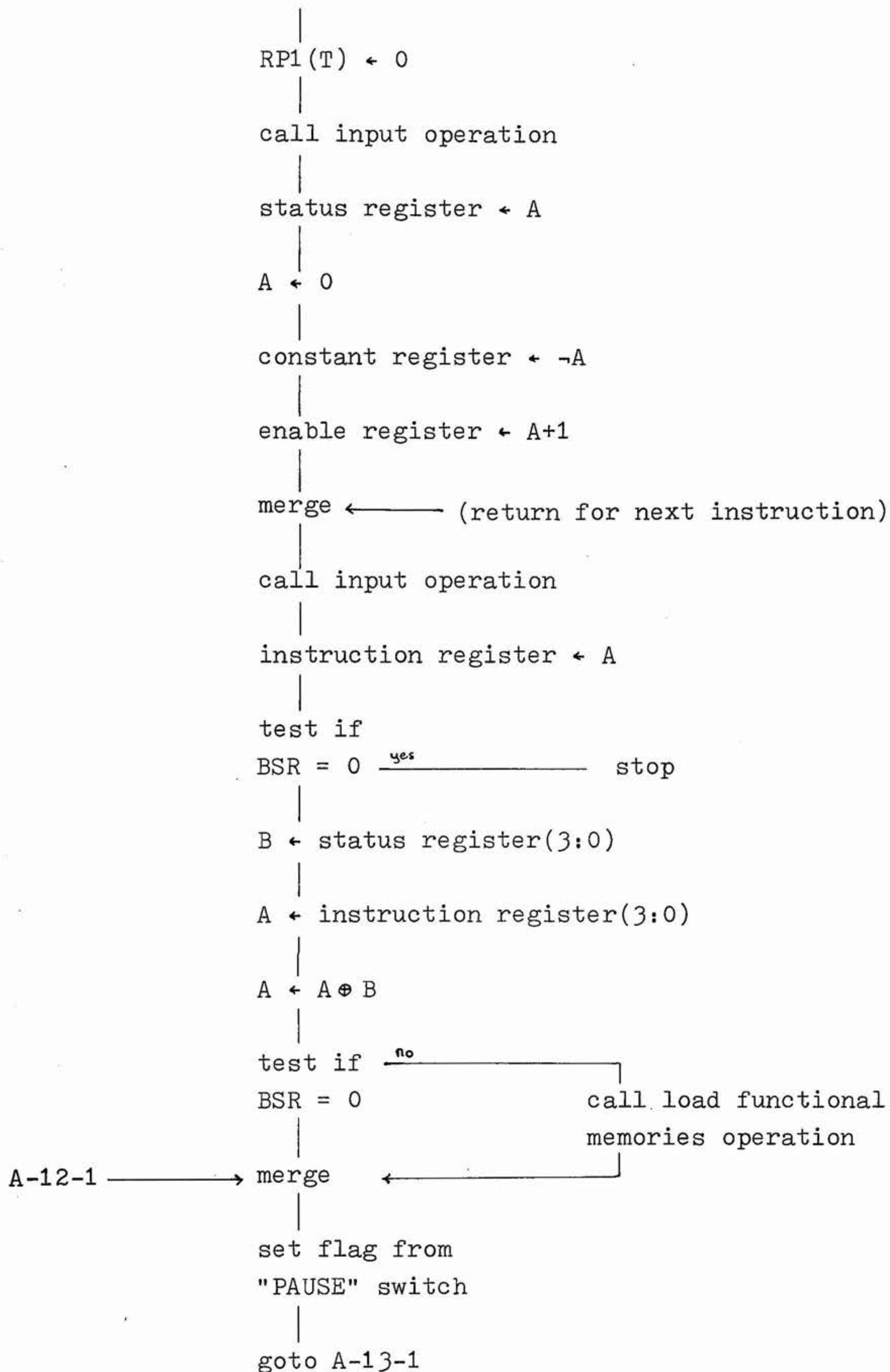


Fig. A-4 (cont.) Control Flow Diagram - main interpretation cycle

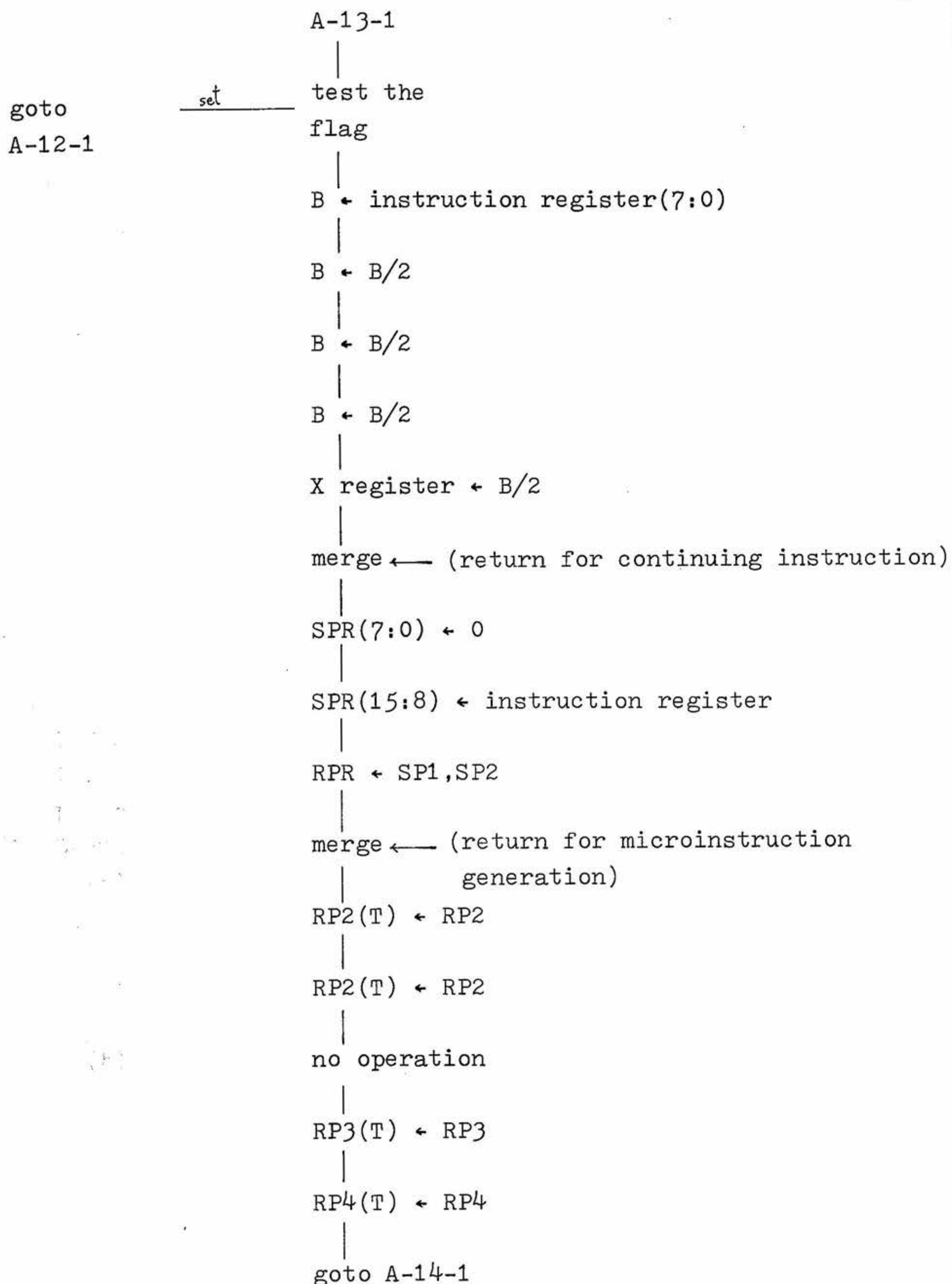
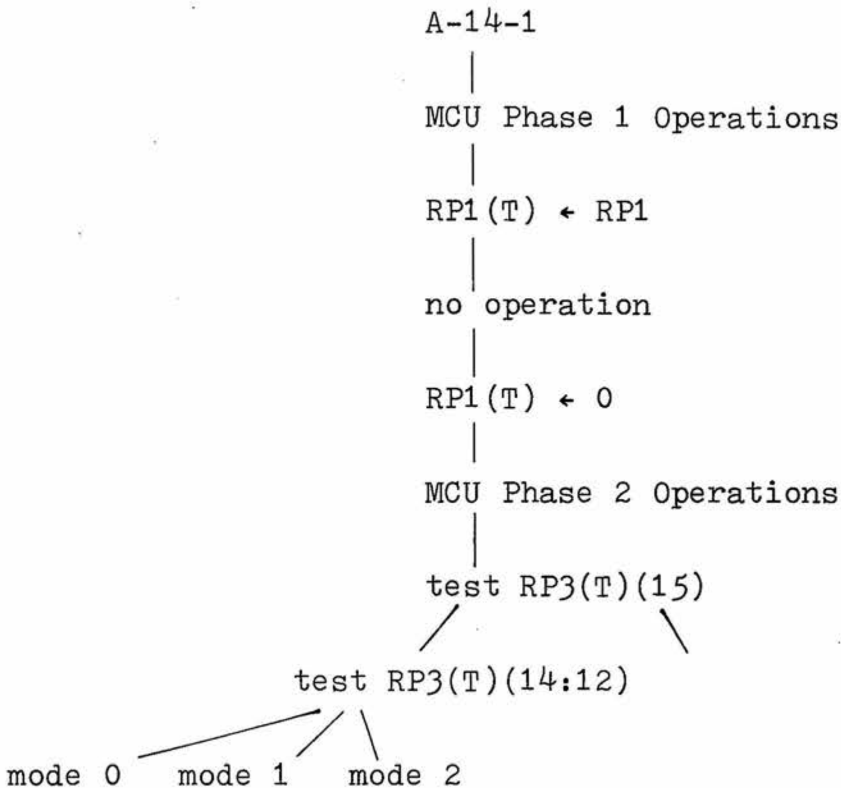


Fig. A-4 (cont.) Control Flow Diagram - main interpretation cycle



Notes :

(1) At the end of the last stage of this diagram, the next SPR contents are available in the A register.

(2) The main operation which is given here is the micro-program interpretation operation.

(3) Operation proceeds as follows :

(a) Initialization - RP1(T) is reset, the initial contents of the status register are transmitted from the SP and stored in the status register, the constant register and enable registers receive their initial values.

(b) Instruction received - the instruction is placed in the instruction register. If zero, the processor halts. Otherwise, the status register is checked to see if the appropriate microprogram is loaded. If not, the status register is updated, and the new microprogram loaded into

Fig. A-4 (cont.) Control Flow Diagram - main interpretation cycle

the functional memories. The processor "pauses" if specified by the PAUSE switch.

(c) Commencement of instruction interpretation - the X register is set from the instruction register. SPR is loaded with the appropriate value. The select phase functional memories are used and RPR is loaded with the appropriate value. The microinstruction is generated and RP2(T), RP3(T) and RP4(T) are set.

(d) MCU Phase 1 - this is a set of routines which provide the microorders for the MCU section of the microinstruction which have to be obeyed before the PEs are exercised. This essentially performs sequentially what would in an ideal machine be performed in parallel.

(e) PE operation - RP1(T) is loaded from the microinstruction and this causes the PEs to execute their own microorders. A NO-OP module is required so that the control lines are active for a sufficient length of time for correct operation.

(f) MCU Phase 2 - this is the set of routines which provide the microorders for the MCU section of the microinstruction which have to be obeyed after the PEs have been exercised. Again, operations are performed sequentially which in an ideal machine would be performed in parallel.

(g) Next SPR determination - at the conclusion of Phase 2 operations, the next SPR contents are available in the A register. At this stage, the mode of interpretation field in the microinstruction is used, to determine how instruction interpretation will continue. This invariably leads back in a loop to some other stage of the main interpretation cycle.

Fig. A-4 (cont.) <u>Control Flow Diagram - main interpretation cycle</u>

(4) Mention has already been made to the fact that several microorders can be executed in parallel. It should be noted that in an ideal machine, it should be possible to execute MCU Phase 1, the PE operation and MCU Phase 2 in parallel.

(5) The (T) after RP2, for example, represents the transfer register which is ordinarily loaded from the RP2 scratch-pad memory. Note the RP2(T) is loaded twice from RP2 and is followed by a NO-OP. Due to timing problems when stepping through the control sequence in MANUAL mode, this was found to be necessary.

2.4. MCU Phase 1 Operations.

We treat here the control flow operations associated with microorders which are executed before any PE operations. Control flow diagrams are not given.

Right Shift Circular Enable Register

The enable register is shifted one bit right, with the bit being shifted out being replaced in the left most bit. Only the lower byte of the register is used, since only 8 bits are required for the PEs.

I/O Operation

The I/O input operation requires that the output buffer of the general purpose interface connected to the PEs be loaded before the PEs are exercised.

A test has to be made on a bit in the RP1(T) section of the microinstruction to find out whether the I/O microorder is to be interpreted as an input or output microorder. This was done primarily to minimise the number of microorder bits and to exploit the redundancy available.

If two bits were allowed for I/O operation, it would be possible to control I/O as follows :

(a) 0 0 - receive data word from SP
(b) 0 1 - receive command word from SP
(c) 1 0 - transmit data word to SP
(d) 1 1 - transmit status word to SP

The data word would come from the A register and the status word from the status register. This is similar to the use of control bits to control a device or channel, except that the command bits are originating within the "device" itself, from its microprogram. If a further two bits were available for similar operations to the PEs, the four bit field which would result would be able to provide for very flexible communication between SP, MCU and PEs.

The reception of a command word, which would simply be the next word in the instruction stream provided by the SP, would essentially allow variable length instructions to be executed.

In this realization, I/O communication has been kept very simple, and the input operation consists of loading the general purpose interface output buffer. This operation may be used simply to reset the output buffer, since it can cause interference during PE-PE communication.

Set Enable Register from Immediate Field

This is the only use of the immediate field of the microinstruction. Whilst it provides complete control of which PEs are enabled, other means of enabling the PEs would not require this immediate field. For example, there could be microorders resetting the enable register, complementing it, and various shift operations.

Reset Vertical Register to X'FFFF'

The vertical register is set to all ones. The vertical register is used in the output operation to select the PEs from which output is to occur. Thus, this operation resets the vertical register to its initial state.

Set Enable Register from Condition Code Test Operation

This operation sets the enable register from the condition codes returned from the PEs. Since the result from the PEs are returned in the higher byte of the 16-bit word transmitted to the MCU from the PEs, a transformation must take place so that this value is in the lower byte of the enable register. The vertical register is used to provide this transformation. Note that the vertical register must then be reset, if it is necessary to use it in any further operation which requires it to be in its initial state.

Complement the Enable Register

This operation complements the value of the enable register. It is valuable when a particular set of operations is to be executed by a subset of the PEs, and another set of operations executed on the complementary subset of PEs.

2.5. MCU Phase 2 Operations.

We treat here the control flow operations associated with microorders which are executed after any PE operations.

I/O Operation

This is the AP to SP output operation. As noted for the corresponding input operation, a test is made of a bit in the RP1 section of the microinstruction before this operation is performed.

The output operation is selective and the vertical register specifies which PEs will be activated on output. Thus as a "1" is shifted in the enable register to activate each PE in turn, its value is ANDed with the contents of the vertical register, to find out whether it actually should perform output or not. If not, X'FFFF' (all ones) is substituted in place of the value which would have been output.

The SP interprets X'FFFF' on input as a null value and does not place it in the data block that its "input stream pointer" is pointing to. This is a dangerous convention, and perhaps X'8000' should have been used instead. This would have been more difficult to substitute for the "disabled" PE's values. However, in an ideal machine, the "1" in the enable register would be "skipped" across to the next "1" set in the vertical register. No superfluous output would occur.

Set Status Register from Condition Code Test Operation

This microorder sets the status register from the condition code test. The status register is used in the next operation to set the vertical register. Thus it is used to preselect a subset of PEs on which later condition code test operations are to be considered significant.

Set Vertical Register from Status Register AND Condition Code

The vertical register affects which PEs will be used in output to the SP. The status register is thus being

used to hold a constant value which will be used repeatedly by a number of different comparison operations.

Set Vertical Register from Vertical Register AND Condition Code

This is similar to the previous operation, but with the vertical register being used instead of the status register. This allows a series of tests to be used to select PEs for later output. Each new test is applicable to only those PEs which have been previously selected for output operations.

Prepare for Next Cycle

This is the last section for the MCU Phase 2 operation and is always executed. If more Phase 2 microorders are added, they must operate before this part.

This section of the control flow sets the A register with the data for the SPR for the next interpretation cycle. In doing so, it may use the first part of the user instruction again. This is determined by a bit in the microinstruction which is tested at this stage. The next SPR data is primarily determined by shifting the contents of the RP3(T) section of the microinstruction, whilst shifting in a four-bit field from the RP4(T) section of the microinstruction.

2.6. Mode of Interpretation Operations.

After the preparation of the next cycle covered above, one of a number of modes of interpretation are used before returning to the beginning of the main interpretation cycle. The modes developed are covered here.

Mode 0

This is a very simple use of the X register. If RP3(T) is non-zero, the next instruction interpretation is commenced. Otherwise, the X register is checked for zero. If so, the next user instruction is used, otherwise the X register is decremented and the same instruction

repeated. The X register, the third field of the user instruction, is thus used as a count. This is very useful for the I/O instructions, shift instructions and the multiplication instruction.

Mode 1

Whereas with Mode 0, the X register was used as a count, with Mode 1, the X register is used directly to modify the contents of the next SPR data. This allows direct control of the MCU operation by the use of the third field of the user instruction.

Note that a bit of the microinstruction is used as an "activate" bit for this operation. This bit could have multiple uses as long as there was no conflict in interpretation.

Mode 2

This mode simply allows the MCU to continue with the next instruction. This is necessary to stop current instruction interpretation when the RP3(T) part of the microinstruction is non-zero.

RPR Generation

The above modes of interpretation lead directly into a section of control which generates the new RPR data from the SPR data.

Ideally, all four SP functional memories should be accessed and the data ORed onto the bus for loading the RPR. Although this method was originally used, problems arose which were associated with the particular bit patterns output from the SP functional memories - that is, successful operation was data dependent. Thus, the longer approach of ORing in the data from each SP functional memory separately had to be adopted.

Indeed, simultaneous loading of the bus by a number of scratchpad memory modules, is an operation for which the RTM design is not designed, although it is successful for operations involving two scratchpad memory modules.

2.7. PE-MCU and PE-PE Data Paths.

The interconnections between the PEs and the MCU have been broadly described elsewhere. Data flow to the PEs is from two sources. Each PE can receive data from the MCU or a neighbouring PE (or both at the same time). Thus, 2-input merge modules are required for each PE. Data flow from the PEs to the MCU is via 4-input merge modules leading through 2-input merge modules to the general purpose interface in the MCU.

2.8. Status Section.

This section describes the circuitry that returns status values of the PEs to the MCU. This is done on the same "channel" as the data word output from the PEs to the MCU.

A diagram of the wiring of this stage is given in Fig. A-5. The connections are numbered and described below.

Lines 1, 3, 5 and 7 are connected to the condition status lines from the Bus Sense Register of the PE. Lines 2, 4, 6 and 8 are from the 4-bit field of the microinstruction specifying the condition code test required. Since negative logic is used throughout this operation, the merge modules act as AND gates.

The outputs of these merge modules 9, 10, 11 and 12 are connected to an enable module (13, 14, 15, 16). Depending on the appropriate bit being set in the enable register, line 31 allows the output of these signals at 17, 18, 19 and 20. These signals are then merged at 21, 22, 23 and 24 into a single signal, 25, and this is merged with the PE data path, 27-28, to give the combined output signal at 29.

2.9. Miscellaneous Circuitry.

Some of the additional circuitry required was as follows :

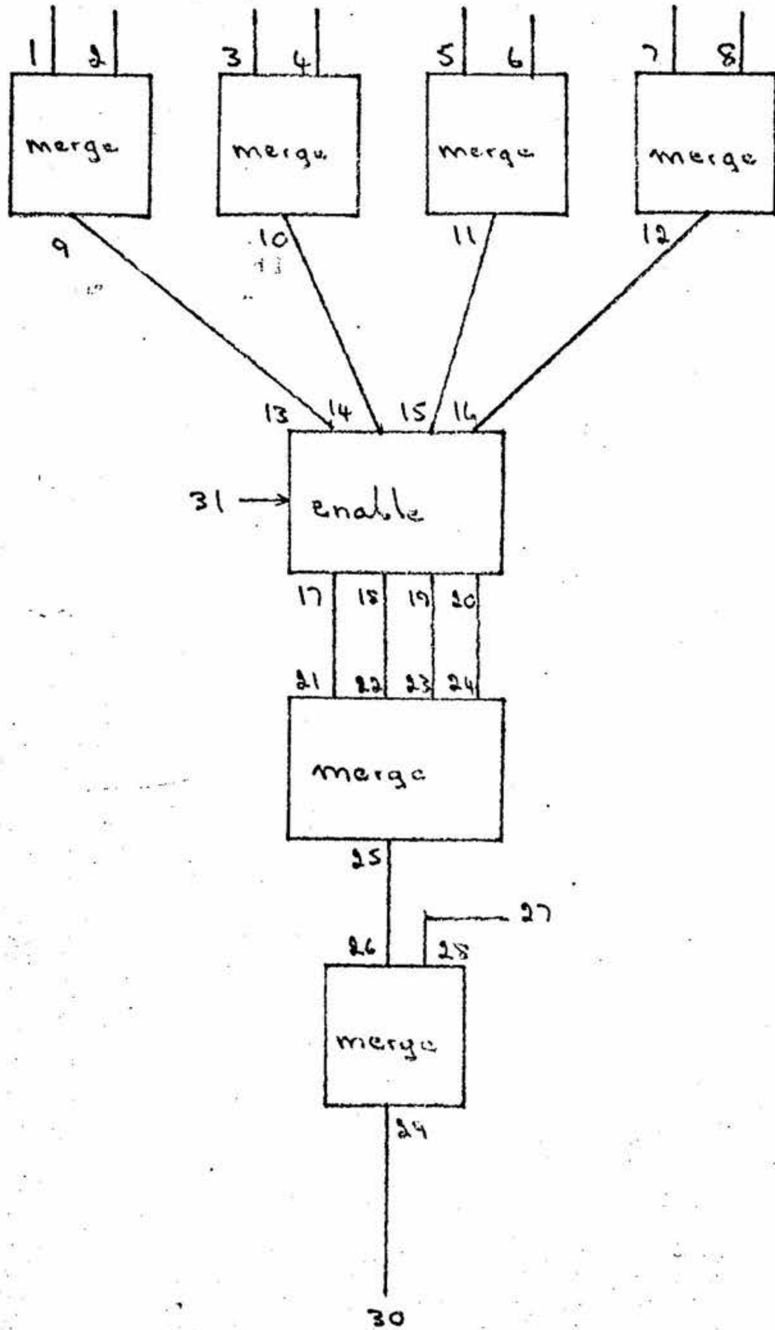


Fig. A-5. The Status Section

(1) Each PE has its control lines associated with the output of enable modules. These are essentially the PE microorders. Each PE has two enable modules associated with it. These modules make sure that the microorders are passed to the PE only if the enable line for that PE has been set.
(2) The 4-bit arithmetic operation field of the micro-instruction is decoded for the PEs by a common set of circuitry. Ideally, such decoding should be done at the PE level but the fact that undecoded signals are required by DEC RTMs dictated this policy.
(3) Similarly, the address lines of the PE scratchpad memories are decoded by a common set of circuitry. Again, the fact that undecoded signals were required at the PE level made this necessary. These signals were sent whether or not the PEs were enabled.

2.10. Merging.

It was not known in advance how many control lines needed to be connected to a particular input terminal. The following policy was therefore pursued. If there were n terminals for a particular operation, when n-1 terminals had been used up in connections, the nth terminal would be connected to the output terminal of a 4-input merge module. After three of the four inputs had been used, the fourth terminal would again be connected to the output terminal of a 4-input merge module. Thus, a particular signal may have to pass through a number of levels of merge modules. The number of levels may be larger than if a more symmetric tree-type interconnection structure was employed, but this would require that the number of control lines to a terminal be known in advance, which was not possible in an evolving design.

3. Software Description.

Here, a description is given of all the instructions implemented by microprogramming the associative processor.

The control level refers to the microprogram which must be loaded for the instruction to be interpreted correctly. The machine instructions are given in hexidecimal. The description of some instructions include relevant programming notes.

3.1. RESET n

control level : 0

semantics : this command resets the enable register to a value determined by the pointer n. n is a pointer to a table.

machine instruction : 1n00

programming notes :

(1) "n" points to the following table :

n	value	
0	11111111	all PEs enabled
1	00000001	last PE enabled

Other entries can be made in this table.

(2) In addition, the B register of all PEs is reset to zero. The reset command is thus handy when the B register may have been corrupted.

(3) The output buffers of all PEs are reset to all ones. This makes sure that there is no interference between the output buffers of the different PEs and the possible inputs through a general purpose interface.

(4) The reset command must be executed after a MANUAL CLEAR and before a WRITE or SEEK-type operation.

3.2. LDA x

control level : 0

semantics : loads the A register (accumulator) from the scratchpad location specified by x (for all PEs

enabled).
machine instruction : 2x00

3.3. LSHFT n

control level : 0
semantics : left shifts logical, the contents of the A register of all PEs enabled, the number of places specified by the number n.
machine instruction : 30m0
where m=n-1

3.4. RSHFT n

control level : 0
semantics : right shifts logical, the contents of the A register of all PEs enabled, the number of places specified by the number n.
machine instruction : 40m0
where m=n-1

3.5. STA x

control level : 0
semantics : stores the contents of the A register (accumulator) in the scratchpad location specified by x (for all PEs enabled).
machine instruction : 5x00

3.6. RSTVR

control level : 0
semantics : this command resets the vertical register back to X'FFFF'.
machine instruction : 6000
programming notes :
(1) This would be required before an unselected write operation for example.

3.7. STZ x

control level : 0
semantics : a zero is loaded into the scratchpad location
addressed by x (for all PEs enabled).
machine instruction : 7x00

3.8. RESTS n

control level : 0
semantics : the same as RESET except that only the enable
register gets set.
machine instruction : 8n00
programming notes :
(1) This is the short form of RESET.

3.9. TRANS

control level : 0
semantics : transmits the data in the accumulator to
the output buffer of each PE enabled.
machine instruction : 9000

3.10. RECV

control level : 0
semantics : receives I/O data and places it in the
accumulator for all PEs enabled.
machine instruction : A000

3.11. ZAC

control level : 0
semantics : zeroes the accumulator of all PEs enabled.
machine instruction : B000

3.12. TEST ER,x

control level : 0
semantics : a test is made on the condition codes of all
PEs enabled, similar to the SEEK instruction. The
enable register is set directly from the result.
machine instruction : Cx00
programming notes :
(1) The vertical register is used, and thus corrupted.

3.13. TEST SR,x

control level : 0
semantics : a test is made on the condition codes of all
PEs enabled, similar to the SEEK instruction. The
status register of the AP is set directly from the
result.
machine instruction : Dx00

3.14. ARSHFT n

control level : 0
semantics : right shifts arithmetic, the contents of the
A register of all PEs, the number of places specified
by the number n.
machine instruction : E0m0
where m=n-1

3.15. READ x,y

control level : 1
semantics : reads data from the SP (provided by the SP
output stream) into consecutive locations of
consecutive PEs. Each PE is written into from
scratchpad location 0 upwards. "x" specifies the
number of words of each PE which are to be filled.
"y" specifies the number of PEs which are going to
be written into consecutively.

machine instruction : 1vw1
where v=x-1
and w=y-1

programming notes :

(1) The enable register must be set up beforehand with only the PE enabled which is just before the PE at which input is to be commenced.
(2) At the completion of the instruction, note that the PE-MCU output buffer still contains the most recent data word to be transferred. Some other instructions may be sensitive to this.

