AN ADAPTIVE COMPUTING SYSTEM

by

R. J. GLOVER

A Thesis Submitted for the Degree
of Doctor of Philosophy at the
University of Kent at Canterbury
1978

ACKNOWLEDGEMENTS

I wish to acknowledge the financial support provided by the Science Research Council for this project.

I express my sincere thanks to Professor I. Aleksander for his guidance and encouragement.

I would like to thank the Staff of the Electronics Laboratories of the University of Kent for providing the research facilities and the Staff of the Department of Electrical Engineering and Electronics at Brunel University for allowing the research to continue.

I am particularly indebted to all my colleagues and friends for their discussions that set the seed of many ideas.

Thanks must also be given to John Ryan, John Husband and Ian Fisher who have helped in the arduous task of constructing the hardware.

Finally, I would like to thank Clare Fisher for the care and concern she has taken in the typing of this thesis.

R. J. Glover
Brunel University
Uxbridge

ABSTRACT

Several natural systems are comprised of networks of similar cells or processes which possess an intelligence or overall stability when interacting among themselves. Well known though not fully understood examples are the central nervous system or the activity of genes during cell replication.

Theoretical studies of network systems have been restricted by complexity to networks of cells possessing linear properties. Computer simulation of networks with non-linear cells is impractical because of the very large number of internal states possessed by networks of non-trivial size.

This thesis presents a practical approach to a system capable of modelling some networks using Random Access Memory (R.A.M.) circuits. Interconnections may be arbitarily specified between the address inputs and data outputs of R.A.M. elements and external network data inputs and outputs. The subsequent output state-activity of the network may be "adapted" according to external data using the R.A.M. write enable inputs.

A custom made R.A.M. integrated circuit is described which has provision for the simultaneous setting and resetting of its store. This is used in a mini-computer controlled network of 1,280 R.A.M.'s with interconnections specified by software.

An interactive high-level language is described that simplifies the specification and manipulation of networks. Its use is illustrated by programs for a single-layer classifier and a feedback network with natural clustering properties.

Finally the optimisation of the programmable network - interconnections and the execution of the high-level network language is presented using hardware micro-program controlled modules.

It is concluded that the system may be useful in a range of activities such as the design of Read Only Memory R.O.M. based pattern recognisers to testing hypotheses regarding neural networks.

CONTENTS

Page

Page

Page

DIAGRAMS

P - Portfolio

CHAPTER 1

1.0 Introduction and Review

This thesis is concerned with the problem of designing practical information processing machines whose state structure is generated by a process of adaptation rather than by an algorithm.

The term "information processing machine" is defined here as a machine that accepts some binary code or sequence of binary codes at its inputs and transforms them to an output sequence dependent upon its internal logic structure and its past inputs. This is too wide a definition to be of any practical use; here the machine's internal logic structure will be restricted to a network of cells, each cell behaving in a universal way although the overall network behaviour is not universal. Adaptation is defined as modifying the internal logic to cause a required state to be entered for the current input and state pair. The goal of adaptation is to produce some artificially intelligent processing by gradually causing ordered output sequences for both seen and previously unseen inputs.

One way of realising an adaptive processor is to write a simulation program for a digital computer. It is the programmer's task to find an efficient and flexible representation of the adaptive logic structure and storage for the current state using the available hardware resources of the computer.

Commercially available digital computers have instruction sets that are designed to perform only a limited number of efficient mappings of input codes on to internal states. These occur when the input codes represent numerical information and the internal logical structure consists of arithmetic calculations. However, a more general case such as processing a binary image, requires programs to manipulate each input bit separately.

The use of a digital computer is perhaps cost effective for those systems where the time taken to execute the programs representing the internal logical structure is equal to the rate at which data can be presented or the output sequence accepted. Where the dimensionality of the inputs is large or the internal logical structure irregular or where there are many internal states or any combination of the above, computer simulation becomes impractical and the conventional architecture of the digital computer must be replaced or enhanced by problem orientated hardware.

The nature of some hardware enhancements is investigated in this thesis with the goal of constructing a practical cellular adaptive processor employing readily available Medium Scale Integration (M.S.I.) components.

1.1 The Organisation of a Cellular Adaptive Processor

The cell or element used by the adaptive processor has been called "an adaptive logic gate" (Albrow R.C. & Aleksander I., 1968) though now it may be identified with a fully decoded Random Access Memory (R.A.M.) circuit that is storing the 'truth table' of its address inputs when they are used as logic inputs. There are no limitations on the structural variety made by interconnecting elements. (A fixed two layer structure is described in Chapter 3 for the network hardware implementation is a compromise between some fast inflexible connections and slow flexible ones. This does not affect the flexibility of structures obtained using the first layer of elements alone.) Adaptation consists of using the write enable and input data terminals of a R.A.M. element to include or delete the minterm formed by its address terminal inputs from the stored truth-table.

1.2 Systems Considerations for an Adaptive Cellular Processor

Control and specification of the network structure requires the flexibility of programming provided by the conventional digital computer. Programming using a machine language or a numerically orientated language bears no semantic resemblance to network structures. This has prompted the design of a high level language whose operands and operators directly relate to the manipulation of network structure. Translation of the high-level language into code controlling the network hardware is accomplished by a compiler program resident in the host digital computer.

Increased system throughput is achieved, especially for network interconnections, by directly implementing the high level operators in hardware using conventional M.S.I. functions and by controlling the data-flow by micro-programs. This ensures that there is no loss of flexibility in network structure.

1.3 Artificial Intelligence: Aims for Adaptive Processing Systems.

The interest in the intelligent behaviour of networks complements the approaches taken by the current exponents of Artificial Intelligence, (A.I.). Their aim is to write computer programs that build some internal model of objects in a universe, adding to a data base relations and rules extracted by interacting with the universe.

A.I. has been applied to game playing (such as chess), theorem proving and to robots. A typical example of the latter is described by Smith *et al.* (1975); here a robot "JASON", is being developed with 5 independent sensors and 4 independent effectors controlled by 10 processors including a multi-processor mini-computer system and a large remote main-frame computer.

A.I. research programs have proved very costly both in programming effort and in computing power and although notable achievements have been made, such as in natural language understanding (Winograd(1972)), overall progress has been reported by Gregory (1977) as modest.

There are two main reasons why the study of adaptive networks is considered to be important; adaptive networks can easily be implemented by low-cost components with an architecture that can be optimised for data-throughput or cost. Networks would be of limited 'intelligence', specific to a given task; some examples are low-cost character recognisers for a blind reading aid (Nappey, 1977) and a simple scene analyser for controlling an industrial robot (Dawson, 1976) which could be used for automatic drill positioning.

The second reason is less direct and more speculative. The case is obliquely supported by Dreyfus's comments upon conventional A.I. research.

Dreyfus (1972) criticises the emphasis of work in A.I. on the grounds that the philosophical understanding of human intelligence by A.I. workers is misconceived. Their emphaiss on finding heuristic procedures that can be realised by a digital computer program is equated to the reductionalist view prevalent in Western scientific thought. This, according to Dreyfus, assumes that heuristic rules can be applied to human situations in the same way, say, that the rules of physics are independent of the physical situation in which they are applied. He suggests that heuristics to compute intelligent behaviour, such as understanding and resolving ambiguities in a natural language, do not exist because the mind may be representational of its environment, behaving like an analogue computer rather than a series of computations in a digital computer. This may explain the impasse at which robot building and chess playing projects have arrived to date. To alleviate these cbjections he proposes "wholistic" processing to approximate to "zero-ing in" a segment of experience for which algorithms or heuristics are unknown.

It is open to speculation whether the parallel processing and adapting properties of R.A.M. networks can model the "wholistic" processing required for intelligent activity.

The scale of numerical problems posed by A.I. is considered by Raj Reddy (1973). Real-time processing of speech and visual data is peported to require a convensional computer with an instruction rate of 100 million instructions per second (m.i.p.s.) and a data rate of 100 million bits per second. Existing parallel processors such as ILLIAC IV and STAR are considered as being not useful in A.I. because <30% of processor time in the algorithms studied by Raj Reddy is spent performing arithmetic instructions. He suggests that pipe-line architectures are more suitable for A.I. using modular hardware similar to Bell's (see section 1.6.4) where parallelism can be tailored to particular algorithms.

1.4 Neural Models and Learning Elements

The following review traces the closely linked development of ideas about neural networks and cellular machines for pattern recognition. Some general ideas of network behaviour can be found that embrace neural networks, adaptive logic networks and genetic networks.

1.4.0 Neuron Modelling

The first attempt to model neuron behaviour was by McCulloch and Pitts (1943). Their neuron model ignores any chemical effects on neuron behaviour and considers only the electrical effects transmitted by the synapses. Each synapse is assigned a weight, of negative value in the case of inhibitory synapses, and a linear summation of the incoming activity of the synapses is made. If this summation exceeds a predetermined threshold, the output of the neuron model becomes active and this activity is transmitted to the synapses of other neurons. It is assumed that the outputs of neurons are binary and that the values of synaptic weights and threshold are fixed for a particular neuron. Several authors (Arbib 1964) have shown that any finite state automaton can be realized by modular networks of McCulloch and Pitts neuron models.

The ease with which an electrical analogue of the McCulloch and Pitts neuron model could be constructed was recognised and many designs have been produced (for example Harmon, 1961) refining the behaviour to that of an actual neuron. Conventionally synaptic weights are represented by resistors and summation is performed by an operational amplifier whose output is taken to a threshold detector circuit. The Boolean functions realis-able by the circuit are called "linearly separable" because when the possible min-terms of n variables are represented by a n-dimensional hyper-cube only those vertices which may be separated by a hyperplane can be included. The study of these devices is a branch of switching theory called threshold or majority logic. A detailed treatment of threshold logic can be found in Lewis & Coates, 1967.

The known behaviour of real neurons does not limit their logical activity to that of synaptic activity. Crane (1962) has shown that the ionic discharge propagating along an active nerve axon is able to perform logical operations such as NAND and OR at junctions and at interacting crossings caused by the nature of the refractory period which follows the discharge. Crane proposes an active transmission line circuit called the "Neuristor" that performs logical computations in a distributed manner. This could be used to solve some problems of power distribution and interconnection found in very large conventional computers.

Several circuit realisations of Neuristors have been described, of which Kulkarni-Kohli and Newcomb (1976) is the most recent and suitable for integration.

Learning or adaptive behaviour in networks was introduced by Hebb (1949). He proposed that the repeated stimulation of specific receptors led to the formation of assemblies of association area cells which continue local stimulation of each other after the passing of the stimulus (short term memory). The continued excitation of a pathway within an assembly is assumed to cause synaptic growth and hence clustering of the association assembly around a particular stimulus (long term memory).

The value of the computer in verifying psychological models is illustrated by one of the earliest computer simulations. Rochester _et al._'s (1956) simulation shows that assumptions have to be added to Hebb's model, such as a bias against firing by neurons which are remote from sensory areas, before clustering into association area assemblies takes place.

Rosenblatt (1962) describes networks known as "Perceptrons" based on a simplified model of the mammalian visual system. This assumes points on a retina are randomly connected to cells (or A-units) in a association area.

"A-units" behave like McCullock and Pitts neurons with unit weight inputs, some of which are inhibitory; they fire when the number of excitory retinal points exceed the inhibitory ones by some fixed threshold. Outputs of the A-units are taken to a response unit, (R-unit) via a variable weight where the sum is computed and a threshold decision made. Learning is accomplished by adjusting the weights by a feed-back mechanism from the R-unit output and can take two forms: supervised learning where an observer causes all the weights of responding A-units to be increased if a 'one' response is required for the stimulus and similarly decreased for a 'zero' response; unsupervised learning is accomplished by reinforcing the particular response given by each stimulus. Perceptrons with no feedback connections between A-units are called simple-Perceptrons. The speed of learning and the range of weight values generated when training a simple-Perceptron depend on the rules applied for adjusting the weights. These have been studied by Roberts (1960) and Nilsson (1965). Proofs of convergence of the training process for linerly separable dichotomies are given by Novikoff (1962), Nilsson (1965), Minsky and Papert (1968), and Block and Levin (1970).

1.4.1 Pattern Recognition by Perceptrons and Learning Networks.

We now look more directly at the relation of Perceptrons to the problem of pattern recognition. If the stimulus inputs of the Perceptron are considered connected to a retina used for sensing some two-dimensional binary pattern, the outputs of the A-units represent a group of n-tuple features extracted from the input pattern. Given a pattern, the set of features representing it can be expressed as a point in a multi-dimensional feature space; pattern recognition by discriminant analysis consists of finding some function that partitions the feature space into mutually exclusive, class dependent regions. Simple-perceptron type classifiers are limited to linear discriminant functions. Nilson (1966) presents two ways of enacting more complicated decision surfaces: piece-wise linear discriminators (Duda, 1966) and polynominal or ϕ-discriminators, (Nilsson, 1966). The piece-wise linear approach approximates the required curved decision surface to a number of separate hyperplanes while the ϕ-discriminator consists of a pre-processor between the retina input and the variable weight and summing element. The pre-processor or ϕ-processor has outputs consisting of all the possible combinations of its inputs up to the degree of polynominal required to perform the dichotomy.

Multi-class dichotomies can be realised by two methods: a separate set of weights can be used for each discriminant function required to separate one particular class from all others, R sets of weights would be required for R classes; a binary tree of decision spaces can be formed requiring $\log_2 R$ sets of weights. Discriminant functions for the latter method would probably be difficult to generate. In the former method the threshold detector comprising the output decision is replaced by a maximum value selector. The size of mis-classification and rejection classes can be controlled by selecting suitable margins between the responses of the different sets of weights to unidentified patterns.

The connections between the input sensors and the inputs to the A-units of perceptrons or learning network classifiers are usually randomly specified. This is to obtain independence of features for differing classes. It is to be expected that improvement in performance could be obtained by re-specifying connections. However, the number of possible connections increases very rapidly with the size of the input array, which would preclude an exhaustive search for the best set of connections. For specific types of classes, connections can be defined intuitively to generate well known features of the class such as junctions or straight lines of a particular orientation and it appears reasonable that classification should be enhanced by choosing these features; however *a priori* knowledge of the training classes is necessary.

1.4.2 Models of Some Neural Pathways

Perceptrons and learning networks with specific connections have been used in attempts to model the neural structures physiologists have found in the visual and other pathways of animals.

In vertibrates pre-processing of the information gathered by the visual receptor cells (rods and cones) of the retina is carried out by ganglion cells before it is passed to higher "perceiving" centres of the brain. Each ganglion cell receives inputs from a localised area of the retina called its receptive field. The nature of the processing performed by ganglion cells has been found to be species specific which perhaps belies more general conclusions about visual perception in man being drawn from animal studies. A general trend is that less pre-processing occurs in more 'intelligent' species.

Four types of ganglion cells have been found in the frog (Lettvin et al. (1959)): sharp boundary detectors; movement gated small convex boundary or "bug" detectors; moving contrast detectors; dimming detectors. In contrast to the frog Hubel and Weisel (1966) report that the cat has only two types of ganglion, each with circular receptive fields, one responding to a small centre field inhibited by the outer field, the other responding to the outer field inhibited by the centre field. Further along the cat's visual pathway there are neurons that respond to edges and lines of particular orientations. Work on new-born kittens shows that these structures exist at birth.

Simple perceptron models of the frog's visual pathway have been made by Rosenblatt (1962).

A multi-layered system of analogue threshold elements to extract similar features to the cat's visual system is described by Fukushima (1969). This uses local feed-back between line detection layers to suppress the detection of lines in a particular orientation from being extracted as lines of unit length in the perpendicular orientation. A simulation of the system written in FORTRAN for an IBM 7044 computer is reported to require 35 minutes computer time to process a figure on a 50 x 50 input matrix.

Ganglia of the pigeon have been found that are sensitive to stimuli moving in a particular direction. Hardware models of these have been demonstrated by Runge(1968). Similar hardware models of the frog retina and ganglia have been made by Herscher and Kelly (1963) and Sutro (1968).

1.4.3 Single-Layer Learning Networks

The perceptron shows similarities to the biological pre-processing stage ganglion cells in that A-units compute fixed predicates of a small sample of the input retina. An alternative approach to learning in this situation is to change the predicates according to data in a training set.

A well known method employing this technique is the n-tuple method of Bledsoe and Browning (1959). This method is particularly relevant here since it is the basis for single layer learning networks of R.A.M. elements. The n-tuple method consists of sub-dividing the input pattern randomly into n-tuples which are then considered as features of the pattern. During training a set of patterns is presented and the occurrence of a particular state of each n-tuple is recorded separately for each class. An unknown pattern is then classified as belonging to the class with which it has the most n-tuple states in common.

Bledsoe and Browning's original simulation program was written for a 36 bit word computer: each bit in a word was used for assigning a different class giving the program a convenient organisation to perform character recognition of the alphabet plus the numerals 0 to 9. Each n-tuple selected one of a group of four words, n in this case being 2.

Storage requirements for a classifier of C classes, with its input sampled by M n-tuples is $CxMx2^n$ bits. Dawson (1976) has shown that the execution timing and storage requirement of a RAM network simulation and an equivalent Bledsoe and Browning organisation have no significant differences.

Generalisation, that is the ability to respond to previously unseen members of a class, is caused by the combining of different n-tuple states from the training set. A dichotomiser trained on two patterns will give a maximum response for any pattern that has any combination of n-tuple states in common with the training set. Clearly the size of this set is the product of all n-tuple states present in the training set for each independent n-tuple. The degree of generalisation obtained with a given training set is dependent upon the size of n-tuple. In the case of n=1 we have maximum generalisation corresponding to "template matching". For larger n, generalisation increases until there is difficulty in obtaining large enough training sets to provide an adequate number of representative n-tuple states.

Aleksander, (1970) has shown that the response of a single-layer learning network to an unseen pattern is dependent upon its "Hamming-Distance" from each member of the training set. Stonham (1977) shows that an improved prediction can also be made by considering the Hamming-distance relationship between members of the training set; however, this is only possible for small training sets ($<$ 15 patterns) because of the number of computations involved.

Optimisation of the size of n-tuple for hand-written character recognition with limited sizes of training set has been considered by Ullmann (1969) and the size of the training set for sufficient training by Cheung, (1973).

One apparent problem with the n-tuple method is its sensitivity to 'rogue' patterns in the training set. Methods to overcome this have been proposed by Bledsoe and Bisson (1962) which involve measuring the frequency of occurrence of n-tuple states in the training set. Discrimination now takes place by predicting which n-tuple states are most likely to occur in the testing set. Cheung (1973) has proposed hardware suitable for integration to perform this. An ingenious method of implementing this has been proposed by Reeves (1973) using conventional R.A.M. integrated circuits with the addition of a parallel adder circuit. Ullmann (1969) has reported difficulty in defining large enough training sets for the maximum likelihood method for character recognition because the distribution of occurrence of n-tuple states is heavily biased towards low frequencies.

Single-layer learning networks have been used successfully by Stonham (1975) to recognise chemical mass spectra data resulting from single compounds with 99% accuracy as one of 42 classes. The method has recently been developed for mixtures of compounds (Stonham, T.J., and Enayat A, 1978). Early results on mixtures of two compounds shows that detection of both compounds can be 79% whereas detection of at least one compound may be 98.5% (Enayat, 1978).

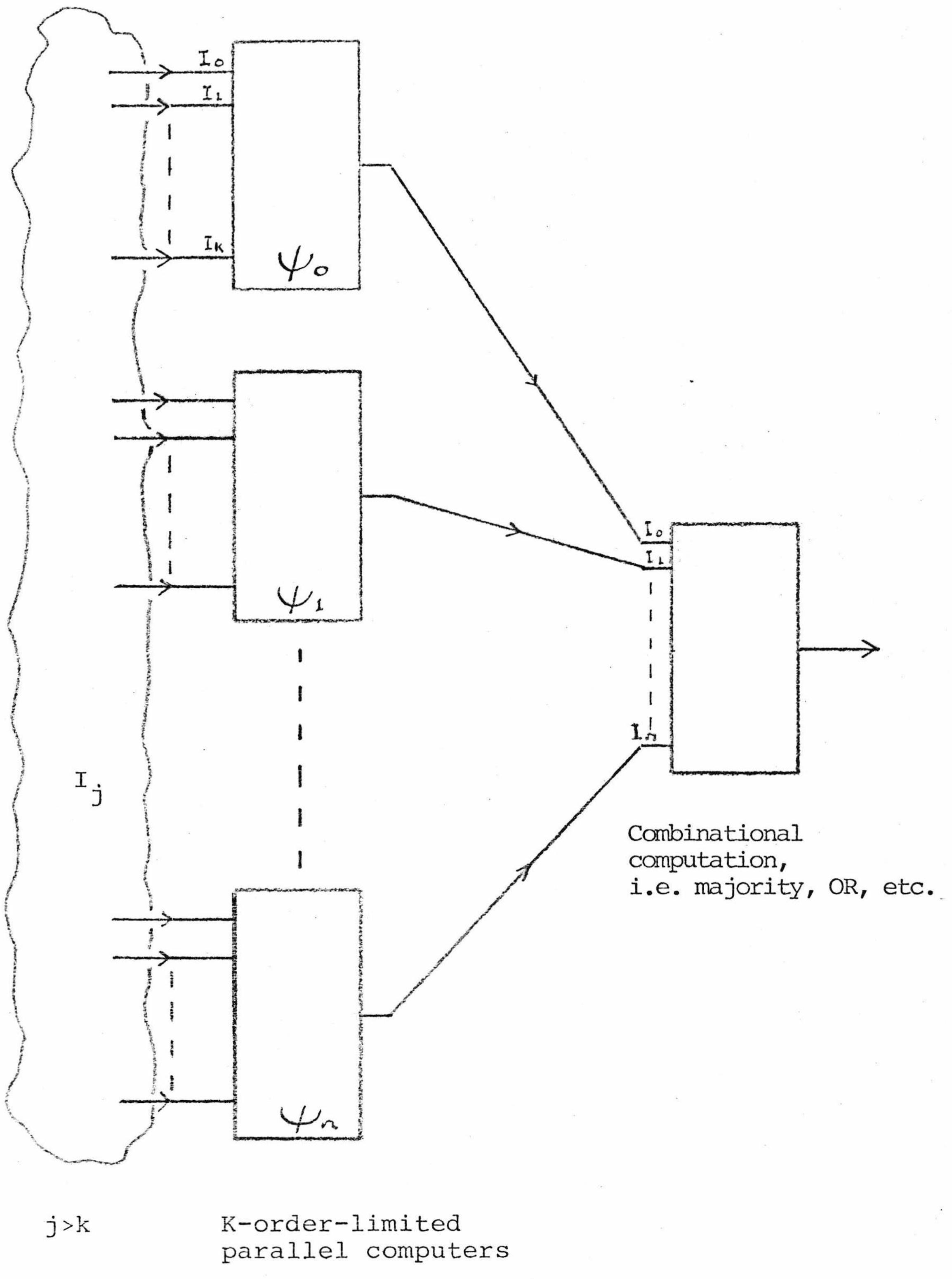
Fault diagnosis in a sequential digital system has also been attempted by single-layer networks (Aleksander and Al - Bandar, 1977). Test points in the digital system provide inputs for a single-layer network which is trained when the system is known to be functioning correctly. A malfunction during use causes patterns on the test points that have not occurred during training and an error signal can be output if the test patterns are outside the nets' generalisation area. Malfunction detection rates of about 90% have been achieved by a preliminary network system.

All methods of pattern recognition where the classification is made on the basis of a combinational computation of localised measurement or n-tuples of the pattern can be represented by the simple parallel processing scheme of figure 1.1. . Minsky and Papert (1968) have considered thoroughly the properties of simple parallel processing schemes, called generically "Perceptrons" although this also includes single-layer learning networks. They show that some simple global properties of the input pattern cannot be computed by Perceptrons of limited order, order in this sense being equivalent to the size of the n-tuple. The global properties considered were parity, specificity, uniqueness and connectedness. These have been shown to be computable by simple ring-like structures of R.A.M. elements (Aleksander, Stonham and Wilson 1974).

1.4.4 Networks with Feedback Connections

Feedback in perceptrons was an interest of some early studies by Rosenblatt, (1967) though its application to R.A.M. learning networks has produced some interesting hypotheses with relation to the concept of mind and human thought processes.

Aleksander and Mamdani (1968) report improved pattern recognition where, instead of a R.A.M. network being trained to ones and a majority output decision made, the network is trained to output some archetype of the training set for each class. The output response to an unknown pattern is fed-back to the network inputs on successive clock pulses to produce closer approximations to the archetype. Recognition is identified with having entered a particular state trajectory. The network



I_0
I_1
I_K
ψ_0
ψ_1
ψ_n
I_0
I_1
I_n
I_j
Combinational computation, i.e. majority, OR, etc.
j>k
K-order-limited parallel computers

Figure 1.1 A Perceptron-Like Parallel Processing Scheme

exhibits short term memory in the sense that it will remain in the same locality of its state space after removal of the input pattern.

The above network operates autonomously once the feedback connections are activated; the properties of networks whose inputs are 'OR'ed with feedback is considered by Fairhurst, (1973). These networks are capable of unsupervised learning and their structure is considered in greater detail in Chapter 4.

1.4.5 Stability and Networks with Feedback Connections

Feedback networks with OR'd feedback show exceptional stability due to the constraining nature of OR'ing the output state with the input pattern on the inputs connected to the RAM elements. However, more general networks whose functions and connections are randomly specified with feedback connections taken directly to RAM element inputs also show an unexpected stability.

A similarity may be drawn between an autonomous network of two input R.A.M. elements and the biochemical behaviour of genes during cell replication. Kauffman (1969) studied random interactions between genes assuming two or possibly three other genes could interact in a binary way as represser or inducer with a particular gene. Digital computer simulation showed that a "network" of 100 genes have a total cyclic activity of about 100 states with typical cycles containing 100 distinct patterns. This agrees with biological information about known numbers of cell types. (no. of discrete cycles), and cell replication times, (length of cycles). Aleksander and Atlas (1973) have shown that in RAM networks cyclic activity is caused by only those elements that are connected in rings of "live" elements (those elements connected in a ring whose inputs cause a change of element output to be propagated around the ring). The stability of the network is analysed as a function of the "liveliness" of the elements where "liveliness" is defined as the probability that a change of an input will be propagated to the output. In networks of low connectivity, (few inputs per element), stability occurs because the probability of finding 'live' rings is low. In networks of high connectivity the probability of live rings occurring is greater but, in the case of neural networks, the activity is restricted by the low "liveliness" of the linearly separable functions to which the neuron is limited.

The concept that mental activity can be explained in terms of the state space trajectories of the neural networks comprising the brain is presented in a general work by Aleksander, (1977). Earlier papers in this field (Aleksander, 1970) have presented models for psychological properties such as sequence recognition and recall and attention in terms of the state space properties of R.A.M. element learning networks. A text covering early work on single-layer networks and simple feedback networks is Aleksander (1971).

1.4.6 Other Feedback Network Applications

Another feedback network with analogous psychological properties is the Action-Orientated Network (Aleksander (1975)). Single-layer networks and Fairhurst's feedback network produce, however, only marginal responses for different classes, dependent upon Hamming Distance relationships, whereas animals are required to resolve some marginal differences very quickly and resolutely in order to escape from a preditor or to catch some quickly moving prey. The action orientated network improves its initial marginal response on successive clock pulses because of the training algorithm used. Each element has one input connected randomly to an element output. During training, the network is partitioned into two, upper U and lower, L. Feedback is initialised so that the partition U feedback is ones and the partition L feedback is zero's. The network is then trained so that partition U outputs ones and L zeros for all members of one category. The state of partitions U and L are interchanged for training on the other category. The effect of feedback is to partition the stores of the R.A.M. elements into two non-overlapping sets one for each category. When an unknown pattern is presented to the network and feedback set to zero the response from partition U and L, denoted by (U) and (L), represents the marginal response caused by common n-tuples between the input and the training sets. The effect of feedback on subsequent clock pulses is to improve the response ratio $^{(U)}/_{(L)}$ by selecting, incrementally, the partition of RAM store trained on the same category as the marginal response. It is open to speculation whether "action-training" takes place in biological networks but the action-orientated network provides a model for modal change at the level of network circuitry that is missing from models of biological mode arbiting networks such as Kilmer, McCullock and Blum's (1968) "S-RETIC" model for the reticular formation or Arbib's (1972), somato-topic model of the frog's tectum.

R.A.M. networks with feedback have also been used by Reeves (1973) to enable the tracking of simple patterns. Feedback is necessary for the network to resolve which exit to take when different tracking paths meet at pattern junctions.

A theory of visual perception by Noton and Stark (1971) suggest recognition of form take place because the eye executes particular sequences of movements. Attempts by Reeves to recognise untrained input patterns by the sequence of tracking commands generated by the network tracking them were only of limited success. A possible explanation for this is that Reeves' system lacked peripheral vision which would enable the system to make large movements to areas containing "interesting" features. The system design of Reeves'software is considered in Chapter 4.

Simple scene analysis has been accomplished by Dawson (1976) using Reeves' hardware and software system. Here networks are used to control the size and positioning of a window viewing a two-dimensional scene. Objects are isolated by a scanning search and normalised and registered in the window by trainable networks. The system could be a suitable basis for a low-cost industrial robot applied,for example,to the automatic drilling of printed circuit boards.

1.5 A Comment

The preceding sections of the introduction and review have concentrated on those areas of neuro-physiology and work on adaptive pattern recognition methods that have influenced the systems specification of the adaptive processing system presented in this thesis. We shall now consider the opposite end of the design hierarchy and present a technique, becoming more generally applied by digital system designers to maximise the flexible use of hardware resources. This technique is of importance in the design of our adaptive processing system where high-level representations of network structure have to be mapped into a hardware implementation limited by manufacturers' available M.S.I. components.

1.6 Micro-Programming Techniques

A complex digital system requires the generation of many timing signals to enable various data paths and clock storage registers. Usually some discrete sequence or pattern of control signals is required to implement each mode of operation that the digital system can carry out. A convenient method for specifying and designing the control circuits is to analyse the control problem in terms of a finite state automaton: choosing a Mealy model, the inputs will be the external mode control or instruction and certain key states of the digital system that can cause conditional branches in the control path. The output mapping provides concurrent control signals for all the control inputs of the digital system. The number of states passed through in a state cycle represent the sequence of control operations required to complete a mode of operation or instruction by the digital system.

The realisation of the control automaton is quite often by hardwired logic designed using <u>ad hoc</u> methods for state identification and assignment. Early in the development of digital computers, Wilkes, (1951) proposed an orderly method for designing the control automat on for digital computers which he called micro-programming. Wilkes' original mechanism is shown in Fig.(1.2). It consists of two look-up tables, one to perform the output mapping (B) and the other to perform the next state mapping (A). The look-up tables were implemented by

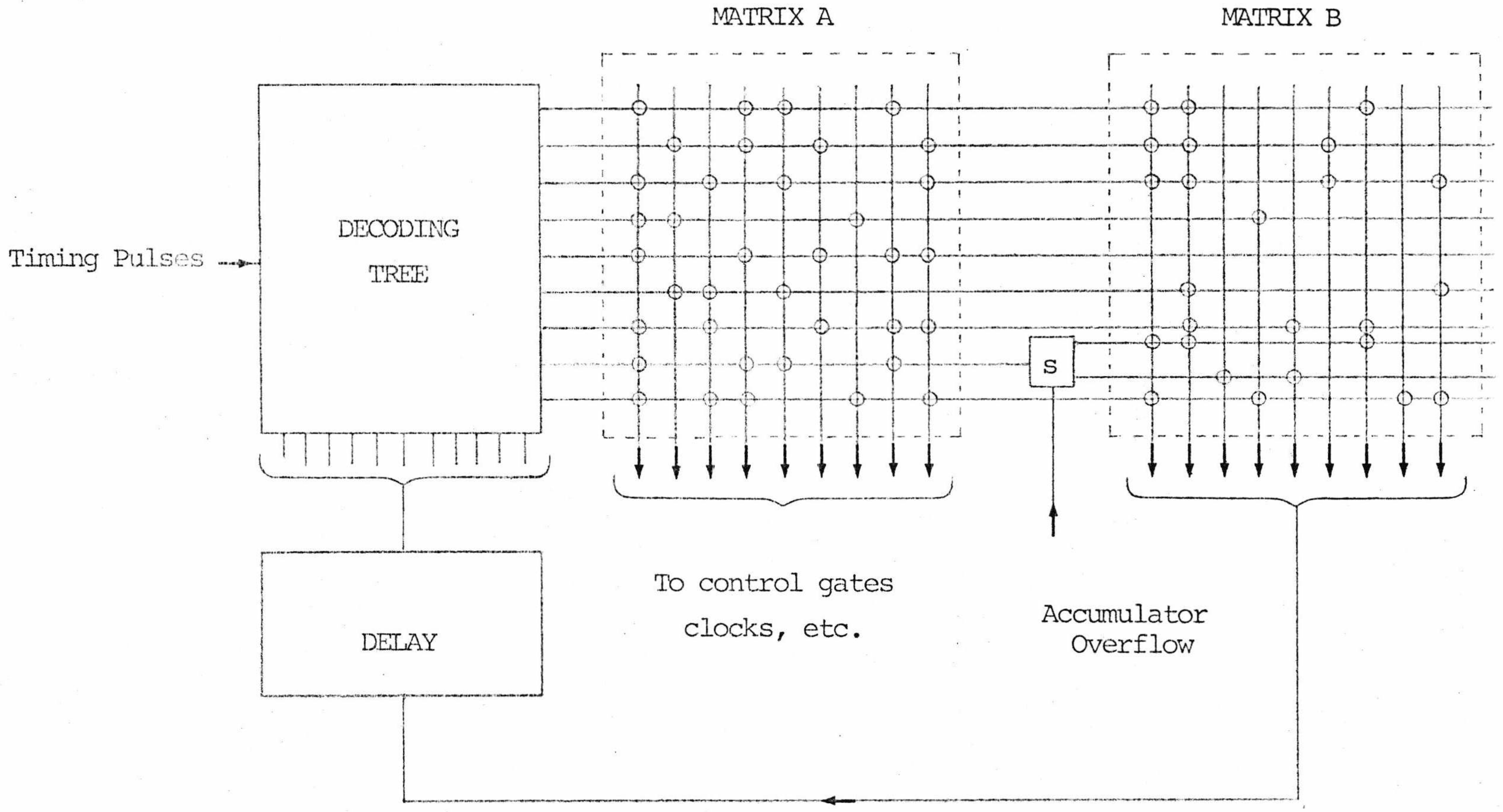


Figure 1.2 Wilkes's Micro-programmed Controller

diode switching matrices now called read-only memory, R.O.M.. Matrix A consists of an n word by m bit R.O.M. where n is the number of "micro-states" required to implement all the operations of the computer and m is the number of control lines of the computer. Each control line is called a micro-order. Matrix B is a similar R.O.M. with $\log_2 n$ bits per word. However, it contains n+k words where each of k extra words represents an alternative next state to one particular present state selected according to one of K test signals from the computer. The functional unit formed by the A and B matrices is commonly called the control store.

1.6.0 The Development of Micro-Programmed Controllers

Although Wilkes's original scheme is straightforward to implement, it requires a large amount of R.O.M. which until recently was very expensive in the speed range required to produce an efficient micro-programmed controller. Developments of the micro-programmed controller have been aimed at the reduction of the store size required.

Storage can be saved without much modification to the Wilkes scheme by encoding micro-orders that do not occur concurrently into fields in the micro-instruction. This is called "single-level encoding". More storage can be saved by two-level encoding. Here the meaning of a field is not explicit, as in single-level encoding, but depends upon some other value such as another field in the micro-instruction or some external machine state.

Encoding requires additional hardware to decode micro-orders and adds several gate timing delays to their generation.

Micro-instructions that are capable of generating several independent control orders are called horizontal micro-instructions while, in the extreme case of encoding, micro-instructions that cause only one possible micro-order are called "vertical micro-instructions". There is an ill defined area where micro-instructions have a limited capacity to generate some micro-orders simultaneously. These are called "diagonal micro-instructions".

A scheme of encoding, proposed by Gerace (1963), enables the B matrix next-state look-up table of Wilkes original scheme to be dispensed with. An incrementor circuit is used to generate the next micro-instruction address from the current micro-instruction address except in the case of branch or conditional branch micro-instructions. Provision is supplied to load the next micro-instruction address from a field in the present micro-instruction word for branch and conditional branch micro-instructions.

Another method of reducing the size of control store is by the use of a residual control scheme. Micro-instruction fields in the residual control scheme do not generate micro-orders directly but control set-up registers which can be manipulated using short vertical micro-instructions.

The set-up registers can be used to generate several simultaneous micro-orders like horizontal micro-instructions.

The original Wilkes control scheme is called a serial one because the fetching of the next micro-instruction from the control store occurs after the execution of the micro-orders generated by the current micro-instruction. The fetching and execution of micro-instructions can be interleaved to produce the so called "parallel implementation".

The parallel implementation requires the addition of a staticizing register, called the "micro-instruction register", at the outputs of the control store to hold the current micro-instruction while the next micro-instruction fetch is being performed. Problems arise in parallel implementation because of conditional branch orders that depend on the result of some micro-operation in the current micro-instruction. In this case fetching the next micro-instruction has to be delayed until the result of executing the current micro-instruction is known. Alternatively, the fetch can be made assuming a particular value for the conditional. In this way, a time overhead is only incurred when the condition proves to be the opposite value.

The effects of control store reduction by encoding and speed-up by parallel-implementation is to complicate the control scheme required to implement the micro-program unit itself. The timing of a micro-programmable controller is called monophase if micro-instructions are enacted by a simultaneous issue of control signals. In the case of a micro-instruction being enacted in several distinct phases or minor clock cycles it is called a polyphase implementation. "Synchronous polyphase" is where the micro-instruction is executed by one major clock cycle comprising several minor clock cycles. Where the number of phases depends upon the complexity of the micro-operations carried out by the micro- instruction it is called "asynchronous polyphase". Polyphase implementation facilitates the parallel fetch and execution of micro-instructions but it requires sequential logic hardware to generate the minor control phases; however, this presents a design problem identical to that which the original conception of micro-programming set out to solve, albeit at a lower level of resource interactions

1.6.1 Writable Control Stores

An important advantage offered by micro-programming is the ability to redefine existing modes of operation and add new ones without having to modify the logic hardware of the controller. Rosin (1969) points out that a beneficial redefinition of floating-point arithmetic across the entire I.B.M. System 360 line of computers, which took place in 1968, was only possible because of the adoption of micro-programmed hardware.

A most significant development has been the introduction of writeable control stores employing high speed integrated circuit R.A.M.'s The areas of application for the writeable control store are, emulation of the characteristics of a variety of machines, the interpretation of high-level languages, machine diagnostics, program enhancement and flexible special purpose peripheral controllers for applications such as graphics or signal processing.

1.6.2 Emulation

Emulation enables programs written in the machine language of one machine (called the target machine) to be executed on a host machine that supports micro-programs written to implement the architecture of the target machine.

The generality of a micro-programmed emulator depends upon the hardware resources of host machine. Commercially available emulators are usually restricted to the emulation of a particular machine to allow customers to run existing software with later ranges of machines. In emulation mode the writable control store is overlayed with the micro-programs to perform the emulation from backing store.

Several general purpose emulators have appeared or are the subject of current research.

In order to provide flexible hardware resources without excessive control store size some machines use two levels of micro-programming. The NANODATA QM-1 (NANODATA Corp 1974) has a main memory of 18 bits per word and a control store also of 18 bits per word with a set of registers to implement a residual control scheme. Micro-instructions are interpreted by "nano-instructions" which are horizontal micro-instructions of 360 bits per word. Both control store and "nano-store" are writeable to produce a machine of great flexibility designed for efficient emulation of other machines and investigation of computer architectures. "Nano-instructions" may be considered polyphase since they are partitioned into fields of 72 bits each. The first field supplies overall control conditions for the system. The remaining fields initiate specific operations; each field is activated circularly in order during each clock cycle until one condition causes the fetch of a new nano-instruction.

Another commercial machine with a similar two level micro-instruction architecture is the Burrough's Interpreter or B700 system (Reigel, <u>et al</u>. (1972)). This originated from a desire to provide smooth and unlimited systems growth in a machine that could support a variety of machine word sizes. The hardware is modular in construction.

A research orientated host machine, the MATHILDA system developed at the University of Aarhus, Denmark (P.Kornerup,

1975) has particularly sophisticated mechanisms for shifting and masking words of up to 128 bits. This is to enable manipulation of fields within data words such as the construction of various floating point formats. The shifter can be compared with the Serial/Parallel mapping technique presented here in section 6.2.2. MATHILDA provides variable radix cyclic shifts by enabling any bit position of the"Accummulator Shifter" to be loaded by the "shifted-out" bit. Local storage for mask patterns are limited to two words compared with the one mask store for each shift class of the S.P.mapper.

1.6.3 The Execution of High Level Languages

Most computer systems are required to support several high level languages with widely differing constructs. Since the machine generally has one instruction set there are many high level constructs whose machine instruction representations are inefficient in terms of execution speed. and memory requirements.

Micro-programming may be used in one of three ways to enhance high-level language execution:

i) The high-level program may be translated directly to micro code;

ii) A series of complicated micro-programs may be used to interpret the high-level language directly;

iii) Software or micro-code routines are used to translate the high-level language to an intermediate language which can be interpreted by fairly simple micro-programs.

Approach i) is used for the execution of the MINIC language by micro-code presented in Chapter 6. This is because the hardware resources were designed specifically for MINIC language constructs which simplifies the translation of MINIC into micro-code.

Approach ii) is taken by some implementations of high level languages such as A.P.L. where type attributes are not bound until execution time and great flexibility can be maintained at run-time by a micro-coded interpreter. A micro programmed interpreter for APL using an IBM System 360, Model 25 computer is described by Hassitt (1973).

Approach iii) covers a variety of situations one of which is the traditional approach where the high level language is translated into the machine instruction set. More useful forms of intermediate languages are the internal representationsof the source program as generated by the syntax analyser of the high level language translator such as Polish strings, triples or quadruples.

An interesting example of a computer without a fixed machine instruction set is the Burroughs B1700 system

(Wilner, 1972). The machine is programmed in S-languages (soft-languages) each one of which represents the constructs of some high-level language. Interpreters for the various S-languages are written in micro-code. The micro-instructions are vertical in format with two-level encoding; arithmetic operations are controlled by use of residual control. Execute and fetch of the next micro-instruction are overlapped but there are no minor clock cycles. An important architectural feature is the addressing scheme. Although memory is physically organised in bytes there is no fixed word length; data is bit addressed and there is a hardware register that contains the length of the field being addressed.

1.6.4 Micro-programming and Modular Design

A Modular realisation of digital systems is described by Bell *et. al.* (1972) based on a description of the digital system using a register transfer language. This uses a standardised set of hardware modules that communicate with each other asynchronously via common bus connections using interlocking protocol signals. Control modules are hardwired to data modules according to a desired order of "register transfers" between modules.

A writable control store module to enable programmable register transfers using the modules described by Bell is described by McDonald, Sustman and Harris (1973). This has vertical micro-instructions of only 8 bits wide that are encoded into two generic groups, one group concerned with micro-program flow and the other group with activating register transfer modules. Two fetches from the writable control store are used to cause register transfers; the first micro-instruction fetch activates a module to write data onto an interconnecting bus, meanwhile an overlapping fetch of the next control store word is completed which then activates the receiving module and causes data transformation.

The scheme is similar to that of the modular processor described at Chapter 6 but differs in that inter-module transfers in the modular processor are initiated by a single control store fetch and more complicated over-lapping of micro-instruction execution is employed. Data modules in the modular processor are designed for special purpose operations and can have complicated internal timing signals. Bell's register transfer modules are simple general purpose structures.

1.7 Concluding Remarks

This section has served to introduce some of the basic concepts of micro-programming and present it as one of an interrelated set of hierarchies comprising physical design, hardware architecture, micro-programmed control, software, high-level language specification and systems specification. How each of these hierarchies should interact in a digital system is an art of systems design rather than a science and the literature can only be studied with reference to the design of special cases.

Micro-programming is perhaps the best documented area of digital systems design and its employment eases the task of specifying the hardware architecture. Husson (1970) has collected early work on micro-programming while a more up-to-date text with the emphasis on user micro-programmable machines is Agrawala & Raucher (1976).

Micro-programming is used in the final form of the MINERVA system,presented in Chapter 6, to preserve the high level constructs developed with the MINIC language, presented at Chapter 4, by simplifying the hardware architecture required for the efficient execution of the MINIC language.

CHAPTER 2

2 The Design and Fabrication of an Integrated Circuit Adaptive Logic Gate.

This chapter describes the design of an Adaptive Logic Circuit and its implementation as a Metal, Oxide, Silicon (M.O.S.) integrated circuit suitable for incorporation in a digital computer controlled learning network. The origins of Adaptive Logic Circuits are discussed and their equivalence to fully decoded Random Access Memory R.A.M. circuits noted.

2.0 Adaptive Logic Circuits

An Adaptive Logic Circuit (A.L.C.) may be defined as a combinational logic circuit whose function depends upon an internal state. A change in this state may be effected by control inputs enabling the function to be 'adapted' to a required goal. Such a circuit of n inputs and m outputs is said to be a universal A.L.C. if all 2^{m2^n} functions of m inputs and n outputs can be realised. A simplification is made here by considering A.L.C.'s of only one output, that is n = 1, since multi-output circuits can be easily realised by connecting the logic inputs of one output A.L.C.'s in parallel and using separate control inputs.

The universal single-output A.L.C. can be implemented by a Universal Logic Circuit,(U.L.C.) with 2^n function control inputs provided by a 2^{2^n} state sequential circuit.

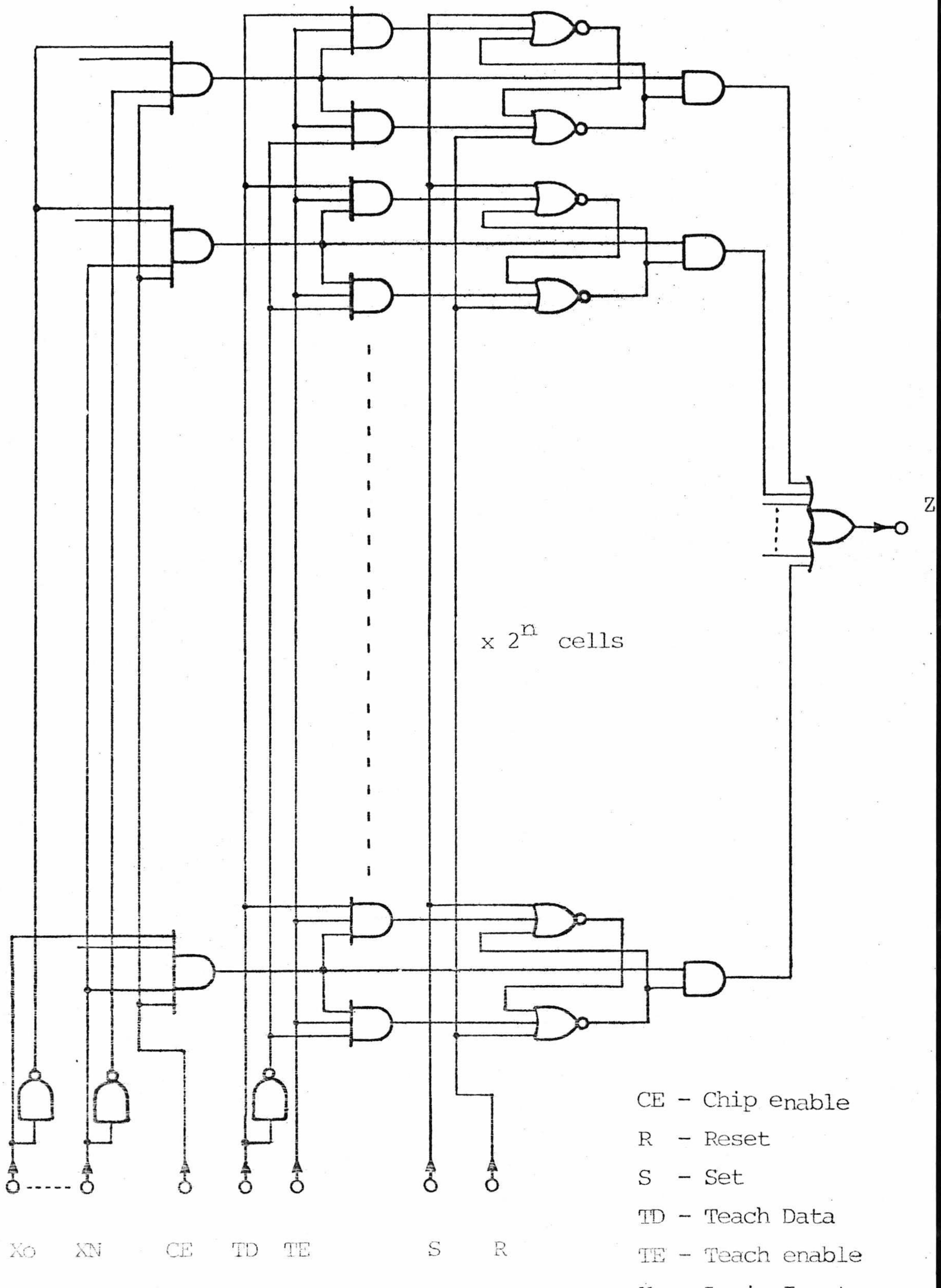
Aleksander (1966)considered the design of U.L.C.'s with regard to optimising the number of gates required, gate fan-in and the number of logic levels for a given number of inputs. For an n input circuit it was shown that it could be realised at two extremes by n+1 levels of 2 input gates or 2 levels of gates with a maximum fan-in of 2^n.