3.16. WRITE x,y

control level : 1

semantics : writes data into the SP (the SP input stream) from consecutive locations of consecutive PEs. Each PE data word is accessed from the scratchpad location 0 upwards. "x" specifies the number of words of each PE which are to be output. "y" specifies the number of PEs which are to be read out of consecutively.

machine instruction : 2vw1
where v=x-1
and w=y-1

programming notes :

(1) The enable register must be set beforehand with only the PE enabled which is just before the PE at which output is to begin.
(2) At the completion of the instruction execution for each PE, it should be noted that the last word to be transmitted for the PE is always a string of ones. Thus, if three words of each PE are required, four should be requested. The fourth word to be transmitted will be a string of ones. This rule is required because of the I/O structure connecting PEs to the MCU. The microprogram could easily be changed if the I/O structure was different.
(3) This operation is a selective write, in the sense

that writing out only occurs if the bit in the vertical register corresponding to the bit set in the enable register for the current PE, is also set. Otherwise, the output from this PE is ignored. The vertical register is normally set to all ones and the only instructions affecting it are certain SEEK and TEST instructions.

(4) Note that an output of X'FFFF' is not stored by the SP but is simply ignored. This allows the selective write to be implemented.

(5) Note that 20x1 will be a dummy I/O instruction.

3.17. ADD x

control level : 2

semantics : adds the contents of the scratchpad word addressed by x to the contents of the A register (accumulator), in all PEs which are enabled.

machine instruction : 0x02

programming notes :

(1) Since the B register is used in this instruction, it must be reset back to zero after completion of the microroutine before executing any instructions that use the B register implicitly.

(2) Location F can not be used.

3.18. SUBT x

control level : 2

semantics : subtracts the contents of the scratchpad word addressed by x from the contents of the A register (accumulator), in all PEs which are enabled.

machine instruction : 1x02

programming notes :

(1) Since the B register is used in this instruction, it must be reset back to zero before certain other instructions are executed.

(2) Note that 1x12 can be used in getting the modulus of a number.

(3) Location F cannot be used.

3.19. MULT x

control level : 2

semantics : multiplies the multiplicand, stored in location x, the multiplier, stored in location x+1 and places the higher 16-bit result in the A register and the lower 16-bit result in location F.

machine instruction : 2xF2

programming notes :

(1) Since the B register is used in this instruction, it is reset back to zero before completing the microroutine.

(2) The multiplication instruction must operate on all PEs and the enable register should be set to all ones.

(3) The A register must be set to zero, before this instruction is obeyed.

(4) Before this operation, the programmer must test the multiplier for negativity and change signs if necessary.

(5) "x" must be even for a normal multiplication.

(6) "x" should be odd if a squaring operation is required. In this case, the multiplier (stored in x+1) is not required.

(7) $x \neq F$ since the location F is reserved for part of the result.

3.20. ADDD x

control level : 2

semantics : adds x to the accumulator and adds in the carry bit from the overflow flag of the Bus Sense Register.

machine instruction : 3x02

programming notes :

(1) This instruction is used in double precision arithmetic.

(2) For this instruction, all the PEs must be enabled.

(3) The overflow flag may need to be checked.

3.21. SUBTD x

control level : 2

semantics : the location x is subtracted from the accumulator, and the result decremented if the overflow flag has been set by a previous instruction.

machine instruction : 4x02

programming notes :

(1) This instruction is used in double precision arithmetic.

(2) For this instruction, all the PEs must be enabled.

(3) The overflow flag may need to be checked.

3.22. SEEK x,y,a;zzzz

control level : 3

semantics : makes a comparison between the y scratchpad location's contents and the data zzzz. The comparison is an ORing of the following tests specified by the bits in the x field :

1 BSR=0

2 BSR>0

3 BSR<0

4 overflow flag

The test takes place on all PEs enabled. The result of the test is returned to the MCU in the most significant 8 bits of the general purpose interface. Its use is specified by the field a, and described in the programming notes.

machine instruction : xya3 ; z'z'zz

programming notes :

(1) For a=3, the more significant byte of the SP status register has a bit set in each position corresponding to the status returned by the respective PE if enabled.
(2) When x=0, the effect of the test is as if x=F, i.e. the test is (BSR≠0 or overflow).
(3) When x=F, no comparison takes place and z'z' is returned as the new status register higher byte.
(4) When a=4, the SP status register is not changed and instead the AP status register is changed by the result of the condition code test.
(5) When a=5, neither the AP or SP status register is changed. Instead, the vertical register is loaded with the AP status register AND the condition code test.
(6) When a=6, the status information is not used at all. This value of "a" is useful for resetting the output buffer of the MCU-PE link.
(7) When a=7, then similar to a=5, we have the vertical register loaded with the vertical register AND the condition code test.
(8) This instruction corrupts the B register, i.e. it does not reset B back to zero. A reset command may be mandatory before an output command (WRITE x,y).
(9) Note that the vertical register may have been used for bit mapping and may need resetting.
(10) A hardware problem makes the placement of the seek instruction data dependent in some cases. There are places where the wrong data can be returned to the SP status register. This problem was solved by rearranging the instructions slightly for the application programs.

3.23. JZSR ; xxxx

control level : 8
semantics : this command is obeyed within the SP. The AP is not involved. The more significant byte of the

status register is checked. If all bits are zero, a jump is made to the location specified by xxxx. This location is relative to the start of the associative processor program.

machine instruction : 0008 ; xxxx

programming notes :

(1) There need not be any delay in the execution of this instruction from the AP point of view. Once it has been recognized by the SP, the instruction stream can be modified accordingly. Thus assumes that the status register of the SP has been prepared a number of instructions before this instruction is encountered. In this implementation, the SP looks ahead one instruction and whilst the AP is executing an instruction, the next instruction is executed to the extent that the SP may be involved in that instruction.

4. The Application Programs.

Listings of the application programs are given on the following pages.

Application 1.

```
                    £DEF *LIST=15
                    £DEF ACC=R0,DEVICE=R1,IODATA=R2,WORKSP=R3
                    £DEF STREAM=R4,STATUS=R5,PSTRM=R6,OSTRM=R7
                    £DEF TST=R8,TST1=R9,ACC1=R10,I=R11
                    £DEF ISTRM=R12
                    £DEF CONSL=1,TT=2,TRANS=X'99',RECV=X'89'
                    £DEF LPLK=X'98'
                    £DEF CALL=BAL R15
                    £DEF CALL1=BAL R14
                    £DEF CALL2=BAL R13
                    £DEF OUT=(R15)
                    £DEF OUTA=(R14)
                    £DEF OUTB=(R13)
                    £DEF INTRET=(X'0040')
                    /
                    / EXTERNAL INTERRUPT ROUTINE
                    /
0000' D000 0E62'    EXINT: STM ACC,MULTIPLE
0004' D100 0E82'           LM ACC,MTP
0008' 9F13                 AI DEVICE,WORKSP
000A' C910 0098            JUMP LPLKINT IF DEVICE=X'98'
000E' 4330 0078'
0012' C910 0002            JUMPS TTINT1 IF DEVICE=2
0016' 2337
0018' D000 0E82'           STM ACC,MTP
001C' D100 0E62'           LM ACC,MULTIPLE
0020' C200 0040            LPSW INTRET
                    /
                    / TELETYPE INTERRUPT
                    /
0024' 4820 0EA6'    TTINT1:  JUMP TTINT4 IF FLAG=0    / READ INTERRUPT
0028' 4330 006A'
002C' 48A0 0EA2'             ACC1=DTBFCT+OUTBUF    / WRITE INTERRUPT
0030' 4AA0 0E52'
0034' DA1A 0000              WD DEVICE,[ACC1]
0038' 2421                   DTBFCT=DTBFCT+1
003A' 6120 0EA2'
003E' 4820 0EA2'             JUMPS TTINT2 IF DTBFCT<400    / BUFFER LENGTH IS 400
0042' C920 0190
0046' 2114
0048' 2420                   DTBFCT=0
004A' 4020 0EA2'
004E' D32A 0000     TTINT2: JUMPS TTINT5 IF [ACC1]≠X'0A'    / EXIT IF NOT LF
0052' C920 000A
0056' 213B
0058' 2521                   FLAG=FLAG-1
005A' 6120 0EA6'
005E' 2137                   JUMPS TTINT5 IF FLAG≠0    / EXIT IF OUTPUT NOT ENDED
0060' 9D13          TTINT3: SS DEVICE,WORKSP
0062' 20F1                   JUMP TTINT3 IF 15
0064' DE10 0082'             OC DEVICE,READ
0068' 2302                   JUMPS TTINT5
006A' 9B12          TTINT4: RD DEVICE,IODATA
006C' D000 0E82'    TTINT5: STM ACC,MTP    / EXIT FROM INTERRUPT ROUTINE
0070' D100 0E62'             LM ACC,MULTIPLE
0074' C200 0040              LPSW INTRET
                    /
                    / LINEPRINTER INTERRUPT
                    /
0078' 4820 0EAA'    LPLKINT: JUMP (LPT)
007C' 0302
```

```
                 /
                 /
                 /
                 £DEF *LIST=14
007E' 4800       WRITN: X'4800' / WRITE MODE WITH INTERRUPTS ENABLED
0080' C800       WRITE: X'C800' / WRITE MODE WITH INTERRUPTS DISARMED
0082' 6400       READ: X'6400' / READ MODE WITH INTERRUPTS ENABLED
0084' 8000       LINK: X'8000' / LINK DISABLED
0086' 4000       LINKON: X'4000' / LINK ENABLED
0088' 8000       NORM: X'8000' / NORMAL CONSOLE MODE
008A' 4000       INC: X'4000' / INCREMENTAL CONSOLE MODE
008C' 1111       CONTROL: X'1111'
008E' 4000       OLDPSW: X'4000'
0090' 07DA'              ‡ST1
                 FM: / CONTROL LEVEL 0
0092' 0000       X'0000',X'0000',X'0000',X'0000',X'0000',X'00FF',X'0001',X'1800
00A2' 5F00       X'5F00',X'0110',X'0000',X'6001',X'9000',X'1101',X'0010',X'2000
00B2' 10FF       X'10FF',X'0200',X'3037',X'00F2',X'1C10',X'2280',X'0002',X'0800
00C2' 20FF       X'20FF',X'0300',X'0FF0',X'0000',X'1990',X'3300',X'0003',X'0004
00D2' 30FF       X'30FF',X'0400',X'0000',X'0000',X'4900',X'4400',X'0000',X'0080
00E2' 40FF       X'40FF',X'0500',X'0000',X'0000',X'2002',X'5500',X'0000',X'0020
00F2' 00F3       X'00F3',X'0600',X'0000',X'0000',X'0720',X'6600',X'0000',X'0000
0102' 70FF       X'70FF',X'0700',X'0000',X'0000',X'4002',X'7700',X'0000',X'0100
0112' 00F0       X'00F0',X'0800',X'0000',X'0000',X'0920',X'8800',X'0000',X'0000
0122' 80FF       X'80FF',X'0900',X'0000',X'0000',X'1040',X'9900',X'0000',X'0000
0132' 90FF       X'90FF',X'0A00',X'0000',X'0000',X'1002',X'AA00',X'0000',X'0000
0142' A0FF       X'A0FF',X'0B00',X'0000',X'0000',X'1980',X'BB00',X'0000',X'0000
0152' 00F4       X'00F4',X'0C00',X'0000',X'0000',X'0000',X'CC00',X'0000',X'0000
0162' 00F5       X'00F5',X'0D00',X'0000',X'0000',X'0000',X'DD00',X'0000',X'0000
0172' B224       X'B224',X'0E00',X'0000',X'0000',X'0000',X'EE00',X'0000',X'0000
0182' 0000       X'0000',X'0F00',X'0000',X'0000',X'0000',X'FFFF',X'0000',X'0000
                 / CONTROL LEVEL 1
0192' 0000'      X'0000',X'20F0',X'0010',X'0101',X'0000',X'0000',X'00F0',X'0013
01A2' 2000       X'2000',X'00E0',X'0020',X'0202',X'8020',X'1000',X'00F1',X'1012
01B2' 1000       X'1000',X'00D0',X'0030',X'0303',X'4040',X'2000',X'00F2',X'2012
01C2' 0000       X'0000',X'00C0',X'0040',X'0404',X'0720',X'3000',X'00F3',X'3012
01D2' 0000       X'0000',X'00B0',X'0050',X'0505',X'0000',X'4000',X'00F4',X'4012
01E2' 0000       X'0000',X'00A0',X'0060',X'0606',X'0000',X'5000',X'00F5',X'5012
01F2' 0000       X'0000',X'0090',X'0070',X'0707',X'0000',X'6000',X'00F6',X'6012
0202' 0000       X'0000',X'0080',X'0080',X'0808',X'0000',X'7000',X'00F7',X'7012
0212' 0000       X'0000',X'0070',X'0090',X'0909',X'0000',X'8000',X'00F8',X'8012
0222' 0000       X'0000',X'0060',X'00A0',X'0A0A',X'0000',X'9000',X'00F9',X'9012
0232' 0000       X'0000',X'0050',X'00B0',X'0B0B',X'0000',X'A000',X'00FA',X'A012
0242' 0000       X'0000',X'0040',X'00C0',X'0C0C',X'0000',X'B000',X'00FB',X'B012
0252' 0000       X'0000',X'0030',X'00D0',X'0D0D',X'0000',X'C000',X'00FC',X'C012
0262' 0000       X'0000',X'0020',X'00E0',X'0E0E',X'0000',X'D000',X'00FD',X'D012
0272' 0000       X'0000',X'0010',X'20F0',X'0F0F',X'0000',X'E000',X'00FE',X'E012
0282' 0000       X'0000',X'0000',X'20FF',X'0000',X'0000',X'F000',X'0000',X'F012
                 / CONTROL LEVEL 2
0292' 0000       X'0000',X'0000',X'0000',X'0000',X'A000',X'0100',X'0001',X'1000
02A2' 0002       X'0002',X'0110',X'0000',X'10F0',X'1501',X'1100',X'0010',X'2080
02B2' 0103       X'0103',X'0220',X'0000',X'FF0C',X'2002',X'2100',X'0020',X'3000
02C2' 8001       X'8001',X'0330',X'0000',X'30F0',X'1401',X'3100',X'0030',X'4000
02D2' 900D       X'900D',X'0440',X'0000',X'4104',X'4A91',X'4100',X'0040',X'5000
02E2' 0000       X'0000',X'0550',X'0000',X'0005',X'1591',X'5100',X'0050',X'6000
02F2' 0000       X'0000',X'0660',X'0000',X'5006',X'4A80',X'6100',X'0060',X'B080
0302' 0000       X'0000',X'0770',X'0000',X'0F07',X'1991',X'7100',X'0070',X'8800
0312' 0000       X'0000',X'0880',X'0000',X'6F08',X'1B00',X'8100',X'0080',X'9080
0322' 0000       X'0000',X'0990',X'0000',X'4F09',X'1600',X'9100',X'0090',X'A100
0332' 0000       X'0000',X'0AA0',X'0000',X'2FFA',X'0000',X'A100',X'00A0',X'0800
0342' 0000       X'0000',X'0BB0',X'0000',X'700B',X'0000',X'B100',X'00B0',X'7100
```

```
0352' 0000     X'0000',X'0CC0',X'0000',X'FF0E',X'0000',X'C100',X'00C0',X'0800
0362' 0000     X'0000',X'0DD0',X'0000',X'0002',X'0000',X'D100',X'00D0',X'C080
0372' 0000     X'0000',X'0EE0',X'0000',X'0000',X'0000',X'E100',X'00E0',X'D800
0382' 0000     X'0000',X'0F00',X'0000',X'0000',X'0000',X'F1FF',X'0000',X'0000
               / CONTROL LEVEL 3
0392' 0000     X'0000',X'0000',X'0000',X'0000',X'9000',X'0F00',X'1000',X'1000
03A2' 0010     X'0010',X'0100',X'0000',X'1002',X'2041',X'1100',X'1010',X'0002
03B2' 0020     X'0020',X'0200',X'0000',X'2003',X'0405',X'2200',X'1020',X'2012
03C2' 0030     X'0030',X'0300',X'0000',X'F0F1',X'0060',X'3300',X'1030',X'000[illegible]
03D2' 0040     X'0040',X'0400',X'0000',X'F0F4',X'0000',X'4400',X'1040',X'0020
03E2' 0050     X'0050',X'0500',X'0000',X'F0F5',X'0000',X'5500',X'1050',X'0040
03F2' 0060     X'0060',X'0600',X'0000',X'F0FF',X'0000',X'6600',X'1060',X'0000
0402' 0070     X'0070',X'0700',X'0000',X'F0F7',X'0000',X'7700',X'1070',X'0200
0412' 0080     X'0080',X'0800',X'0000',X'0000',X'0000',X'8800',X'1080',X'0000
0422' 0090     X'0090',X'0900',X'0000',X'0000',X'0000',X'9900',X'1090',X'0000
0432' 00A0     X'00A0',X'0A00',X'0000',X'0000',X'0000',X'AA00',X'10A0',X'0000
0442' 00B0     X'00B0',X'0B00',X'0000',X'0000',X'0000',X'BB00',X'10B0',X'0000
0452' 00C0     X'00C0',X'0C00',X'0000',X'0000',X'0000',X'CC00',X'10C0',X'0000
0462' 00D0     X'00D0',X'0D00',X'0000',X'0000',X'0000',X'DD00',X'10D0',X'0000
0472' 00E0     X'00E0',X'0E00',X'0000',X'0000',X'0000',X'EE00',X'10E0',X'0000
0482' 30F1     X'30F1',X'0F00',X'0000',X'0000',X'0020',X'FF00',X'2000',X'0000
               PROG:
0492' 1100     AP1: X'1100' / RESET 1   / INPUT - STAGE 1
0494' 6000          X'6000' / RSTVR
0496' 1471          X'1471' / READ 5,8
0498' 1000          X'1000' / RESET 0   / ALLOCATE BOX POSITION
049A' 2300          X'2300' / LDA 3
049C' 3050          X'3050' / LSHFT 6
049E' 0102          X'0102' / ADD 1
04A0' 40B0          X'40B0' / RSHFT 12
04A2' 5300          X'5300' / STA 3
04A4' 2400          X'2400' / LDA 4
04A6' 3050          X'3050' / LSHFT 6
04A8' 0202          X'0202' / ADD 2
04AA' 40B0          X'40B0' / RSHFT 12
04AC' 5400          X'5400' / STA 4
04AE' 2100          X'2100' / LDA 1   / AIRCRAFT POSITION FOR LINEPRINTER
04B0' 4080          X'4080' / RSHFT 9
04B2' 5100          X'5100' / STA 1
04B4' 2200          X'2200' / LDA 2
04B6' 4080          X'4080' / RSHFT 9
04B8' 5200          X'5200' / STA 2
04BA' 8033          X'8033' / SEEK 8,0,3,0000   / TEST FOR END OF DATA STREAM
04BC' 0000          X'0000'
04BE' 1100          X'1100' / RESET 1   / OUTPUT - STAGE 1
04C0' 2571          X'2571' / WRITE 6,8
04C2' 0008          X'0008' / JZSR AP1
04C4' 0492'         ‡AP1
04C6' 0009          X'0009' / SPRTN
04C8' 1100     AP2: X'1100' / RESET 1   / START OF INPUT - STAGE 2
04CA' 1371          X'1371' / READ 4,8
04CC' 1000          X'1000' / RESET 0   / TRANSPOSE COLUMN 1 DATA
04CE' 2000          X'2000' / LDA 0
04D0' 5800          X'5800' / STA 8
04D2' 2100          X'2100' / LDA 1
04D4' 5900          X'5900' / STA 9
04D6' 2200          X'2200' / LDA 2
04D8' 5A00          X'5A00' / STA 10
04DA' 2300          X'2300' / LDA 3
04DC' 5B00          X'5B00' / STA 11
04DE' 000A          X'000A' / CALL
```

```
04E0' 0538'           ‡ASSOC / ASSOC
04E2' 000A            X'000A' / CALL
04E4' 0526'           ‡TRANSP / TRANSP
04E6' 000A            X'000A' / CALL
04E8' 0538'           ‡ASSOC / ASSOC
04EA' 000A            X'000A' / CALL
04EC' 0526'           ‡TRANSP / TRANSP
04EE' 000A            X'000A' / CALL
04F0' 0538'           ‡ASSOC / ASSOC
04F2' 000A            X'000A' / CALL
04F4' 0526'           ‡TRANSP / TRANSP
04F6' 000A            X'000A' / CALL
04F8' 0538'           ‡ASSOC / ASSOC
04FA' 000A            X'000A' / CALL
04FC' 0526'           ‡TRANSP / TRANSP
04FE' 000A            X'000A' / CALL
0500' 0538'           ‡ASSOC / ASSOC
0502' 000A            X'000A' / CALL
0504' 0526'           ‡TRANSP / TRANSP
0506' 000A            X'000A' / CALL
0508' 0538'           ‡ASSOC / ASSOC
050A' 000A            X'000A' / CALL
050C' 0526'           ‡TRANSP / TRANSP
050E' 000A            X'000A' / CALL
0510' 0538'           ‡ASSOC / ASSOC
0512' 000A            X'000A' / CALL
0514' 0526'           ‡TRANSP / TRANSP
0516' 7400            X'7400' / STZ 4
0518' 7500            X'7500' / STZ 5
051A' 7600            X'7600' / STZ 6
051C' 7700            X'7700' / STZ 7
051E' 000A            X'000A' / CALL
0520' 054E'           ‡ASSC1 / ASSOC1
0522' 0009            X'0009' / SPRTN
0524' 0000            X'0000' / STOP
0526' 2400    TRANSP: X'2400' / LDA 4    / ROUTINE TRANSP
0528' 5800            X'5800' / STA 8
052A' 2500            X'2500' / LDA 5
052C' 5900            X'5900' / STA 9
052E' 2600            X'2600' / LDA 6
0530' 5A00            X'5A00' / STA 10
0532' 2700            X'2700' / LDA 7
0534' 5B00            X'5B00' / STA 11
0536' 000B            X'000B' / RETURN
0538' 1100     ASSOC: X'1100' / RESET 1    / ROUTINE ASSOC
053A' 1371            X'1371' / READ 4,8   / INPUT OF NEXT COLUMN
053C' 1000            X'1000' / RESET 0
053E' 2000            X'2000' / LDA 0
0540' 5400            X'5400' / STA 4
0542' 2100            X'2100' / LDA 1
0544' 5500            X'5500' / STA 5
0546' 2200            X'2200' / LDA 2
0548' 5600            X'5600' / STA 6
054A' 2300            X'2300' / LDA 3
054C' 5700            X'5700' / STA 7
054E' 7200     ASSC1: X'7200' / STZ 2    / ROUTINE ASSC1 STARTS HERE
0550' 2800            X'2800' / LDA 8    / SAME BOX TEST
0552' 5000            X'5000' / STA 0
0554' 7800            X'7800' / STZ 8
0556' 000A            X'000A' / CALL
0558' 061C'           ‡TEST / TEST
```

```
055A' 2000     X'2000' / LDA 0     / REPLACE AIRCRAFT ID.NO.
055C' 5800     X'5800' / STA 8
055E' 2400     X'2400' / LDA 4     / RIGHT BOX TEST
0560' 5000     X'5000' / STA 0
0562' 000A     X'000A' / CALL
0564' 061C'    ‡TEST / TEST
0566' 2500     X'2500' / LDA 5
0568' 5000     X'5000' / STA 0
056A' 000A     X'000A' / CALL
056C' 061C'    ‡TEST / TEST
056E' 2600     X'2600' / LDA 6
0570' 5000     X'5000' / STA 0
0572' 000A     X'000A' / CALL
0574' 061C'    ‡TEST / TEST
0576' 2700     X'2700' / LDA 7
0578' 5000     X'5000' / STA 0
057A' 000A     X'000A' / CALL
057C' 061C'    ‡TEST / TEST
057E' 8063     X'8063' / SEEK 8,0,6,FFFF   / TOP BOX TEST
0580' FFFF     X'FFFF'
0582' 2800     X'2800' / LDA 8
0584' 000A     X'000A' / CALL
0586' 0610'    ‡TRNS / TRNS
0588' 5000     X'5000' / STA 0
058A' 000A     X'000A' / CALL
058C' 061C'    ‡TEST / TEST
058E' 000A     X'000A' / CALL
0590' 0642'    ‡SPOT / SPOT
0592' 8063     X'8063' / SEEK 8,0,6,FFFF
0594' FFFF     X'FFFF'
0596' 2900     X'2900' / LDA 9
0598' 000A     X'000A' / CALL
059A' 0610'    ‡TRNS / TRNS
059C' 5000     X'5000' / STA 0
059E' 000A     X'000A' / CALL
05A0' 061C'    ‡TEST / TEST
05A2' 000A     X'000A' / CALL
05A4' 0642'    ‡SPOT / SPOT
05A6' 8063     X'8063' / SEEK 8,0,6,FFFF
05A8' FFFF     X'FFFF'
05AA' 2A00     X'2A00' / LDA 10
05AC' 000A     X'000A' / CALL
05AE' 0610'    ‡TRNS / TRNS
05B0' 5000     X'5000' / STA 0
05B2' 000A     X'000A' / CALL
05B4' 061C'    ‡TEST / TEST
05B6' 000A     X'000A' / CALL
05B8' 0642'    ‡SPOT / SPOT
05BA' 8063     X'8063' / SEEK 8,0,6,FFFF
05BC' FFFF     X'FFFF'
05BE' 2B00     X'2B00' / LDA 11
05C0' 000A     X'000A' / CALL
05C2' 0610'    ‡TRNS / TRNS
05C4' 5000     X'5000' / STA 0
05C6' 000A     X'000A' / CALL
05C8' 061C'    ‡TEST / TEST
05CA' 000A     X'000A' / CALL
05CC' 0642'    ‡SPOT / SPOT
05CE' 8063     X'8063' / SEEK 8,0,6,FFFF   / TOP-RIGHT BOX TEST
05D0' FFFF     X'FFFF'
05D2' 2400     X'2400' / LDA 4
```

```
05D4' 000A            X'000A' / CALL
05D6' 0610'           ÷TRNS / TRNS
05D8' 5000            X'5000' / STA 0
05DA' 000A            X'000A' / CALL
05DC' 061C'           ÷TEST / TEST
05DE' 8063            X'8063' / SEEK 8,0,6,FFFF
05E0' FFFF            X'FFFF'
05E2' 2500            X'2500' / LDA 5
05E4' 000A            X'000A' / CALL
05E6' 0610'           ÷TRNS / TRNS
05E8' 5000            X'5000' / STA 0
05EA' 000A            X'000A' / CALL
05EC' 061C'           ÷TEST / TEST
05EE' 8063            X'8063' / SEEK 8,0,6,FFFF
05F0' FFFF            X'FFFF'
05F2' 2600            X'2600' / LDA 6
05F4' 000A            X'000A' / CALL
05F6' 0610'           ÷TRNS / TRNS
05F8' 5000            X'5000' / STA 0
05FA' 000A            X'000A' / CALL
05FC' 061C'           ÷TEST / TEST
05FE' 8063            X'8063' / SEEK 8,0,6,FFFF
0600' FFFF            X'FFFF'
0602' 2700            X'2700' / LDA 7
0604' 000A            X'000A' / CALL
0606' 0610'           ÷TRNS / TRNS
0608' 5000            X'5000' / STA 0
060A' 000A            X'000A' / CALL
060C' 061C'           ÷TEST / TEST
060E' 000B            X'000B' / RETURN
0610' 9000      TRNS: X'9000' / TRANS    / ROUTINE TRNS
0612' A000            X'A000' / RECV
0614' 8200            X'8200' / RESTS 2
0616' B000            X'B000' / ZAC
0618' 1000            X'1000' / RESET 0
061A' 000B            X'000B' / RETURN
061C' 6043      TEST: X'6043' / SEEK 6,0,4,0000   / ROUTINE TEST
061E' 0000            X'0000'
0620' 2800            X'2800' / LDA 8
0622' 5100            X'5100' / STA 1
0624' 000A            X'000A' / CALL
0626' 0664'           ÷T / T
0628' 2900            X'2900' / LDA 9
062A' 5100            X'5100' / STA 1
062C' 000A            X'000A' / CALL
062E' 0664'           ÷T / T
0630' 2A00            X'2A00' / LDA 10
0632' 5100            X'5100' / STA 1
0634' 000A            X'000A' / CALL
0636' 0664'           ÷T / T
0638' 2B00            X'2B00' / LDA 11
063A' 5100            X'5100' / STA 1
063C' 000A            X'000A' / CALL
063E' 0664'           ÷T / T
0640' 000B            X'000B' / RETURN
0642' 2400      SPOT: X'2400' / LDA 4    / ROUTINE SPOT
0644' 5100            X'5100' / STA 1
0646' 000A            X'000A' / CALL
0648' 0664'           ÷T / T
064A' 2500            X'2500' / LDA 5
064C' 5100            X'5100' / STA 1
```

```
064E' 000A              X'000A' / CALL
0650' 0664'             ≑T / T
0652' 2600              X'2600' / LDA 6
0654' 5100              X'5100' / STA 1
0656' 000A              X'000A' / CALL
0658' 0664'             ≑T / T
065A' 2700              X'2700' / LDA 7
065C' 5100              X'5100' / STA 1
065E' 000A              X'000A' / CALL
0660' 0664'             ≑T / T
0662' 000B              X'000B' / RETURN
0664' 6153          T: X'6153' / SEEK 6,1,5,0000   / ROUTINE T
0666' 0000              X'0000'
0668' 1100              X'1100' / RESET 1   / OUTPUT PAIRS OF AIRCRAFT
066A' 2271              X'2271' / WRITE 3,8
066C' 1000              X'1000' / RESET 0
066E' 000B              X'000B' / RETURN
0670' 1100          AP3: X'1100' / RESET 1   / INPUT - STAGE 3
0672' 1A71              X'1A71' / READ 11,8
0674' 1000              X'1000' / RESET 0
0676' 7D00              X'7D00' / STZ 13
0678' 2300              X'2300' / LDA 3
067A' 0402              X'0402' / ADD 4
067C' 3030              X'3030' / LSHFT 4
067E' 5B00              X'5B00' / STA 11
0680' C200              X'C200' / TEST ER,2
0682' 1B12              X'1B12' / SUBT 11,2
0684' 1000              X'1000' / RESET 0
0686' 0A02              X'0A02' / ADD 10
0688' 5B00              X'5B00' / STA 11
068A' 2800              X'2800' / LDA 8
068C' 0902              X'0902' / ADD 9
068E' 3030              X'3030' / LSHFT 4
0690' 5C00              X'5C00' / STA 12
0692' C200              X'C200' / TEST ER,2
0694' 1C12              X'1C12' / SUBT 12,2
0696' 1000              X'1000' / RESET 0
0698' 0A02              X'0A02' / ADD 10
069A' 5C00              X'5C00' / STA 12
069C' 2100              X'2100' / LDA 1
069E' 1602              X'1602' / SUBT 6
06A0' CC00              X'CC00' / TEST ER,12
06A2' 2300              X'2300' / LDA 3
06A4' C200              X'C200' / TEST ER,2
06A6' 2B00              X'2B00' / LDA 11
06A8' 5D00              X'5D00' / STA 13
06AA' 1000              X'1000' / RESET 0
06AC' 2100              X'2100' / LDA 1
06AE' 1602              X'1602' / SUBT 6
06B0' CC00              X'CC00' / TEST ER,12
06B2' 2800              X'2800' / LDA 8
06B4' CC00              X'CC00' / TEST ER,12
06B6' 2D00              X'2D00' / LDA 13
06B8' 0C02              X'0C02' / ADD 12
06BA' 5D00              X'5D00' / STA 13
06BC' 1000              X'1000' / RESET 0
06BE' 2100              X'2100' / LDA 1
06C0' 1602              X'1602' / SUBT 6
06C2' C200              X'C200' / TEST ER,2
06C4' 2300              X'2300' / LDA 3
06C6' CC00              X'CC00' / TEST ER,12
```

```
06C8' 2B00          X'2B00' / LDA 11
06CA' 5D00          X'5D00' / STA 13
06CC' 1000          X'1000' / RESET 0
06CE' 2100          X'2100' / LDA 1
06D0' 1602          X'1602' / SUBT 6
06D2' C200          X'C200' / TEST ER,2
06D4' 2800          X'2800' / LDA 8
06D6' C200          X'C200' / TEST ER,2
06D8' 2D00          X'2D00' / LDA 13
06DA' 0C02          X'0C02' / ADD 12
06DC' 5D00          X'5D00' / STA 13
06DE' 1000          X'1000' / RESET 0
06E0' 2D00          X'2D00' / LDA 13
06E2' 5C00          X'5C00' / STA 12
06E4' B000          X'B000' / ZAC
06E6' 2CF2          X'2CF2' / MULT 12
06E8' 5A00          X'5A00' / STA 10
06EA' 2F00          X'2F00' / LDA 15
06EC' 5B00          X'5B00' / STA 11
06EE' 2100          X'2100' / LDA 1
06F0' 1602          X'1602' / SUBT 6
06F2' 5800          X'5800' / STA 8
06F4' C200          X'C200' / TEST ER,2
06F6' 1812          X'1812' / SUBT 8,2
06F8' 5800          X'5800' / STA 8
06FA' 1000          X'1000' / RESET 0
06FC' 5900          X'5900' / STA 9
06FE' B000          X'B000' / ZAC
0700' 28F2          X'28F2' / MULT 8
0702' 5800          X'5800' / STA 8
0704' 2F00          X'2F00' / LDA 15
0706' 5900          X'5900' / STA 9
0708' 2200          X'2200' / LDA 2
070A' 1702          X'1702' / SUBT 7
070C' 5600          X'5600' / STA 6
070E' C200          X'C200' / TEST ER,2
0710' 1612          X'1612' / SUBT 6,2
0712' 5600          X'5600' / STA 6
0714' 1000          X'1000' / RESET 0
0716' 5700          X'5700' / STA 7
0718' B000          X'B000' / ZAC
071A' 26F2          X'26F2' / MULT 6
071C' 5600          X'5600' / STA 6
071E' 2F00          X'2F00' / LDA 15
0720' 5700          X'5700' / STA 7
0722' 2900          X'2900' / LDA 9
0724' 0702          X'0702' / ADD 7
0726' 5400          X'5400' / STA 4
0728' 2800          X'2800' / LDA 8
072A' 3602          X'3602' / ADDD 6
072C' 5300          X'5300' / STA 3
072E' 2400          X'2400' / LDA 4
0730' 1B02          X'1B02' / SUBT 11
0732' 5400          X'5400' / STA 4
0734' 2300          X'2300' / LDA 3
0736' 4A02          X'4A02' / SUBTD 10
0738' 5300          X'5300' / STA 3
073A' 6000          X'6000' / RSTVR
073C' 8033          X'8033' / SEEK 8,0,3,0000   / TEST FOR END OF DATA STREAM
073E' 0000          X'0000'
0740' 2500          X'2500' / LDA 5
```

```
0742' 5100              X'5100' / STA 1
0744' 2300              X'2300' / LDA 3
0746' C800              X'C800' / TEST ER,8
0748' 2400              X'2400' / LDA 4
074A' 3000              X'3000' / LSHFT 1
074C' 4000              X'4000' / RSHFT 1
074E' 1000              X'1000' / RESET 0
0750' 5F00              X'5F00' / STA 15
0752' DA00              X'DA00' / TEST SR,10
0754' 6053              X'6053' / SEEK 6,0,5,0000
0756' 0000              X'0000'
0758' 1100              X'1100' / RESET 1
075A' 2271              X'2271' / WRITE 3,8
075C' 0008              X'0008' / JSRZ AP3
075E' 0670'             ‡AP3
0760' 0009              X'0009' / SPRTN
0762' 1100         AP4: X'1100' / RESET 1   / INPUT - STAGE 4
0764' 6000              X'6000' / RSTVR
0766' 1471              X'1471' / READ 5,8
0768' 1000              X'1000' / RESET 0
076A' 2300              X'2300' / LDA 3
076C' 3050              X'3050' / LSHFT 6
076E' 0102              X'0102' / ADD 1
0770' 3000              X'3000' / LSHFT 1
0772' 4000              X'4000' / RSHFT 1
0774' 5100              X'5100' / STA 1
0776' 2400              X'2400' / LDA 4
0778' 3050              X'3050' / LSHFT 6
077A' 0202              X'0202' / ADD 2
077C' 3000              X'3000' / LSHFT 1
077E' 4000              X'4000' / RSHFT 1
0780' 5200              X'5200' / STA 2
0782' 8033'             X'8033' / SEEK 8,0,3,0000
0784' 0000              X'0000'
0786' 1100              X'1100' / RESET 1   / OUTPUT - STAGE 4
0788' 2571              X'2571' / WRITE 6,8
078A' 0008              X'0008' / JZSR AP4
078C' 0762'             ‡AP4
078E' 0009              X'0009' / SPRTN
                   £DEF *LIST=15
                   /
                   / THE MAIN SP PROGRAM STARTS HERE
                   /
0790' 2411         START: DEVICE=CONSL / RESET CONSOLE DISPLAY
0792' DE10 008A'          OC DEVICE,INC
0796' 2420                IODATA=X'0000'
0798' 41F0 0E0A'          CALL CONOUT
079C' 41F0 0E0A'          CALL CONOUT
07A0' 2420                DTBFCT=0
07A2' 4020 0EA2'
07A6' 4020 0EA8'          OTBFCT=0
07AA' C810 0098           DEVICE=LPLK
07AE' DE10 0084'          OC DEVICE,LINK
07B2' C810 0099           DEVICE=TRANS
07B6' DE10 0084'          OC DEVICE,LINK
07BA' C810 0089           DEVICE=RECV
07BE' DE10 0084'          OC DEVICE,LINK
07C2' C810 0088           DEVICE=X'0088'
07C6' DE10 0084'          OC DEVICE,LINK
07CA' 2412                DEVICE=TT
07CC' 2420                FLAG=0
```

```
07CE' 4020 0EA6'
07D2' DE10 0082'             OC DEVICE,READ
07D6' C200 008E'             LPSW OLDPSW / ENABLE INTERRUPTS
07DA' C820 08BC' ST1:        LPT=#LPTST
07DE' 4020 0EAA'
07E2' 242D                   IODATA=X'0D'
07E4' 41F0 0DC0'             CALL TTOUT
07E8' 242A                   IODATA=X'0A'
07EA' 41F0 0DC0'             CALL TTOUT
07EE' 245F                   STATUS=X'000F' / INITIAL AP SETUP
07F0' 0825                   IODATA=STATUS / TRANSMIT INITIAL STATUS
07F2' 41F0 0DF6'             CALL LKOUT
07F6' 2420                   LPFLAG=0   / SET LPFLAG FOR NO OUTPUT YET
07F8' 4020 0EA4'
                  /
                  /
                  /
07FC' C870 0EAE' STAGE1: OSTRM=#AIR1-#DATA
0800' CB70 0E52'
0804' 48C0 0E58'          ISTRM=AIR2-#DATA
0808' CBC0 0E52'
080C' C860 0492'          PSTRM=#AP1
0810' 41D0 0BF4'          CALL2 INSTR / COMMENCE AP EXECUTION FOR STAGE 1
                  /
                  / SET UP AIR SPACE DISPLAY FOR LINEPRINTER
                  /
0814' C810 0098   DEVICE=LPLK
0818' 9D13        A: SS DEVICE,WORKSP
081A' 20F1           JUMP A IF 15
081C' DE10 0084'     OC DEVICE,LINK   / STOP LINEPRINTER
0820' 24B0        I=0
0822' 48A0 0E54'  ACC1=LP
0826' 2420        LPZERO: (ACC1)=0
0828' 402A 0000
082C' 26A2               ACC1=ACC1+2
082E' 26B2               I=I+2
0830' C9B0 0200          JUMP LPZERO IF I#512
0834' 2037
0836' 48C0 0E58'         ISTRM=AIR2-#DATA
083A' CBC0 0E52'
083E' 480C 0E56' LPSET1: ACC=DATA(ISTRM+4)
0842' C900 0040          JUMP LPSET4 IF ACC)=64
0846' 4310 0886'
084A' 9102               ACC=ACC<<2
084C' 48AC 0E54'         ACC1=DATA(ISTRM+2)
0850' C9A0 0040          JUMP LPSET4 IF ACC1)=64
0854' 4310 0886'
0858' 90A4               ACC1=ACC1>>4
085A' 060A               ACC=ACC!ACC1
085C' 48AC 0E54'         ACC1=DATA(ISTRM+2)&X'000F'
0860' C4A0 000F
0864' C880 8000          TST=X'8000'
0868' 08AA       LPSET2: JUMPS LPSET3 IF ACC1=0
086A' 2334
086C' 9081               TST=TST>>1
086E' 27A1               ACC1=ACC1-1
0870' 2204               JUMP LPSET2
0872' 08B0       LPSET3: I=ACC<<1
0874' 91B1
0876' 48A0 0E54'         ACC1=LP+I
087A' 0AAB
```

```
087C' 482A 0000          (ACC1)=(ACC1)!TST
0880' 0628
0882' 402A 0000
0886' 26CA      LPSET4: ISTRM=ISTRM+10
0888' 482C 0E52'         JUMP LPSET1 IF DATA(ISTRM)≠0
088C' 4230 083E'
0890' C810 0098 DEVICE=LPLK
0894' 9D13      B: SS DEVICE,WORKSP
0896' 20F1         JUMP B IF 15
0898' DE10 0086'   OC DEVICE,LINKON   / START LINEPRINTER
089C' 4820 0EA4'   JUMP PUT IF LPFLAG=1   / JUMP IF LINEPRINTER ALREADY ON
08A0' C920 0001
08A4' 4330 0994'
08A8' 2421      LPFLAG=1   / SET LPFLAG FOR LINEPRINTER ON
08AA' 4020 0EA4'
08AE' C820 0024 IODATA='£'
08B2' 9D13      C: SS DEVICE,WORKSP
08B4' 20F1         JUMP C IF 15
08B6' 9A12         WD DEVICE,IODATA
08B8' 430C 0994' JUMP PUT
                /
                / LINEPRINTER SERVICE ROUTINE
                /
08BC' C8B0 0043 LPTST: I=67
08C0' C820 003D LPT0: IODATA='='
08C4' 41F0 0D9E'      CALL LPOUT
08C8' 27B1            I=I-1
08CA' 2035            JUMP LPT0 IF I≠0
08CC' 242D      IODATA=X'0D'
08CE' 41F0 0D9E' CALL LPOUT
08D2' 242A      IODATA=X'0A'
08D4' 41F0 0D9E' CALL LPOUT
08D8' C820 0021 IODATA='!'
08DC' 41F0 0D9E' CALL LPOUT
08E0' C820 0020 IODATA=' '
08E4' 41F0 0D9E' CALL LPOUT
08E8' C820 0021 IODATA='!'
08EC' 41F0 0D9E' CALL LPOUT
08F0' 2480      TST=0
08F2' 24B0      I=0
08F4' C820 0020 LPT1: IODATA=' '
08F8' 2681            TST=TST+1
08FA' C980 0008       JUMPS LPT2 IF TST≠8
08FE' 2135
0900' 2480            TST=0
0902' 26B1            I=I+1
0904' C820 0021       IODATA='!'
0908' 41F0 0D9E' LPT2: CALL LPOUT
090C' C9B0 0008       JUMP LPT1 IF I≠8
0910' 203E
0912' 242D      IODATA=X'0D'
0914' 41F0 0D9E' CALL LPOUT
0918' 242A      IODATA=X'0A'
091A' 41F0 0D9E' CALL LPOUT
091E' 2480      TST=0
0920' 2447      STREAM=7
0922' 2641      LPT3: STREAM=STREAM+1
0924' 2490            TST1=0
0926' C820 0021       IODATA='!'
092A' 41F0 0D9E'      CALL LPOUT
092E' C820 0020       IODATA=' '
```

```
0932' C940 0008          JUMPS LPT4 IF STREAM‡8
0936' 2134
0938' 2440               STREAM=0
093A' C820 003D          IODATA='='
093E' 41F0 0D9E' LPT4: CALL LPOUT
0942' 08B8       LPT5: I=TST+TST1
0944' 0AB9
0946' 4AB0 0E54'         I=I+LP
094A' 480B 0000          ACC=(I)
094E' C8B0 0010          I=16
0952' C820 002A  LPT6: IODATA='*'
0956' 0800               JUMPS LPT7 IF ACC<0
0958' 2113
095A' C820 0020          IODATA=' '
095E' 41F0 0D9E' LPT7: CALL LPOUT
0962' 9101               ACC=ACC<<1
0964' 27B1               I=I-1
0966' 203A               JUMP LPT6 IF I‡0
0968' 2692               TST1=TST1+2
096A' C990 0008          JUMP LPT5 IF TST1‡8
096E' 4230 0942'
0972' C820 0021          IODATA='!'
0976' 41F0 0D9E'         CALL LPOUT
097A' 242D               IODATA=X'0D'
097C' 41F0 0D9E'         CALL LPOUT
0980' 242A               IODATA=X'0A'
0982' 41F0 0D9E'         CALL LPOUT
0986' 2688               TST=TST+8
0988' C980 0200          JUMP LPT3 IF TST‡512
098C' 4230 0922'
0990' 4300 08BC' JUMP LPTST
                 /
                 /
                 /
0994' 24B0       PUT: I=0
0996' 48A0 0E5A'      ACC1=AIR3
099A' 2420       PUT0: (ACC1)=0
099C' 402A 0000
09A0' 26A2               ACC1=ACC1+2
09A2' 26B2               I=I+2
09A4' C9B0 0200          JUMP PUT0 IF I‡512
09A8' 2037
09AA' 48C0 0E58'         ISTRM=AIR2-‡DATA
09AE' CBC0 0E52'
09B2' 480C 0E58' PUT1: ACC=DATA(ISTRM+6)
09B6' C900 0008          JUMP PUT3 IF ACC>=8
09BA' 4310 0A0A'
09BE' 0800               JUMP PUT3 IF ACC<0
09C0' 4210 0A0A'
09C4' 9103               ACC=ACC<<3
09C6' 48AC 0E5A'         ACC1=DATA(ISTRM+8)
09CA' C9A0 0008          JUMP PUT3 IF ACC1>=8
09CE' 4310 0A0A'
09D2' 08AA               JUMP PUT3 IF ACC1<0
09D4' 4210 0A0A'
09D8' 0AA0               ACC1=ACC1+ACC
09DA' 91A3               ACC1=ACC1<<3
09DC' 4880 0E5A'         TST=AIR3+ACC1
09E0' 0A8A
09E2' 4828 0000          JUMPS PUT2 IF (TST)=0
09E6' 233E
```

```
09E8' 2682                 TST=TST+2
09EA' 4828 0000            JUMPS PUT2 IF (TST)=0
09EE' 233A
09F0' 2682                 TST=TST+2
09F2' 4828 0000            JUMPS PUT2 IF (TST)=0
09F6' 2336
09F8' 2682                 TST=TST+2
09FA' 4828 0000            JUMPS PUT2 IF (TST)=0
09FE' 2332
0A00' 2200           STOP1: JUMP STOP1
0A02' 482C 0E52' PUT2: (TST)=DATA(ISTRM)
0A06' 4028 0000
0A0A' 26CA           PUT3: ISTRM=ISTRM+10
0A0C' 482C 0E52'           JUMP PUT1 IF DATA(ISTRM)≠0
0A10' 4230 09B2'
0A14' 4870 0E5A' OSTRM=AIR3-≠DATA
0A18' CB70 0E52'
0A1C' 48C0 0E5C' ISTRM=AIR4-≠DATA
0A20' CBC0 0E52'
0A24' 41D0 0BF4' CALL2 INSTR / COMMENCE AP EXECUTION FOR STAGE 2
0A28' 2420           DATA(ISTRM)=X'0000'
0A2A' 402C 0E52'
0A2E' 48C0 0E5C' ISTRM=AIR4-≠DATA
0A32' CBC0 0E52'
0A36' 4840 0E5E' STREAM=AIR5-≠DATA
0A3A' CB40 0E52'
0A3E' 2495           TST1=5
0A40' 24B0           PROC0: I=0
0A42' 48AC 0E52' PROC1: ACC1=DATA(ISTRM)
0A46' 26C2                  ISTRM=ISTRM+2
0A48' 08AA                  JUMP PROC2 IF ACC1=0
0A4A' 4330 0AB8'
0A4E' 40A4 0E52'            DATA(STREAM)=ACC1
0A52' 2642                  STREAM=STREAM+2
0A54' C83A FFFF             WORKSP=ACC1-1
0A58' 0C29                  MH IODATA,TST1
0A5A' 9131                  WORKSP=WORKSP<<1
0A5C' C880 0EAE'            TST=≠AIR1-≠DATA
0A60' CB80 0E52'
0A64' 0A83                  TST=TST+WORKSP
0A66' 2682                  TST=TST+2
0A68' 4828 0E52'            DATA(STREAM)=DATA(TST)
0A6C' 4024 0E52'
0A70' 2642                  STREAM=STREAM+2
0A72' 2682                  TST=TST+2
0A74' 4828 0E52'            DATA(STREAM)=DATA(TST)
0A78' 4024 0E52'
0A7C' 2642                  STREAM=STREAM+2
0A7E' 2682                  TST=TST+2
0A80' 4828 0E52'            DATA(STREAM)=DATA(TST)
0A84' 4024 0E52'
0A88' 2642                  STREAM=STREAM+2
0A8A' 2682                  TST=TST+2
0A8C' 4828 0E52'            DATA(STREAM)=DATA(TST)
0A90' 4024 0E52'
0A94' 2642                  STREAM=STREAM+2
0A96' 26B1                  I=I+1
0A98' 081B                  DEVICE=I&X'0001'
0A9A' C410 0001
0A9E' 4230 0A42'            JUMP PROC1 IF DEVICE≠0
0AA2' 4820 0EAC'            DATA(STREAM)=VAR
```

```
0AA6' 4024 0E52'
0AAA' 2642                  STREAM=STREAM+2
0AAC' C9B0 0010             JUMP PROC1 IF I≠16
0AB0' 4230 0A42'
0AB4' 4300 0A40'            JUMP PROC0
0AB8' 2480          PROC2:  TST=0
0ABA' 2420          PROC3:  DATA(STREAM)=0
0ABC' 4024 0E52'
0AC0' 2642                  STREAM=STREAM+2
0AC2' 2681                  TST=TST+1
0AC4' C980 000B             JUMP PROC3 IF TST≠11
0AC8' 2037
0ACA' 26B2                  I=I+2
0ACC' C9B0 0010             JUMP PROC2 IF I≠16
0AD0' 203C
0AD2' 4870 0E5E' PROC4:     OSTRM=AIR5-≠DATA
0AD6' CB70 0E52'
0ADA' 48C0 0E60'            ISTRM=AIR6-≠DATA
0ADE' CBC0 0E52'
0AE2' C86C 0670'            PSTRM=≠AP3
0AE6' 41D0 0BF4'            CALL2 INSTR / COMMENCE AP EXECUTION FOR STAGE 3
0AEA' 2420                  DATA(ISTRM)=X'0000'
0AEC' 402C 0E52'
0AF0' 4840 0E60'            STREAM=AIR6-≠DATA
0AF4' CB40 0E52'
0AF8' 48A4 0E52' TTY:  ACC1=DATA(STREAM)
0AFC' 087A             OSTRM=ACC1
0AFE' 2642             STREAM=STREAM+2
0B00' 08AA             JUMP ONWD IF ACC1=0
0B02' 4330 0BD6'
0B06' 41E0 0E12'       CALL1 PRINT
0B0A' C820 0020        IODATA=X'20'
0B0E' 41F0 0DC0'       CALL TTOUT
0B12' C820 0020        IODATA=X'20'
0B16' 41F0 0DC0'       CALL TTOUT
0B1A' 2495             TST1=5
0B1C' C837 FFFF        WORKSP=OSTRM-1
0B20' 0C29             MH IODATA,TST1
0B22' 9131             WORKSP=WORKSP<<1
0B24' 48C0 0E58'       ISTRM=AIR2-≠DATA
0B28' CBC0 0E52'
0B2C' 0AC3             ISTRM=ISTRM+WORKSP
0B2E' 26C2             ISTRM=ISTRM+2
0B30' 48AC 0E52'       ACC1=DATA(ISTRM)
0B34' 41E0 0E12'       CALL1 PRINT
0B38' C820 0020        IODATA=X'20'
0B3C' 41F0 0DC0'       CALL TTOUT
0B40' C820 0020        IODATA=X'20'
0B44' 41F0 0DC0'       CALL TTOUT
0B48' 26C2             ISTRM=ISTRM+2
0B4A' 48AC 0E52'       ACC1=DATA(ISTRM)
0B4E' 41E0 0E12'       CALL1 PRINT
0B52' C820 0020        IODATA=X'20'
0B56' 41F0 0DC0'       CALL TTOUT
0B5A' C820 0020        IODATA=X'20'
0B5E' 41F0 0DC0'       CALL TTOUT
0B62' C820 0020        IODATA=X'20'
0B66' 41F0 0DC0'       CALL TTOUT
0B6A' C820 0020        IODATA=X'20'
0B6E' 41F0 0DC0'       CALL TTOUT
0B72' 48A4 0E52'       ACC1=DATA(STREAM)
```

```
0B76' 087A                OSTRM=ACC1
0B78' 2642                STREAM=STREAM+2
0B7A' 41E0 0E12'          CALL1 PRINT
0B7E' C820 0020           IODATA=X'20'
0B82' 41F0 0DC0'          CALL TTOUT
0B86' C820 0020           IODATA=X'20'
0B8A' 41F0 0DC0'          CALL TTOUT
0B8E' 2495                TST1=5
0B90' C837 FFFF           WORKSP=OSTRM-1
0B94' 0C29                MH IODATA,TST1
0B96' 9131                WORKSP=WORKSP<<1
0B98' 48C0 0E58'          ISTRM=AIR2-÷DATA
0B9C' CBC0 0E52'
0BA0' 0AC3                ISTRM=ISTRM+WORKSP
0BA2' 26C2                ISTRM=ISTRM+2
0BA4' 48AC 0E52'          ACC1=DATA(ISTRM)
0BA8' 41E0 0E12'          CALL1 PRINT
0BAC' C820 0020           IODATA=X'20'
0BB0' 41F0 0DC0'          CALL TTOUT
0BB4' C820 0020           IODATA=X'20'
0BB8' 41F0 0DC0'          CALL TTOUT
0BBC' 26C2                ISTRM=ISTRM+2
0BBE' 48AC 0E52'          ACC1=DATA(ISTRM)
0BC2' 41E0 0E12'          CALL1 PRINT
0BC6' 242D                IODATA=X'0D'
0BC8' 41F0 0DC0'          CALL TTOUT
0BCC' 242A                IODATA=X'0A'
0BCE' 41F0 0DC0'          CALL TTOUT
0BD2' 4300 0AF8'          JUMP TTY
                   /
                   /
                   /
0BD6' 242D         ONWD:  IODATA=X'0D'
0BD8' 41F0 0DC0'          CALL TTOUT
0BDC' 242A                IODATA=X'0A'
0BDE' 41F0 0DC0'          CALL TTOUT
0BE2' C870 0EAE'          OSTRM=÷AIR1-÷DATA
0BE6' CB70 0E52'
0BEA' 08C7                ISTRM=OSTRM
0BEC' 41D0 0BF4'          CALL2 INSTR / COMMENCE AP EXECUTION FOR STAGE 4
0BF0' 4300 07FC'          JUMP STAGE1
                   /
                   /
                   /
                   /
                   / THE MAIN AP SUPPORT PROGRAM COMMENCES HERE
                   /
                   /
                   / TRANSMIT INSTRUCTION
                   /
0BF4' 4806 0000    INSTR: ACC=(PSTRM)
0BF8' 0880                TST=ACC
0BFA' 0825                IODATA=STATUS / DISPLAY INSTRUCTION AND STATUS
0BFC' 41F0 0E0A'          CALL CONOUT
0C00' 0820                IODATA=ACC
0C02' 41F0 0E0A'          CALL CONOUT
0C06' 0890                TST1=ACC&X'0008' / TEST IF SP INSTRUCTION
0C08' C490 0008
0C0C' 4230 0D08'          JUMP APSP IF TST1≠0
0C10' 0820                IODATA=ACC
0C12' 41F0 0DF6'          CALL LKOUT
```

```
0C16' 0800               JUMP END IF ACC=0
0C18' 4330 0D78'
                    /
                    / CHECK CONTROL LEVEL
                    /
0C1C' C400 000F     ACC=ACC&X'000F'
0C20' 0830          WORKSP=ACC\STATUS
0C22' 0735
0C24' C430 000F     WORKSP=WORKSP&X'000F'
0C28' 4330 0C54'    JUMP NOLOAD IF WORKSP=0
                    /
                    / LOAD FUNCTIONAL MEMORIES
                    /
0C2C' 0735          LOAD: WORKSP=WORKSP\STATUS / LOADS STATUS WITH CONTROL LEVEL
0C2E' 0853                STATUS=WORKSP
0C30' 41E0 0DB0'          CALL1 DSTAT
0C34' 0840                STREAM=ACC / CALCULATE DISPLACEMENT
0C36' 9148                STREAM=STREAM<<8
0C38' 2400                ACC=0
0C3A' 4820 008C'          IODATA=CONTROL / OUTPUT CONTROL WORD
0C3E' 41F0 0DF6'          CALL LKOUT
0C42' 4824 0092'    LOOPA: IODATA=FM(STREAM) / OUTPUT FUNCTIONAL MEMORY DATA
0C46' 41F0 0DF6'           CALL LKOUT
0C4A' 2601                 ACC=ACC+1
0C4C' 2642                 STREAM=STREAM+2
0C4E' C900 0080            JUMP LOOPA IF ACC≠128
0C52' 2038
                    /
                    / COMPLETION OF INSTRUCTION TRANSMISSION CYCLE
                    /
0C54' 0808          NOLOAD: ACC=TST&X'F00F'
0C56' C400 F00F
0C5A' 0898                  TST1=TST
0C5C' C900 1001             JUMPS OUTDAT IF ACC=X'1001' / TEST IF AP MUST READ DATA
0C60' 233E
0C62' C900 2001             JUMP INDAT IF ACC=X'2001' / TEST IF AP MUST WRITE DATA
0C66' 4330 0CA6'
0C6A' C400 000F             ACC=ACC&X'000F'
0C6E' C900 0003             JUMP SEEK1 IF ACC=X'0003' / TEST IF SEEK INSTRUCTION
0C72' 4330 0CD6'
0C76' 2662          CONT1: PSTRM=PSTRM+2
0C78' 4300 0BF4'           JUMP INSTR
                    /
                    /
                    /
0C7C' 9084          OUTDAT: TST=TST>>4 / DETERMINE THE NUMBER OF WORDS TO OUTPUT
0C7E' 0808                  ACC=TST&X'000F'
0C80' C400 000F
0C84' 2601                  ACC=ACC+1
0C86' 0898                  TST1=TST>>4
0C88' 9094
0C8A' C490 000F             TST1=TST1&X'000F'
0C8E' 2691                  TST1=TST1+1
0C90' 0C80                  MH TST,ACC
0C92' 2791          OUT1: TST1=TST1-1 / OUTPUT THE DATA
0C94' 4827 0E52'          IODATA=DATA(OSTRM)
0C98' 41F0 0DF6'          CALL LKOUT
0C9C' 2672                OSTRM=OSTRM+2
0C9E' 0899                JUMP OUT1 IF TST1≠0
0CA0' 2037
0CA2' 4300 0C76'          JUMP CONT1
```

```
                   /
                   /
                   /
0CA6' 9084         INDAT: TST=TST>>4 / DETERMINE THE NUMBER OF WORDS TO INPUT
0CA8' 0808                ACC=TST&X'000F'
0CAA' C400 000F
0CAE' 2601                ACC=ACC+1
0CB0' 0898                TST1=TST>>4
0CB2' 9094
0CB4' C490 000F           TST1=TST1&X'000F'
0CB8' 2691                TST1=TST1+1
0CBA' 0C80                MH TST,ACC
0CBC' 2791         IN1: TST1=TST1-1 / INPUT THE DATA
0CBE' 41F0 0E3A'        CALL LKIN
0CC2' C9A0 FFFF         JUMPS IN2 IF ACC1=X'FFFF' / IGNORE INPUT OF X'FFFF'
0CC6' 2334
0CC8' 40AC 0E52'        DATA(ISTRM)=ACC1
0CCC' 26C2              ISTRM=ISTRM+2
0CCE' 0899         IN2: JUMP IN1 IF TST1≠0
0CD0' 203A
0CD2' 4300 0C76'        JUMP CONT1
                   /
                   /
                   /
0CD6' 2662         SEEK1: PSTRM=PSTRM+2
0CD8' 4826 0000           IODATA=(PSTRM)
0CDC' 41F0 0DF6'          CALL LKOUT / OUTPUT COMMON VALUE FOR COMPARISON
0CE0' C490 00F0           TST1=TST1&X'00F0' / CHECK IF STATUS RETURNED
0CE4' C990 0030           JUMP CONT1 IF TST1≠X'0030'
0CE8' 4230 0C76'
0CEC' C450 00FF           STATUS=STATUS&X'00FF' / LOAD STATUS FROM AP
0CF0' 41F0 0E3A'          CALL LKIN
0CF4' 2531                WORKSP=\ACC1
0CF6' 073A
0CF8' 08A3                ACC1=WORKSP&X'FF00'
0CFA' C4A0 FF00
0CFE' 065A                STATUS=STATUS!ACC1
0D00' 41E0 0DB0'          CALL1 DSTAT
0D04' 4300 0C76'          JUMP CONT1
                   /
                   /
                   /
0D08' 0880         APSP: TST=ACC&X'000F'
0D0A' C480 000F
0D0E' C980 0008          JUMPS JZSR IF TST=X'0008'
0D12' 233D
0D14' C980 0009          JUMP RETURN IF TST=X'0009'
0D18' 4330 0D46'
0D1C' C980 000A          JUMP ACALL IF TST=X'000A'
0D20' 4330 0D4A'
0D24' C980 000B          JUMP ARETURN IF TST=X'000B'
0D28' 4330 0D66'
                   /
                   /
                   /
0D2C' 0805         JZSR: ACC=STATUS&X'FF00' / JUMP IF ZERO STATUS OCCURRED
0D2E' C400 FF00
0D32' 2662               PSTRM=PSTRM+2
0D34' 0800               JUMPS JZSR0 IF ACC≠0
0D36' 2136
0D38' 48A6 0000          ACC1=(PSTRM)
```

```
0D3C' 086A              PSTRM=ACC1
0D3E' 4300 0BF4'        JUMP INSTR
0D42' 4300 0C76' JZSRO: JUMP CONT1
                 /
                 /
                 /
0D46' 2662       RETURN: PSTRM=PSTRM+2
0D48' 030D               JUMP OUTB
                 /
                 /
                 /
0D4A' 2422       ACALL: STACK=STACK+2 / STORE AP PROGRAM COUNTER
0D4C' 6120 0E56'
0D50' C826 0004         (STACK)=PSTRM+4
0D54' 4830 0E56'
0D58' 4023 0000
0D5C' 48A6 0002         ACC1=(PSTRM+2)
0D60' 086A              PSTRM=ACC1
0D62' 4300 0BF4'        JUMP INSTR
                 /
                 /
                 /
0D66' 4820 0E56' ARETURN: PSTRM=(STACK) / LOAD PROGRAM COUNTER FROM STACK
0D6A' 4862 0000
0D6E' 2522                STACK=STACK-2
0D70' 6120 0E56'
0D74' 4300 0BF4'          JUMP INSTR
                 /
                 / END OF PROGRAM
                 /
0D78' C820 0045  END: IODATA='E'
0D7C' 41F0 0DC0'      CALL TTOUT
0D80' C820 004E       IODATA='N'
0D84' 41F0 0DC0'      CALL TTOUT
0D88' C820 0044       IODATA='D'
0D8C' 41F0 0DC0'      CALL TTOUT
0D90' 242D            IODATA=X'0D'
0D92' 41F0 0DC0'      CALL TTOUT
0D96' 242A            IODATA=X'0A'
0D98' 41F0 0DC0'      CALL TTOUT
0D9C' 2200       STOP: JUMP STOP
                 /
                 / TRANSMIT A CHARACTER TO THE LINEPRINTER AND EXIT
                 /
0D9E' 9A12       LPOUT: WD DEVICE,IODATA
0DA0' 40F0 0EAA'        LPT=R15
0DA4' D000 0E82'        STM ACC,MTP
0DA8' D100 0E62'        LM ACC,MULTIPLE
0DAC' C200 0040         LPSW INTRET
                 /
                 / DISPLAYS THE STATUS ON THE CONSOLE
                 /
0DB0' 0825       DSTAT: IODATA=STATUS
0DB2' 41F0 0E0A'        CALL CONOUT
0DB6' DE10 0088'        OC DEVICE,NORM
0DBA' DE10 008A'        OC DEVICE,INC
0DBE' 030E              JUMP OUTA
                 /
                 / WRITE SYMBOL TO TELETYPE
                 /
0DC0' 2412       TTOUT: DEVICE=TT
```

```
0DC2' 4880 0E52'          TST=OUTBUF+OTBFCT
0DC6' 4A80 0EA8'
0DCA' D228 0000           [TST]=IODATA
0DCE' C920 000A           JUMPS ON1 IF IODATA#X'0A'
0DD2' 2136
0DD4' 2421                FLAG=FLAG+1   / INCREMENT FLAG ON LF
0DD6' 6120 0EA6'
0DDA' DE10 007E'          OC DEVICE,WRITN   / ENABLES INTERRUPTS IN WRITE MODE
0DDE' 2421         ON1: OTBFCT=OTBFCT+1
0DE0' 6120 0EA8'
0DE4' 4820 0EA8'        JUMP OUT IF OTBFCT<400   / BUFFER IS 400 BYTES LONG
0DE8' C920 0190
0DEC' 021F
0DEE' 2420              OTBFCT=0
0DF0' 4020 0EA8'
0DF4' 030F              JUMP OUT
                   /
                   / TRANSMIT A 16-BIT WORD OVER LINK
                   /
0DF6' C810 0099    LKOUT: DEVICE=TRANS
0DFA' 9D13         LOOP2: SS DEVICE,WORKSP
0DFC' 20F1                JUMP LOOP2 IF 15
0DFE' 9A12                WD DEVICE,IODATA
0E00' 9028                IODATA=IODATA>>8
0E02' 9D13         LOOP3: SS DEVICE,WORKSP
0E04' 20F1                JUMP LOOP3 IF 15
0E06' 9A12                WD DEVICE,IODATA
0E08' 030F                JUMP OUT

                   /
                   / DISPLAY A 16-BIT WORD ON THE CONSOLE
                   /
0E0A' 2411         CONOUT: DEVICE=CONSL
0E0C' 9422                 EXBR IODATA,IODATA
0E0E' 9812                 WH DEVICE,IODATA
0E10' 030F                 JUMP OUT
                   /
                   / TYPE OUT A 16-BIT WORD STORED IN ACC1 IN HEXIDECIMAL
                   /
0E12' 24B0         PRINT: I=0
0E14' 082A         PRNT1: IODATA=ACC1&X'F000'
0E16' C420 F000
0E1A' 91A4                ACC1=ACC1<<4
0E1C' 902C                IODATA=IODATA>>12
0E1E' C920 000A           JUMPS PRNT2 IF IODATA<X'A'
0E22' 2112
0E24' 2627                IODATA=IODATA+7
0E26' CA20 0030    PRNT2: IODATA=IODATA+X'30'
0E2A' 41F0 0DC0'          CALL TTOUT
0E2E' 26B1                I=I+1
0E30' C9B0 0004           JUMP PRNT1 IF I#4
0E34' 4230 0E14'
0E38' 030E                JUMP OUTA
                   /
                   / RECEIVE A 16-BIT WORD OVER LINK, INTO ACC1
                   /
0E3A' C810 0089    LKIN: DEVICE=RECV
0E3E' 9D13         LOOP4: SS DEVICE,WORKSP
0E40' 20F1                JUMP LOOP4 IF 15
0E42' 9B12                RD DEVICE,IODATA
0E44' 08A2                ACC1=IODATA<<8
```

```
0E46' 91A8
0E48' 9D13         LOOP5: SS DEVICE,WORKSP
0E4A' 20F1                JUMP LOOP5 IF 15
0E4C' 9B12                RD DEVICE,IODATA
0E4E' 06A2                ACC1=ACC1!IODATA
0E50' 030F                JUMP OUT
                   /
                   /END OF PROGRAM FILE
                   /
                   DATA:
0E52' 1C52'        OUTBUF: *+X'0E00'   / TELETYPE OUTPUT BUFFER ADDRESS
0E54' 1DE4'        LP: *+X'0F90'   / LINEPRINTER OUTPUT DATA BLOCK ADDRESS
0E56' 1056'        STACK: *+X'0200' / STACK POINTER
0E58' 1068'        AIR2: *+X'0210' / AIR2 RELOCATABLE DATA BLOCK ADDRESS
                                   / STAGE 1 OUTPUT FROM AP
0E5A' 126A'        AIR3: *+X'0410' / AIR3 RELOCATABLE DATA BLOCK ADDRESS
                                   / STAGE 2 INPUT TO AP
0E5C' 146C'        AIR4: *+X'0610' / AIR4 RELOCATABLE DATA BLOCK ADDRESS
                                   / STAGE 2 OUTPUT FROM AP
0E5E' 176E'        AIR5: *+X'0910' / AIR5 RELOCATABLE DATA BLOCK ADDRESS
                                   / STAGE 3 INPUT TO AP
0E60' 1B70'        AIR6: *+X'0D10' / AIR6 RELOCATABLE DATA BLOCK ADDRESS
                                   / STAGE 3 OUTPUT FROM AP
0E62' 0000         MULTIPLE: X'0000' / REGISTER SAVE AREA
0E64' 0000                   X'0000'
0E66' 0000                   X'0000'
0E68' 0000                   X'0000'
0E6A' 0000                   X'0000'
0E6C' 0000                   X'0000'
0E6E' 0000                   X'0000'
0E70' 0000                   X'0000'
0E72' 0000                   X'0000'
0E74' 0000                   X'0000'
0E76' 0000                   X'0000'
0E78' 0000                   X'0000'
0E7A' 0000                   X'0000'
0E7C' 0000                   X'0000'
0E7E' 0000                   X'0000'
0E80' 0000                   X'0000'
0E82' 0000         MTP: X'0000' / REGISTER SAVE AREA FOR INTERRUPTS
0E84' 0000              X'0000'
0E86' 0000              X'0000'
0E88' 0000              X'0000'
0E8A' 0000              X'0000'
0E8C' 0000              X'0000'
0E8E' 0000              X'0000'
0E90' 0000              X'0000'
0E92' 0000              X'0000'
0E94' 0000              X'0000'
0E96' 0000              X'0000'
0E98' 0000              X'0000'
0E9A' 0000              X'0000'
0E9C' 0000              X'0000'
0E9E' 0000              X'0000'
0EA0' 0000              X'0000'
0EA2' 0000         DTBFCT: X'0000' / TELETYPE INTERRUPT OUTPUT BUFFER COUNT
0EA4' 0000         LPFLAG: X'0000'   / LINEPRINTER FLAG - SET WHEN IT STARTS
0EA6' 0000         FLAG: X'0000' / READ/WRITE FLAG
0EA8' 0000         OTBFCT: X'0000' / TELETYPE OUTPUT BUFFER COUNT
0EAA' 0000         LPT: X'0000' / LINEPRINTER ROUTINE RETURN ADDRESS
0EAC' 0700         VAR: X'0700' / INITIAL RADIUS FOR AIRCRAFT
```

```
                    AIR1:
0EAE' 0001          X'0001'
0EB0' 3000          X'3000'
0EB2' 23FF          X'23FF'
0EB4' 0020          X'0020'
0EB6' 0010          X'0010'
0EB8' 0002          X'0002'
0EBA' 7000          X'7000'
0EBC' 3000          X'3000'
0EBE' FFD0          X'FFD0'
0EC0' 0000          X'0000'
0EC2' 0003          X'0003'
0EC4' 3000          X'3000'
0EC6' 1000          X'1000'
0EC8' 0020          X'0020'
0ECA' 0010          X'0010'
0ECC' 0004          X'0004'
0ECE' 7000          X'7000'
0ED0' 1000          X'1000'
0ED2' FFD0          X'FFD0'
0ED4' FFE0          X'FFE0'
0ED6' 0005          X'0005'
0ED8' 3000          X'3000'
0EDA' 6700          X'6700'
0EDC' 0020          X'0020'
0EDE' 0000          X'0000'
0EE0' 0006          X'0006'
0EE2' 7000          X'7000'
0EE4' 3200          X'3200'
0EE6' 0015          X'0015'
0EE8' 0000          X'0000'
0EEA' 0007          X'0007'
0EEC' 3000          X'3000'
0EEE' 3000          X'3000'
0EF0' 0020          X'0020'
0EF2' 0000          X'0000'
0EF4' 0008          X'0008'
0EF6' 7000          X'7000'
0EF8' 3000          X'3000'
0EFA' FFD0          X'FFD0'
0EFC' 0000          X'0000'
0EFE' 0009          X'0009'
0F00' 4269          X'4269'
0F02' 5277          X'5277'
0F04' FFD0          X'FFD0'
0F06' FFD0          X'FFD0'
0F08' 000A          X'000A'
0F0A' 1111          X'1111'
0F0C' 6123          X'6123'
0F0E' 0023          X'0023'
0F10' 0015          X'0015'
0F12' 000B          X'000B'
0F14' 6765          X'6765'
0F16' 2345          X'2345'
0F18' 0005          X'0005'
0F1A' 0023          X'0023'
0F1C' 000C          X'000C'
0F1E' 1567          X'1567'
0F20' 7444          X'7444'
0F22' 0025          X'0025'
0F24' FFD0          X'FFD0'
```

```
0F26' 000D        X'000D'
0F28' 04FF        X'04FF'
0F2A' 1FFF        X'1FFF'
0F2C' 0010        X'0010'
0F2E' 0010        X'0010'
0F30' 000E        X'000E'
0F32' 3F67        X'3F67'
0F34' 4555        X'4555'
0F36' FFE0        X'FFE0'
0F38' FFE0        X'FFE0'
0F3A' 000F        X'000F'
0F3C' 1FEC        X'1FEC'
0F3E' 2300        X'2300'
0F40' 0015        X'0015'
0F42' 0005        X'0005'
0F44' 0010        X'0010'
0F46' 4500        X'4500'
0F48' 4500        X'4500'
0F4A' 0010        X'0010'
0F4C' 0010        X'0010'
0F4E' 0011        X'0011'
0F50' 6500        X'6500'
0F52' 6500        X'6500'
0F54' FFE0        X'FFE0'
0F56' FFE0        X'FFE0'
0F58' 0012        X'0012'
0F5A' 2370        X'2370'
0F5C' 7444        X'7444'
0F5E' 0005        X'0005'
0F60' 0017        X'0017'
0F62' 0013        X'0013'
0F64' 2500        X'2500'
0F66' 3400        X'3400'
0F68' 0023        X'0023'
0F6A' 0000        X'0000'
0F6C' 0014        X'0014'
0F6E' 0000        X'0000'
0F70' 5000        X'5000'
0F72' 0010        X'0010'
0F74' 0010        X'0010'
0F76' 0015        X'0015'
0F78' 5432        X'5432'
0F7A' 0745        X'0745'
0F7C' 0000        X'0000'
0F7E' FFD0        X'FFD0'
0F80' 0016        X'0016'
0F82' 5600        X'5600'
0F84' 0342        X'0342'
0F86' 0000        X'0000'
0F88' 0020        X'0020'
0F8A' 0017        X'0017'
0F8C' 2300        X'2300'
0F8E' 5600        X'5600'
0F90' 0020        X'0020'
0F92' 0010        X'0010'
0F94' 0018        X'0018'
0F96' 0000        X'0000'
0F98' 4500        X'4500'
0F9A' 0005        X'0005'
0F9C' 0015        X'0015'
0F9E' 0000              X'0000'
```

```
0FA0' 0000              X'0000'
0FA2' 0000              X'0000'
0FA4' 0000              X'0000'
0FA6' 0000              X'0000'
0FA8'          ££
```