Various control schemes were later described by Aleksander (1968) whereby U.L.C.'s are used to implement A.L.C.'s. The first of these schemes may be called "function searching". Here the control inputs are provided by a maximal length code generator. Adaption is caused by applying an error signal to a search control input which causes the generator to cycle through its 2^{2^n} states until the correct function is found. This is found to be practicable only where $n<5$ due to the increase in the number of states as a double exponential of n. A better scheme was later proposed whereby each control input is fixed once a change in it has caused a correct response. This reduces the search procedure to 2^n operations and was used in the first micro-circuit implementation of an A.L.C. as reported by Aleksander, Albrow, and Noble (1967).

The device produced above was used as the model for the specification of the A.L.C. described later in this Chapter. It had 3 inputs and 8 flip-flop function control stores and was refered to as S.L.A.M.-8 (Stored Logic Adaptive Microcircuit). As well as having function adapting inputs it also had inputs to set or reset all the storage flip-flops simultaneously. A four level configuration of U.L.C. was used in which the control store was addressed by the orthogonally decoded inputs representing the min-terms of the function. The logic diagram of a S.L.A.M. type A.L.C. is shown in Fig 2.1. S.L.A.M.-8 devices were made using an experimental M.O.S. process for which the photo-masks were prepared by hand, consequently the yield and electrical performance were poor. Thirteen usable devices out of a batch of 100 were obtained and were limited to a maximum clock frequency of 50KHz. This compares with a yield of about 70% and a maximum clock frequency of 1MHz for S.L.A.M.-16 described later.

2.0.0 S.L.A.M.'s and R.A.M.'s

It is now apparent that the S.L.A.M. structure is almost identical to that of the fully decoded Random Access Memory (R.A.M.) of one bit word length, the only differences being the S.L.A.M.'s ability to set and reset all its store simultaneously by means of its S. and R. inputs. These similarities were not immediately apparent due to the technological differences between historical implementations of R.A.M.'s and S.L.A.M.'s



CE	-	Chip enable
R	-	Reset
S	-	Set
TD	-	Teach Data
TE	-	Teach enable
Xn	-	Logic Input
Z	-	Output

Figure 2.1 Logic Diagram of a S.L.A.M. Type Adaptive Logic Circuit

The traditional application of the R.A.M. is in main frame storage of the digital computer. Such storage requires randomly accessible words (as opposed to single bits) in which address decoding circuitry is common for all the bits of the same word. Until the advant of large scale integrated circuit technology they were generally implemented by coincidently addressed planes of ferrite cores or magnetic plated wires. This addressing scheme was also copied by early integrated circuit R.A.M. devices, (Texas Instruments (1974), which precluded their use as A.L.C.'s without extra decoding circuitry. In the present generation of integrated circuit R.A.M.'s, there has been a development towards fully decoded one bit designs for economic considerations concerning the ease of manufacture. The main factor involved is the number of terminals for various R.A.M. architectures for a given number of bits of storage. A two dimensional coincident chip of m bits requires at least 2 m address terminals whereas a fully decoded R.A.M. of n words of m bits requires $\log_2 n$ address terminals. For such a R.A.M. the total number of terminals is:

$$2m + \log_2 n \quad , \qquad \S\ 2.1$$

excluding power connections.

For a given memory capacity this is a minimum when m=1. Using bidirectional data paths with input data and output data terminals common § 2.1 becomes:

$$m + \log_2 n \qquad \text{terminals.}$$

This means that designs at present favoured by integrated circuit manufacturers are 256 x 1 bit, 1024 x 1 bit and 4096 x 1 bit memories using 16 and 18 pin packages. The fourfold size relationship in technical development has been made for commercial reasons. RAM's are implemented by a wide range of integrated circuit technologies but large capacity devices are generally implemented by the M.O.S. process. Two basic types of circuit are used, dynamic and static. In dynamic R.A.M.'s the information is stored as a charge on the gate region of a M.O.S. transistor. They have the advantage of small physical size and low power dissipation but the disadvantage that the information has to be refreshed at periodic intervals. This is normally done by instituting a read cycle of some subset of the addresses and requires additional control circuitry. The use of dynamic R.A.M.'s in networks of A.L.C.'s is precluded because of the extra complexity of the control circuitry caused by the address inputs forming part of a large data space instead of a common address bus. Static R.A.M.'s use flip-flops as the information store which requires no refreshing. They are suitable for use as A.L.C.'s providing the set and reset facilities of A.L.C.'s can be dispensed with.

The first fully decoded integrated circuit R.A.M.'s to be available commercially were the '1101', a 256 x 1 bit static R.A.M. and the '1103', a 1024 x 1 bit dynamic R.A.M. produced by Intel (1970). They were not available until mid 1970, after the decision to produce a custom version of S.L.A.M.-16 had been taken and sample S.L.A.M.-16 devices produced.

2.1 <u>The Development of S.L.A.M.-16</u>

S.L.A.M. -16 is a direct successor to S.L.A.M.-8 mentioned previously. Its main advances over its predecessor are its faster operation caused by increased control over the internal gate geometries and on chip interfacing circuits which enable direct connection with other standard integrated circuit families. The design and development was completed by the author in conjunction with Integrated Photomatrix Limited, Dorchester, during February and March 1970. Production S.L.A.M.-16 devices, produced by Integrated Photomatrix Limited were available during July 1970.

The following sections describe the development for which the author was responsible and only such relevant information of the M.O.S. process needed to illuminate the design steps is included. A more detailed account of M.O.S. process is given in Crawford (1967).

2.1.0 <u>Specification of S.L.A.M.-16</u>

Providing a device specification consistent with all the requirements and constraints of the production process is the first and perhaps the most difficult task of the designer. Once a consistent specification has been achieved, the design process proceeds by the application of known rules to produce a finished circuit. The author was helped here by the existence of S.L.A.M.-8 from which the logic specification for S.L.A.M.-16 could be taken. The remainder of the specification is drawn up by considering the performance required of a device which is to be incorporated into a large computer controlled system. These are tabulated as follows:

a) The overall propagation delay between input and output should be short enough that, when including external buffering and control logic, the maximum data rate via an interface to a Honeywell DDP516 digital computer (c.f. Appendix A) can be realised. This requires a propagation delay, tpd < 3 μS, to allow operation by consecutive Input/Output instructions;

b) For reasons of economy and convenience the device inputs and outputs should be compatible with the logic circuit family of the DDP516 interface hardware. In this case 74 series TTL, Texas (1974);

c) The cost of the device must be less than the cost of implementing the circuit using standard discrete integrated circuits. This would have required 17 TTL integrated circuits to implement S.L.A.M.-16, costing about fifteen pounds. However, the increase in power supply requirements and printed circuit board area rule this as impracticable for a net of about 1000 elements;

d) The logic function should be consistent with that of S.L.A.M.-8.

2.1.1 The M.O.S. Process Used for S.L.A.M.-16

The integrated circuit designer is only concerned with the effects of the two dimensional physical changes in device geometry which he can make using photomasks. The other physical parameters such as impurity doping concentrations, diffusion depths and oxide thicknesses are optimised by the process engineer to make available a wide range of possible device characteristics to the circuit designer. This is done taking account of yield and replicability of the process.

The basic M.O.S. structure is shown in Fig 2.2. This produces p-channel enhancement mode metal gate transistors on an n-doped silicon substate. The processing steps to achieve this are as follows:

1. A layer of silicon dioxide is grown uniformly over the substrate in a oxidation furnace;

2. The oxide layer is etched back to the silicon substrate as defined by the source/drain diffusion photo-mask;

3. The source/drain regions are formed in a diffusion furnace in an atmosphere of Nitrogen containing Borane, (BH_3);

4. A second layer of oxide is grown over the substrate;

5. The oxide is etched back to the substrate at the active gate regions as defined by the gate oxide photo mask;

6. A carefully controlled layer of oxide is grown in the oxidation furnace to form the active gate regions;

7. The oxide is etched through to the source/drain diffusions to enable interconnections as defined in the "through connections" photo mask;

8. A uniform layer of Aluminium is evaporated onto the slice;

9. The Aluminium layer is etched away as defined by the "Aluminium interconnection" photomask to produce the finished structure, (Fig. 2.2).

The above process has the advantage of requiring the least number of processing steps of i.c. manufacturing processes and of requiring only four photomasks. The principal

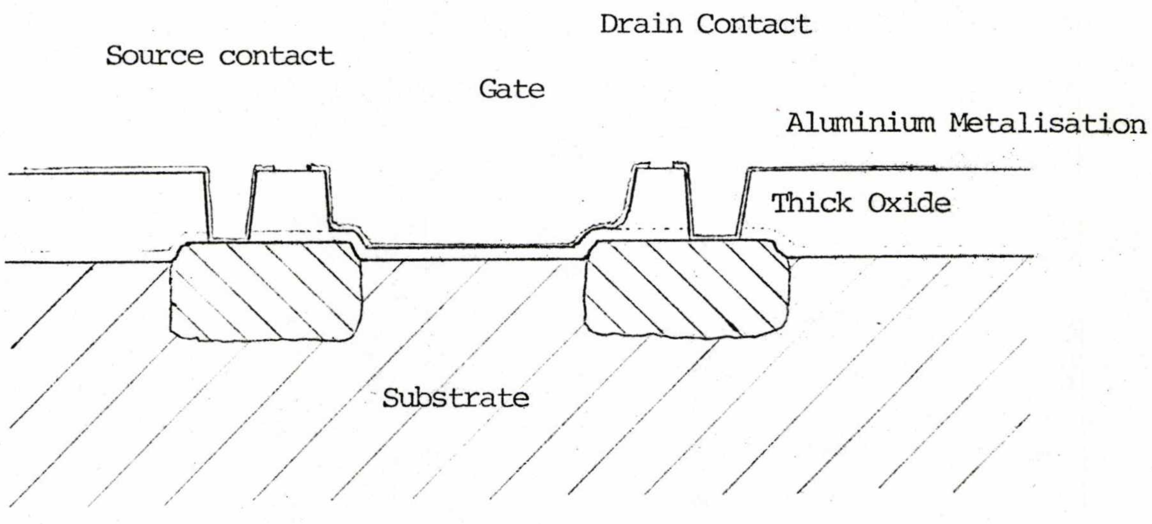


Figure 2.2 Cross-Section M.O.S. Structure

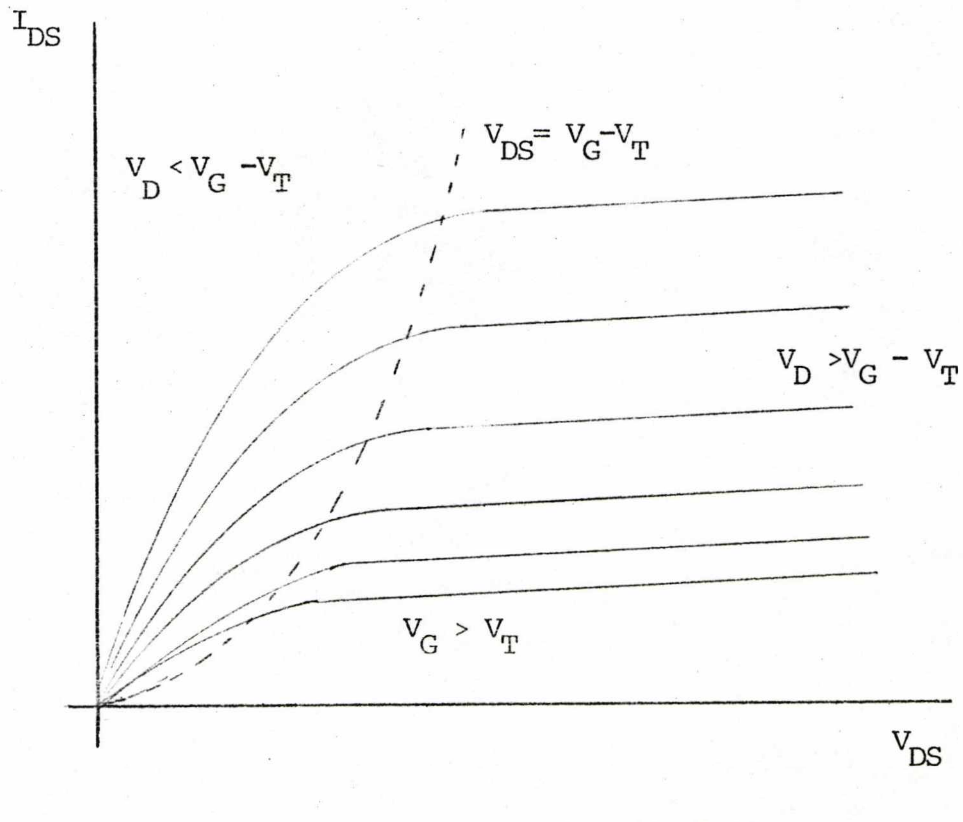


Figure 2.3 Typical M.O.S. Transistor Characteristics

disadvantage is due to the need for an overlap of the source and drain regions by the gate oxide and metalization. This produces parasitic capacities which in the case of the gate to drain capacitance is effectively multiplied by the gain of the stage due to feedback. This is overcome in self-aligning gate processes by growing a polycrystalline layer of silicon in the gate region instead of the metalization. However, the metal gate process used was capable of producing devices able to satisfy condition a) of the specification, (Section 2.1.0).

2.1.2 Logic Design Using M.O.S. Transistors

M.O.S. enhancement mode transistors are ideally suited to producing elegantly simple logic gates requiring no other components. If the source/gate voltage is below a threshold voltage, V_T, then the source/drain current, I_{DS}, will be a negligable amount representing leakage currents. The threshold voltage is dependent upon device geometry and substrate doping (and is controlled by the process engineer to provide maximum yield.) Gate voltages greater than V_T produce two modes of operation corresponding to saturated and unsaturated drain currents. A typical set of M.O.S. transistor characteristic curves is shown in Fig. (2.3).

The design equations necessary for M.O.S. logic circuits are given below.

The current I_{DS} is given by:

$$I_{DS} = \beta(V_D V_E - V_D^{2}/2) - \qquad (\S 2.2)$$

$$\text{for } V_D<V_E, \quad V_E = V_G - V_T \quad ,$$

And,

$$I_{DS} = \frac{\beta V_E^{2}}{2} \quad \text{for } V_D>V_E \text{ (Saturation)} \qquad (\S 2.3)$$

β may be regarded as a constant dependent only on device parameters although it increases slightly according to the amount of reverse bias between the drain and the substrate. It is given by:

$$\beta = \frac{\varepsilon\varepsilon_o \quad \mu}{t_{ox}}\frac{W}{L} \quad , \ \beta = \beta^{*}\frac{W}{L}, \qquad (\S 2.4)$$

where ε_o is the relative permittivity of the gate oxide, μ is the hole mobility in the channel region, $\frac{W}{L}$ is the aspect ratio of the gate and t_{ox} is the thickness of the gate oxide.

A M.O.S. inverter circuit is given in Fig (2.4)

The M.O.S. transistor, Q_L, is used as a load instead of a passive resistor since it requires a very much smaller chip area.

When Q_1 is not conducting the voltage at node $\overline{A}$ will be:

$$V_A(\mathrm{OFF}) = V_{DD} - (V_T + \Delta V_T), \qquad (\S 2.5)$$

where ΔV_T is a correcting factor caused by the substrate biasing effect mentioned above. It has to be determined empirically for the process used.

For V_A to be greater than V_T in order to drive subsequent stages, V_{DD} has to be at least $2V_T$.

When Q_1 is driven to saturation the following equalities hold:

$$V_A(\mathrm{ON}) = V_D(Q_1) = V_S(Q_L) < V_E(Q_1),$$

therefore $I_{DS}(Q_1)$ is given by §2.2 and $I_{DS}(Q_L)$ by §2.3. Equating and making a first-order approximation:

$$V_A(\mathrm{ON}) \simeq \frac{V_{DD}}{2\beta_L/\beta_I} \qquad (\S 2.6)$$

$V_A(\mathrm{ON})$ must obviously be less than V_T in order to satisfy subsequent stages. At intermediate values of V_A, Q_1 will be operating under conditions given by §2.3. An expression for the gain of the inverter can be derived which shows a linear transfer characteristic.

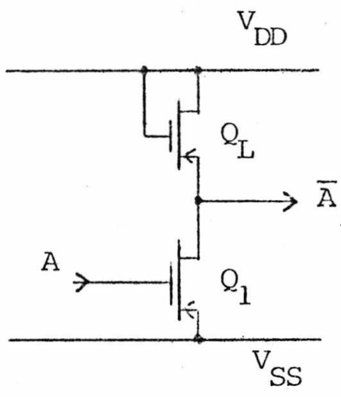


Figure 2.4 M.O.S. Inverter

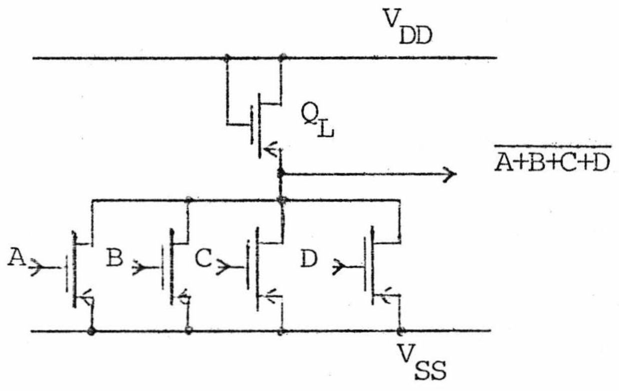


Figure 2.5 NOR Logic Circuit

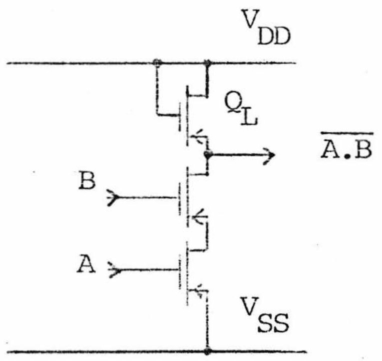


Figure 2.6 NAND Logic Circuit

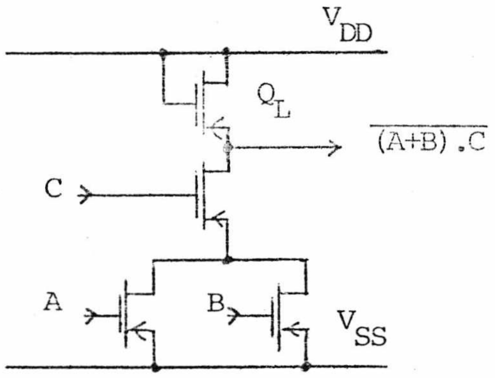


Figure 2.7 Compound Logic Circuit

Using $V_D = 2V_T$,

$$\frac{V_A}{V_{E1}} \approx -\left(\frac{\beta 1}{\beta L}\right)^{\frac{1}{2}} \qquad (\S 2.7)$$

The element in the ON condition I_{DS}(ON) is given by:

§2.3:

$$I_{DS}(ON) = \beta^* W_L \left(V_{DD} - V+ - V_A(ON)\right)^2 \qquad (\S 2.8)$$

Logic gates are implemented using a similar structure to the inverter of Figure 2. 4 . They take the form of NOR, NAND and compound gates. Examples of these are shown in Figs. 2. 5 , 2. 6 , 2. 7 . The NOR structure does not require any circuit design modifications when more inputs are added since V_A(ON) is at the worst case value with only one input conducting in an inverter. However, to preserve V_A(ON) in a NAND or compound structure the transconductances β_1 of all M.O.S.T.'s in series paths must be increased in order to preserve the overall conductance. This requires M.O.S.T.'s of larger gate areas, hence the NOR structure is prefered.

2.2 Circuit Design of S.L.A.M.-16

The logic circuit of S.L.A.M.-16 is given by Fig. (2.1). This is implemented by the M.O.S. circuit shown in Fig. (2. 8). The NOR structure has been largely adhered to except in the design of the steering logic used to write data into individual storage cells where a compound

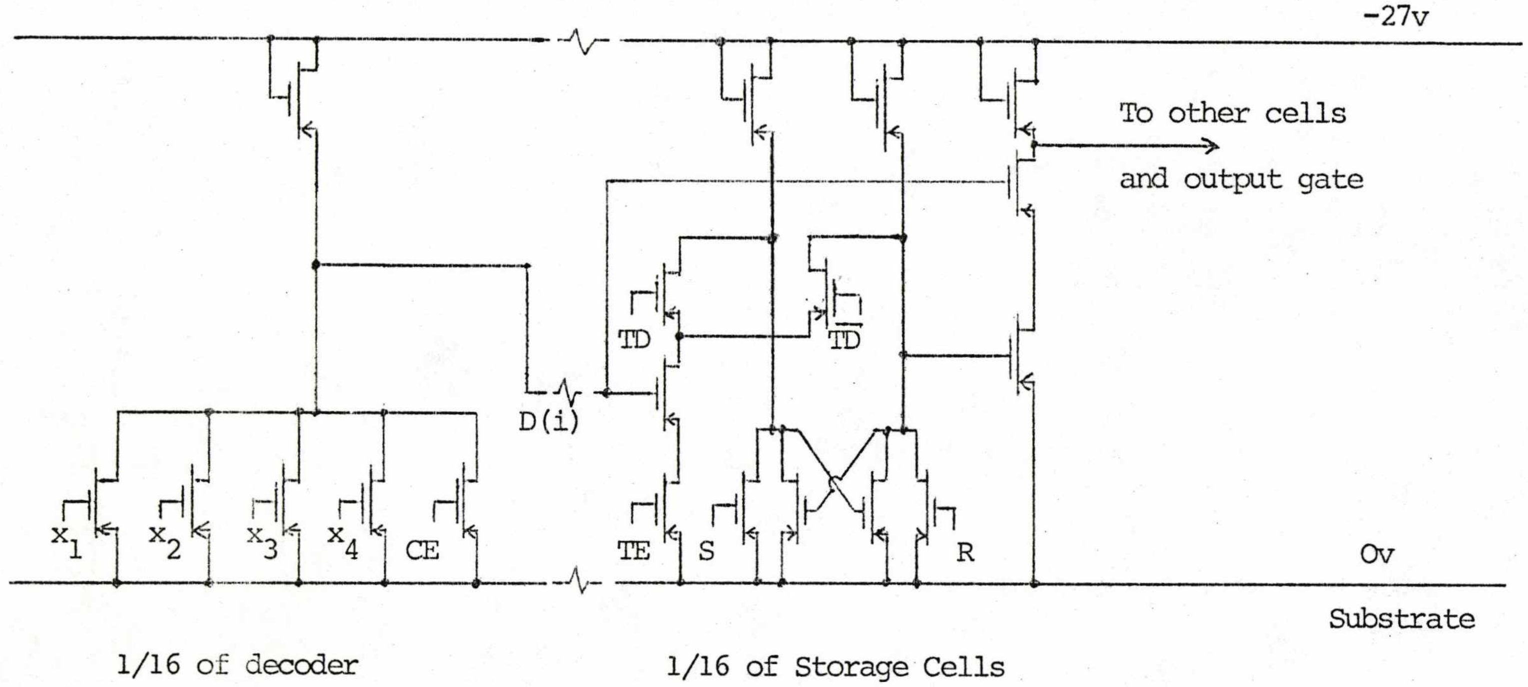


Figure 2.8 S.L.A.M.-16 M.O.S. Circuit

structure is found to require fewer M.O.S.T.'s.

The detailed circuit design is completed by substituting the process parameters given by Table 2.9 into the design equations presented in Section 2.1.2.

V_{DD} is chosen to be -27V. $\Delta V+$ - 1.6V, therefore V_A(OFF), from §2.5 = -20.4V. V_A(ON) is chosen to be -0.9V. to provide adequate noise margin. This requires that $\frac{\beta_i}{\beta_L} = 15$.

The current I_{OS}(ON) is then determined by considering total power dissipation. This is set to give a static dissipation of 100 mW. excluding interface circuitry. I_{DS}(ON) then becomes 75 μA requiring a load M.O.S.T. aspect ratio, $\frac{W_L}{L_L}$, of 1/6; this gives the aspect ratio, $\frac{W_i}{L_i}$, of a standard NOR structure M.O.S.T. to be 2.5:1.

2.2.0 Interface Circuitry

Production variations in the gate oxide thickness mean V_T can only be nominally defined to within ±0.5V. This would make simple level shifting circuits inadequate for interfacing to TTL type logic. This was overcome by the use of an on chip voltage comparator circuit which enables the S.L.A.M.-16 input switching threshold to be set by an external reference voltage input shown in Fig (2.11). The external reference voltage is offset by an internal reference voltage derived from V_T to enable consistent switching

M.O.S. Process Parameters

Capacitances

Diffusions to Substrate	Area	0.065	fF/μ^2
Diffusions to Substrate	Perimeter	0.315	fF/μ
Aluminium to Substrate	Area	0.023	fF/μ^2
Aluminium to Channel	Area	0.23	fF/μ^2
Aluminium to Diffusion	Area	0.12	fF/μ^2
Transconductance Const. β*		3.78	$\mu A/V^2$

Table 2.9

Photomask Layout Rules

Separation between diffusions	10μ	Minimum
Diffusion width	8μ	Minimum
Gate Oxide/diffusion overlap	2.5μ	Minimum
Gate Oxide width	8μ	Minimum
Diffusion/Aluminium contacts	8μ x 8μ	Minimum
Diffusion/Contact overlap	4μ	Minimum
Aluminium/Contact Overlap	2μ	Minimum
Gate Aluminium/Diffusion Overlap	4μ	Minimum
Gate Aluminium/Gate Oxide Overlap	4μ	Minimum
Aluminium line width	8μ	Minimum
Aluminium line separation	8μ	Minimum
Bonding Pads	100μ	square Minimum
Distance between bonding pads	48μ	Minimum

Table 2.10

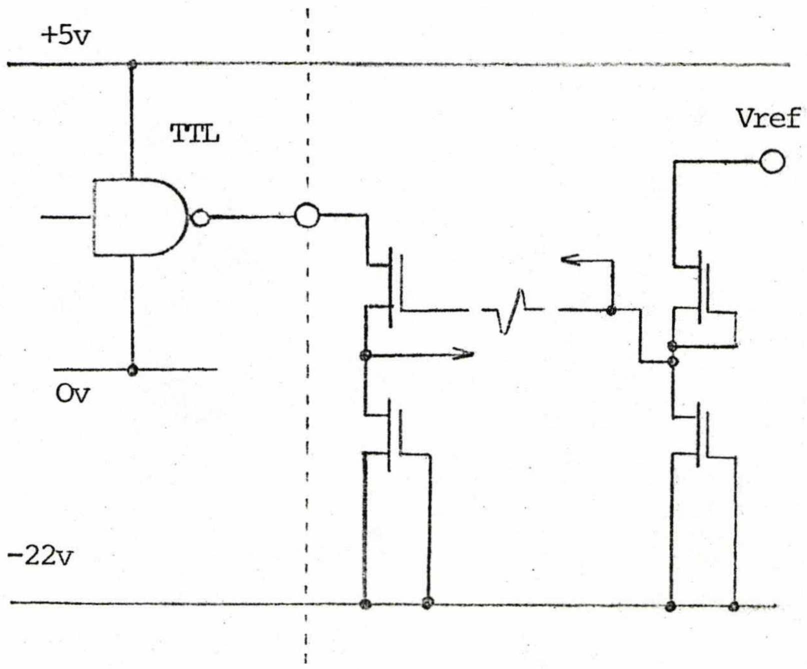


Figure 2.11 S.L.A.M.-16 Input Interface Circuit

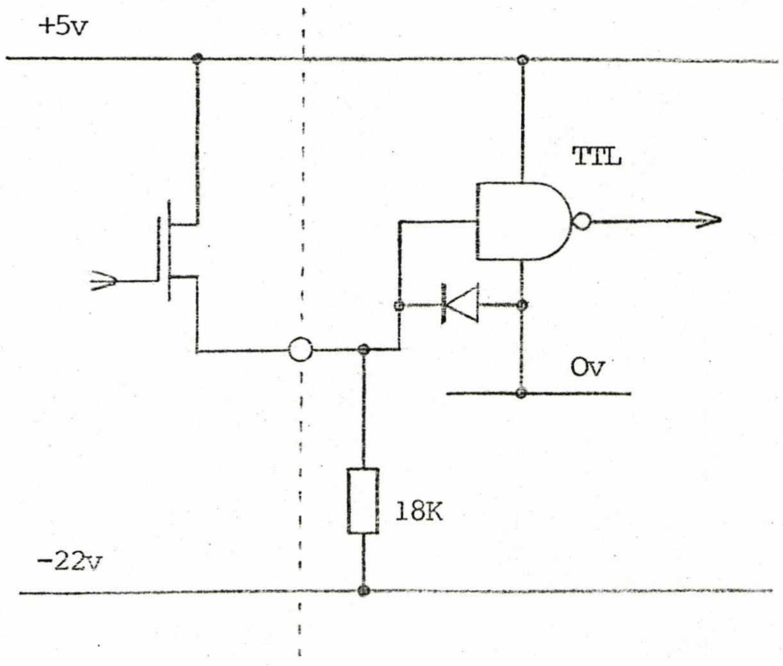


Figure 2.12 S.L.A.M.-16 Output Interface Circuit

thresholds to be set independently of production batch variations. This circuit also has the advantage that the package input terminals do not connect to high-impedance gate inputs which would be liable to damage by static electricity.

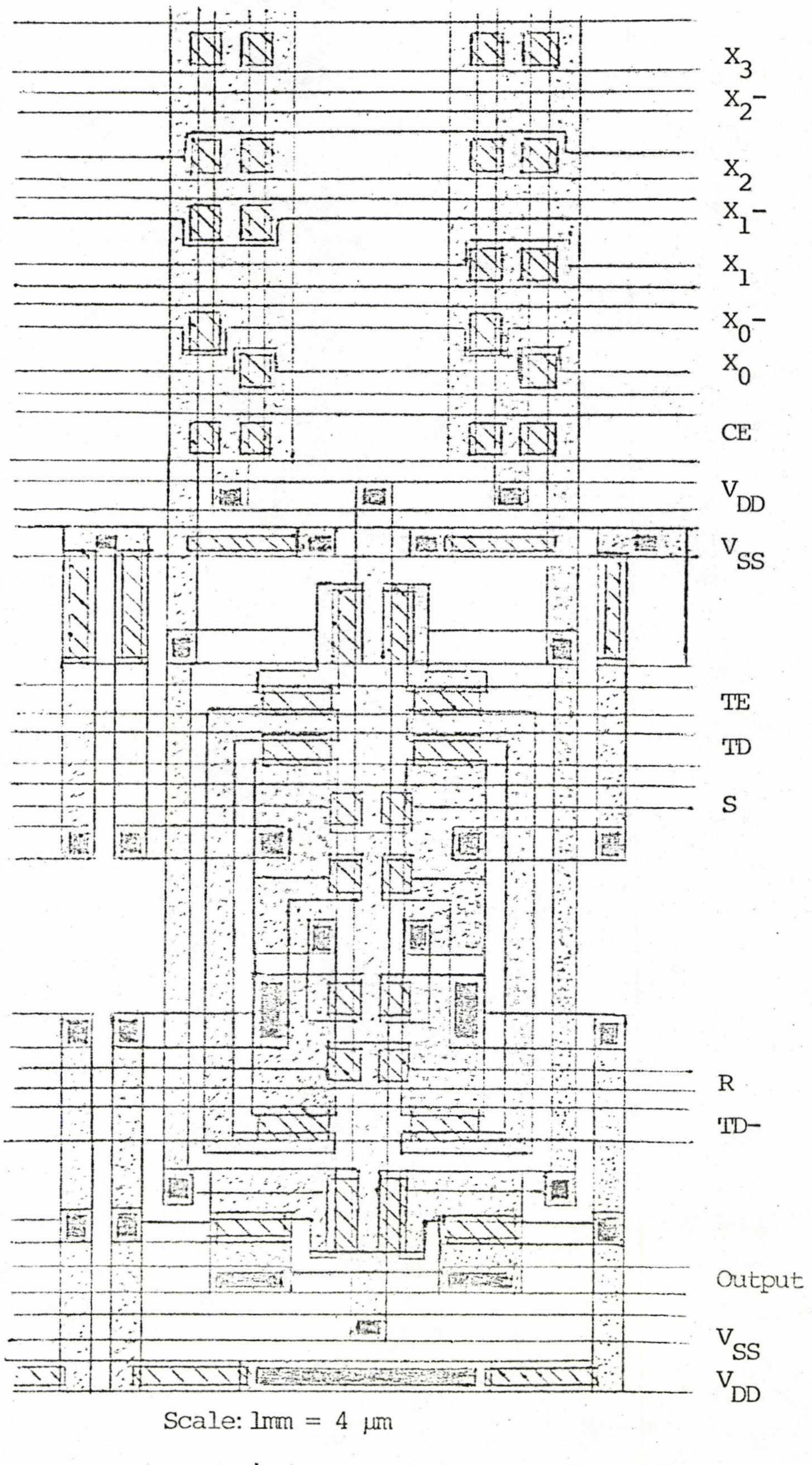
The guaranteed logic '1' noise level of TTL is 1.2V which is the minimum voltage swing which must operate the interface circuit. To achieve adequate gain to operate subsequent stages $\frac{\beta i}{\beta L}$ for the input interface is set to $\frac{1}{64}$.

Output interfacing to TTL is achieved by the circuit of Fig.(2.12). The open drain output M.O.S.T. requires sufficiently high conductance in the ON state to sink 1.8mA while saturating at a voltage lower than -3.6V in order to drive one 74 series TTL input. This uses a M.O.S.T. of aspect ratio 25:1

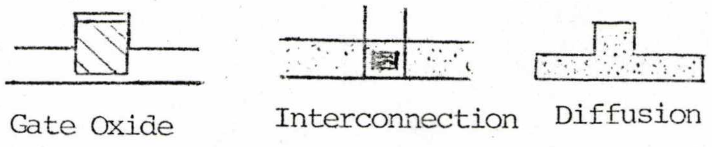
2.2.1 Photo-mask Implementation for S.L.A.M.-16

The circuit design is realised as a four-level two-dimensional layout using the layout rules given in Table (2.10). The source drain separation, L_i of the NOR structure M.O.S.T. is set to be ten microns and all other active dimensions calculated from this. An example of the layout of two storage cells and their associated gates is given in Figure (2.13).

Figure 2.13 Photomask Layout of Two S.L.A.M.-16 Storage Cells



Scale: 1mm = 4 μm



The finished design is then prepared as lists of co-ordination points for input to a numerically controlled graph plotter. The co-ordinate data is first checked and prepared by a computer program which enables the repetition of common cells after transformations through the horizontal and vertical axes. This facility is used to the greatest effect in the layout of the storage cell where only one was designed and the others formed by reflecting through both axes and placing the combined cells at four suitable origins.

The photomask artwork is drawn at a scale of 2000:1 and then photographically reduced in two stages, the final reduction being done by a step and repeat camera to produce the multiple photomask for a slice. A photomicrograph of a processed chip is shown in Fig. (2.14). The dimensions of the chip are 2mm square and are produced on 3.2cm diameter slices, shown in Fig 2.15, containing 160 chips.

2.3 Device Testing

After the processing steps outlined in Section 2.1.1. are completed the S.L.A.M.-16 devices are subjected to four levels of testing outlined as follows:

1. A standard dimension M.O.S.T. included on the chip layout is probe-tested to check that the process parameters are within the design tolerances;

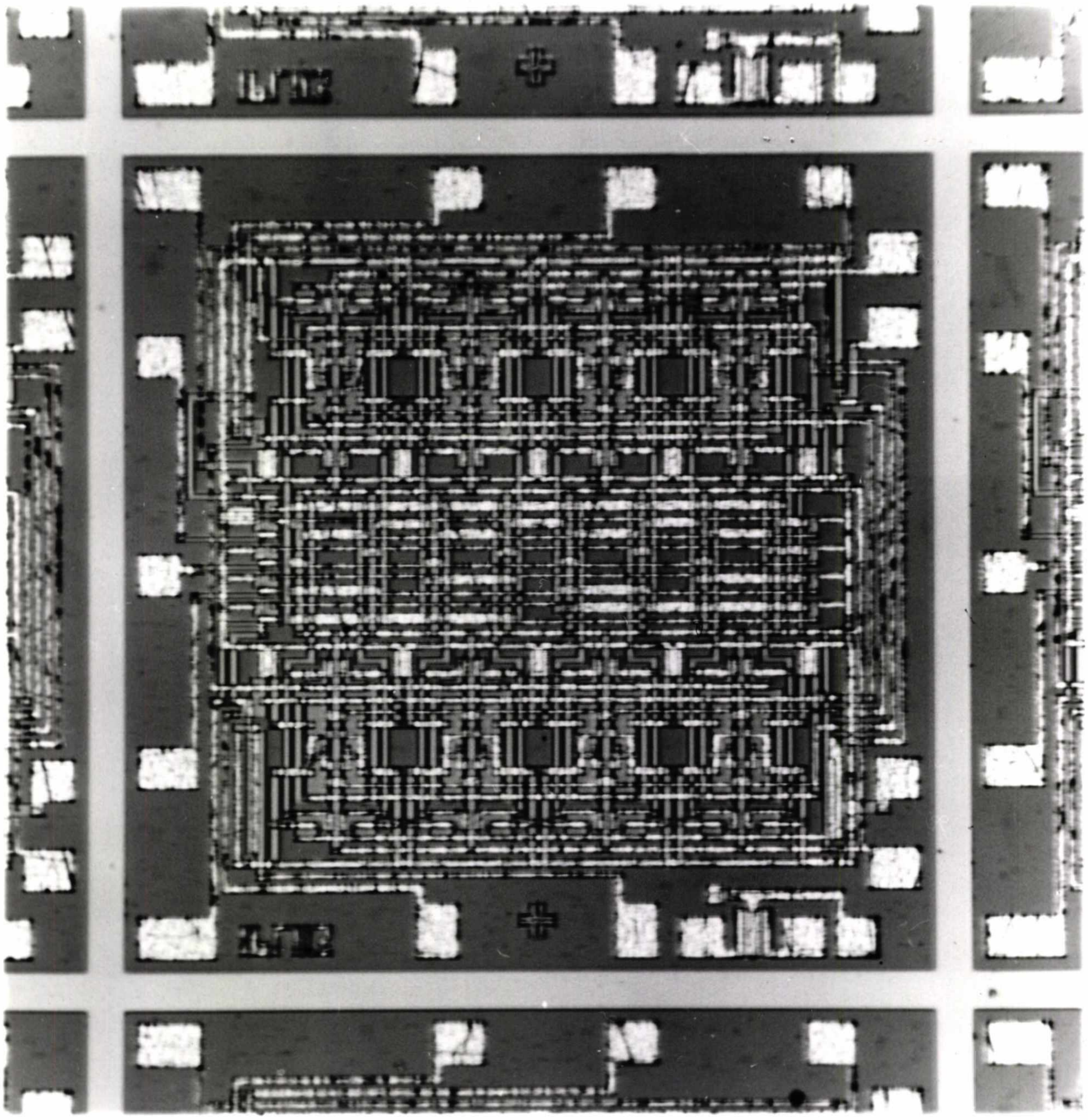


Figure 2.14

S.L.A.M.-16 Photomicrograph

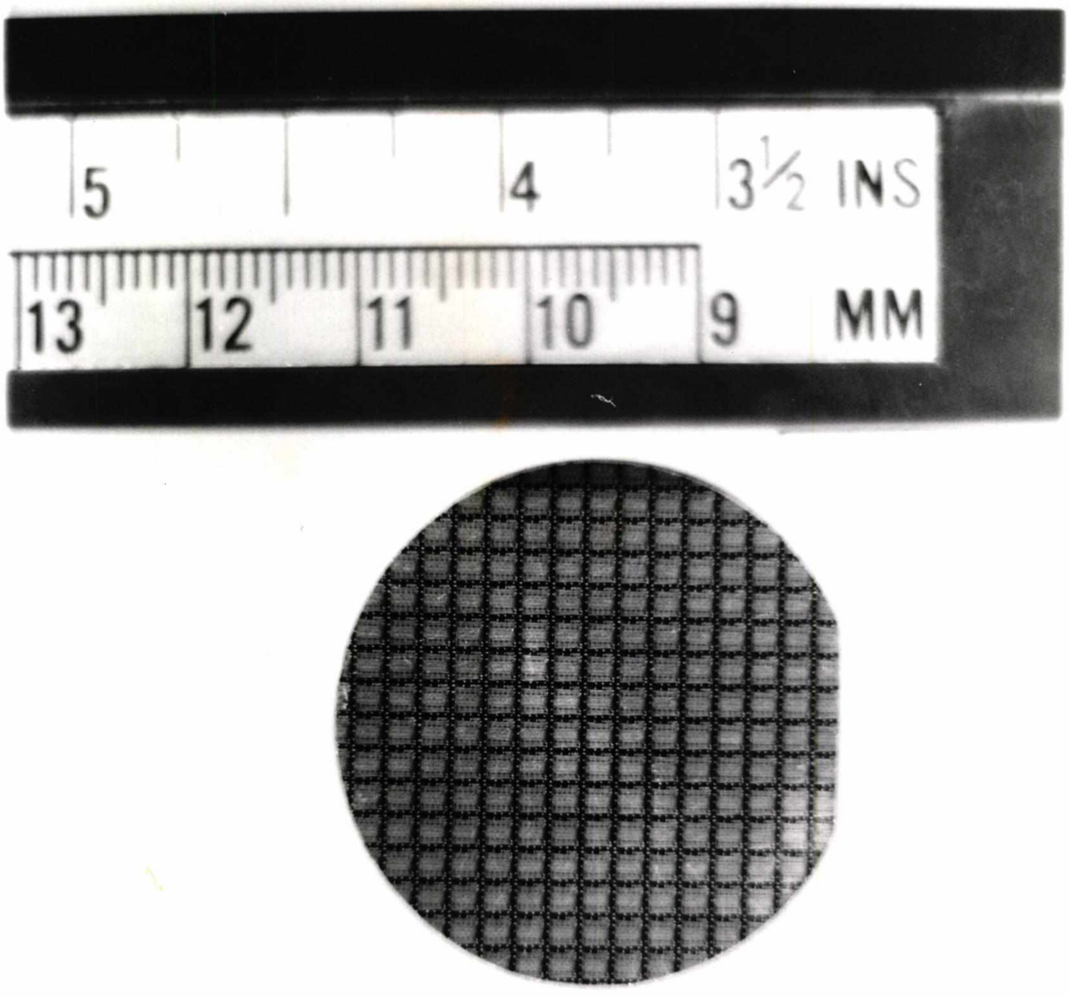


Figure 2.15
Slice of S.L.A.M.-16

2. Automatic probe testing equipment is used to check the logic function of each chip on the slice and ink mark the unusable chips;

3. The usable chips are encapsulated and then tested at worst case TTL logic levels;

4. A sample from each production batch is electrically stressed by operating with V_{DD} at -30V and at a temperature of $70^{o}C$.

The author was concerned, with the design of automatic test equipment for stages (2) and (3).

2.3.0 Test Sequence Design

Exhaustive testing of a complex logic circuit is rarely feasible. S.L.A.M.-16 has 2^{16} internal states and 9 inputs giving 2^{9} input patterns able to effect state transitions; an exhaustive test would require about 2^{25} steps entering each state via all the possible routes. A useful subset of test sequences has to be found using *a priori* knowledge about the most likely fault modes. This can be done bearing in mind that a fault is caused by some physical defect in an essentially two-dimensional structure. The circuit is partitioned into a number of simpler,easily-tested cells and malfunctions are only looked for in the interactions between physically adjacent cells.

Reduction of the test set size is possible with S.L.A.M.-16 by considering each storage flip-flop and its associated address decoder as an individually testable cell by using the control functions set, reset, teach, teach enable and device enable, leaving each cell in a known state with respect to the other cells. Interaction anomalies are checked for by testing the state of the adjacent cells. The photomask implementation organises cells in two separate columns which can be considered as being one-dimensional; each cell has therefore only two neighbours. The cells are ordered such that a binary code applied to the device inputs enables the addressing of adjacent cells.

2.3.1 Test Equipment Design

The chip testing equipment consists of a commercial step and repeat tester with needle probes for making contact with chip bonding pads. This provides a maximum of 16 needle probes that are manually aligned with the bonding pads with the aid of a microscope. An automatic inking device is provided for marking faulty chips. The slice containing the i.c. chips is held on a movable table by air suction. The table is then aligned manually to ensure that the direction of the rows of chips is parallel to the direction of movement of the table. The table is then programmed to step in the horizontal and vertical directions the appropriate distances between chips. Once the manual setting up procedure is complete the stepping movements are generated by the device testing logic.

The logic, shown at Figure 2.16, required to test the chip and control the automatic probe tester was designed using the test sequences shown in the timing diagram of Figure (2.17).

The probe tester automatic cycle proceeds as follows: a 'start of test' signal is generated by the automatic probe tester when a chip is correctly centered under the needle probes and the table has been raised so as to bring the probes in contact with the chip bonding pads; the test sequence is output to the chip via the probes and a signal to operate the inking system is sent to the probe tester if a fail condition is detected after analysis of the chip output; an 'end of test' signal is sent to cause the probe tester to step to the next chip in the horizontal direction. At the end of a row manual intervention is required to move the table to the start of the next row.

After encapsulation the finished devices are tested by a similar circuit implemented using TTL integrated circuits. Instead of generating tests for the required output, the output of the S.L.A.M. under test was compared with that of a known S.L.A.M. operating in parallel. This enabled a performance window to be implemented to give worst case propagation delays and voltage levels. The circuit of the comparison logic is shown in Figure (2.18).