Application 2.

```
                  £DEF *LIST=15
                  £DEF ACC=R0,DEVICE=R1,IODATA=R2,WORKSP=R3
                  £DEF STREAM=R4,STATUS=R5,PSTRM=R6,OSTRM=R7
                  £DEF TST=R8,TST1=R9,ACC1=R10,I=R11
                  £DEF ISTRM=R12
                  £DEF CONSL=1,TT=2,TRANS=X'99',RECV=X'89'
                  £DEF LPLK=X'98'
                  £DEF CALL=BAL R15
                  £DEF CALL1=BAL R14
                  £DEF CALL2=BAL R13
                  £DEF OUT=(R15)
                  £DEF OUTA=(R14)
                  £DEF OUTB=(R13)
                  £DEF INTRET=(X'0040')
                  /
                  / EXTERNAL INTERRUPT ROUTINE
                  /
0000' D000 0A6E' EXINT: STM ACC,MULTIPLE
0004' D100 0A8E'        LM ACC,MTP
0008' 9F13              AI DEVICE,WORKSP
000A' C910 0002         JUMPS TTINT1 IF DEVICE=2
000E' 2337
0010' D000 0A8E'        STM ACC,MTP
0014' D100 0A6E'        LM ACC,MULTIPLE
0018' C200 0040         LPSW INTRET
                  /
                  / TELETYPE INTERRUPT
                  /
001C' 4820 0AB4' TTINT1:  JUMP TTINT5 IF FLAG=0   / READ INTERRUPT
0020' 4330 006E'
0024' 48A0 0AAE'          ACC1=DTBFCT+OUTBUF    / WRITE INTERRUPT
0028' 4AA0 0A6A'
002C' DA1A 0000           WD DEVICE,[ACC1]
0030' 2421                DTBFCT=DTBFCT+1
0032' 6120 0AAE'
0036' 4820 0AAE'          JUMPS TTINT2 IF DTBFCT<400   / BUFFER LENGTH IS 400
003A' C920 0190
003E' 2114
0040' 2420                DTBFCT=0
0042' 4020 0AAE'
0046' D32A 0000  TTINT2: JUMP TTINTA IF [ACC1]#X'0A'   / EXIT IF NOT LF
004A' C920 000A
004E' 4230 00AC'
0052' 2521                FLAG=FLAG-1
0054' 6120 0AB4'
0058' 4230 00AC'          JUMP TTINTA IF FLAG#0   / EXIT IF OUTPUT NOT ENDED
005C' 9D13       TTINT3: SS DEVICE,WORKSP
005E' 20F1                JUMP TTINT3 IF 15
0060' DE10 00BC'          OC DEVICE,READ
0064' 9D13       TTINT4: SS DEVICE,WORKSP
0066' 20F1                JUMP TTINT4 IF 15
0068' 9B12                RD DEVICE,IODATA
006A' 4300 00AC'          JUMP TTINTA
006E' 4880 0ABC' TTINT5: TST=STR+INBFCT
0072' 4A80 0AB0'
0076' 9B12                RD DEVICE,IODATA
0078' C420 007F           IODATA=IODATA&X'007F' / GET RID OF PARITY BIT
007C' D228 0000           [TST]=IODATA
0080' C920 000D           JUMP TTINT9 IF IODATA#X'0D'
0084' 4230 00A6'
0088' 9D13       TTINT6: SS DEVICE,WORKSP
```

```
008A' 2071                 JUMP TTINT6 IF 7
008C' DE10 00BA'           OC DEVICE,WRITE
0090' 9D13        TTINT7:  SS DEVICE,WORKSP
0092' 2071                 JUMP TTINT7 IF 7
0094' 242A                 IODATA=X'000A'
0096' 9A12                 WD DEVICE,IODATA
0098' 9D13        TTINT8:  SS DEVICE,WORKSP
009A' 2071                 JUMP TTINT8 IF 7
009C' DE10 00BC'           OC DEVICE,READ
00A0' 2421                 INFLAG=1
00A2' 4020 0AB2'
00A6' 2421        TTINT9:  INBFCT=INBFCT+1
00A8' 6120 0AB0'
00AC' D000 0A8E'  TTINTA:  STM ACC,MTP / EXIT FROM INTERRPUT ROUTINE
00B0' D100 0A6E'           LM ACC,MULTIPLE
00B4' C200 0040            LPSW INTRET
                  /
                  /
                  /
                  £DEF *LIST=14
00B8' 4800        WRITN: X'4800' / WRITE MODE WITH INTERRUPTS ENABLED
00BA' C800        WRITE: X'C800' / WRITE MODE WITH INTERRUPTS DISARMED
00BC' 6400        READ: X'6400' / READ MODE WITH INTERRUPTS ENABLED
00BE' 8000        LINK: X'8000' / LINK DISABLED
00C0' 4000        LINKON: X'4000' / LINK ENABLED
00C2' 8000        NORM: X'8000' / NORMAL CONSOLE MODE
00C4' 4000        INC: X'4000' / INCREMENTAL CONSOLE MODE
00C6' 1111        CONTROL: X'1111'
00C8' 4000        OLDPSW: X'4000'
00CA' 0516'              ≑ST1
                  FM: / CONTROL LEVEL 0
00CC' 0000        X'0000',X'0000',X'0000',X'0000',X'0000',X'00FF',X'0001',X'1800
00DC' 5F00        X'5F00',X'0110',X'0000',X'6001',X'9000',X'1101',X'0010',X'2000
00EC' 10FF        X'10FF',X'0200',X'3037',X'00F2',X'1C10',X'2280',X'0002',X'0800
00FC' 20FF        X'20FF',X'0300',X'0FF0',X'0000',X'1990',X'3300',X'0003',X'0004
010C' 30FF        X'30FF',X'0400',X'0000',X'0000',X'4900',X'4400',X'0000',X'0080
011C' 40FF        X'40FF',X'0500',X'0000',X'0000',X'2002',X'5500',X'0000',X'0020
012C' 00F3        X'00F3',X'0600',X'0000',X'0000',X'0720',X'6600',X'0000',X'0000
013C' 70FF        X'70FF',X'0700',X'0000',X'0000',X'4002',X'7700',X'0000',X'0100
014C' 00F0        X'00F0',X'0800',X'0000',X'0000',X'0920',X'8800',X'0000',X'0000
015C' 80FF        X'80FF',X'0900',X'0000',X'0000',X'1040',X'9900',X'0000',X'0000
016C' 90FF        X'90FF',X'0A00',X'0000',X'0000',X'1002',X'AA00',X'0000',X'0000
017C' A0FF        X'A0FF',X'0B00',X'0000',X'0000',X'1980',X'BB00',X'0000',X'0000
018C' 00F4        X'00F4',X'0C00',X'0000',X'0000',X'0000',X'CC00',X'0000',X'0000
019C' 00F5        X'00F5',X'0D00',X'0000',X'0000',X'0000',X'DD00',X'0000',X'0000
01AC' B224        X'B224',X'0E00',X'0000',X'0000',X'0000',X'EE00',X'0000',X'0000
01BC' 0000        X'0000',X'0F00',X'0000',X'0000',X'0000',X'FFFF',X'0000',X'0000
                  / CONTROL LEVEL 1
01CC' 0000        X'0000',X'20F0',X'0010',X'0101',X'0000',X'0000',X'00F0',X'0013
01DC' 2000        X'2000',X'00E0',X'0020',X'0202',X'8020',X'1000',X'00F1',X'1012
01EC' 1000        X'1000',X'00D0',X'0030',X'0303',X'4040',X'2000',X'00F2',X'2012
01FC' 0000        X'0000',X'00C0',X'0040',X'0404',X'0720',X'3000',X'00F3',X'3012
020C' 0000        X'0000',X'00B0',X'0050',X'0505',X'0000',X'4000',X'00F4',X'4012
021C' 0000        X'0000',X'00A0',X'0060',X'0606',X'0000',X'5000',X'00F5',X'5012
022C' 0000        X'0000',X'0090',X'0070',X'0707',X'0000',X'6000',X'00F6',X'6012
023C' 0000        X'0000',X'0080',X'0080',X'0808',X'0000',X'7000',X'00F7',X'7012
024C' 0000        X'0000',X'0070',X'0090',X'0909',X'0000',X'8000',X'00F8',X'8012
025C' 0000        X'0000',X'0060',X'00A0',X'0A0A',X'0000',X'9000',X'00F9',X'9012
026C' 0000        X'0000',X'0050',X'00B0',X'0B0B',X'0000',X'A000',X'00FA',X'A012
027C' 0000        X'0000',X'0040',X'00C0',X'0C0C',X'0000',X'B000',X'00FB',X'B012
028C' 0000        X'0000',X'0030',X'00D0',X'0D0D',X'0000',X'C000',X'00FC',X'C012
```

```
029C' 0000          X'0000',X'0020',X'00E0',X'0E0E',X'0000',X'D000',X'00FD',X'D012
02AC' 0000          X'0000',X'0010',X'20F0',X'0F0F',X'0000',X'E000',X'00FE',X'E012
02BC' 0000          X'0000',X'0000',X'20FF',X'0000',X'0000',X'F000',X'0000',X'F012
                    / CONTROL LEVEL 2
02CC' 0000          X'0000',X'0000',X'0000',X'0000',X'A000',X'0100',X'0001',X'1000
02DC' 0002          X'0002',X'0110',X'0000',X'10F0',X'1501',X'1100',X'0010',X'2080
02EC' 0103          X'0103',X'0220',X'0000',X'FF0C',X'2002',X'2100',X'0020',X'3000
02FC' 8001          X'8001',X'0330',X'0000',X'30F0',X'1401',X'3100',X'0030',X'4000
030C' 900D          X'900D',X'0440',X'0000',X'4104',X'4A91',X'4100',X'0040',X'5000
031C' 0000          X'0000',X'0550',X'0000',X'0005',X'1591',X'5100',X'0050',X'6000
032C' 0000          X'0000',X'0660',X'0000',X'5006',X'4A80',X'6100',X'0060',X'B080
033C' 0000          X'0000',X'0770',X'0000',X'0F07',X'1991',X'7100',X'0070',X'8800
034C' 0000          X'0000',X'0880',X'0000',X'6F08',X'1B00',X'8100',X'0080',X'9080
035C' 0000          X'0000',X'0990',X'0000',X'4F09',X'1600',X'9100',X'0090',X'A100
036C' 0000          X'0000',X'0AA0',X'0000',X'2FFA',X'0000',X'A100',X'00A0',X'0800
037C' 0000          X'0000',X'0BB0',X'0000',X'700B',X'0000',X'B100',X'00B0',X'7100
038C' 0000          X'0000',X'0CC0',X'0000',X'FF0E',X'0000',X'C100',X'00C0',X'0800
039C' 0000          X'0000',X'0DD0',X'0000',X'0002',X'0000',X'D100',X'00D0',X'C080
03AC' 0000          X'0000',X'0EE0',X'0000',X'0000',X'0000',X'E100',X'00E0',X'D800
03BC' 0000          X'0000',X'0F00',X'0000',X'0000',X'0000',X'F1FF',X'0000',X'0000
                    / CONTROL LEVEL 3
03CC' 0000          X'0000',X'0000',X'0000',X'0000',X'9000',X'0F00',X'1000',X'1000
03DC' 0010          X'0010',X'0100',X'0000',X'1002',X'2041',X'1100',X'1010',X'0001
03EC' 0020          X'0020',X'0200',X'0000',X'2003',X'0405',X'2200',X'1020',X'2010
03FC' 0030          X'0030',X'0300',X'0000',X'F0F1',X'0060',X'3300',X'1030',X'0008
040C' 0040          X'0040',X'0400',X'0000',X'F0F4',X'0000',X'4400',X'1040',X'0020
041C' 0050          X'0050',X'0500',X'0000',X'F0F5',X'0000',X'5500',X'1050',X'0040
042C' 0060          X'0060',X'0600',X'0000',X'F0FF',X'0000',X'6600',X'1060',X'0000
043C' 0070          X'0070',X'0700',X'0000',X'F0F7',X'0000',X'7700',X'1070',X'0200
044C' 0080          X'0080',X'0800',X'0000',X'0000',X'0000',X'8800',X'1080',X'0000
045C' 0090          X'0090',X'0900',X'0000',X'0000',X'0000',X'9900',X'1090',X'0000
046C' 00A0          X'00A0',X'0A00',X'0000',X'0000',X'0000',X'AA00',X'10A0',X'0000
047C' 00B0          X'00B0',X'0B00',X'0000',X'0000',X'0000',X'BB00',X'10B0',X'0000
048C' 00C0          X'00C0',X'0C00',X'0000',X'0000',X'0000',X'CC00',X'10C0',X'0000
049C' 00D0          X'00D0',X'0D00',X'0000',X'0000',X'0000',X'DD00',X'10D0',X'0000
04AC' 00E0          X'00E0',X'0E00',X'0000',X'0000',X'0000',X'EE00',X'10E0',X'0000
04BC' 30F1          X'30F1',X'0F00',X'0000',X'0000',X'0020',X'FF00',X'2000',X'0000
                    £DEF *LIST=15
                    /
                    / THE MAIN SP PROGRAM STARTS HERE
                    /
04CC' 2411          START: DEVICE=CONSL / RESET CONSOLE DISPLAY
04CE' DE10 00C4'           OC DEVICE,INC
04D2' 2420                 IODATA=X'0000'
04D4' 41F0 0A22'           CALL CONOUT
04D8' 41F0 0A22'           CALL CONOUT
04DC' 2420                 DTBFCT=0
04DE' 4020 0AAE'
04E2' 4020 0AB6'           OTBFCT=0
04E6' C810 0098            DEVICE=LPLK
04EA' DE10 00BE'           OC DEVICE,LINK
04EE' C810 0099            DEVICE=TRANS
04F2' DE10 00BE'           OC DEVICE,LINK
04F6' C810 0089            DEVICE=RECV
04FA' DE10 00BE'           OC DEVICE,LINK
04FE' C810 0088            DEVICE=X'0088'
0502' DE10 00BE'           OC DEVICE,LINK
0506' 2412                 DEVICE=TT
0508' 2420                 FLAG=0
050A' 4020 0AB4'
050E' DE10 00BC'           OC DEVICE,READ
```

```
0512' C200 00C8'          LPSW OLDPSW / ENABLE INTERRUPTS
0516' 245F          ST1: STATUS=X'000F' / INITIAL AP STATUS
0518' 0825                IODATA=STATUS / TRANSMIT INITIAL STATUS
051A' 41F0 0A0E'          CALL LKOUT
051E' 41E0 09C8'          CALL1 DSTAT
0522' 2420                INBFCT=0 / RESET INPUT BUFFER COUNT
0524' 4020 0AB0'
0528' 4020 0AB2'          INFLAG=0
                    /
                    /
                    /
052C' 4820 0AB2' STAGE0: JUMPS STAGE0 IF INFLAG=0
0530' 2232
0532' 48B0 0ABC'           I=STR
0536' 2480                 TST=0
0538' 2490                 TST1=0
053A' D3AB 0000  PREP: ACC1=[I]
053E' 26B1               I=I+1
0540' C9A0 0061          JUMPS PREP0 IF ACC1<X'61' / 'A'
0544' 2116
0546' C9A0 007A          JUMPS PREP0 IF ACC1>X'7A' / 'Z'
054A' 2123
054C' 4300 0596'         JUMP PREP1
0550' 0899       PREP0: JUMPS SPACE2 IF TST1=0
0552' 233B
0554' C820 0020  SPACE1: STRING[TST]=' '
0558' D228 0C04'
055C' 2681                 TST=TST+1
055E' 2691                 TST1=TST1+1
0560' C990 0006            JUMP SPACE1 IF TST1≠6
0564' 2038
0566' 2490                 TST1=0
0568' C9A0 0028  SPACE2: JUMP PREP IF ACC1='('
056C' 4330 053A'
0570' C9A0 0029            JUMP PREP IF ACC1=')'
0574' 4330 053A'
0578' C9A0 0030            JUMP PREP IF ACC1='='
057C' 4330 053A'
0580' D2A8 0C04'           STRING[TST]=ACC1
0584' 2681               TST=TST+1
0586' D2A8 0C04'         STRING[TST]=ACC1
058A' 2681               TST=TST+1
058C' C9A0 000D          JUMP PREP IF ACC1≠X'0D'
0590' 4230 053A'
0594' 230E               JUMPS STAGE1
0596' 2691       PREP1: TST1=TST1+1
0598' CAA0 FFE0          ACC1=ACC1-X'20'
059C' D2A8 0C04'         STRING[TST]=ACC1
05A0' 2681               TST=TST+1
05A2' C990 0006          JUMP PREP IF TST1≠6
05A6' 4230 053A'
05AA' 2490               TST1=0
05AC' 4300 053A'         JUMP PREP
05B0' 4860 0AB8' STAGE1: PSTRM=PROG
05B4' C820 1100            (PSTRM)=X'1100'
05B8' 4026 0000
05BC' 2662                 PSTRM=PSTRM+2
05BE' C820 6000            (PSTRM)=X'6000'
05C2' 4026 0000
05C6' 2662                 PSTRM=PSTRM+2
05C8' C820 1871            (PSTRM)=X'1871'
```

```
05CC' 4026 0000
05D0' 2662                PSTRM=PSTRM+2
05D2' C820 1000           (PSTRM)=X'1000'
05D6' 4026 0000
05DA' 2662                PSTRM=PSTRM+2
05DC' C820 8033           (PSTRM)=X'8033'
05E0' 4026 0000
05E4' 2662                PSTRM=PSTRM+2
05E6' 2420                (PSTRM)=X'0000'
05E8' 4026 0000
05EC' 2662                PSTRM=PSTRM+2
05EE' 24B0                I=0
05F0' 482B 0C04' STAGE2: JUMP ST3 IF STRING(I)=X'2A2A' / CHECK IF **
05F4' C920 2A2A
05F8' 4330 0640'
05FC' C820 8073           (PSTRM)=X'8073'
0600' 4026 0000
0604' 2662                PSTRM=PSTRM+2
0606' 482B 0C04'          (PSTRM)=STRING(I)
060A' 4026 0000
060E' 2662                PSTRM=PSTRM+2
0610' 26B2                I=I+2
0612' C820 8173           (PSTRM)=X'8173'
0616' 4026 0000
061A' 2662                PSTRM=PSTRM+2
061C' 482B 0C04'          (PSTRM)=STRING(I)
0620' 4026 0000
0624' 2662                PSTRM=PSTRM+2
0626' 26B2                I=I+2
0628' C820 8273           (PSTRM)=X'8273'
062C' 4026 0000
0630' 2662                PSTRM=PSTRM+2
0632' 482B 0C04'          (PSTRM)=STRING(I)
0636' 4026 0000
063A' 2662                PSTRM=PSTRM+2
063C' 26B2                I=I+2
063E' 2302                JUMPS STAGE3
0640' 26B2       ST3: I=I+2
0642' 482B 0C04' STAGE3: JUMP ST4 IF STRING(I)=X'2A2A' / CHECK IF **
0646' C920 2A2A
064A' 4330 0692'
064E' C820 8373           (PSTRM)=X'8373'
0652' 4026 0000
0656' 2662                PSTRM=PSTRM+2
0658' 482B 0C04'          (PSTRM)=STRING(I)
065C' 4026 0000
0660' 2662                PSTRM=PSTRM+2
0662' 26B2                I=I+2
0664' C820 8473           (PSTRM)=X'8473'
0668' 4026 0000
066C' 2662                PSTRM=PSTRM+2
066E' 482B 0C04'          (PSTRM)=STRING(I)
0672' 4026 0000
0676' 2662                PSTRM=PSTRM+2
0678' 26B2                I=I+2
067A' C820 8573           (PSTRM)=X'8573'
067E' 4026 0000
0682' 2662                PSTRM=PSTRM+2
0684' 482B 0C04'          (PSTRM)=STRING(I)
0688' 4026 0000
068C' 2662                PSTRM=PSTRM+2
```

```
068E' 26B2                I=I+2
0690' 2302                JUMPS STAGE4
0692' 26B2           ST4: I=I+2
0694' 482B 0C04'  STAGE4: JUMP ST5 IF STRING(I)=X'2A2A' / CHECK IF **
0698' C920 2A2A
069C' 4330 06E4'
06A0' C820 8673           (PSTRM)=X'8673'
06A4' 4026 0000
06A8' 2662                PSTRM=PSTRM+2
06AA' 482B 0C04'          (PSTRM)=STRING(I)
06AE' 4026 0000
06B2' 2662                PSTRM=PSTRM+2
06B4' 26B2                I=I+2
06B6' C820 8773           (PSTRM)=X'8773'
06BA' 4026 0000
06BE' 2662                PSTRM=PSTRM+2
06C0' 482B 0C04'          (PSTRM)=STRING(I)
06C4' 4026 0000
06C8' 2662                PSTRM=PSTRM+2
06CA' 26B2                I=I+2
06CC' C820 8873           (PSTRM)=X'8873'
06D0' 4026 0000
06D4' 2662                PSTRM=PSTRM+2
06D6' 482B 0C04'          (PSTRM)=STRING(I)
06DA' 4026 0000
06DE' 2662                PSTRM=PSTRM+2
06E0' 26B2                I=I+2
06E2' 2302                JUMPS STAGE5
06E4' 26B2           ST5: I=I+2
06E6' C820 1100   STAGE5: (PSTRM)=X'1100'
06EA' 4026 0000
06EE' 2662                PSTRM=PSTRM+2
06F0' C820 2971           (PSTRM)=X'2971'
06F4' 4026 0000
06F8' 2662                PSTRM=PSTRM+2
06FA' 482B 0C04'          JUMPS STAGE6 IF STRING(I)#X'2C2C' / CHECK IF COMPOSITE
06FE' C920 2C2C
0702' 213E
0704' 26B2                I=I+2
0706' C820 1000           (PSTRM)=X'1000'
070A' 4026 0000
070E' 2662                PSTRM=PSTRM+2
0710' C820 6000           (PSTRM)=X'6000'
0714' 4026 0000
0718' 2662                PSTRM=PSTRM+2
071A' 4300 05F0'          JUMP STAGE2
071E' 2428        STAGE6: (PSTRM)=X'0008'
0720' 4026 0000
0724' 2662                PSTRM=PSTRM+2
0726' 4820 0AB8'          (PSTRM)=PROG
072A' 4026 0000
072E' 2662                PSTRM=PSTRM+2
0730' 2429                (PSTRM)=X'0009'
0732' 4026 0000
0736' 4860 0AB8'  STAGE7: PSTRM=PROG
073A' C870 0ABE'          OSTRM=#DT-#DATA
073E' CB70 0A6A'
0742' 48C0 0ABA'          ISTRM=ANS-#DATA
0746' CBC0 0A6A'
074A' 41D0 07FE'          CALL2 INSTR
074E' 2420                DATA(ISTRM)=X'0000'
```

```
0750' 402C 0A6A'
0754' 26C2                      ISTRM=ISTRM+2
0756' 48B0 0ABA' I=ANS
075A' 482B 0000  RESULT: JUMP ONWD IF (I)=X'0000'
075E' 4330 07E4'
0762' 48AB 0000           ACC1=(I)
0766' 41E0 09A8'          CALL1 APRNT
076A' 26B2                I=I+2
076C' 48AB 0000           ACC1=(I)
0770' 41E0 09A8'          CALL1 APRNT
0774' 26B2                I=I+2
0776' 48AB 0000           ACC1=(I)
077A' 41E0 09A8'          CALL1 APRNT
077E' 26B2                I=I+2
0780' C820 0028           IODATA='('
0784' 41F0 09D8'          CALL TTOUT
0788' 48AB 0000           ACC1=(I)
078C' 41E0 09A8'          CALL1 APRNT
0790' 26B2                I=I+2
0792' 48AB 0000           ACC1=(I)
0796' 41E0 09A8'          CALL1 APRNT
079A' 26B2                I=I+2
079C' 48AB 0000           ACC1=(I)
07A0' 41E0 09A8'          CALL1 APRNT
07A4' 26B2                I=I+2
07A6' C820 0029           IODATA=')'
07AA' 41F0 09D8'          CALL TTOUT
07AE' C820 003D           IODATA='='
07B2' 41F0 09D8'          CALL TTOUT
07B6' 48AB 0000           ACC1=(I)
07BA' 41E0 09A8'          CALL1 APRNT
07BE' 26B2                I=I+2
07C0' 48AB 0000           ACC1=(I)
07C4' 41E0 09A8'          CALL1 APRNT
07C8' 26B2                I=I+2
07CA' 48AB 0000           ACC1=(I)
07CE' 41E0 09A8'          CALL1 APRNT
07D2' 26B2                I=I+2
07D4' 242D                IODATA=X'0D'
07D6' 41F0 09D8'          CALL TTOUT
07DA' 242A                IODATA=X'0A'
07DC' 41F0 09D8'          CALL TTOUT
07E0' 4300 075A'          JUMP RESULT
07E4' 242D       ONWD:    IODATA=X'0D'
07E6' 41F0 09D8'          CALL TTOUT
07EA' 242A                IODATA=X'0A'
07EC' 41F0 09D8'          CALL TTOUT
07F0' 2420                INFLAG=0
07F2' 4020 0AB2'
07F6' 4020 0AB0'          INBFCT=0
07FA' 4300 052C'          JUMP STAGE0
                 /
                 /
                 /
                 /
                 / THE MAIN AP SUPPORT PROGRAM COMMENCES HERE
                 /
                 /
                 / TRANSMIT INSTRUCTION
                 /
07FE' 4806 0000  INSTR: ACC=(PSTRM)
```

```
0802' 0880                    TST=ACC
0804' 0825                    IODATA=STATUS / DISPLAY INSTRUCTION AND STATUS
0806' 41F0 0A22'              CALL CONOUT
080A' 0820                    IODATA=ACC
080C' 41F0 0A22'              CALL CONOUT
0810' 0890                    TST1=ACC&X'0008' / TEST IF SP INSTRUCTION
0812' C490 0008
0816' 4230 0912'              JUMP APSP IF TST1#0
081A' 0820                    IODATA=ACC
081C' 41F0 0A0E'              CALL LKOUT
0820' 0800                    JUMP END IF ACC=0
0822' 4330 0982'
                  /
                  / CHECK CONTROL LEVEL
                  /
0826' C400 000F   ACC=ACC&X'000F'
082A' 0830        WORKSP=ACC\STATUS
082C' 0735
082E' C430 000F   WORKSP=WORKSP&X'000F'
0832' 4330 085E'  JUMP NOLOAD IF WORKSP=0
                  /
                  / LOAD FUNCTIONAL MEMORIES
                  /
0836' 0735        LOAD: WORKSP=WORKSP\STATUS / LOADS STATUS WITH CONTROL LEVEL
0838' 0853              STATUS=WORKSP
083A' 41E0 09C8'        CALL1 DSTAT
083E' 0840              STREAM=ACC / CALCULATE DISPLACEMENT
0840' 9148              STREAM=STREAM<<8
0842' 2400              ACC=0
0844' 4820 00C6'        IODATA=CONTROL / OUTPUT CONTROL WORD
0848' 41F0 0A0E'        CALL LKOUT
084C' 4824 00CC'  LOOPA: IODATA=FM(STREAM) / OUTPUT FUNCTIONAL MEMORY DATA
0850' 41F0 0A0E'         CALL LKOUT
0854' 2601               ACC=ACC+1
0856' 2642               STREAM=STREAM+2
0858' C900 0080          JUMP LOOPA IF ACC#128
085C' 2038
                  /
                  / COMPLETION OF INSTRUCTION TRANSMISSION CYCLE
                  /
085E' 0808        NOLOAD: ACC=TST&X'F00F'
0860' C400 F00F
0864' 0898                 TST1=TST
0866' C900 1001            JUMPS OUTDAT IF ACC=X'1001' / TEST IF AP MUST READ DA
086A' 233E
086C' C900 2001            JUMP INDAT IF ACC=X'2001' / TEST IF AP MUST WRITE DAT
0870' 4330 08B0'
0874' C400 000F            ACC=ACC&X'000F'
0878' C900 0003            JUMP SEEK1 IF ACC=X'0003' / TEST IF SEEK INSTRUCTION
087C' 4330 08E0'
0880' 2662        CONT1: PSTRM=PSTRM+2
0882' 4300 07FE'         JUMP INSTR
                  /
                  /
                  /
0886' 9084        OUTDAT: TST=TST>>4 / DETERMINE THE NUMBER OF WORDS TO OUTPUT
0888' 0808                ACC=TST&X'000F'
088A' C400 000F
088E' 2601                ACC=ACC+1
0890' 0898                TST1=TST>>4
0892' 9094
```

```
0894' C490 000F          TST1=TST1&X'000F'
0898' 2691               TST1=TST1+1
089A' 0C80               MH TST,ACC
089C' 2791         OUT1: TST1=TST1-1 / OUTPUT THE DATA
089E' 4827 0A6A'         IODATA=DATA(OSTRM)
08A2' 41F0 0A0E'         CALL LKOUT
08A6' 2672               OSTRM=OSTRM+2
08A8' 0899               JUMP OUT1 IF TST1≠0
08AA' 2037
08AC' 4300 0880'         JUMP CONT1
                   /
                   /
                   /
08B0' 9084         INDAT: TST=TST>>4 / DETERMINE THE NUMBER OF WORDS TO INPUT
08B2' 0808                ACC=TST&X'000F'
08B4' C400 000F
08B8' 2601                ACC=ACC+1
08BA' 0898                TST1=TST>>4
08BC' 9094
08BE' C490 000F           TST1=TST1&X'000F'
08C2' 2691                TST1=TST1+1
08C4' 0C80                MH TST,ACC
08C6' 2791         IN1: TST1=TST1-1 / INPUT THE DATA
08C8' 41F0 0A52'        CALL LKIN
08CC' C9A0 FFFF         JUMPS IN2 IF ACC1=X'FFFF' / IGNORE INPUT OF X'FFFF'
08D0' 2334
08D2' 40AC 0A6A'        DATA(ISTRM)=ACC1
08D6' 26C2              ISTRM=ISTRM+2
08D8' 0899         IN2: JUMP IN1 IF TST1≠0
08DA' 203A
08DC' 4300 0880'        JUMP CONT1
                   /
                   /
                   /
08E0' 2662         SEEK1: PSTRM=PSTRM+2
08E2' 4826 0000           IODATA=(PSTRM)
08E6' 41F0 0A0E'          CALL LKOUT / OUTPUT COMMON VALUE FOR COMPARISON
08EA' C490 00F0           TST1=TST1&X'00F0' / CHECK IF STATUS RETURNED
08EE' C990 0030           JUMP CONT1 IF TST1≠X'0030'
08F2' 4230 0880'
08F6' C450 00FF           STATUS=STATUS&X'00FF' / LOAD STATUS FROM AP
08FA' 41F0 0A52'          CALL LKIN
08FE' 2531                WORKSP=\ACC1
0900' 073A
0902' 08A3                ACC1=WORKSP&X'FF00'
0904' C4A0 FF00
0908' 065A                STATUS=STATUS!ACC1
090A' 41E0 09C8'          CALL1 DSTAT
090E' 4300 0880'          JUMP CONT1
                   /
                   /
                   /
0912' 0880         APSP: TST=ACC&X'000F'
0914' C480 000F
0918' C580 0008         JUMPS JZSR IF TST=X'0008'
091E' C980 0009         JUMP RETURN IF TST=X'0009'
0922' 4330 0950'
0926' C980 000A         JUMP ACALL IF TST=X'000A'
092A' 4330 0954'
092E' C980 000B         JUMP ARETURN IF TST=X'000B'
```

```
0932' 4330 0970'
                      /
                      /
                      /
0936' 0805            JZSR: ACC=STATUS&X'FF00' / JUMP IF ZERO STATUS OCCURRED
0938' C400 FF00
093C' 2662                  PSTRM=PSTRM+2
093E' 0800                  JUMPS JZSRO IF ACC#0
0940' 2136
0942' 48A6 0000             ACC1=(PSTRM)
0946' 086A                  PSTRM=ACC1
0948' 4300 07FE'            JUMP INSTR
094C' 4300 0880'      JZSRO: JUMP CONT1
                      /
                      /
                      /
0950' 2662            RETURN: PSTRM=PSTRM+2
0952' 030D                    JUMP OUTB
                      /
                      /
                      /
0954' 2422            ACALL: STACK=STACK+2 / STORE AP PROGRAM COUNTER
0956' 6120 0A6C'
095A' C826 0004              (STACK)=PSTRM+4
095E' 4830 0A6C'
0962' 4023 0000
0966' 48A6 0002              ACC1=(PSTRM+2)
096A' 086A                   PSTRM=ACC1
096C' 4300 07FE'             JUMP INSTR
                      /
                      /
                      /
0970' 4820 0A6C'      ARETURN: PSTRM=(STACK) / LOAD PROGRAM COUNTER FROM STACK
0974' 4862 0000
0978' 2522                     STACK=STACK-2
097A' 6120 0A6C'
097E' 4300 07FE'               JUMP INSTR
                      /
                      / END OF PROGRAM
                      /
0982' C820 0045       END: IODATA='E'
0986' 41F0 09D8'           CALL TTOUT
098A' C820 004E            IODATA='N'
098E' 41F0 09D8'           CALL TTOUT
0992' C820 0044            IODATA='D'
0996' 41F0 09D8'           CALL TTOUT
099A' 242D                 IODATA=X'0D'
099C' 41F0 09D8'           CALL TTOUT
09A0' 242A                 IODATA=X'0A'
09A2' 41F0 09D8'           CALL TTOUT
09A6' 2200            STOP: JUMP STOP
                      /
                      / TYPE OUT TWO CHARACTERS STORED IN ACC1 OMITTING SPACES
                      /
09A8' 082A            APRNT: IODATA=ACC1>>8
09AA' 9028
09AC' C920 0020              JUMPS APRNTO IF IODATA=X'0020'
09B0' 2333
09B2' 41F0 09D8'             CALL TTOUT
09B6' 082A            APRNTO: IODATA=ACC1&X'00FF'
09B8' C420 00FF
```

```
09BC' C920 0020         JUMP OUTA IF IODATA=X'0020'
09C0' 033E
09C2' 41F0 09D8'        CALL TTOUT
09C6' 030E              JUMP OUTA
                  /
                  / DISPLAYS THE STATUS ON THE CONSOLE
                  /
09C8' 0825        DSTAT: IODATA=STATUS
09CA' 41F0 0A22'        CALL CONOUT
09CE' DE10 00C2'        OC DEVICE,NORM
09D2' DE10 00C4'        OC DEVICE,INC
09D6' 030E              JUMP OUTA
                  /
                  / WRITE SYMBOL TO TELETYPE
                  /
09D8' 2412        TTOUT: DEVICE=TT
09DA' 4880 0A6A'        TST=OUTBUF+OTBFCT
09DE' 4A80 0AB6'
09E2' D228 0000         [TST]=IODATA
09E6' C920 000A         JUMPS ON1 IF IODATA#X'0A'
09EA' 2136
09EC' 2421              FLAG=FLAG+1    / INCREMENT FLAG ON LF
09EE' 6120 0AB4'
09F2' DE10 00B8'        OC DEVICE,WRITN    / ENABLES INTERRUPTS IN WRITE MODE
09F6' 2421        ON1: OTBFCT=OTBFCT+1
09F8' 6120 0AB6'
09FC' 4820 0AB6'      JUMP OUT IF OTBFCT<400    / BUFFER IS 400 BYTES LONG
0A00' C920 0190
0A04' 021F
0A06' 2420            OTBFCT=0
0A08' 4020 0AB6'
0A0C' 030F            JUMP OUT
                  /
                  / TRANSMIT A 16-BIT WORD OVER LINK
                  /
0A0E' C810 0099   LKOUT: DEVICE=TRANS
0A12' 9D13        LOOP2: SS DEVICE,WORKSP
0A14' 20F1              JUMP LOOP2 IF 15
0A16' 9A12              WD DEVICE,IODATA
0A18' 9028              IODATA=IODATA>>8
0A1A' 9D13        LOOP3: SS DEVICE,WORKSP
0A1C' 20F1              JUMP LOOP3 IF 15
0A1E' 9A12              WD DEVICE,IODATA
0A20' 030F              JUMP OUT

                  /
                  / DISPLAY A 16-BIT WORD ON THE CONSOLE
                  /
0A22' 2411        CONOUT: DEVICE=CONSL
0A24' 9422              EXBR IODATA,IODATA
0A26' 9812              WH DEVICE,IODATA
0A28' 030F              JUMP OUT
                  /
                  / TYPE OUT A 16-BIT WORD STORED IN ACC1 IN HEXIDECIMAL
                  /
0A2A' 24B0        PRINT: I=0
0A2C' 082A        PRNT1: IODATA=ACC1&X'F000'
0A2E' C420 F000
0A32' 91A4              ACC1=ACC1<<4
0A34' 902C              IODATA=IODATA>>12
0A36' C920 000A         JUMPS PRNT2 IF IODATA<X'A'
```

```
0A3A' 2112
0A3C' 2627              IODATA=IODATA+7
0A3E' CA20 0030  PRNT2: IODATA=IODATA+X'30'
0A42' 41F0 09D8'        CALL TTOUT
0A46' 26B1              I=I+1
0A48' C9B0 0004         JUMP PRNT1 IF I#4
0A4C' 4230 0A2C'
0A50' 030E              JUMP OUTA
                 /
                 / RECEIVE A 16-BIT WORD OVER LINK, INTO ACC1
                 /
0A52' C810 0089  LKIN: DEVICE=RECV
0A56' 9D13       LOOP4: SS DEVICE,WORKSP
0A58' 20F1              JUMP LOOP4 IF 15
0A5A' 9B12              RD DEVICE,IODATA
0A5C' 08A2              ACC1=IODATA<<8
0A5E' 91A8
0A60' 9D13       LOOP5: SS DEVICE,WORKSP
0A62' 20F1              JUMP LOOP5 IF 15
0A64' 9B12              RD DEVICE,IODATA
0A66' 06A2              ACC1=ACC1!IODATA
0A68' 030F              JUMP OUT
                 /
                 /END OF PROGRAM FILE
                 /
                 DATA:
0A6A' 186A'      OUTBUF: *+X'0E00'   / TELETYPE OUTPUT BUFFER ADDRESS
0A6C' 0C6C'      STACK: *+X'0200' / STACK POINTER
0A6E' 0000       MULTIPLE: X'0000' / REGISTER SAVE AREA
0A70' 0000                 X'0000'
0A72' 0000                 X'0000'
0A74' 0000                 X'0000'
0A76' 0000                 X'0000'
0A78' 0000                 X'0000'
0A7A' 0000                 X'0000'
0A7C' 0000                 X'0000'
0A7E' 0000                 X'0000'
0A80' 0000                 X'0000'
0A82' 0000                 X'0000'
0A84' 0000                 X'0000'
0A86' 0000                 X'0000'
0A88' 0000                 X'0000'
0A8A' 0000                 X'0000'
0A8C' 0000                 X'0000'
0A8E' 0000       MTP: X'0000' / REGISTER SAVE AREA FOR INTERRUPTS
0A90' 0000            X'0000'
0A92' 0000            X'0000'
0A94' 0000            X'0000'
0A96' 0000            X'0000'
0A98' 0000            X'0000'
0A9A' 0000            X'0000'
0A9C' 0000            X'0000'
0A9E' 0000            X'0000'
0AA0' 0000            X'0000'
0AA2' 0000            X'0000'
0AA4' 0000            X'0000'
0AA6' 0000            X'0000'
0AA8' 0000            X'0000'
0AAA' 0000            X'0000'
0AAC' 0000            X'0000'
0AAE' 0000       DTBFCT: X'0000' / TELETYPE INTERRUPT OUTPUT BUFFER COUNT
```

```
0AB0' 0000    INBFCT: X'0000' / TELETYPE INTERRUPT INPUT BUFFER COUNT
0AB2' 0000    INFLAG: X'0000' / FLAG - SET ON INPUT FROM TELETYPE
0AB4' 0000    FLAG: X'0000' / READ/WRITE FLAG
0AB6' 0000    OTBFCT: X'0000' / TELETYPE OUTPUT BUFFER COUNT
0AB8' 0FB8'   PROG: *+X'0500' / PROGRAM ADDRESS
0ABA' 0DBA'   ANS: *+X'0300' / ANSWER ADDRESS
0ABC' 1ABC'   STR: *+X'1000' / INPUT BUFFER
              DT:
              /
0ABE' 4D4F    'M','O','T','H','E','R'
0AC0' 5448
0AC2' 4552
0AC4' 4D41    'M','A','R','Y',' ',' '
0AC6' 5259
0AC8' 2020
0ACA' 4A41    'J','A','N','E',' ',' '
0ACC' 4E45
0ACE' 2020
              /
0AD0' 4D4F    'M','O','T','H','E','R'
0AD2' 5448
0AD4' 4552
0AD6' 4A41    'J','A','N','E',' ',' '
0AD8' 4E45
0ADA' 2020
0ADC' 414C    'A','L','I','C','E',' '
0ADE' 4943
0AE0' 4520
              /
0AE2' 4D4F    'M','O','T','H','E','R'
0AE4' 5448
0AE6' 4552
0AE8' 414C    'A','L','I','C','E',' '
0AEA' 4943
0AEC' 4520
0AEE' 4B41    'K','A','T','E',' ',' '
0AF0' 5445
0AF2' 2020
              /
0AF4' 4641    'F','A','T','H','E','R'
0AF6' 5448
0AF8' 4552
0AFA' 414C    'A','L','I','C','E',' '
0AFC' 4943
0AFE' 4520
0B00' 5445    'T','E','D',' ',' ',' '
0B02' 4420
0B04' 2020
              /
0B06' 5349    'S','I','S','T','E','R'
0B08' 5354
0B0A' 4552
0B0C' 4A41    'J','A','C','K',' ',' '
0B0E' 434B
0B10' 2020
0B12' 5041    'P','A','T',' ',' ',' '
0B14' 5420
0B16' 2020
              /
0B18' 534F    'S','O','N',' ',' ',' '
0B1A' 4E20
```