The timing orders for the test sequence were derived from a 12-bit ring ccunter. The sequence is repeated once for

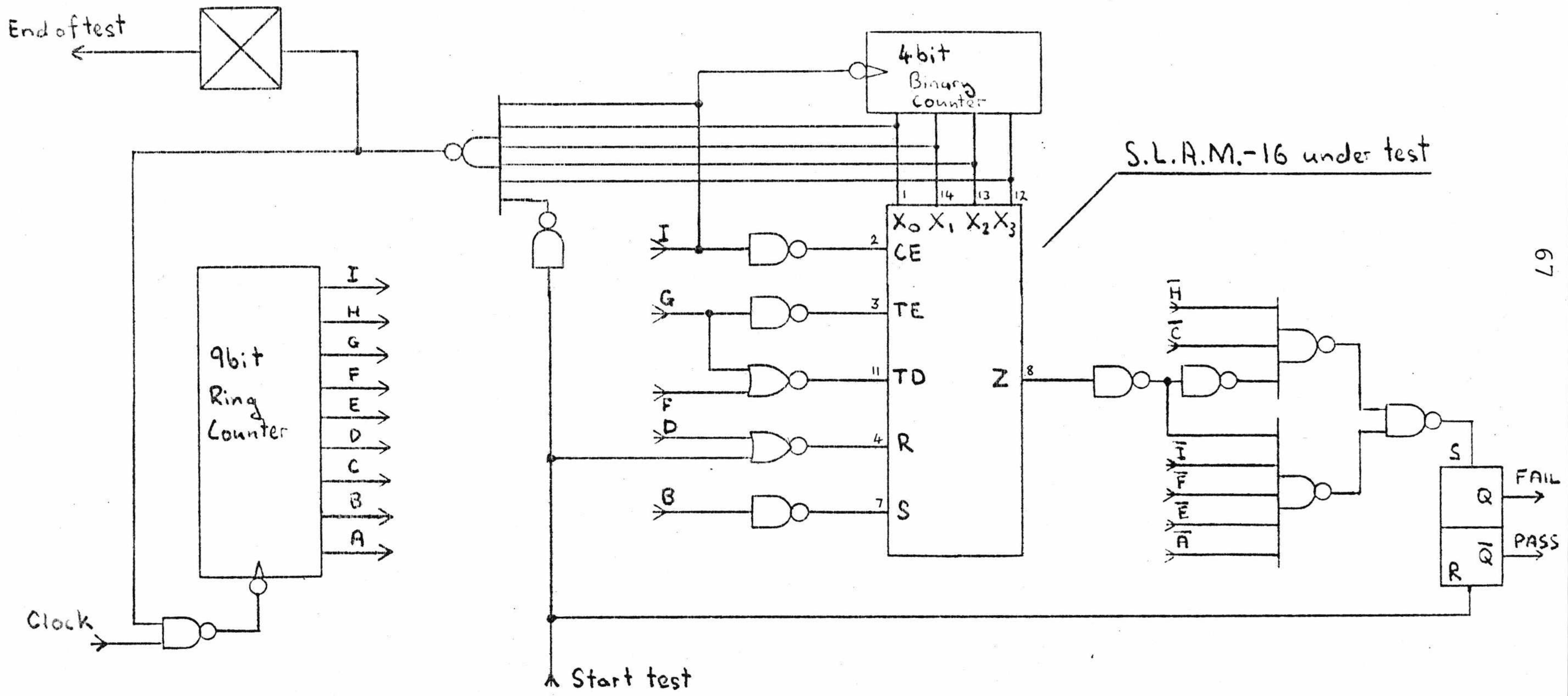


Figure 2.16 S.L.A.M.-16 Tester Logic Circuit

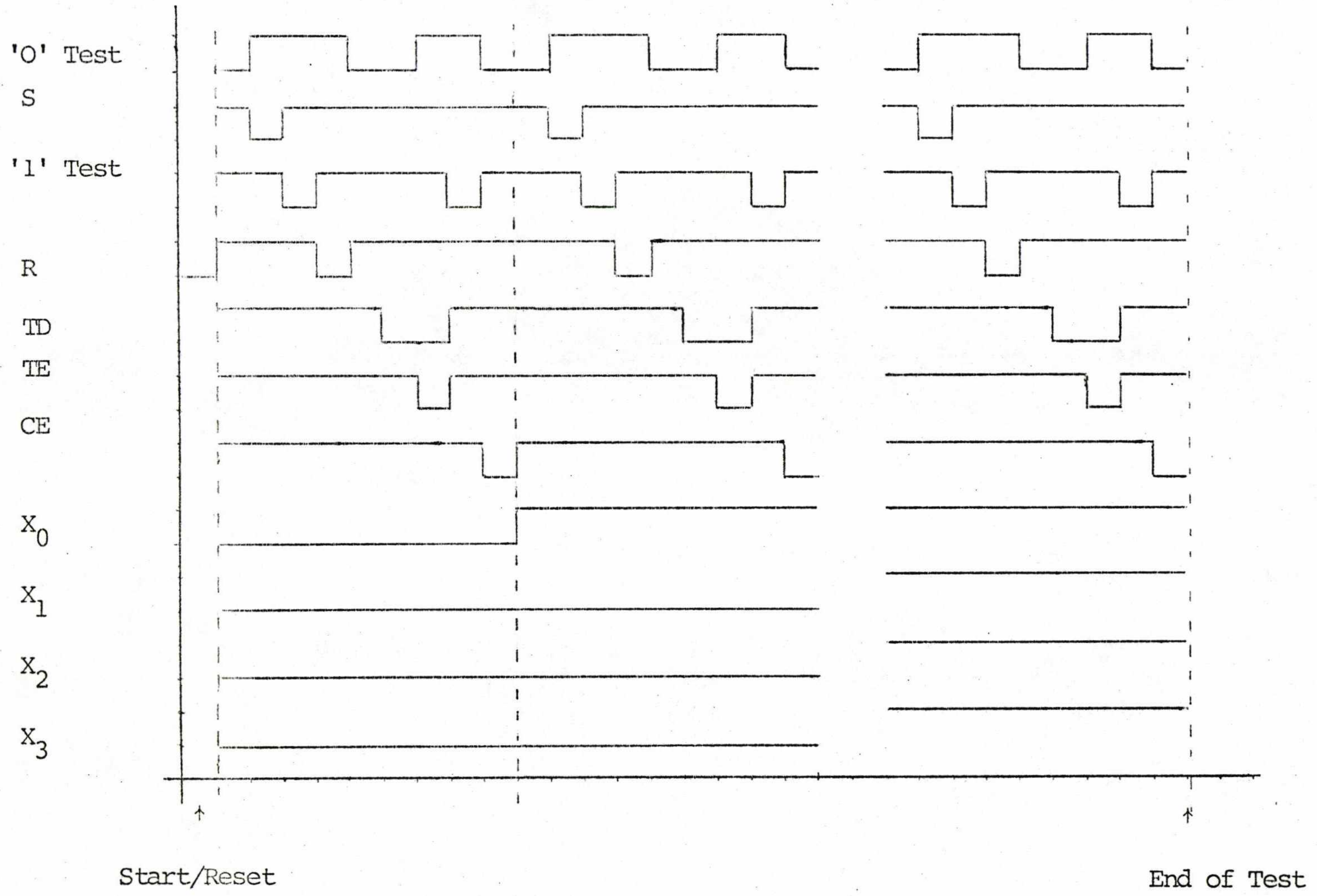
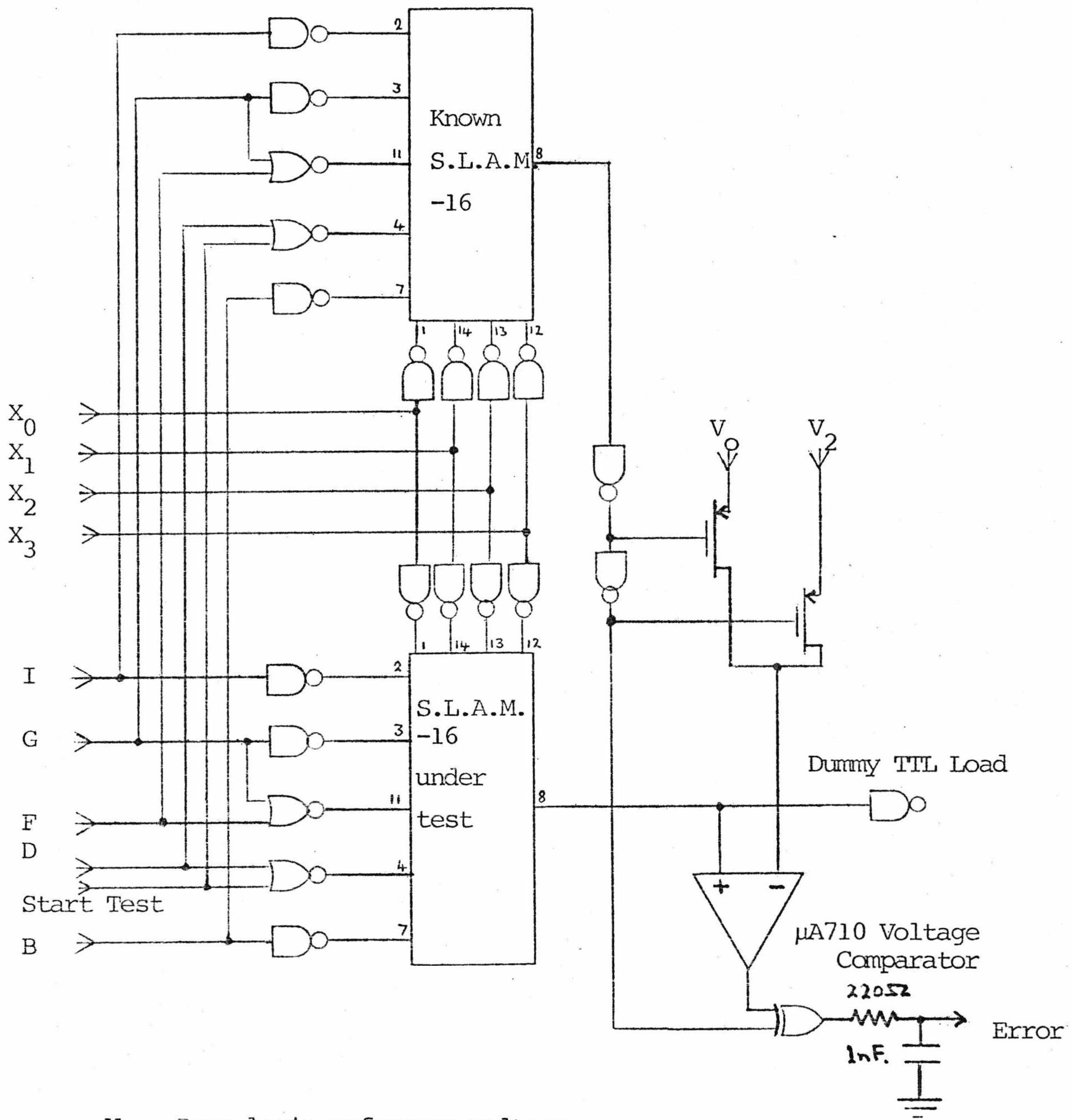


Figure 2.17 S.L.A.M.-16 Tester Timing Sequences



V_0 - Zero logic reference voltage

V_1 - One logic reference voltage

Figure 2.18 S.L.A.M.-16 Tester Output Comparison Logic

each possible S.L.A.M.-16 input minterm generated by a 4 bit binary counter. The test sequence first checks that the contents of a memory cell is reset. This tests for interaction between the neighbouring cells and that the reset input is operating correctly. The control functions are than tested leaving all cells reset except the one under test. The counter is then incremented to address the next cell. The end of test, (EOT), state is detected as the last state of the ring counter and the 1,1,1,1, state of the S.L.A.M. input counter. This is used to inhibit the ring-counter clock.

2.3.2 Performance of Produced S.L.A.M.-16 Devices

A sample of devices were characterised and the results are given in Figure 2.19.

The behaviour of the device was generally as expected from the design equations. Propagation delays were predicted by the extrapolation of results obtained by a computer simulation of a normalised M.O.S. inverter stage and were consistent with an actual S.L.A.M.-16 propagation delay of 75nS. per level of logic.

One area of discrepancy however was found in the expected performance of the open-drain output M.O.S.T. This was found to saturate at a lower current than the design figure but still gave TTL drive capability with reduced noise immunity.

On reinspection of the photomask design this is thought

Characteristics of a Sample of S.L.A.M.-16's

Output characteristics: R_L = 22K, V_{SS} = +5v., V_{DD} =-22v., T = 20^oC.

SLAM No.	Logic '1' voltage	†rise	†fall
1	2.8	1.1μS	0.8μS
2	2.8	1.1μS	0.8μS
3	2.9	1.1μS	0.8μS
4	2.8	1.1μS	0.8μS
5	3.2	1.0μS	0.8μS
6	3.3	1.0μS	0.8μS
7	3.3	1.0μS	0.8μS
8	3.3	1.0μS	0.8μS
9	3.4	1.0μS	0.8μS
10	3.3	1.0μS	0.8μS

Input characteristics: R_{IN} = 2.2K, V_{SS}= + 5v., Vref = 0v., V_{DD} = -22v.

Switching Threshold Voltage

SLAM No.	X_0	X_1	X_2	X_3	S	R	TD	TE	CE
1	2.0	2.0	2.0	1.9	1.9	1.9	1.8	2.1	1.9
2	2.0	2.0	2.0	2.0	1.9	1.9	1.9	2.0	1.9
3	2.0	1.9	1.4	1.9	1.9	1.85	1.8	2.1	1.9
4	1.8	1.8	1.8	1.9	1.8	1.7	1.7	1.9	1.8
5	2.0	1.8	1.8	1.8	2.1	1.8	1.8	1.0	1.8
6	1.9	1.8	1.8	1.9	2.0	1.8	1.8	1.9	1.8
7	1.9	1.9	1.8	1.8	1.9	1.9	1.8	1.9	1.8
8	1.8	1.8	1.8	1.9	1.9	1.8	1.7	1.9	1.8
9	1.8	1.9	1.9	1.9	2.0	1.9	1.8	1.9	1.9
10	1.8	1.8	1.8	1.8	1.9	1.7	1.7	1.8	1.8

Table 2.19 S.L.A.M.-16 Characteristics

to be due to the high resistance of the M.O.S.T. source diffusion which was unfortunatly miscalculated during the photomask realisation.

The yields of the pre-production batches tested by the author were as follows:

Chip Yield:

Total number of testable chips per slice	= 160
Av.number passing chip tests from 6 slices	= 110

chip yield 69%

Total number of encapsulated devices = 572

2.4 A Discussion

The introduction of medium and large scale integration has resulted in a split between the interests of the logic circuit designer and those of the logic system designer. When logic systems were implemented using discrete components or small scale integration, (SSI), the goals of circuit design and those of systems design coincided, namely to produce the minimal design in terms of the number of components required. The high capital costs involved in integrated circuit manufacture have led MSI manufacturers to select logic circuits for integration from existing minimal designs produced by the previous generation of logic system designers. This was to guarantee an immediate large market from users of conventional design methods. The criteria for selection have resulted in MSI building blocks of specific function, such as parallel adders, parity generators, shift registers, etc., whose function may be expandable, but are of little interest to the designer of random logic. Whilst the result of M.S.I. has been to dramatically reduce the cost of a digital system to the user it has necessitated an unsystematic approach to systems design on the part of the logic systems designer whose job has become the respecification of the design problem in a way that can be realised by the current M.S.I. systems components.

Some useful M.S.I. components for general logic implementation have appeared quite accidentally, such as the binary decoder, specifically intended for address decoding or de-multiplexing; this can be used as an universal min-term generator. The multiplexer is specifically intended for routing binary signals but it can be used as a universal logic gate (c.f. Section 2.0).

Faced with a situation where it is not possible for the systems designer to utilise existing M.S.I. circuits economically he is obliged to use custom design for an M.S.I. circuit. Here there are two main pit-falls; working within a fixed budget and unfamiliar with the problems faced by the circuit designer a conservative approach to the design may be taken. This was essentially the approach used for the production of S.L.A.M.-16. It has the drawback that the device may become rapidly obsolete because of later technological innovation. Present day M.O.S. static R.A.M.'s contain 256 times the storage of S.L.A.M.-16, operate 5 times faster, require one +5v supply, are directly compatible with TTL requiring no external components and cost 50% less than S.L.A.M.-16. These were available just 6 years after S.L.A.M.-16.

The alternative approach of including technical innovation and systems innovation removes overall control of the design from the systems designer. This can lead to conflicts between the interests of circuit and systems design which make the design costs unpredictable.

The more recent introduction of user programmable R.O.M.'s. (P.R.O.M.), electrically alterable R.O.M.'s (E.A.R.O.M.'s) and programmable logic arrays, (P.L.A.'s), together with single chip micro-program sequencers promise to enable economic realisation of special purpose processors designed by formal methods.

CHAPTER 3

The Design and Construction of Learning Network Hardware using S.L.A.M.-16.

3.0 Introduction

The aim of this chapter is to describe in some detail the factors influencing the design of the learning network hardware. The structure of the machine is presented in Section 3.3 and the link between the machine hardware and the assembly language instructions in a control computer required to drive it is demonstrated in Section 3.3.2. Details of the performance of the machine are given in Section 3.2.3.

3.1 General System Design Considerations

The designer of a practical learning network is required to solve engineering problems whose magnitude increase combinatorially as the number of network elements increases. Although the hardware is constructed from homogeneous components the complexity arises from parallelism and lack of specification of the network structure. In order to realise non-trivial network hardware within the limitations of cost, restrictions must be placed upon the generality of network structure and parallelism with regard to the external presentation of data and evaluation of network response. Many decisions concerning the systems implementation had to be taken arbitrarily, due to the breadth of the network problem and the inadequacy of the results obtained by simulation programs in providing a general philosophy for implementing useful network structures. The structure of the network hardware described in this chapter is the result of a synthesis of several special cases of network structures investigated by simulation. Examples of these simulated networks are presented in the following paragraphs to demonstrate a paradigm for our network structure.

3.1.0 Single-layer Networks

These are so called because the network is limited to a combinational structure where no element inputs are connected to the outputs of any other elements. Processing by a single-layer learning network is independent of the order in which testing patterns are presented. This implies that in a hardware implementation race conditions can not occur and no internal storage of the individual element responses is required.

Cheung(1973) has investigated the feature extraction properties of randomly-connected single-layer networks. The networks are organised as a hardware implementation of the well known n-tuple pattern recognition method of Bledsoe and Browning (1959). Random n-tuple features are extracted by presenting a training set of patterns to the inputs of a randomly-connected single layer-network and training all elements of the network to respond with a one output for each training pattern. A pattern discriminator using single-layer networks is shown in Figure 3.1. An individual network is required for each training class. Testing patterns are presented to all the networks and the particular response class is obtained by polling the outputs of the networks for the maximum 'ones' response. Of interest in the study of these networks is the elimination of redundant features. This requires mechanisms whereby the content of the S.L.A.M. element stores can be evaluated and appropriate re-connecting of the network inputs done. Cheung's work was done using handwritten alpha-numeric characters. This has been extended to pattern recognition systems for chemical mass spectra data by Stonham (1975). Stonham also considers two-layer networks in which the response pattern of the first layer of elements is used as the input to a second single-layer network.

3.1.1 Feedback Networks.

Natural state-space clustering during unsupervised training of randomly connected feedback networks is a property demonstrated by Fairhurst (1973). This requires a more complicated network structure, with the outputs being OR'ed with the input data after a delay, (Figure 3.2). Each training pattern is derived by sampling the input data and the training of the individual elements is increasingly inhibited as training progresses. (An implementation of Fairhurst's networks using the MINIC system is discussed in Chapter 4. Section 7.1.).

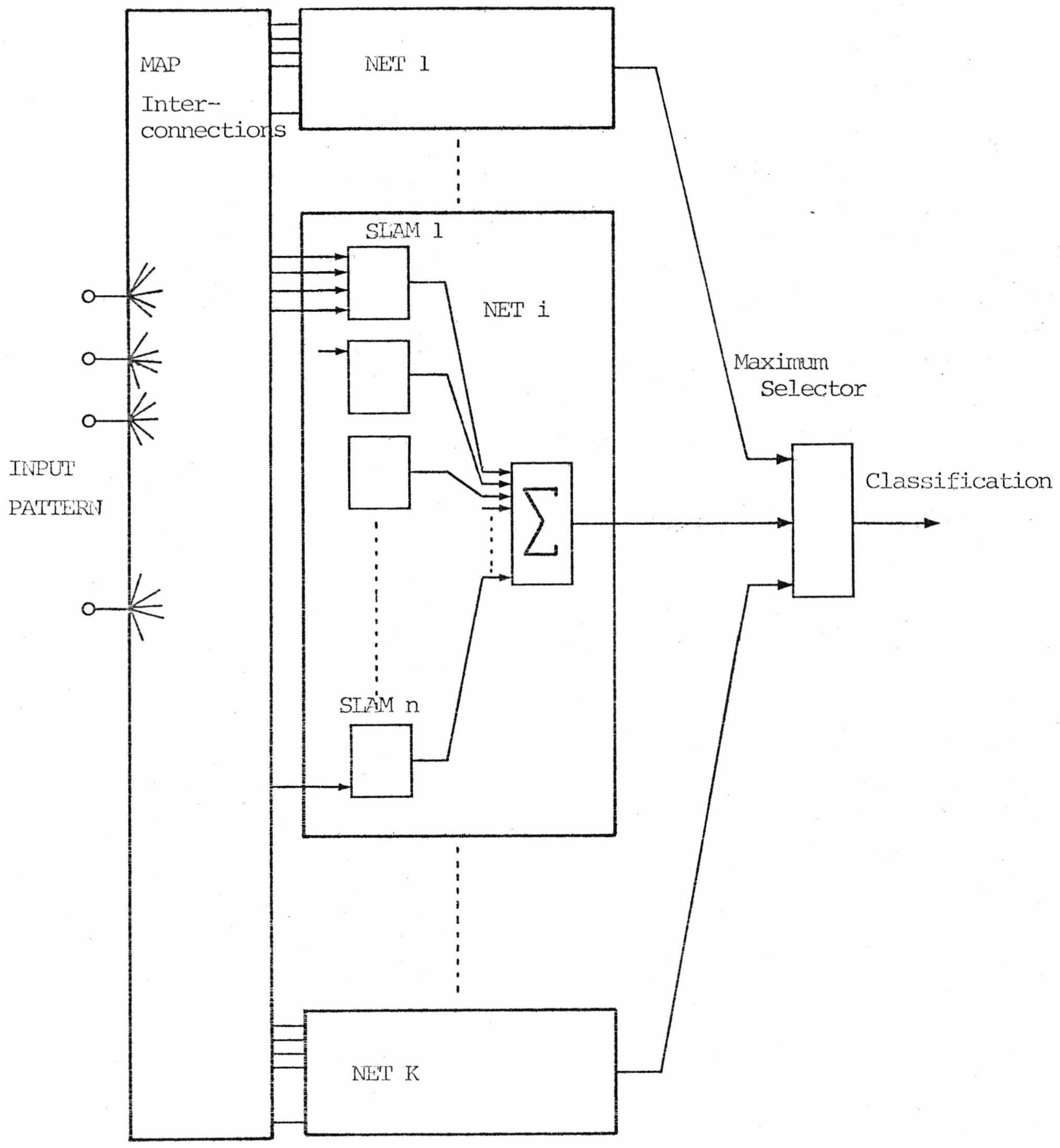


Figure 3.1 A Single Layer S.L.A.M. Pattern Discriminator

Reeves (1973) uses networks to control the visual tracking of simple shapes. His networks have to be capable of operating in a real-time situation by following a human operator's performance during training and replicating this achievement in previously unseen situations.

Feedback networks are necessary to resolve ambiguities of direction at intersections.

3.1.2 Other Network Structures

Work by Aleksander and Wilson (1975) concerns regular connected networks with homogeneous functions. These networks are capable of executing existing image processing algorithms, such as line thinning. However the adaptive properties of the R.A.M. network promises the identification of algorithms using networks with non-homogenous functions. The structure of a R.A.M. regular network is shown in Figure 3.3. In these networks a fixed scheme of interconnections between nearest neighbours is used.

3.2 A Generalised Network Model

A proposed general-network-system model capable of implementing the systems described above is given in Figure 3.4. The design problems of implementing such a system in hardware consist of optimising the parallelism of the data and control paths against the required flexibility and the limitations of cost. Figure 3.4 is separated by the dotted

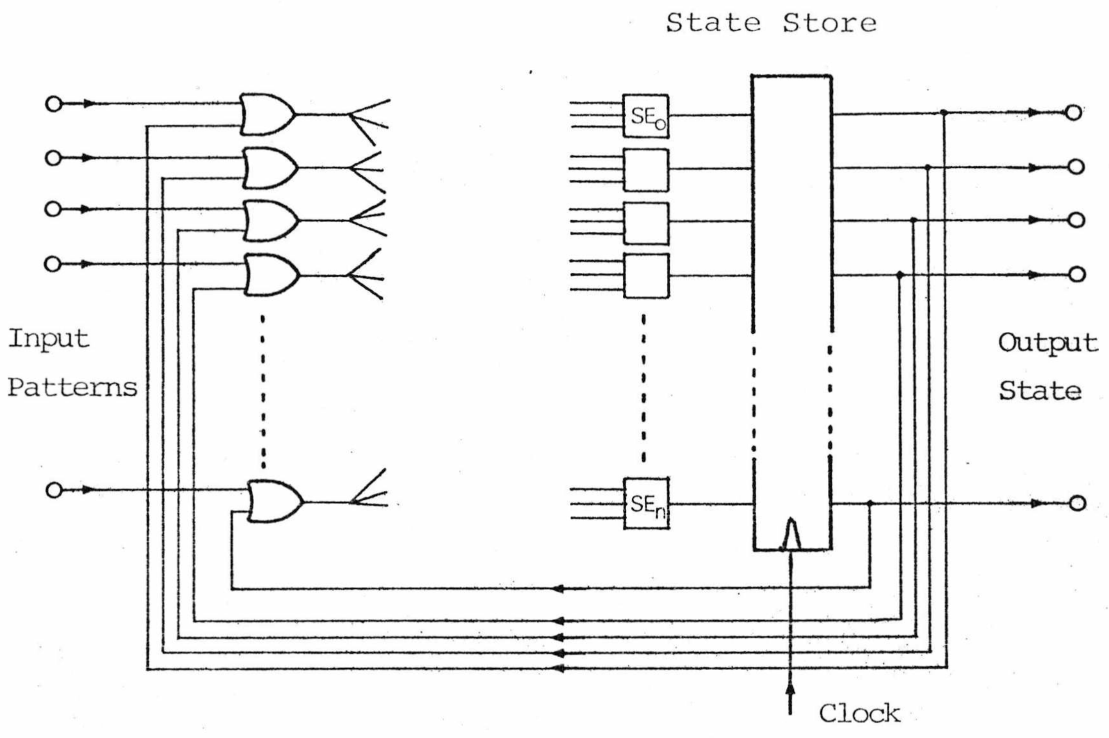
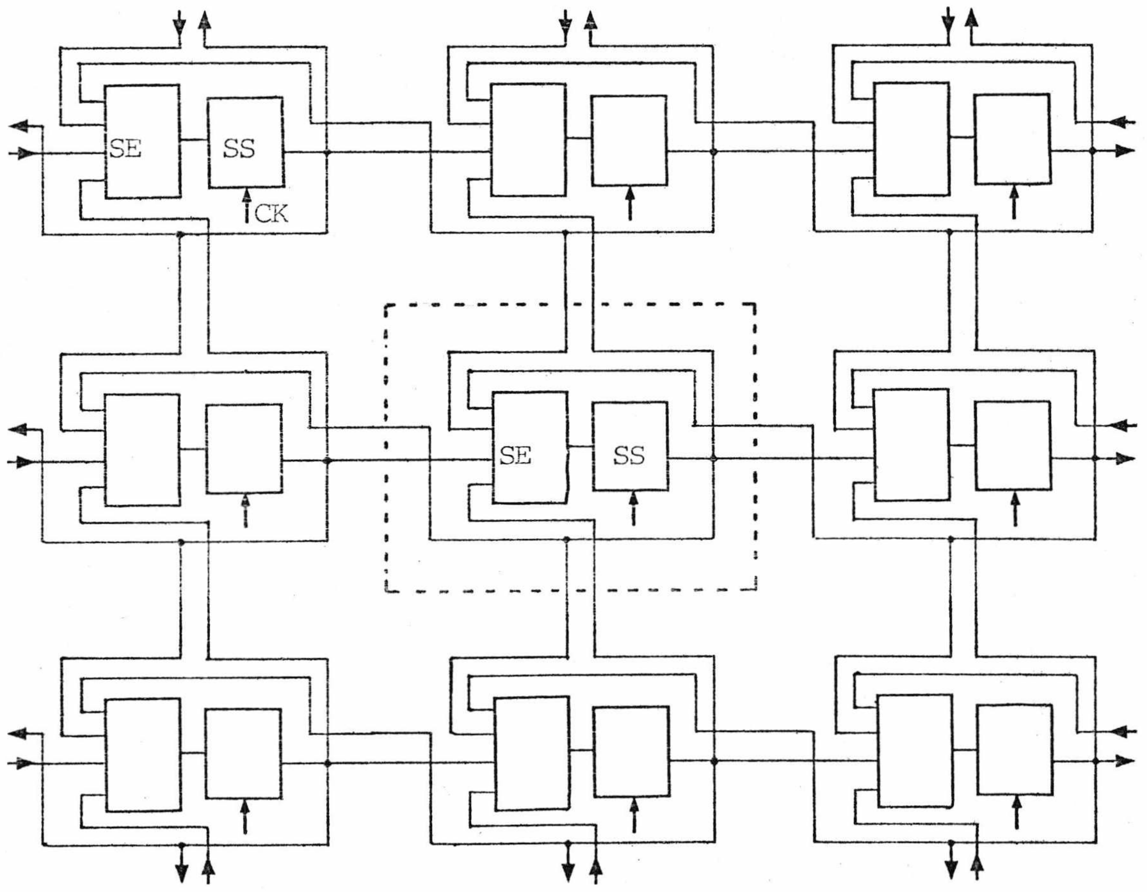


Figure 3.2 Fairhurst's S.L.A.M. Feedback Network

Figure 3.3 A regular Array of S.L.A.M. Elements



SE - S.L.A.M. Element, SS - State Store

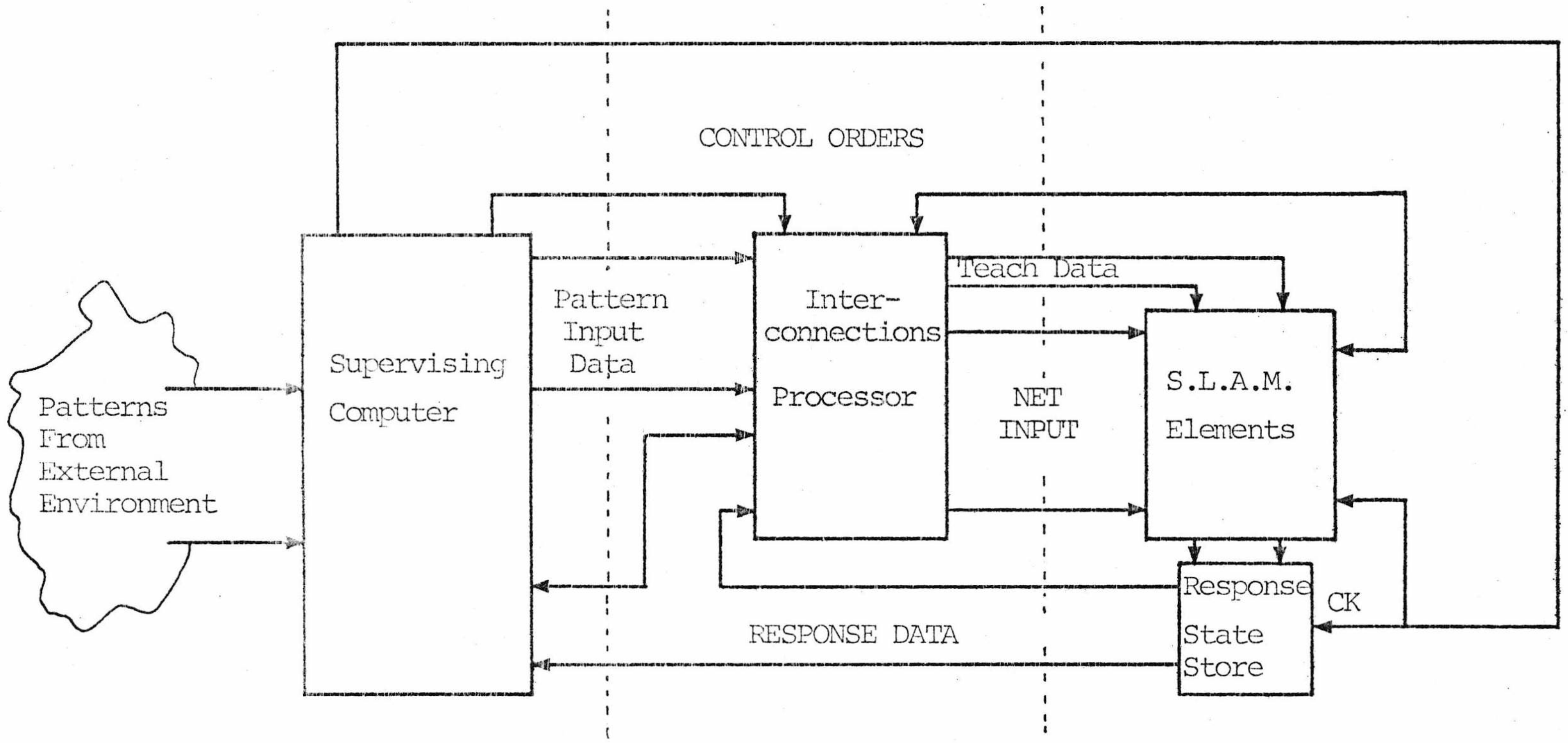


Figure 3.4 Schematic Diagram of a Generalised S.L.A.M. Element Learning Network

lines, into 3 sections presenting differing engineering problems in their implementation.

The network section itself consists of the output state store, S.L.A.M.-16 elements and the Interconnection Mappings. A fully parallel implementation of the network-section will be shown later to be not necessarily advantageous unless the data and response can be presented and evaluated in parallel also.

The supervising computer provides the necessary control signals for the network and also provides a means of interfacing with external peripherals. The complexity required of the supervising computer is inversely dependent upon the amount of pre-defined hardwired interconnections and the degree of parallelism in the network. In this case a powerful mini-computer system (c.f. Appendix A) was available and the necessity for hardwired, and thus inflexible control small.

3.2.0 Network Systems Design Considerations

The upper bound on the through-put performance of the network is the rate at which information can be transferred between the external environment, the supervising computer and the network itself. Normally each data transfer to the network results in a response which must be transferred back to the supervising computer for analysis. However there are occasions when autonomous operation of the network is advantageous such as in a feedback network where one is not concerned with the transient portions of state-space but with stable state-cycles. In order to obtain speedy observation of state-cycles the output state is clocked a number of times without any data transfer to the supervising computer. Observations are then made via the supervising computer to classify the state cycle entered.

The word length used for data transfers between peripheral devices and data channels to and from the supervising computer and the propagation delay of data through the network determines the optimum parallelism with respect to through-put efficiency. If the ratio of time taken per bit for propagation through the network is greater than that for transfer and analysis by the supervising computer then through-put efficiency can be improved by choosing data path width in the network greater than the word length of the supervising computer. This, however, is unlikely to be the case since the network consists of relatively few levels of logic gates and so the solution requiring the least amount of hardware and control is to choose the same data path size for the network as the maximum input/output channel width of the supervising computer.

The other limiting factor on the parallelism of the network system is that of obtaining flexibility at a given hardware cost. The main problem here being the cost of interconnecting the network elements.

Interconnecting the inputs of the element with the output state store and the external inputs can be considered as defining a mapping between binary vectors representing the inputs and outputs. The mapping is single valued since no outputs can be connected together. The amount of information required to specify a connection mapping from a binary vector of n bits to one of m bits is,

$$m\log_2 n \quad \text{bits.}$$

There are several algorithms which can be used to implement connection mappings effectively, either for parallel systems or serial ones. Further discussion of these will be found in Chapter 6 on 'Improvements to the Minerva Learning Network System'.

The least flexible solution is to use hardwired patch-panels for network interconnections. This is a relatively inexpensive solution but presents considerable constructional problems in wiring patch-panels for the 6,656 connections required for a system the size of Minerva. Obviously the most serious drawback to hardwired interconnections is that they would not be under the control of the supervising computer and experiments that require modification of the network interconnections say, heuristically, would not be possible.

A possible scheme for fully parallel universal interconnection hardware using standard M.S.I. integrated circuits is shown in Figure 3.5 An implementation of this for a network with

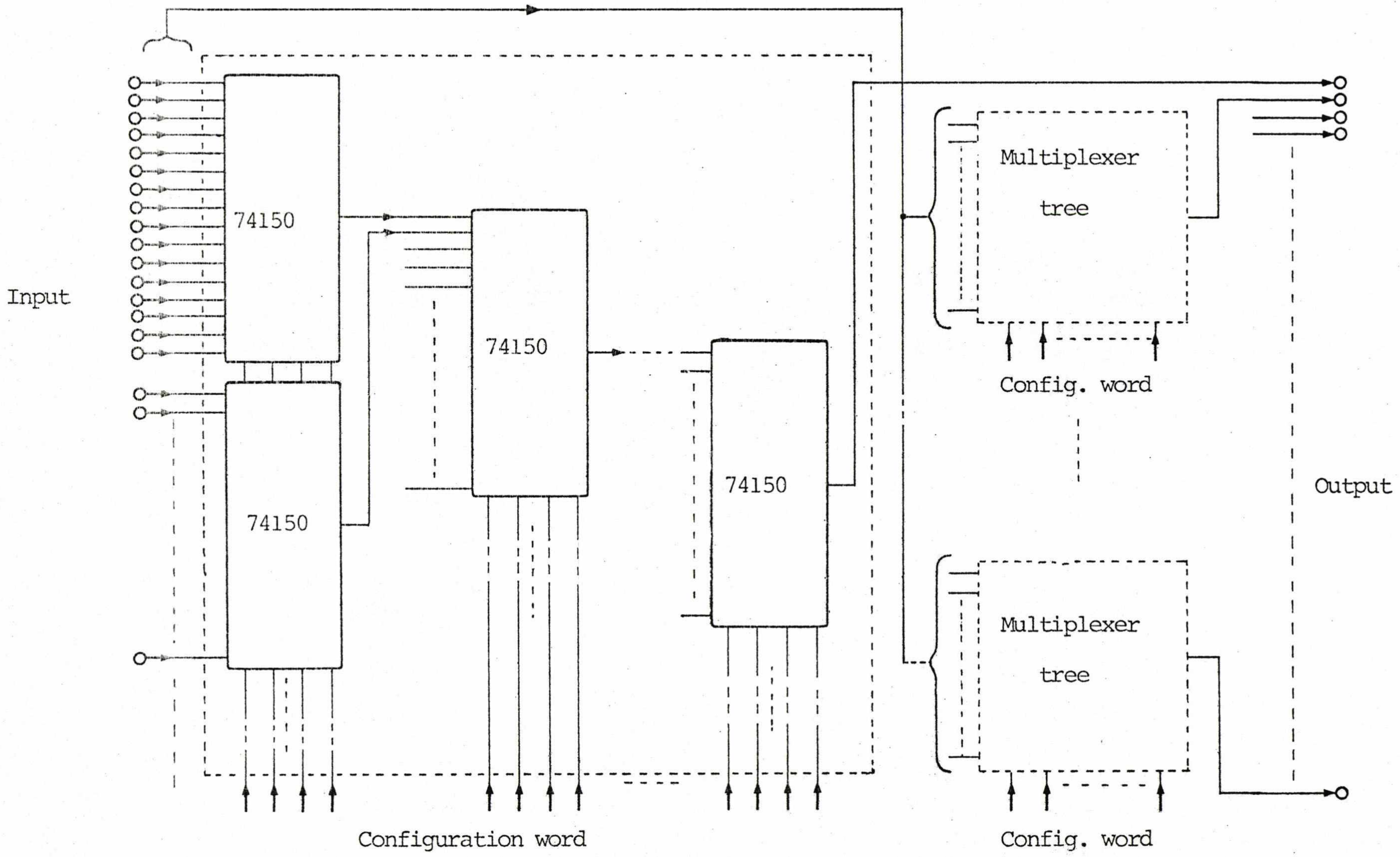


Figure 3.5 A Fully Parallel Random Interconnection Scheme

4,096 input spaces to 4,096 element inputs with 1,280 feedback connections would require parallel interconnection specifying vector of 53,248 bits and about 14.5 x 10^6 data selection integrated circuits. The scheme can be modified into a partially parallel, partially sequential one but a practical difficulty arises over the implementation of buffer storage for the output state and the input space. Twice as many storage bits are required for the output state store when serialism is introduced since as the current response is obtained, the whole of the previous response (or state) must be available for connection to succeeding inputs. Independent random access of each bit in the output state and input space stores is required for each parallel computed interconnection. In order to be able to exploit standard R.A.M. integrated circuits this design would have to be restricted to a single interconnection at a time. In the sequential case the interconnection specifying vector becomes a list of interconnection specifying elements available in some sequentially accessible store.

The cost of suitable storage hardware although available, was prohibitive at the time of the design specification of the system and the implementation of interconnection hardware was delayed until a later date. This hardware is described in Chapter 6.

3.2.1 The Realisation of Network Interconnections by Software

A similar algorithm to that given for the universal interconnection hardware of Section 3.2.0 as a fully sequential implementation may be used to design software to implement interconnections. Interconnections can be viewed as a mapping between two binary vectors representing an input and output space. A mapping between an input vector of n elements and an output vector of m elements requires a map specification vector of m integer elements. The value of each map element i is in the range $0 < i = n$. Each element of the map vector is used as an index to an element of the input vector. The value of the input vector element indexed by a particular map element is transfered to the element of the output vector corresponding to the index of the particular map vector element.

The algorithm given above can be easily implemented in software by the supervising computer if vectors representing the input and response spaces of the network are resident in the computer memory together with the mapping vector. This requires 4,448 memory locations of 16 bit words for the MINERVA network, a two layer network of 1,280 S.L.A.M.-16.

The main advantage obtained by the software scheme is the ability to monitor and directly control the network structure, enabling dynamic optimisation in cases such as feature extraction. However this increase in flexibility is obtained at a considerable loss in though-put as demonstrated in the following Section.

3.2.2 Data Through-put of Various Network Implementations

An autonomous network of S.L.A.M.-16 elements similar to MINERVA with hardwired, patch-panel interconnections and parallel architecture has a net response time, independent of the number of elements, of about 2 μ sec., due to the combined propagation delays of the S.L.A.M.-16 elements and input and output driver circuits.

When the autonomous network is connected to a Honeywell DDP 516 computer with network input and output data resident in the core memory and parallel 16 bit input and 5 bit output interfaces, the increase in network cycle time is 256 x 10 + 256 x 10 μ S., that is 5,120μS. (Values of input and output data rates of the Honeywell DDP 516 are justified in Appendix D).

The software interconnection routine described in Section 5.4.2. performs each interconnection in approximately 40 μ S. The increase in network cycle time due to software interconnections is then 163,840 μ S., or some 32 times. This figure loses importance when considerable processing of the input or the response is carried out or the data transfer rate between the network system and an external environment is comparitively slow. However it represents an upper-bound on the through-put performance of any network architecture with software interconnections.

Considering a serial architecture for the network, chosing a net input bus width of 16 bits; after each data word is output from the supervising computer the inputting of the response must be delayed by the net response time. Since this time is short, ~2 μ S., when compared with interconnection and interface time, there is negligible performance difference between parallel or serial computer-controlled-network structures. Furthermore, the network operates concurrently with the computer, therefore the response time may be able to be ignored if housekeeping instructions can be carried out during the response time.

A serial structure has advantages of requiring no buffer storage or multiplexing circuitry as part of the hardware interface with the computer. The cost of building a serial structure is significantly smaller than that of any parallel system for this reason.

Some data through-put improvements can be made for a restricted class of networks having a second layer of elements regularly connected to the outputs of a single-layer network. The sequential network structure may still be exploited without the need for local storage registers. A two-layer network of 1,280 S.L.A.M.-16s made up of single-layer networks of 1,024 and 256 S.L.A.M.-16s using software interconnections requires an extra 1,280 μ S., (assuming a 4 bit computer input interface), as compared to one similar to MINERVA having a hardwired second layer and a 5 bit computer input interface.

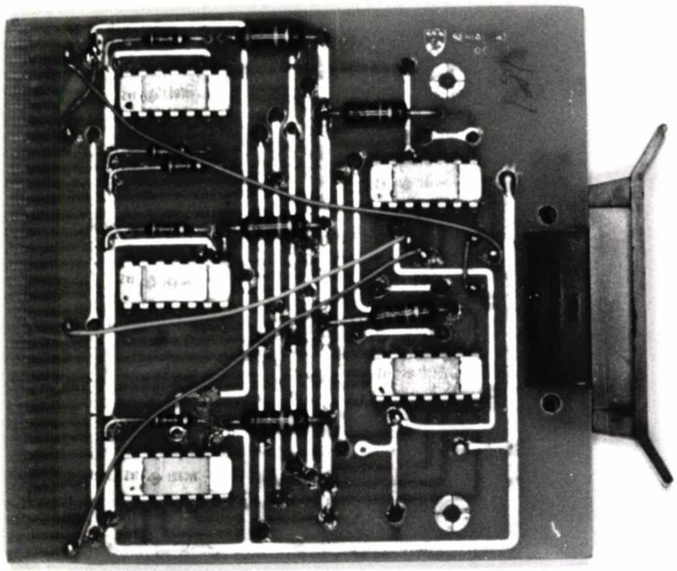
It must be noted that values of network through-put derived in this Section represent the theoretical maximum for the MINERVA learning network without any analysis of the network response.

3.3 The Network Element Hardware

This section describes the design and construction of a 1,280 S.L.A.M.-16 element learning network, referred to as MINERVA, and its interfaces with a Honeywell DDP-516 mini-computer. A description of the DDP-516 is to be found in Appendix A.

3.3.0 The Use of S.L.A.M.-16 as a Network Element

The DDP-516 computer has a 16 bit word length and for reasons given above the maximum data path width of the Network Element Hardware (N.E.H.) was chosen to be 16 bits wide. This could accommodate the inputs to four S.L.A.M.-16's arranged as a parallel network segment. The second layer of one S.L.A.M.-16 derives its inputs directly from the outputs of the four first layer S.L.A.M.-16's. All five S.L.A.M.-16 outputs are made available. This forms a singly addressable segment of the N.E.H. and is implemented on a single printed circuit card having 34 edge connections. This is called the Network Element Card (N.E.C.), and is illustrated in Figure 3.6 . The N.E.H. contained 256 Network Element Cards. The circuit diagram of the N.E.C. is shown in Figure 3.7.



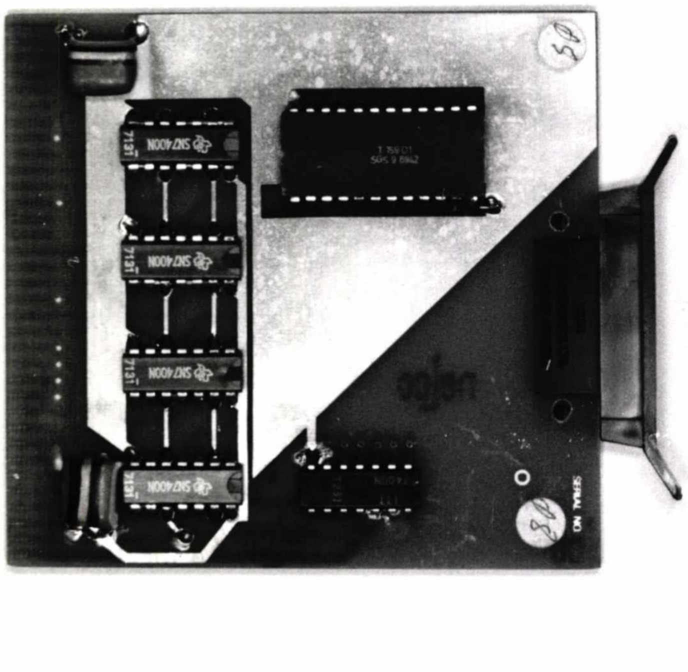
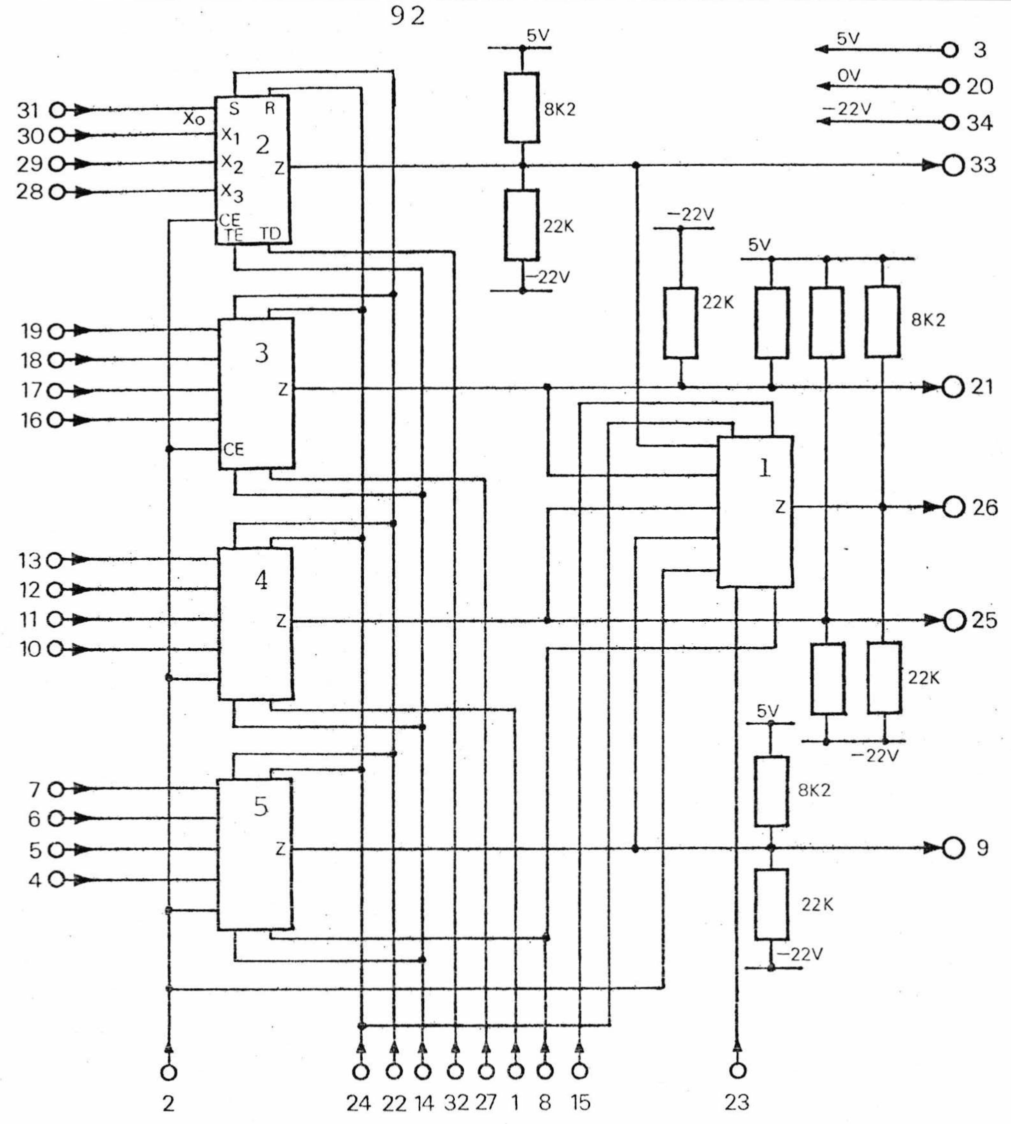


Figure 3.6
A Network Element Card (top)
and Address Decoder Card



1	(TD4)	18	NDB 6
2	CE	19	NDB 7
3	+5v.	20	Ov.
4	NDB 12	21	SLAMOP3
5	NDB 13	22	RAC1
6	NDB 14	23	TE2
7	NDB 15	24	SAC1
8	(TD5. TD1)	25	SLAMOP4
9	SLAMOP5	26	SLAMOP1
10	NDB 8	27	SAC2, (TD3)
11	NDB 9	28	NDB 0
12	NDB 10	29	NDB 1
13	NDB 11	30	NDB 2
14	TE1	31	NDB 3
15	RAC 2	32	(TD2)
16	NDB 4	33	SLAMOP2
17	NDB 5	34	-22v.

Figure 3.7 Network Element Card

The card was initially configured with separate Set, Reset and Teach Enable inputs to each layer but with the Teach Data and Chip Enable inputs common to all the S.L.A.M.-16 devices. Provision is made for this to be altered by jumper connections to provide separate Teach Data inputs for each device at the expense of having common Set inputs for both layers. The independent Teach Data configuration was implemented after the initial commissioning of the network had been completed and the final network configuration is described in Chapter 6.

The interfacing scheme of the N.E.C.'s with the DDP-516 computer is shown in Figure 3.9. The output lines from the computer are connected by a common data highway to the inputs of storage registers containing the card data information and the card address information. The five N.E.C. response signals are taken directly to the computer input interface. The delay involved in the execution of an input instruction preceded by an output instruction by the DDP-516 computer is longer than the propagation delay through the total N.E.C. and interfacing hardware. This means that the normal hand-shaking protocol signals between processors and peripherals are unnecessary, enabling the learning network to be operated synchronously with the instruction sequence programmed on the DDP-516 computer. The DDP-516 computer is also capable of generating a control pulse on a specified output line under direct program control. These instructions, referred to by OCP in Appendix A, are used for all the internal micro-orders required by the learning network hardware.

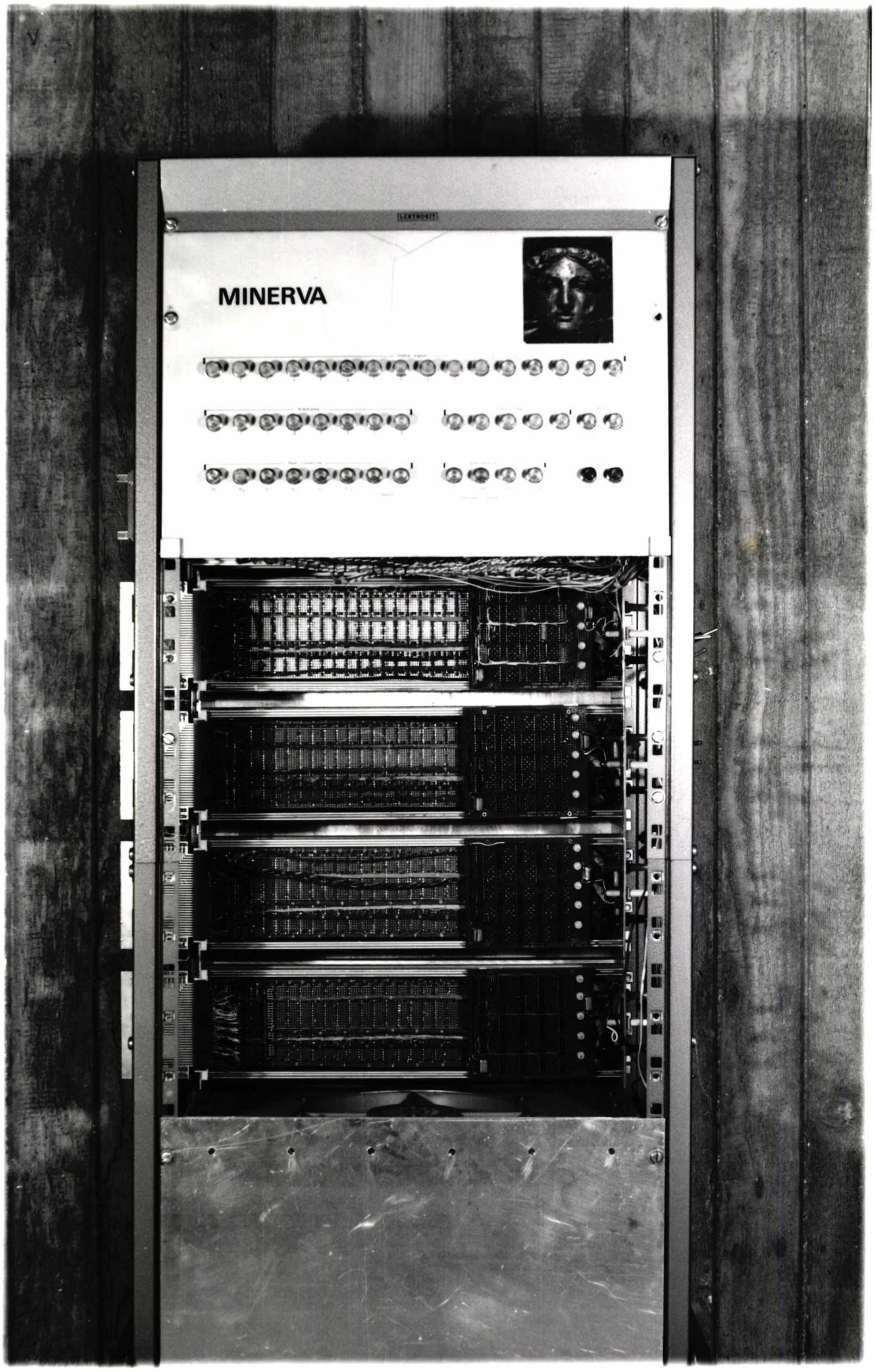


Figure 3.8

Network Element Rack
Interconnections

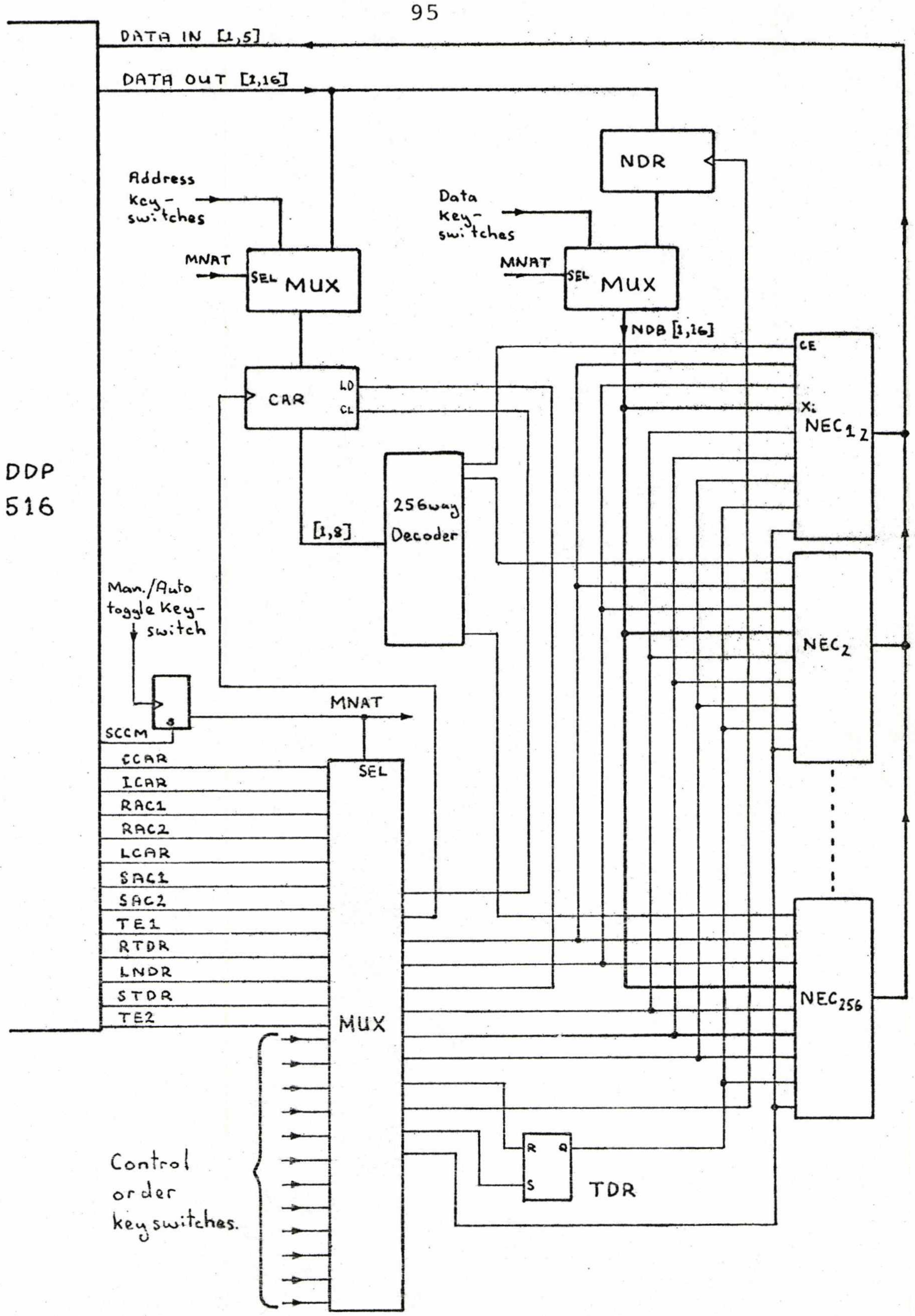


Figure 3.9 Network Hardware to DDP 516 Interfaces

The network hardware consists of 256 N.E.C.'s separated between eight racks each containing 32 N.E.C.'s All control and data inputs except enable inputs, are taken to common bus connections provided by a proprietory printed strip circuit board. It was originally intended that the S.L.A.M.-16 outputs should also be connected via common bus lines to high impedance input line receiver circuits. This was found impracticable for two reasons: the S.L.A.M.'s did not reach their original output drive specification due to the design error mentioned in Chapter 2; cross-talk induced by neighbouring low impedance bus lines driving the inputs to the N.E.C.'s triggered the line receiver circuits. This is overcome by taking the individual S.L.A.M.-16 outputs to the inputs of low-power TTL gates. NAND gates with a fan-in of eight are used. The NAND outputs are taken to standard power TTL open collector drivercircuit before being connected to a common bus carrying the output information to the interface. The output buffering logic requires a circuit board mounted parallel to the N.E.C. back circuitry as shown in Figure 3.8

The N.E.C. enable inputs are provided from two card address decoder circuit boards, each controlling 16 N.E.C.'s of each rack. The circuit of the card address decoder board is shown in Figure 3.10. All the common bus lines of the N.E.C.'s are connected to the control and interface circuitry by a daisy chain cable to a connector on each N.E.C. rack. The impedence of the daisy chain cable is controlled to be 100Ω by using twisted pair cable for each signal line and signal lines are terminated by a 100Ω load

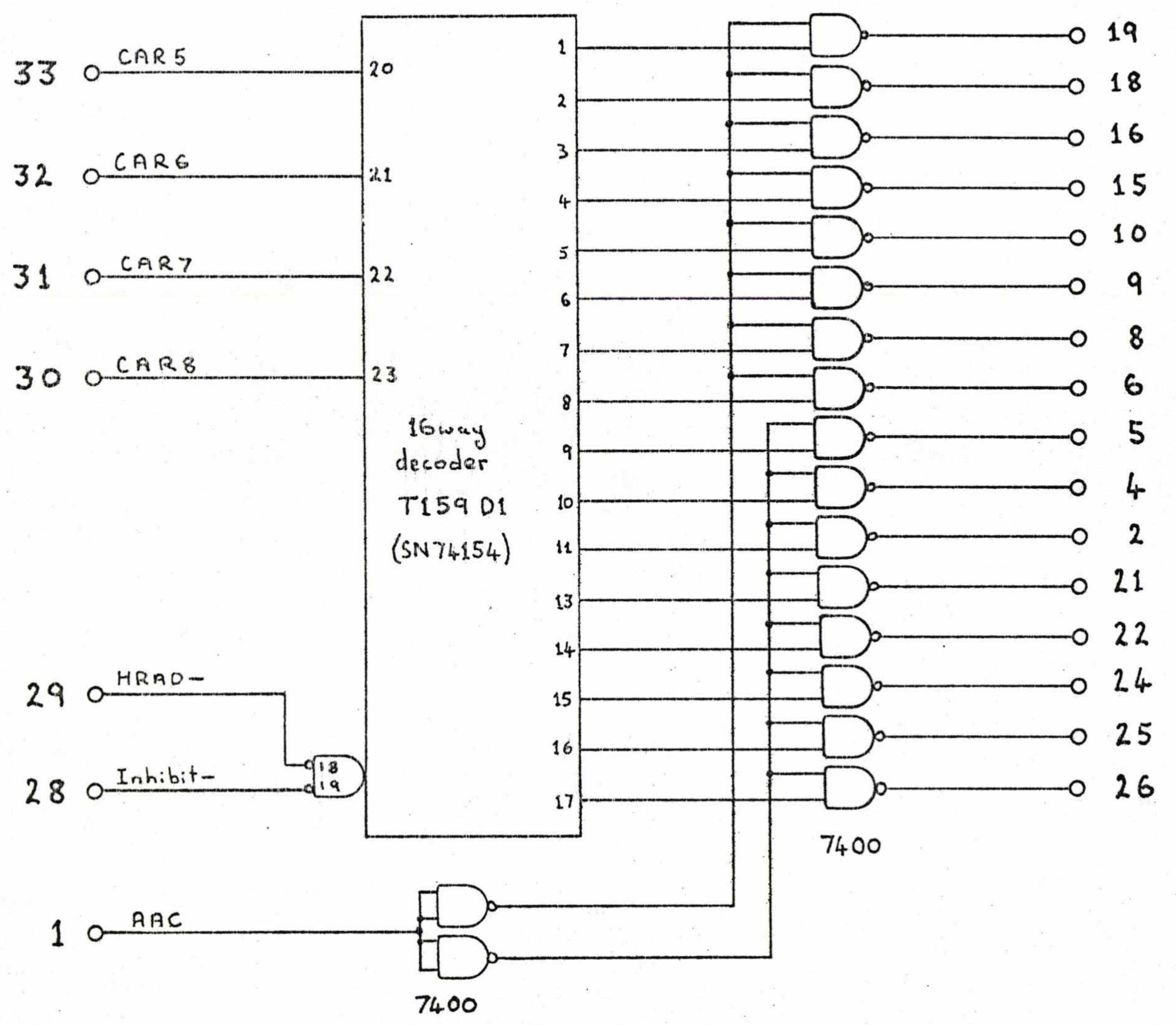


Figure 3.10 N.E.C. Address Decoder Card

resistor on the N.E.C. rack furthest from the control circuitry. This is to avoid problems caused by signal line cross-coupling and multiple pulses caused by signal reflections.

3.3.1 Control Circuitry for the N.E.C.'s and the Computer Interface

The learning network hardware has two modes of operation. In the automatic mode all data and micro-orders are provided by the DDP-516 computer. In the manual mode the data and micro-orders are provided by manually operated keyswitches. The manual mode is provided for the off-line testing of the network. The state of all the data lines and contents of the registers are displayed by indicator lights on the operators panel. Either mode can be selected manually, though the automatic mode can also be selected by a computer generated micro-order.

The control circuitry is the most modified part of the learning machine because of the conversion of the supervising computer to a Digital Equipment Corporation PDP 11/40 computer and changes to simplify the software. The operation of the original network used by the software presented in Chapters 4 and 5 is presented here. Readers interested in the current network hardware are referred to Chapter 6.

3.3.2 The Operation of the Network Hardware

Micro-orders are received from the DDP-516 as 500 nS. pulses and are lengthened to 1.5 μS. pulses for operating the S.L.A.M.-16 control inputs. A list of the mnemonics representing the micro-orders and their effects is presented at Appendix E for further reference.

The learning network operates under total programme control by the DDP-516 computer. There are two modes of operation: the teach mode and the response mode.

The Response Mode:

1) Output address of 1st card to interface;
2) Generate LCAR;
3) Output data word after performing software interconnections;
4) Generate LNDR;
5) Input 5 bit response data to computer;
6) Generate ICAR;
7) Repeat (3) et seq. until the number of N.E.C.'s allocated complete.

A DAP-16 assembly language program segment to perform the response mode is as follows:

```
        (3) NRSP   LDX  NNEC     ;-no. of cards in index
1)      (2)        LDA  SNEC     ;address of 1st card
        (3)        OTA '0242     ;output to interface
```

```
                   HLT               ;never here, hardware error
2)    (3)          OCP   '1142       ;LCAR
3)    (2) LOOP     LDA    DATA,1     ;get data indexed
      (3)          OTA   '0242       ;output to interface
                   HLT               ;never here, hardware error
4)    (3)          OCP    '1642      ;LNDR
5)    (3)          INA   '0042       ;input response
                   HLT               ;never here, hardware error
      (2)          STA    RESP,1     ;save response, indexed
6)    (3)          OCP   '1342       ;ICAR
7)    (3)          IRS   Ø           ;skip if last card
      (1)          JMP    LOOP       ;do next card
```

The left-most column above cross references with the response algorithm and the next column gives the execution time in processor clock cycles of the machine instruction. A processor clock cycle is 0.96 μS.

The time taken to execute the response sequence is

$$(10 + 20n)\ 0.96 \quad \mu S.,$$

where n = number of N.E.C.'s.

A similar algorithm for the teach mode is as follows:

1) Output address of 1st card;
2) Generate LCAR;

3) Output data word after performing software connections;

4) Generate STDR or RTDR according to the teach data;

5) Generate TE1 or TE2;

6) Generate ICAR;

7) Return to (3) <u>et seq.</u> until number of N.E.C.'s allocated complete.

A DAP-16 assembly language program to perform the teach mode is as follows:

```
1)    (3)         LDX    NNEC     ; no. of N.E.C.s to index
      (2)         LDA    SNEC     ; address of 1st N.E.C.
      (3)         OTA    '0242    ; output N.E.C. start
                  HLT             ; never here, hardware error
2)    (3)         OCP    '1142    ; LCAR
      (2) LOOP    LDA    =-16     ; bits in teach data word
      (2)         STA    BCNT     ; counter for teach data bit
3)    (2) L1      LDA    DATA,1   ; get training i/p data
      (3)         OTA    '0242    ; output data
                  HLT             ; never here, hardware error
      (3)         OCP    '1642    ; LNDR
4)    (3)         OCP    '1042    ; RTDR
      (3)         LDA*   TDAT     ; get word of teach data
      (2)         LGL             ; rotate,setting carry bit
      (1)         SCR             ; skip on carry reset
      (3)         OCP    '0742    ; STDR
      (3)         STA*   TDAT     ; put back teach data
```

```
5)      (3)         OCP    '0442      ; TE1

        (3)         OCP    '1342      ; Next N.E.C. ICAR

6)      (3)         IRS    Ø          ; skip if last card

7)      (1)         SKP               ; more cards

        (1)         JMP    OUT        ; jump out

        (3)         IRS    BCNT       ; next bit of teach data

        (1)         JMP    L1         ; loop for next teach

        (3)         IRS    TDAT       ; increment to next word
                                        of teach

        (1)         JMP    LOOP

              OUT
```

The time taken to execute the teach mode sequence is:

(11 + 34n) 0.96 μS.,

where n = no. of N.E.C.'s.

Training of second layer S.L.A.M.-16's uses the same program given above with the TE2 micro-order substituted for TE1. The training of both layers simultaneously, using constant input data, causes the second layer to perform the trivial AND function.

3.3.3 Hardware Implementation of Network Control Circuitry

The control circuitry is implemented using T.T.L. logic gates with discrete circuitry for driving indicator lamps and the S.L.A.M.-16 data and control buses.

The voltage levels required to operate the S.L.A.M.-16 inputs are directly compatible with those for T.T.L. although the conduction of the inputs is reversed; the M.O.S. inputs requiring their maximum current of typically 100μA. in the high voltage logical '1' input state. The P-N-P transistor circuit of Figure 3.1 1 possesses the correct characteristics to be driven by T.T.L. and drive 1,024, S.L.A.M.-16 input loads with rise and fall times of 100nS. and 200nS. respectively. The lamp driver circuit, shown at Figure 3.10, is of similar design to the M.O.S. interface due to the physical construction of the lamp holders. These made one of their connections through the chassis mounting screw which is used as a Ov. return.

The control circuitry is implemented on 11 circuit cards having connections to the DDP-516 interface and the N.E.C. racks by four 37 way connectors. The actions performed by the individual cards are as follows:

Card (1) is shown in Figure 3.1 1 and contains 16 discrete lamp driver circuits displaying the state of the Net Data Bus, (N.D.B.);

Card (2) is similar to Card (1) and contains 16 discrete driver circuits to transmit N.D.B. to the N.E.C. racks. The circuit of the driver circuits is given at Figure 3.1 1;

Card (3) and (4) are identical and contain the Key Data

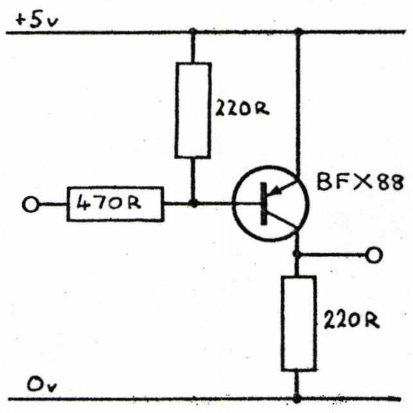
Register, (K.D.R.), multiplexer circuits for selecting the source of N.D.B. according to the manual/automatic mode signal. The circuit of cards (3) and (4) is shown at Figure 3.12;

Card (5) contains the Card Address Register, (C.A.R.), and T.T.L. driver circuits to transmit the 4 least significant bits of C.A.R. to the card address decoder boards. The inputs to C.A.R. are provided from the DDP-516 interface or the Key Address Register, (K.A.R.), by two-way multiplexer circuits according to the manual/automatic mode signal. C.A.R. is an 8 bit binary counter with parallel load and clear inputs. The circuit of Card (5) is shown at Figure 3.13;

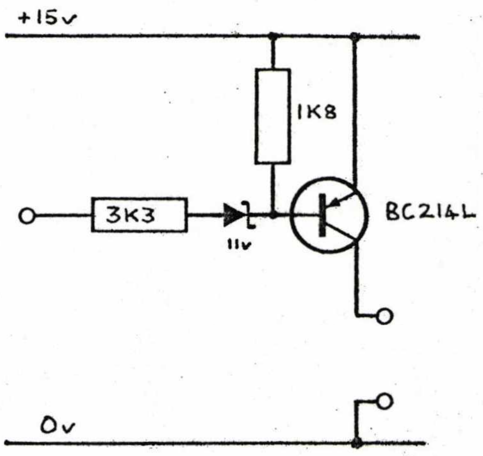
Card (6) contains lamp driver circuits to display the contents of C.A.R. The circuit of Figure 3.11 is used;

Card (7) is shown in Figure 3.14. It contains a 16 way decoder to decode the 4 most signigicant bits of C.A.R. into half-rack select signals and driver circuits to transmit the micro-orders to the control inputs of the N.E.C.'s;

Card (8) is shown in Figure 3.15. It contains multiplex circuitry for the origin of the micro-orders according to the manual/automatic mode signal and monostable circuits to convert the micro-orders to 1.5μS pulses. It also contains the 1 bit Teach Data Register.



M.O.S. Driver
Card 2.



Indicator Lamp Driver
Card 1.

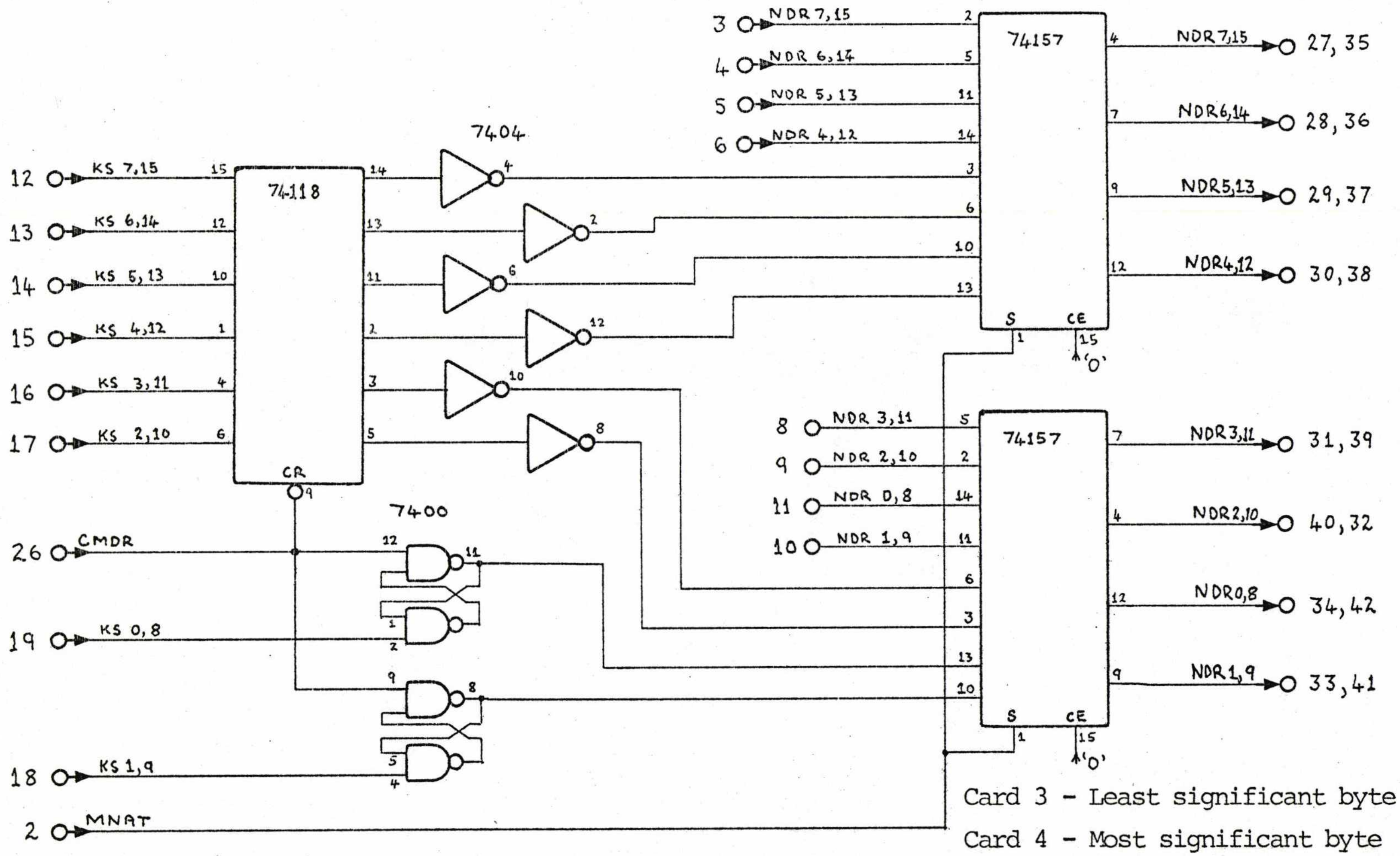
Edge Connections

1	+5v.	23	
2		24	
3		25	
4		26	
5	NDB 0	27	NDR 15
6	NDB 1	28	NDR 14
7	Key	29	NDR 13
8	NDB 2	30	NDR 12
9	NDB 3	31	NDR 11
10	NDB 4	32	NDR 10
11	NDB 5	33	NDR 9
12	NDB 6	34	NDR 8
13	NDB 7	35	NDR 7
14	NDB 8	36	NDR 6
15	NDB 9	37	NDR 5
16	NDB 10	38	NDR 4
17	NDB 11	39	NDR 3
18	NDB 12	40	NDR 2
19	NDB 13	41	NDR 1
20	NDB 14	42	NDR 0
21	NDB 15	43	Ov
22			

Edge Connections

1	+5v.	23	LP 0
2	+15v.	24	
3	LP 15	25	
4		26	
5	LP 14	27	NDR 15
6	LP 13	28	NDR 14
7	Key	29	NDR 13
8	LP 12	30	NDR 12
9	LP 11	31	NDR 11
10	LP 10	32	NDR 10
11	LP 9	33	NDR 9
12	LP 8	34	NDR 8
13		35	NDR 7
14	LP 7	36	NDR 6
15		37	NDR 5
16	LP 6	38	NDR 4
17	LP 5	39	NDR 3
18	LP 4	40	NDR 2
19		41	NDR 1
20	LP 3	42	NDR 0
21	LP 2	43	Ov
22	LP 1		

Figure 3.11 Control Logic Cards 1 & 2



Card 3 - Least significant byte

Card 4 - Most significant byte

Figure 3.12 Control Logic Cards 3 & 4

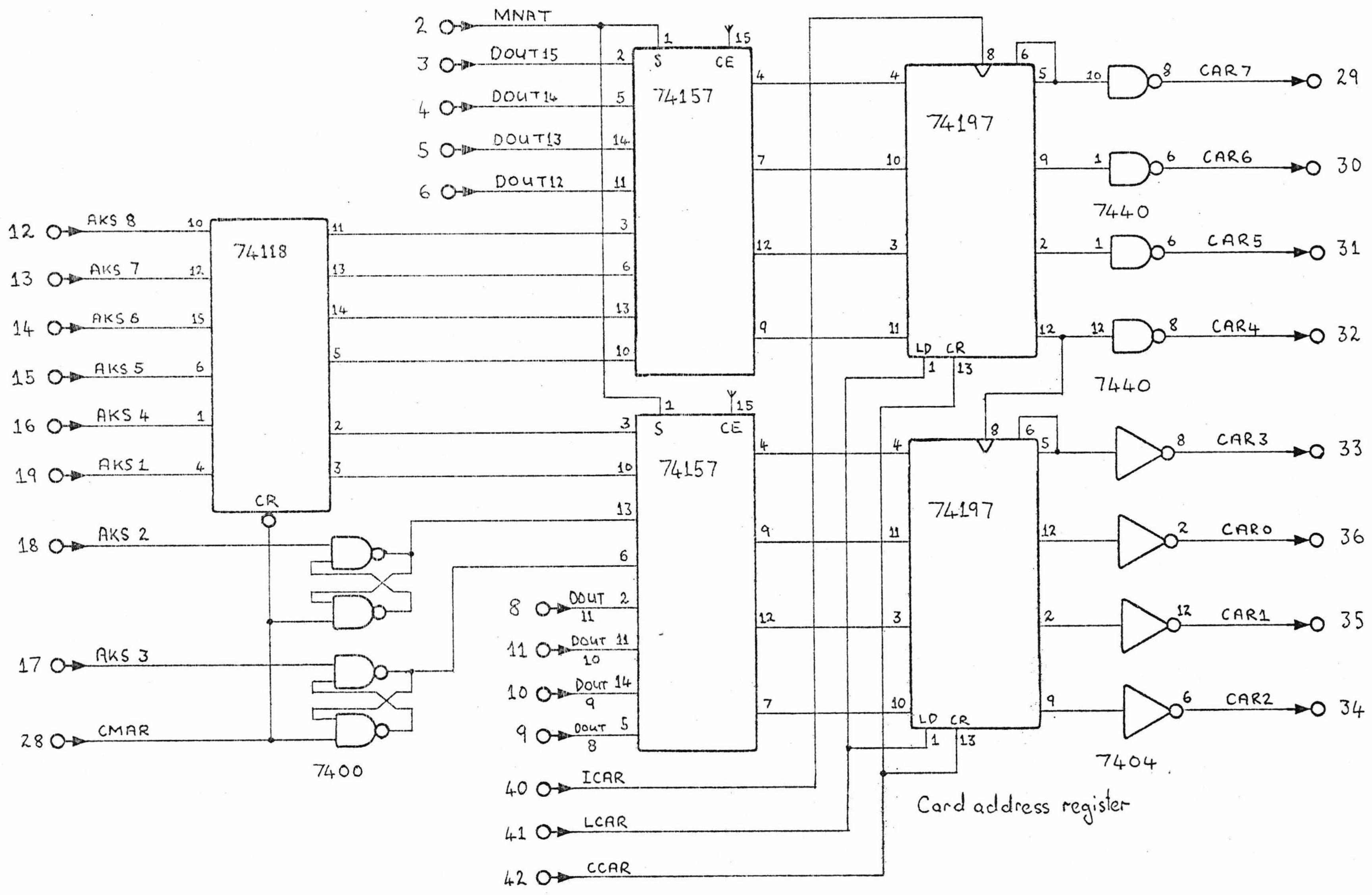
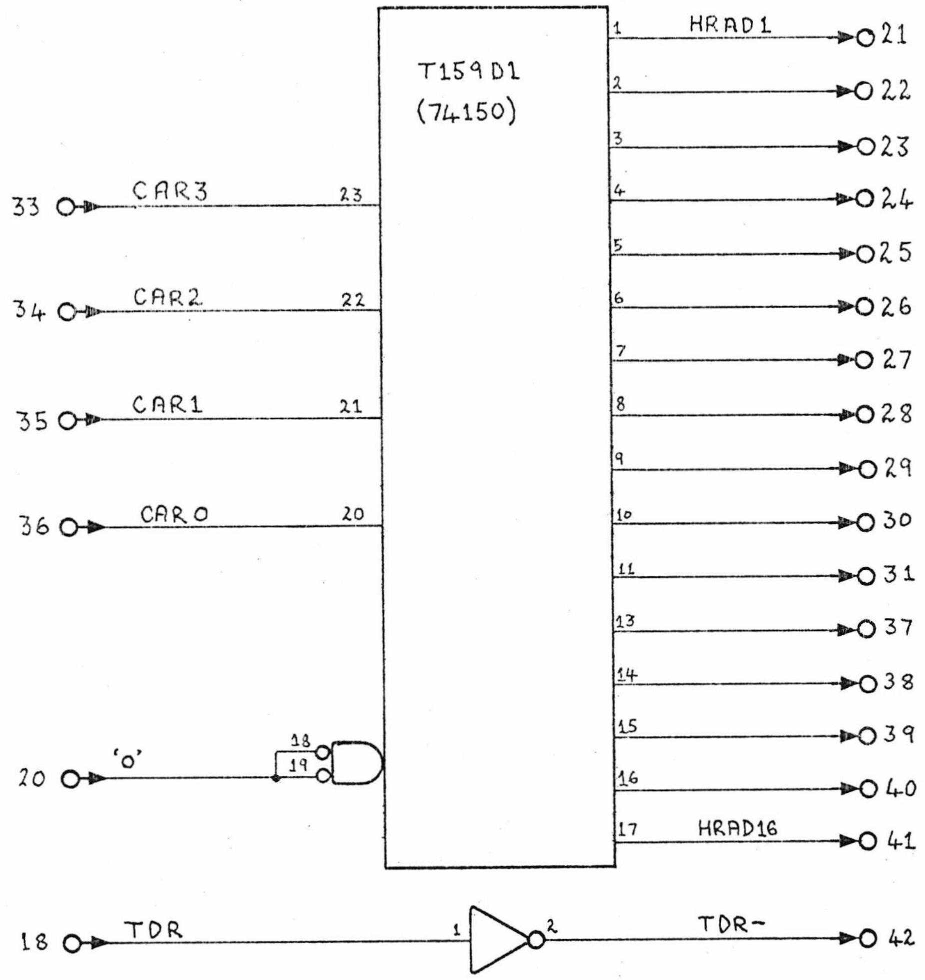


Figure 3.13 Control Logic Card 5



M.O.S. Drivers

IN	SIGNAL	OUT
12	SAC 1	4
13	RAC 1	5
14	SAC 2	6
15	TE 2	9
16	RAC 2	10
17	TE 1	11
18	TDR	3

Figure 3.14 Control Logic Card 7

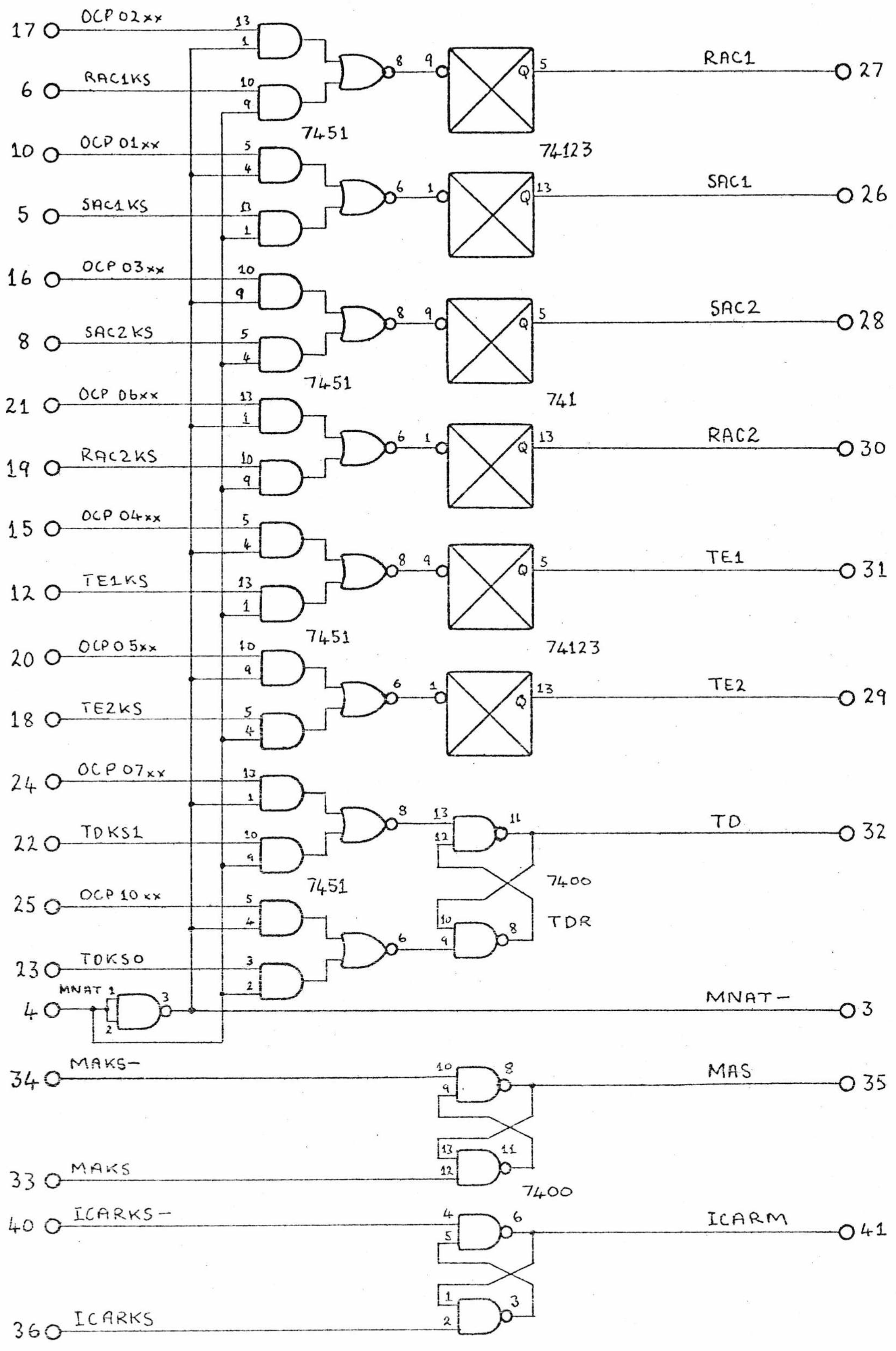


Figure 3.15 Control Logic Card 8

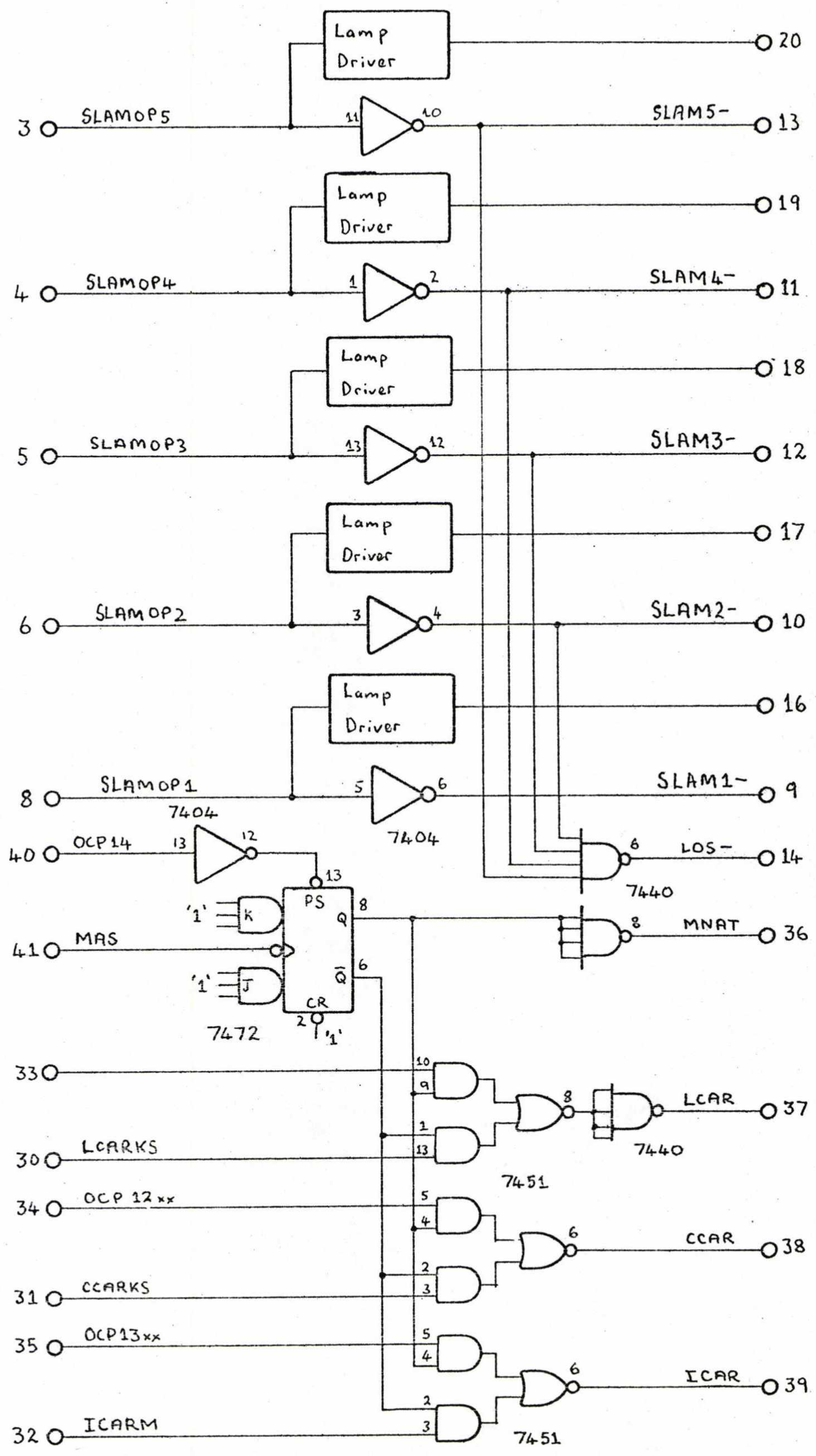


Figure 3.16 Control Logic Board 9

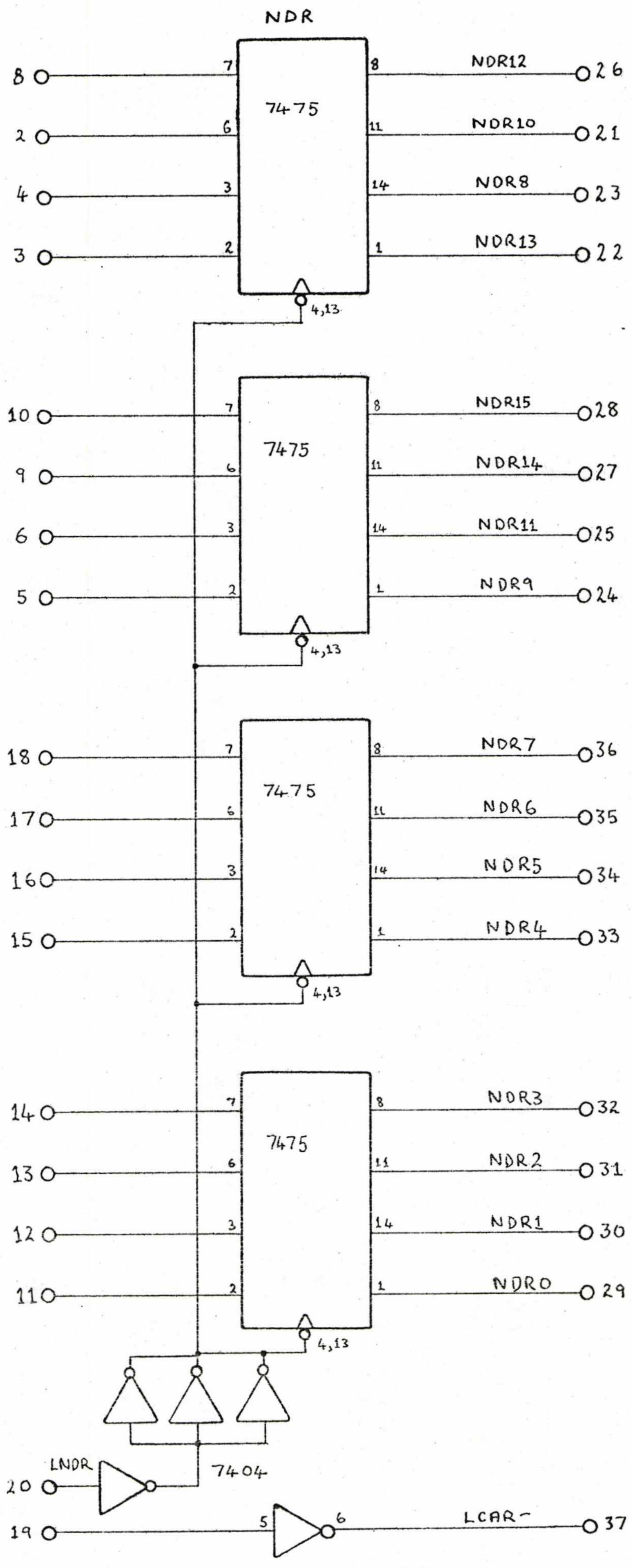


Figure 2.17 Control Logic Card 10

Card (9) is shown in Figure 3.16. It contains the manual/automatic mode control flip-flop and lamp driver circuits to display the current N.E.C. response;

Card (10) is shown in Figure 3.17. It contains the 16 bit Network Data Register, (N.D.R.);

Card (11) contains Schmidt trigger buffer circuits to receive micro-orders from the DDP-516 interface. These are used to ensure low susceptibility to spurious noise.