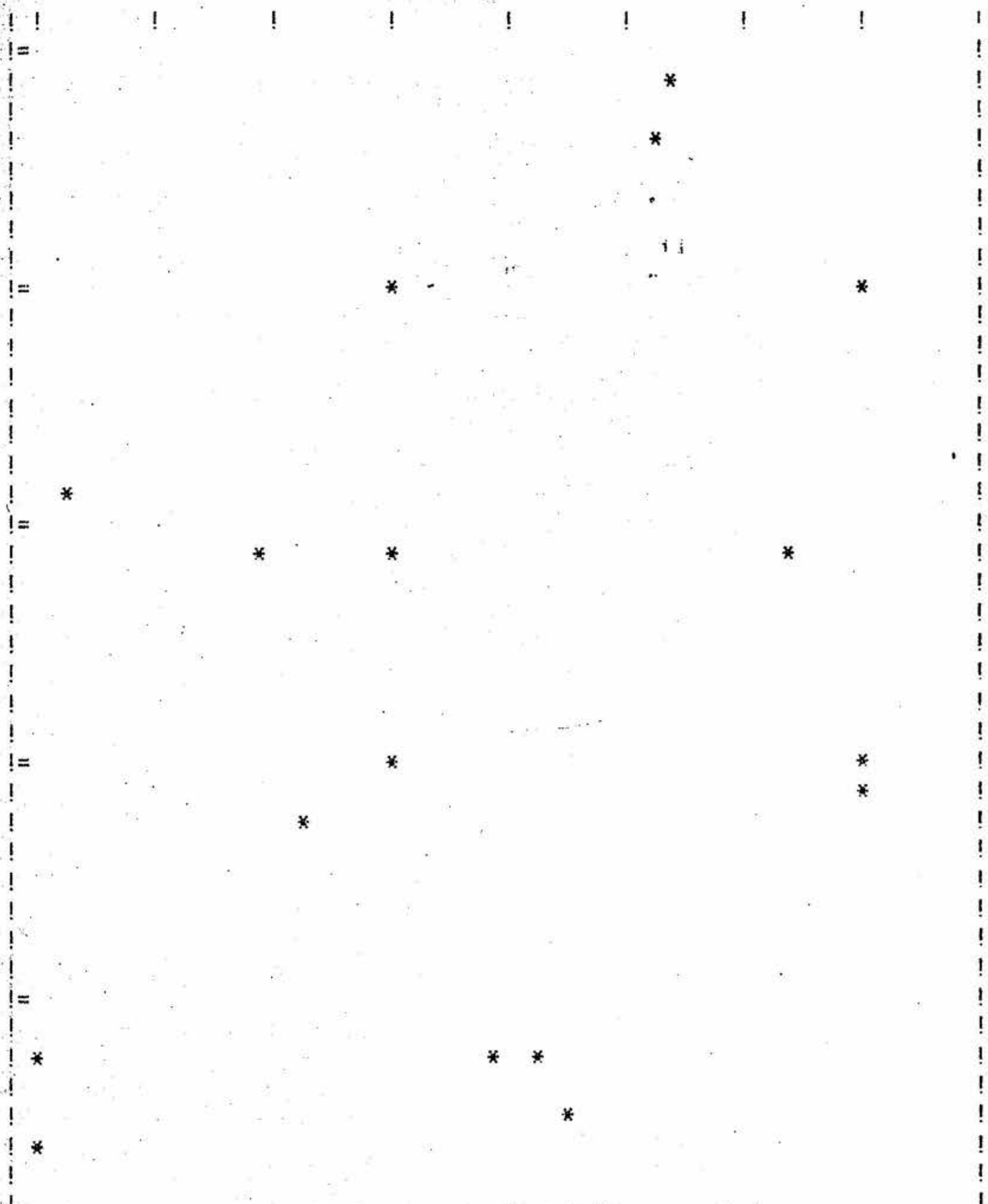
```
0B1C' 2020
0B1E' 544F        'T','O','M',' ',' ',' '
0B20' 4D20
0B22' 2020
0B24' 4249        'B','I','L','L',' ',' '
0B26' 4C4C
0B28' 2020
                  /
0B2A' 534F        'S','O','N',' ',' ',' '
0B2C' 4E20
0B2E' 2020
0B30' 544F        'T','O','M',' ',' ',' '
0B32' 4D20
0B34' 2020
0B36' 4A41        'J','A','C','K',' ',' '
0B38' 434B
0B3A' 2020
                  /
0B3C' 4252        'B','R','O','T','H','E'
0B3E' 4F54
0B40' 4845
0B42' 4A41        'J','A','C','K',' ',' '
0B44' 434B
0B46' 2020
0B48' 5445        'T','E','R','R','Y',' '
0B4A' 5252
0B4C' 5920
                  /
0B4E' 4252        'B','R','O','T','H','E'
0B50' 4F54
0B52' 4845
0B54' 4A41        'J','A','C','K',' ',' '
0B56' 434B
0B58' 2020
0B5A' 4A4F        'J','O','H','N',' ',' '
0B5C' 484E
0B5E' 2020
                  /
0B60' 4641        'F','A','T','H','E','R'
0B62' 5448
0B64' 4552
0B66' 414C        'A','L','F','R','E','D'
0B68' 4652
0B6A' 4544
0B6C' 544F        'T','O','M',' ',' ',' '
0B6E' 4D20
0B70' 2020
                  /
0B72' 4252        'B','R','O','T','H','E'
0B74' 4F54
0B76' 4845
0B78' 414C        'A','L','F','R','E','D'
0B7A' 4652
0B7C' 4544
0B7E' 5041        'P','A','U','L',' ',' '
0B80' 554C
0B82' 2020
                  /
0B84' 4252        'B','R','O','T','H','E'
0B86' 4F54
0B88' 4845
```

```
0B8A' 414C        'A','L','F','R','E','D'
0B8C' 4652
0B8E' 4544
0B90' 5045        'P','E','T','E','R',' '
0B92' 5445
0B94' 5220
                  /
0B96' 5349        'S','I','S','T','E','R'
0B98' 5354
0B9A' 4552
0B9C' 4A49        'J','I','M',' ',' ',' '
0B9E' 4D20
0BA0' 2020
0BA2' 4C55        'L','U','C','Y',' ',' '
0BA4' 4359
0BA6' 2020
                  /
0BA8' 5349        'S','I','S','T','E','R'
0BAA' 5354
0BAC' 4552
0BAE' 4A49        'J','I','M',' ',' ',' '
0BB0' 4D20
0BB2' 2020
0BB4' 414E        'A','N','N',' ',' ',' '
0BB6' 4E20
0BB8' 2020
                  /
0BBA' 4641        'F','A','T','H','E','R'
0BBC' 5448
0BBE' 4552
0BC0' 4A49        'J','I','M',' ',' ',' '
0BC2' 4D20
0BC4' 2020
0BC6' 4A4F        'J' 'O','H','N',' ',' '
0BC8' 484E
0BCA' 2020
                  /
0BCC' 4641        'F','A','T','H','E','R'
0BCE' 5448
0BD0' 4552
0BD2' 4A41        'J','A','C','K',' ',' '
0BD4' 434B
0BD6' 2020
0BD8' 4249        'B','I','L','L',' ',' '
0BDA' 4C4C
0BDC' 2020
                  /
0BDE' 4D4F        'M','O','T','H','E','R'
0BE0' 5448
0BE2' 4552
0BE4' 4A49        'J','I','M',' ',' ',' '
0BE6' 4D20
0BE8' 2020
0BEA' 4D41        'M','A','R','Y',' ',' '
0BEC' 5259
0BEE' 2020
                  /
0BF0' 4D4F        'M','O','T','H','E','R'
0BF2' 5448
0BF4' 4552
0BF6' 4A41        'J','A','C','K',' ',' '
```

```
0BF8' 434B
0BFA' 2020
0BFC' 4A49        'J','I','L','L',' ',' '
0BFE' 4C4C
0C00' 2020
                  /
0C02' 0000        X'0000'
                  STRING:
0C04'             ££
```

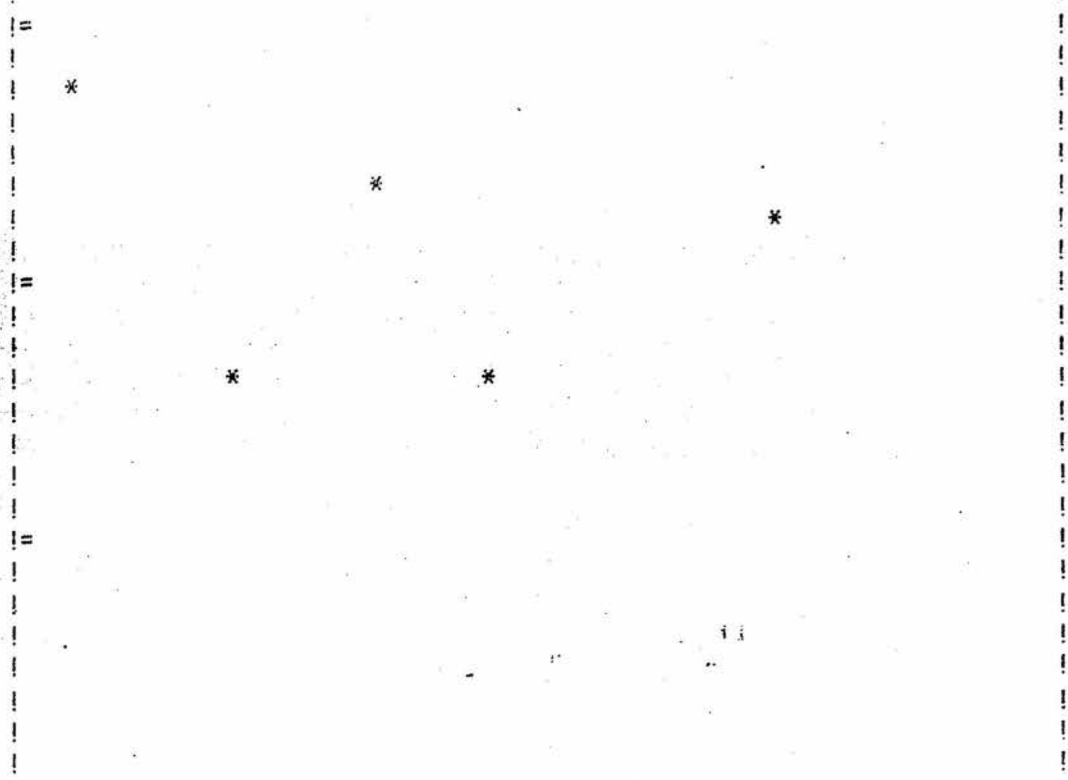
5. Sample Output.

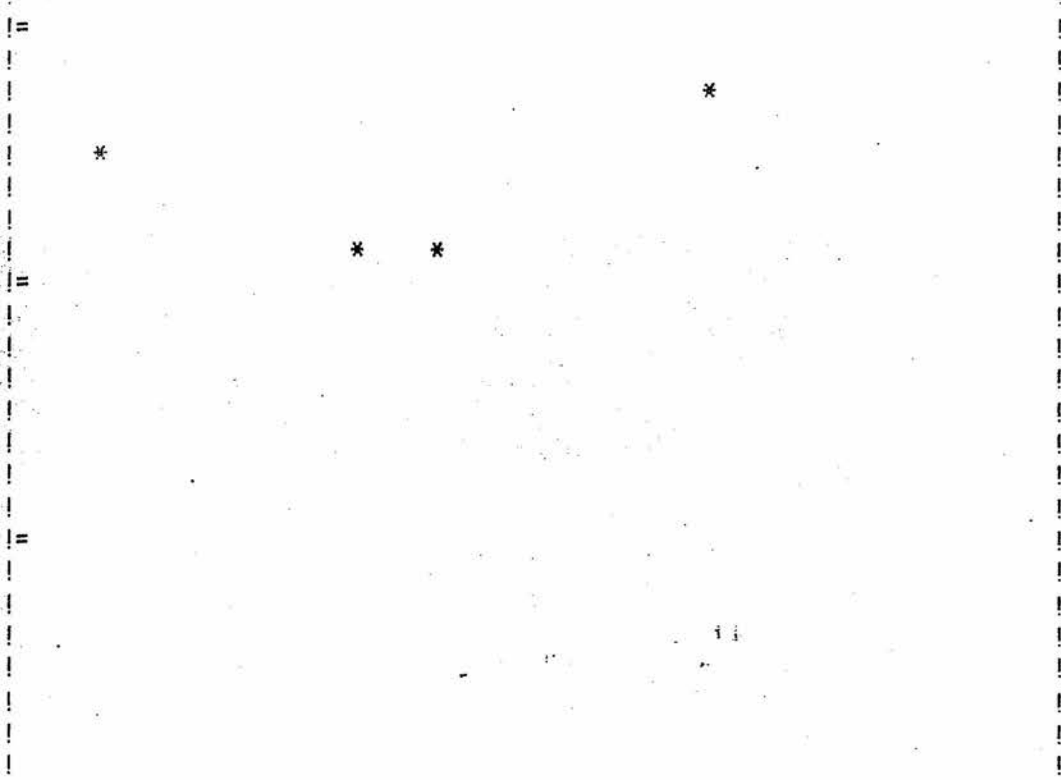
The following pages provide :
(1) The lineprinter listing of the simulated air space on running the first application program.
(2) The console typewriter output of potential conflicts on running the first application program.
(3) A sample listing for an interactive session using the second application program.

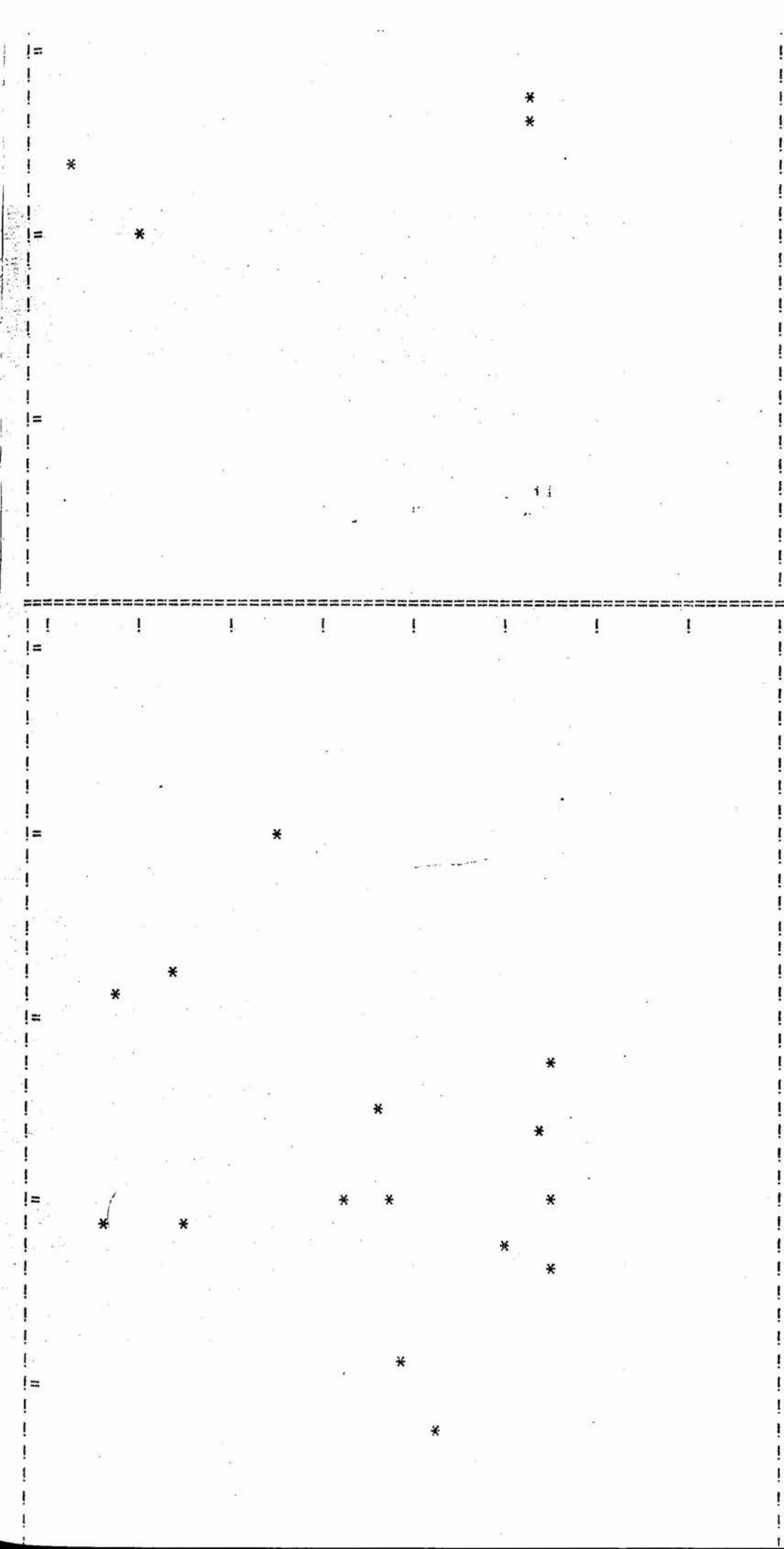


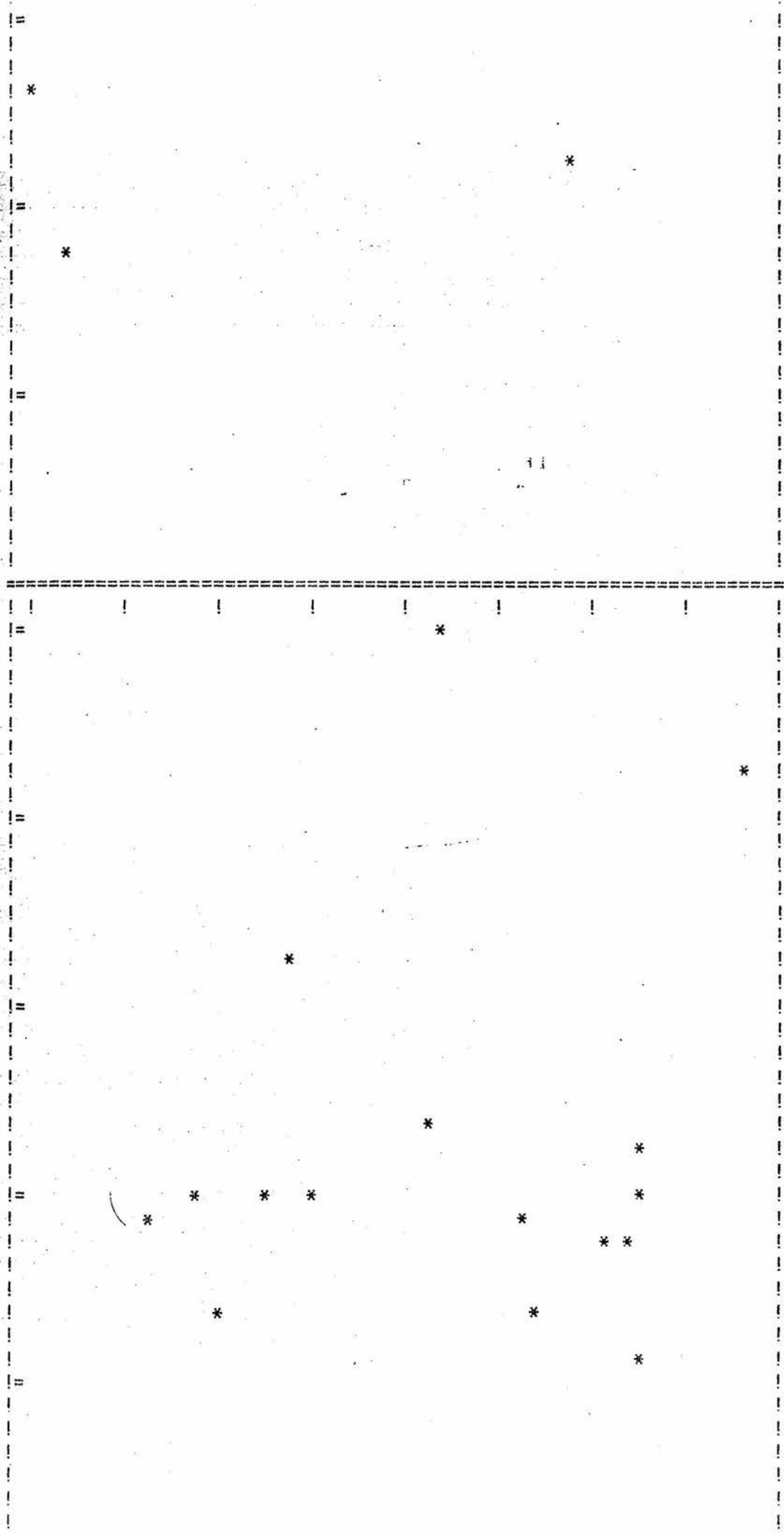
J

Application 1.