3.4 Commissioning the Network Element Hardware.

Testing the N.E.H. consists of proving the various circuit boards achieve their electrical and logic specifications by individual bench testing methods and also providing diagnostic procedures for identifying faults in the completed network system. Static testing of the completed system is accomplished using the manual mode of operation.

3.4.0 Computerised Testing of the N.E.H.

Initially, a test interface to the DDP-516 computer for an N.E.C. was constructed. This provided valuable experience in using computer interfaces and validated the operation of the N.E.C.'s before their insertion into the N.E.C. racks. The tests are designed to exercise the functions of the S.L.A.M.-16 elements of the card in a manner similar to that of the hardware tests as described in Chapter 2.

As construction of the back wiring and control logic progressed, simple I/O looping programs were used to validate the noise immunity of signal lines. Design changes were made at this point concerning the routing of the Ov signal return from the N.E.C. racks to the control logic.

This type of testing requires "hands-on" use of the computer facility for long unpredictable periods during which its data processing capacity is little used. This probably means a project of this nature can only be attempted satisfactorily where a dedicated computer system is available.

On completion of the hardware a software exerciser was written which tested each N.E.C. function as well as the control and addressing functions. Provision was made for the program to halt on finding an error or to go into a tight loop to enable hardware diagnostics to be carried out.

The N.E.H. was considered commissioned during October 1972 when the exerciser program would run continuously for over 12 hours without an error.

3.4.1 Type of Error

The most common error is the failure of a S.L.A.M.-16 element. These were numerous during the initial commissioning period and were typified by the S.L.A.M.-16 output in question going permanently into the 1 or 0 state. Due to the bussed nature of the outputs the position of the offending element can not be ascertained by software diagnostics if the offending element has failed by going permanently low.

The element has to be physically isolated by disconnecting portions of the bus until the fault is cleared.

CHAPTER 4

Learning Network Software

4.0 Introduction

The "systems architecture" approach to the design of the network hardware, which was presented in the last chapter, is further developed in this chapter and provides a link between the abstract hardware data structure and a command language which enables the user to express algorithms for the control of the network. The parameters affecting the design of a language are investigated and a special purpose language, helpful in the conceptualisation and realisation of different networks is presented. This language in turn has influenced the latter development of the hardware presented in Chapter 6.

The structure and scope of the language were influenced greatly by the limitations of the techniques available for its development and execution. These are presented fully in Chapter 5. A formal definition of the language is presented using a modified B.N.F. notation. However this does not imply that the definition was used in the implementation of a formal parsing algorithm.

4.1 An Overview of Previous Network Software

Until the construction of the MINERVA learning system, large scale studies of learning network behaviour have been carried out by simulation using either a general purpose high level language or assembly language. Broadly those experiments in which the aim was to investigate general principles such as Atlas (1973) were implemented in a high level language and those requiring interactive intervention , or using networks in a real time system, were implemented in assembly language. An example of the latter is the real time T.V. camera executive of Reeves (1973). There are obvious advantages in program development using high level languages but the necessary computing resources were not available either to the author or to the other experimenters whose software systems are described below.

4.1.0 Assembly Language Simulation Systems

In order to avoid the need to recode each experiment individually in assembly language some limitations can be placed on the scope of the network organisations that can be realised. This enables a skeleton executive simulation to be written which will allow the linking of a library of user-defined subroutines. Such systems are described by Tollyfield and Dawson and are based on the executive system designed by Reeves.

The Reeves executive was a total system in that a large part of it carried out normal operating system tasks, such as the time-shared handling of all peripheral input and output, and it required no other systems programs. The main features of its design were for the control of a digitised television camera peripheral and in providing the common data structures for patterns and mapping vectors used by the user-defined network simulation programs.

The executive passed control to each of its user-defined programs by interpreting a command string of mnemonics each of which caused the execution of a routine. Any control arguments required by the routine followed the mnemonic. Storage for pattern variables and network element stores was allocated by the executive as a fixed vector and could be accessed from user routines by passing an index.

This system of command strings containing mnemonics could in itself be considered a programming language and user routines were written by the author to enable the MINERVA hardware to be used with the Reeves executive. Further development was however, abandoned since the Reeves programming system lacked the richness of syntax necessary to enable manipulation of semantic constructions. In order to have overcome the inherent limitations of the Reeves executive and provided a flexible system for users of varied interests, considerable foresight would have been required in compiling a comprehensive user routine library.

Any changes in a routine could only have been made by a skilled assembly language programmer and documentation of the system would have been a time-consuming secondary activity. As in the case of the present systems developed from the Reeves executive (Dawson and Tollyfield) there would be a tendency for routines to be developed with incompatible characteristics and for programming inefficiency caused by unnecessary duplication. Since program development could not take place interactively, in real time, there would be less opportunity for experimentation with novel network structures.

An attempt to write a general purpose S.L.A.M. simulation was made by Alan (1972) for a medium size ICL 4130 computer. The simulation was written in assembly language as a sub-system of a terminal-based multi-user operating system. The program was initiated from a remote terminal and an initialisation dialogue entered. S.L.A.M.'s of any number of inputs could be realised, albeit limited by the system workspace allocation. All inputs and outputs to the simulated S.L.A.M. elements were uniquely labelled. Interconnections between elements were supplied at initialisation time. Storage of the input and output state of the network were implicit in the nature of the simulation. The system comprised two program modules: SLAMDATA used to prepare and format data and interconnections for the simulation; SLAMSIM, the simulation program itself. SLAMSIM proved useful for demonstrating the behaviour of small networks but was limited by a lack of data analysis features. Its real-time capabilities were limited unpredictably by the computing demands of other users of the on-line system.

4.2 The Development of a Pattern Handling Language

It was evident, once hardware implementation of the network elements became available, that a language enabling manipulation of the fundamental network data structures was needed if individualistic approaches to network experimentation,such as those described in the last section, were to be avoided. The language described in the following sections is conceived as existing in an interactive environment. Just as in the Reeves executive where commands may be executed immediately from the terminal, so immediate execution of program statements and editing of programs may be accomplished without disturbing the network workspace. This feature is reflected in the acronym MINIC chosen to represent future references to the language and its compiler system. MINIC is an abbreviation of MINerva Interactive Code.

The MINIC language developed in the following sections shows some similarity of semantics with the more general and powerful language of Iverson (1962), A.P.L.. A discussion of these similarities is presented in Section (4.8).

It was recognised at the inception of MINIC that some arithmetic capability would be required both for program control and for analysis of data and results. In order to provide the user with the necessary program flow controlling features of procedural languages the pattern manipulating facilities of MINIC are designed to be an extension to a well known high level language. The syntax used for the MINIC extensions is chosen to be compatible with that of the host language. This provides an obvious advantage to the user in learning how to use the system.

4.3 Primitive Operators and Variables.

The pattern manipulating facilities of MINIC are derived from consideration of a representation of the generalised network presented in figure 3.4 in terms of primitive variables and operators.

The fundamental variable of the generalised network (Fig.3.4) is the state of an interconnection bus. This can be represented by a binary vector of the same order as there are ways in the bus. Syntactically this is represented by a variable identifier with the suffix PAT to distinguish a Pattern Variable. A direct connection between two buses is represented by an assignment expression, using the assignment operator, BE.

e.g. PAT X BE PAT Y

Both vectors have to be of the same order. To enable the representation of merging, masking and comparison operations, expressions may be formed consisting of pattern variables and the Boolean operations OR, AND, Exclusive OR, and NOT or compliment, Hence:

```
PAT X BE PAT Y OR PAT Z
```

computes the inclusive OR of each element of vector Y and Z and assigns them to vector X.

4.3.0 Mappings and Interconnections

Interconnections are called "indirect" where the vectors are of different orders or where some interconnection mapping is specified. A map variable is used to specify a re-ordering of a pattern variable and is represented by a vector of integer elements, the order of which defines the order of the resultant pattern variable, and the maximum possible magnitude of the elements is defined by the order of the pattern variable operand. The effect of a mapping is that the 'i'th element of the Map Variable is used as an index to select the 'n'th element of the operand pattern variable which then becomes the 'i'th binary element of the resultant pattern vector. For syntactic reasons a dummy mapping operator 'BY' must appear between the pattern and map variables. The following example illustrates the action and syntax of the mapping operation:

```
    PAT X BE PAT Y MAP M EX PAT Z

1   1          3      0          0
2   0          6      1          0
3   1          2      1          0
4   1     ⊕    6  BY  0    BE    1
5   1          1      1          1
6   0          3      0          1
7   0          5                 1
8   1          4                 1

    PAT Z      MAP M  PAT Y      PAT X
```

4.3.1 Net Variables and Hardware

Net elements are considered in a similar fashion to map variables since they perform an operation on some input pattern variable and reduce its order by a factor of n, where n is the number of inputs per element. In the implementation of MINIC Net variables were restricted to those available using the learning Network hardware defined as a contiguous column of N.E.C.'s In this case n = 4. Two parameters are initially assigned to a Net variable, the address of the 1st. N.E.C. and the number of N.E.C.'s making up the net identified by the particular net variable label. This is done in a variable declaration statement. The two modes of N.E.C. operation training and response are distinguished by the use of context. The NET assignment expression being used for training. As with the MAP variable a dummy operator IP is defined for syntactic reasons which are made apparent in Chapter 5.

The following examples show the use of context in net expressions:

a) Response mode:

```
PAT R BE NET N IP PAT X BY MAP M
```

b) Teach mode:

```
NET N IP PAT X BY MAP M BE PAT T
```

Another important operation is the arithmetic sum of the elements of a pattern vector. Some of its uses include; a) Hamming Distance measurements between pattern vectors and; b) weighing of network responses:

a) Hamming Distance:

```
SUM PAT P EX PAT R1
```

;

b) Weight of Response:

```
SUM NET N IP PAT X BY MAP M
```

4.4 Choosing a Syntactic Vehicle for the Language Primatives

The features described in the preceding sections are those that are unique to the MINIC language. The remainder of MINIC with the exception of those features explained in Section 4.6.2, is identical to the BASIC language as described by Kurtz and Kemmeny. BASIC was chosen as the result of a study whose prime concern was to investigate the programming effort and computing resources needed to modify one of the existing translator systems for high level languages available on the DDP 516 computer or to program a new translator system. There are several design criteria for a flexible learning network system that BASIC fulfils in preference to other high level languages available on the DPP 516 computer:

a) The language is well known among non-specialist programmers;
b) It is designed for interactive execution and program development;
c) The syntax has a field free format and no special character symbols are required;
d) Programs written for the existing DDP 516 BASIC interpreter can be used to verify and benchmark the development of an independent translator.

The translator systems available on the DDP 516 system were for the FORTRAN IV, PL516 and BASIC languages. Reasons for rejecting a translator system based on any of the existing translator programs are given below.

4.4.0 The DDP 516 FORTRAN System

FORTRAN IV was rejected for 3 reasons: The language itself was considered unsuitable since it was designed for a batch environment both for program development and execution ; a programming overhead would have to be written in by the user to enable interactive operation.

The compiler itself was an assembly language program designed along ad. hoc. lines which translated the source program into 'triples' and then carried out optimisation for common terms orientated towards arithmetic processing. Modification of the existing FORTRAN parser and code generators for MINIC expressions, where different precedence rules and variable length word sizes are used, was considered too complex. The FORTRAN routine support system was designed for high speed execution of arithmetic routines at the expense of memory. The complete single-precision and integer routines plus input/output library for the DDP 516 occupied nearly 8K locations (as compared with the simplified mathematical routine library used in MINIC described in Appendix D, which require only 1K locations). When the ADMOS operating system requirement of 4K locations is included this does not leave sufficient user workspace for mapping variables large enough to interconnect the inputs of the MINERVA hardware.

4.4.1 PL 516: A "High-Level" Assembler

PL 516 is an ALGOL inspired block structured language designed as an assembly language replacement for writing systems programs. The translator program would have been easier to modify than the FORTRAN translator since it had been written in PL 516 itself, enabling a bootstrapping technique for development. There were however, two main drawbacks. While it may be considered that block structured languages allow elegant and powerful programming techniques, such as recursion and structured programming, the architecture of the DDP 516 was not designed to support them efficiently. This was due to the lack of a hardware stack facility and the sectored addressing structure. This led to problems in PL 516 when the scope of variables crossed sector boundaries the solutions to which had to be resolved by the user. The other drawback was in the nature of the PL 516 language which is aimed at manipulating words and addresses in the computer rather than the high level atoms of net variables and mappings.

The BASIC language available on the DDP 516 was interpreted and unlike the FORTRAN system had been designed to occupy the least amount of space at the expense of execution time. No attempt to use this interpreter was made since the overhead due to interpretation were considered too costly. (A comparison between the arithmetic expression execution timing for this interpreter and the MINIC compiler is presented in Chapter 5.) However the mathematical routines and character string conversion routines from this interpreter were used for the MINIC compiler implementation after some modification to exploit the high speed arithmetic option of the DDP 516 processor.

4.5 Formal Definition of MINIC

A complete definition of the MINIC language using B.N.F. notation is given at Appendix G. This is a description of the language as implemented by the final version of the compiler written for the DDP 516. The points which differ from the standard BASIC definition are discussed below.

The syntax chosen for MINIC expressions is that of an operator precedence grammar due to Floyd (1963). This has been chosen since phrases are reduced by considering precedence relationships derived between operators which must be terminal symbols of our language; information about non-terminal symbols need not be kept i.e. no syntax tree is produced during parsing. Semantic routines are called at each reduction to generate target code for the expression. This approach was necessary for two reasons; memory was not available to store the complete parse tree and the requirement for interactive execution of source text meant the time taken to compile must be kept to a minimum.

4.6 MINIC Statements

Three statement types peculiar to MINIC have been added, these being OUTPUT, PAD, and DISPLAY.

4.6.0.0 The OUTPUT Statement

The OUTPUT statement is used to enable binary format files to be generated. The syntax is similar to the PRINT statement for character files the same terminal symbols having been retained to separate the OUTPUT list. However they have different semantic interpretation, The";" symbol produces no output but the "," symbol causes five blank words to be output. This can be useful when formatting pattern files. The OUTPUT statement with no separator before the carriage return character causes the file to be closed in the case of magnetic tape or disc output. Expressions appearing in the OUTPUT statement are evaluated and their binary value transferred to the channel number given after the channel setting symbol "#" or to default channel 4. Arithmetic values are truncated to integers before being OUTPUT.

4.6.0.1 The DISPLAY Statement

The DISPLAY statement is used to enable the setting of a C.R.T. display of a 16 x 16 binary matrix. This operates in real time on a cycle stealing basis enabling the whole or part of the contents of a pattern variable to be monitored during program execution.

4.6.0.2 The PAD Statement

The PAD statement initiates input in real-time from a 16 x 16 array of photo-diodes. This has several hardware modes of operation enabling pattern editing and creation.

4.6.1 MINIC Expressions

There are two types of expressions in MINIC, one having the value of a binary vector of variable order or a floating point arithmetic value consisting of a 32 bit floating point number in the range:

$$\mp\ 1.677215 \times 10^{-48} \text{ to } 1.677215 \times 10^{48}$$

Arithmetic expressions are evaluated according to normal rules of precedence of operators, unary operators and transcendental functions being the most binding followed by exponentiation; assignment being the least binding. Pattern expressions are made up of the variable types introduced earlier. The standard BASIC syntax for variable names of a single alphabetic character or an alphabetic one followed by a single numeric character is used. For ease of self-documentation variables may be prefixed by a type mnemonic. Since semantic information about the size of variables is required before syntax checking and code generation can take place all variables other than arithmetic ones must be declared in some previously input DIMension statement. All variable types may be subscripted to form vectors and arrays, the BASIC limitation of up to 2 array dimensions has been relaxed to enable an unlimited number of dimensions, although in practice this is limited to four-dimensional arrays. Unlike BASIC all arrays have to be previously declared. (There is no default declaration of order 10 per index).

4.6.1.0 Precedence of Operators

Pattern expressions are evaluated according to rules of precedence shown below, corresponding to a "sum of products" form.

BY>NT>AN>OR>EX>IP>BE>SUM

4.6.1.1 Use of Parenthesis

The order of evaluation of any expression may be altered by the use of parenthesis, but it is obvious from lines 22 to 34 of the syntax rules that this is not universally so. Although the syntax implies that pattern expressions are evaluated as a parallel process this is not so in practice. Evaluation takes place in successive 16 bit word segments of the expression. When temporary storage variables are required for storing intermediate results only the 1 word segment under evaluation is kept. This places a restriction on the MAPPING operation since random access of the argument pattern vector is required:

e.g. the expressions

```
        (PAT X OR PAT Y) BY MAP M
```

and

```
        PAT X BY MAP M BY MAP M1              are illegal.
```

4.6.1.2 Restrictions on the Universality of Variables

Another restriction occurs because of the way in which NET variables are implemented using the hardware NEC's: since cards are accessed sequentially using the ICAR micro-order, the current card address is not saved after the card has been accessed. This means that only one NET variable may appear in any pattern expression. However, Net expressions may appear more than once as terms in a subscript list, although this is an unlikely programming requirement.

4.6.2 Other Differences between BASIC and MINIC.

4.6.2.0 The IF Statement

The equivalence and not equivalence relational operators "=" and "<>" have been defined for pattern expressions. This enables more efficient execution than the semantically equivalent statement :

```
IF SUM< pat exp1>EX<pat exp2>(<=>)   0 THEN <cond true>
```

To enable asynchronous real-time interaction with a program the testing of four hardware sense-switches has been incorporated. When using sense-switches there is no halt in the program flow for a peripheral transfer to take place. They use identical identification mnemonics to those used in the DDP 516 assembly language.

Nested IF statements are not allowed.

4.6.2.1 PRINT Statement

This enables the value of MINIC expressions to be represented in character format. In order to print pattern expressions a FORMAT clause must be present before the expression. This is necessary since, internally, pattern expressions are evaluated to binary vectors but, externally, they usually represent two-dimensional patterns. The FORMAT clause also provides the characters for the representation of a 0 or 1. This may be omitted, in which case the default characters .,X are used for 0, 1 respectively.

4.7 Some examples of MINIC in Use

The following examples illustrate how the networks of Figures 3.1 and 3.2 could be programmed using the MINIC language.

4.7.0 A Ten Class n-tuple Discriminator

Example (4.1) is an implementation of a ten class single layer learning network for 16 x 16 bit binary patterns. The network consists of ten identical discriminators randomly connected to an input retina. The discriminators are assumed to be initialised to give zero responses by a manual reset order. They are then trained on a sample training set of 100 patterns each of their respective classes and the responses tabulated for a testing set of 500 patterns of each class. The program commences with statements for declaring variables as follows:

```
10   DIM PAT I (16,16), MAP M (256,256), R(10,10)
20   DIM NET N (0,16)(9), PAT T (1,16)
```

Statement 10 allocates storage for an input vector, PAT I of 256 bits, an interconnection vector MAP M of 256 elements and an array of 10 by 10 elements for tabulating the responses. Statement 20 allocates 10 nets of 64 S.L.A.M..- 16's, each of which has card starting addresses 16 cards apart originating from card zero, and a teach vector of 16 bits. Statements 30 to 60 below, initialise the teach vector and input the MAP vector. Subsequent re-entry of the program will ignore statements 40 to 60.

```
30 IF X <> 0 GOTO 70
40 INPUT#2 MAP M
50 LET PAT T BE NT PAT T
60 LET X = 1
```

Each NET is trained on 100 patterns of a particular class by lines 70 to 120.

```
 70 FOR A = 0 TO 9
 80 FOR B = 1 to 100
 90 INPUT#3 PAT I
100 LET NET N (A) IP PAT I BY MAP M BE PAT T
110 NEXT B
120 NEXT A
```

During testing, each input pattern is presented to every net and the one with the greatest responses causes the increment of the appropriate element of the response table. In the case of ideal classification the response table will be a diagonal matrix.

```
130 LET R1 = 0, C1 = 0
140 FOR C = 0 to 9
150 FOR B = 1 to 500
160 INPUT #4 PAT I
170 FOR A = 0 to 9
180 LET R2 = SUM NET N (A) IP PAT I BY MAP M
190 IF R1 > R2  GOTO 210
200 LET R1 = R2, C1 = A
210 NEXT A
220 LET R(C,C1) = R(C,C1) +1
230 NEXT B
240 NEXT C
250 FOR A = 0 to 9
260 FOR B = 0 to 9
270 PRINT R (A,B),
280 NEXT B
290 PRINT
300 NEXT A
310 END
```

An assembly language program of the above network was written as a commissioning test for the network hardware before the MINIC compiler was available. The comparative ease with which the MINIC program was developed can be judged from the fact that the assembly language program required 466 program statements as compared to 31 for the

MINIC program.

The above program is an example of the expanded format of MINIC expressions and is useful in being virtually self-documentary. During an on-line programming session line 100 of the above example could be typed as:

```
100N(A)IPIBYMBET
```

This reduces the number of key strokes from 44 to 17 by taking advantage of the free formatting of text and previously declared variable-type defaults.

4.7.1 A Feedback Network

The above example shows the use of MINIC where, essentially numerical data about the input patterns is required, the following example illustrates its use in implementing an 'OR'ed feedback network of the type described by Fairhurst (1973) already introduced in Chapter 3. Figure 3.2 outlines the structure of the network.

The network consisted of 36 3-input R.A.M's connected to a 6 x 6 matrix at both input and output. Random connections were made between the input matrix and the inputs of the R.A.M.'s such that each element of the input matrix was connected to three different R.A.M. inputs.

We thus have to declare the following variables:

```
10 DIM PAT R (6,6), PAT I (6,6), PAT I1 (6,6)
20 DIM NET N (0,9), MAP M (144, 36)
```

Since only 3 input R.A.M.'s are needed MAP M is organised in a way which connects one of the four inputs of our N.E.C. hardware R.A.M's to a '0'.

The input patterns are generated from two prototypes, T and H, to give patterns of a given Hamming distance away from each generator. This can conveniently be accomplished by the following MINIC program for a Hamming distance of 2. The algorithms may easily be extended for other Hamming distances.

```
10 FOR A = 1 to (N-1); N = no. of bits in generator.
20 LET PAT M2 BE PAT M1 BY MAP S
30 FOR B = 1 to (A-1)
40 LET PAT X BE PAT G EX (PAT M1 OR PAT M2)
50 LET PAT M2 BE PAT M2 BY MAP S


   NEXT B
   LET PAT M1 BE PAT M1 BY MAP S
   NEXT A


   END
```

This will exhaustively generate all the combinations of Hamming distance 2, which for N = 36 is 630 patterns. For larger Hamming distances (example, HD = 10) there are approximately 2.5×10^8 patterns which means that a random sample of patterns would have to be taken and a statistical result accepted.

The net is initialised by storing random functions in the R.A.M. elements. This is accomplished with 8 input patterns selected to provide the 8 disjoint addressing functions for each R.A.M. element. The net is then trained on each of 8 training patterns in which the occurrence of '1''s and '0''s has been randomly specified.

```
350 FOR A = 0 to  7
360 LET NET N IP PAT D(A) BE PAT T (A)
370 NEXT A
380 END
```

After training by the initial functions the network is then trained on its incoming environment and is "aged" by disconnecting teach-inputs exponentially according to the number of patterns seen. The network has been shown to perform a natural clustering of its input environment mapped on to its internal state structure. When the current input is removed the elements will still be seeing a pattern similar to its past input being fed-back through the OR input gates. This results in a cycle in the same region of state space as caused by the previous pattern and so the network can be said to display "memory"

The evaluation of the network requires the building of a directory of its state cycles. This cannot be accomplished exhaustively because 2^{36} starting states would have to be investigated.

The next state can be obtained by the following program defined as a subroutine:

```
1000 LET PAT C BE PAT R OR PAT X
1010 LET PAT R BE NET N IP PAT C BY MAP M
1020 RETURN
```

The following subroutine discards the transient part of the state trajectory and then checks for a closed cycle:

```
2000 FOR A = 1 to 100
2010 GOSUB 1000
2020 NEXT A
2030 LET PAT R1 BE PAT R
2040 FOR A = 1 TO 50
2050 GOSUB 1000
2060 IF PAT R1 = PAT R RETURN
2070 NEXT A
2080 GOTO 2030
```

The identified state cycle is then checked against a table of state cycles. In the above example an arbitrary state was chosen to identify a particular cycle. This means the complete cycle must be generated and each state checked against the key state for equivalence. The process may be made more efficient by defining some relation which will extract a state uniquely from a cycle. If the state vectors are considered as binary integers then arithmetic relational operators other than equivalence may be used. These have not been defined for pattern expressions in the current version of the MINIC language but their incorporation into future versions of the compiler would require only minimal alterations.

A program to search a table of state cycles and add the current cycle to the table if a match is not found is as follows:

```
400 FOR C =  1 TO D
410 FOR B = 1 TO A
420 IF PAT C1(D) = PAT R GOTO 480
430 GOSUB 1000
440 NEXT B
450 NEXT C
460 LET D = D + 1
470 LET PAT C1(D) = PAT R
480 REM  D is the key to the state cycle.
```

Where the number of state cycles is large the sequential search of the state table may be replaced by hashing or organising as a binary tree.

The disconnection of teach inputs cannot be achieved directly using the network hardware of Chapter 3 because the teach clock input is common to all the S.L.A.M. -16 elements. This may be accomplished indirectly in MINIC by obtaining the response of the network to the training pattern. A mask pattern, containing bits set according to teach inputs that are required to be inactive, is used to select bits from the network response. The inverse of the mask is used to select bits from the teach data pattern and the network is 'trained' on the resultant OR of the 'selected' patterns. A MINIC program for "aged" teaching is given below, the masking patterns, PAT K(A1),have exponentially less bits set with increasing A1. Each one is a randomly selected subset of the previous one.

```
500 FOR A1 = 1 TO 6

510 FOR A2 = 1 TO 6

520 LET PAT X BE PAT I1 (A2)

530 GOSUB 2000

540 LET PAT T BE PAT K(A1) AN PAT R OR NT PAT K(A1)
                                              AN PAT X

550 LET NET N IP PAT C BY MAP M BE PAT T

560 NEXT A2

570 NEXT A1
```

Hardware for indirectly disconnecting the teach clocks has been implemented for the D.E.C. PDP 11/40 controlled MINERVA system and is described in Chapter 6.

4.7.2 Other Examples of MINIC in use

The programs described in §4.7.1 are presented as examples of MINIC programming technique. MINIC and the Minerva System have been used to implement 'Action-Orientated Networks', Aleksander (1975) described in Chapter 1.4.6 and in preliminary investigations of regularly connected cellular arrays of R.A.M.'s. (Wilson & Aleksander (1975), Tomé (1976)).

Cellular arrays are of interest because of their ability to compute global properties and be somewhat easier to analyse than random structures. R.A.M.'s are connected according to a two-dimensional grid with connections to four neighbours, a local state and an external input. This requires R.A.M.'s of 6 inputs. These are realised by the 4 input R.A.M.'s of the MINERVA hardware by the use of a four-phase clock considering the propagation of the output state in one direction at a time.

MINIC has also been found of use in applications which were not considered during its development. The main one is the generation of codes for digital communications. The range of operations on binary vectors readily enables the realisation of shift registers with feedback and encoders. The application is described in Rocha (1976).

MINIC has also found some use in general-purpose numerical applications since the implementation described in the following chapter out-performs the manufacturer supplied BASIC and FORTRAN systems on the DDP 516 computer by factors of 10 and 2 respectively. A particular program modelled Schottky barrier junctions.

4.8 MINIC and A.P.L.; Similarities and Differences

A language that is potentially useful in realising network structures is the A.P.L. developed by Iverson (1962). The language has been designed to represent algorithms with a concise and consistent notation in a manner independent of a particular data representation. Variables may be scalar, vectors or arrays and operations are carried out on an entire data representation whether it is an array or a scaler. This means that the program can be written in its conceptualised form free of control loops and branching instructions, requiring fewer program steps. To enable manipulation of the complex data structures realisable, a comprehensive range of operators is provided. These require a special character set to enable a concise representation. There are no precedence rules between operators in A.P.L., the general rule is applied that expressions are evaluated right to left. This is more logical than the intuitive left to right evaluation since the assignment operator appears in its conventional position to the left of an expression. Parenthesis may be used in the usual manner to modify the order of evaluation.

The ease in which A.P.L. may be used to simulate a learning network can be demonstrated by coding the single-layer-net response expression from the example of Section 4.7.0:

```
180 LET R2 = SUM NET N(A) IP PAT I BY MAP M
```

In A.P.L. the S.L.A.M. -16 elements are simulated. The net stores are represented by an array of 16 columns and n rows, where n is the number of S.L.A.M. -16's being simulated.

The mapping or re-ordering of the binary vector I by integer vector M is accomplished by the indexing operation.

```
I[M]
```

The re-ordered result is then reshaped to form an array whose rows correspond to a 4-tuple input to a S.L.A.M. -16 element.

```
4 n ⍴ I[M]
```

To obtain the response vector from the S.L.A.M. -16 stores the rows of the binary input array must be decoded and used to index the rows of store array:

```
N[⊥4  n⍴ I[M];]
```

The number of 1's in the response vector is the reduction of the above, the subscript A, may be included as another dimension in the store array:

```
R2 ← +/N{ A; ⊥ 4  n ρI[M]; }
```

The above example illustrates how by gaining generality in the power of the language the intelligibility of the form of the program is reduced. It also illustrates how MINIC can be considered as implementing a subset of the A.P.L. semantics concerning the hardware specification uses of A.P.L.. In its original form A.P.L. was conceived without an interpreter available to execute program statements and found extensive use within I.B.M. as a systems description language for both computer hardware and software. The usefulness of MINIC would not be as a design and documentation standard but in the area of simulation where it has already found some use as mentioned in Section 4.6 in the simulation of Hamming codes.

Correspondences between MINIC and A.P.L. operations are presented in Figure 4.1.

The possibility of simulating learning networks by an A.P.L. time-shared system has been investigated by Hanna (1973) using a remote terminal connected to an I.B.M. computer at the Computer Department of Newcastle University. The C.P.U. time taken to obtain a response from a net of 1000 simulated S.L.A.M. -16 elements was 700 mS. at a cost of about 1.4p..

Figure 4.1 Equivalences between MINIC and A.P.L. Operations.

MINIC	A.P.L.	Comments
NT A	∼A	MINIC:A is binary vector. A.P.L.:A is binary vector or array
A AN B	A∧B	MINIC A and B are binary vectors
A OR B	A∨B	MINIC A and B are binary vectors
A EX B	(∼A∧B)∨(A∧∼B)	MINIC A and B are binary vectors
A BY M	A{M}	
SUM A	+/A	A.P.L. can perform generalised reduction
X+Y	X+Y	MINIC: X, Y real variables. A.P.L.: X,Y scalar or arrays
SINX	1○X	
COSX	2○X	
TANX	3○X	
ATNX	−3○X	
RNDX	?X	MINIC: X is random no. < 1, A.P.L. result is random < X
X*Y	X×Y	X times Y
X−Y	X−Y	X minus Y
X/Y	X÷Y	X divided by Y
EXPX	*X	e raised to the Xth power
ABSX	|X|	absolute value of X
PRINT X	⎕←X	
INPUT X	X←⎕	
LOG X	⍟X	natural logarithm of X
SGN X	×X	sign of X

Generally variable type attributes are fixed in MINIC by the DIM statement.

In A.P.L. type attributes are not fixed until program execution.

4.9 Further Developments of the MINIC Language

The MINIC language set described here is that which is implemented or was planned to be implemented as described in Chapter 5. This uses the available hardware resources at the completion of the first stage of hardware construction, namely the building of the net hardware using S.L.A.M.-16 elements.

There are several extensions to MINIC which would improve the generality of the semantics. The most important would appear to be in the interface between variable types. The MAP variable is essentially a read-only variable, under direct program control, due to the illegality of MAP variable assignment. This was ignored during the development of MINIC because of the intention to implement integer variables and arithmetic at a later date. Since MAP variables are simply integer vectors the interface between a MAP variable and the result of a sub-scripted integer variable is just one of symbol table definition. The mapping operation has been designed for randomly specified mapping and is inefficient in terms of memory required when the mapping consists of some algorithmically orientated operation. Examples of these are shifting, rotation or ordered bit selection with reduction of a binary vector. Such a mapping can be specified by an arithmetic mapping function which is iterated by stepping a control variable and evaluating each time a MAP-element is required. A possible syntax for mapping functions would be to use the existing function definition facilities of the BASIC language:

```
e.g.    <map term> ::= <pat var> BY  {<map var>|<  funct.
                                                  term>}

        <funct.term> ::= FN<var name> (<arith exp>)


         DEF FN F1(A) = A + 1

         PAT X BY FN F1(1)              ; shifts right one place
```

Multi-line functions would enable end conditions to be programmed. A similar interface, although a less natural one, can be defined between pattern variables and integer vectors as between map variables and integer vectors. When used in integer arithmetic statement a control variable would be used to specify a 16 bit-word-boundary segment of the pattern variable or pattern expression which would then be treated as an integer. Special arithmetic operations could then be defined to allow single bit addressing. The advantage obtained would enable localities of interest to be evaluated without the redundant evaluation of the remainder of the binary vector and improve the input and output of pattern variables by increasing the range of data representations available.

The present use of the NET variable may be viewed as representing the contents of the S.L.A.M.-16 stores selected by the IP expression. The occurrence of the NET variable not followed by an IP operand could be used to represent the binary vector formed by the whole of the S.L.A.M.-16 stores. Since accessing of the whole of a S.L.A.M.-16 store must be accomplished sequentially considerable bit manipulation would be required to arrange that all the bits of a store word are present in the same word of a NET store vector.

The implementation of integer variables is the extension of MINIC with the highest priority since the proposed extensions in the use of MAP variables and pattern expressions can then be conveniently be integrated into the language. Integer variables and expressions will improve the performance of simple FOR, NEXT loops and subscripting. However, it is shown in Section 5 that the mapping operation is the determining factor in the speed of pattern expressions therefore the improvement to the execution speed to a net program would be small. For this reason the implementation of integer variables is not contemplated until the commissioning of the micro-programmable hardware described in Chapter 6. The availability of this hardware will enable the efficient implementation of the NET stores feature described above. The syntax of the proposed integer variable would use the '%' symbol as a postfix to the variable name.

The response of the 2nd layer of S.L.A.M. -16's comprising the network hardware has been ignored in the current definition of MINIC owing to the amount of bit manipulation required to pack the 5 bit card responses in a continuous vector. This restriction is removed by the hardware modification described in Section 6.1.1 and to take advantage of this in MINIC a NET variable pre-fix to define one of 3 possible interpretations of the NET variable would be used. Operations on layer-one would use the existing syntax, those on layer-two would use the pre-fix NL2 and those in which the response vector consists of a concatenation of layers one and two would use the pre-fix NCL. Teaching of layers one and two simultaneously (by the NCL net assignment expression) would be illegal as would the generalisation of the second layer NET variables to expressions without the IP operation. This last mentioned restriction is caused by the need to have known functions in the layer one S.L.A.M.'s to be able to access the stores of the layer two S.L.A.M.'s.

4.10 A Summary

The natural development of MINIC in its relation to a particular hardware structure has been presented but it is apparent that the data types that can be manipulated are binary codes of any length; using the Boolean operators and the mapping operation it is apparent that any arbitrary transformation between binary codes may be synthesized. This leads one to question whether MINIC would be applicable to the representation of algorithms, in the theory of finite state automata and switching circuits. A complete answer to this question would be to attempt to implement some of the important synthesis algorithms, such as the Quine McClusky method for minimising Boolean functions, or methods of state minimisation of discrete automata. However this would be outside the scope of this thesis. A partial answer is given by the equivalences that are shown between MINIC and A.P.L. or between MINIC and LYaPAS a language specifically designed for programming synthesis algorithms. An advantage which may be claimed for a MINIC representation over the above mentioned languages is that the syntax does not contain any abstract symbols not found on standard keyboards and which are confusing to the network designer confronted with a practical situation.

The examples of network programs presented in the chapter show that MINIC is well suited to the task of manipulating networks at a practical level. The type of network is limited to synchronous ones because of the restriction of one NET variable in an expression and the sequential nature of evaluation. MINIC is not suited to networks where asynchronous feedback takes place, such as the CLIP array of Duff (1973) or in modelling real neural networks. In the case of the CLIP array, practical interest is only shown in fixed-point stable states of the array. A synchronous array is introduced in Chapter 7 in which the MINIC language enables investigation of state cycles with particular interest in oscillatory state behaviour.

The following chapter presents in detail the design of a translator for MINIC and the nature of the machine code generated for the control of the network hardware and the execution of MINIC statements.

CHAPTER 5

The Implementation of a Compiler for the MINIC Language

5.0 Techniques for compiler writing have occupied an area of major interest in computer science research for the past two decades. The commercial success of a computer manufacturer's hardware depends upon the efficiency and range of compilers available to the users. The effort expended in producing a commercial compiler is very large; the original FORTRAN compiler for an I.B.M. 704 took some 18 man years to write. This is described in Backus *et al.* (1957).

It is not intended to give a review of compiler writing techniques here except to justify the methods used by the author. These are comprehensively covered in texts by Gries (1971) and Hopgood (1969).

5.1 Introduction to Compiling Techniques

The object of a compiler is to accept as its input a string of symbols representing some program in a defined language and at a later time to cause some actions to be carried out which are equivalent to those defined by the program.

5.1.0 Types of Translator Systems

A compiler is more rightly called an Interpreter if the actions are carried out simultaneously with the recognition of sentences of the program string instead of causing the output of equivalent machine code to execute the actions.

Execution-wise, it is seen that an interpreter must be more inefficient than a true compiler since recognition has to take place each time an action is executed. However, interpreters for a particular language are generally easier to write since there is no actual translation required into a string of low level commands. They become useful when the language statements imply a large amount of repetitive computation, such as in matrix manipulation or string handling, which can be executed by system routines. To some extent this appears true of the MINIC language except for the fact that it is designed to the advantage of special purpose hardware which eliminates some repetitious calculations by system routines. A comparison between the BASIC interpreter and the MINIC compiler on the DDP516 computer for the execution of arithmetic statements is presented elsewhere in this chapter.

5.2 The Nature of the Compiler Problem

A compiler can be considered as consisting of three parts (i) Lexical Analysis, (ii) Syntax Analysis, (iii) Object Code Generation. In many practical compilers the boundaries between the above sections are not always distinct; in the MINIC compiler (ii) and (iii) are done simultaneously.

5.2.0 Lexical Analysis

Lexical analysis consists of scanning through the input program text, extracting variable names and building the symbol table, extracting constants, decoding reserved words and translating the program text into a more compact form for easier processing by subsequent stages of compilation. Slight modification to the syntax of the program text, such as the addition of delimiter symbols or the differentiation of operators such as unary minus and subtraction, may be made to simplify the Syntax Analysis. The syntax of reserved words and character conversions is usually expressed as a Chomsky type 3 or Regular grammar. It is well known that an acceptor for a Regular expression is a deterministic finite state automata, therefore we may program the lexical analyser using formal methods for realising f.s.a.'s. This can be as a state transition table or as a linked set of routines to analyse each state such as described by Glennie (1960).

The main disadvantage of the state transition table is the storage requirement due to the number of empty positions in the table. However this allows very comprehensive error diagnostics and error recovery to be made. The prime impetus towards formal methods however is to produce scanners that can be easily modified during the development of a new programming language.

5.2.1 The Syntax Analyser

The purpose of syntax analysis is to parse the s-language produced by lexical analysis. This verifies that the input program conforms to the grammar of the language and breaks it up into a series of structurally encoded forms that can be accepted by the target code generator.

There are two approaches to parsing; top-down and bottom-up. A top-down parse consists of making successive substitutions of the rules of the grammar starting from the head of the language until a match is found with the input string. If none is found or subsequent substitutions fail, the algorithm must back-track until all other possible substitutions have been explored. Bottom-up analysis consists of starting with the input string and searching the rules until one is found which starts with a matching leftmost terminal. A substitution is made and a new rule starting with the substituted non-terminal is looked for. If none is found back-tracking must take place and the next rule starting with the last substitution tried.

Computer programs for implementing the above general parsing algorithms would be slow and inefficient due to the many substitutions that can occur and the storage and time taken to handle the backtracking. Such algorithms are used in compiler writing systems where efficiency must be sacrificed for flexibility in language specification By placing restrictions on the form of the rules more practical parsing algorithms may be used. In the top-down case, the grammar may be restricted to ensure that at every substitution, if recognition is obtained and a later substitution fails, then no attempt need be made to try other matches for the first substitution. The parse is assumed to have failed. This is termed top-down fast-back analysis.

In the bottom-up case, the grammar can be so restricted that recognition can occur by defining precedence relationships between each symbol, terminal and non-terminal, of the language. For a unique set of relations to exist, the right parts of syntax rules must also be unique. There can be shown to be 4 types of relations, one being _no_ relation where the adjacency of the symbols is illegal. A precedence parsing algorithm would require a matrix of all the symbols of the language. Thus the MINIC language would require 124 x 124 x 2 bits, that is, 30,752 bits of storage. Since, for real-time operation using the DDP516 computer, all the MINIC compiler needed to be in core and overlays were not available, such table-driven methods were not considered useful.

If further restrictions are placed on the grammar, such that there are no adjacent non-terminal symbols, relations need only be defined between terminal symbols. This is called an operator precedence grammar due to Floyd (1963). Terminal symbols are referred to as operators and non-terminals as operands. The operator precedence parsers can be implemented in a similar manner to the simple precedence one using a matrix of relations between operators. However, the amount of storage required may be reduced by the use of precedence functions, as described in Floyd (1963). These consist of two numerical values associated with each operator. The precedence rules consist of comparing the appropriate precedence functions according to the order in which the operators appear. For a given grammar, precedence functions do not necessarily exist which will satisfy it unambiguously and their values are not unique.

Gries gives a method of constructing precedence functions from a precedence matrix consisting of a directed graph with nodes representing each precedence function, i.e. twice the number of operators. Each precedence relation then consists of a directed arc drawn between two nodes. The direction of the arc depends on whether a derivation of an operator exists to the left or right of the operator in question. The precedence functions are then given by the number of other nodes accessible from each node.

If we consider a simple grammar for arithmetic expressions

(1)	<P>::=	(<E>)	§ 5.1
(2)	<E>::=	<T>	
(3)	<E>::=	<E> + <T>	
(4)	<T>::=	<P>	
(5)	<T>::=	<T> + <P>	

Defining operator precedence relations in the conventional manner:

R ⊜ S iff <U>::= ... RS ... or <U> ... R<V>S

R ⧀ S iff <U>::= ... R<W>... where <W> reduces
to S ... or <V>S ...

R ⧁ S iff <U>::= ... <W> S ... where <W> reduces
to ... R or ...<R>V

§ 5.2

the operator precedence matrix for the grammar § 5.1 is as follows:

§ 5.3

	+	*	(	)
+	⧁	⧀	⧀	⧁
*	⧁	⧁	⧀	⧁
(	⧀	⧀	⧀	⊜
)	⧁	⧁		⧁

Precedence functions exist and can be shown to be:

$$f(+) = 4, \quad g(+) = 2 \qquad \S\ 5.4$$
$$f(*) = 6, \quad g(*) = 5$$
$$f(() = 2, \quad g(() = 7$$
$$f()) = 6, \quad g()) = 2$$

A parser using these functions will scan the incoming expression comparing f (current operator) with g (head of input string). If f (c.o.) $\leq$ g(h.o.i) then h.o.i. is pushed on a stack and becomes the new c.o.. Objects are removed from the input string and stacked until the next operator is found. If f(c.o.) > g(h.o.i.) then a reduction occurs and items are removed from the stack. This can be done by a semantic routine determined by c.o. to produce some target code or a syntax tree.

Inspecting the precedence functions of § 5.4 several points are apparent; f(+) and g(+) are both less than f(*) and g(*); g(() is greater than any f(i) and f(() is less than or equal to any f(i) and conversely for f()) and g ()).

This means single precedence functions may be used for + and *. The effect of parenthesis is to increase the value of precedence of those operators immediately enclosed by them. This can be accomplished, leaving out parenthesis from the operator precedence scheme, by calling a semantic routine which artificially increases a single precedence function for operators still in the input string after a '(' symbol. The '(' symbol is then discarded. Conversely after a ')', all precedence of later operations must be reduced. We call this a precedence multiplier. Clearly the value of this must be larger than the largest precedence function.

The preceding paragraphs describe the basic algorithm of the parser used for MINIC expressions. Considering the restrictions on the size of the MINIC compiler program, several advantages are apparent. We can use a unique value as precedence function for each MINIC operator and this can also be used as an internal code for the operator generated by the lexical analyser. Unique precedence functions are not necessary for associative classes of operators such as -, + or *, / but here they serve a useful identifying purpose. At each reduction of the stack no non-terminal symbols are involved; at the end of the parse both the stack and the input string will contain only terminating characters. This is efficient since the non-terminal symbols convey no semantic information.

Subscripted variables (operands) are not handled directly by the MINIC implementation of the algorithm described above. This is mainly due to the semantics of the pattern expressions which evaluate vectors of more than 16 bits long by a loop set up around the code for the expression. If subscripts were parsed in line with the rest of the expression an inefficient constant assignment would take place within the loop. The algorithm is implemented as a recursive routine that first scans through each expression parsing and generating code for each level of subscription then deleting the subscript expression. This places the code for subscript evaluation at the head of each expression code block.

The remainder of the syntax analysis of the MINIC language is self-evident because recursive rules only appear in the definition of expressions. The method of parsing is similar to top-down methods using recursive descent although in this case the routines required need not be re-entrant. Since the right parts of all rules defining statements start with a terminal symbol, we can easily recognise which rules to apply and then predict what other routines to apply to recognise symbols when necessary. The implementation of the whole of the syntax analyser could have been accomplished using recursive descent. However, this could have required many recursive routines which would have been difficult to debug when written in assembler language. Also the compile time storage requirement would be less predictable and much manipulation of the language might have been required to ensure that no back-up was necessary.

5.2.2 Target Code Generation

This is the least formally understood area of compiler writing and there are no general methods with which we can compare the approach taken for the MINIC compiler.

In the MINIC compiler, target code generation is linked very closely with syntax analysis. As recognition of each phrase is accomplished, so a semantic routine is called which manipulates information from the symbol table and generates machine code which is placed directly into memory for execution immediately after the syntax pass is completed. This is very fast in operation but the code generated is not optimal due to the way in which only two operands and an operator are considered at the generation time. There are three main types of non-optimality. The first may be overcome by a careful MINIC source programmer. Consider the following example: (Arithmetic expressions are used in the examples for compactness, the points illustrated are applicable to all MINIC expressions.)

```
Let  a = (b+c) * d + (b+c) *e
```

The common sub-expression, (b+c) is coded at separate times and therefore twice. This can easily be avoided by careful programming. The second type involves the order in which parentheses are evaluated. Consider the following example:

```
(a+b) * (f*g+(d+e)*(h+k))              §5.5
  1   2   3 4   5  6  7
```

Code is generated from left to right. Sub-expression 1 is encoded and the result is put in T1 while 3 is encoded.

This is stored at T2 and then 5 encoded and stored at T3. The remainder of the expression can be encoded without recourse to mcre temporary variables. The expression could however be encoded with only one temporary variable if sub-expressions 5 or 7 (the ones with the largest precedence function + precedence multipliers) were evaluated first. The third case of non-optimal code arises from the use of subscripts. Consider the following:

```
LET A(N,M) = A(N,M)  + 1
```

On its scanning through the expression for subscripts, the expression analyser treats the calculation of the array offset as two separate occurrences, which means that lengthy code to evaluate the subscript expressions and check array bounds is generated and executed twice.

In order to produce better target code, the output from the syntax analyser must be in a form whereby common sub-expressions can be recognised and their order manipulated. Such a representation would be a tree structure where common nodes could be compared and replaced to give a directed graph. Generation would take place by traversing the graph from its root node. This was not considered for the MINIC compiler since there was not enough memory to store the whole syntax tree and the speed of compilation would suffer accordingly. The only optimisation where any serious execution time benefits would occur is in the third case described above.

Another consideration in the generation of target code is the ease with which it can be relocated in memory. Position independency simplifies the handling of overlays for large programs and enables pre-compiled libraries to be built. The DDP516 FORTRAN compiler obtains position independency by the production of a relocatable object code which must be loaded into memory by an object linker program as a separate operation. This was not considered acceptable in a real-time situation. MINIC target code is not relocatable but limited relocatability could be implemented as relocation by units of a sector. This would require, however, an extra level of indirection when accessing locations outside the current sector. A table would be kept with absolute address containing the indirect links to any referenced, relocatable addresses. The appropriate entry of the table would have to be updated whenever a section of code was relocated. The relocation system was not implemented in MINIC because there was no urgent need for program chaining capability.

5.3 The Honeywell DDP516 MINIC Compiler System

The MINIC compiler programs are encoded using the DAP assembly language for the DDP516. The compiler and run-time system form one complete program which is permanently resident in memory.

It runs as a sub-system of the ADMOS operating system which enables input and output file handling to be accomplished in a device-independent manner. The operating system also provided some of the utility routines for converting character strings and for character string matching. The ADMOS operating system was specially developed for the Kent Honeywell DDP516 system a full description of which is given in Ball (1974) and a brief description in Appendix B.

5.3.0 Memory Allocation of the MINIC System

A memory map of the MINIC compiler system is shown at figure 5.1.Due to the addressing structure of the DDP516 (described in Appendix A) the memory is broken into sectors of 512 ($1,000_8$) locations. The total memory available is 16K (1K = 1024) locations i.e. 32 sectors.

Sector zero is referred to as the base sector and contains hardware interrupt locations, a hardware protected bootstrap locations for intercommunicating with ADMOS, the systems object-linker generated base links for sector intercommunication and global variables and parameters pertaining to MINIC.

The start entry point of MINIC is location 1000_8. This is used normally for a fresh copy only, except where a re-run is being made using different set-up parameters. On entry, stacks and tables are initialised and work space allocated, and when successful, an identification message is output. The restart entry-point is stored at the ADMOS location A$EP.

The program is broken up into 4 basic sections, stored in increasing memory locations:

a) A small command executive for system level commands;

b) The lexical routines;

c) The Semantic routines;

d) Common utilities, mathpack, runtime system.

The remainder of memory below ADMOS is defaulted to the workspace. The exact limits of the workspace may be set using two optional parameters input with the command line when calling MINIC from the ADMOS system. This enables programs such as DEBUG to be available when running development versions of MINIC. The first part of the workspace is allocated to fixed length tables starting with the statement number table, symbol table, forward referenced GOTO table, temporary variables, expression parser stacks, and finally the integer constants table. The addresses of all the preceeding tables remain constant throughout execution of the compiler. Immediately above the integer constant table is the source string (s-string). This is the internal form of the MINIC source program produced by lexicalanalysis. The string is permanently lengthened after every numbered statement is entered. This overwrites any code following it. The target code is placed immediately following the s-string. This is built up in two ways: executable code is written upwards from the bottom of each sector and indirect address links, integer constants and storage for temporary variables are allocated from the top of the sector downwards to the code.

The systems stack, used for recursive routines, is placed in the next free sector above the growing code or s-string. Storage space for runtime variables and real constants is allocated from the top of workspace working downwards towards the systems stack.

There are two comments on the use of the workspace. First, the use of direct access fixed length tables for the symbol tables is inefficient in terms of space. While an indirect table pointing to entries embedded in the s-string for DIM statements is a possible alternative, this would slow down table access. Where space is critical, the table sizes can be set for a particular source program by adjusting the initialisation parameters in sector 0. The second comment concerns the s-string. Space cannot be recovered when statements are being inter actively deleted from the program and can only be recovered by the scratch (SCR) command which deletes the entire workspace. Cases in which workspace is insufficient to allow online program development may be handled by the ADMOS EDIT sub-system. No attempt was

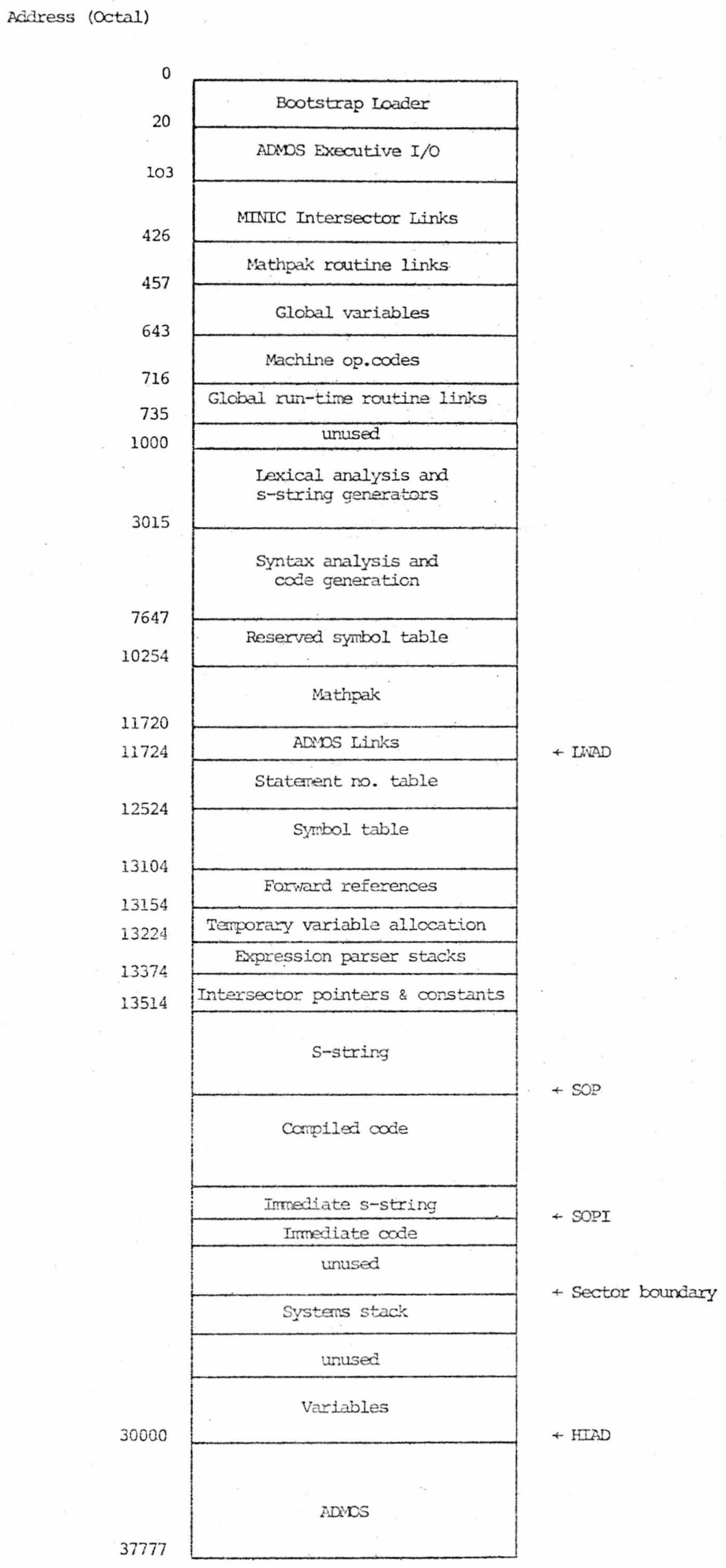


Figure 5.1 Memory Map of MINIC

made to implement a garbage collection algorithm.

5.3.1 MINIC as an ADMOS Sub-system

The MINIC compiler is entered in the sub-systems table of ADMOS. This enables the current version of the compiler to be automatically loaded and entered whenever MINIC is typed as a response to a command input request. The MINIC executive requests system commands using the ADMOS 'input-command record' routine. Any input is first scanned for ADMOS global commands before being passed to the MINIC executive. This makes all the ADMOS utilities available while a MINIC program is being developed. Program and data input use the 'input data record' routine and when assigned to a terminal both command input and data input routines allow local editing of the input strings using the backspace command.

Other facilities provided by ADMOS include the handling of program breaks: When an unidentified interupt or a 'control C' character is typed on a terminal a program break occurs. The execution of MINIC is suspended independently of the current status and control is passed back to a specified entry point in the MINIC compiler. This is set to point to the system executive and a system command input is requested. No continuation recovery is possible after program breaks but it would be feasible to program this facility into MINIC, if required.

MINIC uses ADMOS for all input and output requests. Whereas a MINIC program may assign its input or output to any channel, the compiler uses Channels 1 and 2. Output Channel 1 is shared with ADMOS and is used for systems messages such as from the SPACE command and for reporting error messages. Input Channel 1 is shared with ADMOS and is used for the system commands SCR, SPACE, PROG, COMPILE and RUN. Input Channel 2 is used for inputting program text.

5.3.2 The MINIC Command Executive.

The purpose of the command executive is to separate the system communication statements of MINIC from the program statements. This is not the case in other BASIC compiler systems but in this case there are several advantages. Program input can be made on a separate data channel thus leaving the command channel open for system communication. There is also less possibility that future developments of MINIC and ADMOS will lead to ambiguities since the range of validity of ADMOS commands is restricted. When the command input routine requests input, an "&" character is output to the command device if it is a terminal. Commands can come from a file by the appropriate assigning of input Channel 1, enabling batch mode operation, in which case the prompt character is suppressed.

A flow-chart for the command executive is found at Figure C.1 Decoding of the command string is carried out using the ADMOS decode utility, A$DC. This returns with a numerical decodee which is used as an index to a table of system routine addresses. Control is passed to the appropriate set of system routines by an indirect jump through the table.

<u>Executive Commands</u>

SCR - The SCRatch command deletes the current MINIC program by clearing all the workspace and fixed length tables and resetting the table pointers.

SPACE - This command calculates the free space remaining in the workspace from the top of the s-string, or the space remaining in the current code sector, to the start of the systems stack, plus the space between the end of the systems stack and the start of the MINIC variables. The result is output via ADMOS Channel 1.

PROG - After this command, program text is requested via ADMOS Channel 2. If this is assigned to a terminal, a ':' is output as a prompt character when the lexical routines are ready to accept input. Return to the executive is by a '.' input immediately following the prompt, by a "control C" character in the program text or a program break ("control G") via the command terminal.

COMPILE - This causes any numbered MINIC program statements previous by input to be compiled, without execution, returning directly to the executive.

RUN - If numbered MINIC statements have been added since the last RUN or COMPILE command the whole program is recompiled and then executed consecutively. Return to the executive occurs at the execution of an END or STOP statement or after a program break.

The COMPILE command has been included to enable immediate branching statements (in PROG mode) allowing execution to commence at any point in the compiled MINIC program.

5.3.3 Program Input and Lexical Analysis

The flow chart of the PROG command is shown at Figure C.2 One line of input is transfered by ADMOS from the input device buffer to the ADMOS command input buffer. ADMOS then maintains a pointer, A$CP, which is used by the lexical routines to access characters from the command buffer. Control is passed to the appropriate lexical routines in similar manner to the dispatching to the system command routines. The value of the index to the appropriate statement lexical routine is also used as the internal terminal symbol of the s-language grammar, thus providing a fast and compact representation for manipulating the s-string. Since there are no recursive rules over statements (the IF statement cannot be an argument of itself), none of the statement lexical routines are re-entrant.

When the lexical routines have successfully accepted the input line control is passed back to PROG input by way of the routine EOST (End Of Statement)which adds the line number,if defined, to the statement number table and makes a permanent record of the s-string pointer. If no statement number was input then the immediate execution flag is set and control passed directly to the RUN command. The s-string record is not made permanent after an immediate statement.

5.3.4 <u>The Symbol Table</u>

As variables and line numbers are extracted during lexical analysis, information about them is entered in the symbol table and statement number table. The tables are ordered numerically and a binary search implemented when comparing an entry. This tends to be slow when the tables are being built since each time an entry is made in the body of the table all higher ordered entries must be moved up. In practice this is not serious since in an interactive programming environment the delay for scanning each program line is always less than the data rate of the terminal keyboard. However, since symbol table access is fast, target code generation is also fast and so the delay after typing the RUN command is minimised. Ordering the statement number table also saves time during target code generation.

The statement number table consists of three arguments per entry: the statement number; a pointer to the s-string for the start of the statement; and a pointer to the target code. The last argument of each entry is filled in during target code generation. When lines are deleted from the input program text the entry for the s-string for the particular line is set to zero.

The symbol table consists of five arguments per entry: the variable name; the variable type; the run time address or origin; if an array a pointer to a dope vector, the base link address. Variable types are ascertained from a DIM statement and are encoded as follows: 0 -real; 1 - PAT; 2 - NET; 3 - MAP. Run-time addresses of variables are allocated during lexical analysis as described in Section 5.3.8. In order to simplify array addressing a dope vector is used. This is also described in Section 5.3.8. It is generated during the lexical analysis and is embedded in the s-string. Unsubscribed variables have zero for this argument. The base link address is kept during code generation and contains the address in the sector for which code is being currently generated of the pointer to the variable outside that sector.

Information about the size of a variable other than a real is kept in two locations preceding the data part of each variable, as described in Section 5.3.8 .

5.3.5 The S-String

This is the internal form of the input program and follows a similar infix format. It is placed in memory as described in Section 5.3.0. If the statement under lexical analysis is for immediate execution a temporary s-string is started at the end of any existing target code. Numbered statements cause new s-string to be appended to the existing s-string, overwriting any target code and causing the already compiled flag (ACFG) to be reset.

Information is stored in the s-string as positive integers occupying a whole word or as characters using ASCII code requiring an even number of half words. The s-language codes for reserved MINIC symbols are tabulated in Appendix F. Character information is denoted by negative words. Reserved words and operators are decoded into their internal form and are stored as positive integers with a maximum value of 161_8. Constants are converted into real numbers and stored in the run-time variables area. Their address is then stored in the s-string as an integer distinguished from operators by always being greater than 161_8.

Variable names, quote strings and REM statements are copied directly from the input program text to the s-string and are represented as characters. Statement numbers and variable type prefixes do not appear. Terminating symbols are added at the end of each subscript expression to simplify the syntax and target code generation stages. The following examples illustrate the form of the s-language.

```
10      LET PAT X BE NET N IP PAT I1 BY MAP M
         1   'X   ' 60   'N ' 61   'I1'  65 'M   '

20      IF G (A ,B )  <= F THEN GOSUB 1000
        6'G  '140'A '161 141'B '161 142 46'F '161 10 1750

30      PRINT  #3 "RESULT",        R x 100/256
        3      3   ,"RESULT",    210'R '53 144 54 400 161
```

5.3.6 Recursive Lexical Routines

As lexical analysis proceeds, each statement lexical routine calls common routines for creating the s-string and the symbol table or for recognising sub-goals. Re-entrancy only takes place when subscripted variables are encountered.

Expressions are separated into two classes by recognition of the type of the first variable encountered. Separate routines have been implemented for numerical routine expressions and for pattern value expressions. Flow-charts for the routine EXXP for numerical expressions and routine EXPX for pattern expressions are given at FiguresC.12 and C.13respectively.

Both routines implement a modified f.s.g. for the unparenthesised expression grammar below:

```
<U>    ::=    UOP|<U> UOP        |UOP - Unary Operator
<C>    ::=    <U> CONST|CONST    |CONST - Constant
<V>    ::=    <U> VAR|VAR        |VAR - Variable
<U>    ::=    <O> UOP            |OP - Binary Operator
<O>    ::=    <V>OP|<C> OP       |T - Terminating symbol
<V>    ::=    <O> VAR
<C>    ::=    <O> CONST
<G>    ::=    <V> T|<C>T
```

Parenthesis are included by considering "(" as a unary operator and incrementing a parenthesis counter. Closing Parenthesis, ")" are included by defining another state <RHP>and the following rules:

```
<RHP> ::=  <V>)|<C>)|<RHP>)
 <O>  ::=  <RHP>T                RHP - Right Hand Paren-
                                       thesis
 <G>  ::=  <RHP>T
```

The parenthesis counter is decremented at each occurrence of ")". When goal state G is reached a check is made to ensure that the parenthesis counter is zero.

Variables are extracted from the input by the routine EXV (shown in Fig.C.16) and when the subscript expressions are encountered routine EXXP is called re-entrantly using the systems stack.

It is quite possible for lexical analysis to be accomplished without recursion simply by extending the Chomnsky type 3 grammar given above. Subscript brackets could be counted as in the case of parenthesis, however checking of the subscript list against semantic information in the dope vector would not be possible thus delaying error detection until the COMPILE phase.

5.3.7 Systems Stack Organisation and Recursion

The architecture of the DDP516 is not particularly efficient for the implementation of recursive routines for two reasons: (a) the lack of hardware implemented stack and associated registers; (b) the semantics of subroutine calls and interrupts causes the return address to be stored in the penultimate location to the entry point of the routine resulting in the overwriting of any existing calls.

In the re-entrant routines of the MINIC compiler storage for local variables is maintained on the systems stack. Routines are entered in the normal way and then space is reserved for local and temporary variables on the systems stack. The routine ENTR is then called which puts the return address on the stack pushes the old stack base pointer onto the stack and loads the stack base pointer with the current stack pointer. All references to local variables on the stack are made by offsets to the stack base pointer.

The routine RETN is used to return from a re-entrant routine. This obtains the return address and previous stack base then flushes local variables from the stack. Arguments to be transferred back to the calling routine are placed in locations above the stack base. These are copied when the local variables are removed so that their relationship with the stack base is constant.

5.3.8 The DIM Statement and Internal Representation of Variables

The DIM statement is used for communicating with the symbol table during the lexical phase of compilation. It is not an executable statement and re-DIM-ing of variables is not allowed due to the absolute storage allocation of variables before target code generation takes place. All variables other than unsubscripted real variables must appear in a DIM statement which precedes the use of the variable in any other statement.

The s-string for the DIM statement contains the dope vector for each subscripted variable. The flow chart of the DIM statement processor is shown at Figure C.6

MINIC variables are stored as follows:

Real variables: Two word 32 bit floating point format, 23 bit mantissa plus sign bit and 8 bit exponent. Lowest address contains sign, exponent and most significant 7 bits of mantissa.

Pattern variables: The first two words contain the two-dimensional size in bits of the pattern corresponding to the limits appearing in the DIM statement. Data is stored in successive locations, 16 bits to a word. The number of bits is rounded up to a multiple of 16.

MAP variables: The first two locations contain the maximum range and domain of the mapping as specified in the DIM statement. The map is stored as a vector of 16 bit signed integers in increasing locations corresponding to the range of the mapping. A negative value of a domain element causes a '1' mapping and a zero value a 'O' mapping irrespective of the domain pattern. This enables masking operations to be made directly during a mapping.

NET variables: These require two words, the lowest address containing the card start address in the MINERVA hardware and the next, the number of cards.

ARRAYS: These are mapped into store as vectors requiring the minimum storage, such that integer changes in the values of the last subscript addresses adjacent elements in the vector. At run-time the code uses a pointer to a dope vector to check the array bounds and calculate the mapping function. For an array declared as follows:

A (j0, j1, ..., jn)

array dimensions are labelled 0 to n inclusive.

The dope vector is constructed as:

j0, j1, di, j2, d2, ... jn, dn;

where dn =((jn - 1) + 1) dn-1 , d0 = 1 .

It is unnecessary to include d0 in the dope vector. The array element A (i0, i1, ... in) is mapped into the storage vector element by:

$$\sum_{0}^{n} i_k \, d_k$$

The absolute address of the element is given by:

$$CVA + CVS \left(\sum_{0}^{n} i_k \, d_k \right)$$ where CVA is base element address
CVS is size of element.

The size of the element CVS includes the two size defining locations prefixing each element for pattern, net and map variables. These are copied into each element of the array in the run-time variables storage area by the DIM statement processor. As net arrays are generated so the card starting address for each element is suitably modified.

Array accessing in MINIC is conventional and tree structured methods for obtaining faster access were not considered due to the larger storage requirements and the amount of processing carried out on each element after accessing being generally more than that required to perform the access make any speed savings insignificant. Information normally kept in the symbol table about element sizes is kept in the run-time variable area.Since this is the only symbol table information required at run-time the target code can easily be made to run free-standing.

5.3.9 Errors During Program Input

The object of the lexical analysis phase of the MINIC compiler is to trap as many programming errors as possible and inform the operator of them immediately after each statement line is processed. Lines containing identified errors are not inserted in the statement number table or in the s-string. When used interactively the line may be corrected and re-typed immediately.

Errors are reported to the operator via ADMOS channel 1 in the form of a two character mnemonic followed by identification of the statement number. A listing of the error mnemonics given by MINIC 21/8/74 is given at Appendix I.

There are two types of error that may occur during Program input and lexical analysis. Overflow of a stack or table causes a system error which is generally unrecoverable. However, the compiler may be re-entered from address 1000_8 with different initialisation parameters to enlarge tables where the program text itself is not too large. Syntactic errors in program text either have specific error mnemonics such as M((missing left hand parenthesis) or they print the first character of the unrecognised field e.g.

```
100  LET  A  =  A+ * B
*? IN LINE NO 100
```

Where a non-printing character is involved, the octal representation of the offending byte is given.

Some semantic errors are detected, namely the checking of variable sizes for compatibility in pattern expressions and the dimensionality of arrays but others, such as the correct nesting of FOR loops and undefined line number references, are detected during target code generation.

Serious system errors, such as the passing of program control into unused locations, result in a logical error condition. These are caused by undiscovered faults in the compiler logic and after printing the originating address return control to ADMOS.

5.3.10 The Syntactic Routines and Target Code Generation

When the RUN or COMPILE system command is executed control is passed first to an initialisation routine and then to a set of Syntax analysis routines, one for each statement type.

The initialisation routines are shown in the flow chart at figure C.18. These set pointers to generate target code directly above the s-string and obtain the s-string pointer to the lowest numbered statement from the first entry in the statement number table. As this and subsequent s-string pointers are obtained so the entry for the target code address is completed by entering the current sector code pointer (C S C P) in the table.

The next field in the s-string determines the statement type. Since the"true"clause in IF statements is a statement they are completed here. Location IFA1 contains the IF statement false clause code which is handled as a forward referenced GOTO to the next line. Control is then passed to the appropriate syntactic routine where target code generation takes place.

Return from the syntactic routines is to the entry-point Statement Compiled (SCOM) shown in the flow chart of figure C.20 This completes any previous references to the current statement by searching the forward reference table and completing the indirect link addresses. If the statement is for immediate execution, an instruction to return control to the PROG mode is placed at the end of the immediate execution code and then it is executed.

A permanent record of the workspace pointers is made and then a check to see if all the statements in the statement number table have been compiled. If no more statements exist the compile only flag is checked and control passed to the command executive or the compiled code accordingly.

5.3.11 Desectoring Target Code

The addressing structure of the Honeywell DDP516 only allows direct addressing of arguments within sector zero or the same sector as the instruction. This means that a table of address constants must be maintained accessible to the instruction to enable indirect addressing of locations in other sectors. In the MINIC system, sector zero is reserved for system address constants and global variables and so the indirect address constants must be maintained in the sector in which target code is being generated. Locations are allocated to this starting from the top of the sector downwards. An ordered table is also kept. This is called the integer constant table and it contains two entries; the value of the address or constant as the key and its address in the current sector. As each memory reference instruction is generated the integer constant table is searched using binary search to see if the literal value of the constant or address already exists in the current sector. If it is not in the table a location is allocated from the current sector variables area and its address entered in the integer constant table. If the value of the integer constant is to be used as an inter-sector link the indirect bit of the memory reference instruction is set.

As target code generation proceeds, before a new instruction or address constant is added to the current sector, a check is made to ensure that at least two locations remain between the top of the code and the bottom of the address constants. When there is no more room in the current sector the generation of the current statement is aborted and the code:

```
JMP* * + 1          ; jump indirectly
DAC <next sector>
```

emitted. The integer constant table is then cleared and the systems stack, containing any uncompleted FOR/NEXT loops, rewritten in sector higher after flushing any entries made by the recursive routines.

5.3.12 Forward Referenced Locations in the Target Code

Forward referenced locations are generated explicitly, as in the case of the GOTO, GOSUB and ON statements, or implicitly, as in the case of the IF and FOR statements. When generating code for program jumps, those within the current sector may be made with a direct instruction, whereas outside require an indirect instruction. For backward jumps the decision to generate an direct or indirect instruction can be made immediately but to simplify the generation of forward jumps they are generated as indirect instructions only.

In order to complete forward references a table is kept consisting of the forward referenced line number and the address of the indirect link. When a forward reference is made the table is searched and if no link is found in the current sector an entry is made. At the completion of each statement line, excluding immediate statements, the whole of the forwards reference table is searched for references to the current statement. The appropriate address is then inserted in the indirect link and the entry deleted from the table.

When the whole program text has been compiled any non-zero entries are flagged as "Jump errors".

5.3.13 Target Code for Expressions

All MINIC expressions, both pattern and numeric, including those with relational operators are handled by one recursive routine called GEXC. The flow chart for GEXC is given in figure C.29. Unlike the remainder of the target code generation, which only requires a single pass through the s-string, expressions require two or three passes. This is necessary to ensure that the instructions for evaluating subscripts and initialising pattern and net variables occur outside any loops used in the evaluation of pattern expressions.

The method of parsing has already been presented in Section 5.2 of this Chapter. The implementation described here requires two stacks, one for operators and one for operands or variables. Each operator requires one location but variables have two entries, one being the variable name and the other being non-zero for subscripted variables, when it contains the address of the temporary variable used as the array index.

The operation of GEXC is best illustrated by use of an example, for which we shall use statement number 180 of Section 4.7.0.

```
180 LET  R2 = SUM NET N(A) IP PAT I BY MAP M
```

This is translated by the lexical routines into the s-string:

```
 1 2  3  4  5  6  7   8  9    10  11 12 13 14 15 16  17
|1|R2|47|37|40|N|140|A|161|142|O|61|I|65|M|161|215|
LET  = SUM       (          )    IP    BY
   ↑
  COP
```

On entering GEXC the s-string pointer, COP , is pointing to entry 2. COP is saved on the systems stack to enable multiple passes and recursive re-entry to GEXC. The first pass through the s-string is used to evaluate the subscripts. SUM blocks, however, are evaluated by a recursive call of GEXC during the final pass and are ignored until then. S-string entries are evaluated incrementally until entry 4 and then ignored until entry 16. Entry 17 is the statement terminator which causes GEXC to enter its second pass. This is used to initialise the expression if it is a pattern one, again SUM blocks are ignored. The type of expression is ascertained by examination of the first variable, in this case R2, which being a real variable, causes no further scanning of the expression. The pointer COP is restored to the beginning of the expression and the third pass started. As variables are encountered they are placed on the variables stack and the next entry brought from the s-string. Since subscripts have already been dealt with they are ignored during this pass. Returning to the example at this stage the stacks are as follows:

```
                          3   4   5          17
                          47|37|40|.....|215|
                          ↑
                         COP


VSKP →     0   R2

SVSK →     0   0      0  ← OSKP                    § 5.6

                             SOSK
```

Since there have been no parenthesis the precedence function for the current operator is simply its internal code. The precedence rule {OSKP} < {COP} is applied and the current operator placed on the operator stack. The next entry in the s-string denotes the start of a SUM block. Since {OSKP} > {COP} a semantic routine is called which acts upon the SUM block in the s-string rather than reducing the stacks. The SUM block semantic routine now makes a recursive call of GEXC.

The state of the stacks at this point is :

```
                              5  6   7          17
                             |40|N|140|.....|215|
                              ↑
                             COP


VSKP
SVSK  →    0    0        0   ← OSKP                § 5.7
                                SOSK
           0    R2       47

           0    0        0
```

Effectively, new stacks for operators and variables have been started above the existing entries.

The first initialisation pass now takes place to evaluate subscripts in the SUM block. The variable N is found to be subscripted. A recursive semantic routine is called which generates code to initialise the dope vector pointer. Temporary variables are generated to hold the values of the pointers.

```
The code:    (1)    LDA   DVA       ; Dope vector address
             (2)    STA   DVAP      ; Points to dope vector
```

is placed in the current code sector. GEXC is then called recursively to generate code to evaluate the first subscript expression.

The state of the stacks at this point is:

```
                                   8   9    10  11        17
                                  |A|161|142|0|.....|215|
                                   ↑
                                  COP



                         0  ←  OSKP
                               SOSK
VSKP →   0     0        40
SVSK                                                  § 5.8
         0     0         0

         0     R2       47

         0     0         0
```

Initialisation passes of the expression up to entry 9 then take place with no initialisation code being found necessary. Code generation then takes place with variable

A being stacked. Entry 9, the recursion terminator, causes special action and is translated to the current operator, {COP} = -1. Since {COP}<{OSKP} , a semantic routine, determined by {OSKP}, is called to perform stack reduction. This is a termination routine which completes any pattern expression loops, not applicable in this case, and then uses the final entry in the variables stack as the value of the expression. Code is generated to place this in the machine accumulator e.g.

```
        (3)     DBL         ; go 32 bit
        (4)     DLD* AADR   ; get A
```

A recursive return then takes place from GEXC and the stacks revert to the condition of § 5.7. However, COP now points to entry 10. The generation of code for subscripts is continued, first with code to convert from floating point to integer and then to compare with the first array bound in the dope vector.

e.g.

```
        (5)     JST*  IFXT
        (6)     CRA                 ; F.P. to integer
        (7)     CAS*  DVAP          ; Compare
        (8)     JMP*  OBEA          ; Out of bounds
        (9)     NOP
```

COP is then interrogated and corresponds to the subscript terminator at entry 10. Code is generated to calculate the array index according to the type and size of the subscripted variable.

e.g.

```
(10)        LGL 1          ; CVS = 2
(11)        ADD NADR       ; Add base to index
(12) S1     NOP            ; Just for now
```

Location S1 may be modified later in the generation according to the context of the subscripted variable. The address of S1 is then written into the s-string at position 11. The next pass through the SUM block initialises the NET and PAT variables. On recognition of the first variable, N, a check is made on the context to see if the expression is a teach or response one. A simple global flag is used to ascertain whether an assignment operator has been scanned since the start of the expression. In this case, the flag will be set denoting initialisation for a response is required. The following code is generated:

```
(13)        LDX    NEXS    ; Net expression size
(14)        LDA*   SCVA    ; Get start of net
(15)        JST*   MOAS    ; Minerva Output address
```

The subscript block is ignored, then S1 is rewritten as STA SCVA, and the address of SCVA is placed at entry 11 in the s-string.

The IP part of the expression is then analysed to calculate the size of the IP pattern expression and a temporary variable is initialised to store the incremental value of the SUM block.
e.g.

```
(16)      LDA PEXS          ; Pattern expression size
(17)      STA NTMP
(18)      CRA               ; initialise SUM temp.
(19)L1    STA STMP          ; SUM loop
```

The address L1 is noted to enable looping after the response from a group of 4 cards comprising a 16 bit response word has been 'summed'. Code to initialise the index register for the IP expression is now generated. For this the contents of the index register for the net expressions must be saved and a loop set up to interate 4 times.
e.g.

```
(20)      STX NST           ; save net loop index
(21)      LDA = -4
(22)      STA NCNT          ; net response counter
```

This completes the initialisation of variables in the SUM block, some more initialisation is yet to take place on recognition of the IP operator from the s-string.

The s-string pointer is returned to the beginning of the SUM block in the s-string and the generation pass is started. The stacks are in the same condition as in § 5.7 and the head of the s-string is entry 6. Generation proceeds by stacking N and ignoring any subscript blocks until the next operator, entry 11, (SCVA), is also stacked with N. The occurrence of IP at the head of the s-string causes a semantic routine to be executed which initializes the inner loop to evaluate the IP expression and generate a temporary variable for the partial net response.

The code is as follows:

```
(23)      LDX NTMP    ; index for IP response
(24)      CRA         ; initialise net response
(25)L2    LGL 4       ; accumulate net response
(26)      STA NRES    ;
```

Stacking of operators and variables proceeds, obeying the precedence rule, until the terminating entry 16 is reached. This is translated as before, to the precedence function -1. At this point the stacks are as follows:

```
                              16   17
                             |161|215|   s-string
                               ↑
                              COP

VSKP→      0       M          65       ←OSKP

           0       I          61

         (SCVA)    N          40                     § 5.9

SVSK→      0       0          0        ←SOSK

           0       R2         47

           0       0          0
```

Stack reduction now takes place and a semantic routine corresponding to the BY(65) operator is called. This removes the BY operator and the top two variables from the variable stack and generates the following code:

```
(27)      JST*   MPSB      ; MAP routine
(28)      DAC    I
(29)      DAC    M,1
```

Since the result of executing the above code is in the accumulator an identification flag is placed on the variable stack:

```
                                   16   17
                                  |161| 215|  s-string
                                    ↑
                                  COP = -1

VSKP→   0       (A)         61     ←OSKP

      (SCVA)     N          40                    §   5.10

SVSK→   0        0          0      ←SOSK

        0        R2         47

        0        0          0
```

The reduction of the IP (61) operator generates a call to a routine to output the accumulator to MINERVA and obtain a 4 bit response in the accumulator. Code is then generated to complete the inner IP pattern expression loop, L2.

```
(30)   JST*  MINI    ; get net response
(31)   ERA   NRES    ; combine responses
(32)   IRS   NTMP
(33)   NOP
(34)   IRS   NCNT    ; loop for 4 responses
(35)   JMP   L2
(36)   LDX   NST     ; index for net expression
```

The next operator reduced, treated as a unary operator is the internal SUM operator. The external SUM operator, so called because it is external to the recursive evaluation of the SUM block, has already been reduced. The SUM semantic routine generates a call to a routine which calculates the number of bits set in the accumulator. The accumulator at this point contains the partial expression result. This is added to previous partial results.
e.g.

```
(37)   SZE           ; saves time if zero
(38)   JST * SUMR    ; SUM bits
(39)   ADD   STMP    ; accumulate
```

The next entry on the operator stack is the terminator and this causes the net expression loop to be completed and a recursive return from GEXC.

```
e.g.        (40)     IRS 0              ; complete net loop
            (41)     JMP L1
```

Control is passed back to the semantic routine for the external SUM operator. Generation for this is now completed with the conversion routine for integer to floating point and a symbol representing the floating point accumulator, (A), (B), pushed on to the variables stack. Generation of the expression proceeds, ignoring the SUM block. The stacks are now as follows:

```
                                                  17
                                                 |215|   s-string
                                                  ↑
                                                 COP= -1


 VSKP →    0       (FP)                                         §5.11
           0        R2          47   ← OSKP
 SVSK →    0        0            0   ← SOSK
```

Entry 17 in the s-string is translated, as a terminator and causes the final reduction of the stack. The semantic routine called by the assignment operator (47) simply generates code to move the result specified by the top of the variables stack into the location specified by the penultimate variables stack entry.

```
e.g.        (42)    JST*  FLOT      ;convert to F.P.
            (43)    DST*  R2AD
```

A final termination routine is executed which completes the loops in pattern expressions as above and final exit and systems stack clean up is performed.

The target code generated for several other examples of expression are given in Appendix C.

During generation, a global variable, DBLF, is kept according to the required mode of the DDP516 arithmetic unit. Real variables are loaded and stored using double precision mode. Run-time routines that have integer or real arguments always exit in the appropriate mode, otherwise mode change instructions have to be generated to set the correct mode for following instructions. End of line statement boundaries are always set to be single precision mode (even though this causes some obvious inefficiency) to simplify the generation of line referencing instructions. This could be obviated by another argument in the statement number table containing the mode on entry to each statement and a modification of a forward referenced institution to enable the latter insertion of a mode changing instruction. The extra system overheads appear large to the gains thus obtained.

5.3.14 Temporary Variables

During the course of generating code for expressions, many temporary variables are required. Whenever the code for the single precision or double precision accumulators falls below the second place in the variables stack in GEXC, a temporary variable is required to store a partial result. The allocation of temporary variables is fully optimised and is made in the same sector in which the first reference to it is made. A list is kept of addresses as they are allocated to temporary variables, each entry is marked with a flag when it is busy, i.e. when it appears in the variables stack, or used in an expression loop. When a new temporary variable is required, the whole list is searched to find one with its busy flag reset. If none is found, a new location is allocated from the current sector. There are two such lists for temporary variables; one contains single word temporary variables for use in loops and pattern expressions and the other two word temporary variables for use in floating point real expressions. When there is no more room in the current sector, the temporary variable lists are cleared.

5.4 Execution and Run-Time Routines

Examples of target code produced by various MINIC statements are given at Appendix C and the run-time routines are tabulated at Appendix D. The code for pattern expressions evaluates the vector arguments incrementally, one 16 bit word at a time. For all operations with a one-to-one function between elements, such as boolean operations, when temporary variables are required, only the 16 bit segment, currently being evaluated, needs to be stored. Mapping operations are not restricted to a one-to-one correspondence and limitations are placed on the syntax to ensure temporary variables are not needed.

5.4.0 Run-Time Addressing of Variables

Successive words of pattern variables are addressed using the index register. This is also used as a counter for the pattern loop iteration. The index register is initialised to contain the two's complement of the number of segments in the vectors of the expression and the indirect link word, for each variable is set to point to the last segemnt of the vector with the index or tag bit of the link word also set. This is called post-indexing. After initialisation, an indirect reference through the link will point to the first segment of a variable. At the end of each iteration the index register is incremented and a branch is made out of the expression loop when it contains zero.

5.4.1 Evaluation of Operators

Where possible, operators have been translated directly into machine code equivalents:

```
NT  -  CMA       ; complement
EX  -  ERA       ; exclusive OR
AN  -  ANA       ; AND
```

Since boolean operators are commutative, the order in which the top two variables are reduced from the variables stack is modified accordingly in order to reduce unnecessary load and store operations. This means that code produced for pattern expressions is very efficient. Operations such as OR and mapping require calls to subroutines. Floating point arithmetic operations are also executed by subroutine calls. The calling sequences are similar in all cases, arguments being transfered directly in the accumulator where possible.

```
A  <op>  B

<Arg A in Acc.>

JST * op subr.      ; Base link to external routine
DAC B Arg           ; address of B argument
<return arg. in Acc>
```

Minerva operating routines have only one argument and result because the net addressing arguments are set up during net initialisation. They are concerned with generating net micro-orders to output data, input response or teach and increment the card address counter.

5.4.2 The Mapping Routine

The mapping routine has both the argument addresses following it and returns with the partial map results in the accumulator:

```
JST * MAP
DAC <PAT var>
DAC <MAP var>, 1     ; post indexed
<return arg.in acc.>
```

The mapping operation is by far the most time consuming single operation because the mapping routine has to make sixteen accesses of the map variable and use each map element to bit access the pattern argument. The routine uses masks to achieve bit addressing rather than by using shift operations. The appropriate mask is selected by the four least signigicant bits of the map element. Improvement in the speed of the mapping routine can only be bought by doubling the memory requirement of the map variable. The decoded bit address could be included in each map element increasing the storage of each element to two locations. This would reduce the execution time of the mapping routine by 30%

5.5 The Performance of the MINIC Computer System

The execution time of various MINIC statements can be assertained from the examples given in Appendix C and the summary of the run-time routines given in Appendix D.

The performance of arithmetic expressions is governed by the mathmatical routines for floating point operations and that of pattern expressions by the number of mapping operations. Other pattern operations represent about 0.5% of the execution time in a mapping expression, Subscripted variables in expressions incurr an overhead of about 80 μ S plus the subscript evaluation per subscript. Since this is independent of the size of any pattern expressions it represents only a small amount of computing time in a practical network. For this reason there would be little advantage from implementing integer variables for use in control and subscripting expressions.

5.6 A Summary

The relevance of some formal techniques of compiler writing in their application to the development of a practical compiler using a system of limited resourses has been discussed and an algorithm for producing target code directly from infix expressions of unspecified variable lengths has been described. An example containing the salient features of the MINIC language has been used to illustrate the implementation of the algorithm. It has been shown that the mapping operation is the limiting factor in the execution speed of pattern expressions to which only limited improvements imposing a disproportionally greater memory requirement, can be made by software alone.

The following Chapter discribes hardware specifically designed to overcome the limitations imposed by mapping and yet preserve the generality proffered by the MINIC language.

CHAPTER 6

Micro-programmable Modules and Other Hardware Features for Improving the MINIC System

6.0 This chapter describes the second phase of hardware development which was instigated by the need for improved execution-time performance of network interconnections. The limitations of the hardware system were noticed at the inception of its design as has been presented in Chapter 3. Various proposals for mapping hardware were considered but the final decision concerning the hardware was not taken until the software specification had been developed as described in Chapters 4 and 5.

During this phase of hardware development the system was transfered from the University of Kent to the Department of Electrical Engineering, Brunel University. The system is in the process of major reconfiguration around a Digital Equipment Coporation PDP11/40 as a host computer. While interfaces to this machine were being prepared modifications to the network hardware were carried out. These are described in the following section.

6.1 Modifications to the Network Hardware

Three modifications have been made to the network hardware described in Chapter 3. The first concerns alterations to the control logic to enable interfacing with a D.E.C. PDP11/40 computer, the second and third modifications implement hardware for response and training manipulation which was previously done by software. A schematic diagram of the modified network hardware is shown at Figure 6.1.

6.1.0 Interfacing with a D.E.C. PDP 11/40 Computer

The network hardware is designed to interface directly with two control processors: a special purpose modular microprogrammable processor and a D.E.C. PDP 11/40 computer via a DR11C general purpose interface card. The interface with the micro-programmable processor is fully described in Section 6.6.1.1. The main difference between interfaces for the Honeywell DDP 516 and that for the D.E.C. PDP 11/40 occurs in the generation of micro-orders for control of the network. The PDP 11/40 has no equivalent of the Output Control Pulse (OCP) instruction of the DDP516. This means that micro-orders cannot be generated directly under program control and have to be derived from a data channel. The DR11C interface consists of independent input and output channels of 16 bits and a control register. Bits in the control register are used as flags for synchronising input and output and for interrupt control, two bits being available under program control for passing control information to a peripheral device. One of these, CSR0, is used by the network hardware to specify whether the current output from the PDP 11/40 is to be interpreted as a data or a control word. The disadvantage of this is that an extra instruction must be executed before any output or

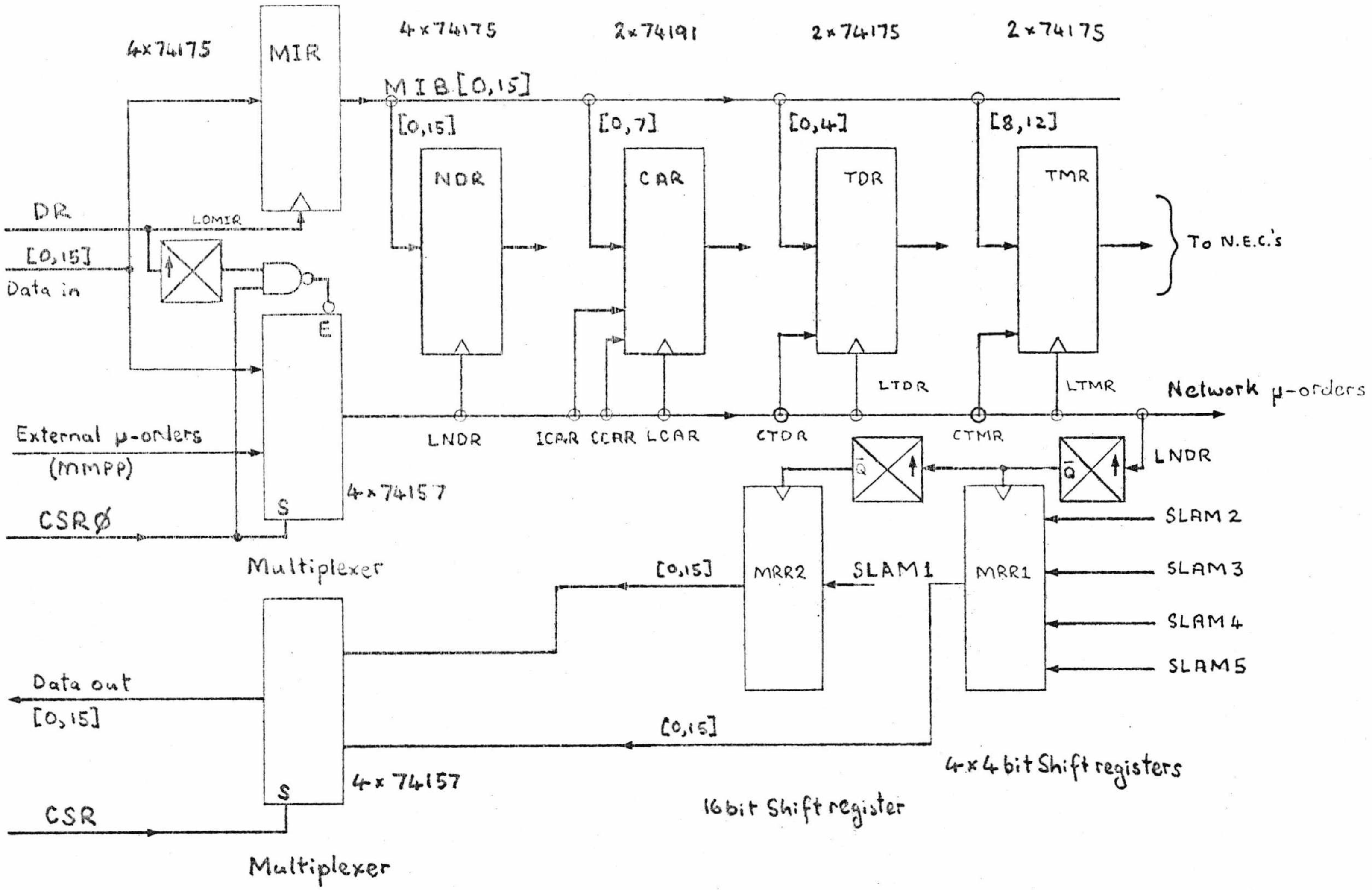


Figure 6.1 Modified Network Hardware (Schematic)

micro-order is given However, micro-orders are now horizontal in format enabling two or more to be generated simultaneously.

In order to hold the current data output from the PDP11/40 interface while micro-orders are generated a 16 bit register (labelled MIR) is provided. This is loaded each time a data word is output from the PDP11/40. The output of MIR may be transfered, via the Minerva Input Bus, (MIB), to any of the input registers on receipt of an appropriate micro-order. Unlike the DDP 516 computer interface where peripheral interchanges take place synchronously with the processor instruction clock, the PDP11/40 computer uses asynchronous interchanges. This requires that protocol signals are provided from the network hardware to acknowledge when a data transfer has taken place.

The micro-order assignments for the PDP11/40 interface are tabulated in Appendix E2.

6.1.1 Response Logic for the Network Hardware

The five bit output bus of the network hardware described in Chapter 3 had two disadvantages and led to the delaying of the incorporation of layer 2 responses into the MINIC language as described in Chapter 4, Section 9. The first disadvantage is caused by having the response word smaller than the input channel width of the computer. This is inefficient due to the multiple input and bit manipulation instructions that are needed to pack the response bits into 16 bit words. The second disadvantage is caused by inputting the 2nd layer response at the same time as the first layer response. This is also inefficient because the second layer response is handled one bit at a time and needs to be sent to a separate destination address from the first layer response.

Instructions are saved by shifting the card output response into five shift registers as shown in Figure 6.2P. The second layer response is shifted into a 16 bit serial-in, parallel-out register and the 4 first layer responses are shifted into 4, 4-bit serial-in parallel-out registers. The clock pulses for the shift registers are derived from the LMDR micro-order. A mono-stable chain is used to delay the respective shift register clock pulses by the propagation delay through the S.L.A.M.-16 elements. This is set to 1.5 μS for layer one and 3.0 μS for layer two. The outputs from the layer one shift registers and the layer two shift register are taken via a two-way data multiplexer to the PDP11/40 input interface. The select input of the multiplexer is derived from bit one of the Control and Status Register (CSR1), which is used to enable separate input of either of the S.L.A.M.-16 layers. The response registers are identified by the mnemonics MRR1 and MRR2 respectively.

The execution time improvement for layer one responses is 28% for the example of the NET expression given in Appendix C8.8 and the number of locations required reduced by four from 33 to 29.

6.1.2 Teach Logic for the Network Hardware

The example of a feedback network given in Chapter 4.7.1 requires two features not present in the network hardware of Chapter 3. In order to supply the S.L.A.M.-16 elements with individual teach and teach clock data more sophisticated teach control logic was required. The modified teach logic is shown in Figure 6.3P.

Separate teach data inputs to each Network Element Card, (N.E.C.), are already provided and control of these inputs is provided by a 5 bit teach data Register (MTDR). The original setting and resetting of all the teach information directly by micro-orders is not used except that a clear teach data register,(CTDR),micro-order is provided. Teach data information is obtained from the computer output data interface via the Minerva Input Register (MIR) and it is loaded from the least significant bits of the right-hand halfword of Minerva Input Bus,(MIB 10→15), into MTDR on receipt of the micro-order (LTDR). In order to minimise the number of bus interconnections to the N.E.C.'s advantage is taken of the fact that teaching of both layers cannot be done simultaneously. Only four teach data inputs are provided to an N.E.C.; the teach data to S.L.A.M.-16 one of layer one and the teach data to the second layer S.L.A.M.-16 are multiplexed my means of the respective teach clock micro-orders, TE1 and TE2.