Application 1

0002	0038	0018	0008	0038	0018
000B	0033	0011	0002	0038	0018
000B	0033	0011	0008	0038	0018

0009	001B	0023	000E	001B	001E
0013	0016	001A	0009	001B	0023
0013	0016	001A	000E	001B	001E
000C	000F	0034	000A	000C	0033
0001	001C	0013	0007	001C	0018
0002	0032	0018	0008	0032	0018
0004	0032	0004	0016	002B	0005

000E	0017	001A	0009	0015	001D
000E	0017	001A	000F	0015	0012
0008	002D	0018	0002	002C	0018
0001	0020	0015	0007	0020	0018
0004	002C	0002	0016	002B	0009
0011	002A	002A	0010	0026	0026
0006	003D	0019	000B	003C	001A

0009	000F	0017	000D	0008	0015
0009	000F	0017	000E	0013	0016
0002	0026	0018	0008	0028	0018
0002	0026	0018	0013	001F	001A
0001	0024	0017	0002	0026	0018
0001	0024	0017	0008	0028	0018
0007	0024	0018	0002	0026	0018
0007	0024	0018	0008	0028	0018
0001	0024	0017	0007	0024	0018
0002	0026	0018	0002	0026	0018

0002	0020	0018	0008	0023	0018
0004	0021	003F	000A	001A	003B
000F	001A	0014	0002	0020	0018
0011	0023	0023	000C	001D	0027
0003	0028	0010	0016	002B	0011
0001	0028	0019	0007	0028	0018
0001	0028	0019	0013	0024	001A
0015	002A	002B	0010	002A	002A

0008	001E	0018	0002	001A	0018
000F	001D	0014	0008	001E	0018
0001	002C	0018	0013	0028	001A
0007	002C	0018	0013	0028	001A
0007	002C	0018	0016	002B	0015
0003	002C	0012	0016	002B	0015
0001	002C	0018	0007	002C	0018

000D	000E	001B	0002	0014	0018
0011	001B	001B	0008	0019	0018
000C	002E	001E	0015	002A	001F
0013	002C	001A	0016	002B	0019
0001	0030	001D	000C	002E	001E
0003	0030	0014	0007	0030	0018

0002	000E	0018	0006	000A	0019
0014	000E	0036	0004	0012	003C
000D	0010	001D	0008	0014	0018
0011	0018	0018	0008	0014	0018
0011	0018	0018	000D	0010	001D
0007	0034	0018	0013	0031	001A
000C	0033	001A	0013	0031	001A
0003	0034	0016	0007	0034	0018
000C	0033	001A	0007	0034	0018

0006	000D	0019	0008	000F	0018
0011	0014	0014	0006	000D	0019
0009	0032	0039	0017	0031	003B
0003	0038	0018	0007	0038	0018
0003	0038	0018	0013	0035	001A
000C	0038	0016	0003	0038	0018
000C	0038	0016	0007	0038	0018

0004	0007	003A	000B	0002	0039
0017	0035	003D	000E	003B	003E

Application 2.

```
*(*)=*
MOTHER(MARY)=JANE
MOTHER(JANE)=ALICE
MOTHER(ALICE)=KATE
FATHER(ALICE)=TED
SISTER(JACK)=PAT
SON(TOM)=BILL
SON(TOM)=JACK
BROTHE(JACK)=TERRY
BROTHE(JACK)=JOHN
FATHER(ALFRED)=TOM
BROTHE(ALFRED)=PAUL
BROTHE(ALFRED)=PETER
SISTER(JIM)=LUCY
SISTER(JIM)=ANN
FATHER(JIM)=JOHN
FATHER(JACK)=BILL
MOTHER(JIM)=MARY
MOTHER(JACK)=JILL

*(jim)=*
SISTER(JIM)=LUCY
SISTER(JIM)=ANN
FATHES(JIM)=JOHN
MOTHER(JIM)=MARY

father(*)=*
FATHER(ALICE)=TED
FATHER(ALFRED)=TOM
FATHER(JIM)=JOHN
FATHER(JACK)=BILL

*(*)=jill
MOTHER(JACK)=JILL

sister(jack)=jane

sister(jack)=pat
SISTER(JACK)=PAT

*(*)=jill,*(*)=bill
SON(TOM)=BILL
FATHER(JACK)=BILL
MOTHER(JACK)=JILL

*(jim)=lucy
SISTER(JIM)=LUCY
```

APPENDIX B

FUNCTIONAL MEMORY TECHNIQUES APPLIED TO THE MICROPROGRAMMED CONTROL OF AN ASSOCIATIVE PROCESSOR

(published in the Conference Proceedings of the Second Annual Symposium on Computer Architecture, sponsored by IEEE Computer Society and ACM, University of Houston, 20-22 January 1975)

FUNCTIONAL MEMORY TECHNIQUES APPLIED TO THE MICROPROGRAMMED CONTROL OF AN ASSOCIATIVE PROCESSOR

C.V.W. Armstrong

Dept. of Computer Science, University of Edinburgh

Edinburgh, Scotland

Summary - Consideration is given to the separation of the data and control structures of a microprogrammed processor. The use of functional memory techniques to provide a suitable medium for containing and processing the control structure and giving one possible solution to this separation problem is described. The design considered is that of an associative processor but the techniques involved are applicable to other types of processors. Some of the advantages of this approach are given, together with their implications in the light of advances in microprocessor technology and cellular logic.

Introduction

The Control Structure

The microprogrammed control unit of a processor has the task of fetching and executing instructions. These two functions can be separated out and handled by different control units. Similarly, the execution unit can be divided into a section that processes the data and a section that processes the control structure related to that data. Separation of this sort is useful because each control section is specialised for a particular type of operation and can work in parallel with the operation of the other sections. It seems possible to separate out these control sections[1] and in fact the use of a microprogrammed control unit can go some way towards making this separation. However, when the control structure requires processing, the microprogrammed control unit may borrow resources provided primarily for the processing of the data structure and parallel operation is no longer possible.

Take for example the case of the Interdata Model 70 and related models, a minicomputer where vertical microprogramming predominates. Testing may require the use of both S and B busses and does not allow multiway branches to take place. This, for example, would be very useful in the processing of interrupts. The processing of the control structure preempts any processing of the data structure during microroutine execution. In fact, the processing of the control structure is constrained by the data processing architecture of the processor[2] A number of instructions have the same microroutines except for minor changes, which are restricted to small differences in the processing of the data structure. A means of separating out these two control sections may go some way in compacting the amount of space required to hold the differing microroutines into one where no such replications are necessary.

The attempt is made here to show how the use of functional memory techniques may allow a single microprogrammed control unit to process, in parallel, the data and control operations involved in executing instructions. This approach may be useful when a number of modules, such as microprocessors, must be controlled in parallel.

Functional Memory.

The term functional memory[3,4] was used to describe the use of a special type of memory to realise various combinational and sequential functions. A cellular structure was considered in which each cell could have three possible values : 0, 1 or X (corresponding to "don't care"). A rectangular array of these cells could perform searching operations on selected fields of rows of cells and data fields from selected rows could be ORed together during output. Functional memory had the advantages of regular circuitry such as RAMs and ROMs when implemented in LSI. The approach given here does not require customized LSI.

The Design of an Associative Processor.

An associative processor was chosen as a particular example of a subset of parallel processors, the subset of all single instruction stream - multiple data stream processors. The microprogrammed control of such a processor was studied. It is an example where the data structure is considerably different from the control structure and where control problems could be acute. Associative processors are examples where much of the cost of the machine is concentrated in the control unit and other supporting modules, and where improvements in control unit design can have a significant effect. In Goodyear's STARAN processor, an array of 256 associative elements uses 2,500 integrated circuits whilst the circuits necessary to control and support this array number 6,500.[5]

There are no fundamental associative properties in the processor which was developed. These associative properties are provided by the management of the processing elements by the microprogrammed control unit. The associative operations are emulated by standard arithmetic, logical and testing operations on 16-bit word-slices of a 256-bit word, as opposed to bit-slices. The control unit could alternatively emulate a different type of parallel processor in this subclass.

A hardware realization of an 8 processing element associative processor was built using the inventory of DEC RTM modules available in this Department, with the addition of a small number of other integrated circuits.

Architectural Decisions.

It was necessary to choose applications for which the associative processor could be used whilst its operation was being studied. It was decided to consider simple programs for:

(i) air traffic control conflict detection, and
(ii) information retrieval query processing.

Both these applications are suitable for associative processing. They also seemed complementary in their use of available operations of an associative processor.

The hardware realization was constrained by the inventory of DEC RTM modules and other circuits available, and this meant that the associative processor would have 8 processing elements with each element storing 256 bits as 16 words of 16 bits. Extensive processing capability was available at the processing element level.

The associative processor is interfaced to a sequential processor - an Interdata Model 74. This stores the program for the associative processor together with the data for its processing elements, which can only be accessed in sub-blocks. Thus, the sequential processor emulates the fetch section of the ideal associative processor being considered.

There would be I/O traffic of data for PEs during associative processing which would be reduced as the number of PEs increased. The point could be reached when all the data necessary for a particular search or data processing operation is loaded in one block.

The instruction stream is provided by the sequential processor and instructions modifying this stream are trapped by the sequential processor. If the modification is conditional, it is based on status information provided by the associative processor. The associative processor microprogrammed control unit can determine whether the status information in the sequential processor requires modification and transmits this new status, or whether the new status is for local control and can remain in the associative processor status register.

The following policies were pursued:-

(a) Apart from supplying the instructions and data, the sequential processor would be free to process programs in parallel with associative processor operation.
(b) An operation or design guideline would be adopted for the hardware realization only if it could be employed effectively or even more effectively in a larger (full-scale) associative processor.
(c) By careful choice of instructions, loops would be kept at the level of the microcode where they could be handled more efficiently, thus bringing the instruction stream as close to a sequential stream as possible. This is much easier to do in an associative processor, where iteration can be handled by exploiting parallelism.
(d) Delays due to modification of the instruction stream would be minimised by careful choice of the status information to be transmitted to the sequential processor and transmission would be well before this data is required for modifying the instruction stream.

The Data Processing Structure

The data processing structure consists of eight processing elements (PEs). Each PE has 16 16-bit words, a simple ALU, two accumulators, and a means of communicating with the MCU. It is a simple data processor which could be replaced by an LSI microprocessor.

In examples of associative processors such as STARAN, serial processing of a 256-bit word allows several fields of differing length starting at differing places to be processed. Here, processing is in parallel on 16-bit words, so the data fields must be integral values of 16-bits and aligned on a 16-bit word boundary. The STARAN example offers considerably more flexibility in the layout of data. However, the necessary control is correspondingly more complex and may require a lower level of microprogramming to achieve results such as the addition of two 16-bit fields.

The possible operations of a PE can be seen from the first part of Fig. 1.

As shown in Fig. 2, the PEs are connected in a ring structure, each being able to transmit a 16-bit word to one neighbour and receive a 16-bit word from its other neighbour. In addition, the microprogram control unit (MCU) can transmit a 16-bit word to all enabled PEs. The MCU can receive the ORing of 16-bit words transmitted from all the PEs enabled. The MCU can also receive an 8-bit status word from all the PEs enabled with each PE represented by a corresponding bit of the 8-bit word.

Note that this processing structure is a general purpose parallel processor with a simple interconnection pattern and that the associative properties are provided by the MCU which alternatively could emulate a different sort of parallel processor. The interconnection structure is easily modified in the hardware realization if this should prove necessary.

The Microprogram Control Unit.

The objective is to supply minimally encoded microinstructions to both the PEs and the MCU. Here, "minimally encoded" means that the microinstruction is decoded as much as economically possible with respect to the number of control lines and the corresponding number of pins on functionally organized LSI chips such as microprocessors, RAMs and ROMs. Thus, the size of the microinstruction wordlength is not considered a restraint in the ideal case.

The requirement, in this case, is that the MCU must perform a mapping from a user instruction of 16-bits to a variable number of 64-bit microinstructions where the first 24 bits are minimally encoded for use by the PEs (the data structure) and the second 40 bits are minimally encoded for use by the MCU (the control structure). Note that the MCU microorders are at a slightly higher level in order to reduce the number of bits required in the microinstruction.

The MCU is essentially as shown on Fig. 3. Each block named SPn or RPn represents a 16*16 scratchpad memory for the select phase or read phase, respectively. These will be called functional memories because of the role that they play in the design.

The basic interpretation cycle is as follows:- The select phase register breaks up the 16-bit word stored in it into 4 4-bit fields which address 4 16-bit words in the select phase functional memories. The 4 16-bit words so accessed are ORed together and form the 16-bit word stored in the read phase register. This word in turn is broken up into 4 4-bit fields and used to address the read phase functional memories. This provides the 64 bits of the microinstruction. The first 24 bits are used to control all the PEs enabled by the enable register. The second 40-bit field controls the MCU and either generates another 16-bit word for loading the select phase register or uses the next 16-bit user instruction to load it. The same interpretation cycle continues with the only variations allowed being the way in which the next 16-bit word for the select phase register is determined.

Fig. 1 shows the format that was used for the microinstruction in the design of this associative processor. Note that a few bits are as yet unused and may have their roles assigned later.

A number of other registers, mainly self-explanatory, are shown in Fig. 3. With proper timing, one register could hold the 16-bit word for both the select phase and the read phase. The user instruction or portions thereof can be loaded directly into the select phase register. In the DEC RTM realization, the microinstruction is used directly. If this was not possible due to synchronization problems then a 64-bit master-slave register could be used with the MCU and the PEs controlled by the master flipflops whilst the slave flipflops are being prepared with the next microinstruction.

A simple example of the use of this MCU is now given. Consider that the functional memories are loaded as shown in Fig. 4. We consider the example of the user instruction 0001 0010 0111 0001 to read in a variable amount of data into a variable number of PEs. Now the user instruction is in general broken into 4 4-bit fields as follows:-

(1) First Field - the instruction operation code. For example - 0001 - read in a variable amount of data into a variable number of PEs.
(2) Second Field - any necessary information for the operation and the PEs, including data or information where data is to be found (such as data or address in the next 16-bit word of the instruction stream). For example - 0010 - load the first (2+1) scratchpad words of each PE.
(3) Third Field - any necessary information for the MCU. For example - 0111 - load (7+1) PEs consecutively. This field is stored in the X register of the MCU.
(4) Fourth Field - the control level - corresponds to a particular microprogram. This is inspected by the MCU to cause switching from one set of functional memories to another and/or loading or preloading of functional memories. For example - 0001 - the control level 1 microprogram.

Operation commences as follows:-

The fourth field is used for checking that the appropriate microprogram is loaded. The first field is loaded into the first 4-bit field of the select phase register and the second field into the second field of the select phase register. The third field is loaded into the X register of the MCU. Thus, initially the select phase register is
0001 0010 0000 0000
Only SP1 and SP2 are used during the initial user instruction interpretation. This causes the read phase register to become
0010 0000 1101 0000
Note that the one's complement of 0010 (the count of the number of words to load into a PE) has been formed in the third field. The first microinstruction is output (see Fig. 1 for the operations) and the select phase register is now loaded directly from this microinstruction with
0001 1111 1101 0000
Note that a particular bit of the microinstruction causes the first field of the select register to be loaded from the first field of the instruction. The next read phase register contents are
0010 0001 1110 0001
Note that the second field has been incremented to 1 to access the next word in the PE and that the fourth field also has this value, which will be used in the next cycle for incrementing the second field again. Note that the third field has been incremented. The test whether the right number of words have been input to a PE has been made implicit because when the third field reaches 1111, it will cause RP3 to become all zero. This will cause the user instruction to be used in loading the select phase register as described above. If the third field of the user instruction as stored in the X register of the MCU is non-zero, its value is decremented and the same instruction used again. Otherwise, the next instruction is used.

The following points need to be made about this microroutine example:-
(1) The functional memories have enough space for more instructions than was shown above. The corresponding write instruction would be
0010 0010 0111 0001
There is space for 14 other instructions which require the performing of a particular operation sequentially on a variable number of 16-bit fields of a variable number of PEs. There are many other possible methods of using the MCU of which only one example has been given here. Space restrictions do not permit a detailed description of the other microprograms for parallel and associative operations. In the above example, the minimum amount of interaction between select phase functional memories in the generation of the read phase functional memories data has taken place. It is possible for two select phase functional memories to contribute alternate bits to a 4-bit field of the read-phase register and thus allow multiway branching within the one interpretation cycle. This could prove useful in instructions where considerable decoding and decision making was being employed.
(2) Although primarily designed for associative processing, the MCU is suitable for other types of single instruction stream - multiple data stream processing. Possibly this would require some extension of the interconnection pattern considered here.
(3) In addition, it is possible to pick different modes of interpretation by suitable use of the mode of interpretation field in the microinstruction. For example, in the mode of interpretation considered above, the select phase register is loaded from the previous microinstruction except in the case when RP3 is zero. In this case, the previous instruction is used again if the X register of the MCU is non-zero, otherwise the next user instruction is loaded.

Another mode tests the activate bit in the micro-instruction and ORs the value in the X register into the SPR before the execution of the next stage of interpretation. Other modes could make direct use of the status bits in the loading of the select phase register, thus allowing much of the branching and iteration to remain at the level of the microprogram. Although there is much repetition in particular fields of the above microprogram example, which is provided for explanatory purposes, the fact remains that another microprogram when loaded can use these same fields for entirely different control purposes and with other forms of repetition. See (vi) below. Note that the incorporation of the mode of interpretation as a field in the microinstruction allows a microroutine to change the mode dynamically during user instruction interpretation.
(4) Four different microprograms are now available and can be loaded corresponding to the four possible control levels specified in the last field of the user instruction. (Sixteen possible microprograms could be specified). There would be a delay during loading. Since a user instruction never requires the use of more than one of the microprograms, if two function memory "units" were available, one could be loaded whilst the other was being used. This is especially practical since the instruction stream is close to a sequential stream, and the fetch control unit could look ahead to the last 4-bit field of the next instruction. The second functional memory unit could be loaded by looking ahead for the first instruction which does not use the current microprogram and starting to load the required microprogram in parallel with the rest of the MCU operation. When this instruction is reached, the MCU switches over to the second functional memory unit and the above process can be applied to the first functional memory unit. Having three functional memory units would take care of the worst case of user instruction branching with minimum delay. With a judicious choice of the instruction classes and the corresponding microprograms for these classes, the amount of reloading can be decreased.[6] By increasing the permitted size and number of functional memories, this problem could be eliminated altogether.

The use of the above microprogramming technique has the following additional advantages:-
(i) This form of microprogramming does not seem to be any more difficult than the microprogramming schemes of many existing machines, especially when a simulator is available. It has the advantage of postponing until late in the design process, the microprogramming stage, the specification of the control structure and the operations on it. User microprogramming although possible is not the main objective of this approach.
(ii) In considering the firmware-software interaction, this approach allows much of the software to be concentrated in the firmware. The user writes instructions at a variety of levels corresponding to the control level specified in the instruction. At present, the microprograms are used as follows:-
(1) control level 0 - resetting, load, store, shifts, tests of PE condition codes, inter-PE communication
(2) control level 1 - input and output
(3) control level 2 - addition and subtraction with provision for multiple precision arithmetic, multiplication
(4) control level 3 - general associative operations - the first field of the instruction gives the type of test. The second field gives the scratchpad location on which the test is to take place. The third field directs what will be done with the result of the test.

The user can write programs using the higher level instructions (such as the general I/O instruction and the general associative operation) descending to the lower level instructions only when the final elements of control are unavailable higher up the hierarchy (such as instructions that set particular bits in the enable register). This leads to more compact code and faster operation.
(iii) In considering the hardware-firmware interaction, many of the simple flags and small field operations that a control unit may use, such as condition and emit fields, are eliminated completely by using this particular firmware approach. In particular, tests, multiway branches, complementation, incrementing, shifting, masking and reformatting can take place implicitly. Key variables or bits of instructions can be stored and used when necessary without requiring any additional emit fields, flags or registers. Apart from the functional memories, the barest minimum of extra circuitry and registers are required.
(iv) Fault recovery can be facilitated if this is a critical factor. A special microprogram could test all the PEs in parallel, and also test the MCU. The isolation of the malfunctioning PEs can be accomplished by microprogrammed control of the enable register. Since the MCU consists mainly of identical functional memories, a spare can be switched in if one should fail. The only critical circuitry requiring consideration are the small number of MCU registers and the combinational networks. The necessary redundancy is therefore limited to these circuits and a small number of PEs and functional memories. This would be a negligible cost for a large associative processor.
(v) There is a minimum delay in processing the data structure, since by the time the data structure has been acted upon by a microinstruction, the MCU has cycled back in parallel and produced the next microinstruction. Status information when required need never come from the current microinstruction execution. This holds true even when a new user instruction interpretation has commenced. In particular, there is no delay when a test has to be made at the microprogram level, where hopefully the majority of all branches will take place.
(vi) Many instructions can use the same microinstruction fields (4 16-bit fields) and only the fields which are different require placement in a word of the read phase functional memories. The other existing fields can be used unchanged. This is one answer to the field combination problem.[4]

Microprogrammed Processors.

The previous section has attempted to show how it is possible to gain an increase in the capability of containing and processing the control structure of a processor by the addition of a minimum amount of extra complexity in the microprogram control unit. This was achieved by:-
(i) breaking up decoders for the functional memory units (or control memory in the conventional case) into a number of separate decoders,
(ii) breaking up the read phase of the control memory into a select phase and then a read phase, and
(iii) performing an ORing of the output from the select phase functional memories. This requires no additional circuitry when negative logic and open-collector drivers are used as in the DEC RTM TTL implementation.

Thus, this is one possible solution to the problem of designing processors with separate data and control structures. It would be interesting to consider this approach in different types of processors.

Microprocessors

In the design of the associative processor, each processing element was a 16-bit parallel processor with a limited number of possible low level operations. This approach was used because of availability, speed and simplicity. It simplified the microprogramming and did not require the lower level of control that a serial processor would require.

Consider however the case where a processing element is replaced with a microprocessor. This could still provide enough processing power and storage per chip if present trends continue. If microprocessors cost $20-$30 each in large quantities, then a 1K processing element associative processor would have a hardware cost of $20,000-$30,000 plus the cost of the control and supporting equipment. The fact that suitable microprocessors are available and that a possible MCU uses standard RAMs, ROMs and MSI logic circuitry means that the development costs are reduced. For example, the RCA COSMAC LSI microprocessor is very similar to the PE considered here.

Cellular Logic.

Looking further into the future and considering the continuum described by Weinberger[7], by increasing the number of decoders further, the point is reached when a decoder is addressing one of two possible output lines. The functional memory has then reached the cellular complexity of those originally considered by Flinders, et al[3]. Thus, it is possible to envisage the same sorts of operations considered here, in a cellular logic implementation with all of the many advantages covered by Kautz[8].

Acknowledgement : I am indebted to Peter Gardner for mentioning how 16 word memories could be considered as an extension of functional memories.

REFERENCES

(1) C.G. Bell and J. Grason. The Register Transfer Module Design Concept. Computer Design, May 1971, pp.87-94.
(2) P.C. Anagnostopoulos, M.J. Michel, G.H. Sockut, G.M. Stabler and A. van Dam. Computer Architecture and Instruction Set Design. AFIPS 1973 National Computer Conference Proceedings, Vol 42, pp 519-527.
(3) M. Flinders, P.L. Gardner, R.J. Llewelyn and J.S. Minshull. Functional Memory as a General Purpose Systems Technology. Technical Report T.R. 12.088, IBM United Kingdom Laboratories Ltd, Hursley Park, Winchester Hampshire, July 1970.
(4) P.L. Gardner. Functional Memory and its Microprogramming Implications. IEEE Trans. on Comp., Vol. C-20, No. 7 (July 1971) pp.764-775
(5) J.D. Feldman and O.A. Reimann. RADCAP : An Operational Parallel Processing Facility. AFIPS 1974 National Computer Conference Proceedings, Vol. 43, pp. 7-15.
(6) C.C. Foster and R. Gonter. Conditional Interpretation of Operation Codes. IEEE Trans. on Comp, Vol. C-20, No. 2, (January 1971), pp. 108-111.
(7) A. Weinberger. Hybrid Associative Memory Concept. Computer Design, January 1971.
(8) W.H. Kautz. Cellular Logic-in-Memory Arrays. Trans. on Comp., Vol. C-18, No. 8, (August 1969), pp. 719-727.