Section 4.7.1 has described a MINIC program segment that enables the selective teaching of S.L.A.M.-16 elements. This requires the network to be cycled twice for each training pass and storage to be provided for the network response. The logic of lines 540 and 550 of the MINIC program in Section 4.7.1 is implemented by hardware without the need of storing the whole of the network response. The network hardware is used in a "read before write" manner. A 5 bit teach mask register (MTMR) is provided whose contents are used to select whether a particular N.E.C. teach data input takes the value of a teach data register (MTR), output or the corresponding current addressed N.E.C. output. The inputs to the teach mask register (MTMR) are connected to the least significant bits of the left-most half-word of the Minerva input bus. (MIB 3→7). This enables simultaneous loading of the teach data and teach mask registers.

6.2 Further Considerations for Interconnection Mapping Hardware

It was considered that at the time of network hardware design and construction described in Chapter 3, flexible mapping hardware could not be built cost-effectively and specification of mapping hardware requirements could not be made fully until development work of the software system completed. However the rapid introduction of Medium and Large Scale Integration (M.S.I. and L.S.I.) and the corresponding decrease in hardware system cost which occurred during the time of network hardware's construction and software development inspired several attempts to design low cost flexible mapping hardware. Two alternative designs are described in this section. One uses a sequential algorithm similar to the software algorithms already described and the other achieves some parallelism by decomposition of the map vector into equivalent shift classes.

6.2.0 A Fixed Structure Sequential Fast-Mapper

By imposing restraints on the variety of network structures realisable a special purpose mapping controller was designed to operate with the network hardware. A schematic diagram of the controller and its interfaces is shown at Figure 6.4. The existing network hardware is reconfigured as 16 networks of 16 N.E.C.'s each. All the cards of a single network are addressed in parallel. This does not require any modification of the network hardware since the N.E.C. address decoding logic is physically partitioned into decoding two groups of 16 N.E.C.'s per rack.
The N.E.C. input buses for each rack are provided from the parallel outputs of 256 bit shift register, (NISR). The S.L.A.M. outputs are taken to a buffer memory (ORB) consisting of two banks of 16 x 80 bit words. One bank is used to store the current response and the other the immediately previous response. The address decoding of the Output Response Buffer is organised such that the writing of 16 bit words can also take place and reading can be of 16 bit or single bit words.

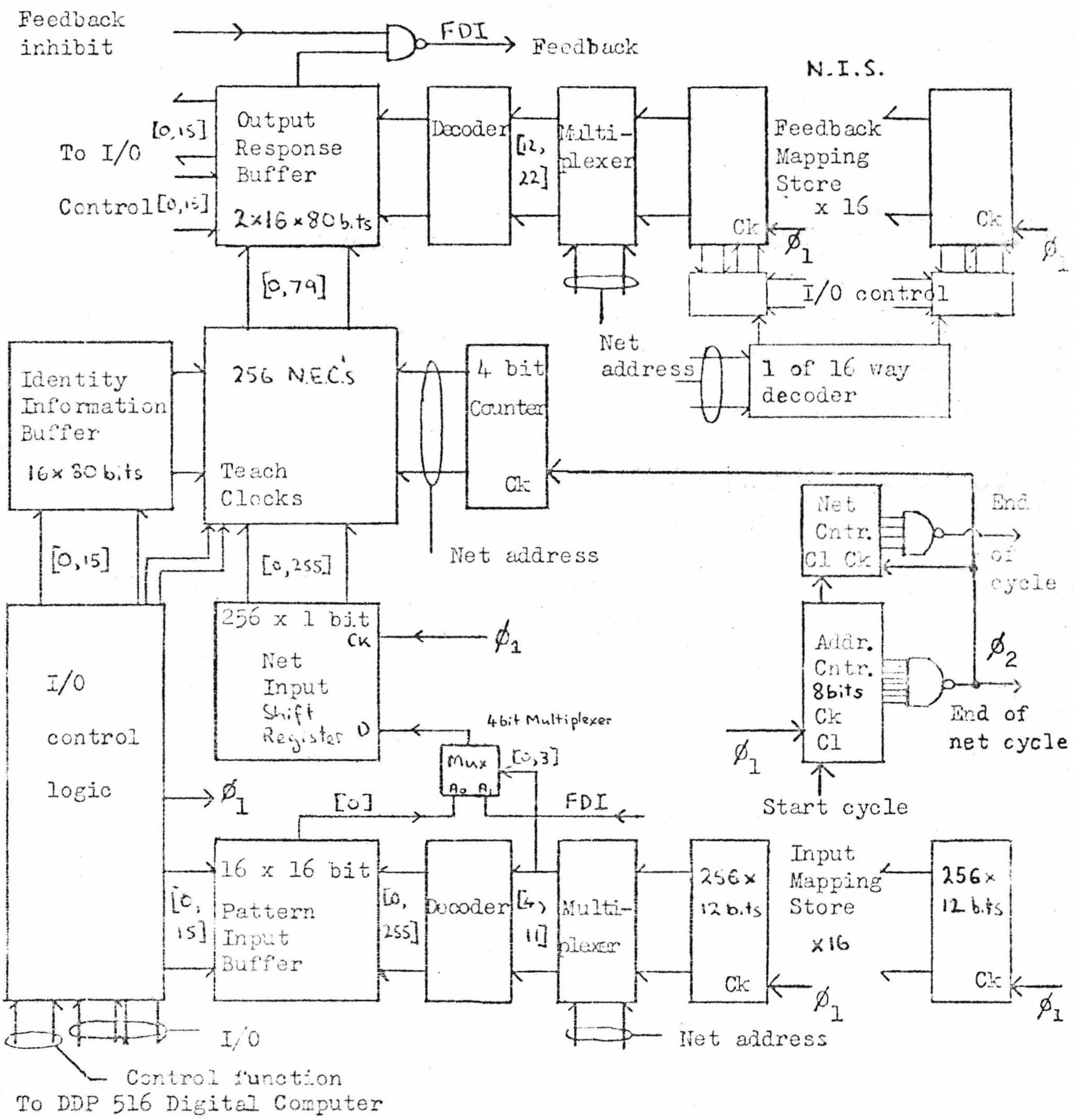


Figure 6.4 A Fixed Structure Sequential Fast Mapper

Input data is stored in a 16 word x 16 bit Pattern Input Buffer (PIB), to which data is written as 16 bit words but is accessed as single bit words. The serial input to the Net Input Shift Register is taken from the output of a universal logic circuit performing a network input function between two inputs, one, the Pattern Data Input (PDI) comes from the one bit output of the Pattern Input Buffer (PIB); the other, the Feedback Data Input, (FDI) comes from the one bit output of the Previous Output Response Buffer.

The Pattern Data Input is selected from the Pattern Input Buffer by an 8 bit field in a map element, likewise the Feedback Data Input by an 11 bit field and the Network Input Function by a 4 bit field. The map elements are stored in a memory of 4,096 words of 23 bits, labelled N.I.S. in Figure 6.4.

Control signals for the various memory and shift register clocks are derived from a 12 bit binary counter which is incremented by a free-running master clock signal. Each master clock cycle causes the fetching of the next interconnection and input function word from the Network Interconnection Store and the serial loading of the Net Input Shift Register. After 256 master clock cycles, determined by the control counter, a write order is given to the current Output Response Buffer or a NEC teach order generated according to cycle instruction information. The write address of ORB and the NEC address are also derived from the control counter. A Network Cycle requires 4,096 master clock cycles.

Network Cycles are initiated by a command from the host computer. During a Network Cycle no data transfers can take place between the Network buffers and the host computer. Since the master clock is free-running time may elapse after Network Cycle initiation until the zero state of the control counter is reached.

The system was designed with close regard to the types of integrated circuits available for its implementation which were to be selected from the standard ranges of TTL and MOS circuits. The Network Interconnection Store was to be implemented using 92 1,024 bit dynamic M.O.S. shift registers. These have a minimum clock frequency of 500Hz requiring continuous master clock operation. The Output Response Buffer, Pattern Input Buffer and Teach Data Buffer would require a total of 60, 16 word by 4 bit T.T.L. integrated circuit memories and the Network Input Shift Register would require 32, 8 bit serial-in, parallel-out shift register, T.T.L., integrated circuits. The total number of integrated circuits required would have been approximately 220 and the cost including power supplies, racks and circuit boards would have been about £1,200, according to prices in force during mid 1973.

The performance of the system would primarily be limited by the maximum shift frequency of the Network Interconnection

Store, which would be about 4 MHz. A possible improvement could be obtained by interleaving the store with several clock phases but this was not considered in this design owing to the practical complications caused by clock skews. The upper limit to overall systems performance is given by the data transfer rate with the host computer as discussed in Chapter 3.

During a Network Cycle the master clock would have to be inhibited for about 3 μS every 256 cycles for the network output to be loaded into the Output Response Buffer. A Network Cycle would take 1.07mS which is comparable to the time required to load the Pattern Input Buffer and send the Output Response Buffer via the host computer interface. There would have been an average latency time of 512 μS for randomly initialised Network Cycles due to the refreshing of the Network Interconnection Store.

When used to implement a single layer learning network of the sort described in Section 4.7.0 the response time would be on average 2.67 mS. for random presentation of input pattern but would be 3.19 mS for sequential presentation of input patterns.

6.2.1 A Serial/Parallel Mapping Technique

This section considers an alternative mapping technique that may be implemented in hardware. Unlike the sequential method the Serial/Parallel method (S.P method) does not require a fixed amount of storage or take a fixed time for a given number of interconnections but the storage and timing is determined by the nature of the interconnection.

The S.P. method consists of decomposing the interconnection mapping vector into a set of equivalent shift classes. Each shift class contains all the bits of a pattern that can be cyclically rotated by an equal number of positions in order to satisfy the interconnection mapping. The shift classes can be constructed from the interconnection mapping vector and the pattern to be mapped by first obtaining as equivalent shift vector to the interconnection mapping vector. This is accomplished by taking the difference between the value of an interconnection mapping vector element and its index. Since cyclic shifts are being considered, a negative value is equivalent to a complete rotation in the positive direction, minus the negative value, to give a shift vector of positive elements. The equivalent classes of elements present in the shift vector is then enumerated. These are then used to generate a set of binary mask patterns; the elements of the interconnection mapping are processed one by one and for every index of the domain element present as an element of the interconnection mapping vector the equivalent element of the mask for the shift class given by the shift vector is set to a one. The generation of shift classes and masks is illustrated in Figure 6.5. The set of masks is ordered corresponding to the magnitude of its shift class starting with the largest. A number equal to the difference between the value of its shift

Index	Map Vector	Shift Vector	Equivalence masks					
i	m	m-i	Class No: 7	6	5	4	3	1
1	2	1	0	0	0	0	0	1
2	5	3	0	0	0	0	1	0
3	7	4	0	0	0	1	0	0
4	1	7	1	0	0	0	0	0
5	8	3	0	0	0	0	1	0
6	3	7	1	0	0	0	0	0
7	10	3	0	0	0	0	1	0
8	9	1	0	0	0	0	0	1
9	4	5	0	0	1	0	0	0
10	6	6	0	1	0	0	0	0
		Shift Factor:	1	1	1	1	2	1

Figure 6.5 The Generation of Shift Classes and Masks.

class and the value of the shift class following is assigned to each mask. We call this value the shift factor of each mask.

The interconnection mapping is accomplished by the hardware of Figure 6.6. Elements from the domain vector are selected according to a mask pattern and 'OR'ed with the contents of a shift register. The shift register is then rotated by the value of the shift factor assigned to the mask and the process repeated for all the remaining masks. The resultant interconnection is held in the shift register.

In the worst case the S.P. mapper requires as many operating steps as the sequential method and has the disadvantage of requiring more memory for mask patterns than an equivalent mapping interconnection vector. However the occupancy of the masks in cases approaching this limit would be sparse enabling encoding of the element positions.

The S.P. mapper has its greatest likelihood of efficiency for interconnection mappings that are restricted to permutations of the domain vector and where the domain and range vectors are of similar order. Its greatest efficiency is where the mapping is simply a rotation by a given number of places. This would be of advantage in regular connected networks or ones where connections between elements are "diameter" limited as is apparent in some neural networks. The limitations of the S.P.

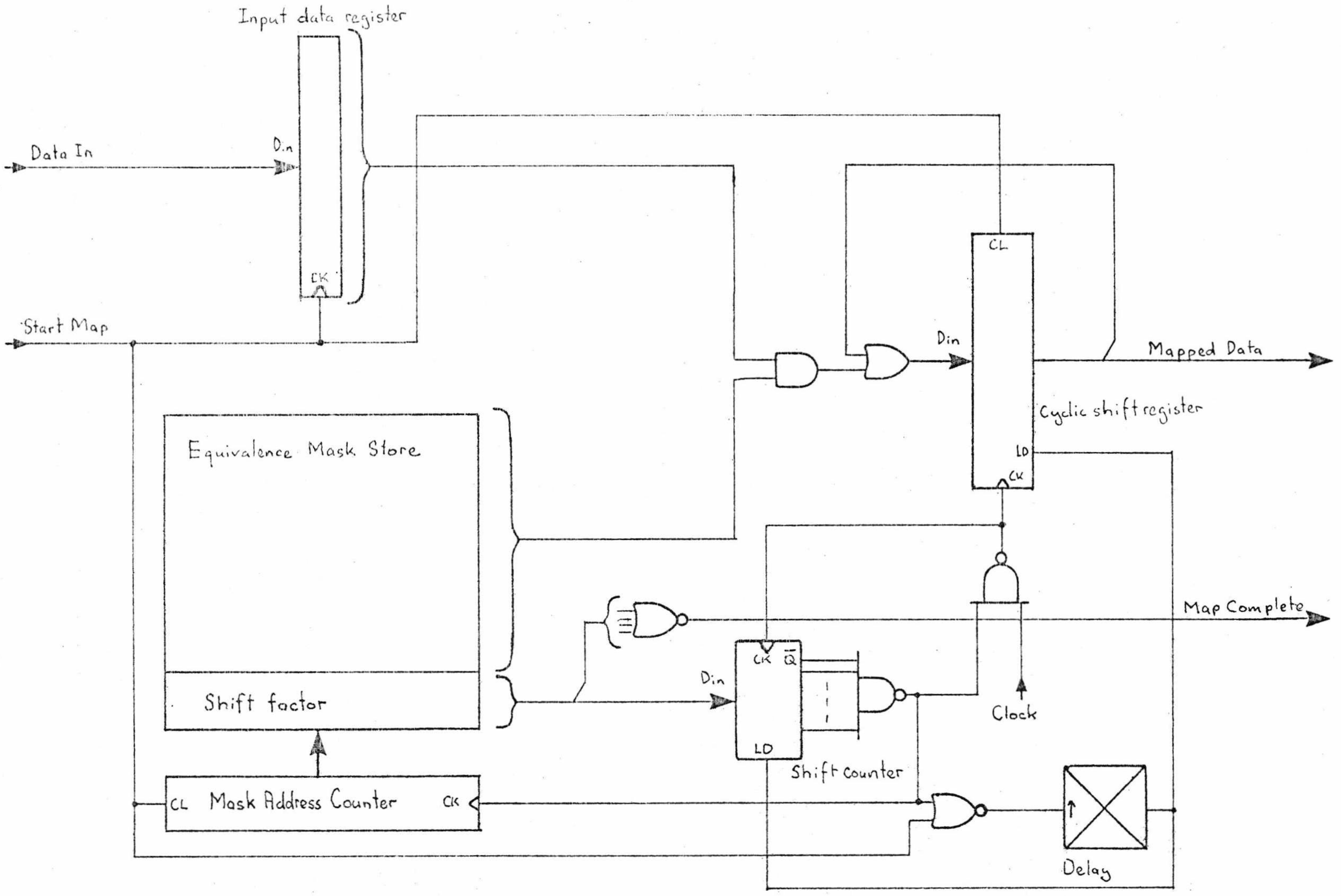


Figure 6.6 A Shift Register Mapper (Schematic)

mapper occur because all interconnections, in a hardware implementation, are in parallel, causing constructional problems and high cost. Where a conventional computer controlled network, similar to the Minerva system,is concerned only a limited improvement in performance can be achieved by parallelism in the network hardware. This means that the S.P. mapper would only exhibit better performance than the serial mapper of Section 6.2.1 in a dedicated network of parallel architecture in which limited flexibility of network structure is required.

6.3 A Processor for the Implementation of MINIC Language Features

Although detailed design had been carried out on the interconnection hardware described in Section 6.2.1 it became clear that it could not satisfy the specifications for network flexibility that were required by the MINIC language. In producing specification on the one hand for the MINIC language and on the other for the serial mapping hardware,different criteria had been considered. The MINIC language specification was produced by reduction of general network structures into their elemental components which could then be manipulated in a universal manner. The serial mapping hardware was specified from a synthesis of existing simulated network structures that had been found to possess interesting properties. Inherent in the nature of the latter specification is a limitation of novel network structures to combinations of existing ones.

When the semantics of MINIC had been decided a new specification for interconnection hardware was produced based upon a modular realisation of the elementary network operations chosen for the MINIC semantics.

Several technological innovations that became readily available during this period made a modular approach an attractive practical proposition. These were the introduction of TTL M.S.I. integrated circuits utilising Schottky barrier technology that are directly compatible with,yet four times faster than,standard TTL M.S.I. This enabled the implementation of low cost control circuitry that appeared transparent to data transfers between modules with regard to the execution time of control functions.

Another innovation was the introduction of high speed interface integrated circuits for interconnecting data lines between modules. This enabled the parallel operation of some modules by multiplexing data through common highways. The use of common control and data highways lend directly to the modular approach to design whereby specific primitive operations could be identified, and optimised hardware designed to execute the operation without reference to the hardware of other modules. This leads to some redundancy in the implementation which in the case of a "one off" product is outweighed by straightforward design procedures.

Control of data transfers between modules is performed using a micro-programmed controller to enable interpretation of MINIC semantics and to allow the incorporation of new modules at a latter stage of system development. The architecture and micro-instructions of the processor are presented in the following sub-sections.

6.3.0 Design Criteria for the Modular Micro-Programmable Processor.

The following design criteria are used to select a suitable architecture for the Modular Micro-Programmable processor, M.M.P.P..

a) Hardware Considerations:

1) The design should use standard, readily available M.S.I. and L.S.I. integrated circuits where practicable;

2) It should require little or no modification of the existing network hardware;

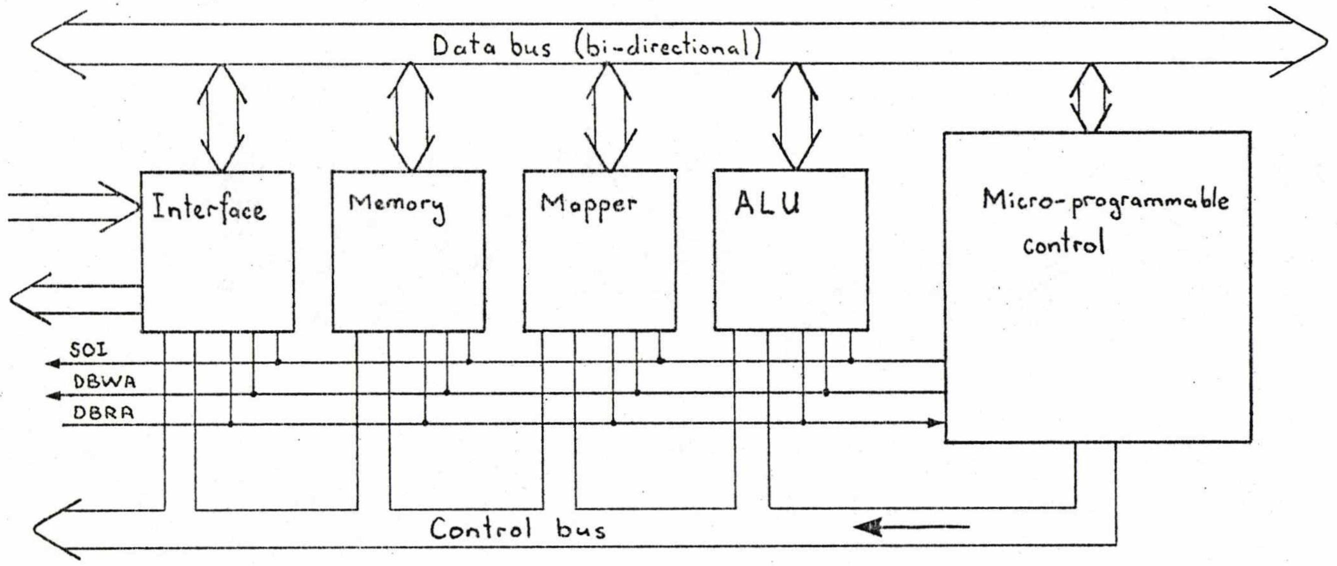
3) It should have a writable control store to allow for continuing hardware and software developments;

4) The speed of the mapping-interconnection operation should be approximately equivalent to that of the serial mapping hardware of Section 6.2.1.

b) <u>Software Considerations</u>:

The operator precedence grammar of MINIC expressions naturally suggests that data is held in a register or registers and that some transfer of data takes place between registers, during which a transformation of the data occurs. This implies that an architecture suitable for implementation of MINIC semantics is a vertical one in which various storage elements can be interconnected via logic units <u>one</u> at a time.

The architecture chosen in the light of the above requirements is a modular bus-oriented one. A general bus orientated processor is shown in Figure 6.7. There is one control module and a number of data-processing modules. Two information highways are used to link processing modules; one carries data, (D BUS), and the other control information, (C BUS). On receipt of the appropriate addressing command from C BUS a data-processing module assumes control of D BUS and places data on D BUS which then is received by another data-processing module specified by C BUS. Processing actions are carried out on the data by the internal logic of the module <u>after</u> the data has been received. Processing actions are made up of primitive operations required for efficient execution of the MINIC language and housekeeping operations such as the storage of data in buffer memory and program loop control.



SOI -	Start of Instruction
DBWA -	Data bus write acknowledge
DBRA -	Data bus read acknowledge

Figure 6.7 A Bus Orientated Processor

The advantages of a bus structured architecture are in the areas of flexibility and ease of system design that occur because of the standard interface each module has with the D BUS and C BUS. This enables the logic design of each module to be treated independently. Modifications to a particular module are restricted to having a local effect and the logic design is simplified since each module has similar control logic based on a skeleton control logic design.

The design of the bus interfacing protocol and skeleton control logic is presented in the following section. (To simplify interfacing with the host computer and the existing network hardware both D BUS and C BUS are chosen to be 16 bits wide).

The main disadvantage with the single bus architecture is that both data and the address information about the data must be transfered via the source data highway causing a loss of parallelism with a corresponding loss of speed. This is partially overcome by the use of very high speed bus interface circuits and by overlapping the asynchronous operation of the processing modules.

6.3.1 Bus Interfacing and Skeleton Control Logic

The control logic of each module has three important functions: 1) It must decode instructions from C BUS and activate the module logic when a feature of the module is requested ; 2) It must be aware of the internal state of readiness of the module to transmit or receive data ; 3) It must inform other modules and the C-module of the successful receipt or transmission of data.

Information about data transfers is transmitted to all the modules by means of three protocol signals, Start of Instruction (SOI), D BUS Write Acknowledge, (DBWA), and DBUS Read Acknowledge, (DBRA). SOI is generated by the C-module each time a new instruction is placed on C BUS. It is reset by DBRA. DBWA is generated by the transmitting module a short delay ($\sim$20nS) after data is gated onto D BUS. The receiving module uses DBWA to gate the data from D BUS into its internal logic and simultaneously generates DBRA to inform the C-module that the transfer has been successfully completed.

All D BUS transfers take place asynchronously. Every effort is made in the design of the module logic to ensure that the time a module is unable to receive data is a minimum. This enables, for example, the initiation of a memory read cycle on transfering data to the memory address register of a memory module and while the memory cycle is in operation an arithmetic module could update a control loop counter.

An example of a typical sequence of D BUS protocol signals is given in Figure 6.8.

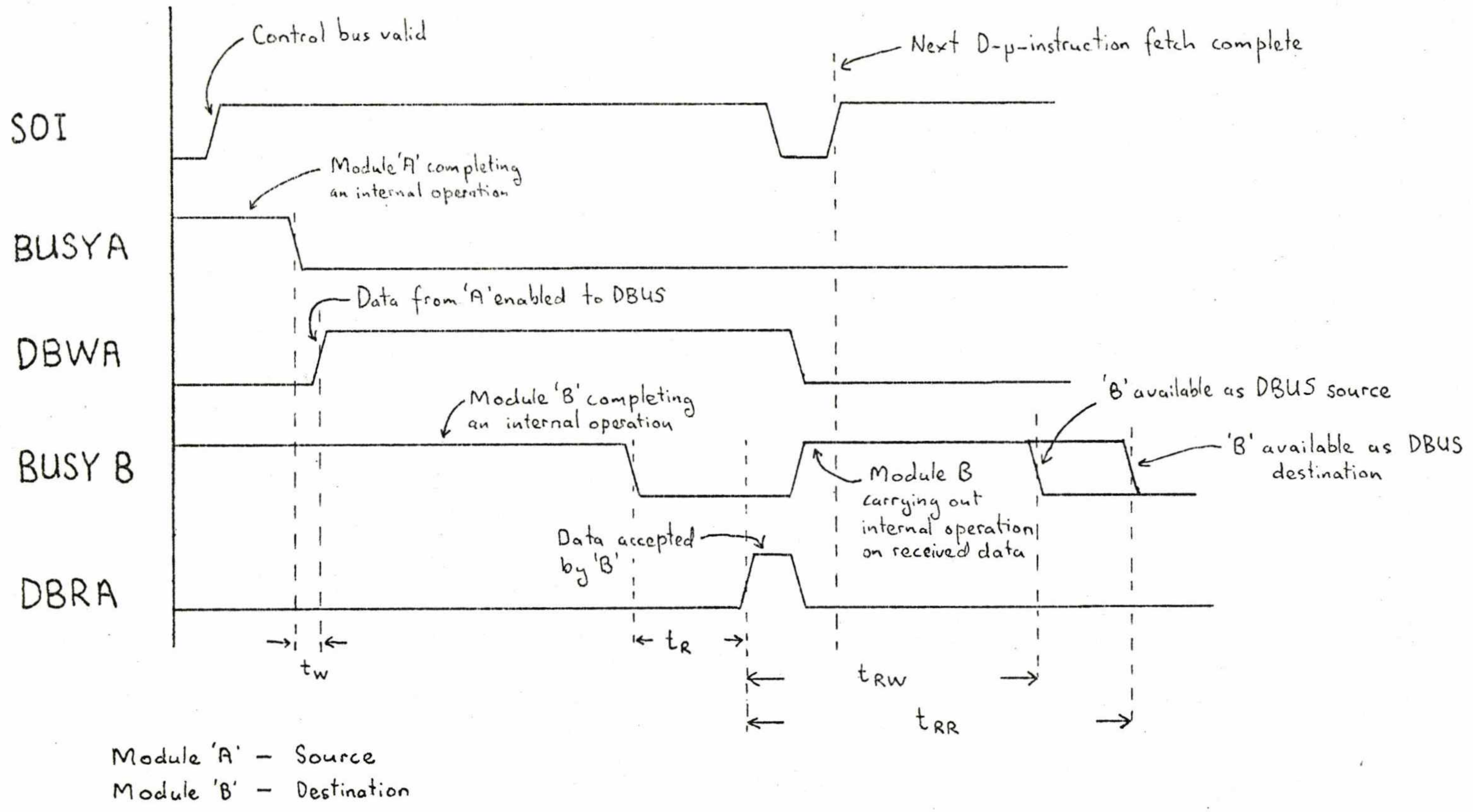

SOI
BUSYA
DBWA
BUSY B
DBRA
Control bus valid
Next D-μ-instruction fetch complete
Module 'A' completing an internal operation
Data from 'A' enabled to DBUS
Module 'B' completing an internal operation
'B' available as DBUS source
'B' available as DBUS destination
Module B carrying out internal operation on received data
Data accepted by 'B'
t_W
t_R
t_{RW}
t_{RR}
Module 'A' – Source
Module 'B' – Destination

Figure 6.8 DBUS Protocol Signals

Instructions placed on C BUS take the form of vertical micro-instructions consisting of three highly encoded fields. The first field, consisting of 6 bits, is called the Function field and is available to control the internal logic of a transmitting or receiving module. The remaining two fields are of 5 bits each and are used to select the source and destination modules.

Skeleton module control logic for data-processing modules is shown in Figures 6.9 and 6.10. C BUS decoding has to be performed by the fastest method possible to avoid any dead time between data transfers and in acknowledging data transmission and reception. The Function field and source or destination field as appropriate are decoded by two levels of Schottky barrier TTL logic with a typical propagation delay of 8nS. The requirement for a third layer is avoided by providing C BUS and its complement to all modules, C BUS thus requires 32 signal lines. The theoretical minimum time for a data transfer cycle using the devices labelled in Figures 6.9 and 6.10 is approximately 80 nS.

The time taken to perform data transfers is composed of four parameters; t_W, t_R, t_{RW} and t_{RR} defined as follows:

a) t_W - this is the time between SOI being asserted and DBWA being generated assuming that the module is not busy on receipt of SOI. This is a minimum of 30nS for the skeleton module control logic;

b) t_R - this is the time between the module sensing the assertion of DBWA and the generation of DBRA. t_R represents the time required to gate data into the internal logic of the module. As in (a) above the module is assumed to be not busy. It has a minimum value of 30nS;

c) t_{RW} - this is the time a module is unavailable for sending data after it has acknowledged its use as a destination module, an example being memory access time;

d) t_{RR} - this is the time a module is unavailable for use between successive destination references, an example being memory cycle time.

(Where the internal operation of the module is asynchronous the above parameters are given as maximum and average values.)

A speed improvement of about 50% could be achieved using E.C.L. integrated circuits but this would have two disadvantages. 1) There are fewer standard M.S.I. circuits for implementation of module logic in manufacturers' E.C.L. product ranges. 2) There are restraints on the physical layouts of systems when using E.C.L. because accurate impedance matching of interconnections is necessary.

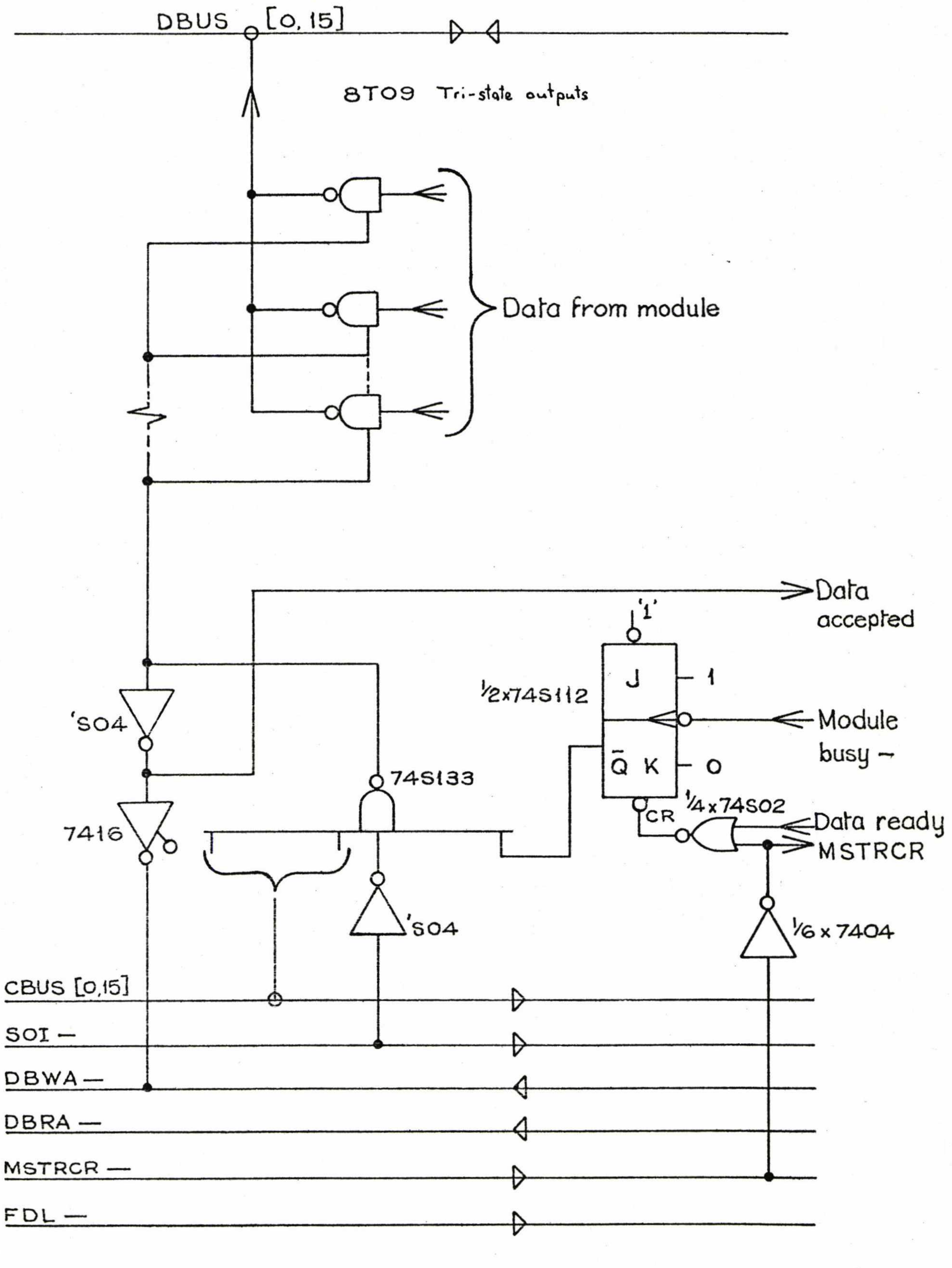


Figure 6.9 DBUS Write Control Logic

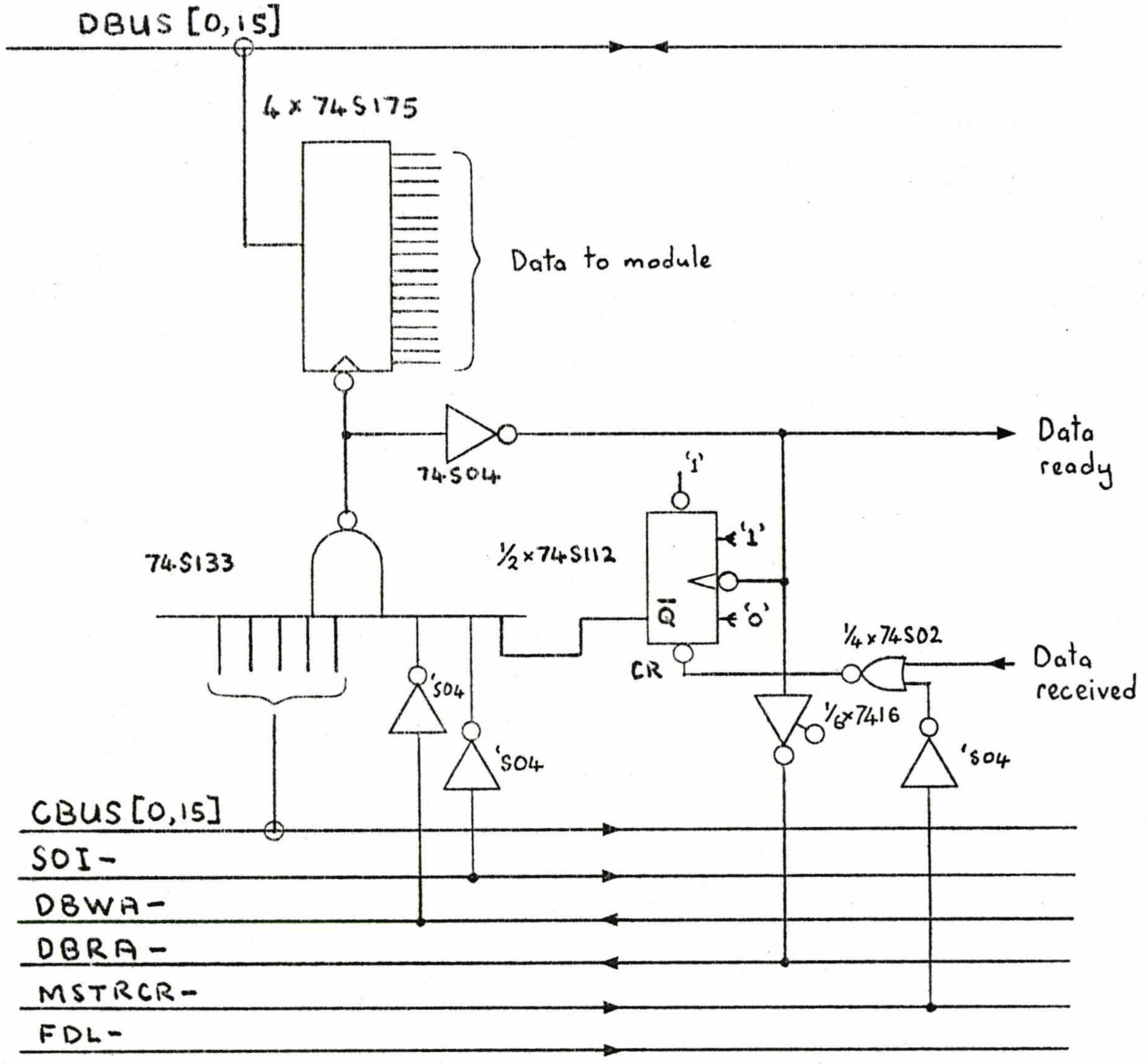


Figure 6.10 DBUS Read Control Logic

6.3.1.0 Other Module Interconnections

As well as the general information pathways C BUS and D BUS certain internal states of modules such as error conditions, carry bits, etc. are made available to all other modules and are used specifically by the C-module to enable conditional instructions. Sixteen interconnections have been allocated for this purpose and are referred to as the T BUS.

In order to facilitate initial program loading two control signals are provided. One is used to return the internal logic of all the modules to an initial state and is called masterclear, (MSTRCR), the other is called force data load, (FDL), and is used to force data from D BUS into the input registers of certain pre-wired modules irrespective of SOI and the other bus protocol signals. FDL is generated by external operator action when programs are being manually entered into the C-module control store.

6.3.1.1 Implementation of BUS Interconnections

In view of the high speeds obtainable by the D BUS protocol logic, careful design of the interconnection hardware is necessary. The total number of common connections between all modules excluding supply connections is 69 made up of: C BUS, (32); D BUS, (16); SOI-; DBWA-; DBRA-; MSTRCR- and FDL-.

All inter-module connections are made via a printed-circuit mother board shown in the photograph of Figure 6.11. This has provision for 142 parallel signal inter-connections which we call the General Bus.

The impedance of the signal tracks is controlled to be 100Ω by providing a ground plane on the reverse side of the mother board and by selection of the track widths and edge connector combination. Connection from the mother board to printed circuit boards containing modules is via a double-sided 76 way edge connector. Only one side of the edge connector is used for the common intermodule connections. The remaining side is used for modules that require more space than is provided on a single printed circuit card such as the Dynamic M.O.S. memory controller module described at Section 6.5.3. Four tracks each are allocated to ground return and power supply inter-connections. The printed circuit tracks for power supply connections are supplemented by copper bus strips to provide a current carrying capacity of 20 Amperes.

6.4 The Control-Module

The Control-Module (C-module) provides sequences of data micro-instructions (D-μinstructions) and initiates their execution (using S.O.I.) by the D BUS interconnected data processing modules (D-modules). In order to take advantage of the high speed nature of D BUS transactions the cycle time between successive D-μinstructions must

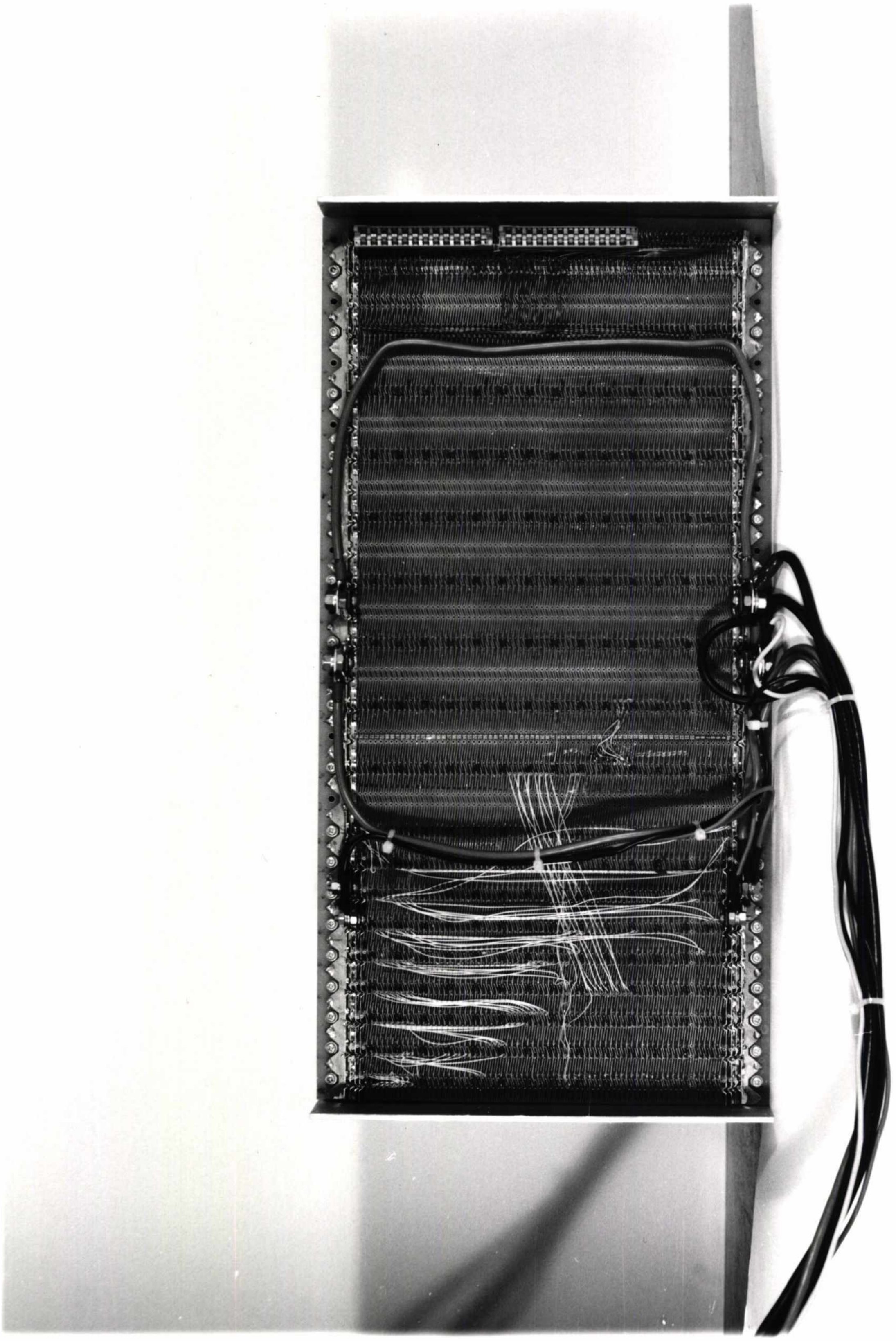


Figure 6.11
The General Bus Interconnections

be of a comparable speed. This can only be achieved by using a high speed memory for micro-instructions and a control architecture that enables overlap between next micro-instruction address calculation and micro-instruction fetch phases.

In order to explain the design of the C-module the structure of the micro-instruction set is presented in the following sub-section.

6.4.0 The Micro-Instruction Set

The micro-instruction format of the modular processor consists of highly-formulated 16 bit words. They are split up in an hierarchical manner starting with two groups: data-micro-instructions (D-μinstructions) and control micro-instructions (C-μ instructions). D-μ instructions have been briefly introduced in Section 6.3.1 and are concerned with data manipulations involving D-modules and D BUS. They are further sub-divided into three types: 1) A.L.U. micro-instructions, 2) MOVe micro-instructions and 3) Monadic micro-instructions.

There are 32 possible A.L.U. micro-instructions which correspond to the arithmetic and logical functions obtainable by the universal Arithmetic-Logic Unit integrated circuit type SN74181N described in Texas (1974). These are performed by a dedicated D-module that is fully described in Section 6.5.0. In the case of A.L.U. micro-instructions the data source field has only local relevance to the A.L.U., D-module. The destination field has global relevance and may be used to specify any other D-module destination. The function field is used to specify the A.L.U. function; the most signigicant bit of the function field is a zero for all A.L.U. micro-instructions.

6.4.0.0 MOVe Micro-Instructions

There are 8 possible types of MOV(e) micro-instruction. In these micro-instructions both source and destination fields have global relevance. As their name implies they are normally used for transferring information between D-modules although the increment, (INC), micro-instruction is included in this category. The increment micro-instruction enables the addition of one to certain D-module sources and destinations which are implemented by dual purpose binary counter/storage register integrated circuits. Since both source and destination fields are active two such registers may be incremented simultaneously without the need to transfer data to the A.L.U. D-module. No data transfer takes place during the INC micro-instruction. The most significant bits of the function field in a MOV micro-instruction are 110 or 6_8. Other bits in the function field are used to specify actions by particular D-modules such as memory cycle initiation.

6.4.0.1 Monadic Micro-Instructions

These are a restricted class of micro-instructions which apply to dedicated D-modules that are selected by the function field. Such micro-instructions are used for shifting data, performing interconnection mappings, and generating literal constants and control orders. The monadic micro-instructions are so called because the operations performed are closely related to single specific module functions rather than some general types of operation which are valid for data transfers between different D-modules. They are distinguished by having the 3 most significant bits of the function field set to 1, i.e. 7_8. Only one operand, which may be a source or destination module, is specified in the micro-instruction. The remaining bits are used as a literal or control constant.

6.4.0.2 C-μ instructions

The remainder of the micro-instruction set are local to the C-module and are called C-μ instructions. They are used for controlling the sequence of micro-order execution. Unlike D-μ instructions, the C-μ instructions are formulated into two equal fields of 8 bits each. The most significant field is called the micro-operation code and the remaining field is called the address modifier field. The micro-operation codes are identified by nmemonics similar to those used for assembly languages. A description of these and the various micro-instruction word formats is given in Appendix H. The action of the micro-operation codes is as follows:

BCT - Branch Conditional True performs a branch to the <u>relative</u> address given in the address modifier field if a one is present on a specified line of the T BUS;

BCF - Branch Conditional False performs a branch to the <u>relative</u> address given in the address modifier field;

(P)BRA - Branch to absolute address given by the address modifier field;

(P)BRR - Branch to the address given by the sum of the current address and the address modifier field. (The most signigicant bit of the address modifier signifies the sign);

(P)BRAI- Branch to the absolute address given by the micro-index register;

(P)BRRI- Branch to the address given by the sum of the current address and the contents of the micro-index register;

GCO - Generate Control Order. The address modifier field is used as a control literal for changing internal states of the C-module;

JS(n) - Jump to Subroutine. This saves the current address plus one on the micro-subroutine return address stack and performs a branch specified by a post-fix (A,R,IA,IR) similar to unconditional branch C-μ-instructions;

RTN - Return. This obtains the address of the next micro-instruction from the top of the micro-subroutine return address stack;

WRT(n) - Write. This causes the contents of the control store data register to be written at the address calculated according to the address modifier field and the post fix (A,R,AI,RI) as for Branch micro-instructions above.

The unconditional branch C-μ instructions may be modified by one of the suffixes P or W. The suffix P, (POP) causes the top element of a micro-subroutine return address stack to be discarded concurrently with the execution of the branch and the suffix W (WAIT) delays the execution of the branch until the current D-micro instruction has been completed.

An extensive repetoir of C-μ instructions is provided in order to help overcome the limitations on the size of the writeable control store imposed for financial reasons. This strategy naturally diverges from that of a purely vertically micro programmed architecture to one of mixed strategy in which several fields of highly encoded information are used in the micro-instruction.