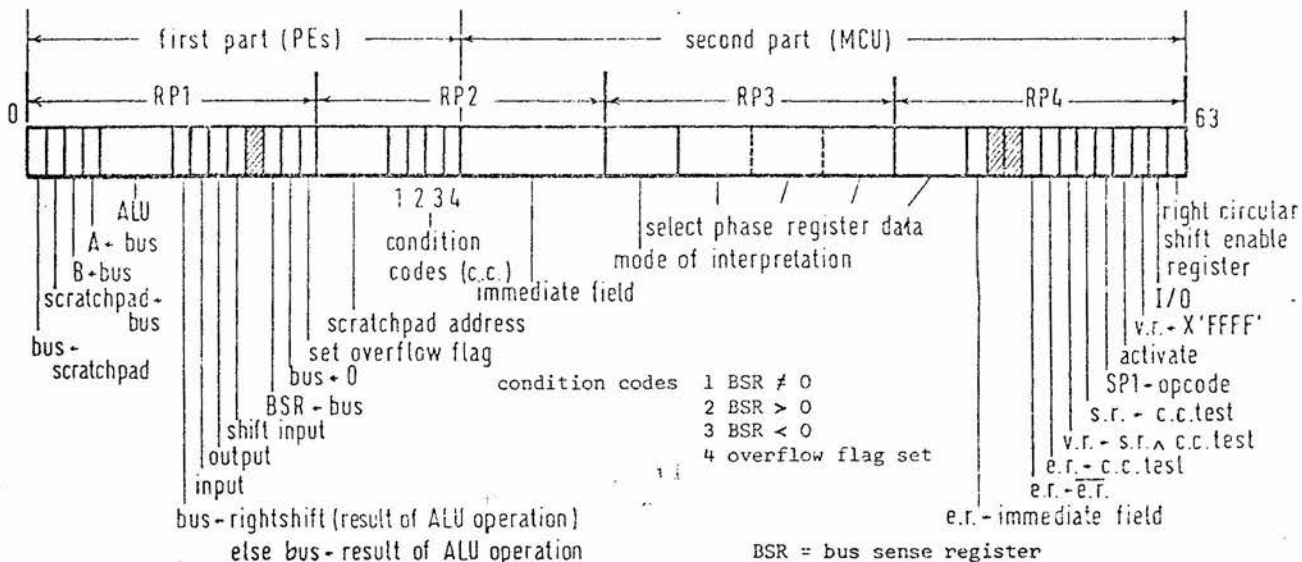
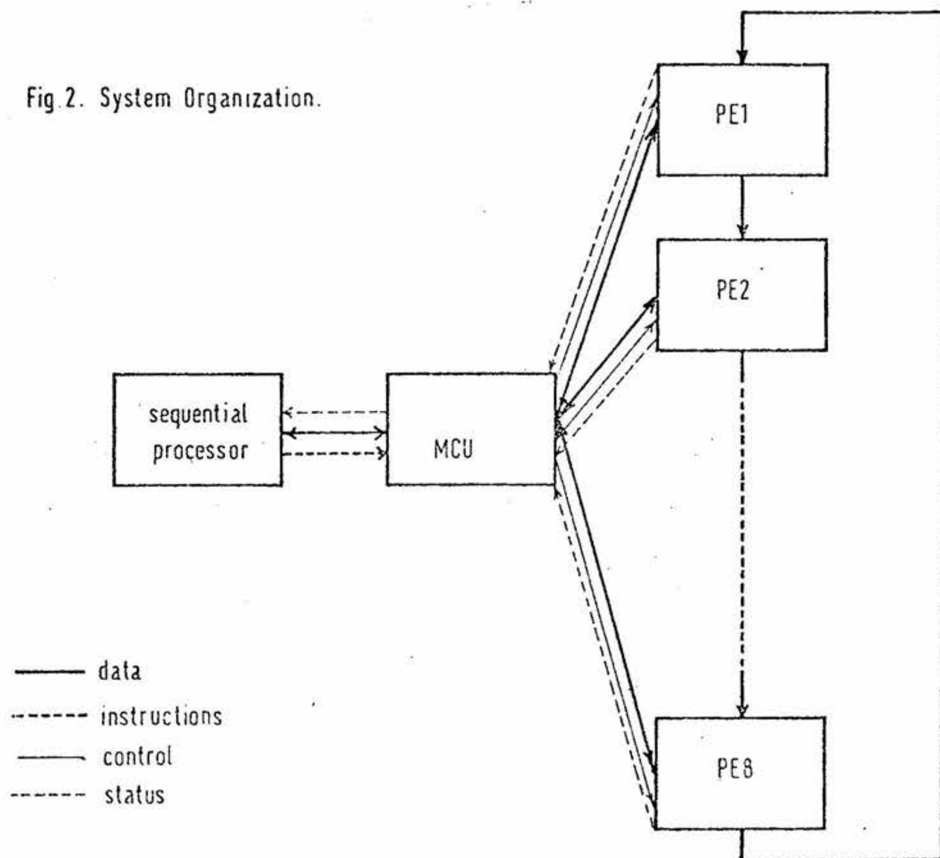


Fig.1. Microinstruction.

Fig.2. System Organization.



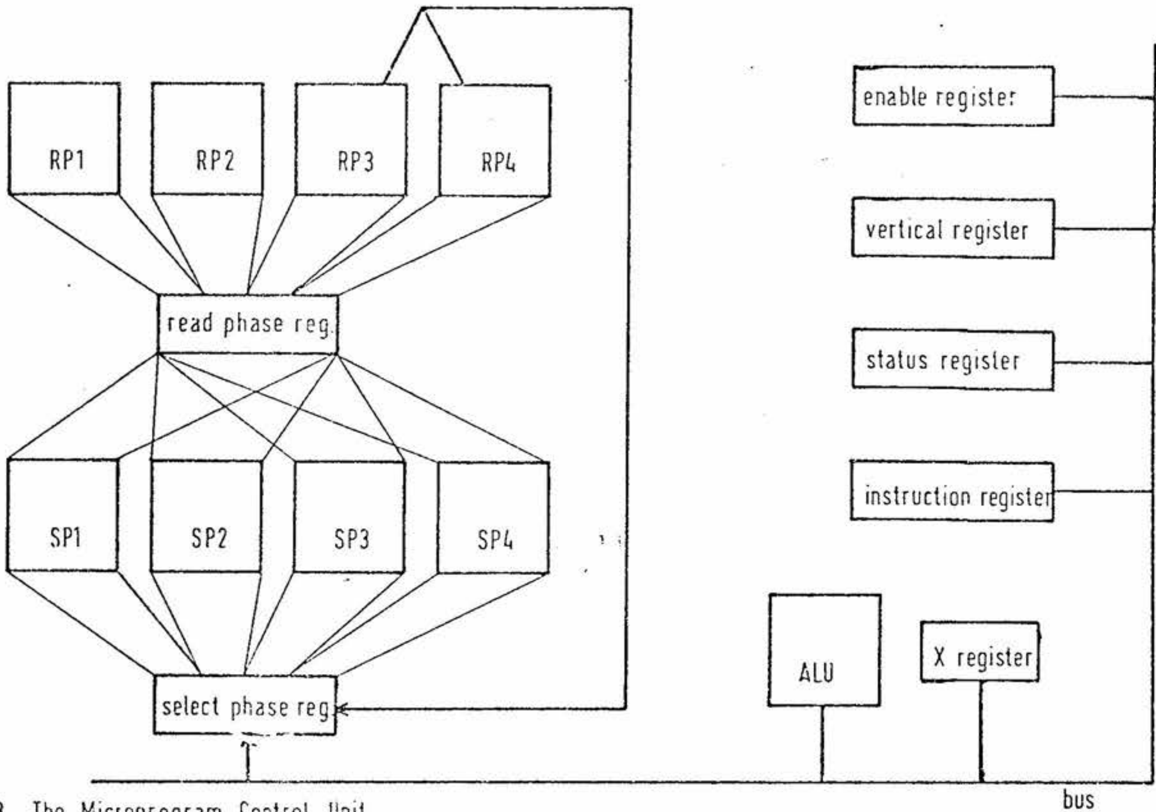


Fig.3. The Microprogram Control Unit

Fig.4. Microprogram (for I/O)

	SP1	SP2	SP3	SP4
0	0000000000000000	0010000011110000	0000000000010000	0000000100000001
1	0010000000000000	0000000011100000	0000000000100000	0000001000000010
2	0001000000000000	0000000011010000	0000000000110000	0000001100000011
3	0000000000000000	0000000011000000	0000000001000000	0000010000000100
4	0000000000000000	0000000010110000	0000000001010000	0000010100000101
5	0000000000000000	0000000010100000	0000000001100000	0000011000000110
6	0000000000000000	0000000010010000	0000000001110000	0000011100000111
7	0000000000000000	0000000010000000	0000000010000000	0000100000001000
8	0000000000000000	0000000001110000	0000000010010000	0000100100001001
9	0000000000000000	0000000001100000	0000000010100000	0000101000001010
10	0000000000000000	0000000001010000	0000000010110000	0000101100001011
11	0000000000000000	0000000001000000	0000000011000000	0000110000001100
12	0000000000000000	0000000000110000	0000000011010000	0000110100001101
13	0000000000000000	0000000000100000	0000000011100000	0000111000001110
14	0000000000000000	0000000000010000	0010000011110000	0000111100001111
15	0000000000000000	0000000000000000	0010000011111111	0000000000000000

	RP1	RP2	RP3	RP4
0	0000000000000000	0000000000000000	0000000011110000	0000000000010011
1	1000000000100000	0001000000000000	0000000011110001	0001000000010010
2	0100000001000000	0010000000000000	0000000011110010	0010000000010010
3	0000011100100000	0011000000000000	0000000011110011	0011000000010010
4	0000000000000000	0100000000000000	0000000011110100	0100000000010010
5	0000000000000000	0101000000000000	0000000011110101	0101000000010010
6	0000000000000000	0110000000000000	0000000011110110	0110000000010010
7	0000000000000000	0111000000000000	0000000011110111	0111000000010010
8	0000000000000000	1000000000000000	0000000011111000	1000000000010010
9	0000000000000000	1001000000000000	0000000011111001	1001000000010010
10	0000000000000000	1010000000000000	0000000011111010	1010000000010010
11	0000000000000000	1011000000000000	0000000011111011	1011000000010010
12	0000000000000000	1100000000000000	0000000011111100	1100000000010010
13	0000000000000000	1101000000000000	0000000011111101	1101000000010010
14	0000000000000000	1110000000000000	0000000011111110	1110000000010010
15	0000000000000000	1111000000000000	0000000000000000	1111000000010010

References

The references are given in alphabetic order of the author's name and by date of publication.

A.M.Abd-Alla and D.C.Karlgaard, Heuristic Synthesis of Microprogrammed Computer Architecture, IEEE Trans. Comp., Vol.C-23, No.8, August 1974, pp.802-807.

T.Agerwala, Microprogram Optimization : A Survey, IEEE Trans. Comp., Vol.c-25, No.10, October 1976, pp. 962-973.

A.K.Agrawala and T.G.Rauscher, Microprogramming : Perspective and Status, IEEE Trans. Comp., Vol.C-23, No.8, August 1974, pp.817-837.

A.K.Agrawala and T.G.Rauscher, Foundations of Microprogramming. Architecture, Software and Applications, Academic Press, Inc., New York, 1976. ACM Monograph Series.

M.W.Allen and T.Pearcy, Developments in Machine Architecture, Proc. Fourth Australian Computer Conference, Adelaide, South Australia, Griffin Press, 1969, pp.227-230.

P.C.Anagnostopoulos, M.J.Michel, G.H.Sockut, G.M.Stabler and A. van Dam, Computer Architecture and Instruction Set Design, AFIPS, NCC, 1973, pp.519-527.

C.V.W.Armstrong, Functional Memory Techniques Applied to the Microprogrammed Control of an Associative Processor, Second Annual Symposium on Computer Architecture, University of Houston, January 20-22, 1975, pp.34-40.

W.L.Ash and E.H.Sibley, TRAMP : An interpretative Associative Processor with Deductive Capabilities, 1968 ACM National Conference, pp.143-156.

D.Aspinall, D.J.Kinniment, D.B.G.Edwards, Associative Memories in Large Computer Systems, Proc. IFIP Congress, 1968, Hardware 1, Booklet D, pp.81-85.

G.H.Barnes, R.M.Brown, M.Kato, D.J.Luck, D.L.Slotnick and R.A.Stokes, The ILLIAC IV Computer, IEEE Trans. Comp., Vol.C-17, No.8, August 1968, pp.746-757.

K.E.Batcher, Flexible Parallel Processing and STARAN, Western Electronic Show and Convention (WESCON), 19-22 September 1972, Los Angeles, California, Session 1.

K.E.Batcher, STARAN/RADCAP Hardware Architecture, 1973 Sagamore Computer Conference, August 22-24, 1973, Sagamore, N.Y.

K.E.Batcher, STARAN Parallel Processor System Hardware, AFIPS, NCC, 1974, pp.405-410.

K.E.Batcher, The Multi-Dimensional Access Memory in STARAN, Paper submitted for publication in the IEEE Trans. Comp. Special Issue on Parallel Processing.

C.G.Bell and J.Grason, The Register Transfer Module Design Concept, Computer Design, May 1971, pp.87-94.

C.G.Bell, J.Grason and A.Newell, Designing Computers and Digital Systems Using PDP16 Register Transfer Modules, Digital Equipment Corporation, Digital Press, 1972.

R.O.Berg and M.D.Johnson, An Associative Memory for Executive Control Fuctions in an Advanced Avionics Computer System, Proc. COMPCON 1970, pp.336-342.

H.Berndt, Functional Microprogramming as a Logic Design Aid, IEEE Trans. Comp., Vol.C-19, No.10, October 1970, pp.902-907.

P.B.Berra, Some Problems in Associative Processor Applications to Data Base Management, AFIPS, NCC, 1974, pp.1-5.

D.P.Bhandarkar and S.H.Fuller, Markov Chain Models for Analyzing Memory Interference in Multiprocessor Computer Systems, First Annual Symposium on Computer Architecture, University of Florida, December 9-11, 1974, pp.1-6.

B.R.Borgerson, The Viability of Multiprocessor Systems, Computer, Vol.9, No.1, January 1976, pp.26-30.

G.F.Casaglia, Nanoprogramming vs. Microprogramming, Computer, Vol.9, No.1, January 1976, pp.54-58.

W.A.Clark et al, Macromodular Computer Systems, (series of papers) AFIPS, SJCC, 1967, pp.337-401.

S.H.Dalrymple, A Parametric Associative Memory Emulation, SIGMICRO Newsletter, Vol.6, No.3, September 1975, pp.10-36.

S.Dasgupta and J.Tartar, The Identification of Maximal Parallelism in Straight-Line Microprograms, IEEE Trans. Comp., Vol.C-25, No.10, October 1976, pp.986-992.

P.M.Davies, Readings in Microprogramming, IBM Systems Journal, Vol.11, No.1, 1972, pp.16-40.

E.W.Davis, STARAN/RADCAP System Software, 1973 Sagamore Computer Conference, August 22-24, 1973, Sagamore, N.Y.

E.W.Davis, STARAN Parallel Processor System Software, AFIPS, NCC, 1974, pp.17-22.

B.Dejka, Projections of Microcomputer Usage, Workshop on Microprocessor Architecture and Systems, Northwestern University, May 6-7, 1976. (Referenced by M.J.Gonzalez, Jr., Microprocessor Architecture and Systems, Computer, September 1976, pp.49-51.)

B.G.Dietzler, Knoxville Associative Processor Evaluation Report, Document No. PX 6406, Univac Defense Systems Division, St. Paul, Minn., October 1972.

W.E.Donath, Equivalence of Memory to "Random Logic", IBM Journ. Res. & Dev., Vol.18, No.5, September 1974, pp.401-407.

H.R.Downs, Aircraft Conflict Detection in an Associative Processor, AFIPS, NCC, 1973, pp.177-180.

J.A.Dugan, R.S.Green, J.Minker and W.E.Shindle, A Study of the Utility of Associative Memory Processors, Proc. ACM National Meeting, 1966, pp.347-360.

J.D.Feldman and L.C.Fulmer, RADCAP - An Operational Parallel Processing Facility, AFIPS, NCC, 1974, pp.7-15.

J.D.Feldman and O.A.Reimann, RADCAP : An Operational Parallel Processing Facility, 1973 Sagamore Computer Conference, August 22-24, 1973, Sagamore, N.Y.

H.Fleisher and L.I.Maissel, An Introduction to Array Logic, IBM Journ. Res. & Dev., March 1975, pp.98-109.

H.Fleisher, A.Weinberger and V.D.Winkler, The Writeable Personalized Chip, Computer Design, Vol.9, No.6, June 1970, p.59.

M.Flinders, P.L.Gardner, M.H.Hallett, J.W.Jones, J.F.Minshull and K.G.Taylor, Error Detection Circuitry, British Patent No. 1,265,013, 24 April 1969 (1969a).

M.Flinders, P.L.Gardner, M.H.Hallett, J.W.Jones and J.F.Minshull, Digital Data Storage Systems, British Patent No. 1,265,014, 22 October 1969 (1969b).

M.Flinders, P.L.Gardner, R.J.Llewelyn and J.S.Minshull, Functional Memory as a General Purpose Systems Technology, Technical Report T.R.12.088, IBM United Kingdom Laboratories Ltd., July 1970. (Also Proc. COMPCON 72, pp.314-324.)

M.Flinders, D.J.Craft, K.G.Taylor, M.H.Hallett, F.T.Moth, J.W.Jones, J.F.Minshull, R.J.Llewelyn, P.L.Gardner and W.A.Christopherson, Data Handling Systems, British Patent No. 1,317,714, 28 January 1971.

M.J.Flynn, Some Computer Organizations and Their Effectiveness, IEEE Trans. Comp., Vol.C-21, No.9, September 1972, pp.948-960. (1972a)

M.J.Flynn, Toward More Efficient Computer Organizations, AFIPS, SJCC, 1972, pp.1211-1217. (1972b)

M.J.Flynn and R.F.Rosin, Microprogramming : An introduction and a Viewpoint, IEEE Trans. Comp., July 1971, pp.727-731.

C.C.Foster, Determination of Priority in Associative Memories, IEEE Trans. Comp., Vol.C-18, No.8, August 1968, pp.788-789.

C.C.Foster and R.Gonter, Conditional Interpretation of Operation Codes, IEEE Trans. Comp., January 1971, pp.108-111.

S.H.Fuller, V.R.Lesser, C.G.Bell and C.H.Kaman, The Effects of Emerging Technology and Emulation Requirements on Microprogramming, IEEE Trans. Comp., Vol.C-25, No.10, October 1976, pp.1000-1009.

S.H.Fuller and G.A.Mathew, Implementing Microprogram Storage with PLA's, Computer Architecture News, Vol.5, No.2, June 1976, pp.6-11.

L.C.Fulmer and W.C.Meilander, A Modular Plated-Wire Associative Processor, Proc. COMPCON 1970, pp.325-335.

P.L.Gardner, Functional Memory and its Microprogramming Implications, IEEE Trans. Comp., Vol.C-20, No.7, July 1971, pp.764-775.

D.C.Gunderson and W.L.Heimerdinger, A Multiprocessor with Associative Control, in "Prospects for Simulation and Simulators of Dynamic Systems" edited by G.Shapiro and M.Rogers, Spartan Books, New York, 1967.

A.G.Hanlon, Content-Addressable and Associative Memory Systems. A SURVEY, IEEE Trans. Elect. Comp., Vol.EC-15, No.4, August 1966, pp.509-521.

T.Jayasri and D.Basu, An Approach to Organizing Microinstructions Which Minimizes the Width of Control Store Words, IEEE Trans. Comp., Vol.C-25, No.5, May 1976, pp.514-521.

L.R.Johnson and M.H.McAndrew, On Ordered Retrieval from an Associative Memory, IBM Journal, April 1964, pp.189-193.

J.W.Jones, Array Logic Macros, IBM Journ. Res. & Dev., Vol.19, March 1975, pp.120-126.

L.H.Jones and R.E.Merwin, Trends in Microprogramming : A Second Reading, IEEE Trans. Comp., Vol.C-23, No.8, August 1974, pp.754-759.

J.E.Juliussen and F.J.Mowle, Multiple Microprocessors with Common Main and Control Memories, IEEE Trans. Comp., Vol.C-22, No.11, November 1973, pp.999-1007.

J.R.Jump, Asynchronous Control Arrays, IEEE Trans. Comp., Vol.C-23, No.10, October 1974, pp.1020-1029.

J.R.Jump and D.R.Fritsche, Microprogrammed Arrays, IEEE Trans. Comp., Vol.C-21, No.9, September 1972, pp.974-983.

W.H.Kautz, Testing for Faults in Combinational Cellular Logic Arrays, Proc. 8th Annual Symposium on Switching and Automata Theory, October 1967, pp.161-174.

W.H.Kautz, Fault Testing and Diagnosis in Combinational Digital Circuits, IEEE Trans. Comp., April 1968, pp.352-366.

W.H.Kautz, Cellular Logic-in-Memory Arrays, IEEE Trans. Comp., Vol.C-18, No.8, August 1969, pp.719-727.

W.H.Kautz, An Augmented Content-Addressed Memory Array for Implementation with Large-Scale Integration, Journ. ACM, Vol.18, No.1, January 1971, pp.19-33. (1971a)

W.H.Kautz, Programmable Cellular Logic, in "Recent Developments in Switching Theory", Academic Press, Inc., New York, 1971, Chap. IX, pp.369-422. (1971b)

W.H.Kautz, K.N.Levitt and A.Waksman, Cellular Interconnection Arrays, IEEE Trans. Comp., Vol.C-17, No.5, May 1968, pp.443-451.

W.H.Kautz and M.C.Pease III, Cellular Logic-in-Memory Arrays, Stanford Research Institute, Final Report - Part II, Office of Naval Research, Department of the Navy, Contract N00014-C-0404, SRI Project 5509, November 1971.

D.H.Lawrie, Access and Alignment of Data in an Array Processor, IEEE Trans. Comp., Vol.C-24, No.12, December 1975, pp.1145-1155.

H.W.Lawson, Jr. and B.Magnhagen, Advantages of Structured Hardware, Second Annual Symposium on Computer Architecture, University of Houston, January 20-22, 1975, pp.152-158.

M.M.Lehman, Microprogramming Trend Considered Dangerous, SIGMICRO Newsletter, Vol.6, No.3, September 1975, pp.37-39.

K.N.Levitt and W.H.Kautz, Cellular Arrays for the Parallel Implementation of Binary Error-Correcting Codes, IEEE Trans. Information Theory, Vol.IT-15, No.5, September 1969, pp.597-607.

K.N.Levitt and W.H.Kautz, Cellular Arrays for the Solution of Graph Problems, Comm. ACM, Vol.15, No.9, September 1972, pp.789-801.

P.J.Lindsay, A Simple Asynchronous Interface for Linking Small Computers, DECUS Europe, Eighth Seminar Proceedings 1972, pp.253-256.

P.J.Lindsay, A Link Coaxial Line Standard, University of Edinburgh Department of Computer Science Internal Memorandum, 29th March 1974, Edinburgh, Scotland.

J.C.Logue, N.F.Brickman, F.Howley, J.W.Jones and W.W.Wu, Hardware Implementation of a Small System in Programmable Logic Arrays, IBM Journ. Res. & Dev., Vol.19, March 1975, pp.110-119.

H.H.Love and D.A.Savitt, The Association-Storing Processor, in "Prospects for Simulation and Simulators of Dynamic Systems" edited by G.Shapiro and M.Rogers, Spartan Books, New York, 1967, pp.165-182.

O.E.Marvel, HAPPE Honeywell Associative Parallel Processing Ensemble, First Annual Symposium on Computer Architecture, University of Florida, December 9-11, 1974, pp.261-267.

J.E.Minshull and A.S.Murphy, Associative Search Structure, IBM Technical Disclosures Bulletin, Vol.15, No.3, August 1972, pp.1002-1004.

R.Moulder, An Implementation of a Data Management System on an Associative Processor, AFIPS, NCC, 1973, pp.171-176.

A.Mukhopadhyay and H.Stone, Cellular Logic, in "Recent Developments in Switching Theory", Academic Press, Inc., New York, 1971, Ch.VII, pp.255-313.

B.Parhami, A Highly Parallel Computing System for Information Retrieval, AFIPS, FJCC, 1972, pp.681-690.

B.Parhami, Associative Memories and Processors : An Overview and Selected Bibliography, Proc. IEEE, Vol.61, No.6, June 1973, pp.722-730.

B.Parhami and A.Avizienis, Design of Fault-Tolerant Associative Processors, First Annual Symposium on Computer Architecture, University of Florida, December 9-11, 1974, pp.141-145. (1974a)

B.Parhami and A.Avizienis, A Study of Fault Tolerance Techniques for Associative Processors, AFIPS, NCC, 1974, pp.643-652. (1974b)

D.A.Patterson, Strum : Structured Microprogram Development System for Correct Firmware, IEEE Trans. Comp., Vol.C-25, No.10, October 1976, pp.974-985.

C.H.Radoy and G.J.Lipovski, Switched Multiple Instruction, Multiple Data Stream Processing, Second Annual Symposium on Computer Architecture, January 20-22, 1975, pp.183-187.

C.V.Ramamoorthy and M.Tsuchiya, A High-Level Language for Horizontal Microprogramming, IEEE Trans. Comp., Vol.C-23, No.8, August 1974, pp.791-801.

T.G.Rauscher, Microprogramming the AN/UYK-17(XB-1)(V) Signal Processing Element Arithmetic Unit, SIGMICRO Newsletter, Vol.5, No.2, July 1974, pp.29-63.

R.F.Rosin, An Organization of an Associative Cryogenic Computer, AFIPS, SJCC, 1962, pp.203-207.

G.E.Rossman, Review 29,830, ACM Computing Reviews, Vol.17, No.4, April 1976, p.166. On W.E.Donath, "Equivalence of Memory to 'Random Logic'", IBM Journ. Res. & Dev., Vol.18, No.5, September 1974, pp.401-407.

J.A.Rudolph, A Production Implementation of an Associative Processor : STARAN, AFIPS, FJCC, 1972, pp.229-241.

B.C.Searle and D.E.Freberg, Tutorial : Microprocessor Applications in Multiple Processor Systems, Computer, Vol.8, No.10, October 1975, pp.22-30.

E.H.Sibley, R.W.Taylor and D.G.Gordon, Graphical Systems Communication : An Associative Memory Approach, AFIPS, FJCC, 1968, pp.545-555.

D.L.Slotnick, Logic Per Track Devices, in "Advances in Computers", Vol.10, Academic Press, New York, 1970, pp.291-296.

D.C.Snyder, Computer Performance Improvement by Measurement and Microprogramming, Hewlett-Packard Journal, February 2, 1975. (Reprinted in SIGMICRO Newsletter, Vol.6, No.1, April 1975, pp.7-14.)

STARAN APPLE Programming Manual, GER-15637A, Goodyear Aerospace Corporation, September 1973.

P.Sylvain and M.Vineberg, The Design and Evaluation of the Array Machine : A High-Level Language Processor, Second Annual Symposium on Computer Architecture, University of Houston, January 20-22, 1975, pp.119-125.

R.T.Thomas, Organization for Execution of User Microprograms from Main Memory : Synthesis and Analysis, IEEE Trans. Comp., Vol.C-23, No.8, August 1974, pp.783-790.

K.J.Thurber, An Associative Processor for Air Traffic Control, AFIPS, SJCC, 1971, pp.49-59.

K.J.Thurber, A Systematic Approach to the Design of Digital Bussing Structures, AFIPS, FJCC, 1972, Vol.41, pp.719-740.

K.J.Thurber and R.O.Berg, Applications of Associative Processors, Computer Design, November 1971, pp.103-110.

K.J.Thurber and L.D.Wald, Associative and Parallel Processors, ACM Computing Surveys, Vol.7, No.4, December 1975, pp.215-255.

G.Tjaden, MIMD Multimicroprocessors as Main Frame Replacements, Workshop on Microprocessor Architecture and Systems, Northwestern University, May 6-7, 1976. (Referenced by M.J.Gonzalez, Jr., "Microprocessor Architecture and Systems", Computer, September 1976, pp.49-51.)

M.Tsuchiya and M.J.Gonzalez, Towards Optimization of Horizontal Microprograms, IEEE Trans. Comp., Vol.C-25, No.10, October 1976, pp.992-999.

H.Weber, A Microprogrammed Implementation of EULER on IBM System/360 Model 30, Comm.ACM, Vol.10, No.9, September 1967, pp.549-558.

A.Weinberger, Hybrid Associative Memory, IBM Technical Disclosures, Vol.11, No.12, May 1969, pp.1744-1745.

A.Weinberger, The Hybrid Associative Memory Concept, Computer Design, January 1971.

A.Weinberger, Hybrid Associative Memory, U.S.Patent 3,644,906, February 22, 1972.

A.Weinberger, Functional Memory Using Multi-State Associative Cells, U.S.Patent 3,761,902, September 25, 1973.

A.Weinberger, H.Fleisher, and V.Winkler, The Writeable Personalized Chip, Computer Design, No.6,59, June 1970.

A.Weinberger, H.Fleisher and V.Winkler, Partitioning Logic Operations in a Generalized Matrix System, U.S.Patent 3,593,317, July 13, 1971.

W.T.Wilner, Burroughs B1700 Memory Utilization, AFIPS, FJCC, 1972, pp.579-586. (1972a)

W.T.Wilner, Design of the Burroughs B1700, AFIPS, FJCC, 1972, pp.489-497. (1972b)

---oOo---

(N.B. Some additional references which were relevant to this research have been given here although they have not been explicitly referred to in the chapters.)

Declaration

I declare that this thesis was composed by myself and that the research work described was performed by myself.

C.V.W.Armstrong