6.4.1 C-module Architecture

The C-module architecture used to realise the C-μ instruction set is shown at Figure 6.12. Provision is made for a writable control store of 4,096 words of 16 bits by arranging the micro-instruction address logic to be 12 bits wide. At present 512 words of writable control store, (W.C.S.), are implemented. The W.C.S. is arranged in pages of 256 words. Page boundaries may be crossed by relative, indexed or incremental addressing but <u>all</u> absolute addressing refers to the lowest sector.

Incremental addressing takes place when the action of the micro-instruction and the C-μ instructions WRT and GCO. While the next μ instruction fetch is taking place from the W.C.S. the contents of the Control Store Address Register, (C.S.A.R.), are fed-back to the inputs of a 12 bit adder. During this time the control order RELAD is enabled, MODADR disabled and CIN is set to one.

The output of the 12-bit adder is the current contents of CSAR plus one. This is placed on the inputs to CSAR and the Control Store Address Save Register, CSASR. Once the micro-instruction fetch has been accomplished micro-orders to load CSIR, (LDCSIR) and CSASR, (LDCSASR) are generated. CSASR is required for execution of the WRT and JS micro-instructions. The micro-order to decode the micro-instruction is then given.

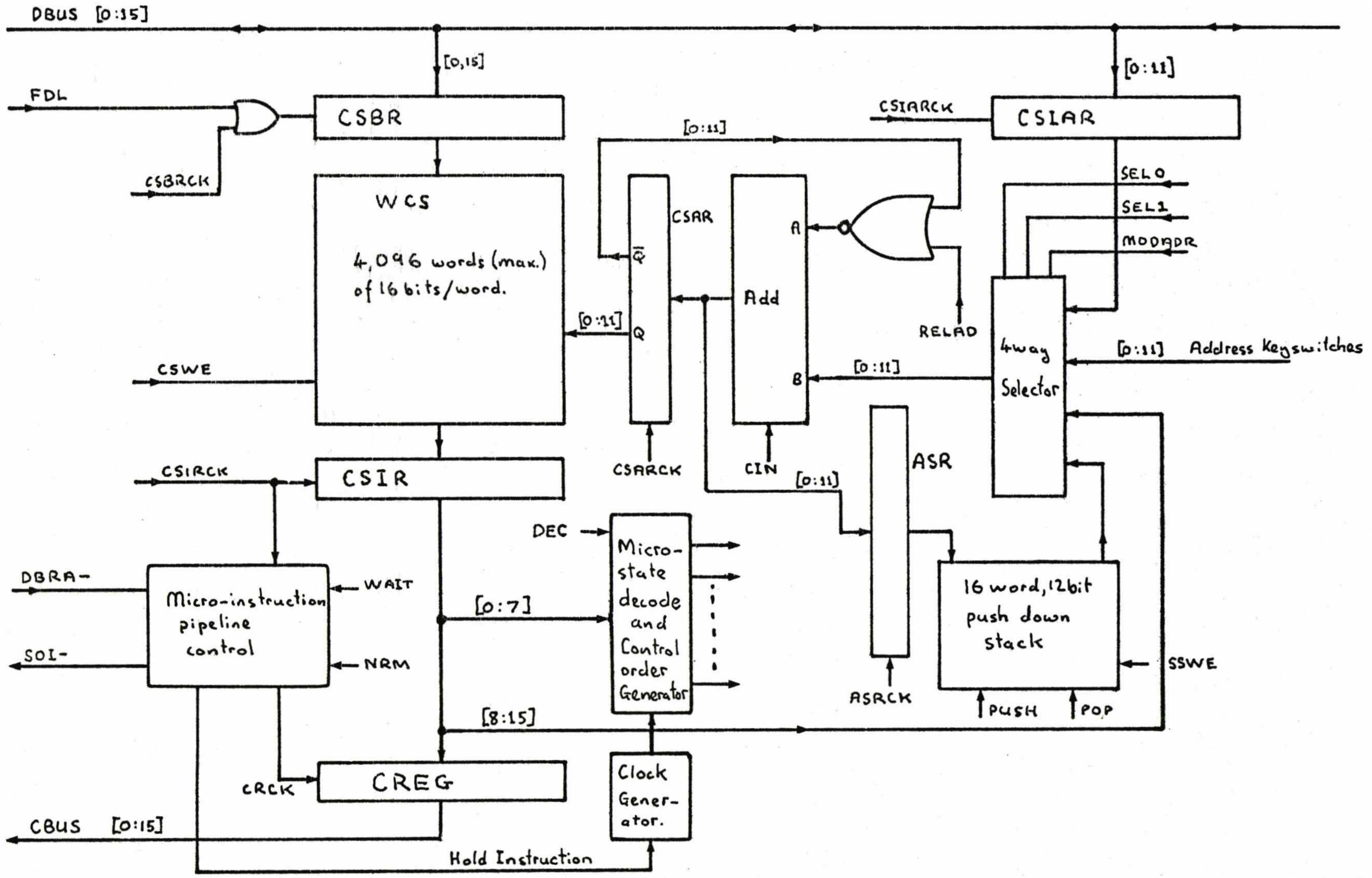


Figure 6.12 C-module Architecture

A similar but longer cycle takes place if the micro-instruction modifies the micro-program address. On completion of the micro-instruction fetch the decode micro-order is given and appropriate orders are then generated to select the address data origin according to the index flag and the addressing mode. CIN is reset, and after a suitable delay to enable the address to stabilise, CSAR is clocked to commence the fetch cycle of the next micro-instruction. Relative addresses are in 8 bit two's complement format; the sign bit is extended to 12 bits at the address selector circuits.

A third type of cycle is necessary for the WRT micro-instruction. This proceeds as for a modified address micro-instruction cycle except CSASR is transfered to the return address stack while the WCS address to be loaded is calculated. Once CSAR has been clocked a write cycle is caused. The WRT micro-instruction cycle is completed by popping the incremental address of the next micro-instruction from the subroutine stack and placing it in CSAR to initiate the next micro-instruction fetch.

Micro-instructions are decoded into micro-states which correspond to the generic families of micro-instructions. Thus the micro-states are: DINS - D-μ instruction, BC - branch conditionally; BR - branch unconditionally; WBR - wait and branch unconditionally; PBR, - pop stack and branch unconditionally; JS - jump to subroutine; RTN - return; WRT (1 and 2) - write in WCS (for timing requirements there are two consecutive micro-states) and GCO - generate control order. Micro-states are encoded as an orthogonal binary code in order to simplify the design of the control order logic.

6.4.1.0 Manual Operation of the C-module

In order to facilitate initial program loading of micro-instructions key switches are provided on the C-module front panel. The C-module may be operated in one of 3 modes; RUN, single instruction, (SI) and examine, (EXM), selected by a three position key switch. Twelve key switches are used to enter data into the micro-instruction address register, CSAR, on receipt of a signal from a LOAD Key switch. A continue, (CONT), Key switch is provided to step the micro-instruction address register, (CSAR), by one. With the mode switch in the S.I. position the CONT Key switch enables single step execution of a micro-program and in the EXM position the contents of successive locations of the WCS may be displayed on 16 L.E.D.'s. Also when in the EXM, position micro-instruction data from the control store buffer register, (CSBR), may be written in the WCS at the current address by means of a WRITE keyswitch. The CSBR can be loaded from D BUS using the FDL- (Force Data Load) signal. This is normally generated by the BUS display and keyswitch D-module to enable the forced entry of an initial micro-program as described in Appendix H 1.3.

6.4.1.1 C-μ instruction Implementation

The control orders to implement the C-μ instruction set using the C-module of Figure 6.12 are designed using conventional techniques involving hardwired logic. The actual design procedure is outlined below. Minor timing points and delays are adjusted by selecting the logic family used for the decoding logic. A wide range of gate propagation delays are available by selection from the compatible 74 series TTL, these vary from 3nS for Schottky 74S series, 6nS for high speed 74H series, 12nS. for standard 74 series to 30nS. for low power 74L series.

The timing orders are derived from a master clock whose frequency is determined by a consideration of the number of timing levels required to execute the micro-instruction fetch and execute phases. The number of timing levels corresponds to the highest common factor of the device propagation delays around the micro-instruction fetch and next address calculation loop.

These are as follows:

(A)	TPDAR	-	propagation delay from CSARCK	11nS.
	TWCSA	-	read access time of W.C.S.	30nS.
	TSIR	-	set-up time of CSIR	8nS.
(B)	TPDIR	-	propagation delay from CSIRCK	11nS.
	TPDDEC	-	micro-state decode propagation delay	8nS.

	TSSR	-	set-up time micro-state register	10nS.
(C)	TPDSR	-	propagation delay of micro-state register	7nS.
	TPDMODADR	-	propagation delay to MODADR	6nS.
	TPDSE	-	propagation delay from MODADR to selector output	12nS.
	TADD	-	adder propagation delay	27nS.
	TSAR	-	set-up time of CSAR	8nS.

The total micro-instruction cycle requires 138nS..

The clock period is determined by the highest common factor of the times between the major timing orders (A), (B), and (C) above. These are: A→B, 49nS.; B→C, 29nS.; and C→A, 60nS.. A near perfect solution could be obtained by a clock period of 10nS which would mean a clock frequency of 100 MHz. However this is above the guaranteed maximum clock frequency of Schottky TTL devices which is about 70 MHz. A practical compromise is obtained by choosing a clock period of 17nS. which gives a clock frequency of 59 MHz, well within the capabilities of Schottky TTL. In order to complete the modified address micro-instruction cycle shown above 8 clock periods are required. The clock generator is designed to produce a cycle of 8 major timing levels which can be modified to a cycle of 5 major timing levels for D-μ instructions or two complete cycles for a WRT micro-instruction.

6.4.1.2 Timing Level Generation

The clock generation circuit is capable of generating variable length cycles of timing levels and also of being halted at particular cycles. Halting the clock generator is necessary to enable single step micro-instruction execution and to cause waiting until D BUS has completed an existing D-μ instruction when a following D-μ instruction has been fetched from the control store.

A shift register integrated circuit type SN74S194N is chosen to implement the clock generator. The clock input is provided by a free-running clock signal of 59 MHz. A twisted-ring counter configuration is used because timing orders of any number of timing levels may be obtained with decoding gates of no more than 2 inputs, also the Hamming distance between successive clock states being one enables hazard free timing order decoding. The circuit diagram of the clock generator is shown in Figure 6.19P.

The clock generator uses the parallel load facility of the SN74S194 integrated circuit to change the clock cycle length between 5 cycles for D-μ instructions and 8 cycles for C-μ instructions.

The synchronous halt facility is used to suspend the timing level on three possible conditions: 1) during an indexed C-μ instruction if the C-module index register (CMIR) is busy due to a D-BUS transfer ; 2) if CREG contains an active D-μ instruction and CSIR contain a D-μ instruction or a WBR instruction; 3) if the operators "examine/single-instruction/run" Keyswitch is not in the"run"position. The clock generator halt is freed for one cycle after a pulse from the operators "continue" Keyswitch.

The clear facility is used in conjunction with the examine/single-instruction mode of operation to enable the loading CSAR from the operator "address" key switches. A "load address" Keyswitch is provided to generate control orders which select the address loop adder inputs from the "address" keys and set up the adder for an absolute address. The clock generator shift register is cleared which releases the clock generator to perform only the address calculation segment of a micro-instruction cycle before halting.

6.4.1.3 Generation of the Control-Orders

The synthesis of the logic functions for generating the control orders is done by producing a timing matrix the rows of which correspond to the control orders and the columns of which correspond to the timing levels, for each micro-state. Entries in the timing matrices correspond to the required value of the control order given by the row at the timing point given by the column,"don't care" values are also included. Boolean equations for each control order are obtained by taking the "and" of the rows and columns and forming the "or" of all terms corresponding to each row. The equations are minimised by inspection and graphical methods such as presented in Aleksander

(1970). Timing order matrices are given in Figures 6.13 - 6.18.

The circuit diagram of the logic net used for control-order generation is shown at Figure 6.20P.

6.4.1.4 The Micro-Instruction Pipeline

Those C-μ instructions that are concerned exclusively with the flow of the micro-program can be executed concurrently with the execution of a D-μ instruction. In order to implement this, micro-instructions fetched from the W.C.S. are fed into a micro-instruction decoding pipeline of 2 levels comprising the CSIR and CREG registers of Figure 6.12. Decoding of micro-instructions takes place immediately after their loading into CSIR. The circuit diagram of the micro-instruction pipeline control logic is shown in Figure 6.21P.

The pipeline also serves the purpose of buffering between the synchronous source of micro-instructions and an asynchronous sink provided by the D-modules. This enables D-μ instructions that are completed faster than the micro-instruction fetch cycle to be performed at their maximum repetition rate when they are interleaved with slower D-μ instructions, subject to the limitation that the average D-μ instruction execution time is greater than or equal to the average micro-instruction fetch and C-μ instruction execution times.

Clock state

Control Order

	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
MODADR	1	1	1	1	1	C-	C-	C-	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	1	1	1	1	1
SEL0	d	d	d	d	d	0	0	0	d	d	d	d	d	0	0	0	d	d	d	d	d	0	0	0	d	d	d	d	d
SEL1	d	d	d	d	d	0	0	0	d	d	d	d	d	I	I	I	d	d	d	d	d	I	I	I	d	d	d	d	d
CSARCK	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
ASRCK	0	0	0	0	0	d	0	0	0	0	0	0	0	d	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
CSIRCK	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
RELAD	1	1	1	1	1	1	1	1	1	1	1	1	1	R-	R-	R-	1	1	1	1	1	R-	R-	R-	1	1	1	1	1
CIN	1	1	1	1	1	C-	C-	C-	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	1	1	1	1	1
DEC	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
PUSH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
POP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SSWE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0
CSWE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	D-type					BC								BR								JS							

d- don't care
C – condition true
I - indexed
R- relative

Figure 6.13 to 6.16 Timing Order Matrices

Clock state

Control Order

					0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
MODADR					0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1
SEL0					1	1	1	d	d	d	d	d	d	d	d	0	0	0	0	1	1	1	1	d	d	d	d	d
SEL1					1	1	1	d	d	d	d	d	d	d	d	I	I	I	I	1	1	1	1	d	d	d	d	d
CSARCK					0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0
ASRCK					d	0	0	0	0	0	0	0	d	0	0	0	0	0	0	0	d	0	0	0	0	0	0	0
CSIRCK					0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
RELAD					0	0	0	1	1	1	1	1	1	1	1	R-	R-	R-	R-	0	0	0	0	1	1	1	1	1
CIN					0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1
DEC					0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
PUSH					0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
POP					0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
SSWE					0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
CSWE					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0
					RTN								WRT															

Figure 6.17 and 6.18 Timing Order Matrices

Operation of the micro-instruction pipeline circuit is as follows: Register CSIR is loaded with a micro-instruction from WCS at timing level T6; after decoding the micro-instruction, flip-flop Hg(l) is set on the rising edge of clock shift register SRl (see Figure 6.21) when CSIR contains a D-μ instruction. No further action is taken in the case of a C-μ instruction. Signal NINH-, generated by Hg(a),causes the clock generator to halt at timing level, T5. Flip-flop Hg(b) contains information about the state of the D-μ instruction in CREG. When Hg(b) is set the D-μ instruction is being executed by the D-modules. It is reset by the leading edge of the D BUS read acknowledge signal, DBRA- or the Examine mode signal, EXM-, derived from the operators key switches. When Hg(a) is set and Hg(b) is reset the order CREGCK is generated to load CREG from CSIR. This causes the resetting of Hg(a) and the setting of Hg(b). The Start of Instruction signal, SOI-, is derived from Hg(b) after passing through a cascade of gates to provide a delay while the C BUS signal lines are settling.

6.4.1.5 Decoding of C-micro-instructions

Immediately a micro-instruction is placed in CSIR its function field is decoded into one of eight discrete micro-states. The circuitry to perform this is straightforward and is given in Figure 6.22 P. When the mode control keyswitch is in the EXM position the micro-state decoder is forced into the DINST (D-μ instruction) state. The micro-

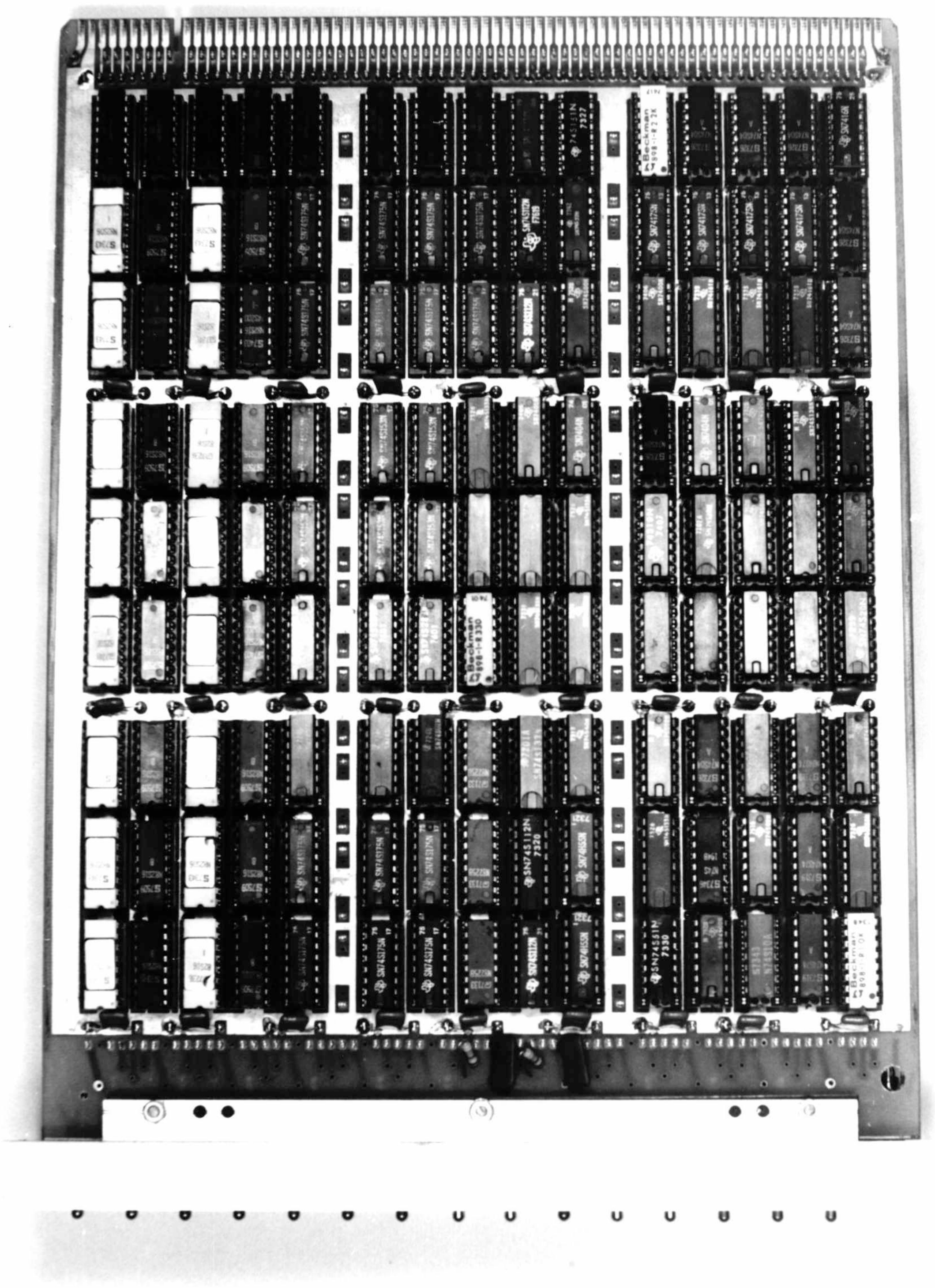


Figure 6.19a
The C-module

instruction pipeline operation is also inhibited in this mode and so micro-instructions may be examined using the continue keyswitch without any action being taken either by the C-module or any D-modules.

6.5 D BUS Processing Modules

The D BUS processing modules perform the intrinsic data processing operations of the Modular Processor. These operations are split among three D-modules called the Arithmetic Logic Unit (A.L.U.) module, the Shifter and Bit Counter module and the Mapper Module.

The A.L.U. module has been designed to perform the Boolean operators of the MINIC language and affords some optimisation by directly encoding any complement (NT) of the operands in a binary operation as a single micro-instruction. Arithmetic facilities are used for those control functions and address transformations that are invisible to the MINIC programmer. MINIC arithmetic expressions are intended to be evaluated by the host computer system and passed as arguments, where necessary, to the Modular Processor. In designing the A.L.U. module, however, the interactive mapping function extensions of MINIC, presented in Section 4.9 have been taken into consideration.

The Shifter and Bit Counter Module provides facilities for arithmetic shifts, logical shifts and rotation by a specified number of places. Together with the A.L.U.

module micro-programs for multiplication and division may be realised. During a shift micro-instruction a count is kept of any overflow or underflow bits and these are used to implement directly the SUM <pattern expression> operation of MINIC.

The Mapper Module performs the bit accessing operation required by the sequential mapping alogrithm and the assembling of the result into 16 bit words. A monadic micro-instruction is provided to perform a 1 bit mapping and a micro-program subroutine to execute the 16 bit partial mapping as performed by the routine MPSB of the MINIC Run-time Library (c.f. Appendix D2) requires 17 locations. A discussion of micro-programming MINIC features is to be found in Section 6.7.

The operation of the D BUS processing modules is presented in the following sub-sections.

6.5.0 <u>The Arithmetic and Logic Unit D-module</u>

The A.L.U. D-module is based on the SN74181 universal arithmetic/logic unit integrated circuits produced by Texas Instruments (1974). Arithmetic is performed using two's complement representation on parallel 16 bit signed operands. After an A.L.U. micro-instruction is performed a four bit condition register is set according to the result. These bits are labelled C, O, Z and S. Condition C is the carry out from the most significant

A.L.U. i.c. and can be used when unsigned arithmetic is required or for performing multiple precision arithmetic. Condition O is derived by comparison of the signs of the input operands and the sign of the result. It represents an overflow or underflow during a signed arithmetic operation. Condition Z is set when the result of an operation, logical or arithmetic is zero. Condition S contains the sign bit of the A.L.U. result. All four condition bits are connected to the C-module by the T BUS and may be tested by conditional C-μ instructions. A.L.U. condition codes are unaffected by D-μ instructions that use the A.L.U. to access the register file store as a D BUS source.

The architecture chosen for the A.L.U. module is shown in Figure 6.23 P. Importance has been placed on providing flexible operand manipulation by means of a 8 x 16 bit register file store. This has separate address channels for reading and writing data eliminating the complications of multiplexing the file access when the result of an A.L.U. operation is to be placed in the register file store. Source operands for the A.L.U. i.c.'s are obtainable only from the register file store; for this purpose the register file store is organised as two pages of 4 registers each page supplying the A or B operands of the SN74181. The carry-in input to the A.L.U. is supplied by the most signigicant bit of the micro-instruction source field. The destination result can be sent to any D-BUS destination address including either page of the register file store. This organisation is prefered because the availability of storage for more than one A or B operand reduces the number of micro-instructions required exclusively to transfer operands between, say, a single A or B register due to non-commutative operations.

The result from the A.L.U. i.c.'s is held temporarily in a register before it is gated to D BUS to prevent a race occurring when the destination modifies one of the source operands.

The register file store is loaded from D BUS by specifying an address between 0 and 7 (inclusive) in the destination field of a D-μ instruction. The lower four addresses refer to the A operand registers, similarly the upper four addresses refer to the B operand registers. The A operand registers may also be loaded by the Data Literal, DLIT, monadic micro-instruction. This causes 8 bits of the D-μ instruction from CREG to be loaded into a specified A operand register. The format of the DLIT micro-instruction is given in Appendix H 1.1.4. This feature is included to enable the generation of address constants and the initialisation of control variables.

6.5.0.0 Execution Speed of the A.L.U. Module

Internal timing signals for the A.L.U. module are derived by tapping off signals of the required delays from a cascade of gates. The speed performance is slower than

D BUS's capacity because Schottky clamped integrated circuits are not available to implement the register file store in the configuration chosen. Standard TTL versions of the A.L.U. i.c. are also used. A Schottky alternative would improve the A.L.U. module source response time, t_W, by 25%. They were not included in the original specification because of cost limitations.

D BUS source acknowledge times of the register files for ALU and MOV micro-instructions, t_W, is 90nS.. The D BUS destination acknowledge time for the register files, t_R, is 45nS. for B registers and 50nS. for A registers. The difference is caused by the DLIT micro-instruction control logic.

6.5.1 The Shifter and Bit Counter Module

The Shifter and Bit Counter Module exploits M.S.I. universal shift register i.c.'s to perform multiple shifts counter circuitry to enable multiple shifting independent of further micro-program control. A diagram of the Shifter and Bit Counter Module is given at Figure 6.24P. The module generates two D BUS source operands, namely the Shift Register,(SR)and the Bit Counter Register(BCR).

The shift control circuitry is activated when the four most significant bits of the D-μ instruction have the value 1110. The destination field of the micro-order is used to specify the number of shifts to be performed. The four least significant destination bits represent the number of shifts minus one. This allows the number of shifts to vary between one and sixteen, the latter number being specified when a bit count is required. The direction of the shift is controlled by the most significant bit of the destination field, C BUS_6. When this is set shifting takes place towards the most significant bit (left shift). C BUS_4 and C BUS_5 are used to specify the origin of the shift-in bit, Rotation, signed shifts and logical shifts with either 0 or 1 shift-in can be obtained as shown in Appendix H 1.1.2.

When a shift instruction is placed on C BUS a normal D BUS transfer is initiated with the shift register, (SR), as the destination module. On receipt of the DBWA signal the 1 to 0 transition of the shift decode signal, Im9, sets flip-flop Fc9 and loads C $BUS_{4\to6}$ into a staticising register, En. The setting of Fc9 causes the clearing of the bit counter register (BCR) Fd and Eb9, and the enabling of the shift register clock generator circuit by setting flip-flop Fc5. The first clock pulse from the SR clock generator is used to load SR with the data from D BUS and to load the shift counter, SC, (Ed) with a value given by the one's complement of the 4 least significant bits of the destination field of the instruction. The sending of the destination acknowledge, DBRA, is enabled immediately following the first clock pulse by resetting flip-flop Fc9. Flip-flop Eb5 is set according to the carry signal from the shift counter Ed. When Eb5 is set the control inputs to SR and SC are placed in the load mode. The 1 to 0 transition of Eb6 on setting Eb5 is used to reset flip-flop

Fc5 thus halting the SR clock generator. While flip-flop Fc5 is set the source module decoders, gates Ik9 and Il9, for SR and BCR are inhibited. This delays any D-μ instructions that reference SR or BCR while the shifter module is active until the resetting of Fc5.

6.5.1.0 Execution Timing for the Shifter Module

The execution speed of shift micro-instructions is limited by two factors: the response time of the asynchronous control logic and the shift register clock generator frequency f_{SH}. Frequency, f_{SH}, is limited to 25 MHz by the use of standard speed 74 series M.S.I. integrated circuits. Typically, a threefold increase of f_{SH} could be obtained by the use of Schottky clamped integrated circuits for SR, BCR, SC. Schottky clamped integrated circuits are used in critical propagation paths in the shift control logic, notably, to obtain a fast D BUS destination response time, t_R of 60nS..

The time taken to execute a shift instruction is given by:

$$t_W(\text{source}) + 60 + 40s \quad \text{nS.}$$

However since D BUS is freed after t_W(source) + 60 nS. the functional operation of the shifter can be made to appear transparent to the instruction timing if the program organisation is such that a source reference to SR or BCR is delayed by sufficient intervening micro-instructions. Schottky TTL enables a D BUS source acknowledge time t_W of 34 n S. for the shift register.

6.5.2 The Mapper Module

The Mapper Module has two modes of operation. The mapping may be defined by a map variable-element or by a map literal contained in the destination field of the MAP micro-instruction. The map literal mode enables the Mapper Module to be used for addressing individual bits of a data word without the need for masking patterns. The most signigicant bit of the destination field ($CBUS_6$), when a one, denotes a literal mapping. The circuit of the Mapper Module is given in Figure 6.25P.

The map-variable-element is held in the Map Register (MR) which is loaded by specifying it as the destination field of a D-μ instruction. The MER serves two functions in the implementation of the MINIC mapping algorithm: the least signigicant four bits are used to select the mapped bit from the data word obtained by a MAP micro-instruction and the remaining bits of MR, except the two most significant bits, are available as a D-μ instruction source after being shifted right four places. This generates the word index which is added to the base address of the domain pattern to obtain a pointer to the word specified by the source field of the MAP micro-instruction. The most significant bit of MR causes the map result to be forced to a zero when it is set. Similarily the second most signigicant bit of MR causes the map result to be forced to a one.

On execution of the MAP micro-instruction the source data is placed on D BUS and on assertion of DBWA by the source module, the bit being currently selected by data selector integrated circuit Kw is clocked into a 16 bit shift register. This register is called the Map Result Register, MRR, and it holds the partial result of a mapping. The result of the last MAP instruction is connected to T BUS to enable the mapper module to be used for testing specified bits of a pattern in conjunction with the map literal mode of operation. Register MMR is available as a D BUS source by D-μ instructions.

6.5.2.0 Execution Timing for the Mapper Module

Due to the simplicity of its internal control structure the mapper module is able to be operated at the maximum data rate of D BUS. There are no internal busy states of the module and the source and destination acknowledge times, t_W and t_R, are 38 nS. and 52nS. for MR and t_W, 32nS. for MW.

The MR scaling and single bit selection features of the mapper module enable very efficient implementation of the MINIC mapping algorithm by a micro-program subroutine, (presented in Section 6.7.0). In the interests of simplifying the mapper module hardware the micro-programmed algorithm differs from the MINIC mapping algorithm in the interpretation of a map, variable-elements in the following points:

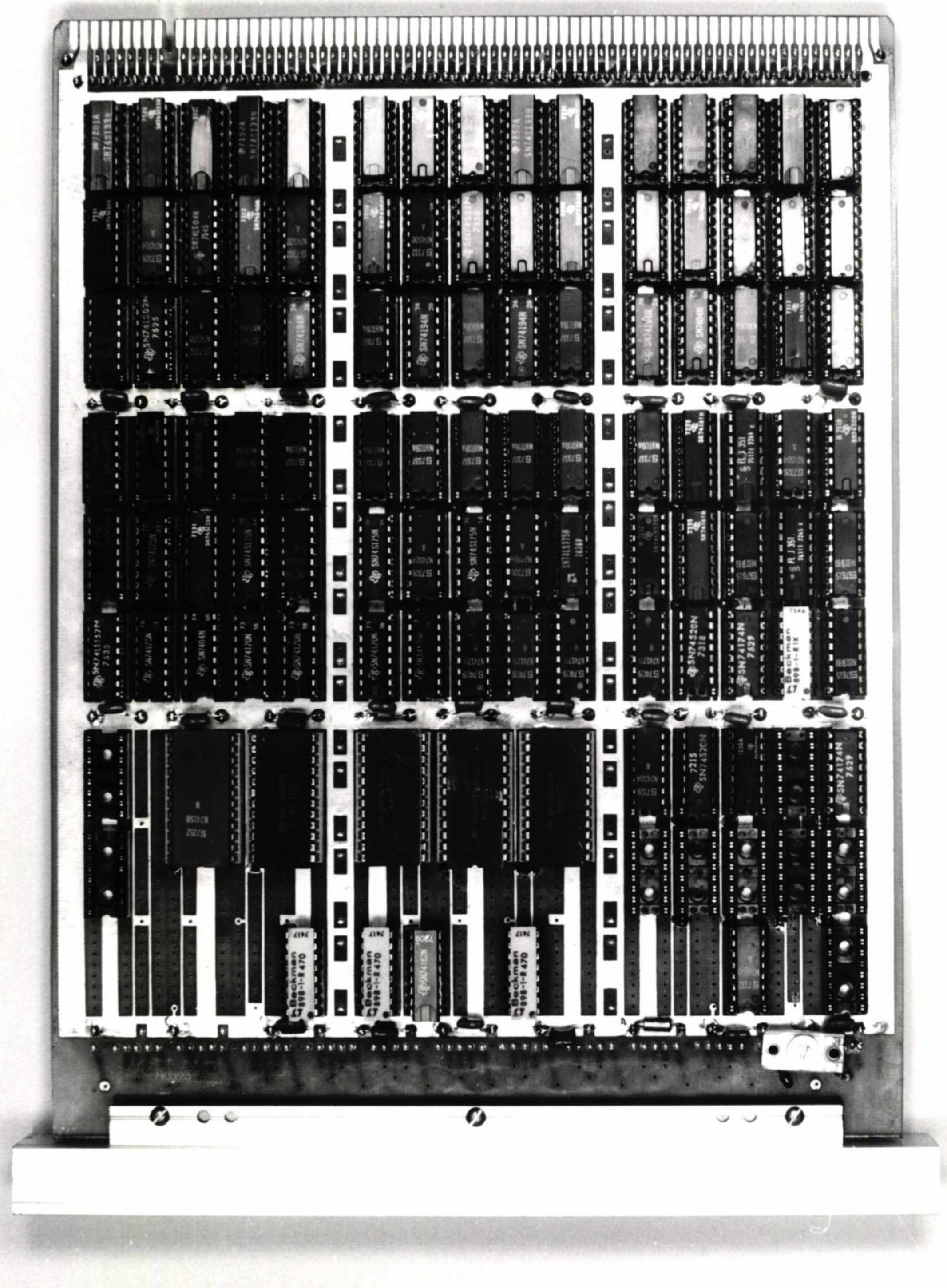


Figure 6.23a

The A.L.U. Module

a) Zero origin indexing is used for the micro-programmed implementation;

b) The order of the range vector is limited to 2^{14} bits instead of 2^{15} -1 bits;

c) The force 'zero' mapping is defined by the second most signigicant bit of the map-variable element instead of a zero map-variable-element. The 'force one' mapping is unaltered using the sign bit in both cases.

The effects of a) and c) above can be made transparent to the MINIC user by host computer software to preprocess the map variable elements before transferring them to the Modular Processor. The limitation of b) above is considered to be unimportant because the hardware network input space is of order 4,096 bits only.

6.5.3 The Data Memory Module

The Data Memory Module, (D.M.M.), comprises a D BUS interface, dynamic M.O.S. R.A.M. and a controller circuit to provide data storage for pattern variables and map variables of the MINIC system. It may also be used for the temporary storage of micro-program segments before their transfer to the writable-control store of the Control Module. The memory is organised as words of 16 bits and may be incremented in 4K word blocks up to a maximum 16K words. The hardware configuration at present consists of a wirewrapped memory controller circuit module and four memory circuit boards each providing 4k x 8 bits of storage. The memory boards use the 1103, 1K M.O.S. integrated circuit R.A.M. (Intel,1970).A dedicated position is used for the D.M.M. in the modular processor rack since the backwiring interconnections between the controller circuit and the memory boards do not form part of the General Bus backplane circuit board of Section 6.3.1.1. The memory controller circuit module and the memory boards are illustrated at Figures 6.26 and 6.28. Circuit diagrams for the memory controller and memory circuit boards are shown at Figures 6.27P and 6.28P respectively.

6.5.3.0 Operation of the Data Memory Module

The D.M.M. contains two D BUS destination addresses and one D BUS source address. These are called the memory address register, (M.A.R.) and the memory data register, (M.D.R.). M.D.R. may be used as a D BUS source as destination and it holds information read from the location addressed by MAR or about to be written into the location addressed by MAR. Memory cycles are initiated by a dedicated MOVμ-instruction operation code. Data may be transferred to MDR or MAR using standard MOV or ALU micro-instructions, in which case no memory cycle action takes place. When data is placed in the MDR then the memory controller is initialised such that the next memory cycle to be initiated is a write cycle. Once the MDR has been used as a D BUS destination address no further data transfers can take place using the MDR as an operand until a write memory cycle has been initiated and the memory access time elapsed.

Memory cycles are of three types: external memory cycles; indirect memory cycles; and refresh memory cycles. External memory cycles are initiated by the execution of the MOVC and INCC micro-instructions using either the MAR or MBR as the destination address. The memory cycle will be a read or a write one as described in the previous paragraph.

An indirect cycle is caused if the most significant bit of the address in MAR is set. Indirect cycles are always read cycles, the result being placed in the MAR instead of the MBR and a consecutive cycle is initiated which will be a read, write or indirect cycle depending upon the original cycle initiation and the value of MAR. The danger that endless loops of indirect cycles may occur is avoided by counting consecutive indirect cycles until a limit of 16 cycles is reached; MAR_0 is forced reset and an error condition is then flagged which is communicated to C-module by the TBUS.

Refresh cycles occur asynchronously at about sixty micro-second intervals and are initiated from within the memory controller. They are necessary to refresh the information stored in the dynamic MOS RAM integrated circuits of the Memory Circuit Boards. The internal address decoding of the 1103 RAM is organised into rows and columns of dynamic storage cells. In order to refresh the information a read cycle must be accomplished for each of 32 row addresses within two milli-seconds. The memory controller performs refreshing by switching the row address from MAR to a refresh address counter and generating a "Chip Select", CS, signal for each 1K address segment simultaneously. A read cycle is then caused with the data in MDR undisturbed on its completion. Refresh cycles are transparent to the externally initiated cycles, their only effect being an apparent increase in the memory cycle time affecting about 2% of the memory cycles.

6.5.3.1 Memory Controller Timing Generation

On initiation of a memory cycle the appropriate micro-orders are generated by the memory controller circuit module and transmitted to the memory circuit boards. Timing control is achieved by propagating a signal down a cascade of TTL inverter circuits. When the signal reaches the end of the cascade the inverse of the signal is propagated from the start of the cascade. The cascade can be viewed as a dynamic, asynchronous version of the Johnson counter used to generate timing orders for the C-module. The total propagation delay through the cascade is adjusted by adding or removing stages to equal half the required memory cycle time for the 1103 MOS memory i.c.s. The pulse widths required for the 1103 precharge, chip enable and read/write inputs are generated by decoding the appropriate tappings on the cascade.

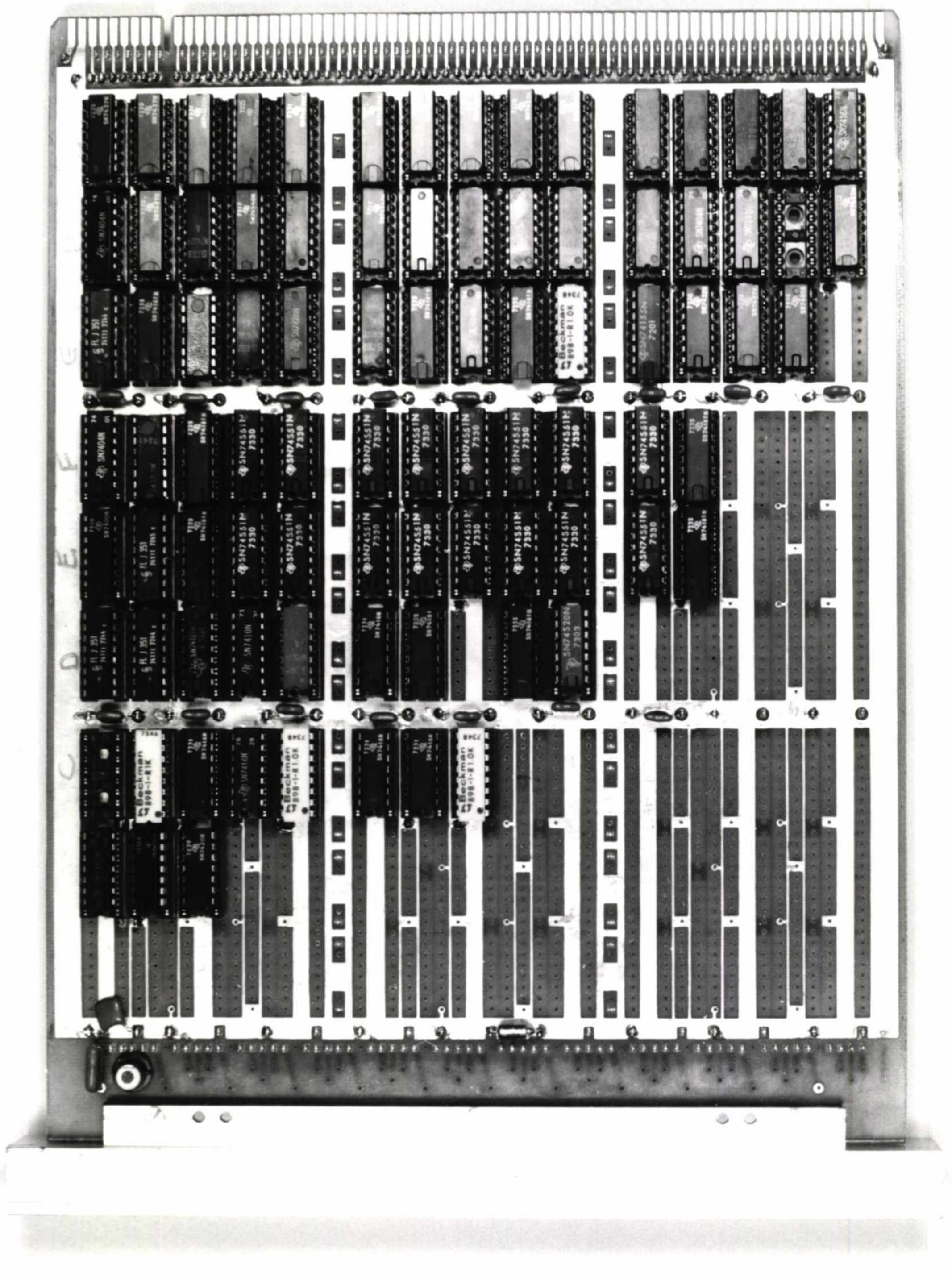


Figure 6.26
The Memory Controller

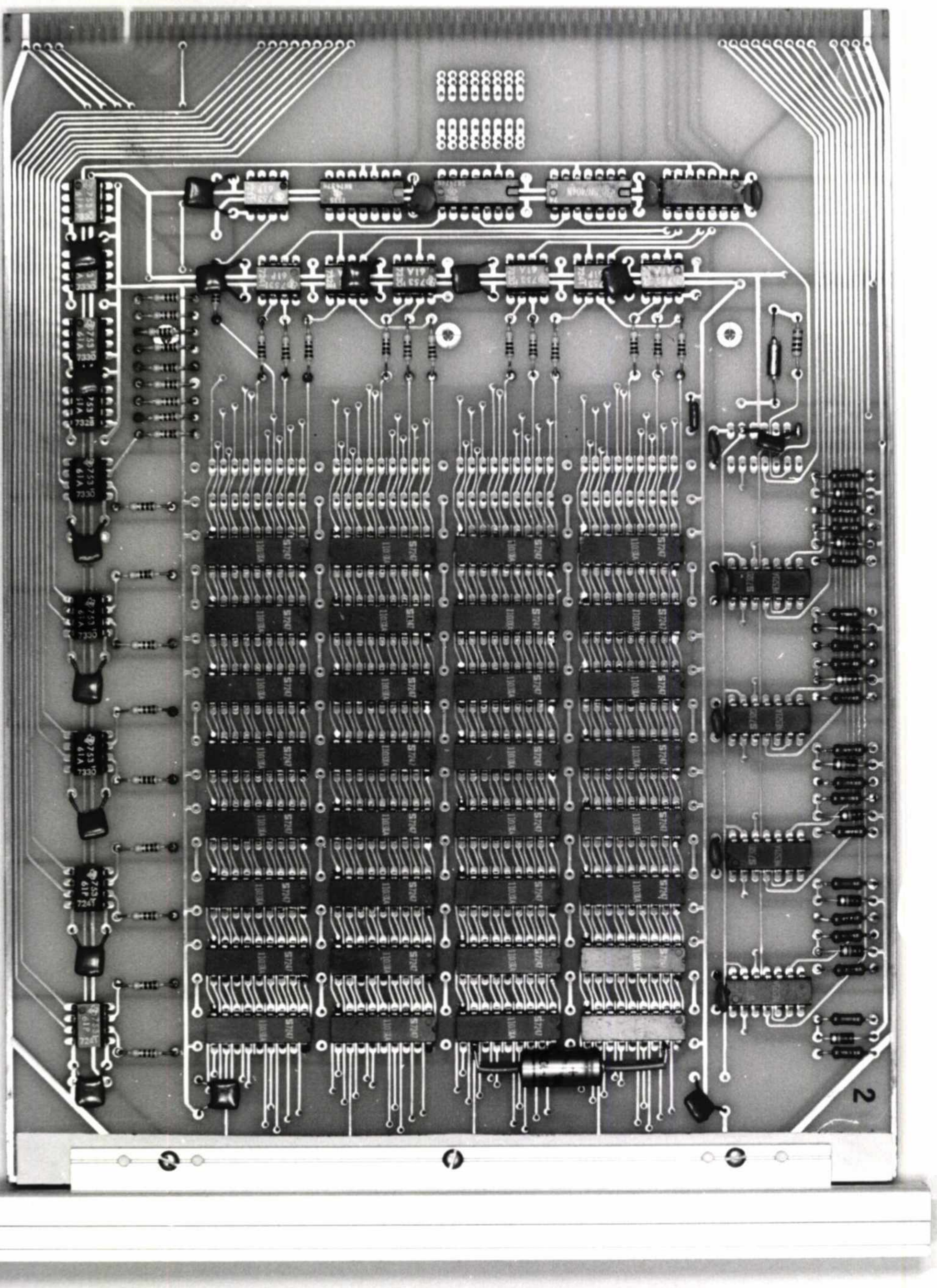


Figure 6.28
M.O.S. Memory Board

An active delay timing generator circuit is chosen in preference to a passive delay-line based circuit mainly for reasons of cost and constructional convenience. Delay lines are not readily available in Dual-In-Line, (DIL), packages suitable for use with the wire-wrap constructional methods used for the memory controller module. A passive delay-line implementation would be expected to have greater stability compared to the active delay cascade with respect to temperature and power supply variations. However no practical problems have been encountered with the active delay implementation.

A diagram illustrating the timing relationships of the MOS memory micro-orders is shown at Figure 6.30P.

During a read-cycle the data becomes available to be read from MDR by DBUS after 380 nS.. A new cycle cannot be initiated until after the end of cycle signal, (EOCYC). This enables MAR and MDR as DBUS destinations by resetting their respective busy flip-flops. EOCYC occurs after 560 nS.. The minimum DBUS destination acknowledge times for MDR and MAR, t_R, are 60 nS. each.

6.6 Interface Modules

Interface modules (I-modules) unlike D-modules described in the preceeding section do not perform any data transformations, their purpose being to transfer data between the modular processor and an external process which may be outside the control of CBUS. The internal structure of I-modules is generally uncomplicated. Where the external process is asynchronous with respect to the C-module,semaphore flip-flops are used together with protocol signals to synchronise data transfers. The scheme is similar to that described in Section 6.3.1 concerning the skeleton interfaces between DBUS and the internal logic of modules.

Three modules are classified as I-modules, these being: the Keyswitch and Display module, (KD-module); the Minerva Interface Module, (MI-Module); and the host Computer Interface module, (CI-module). These are described in detail in the following sub-sections.

6.6.0 <u>The KD-module</u>

The KD-module provides facilities for displaying the state of CBUS, DBUS and TBUS using three sixteen bit L.E.D. displays. Sixteen data Key switches are provided whose positions may be transferred to any DBUS destination by an appropriate D-μ instruction. Key Switches are also provided to enable the DBUS protocol signals DBWA-, DBRA- FDL- and MSTRCR-. The state of the data key switches are 'force' writen to DBUS by the FDL- signal to enable the initial loading of micro-programs in the C-module WCS, (c.f. Section 6.4.1.0).

Initial micro-program load using the KD-module proceeds as follows:

a) The C-module mode keyswitch is placed in the EXM position and the start address of the micro-program to be loaded is placed in CSAR as described in Section 6.4.1.0;

b) The binary code representing the first micro-instruction is placed on the KD-module data keyswitches and the FDL Keyswitch depressed. This causes CSBR on C-module to be loaded with the value of the data Keyswitches;

c) The WRITE keyswitch on C-module is depressed to deposit CSBR in WCS;

d) Subsequent micro-instructions may be loaded by depressing the CONT C-module keyswitch and repeating steps b) and c).

6.6.0.0 Circuit Description of the KD-module

The circuit diagram of the KD-module is given at Figure 6.31P. Display of the TBUS and CBUS is accomplished directly through buffer gates to LED's. DBUS is displayed as the result of the last DBUS transfer and requires a staticizing register to be loaded by the DBRA- signal. Specific conditions such as, all zero data, all ones data, the most significant bit, and the least significant bit of the last DBUS transfer are detected and connected to TBUS to provide conditional micro-instruction branching according to the state of DBUS.

The data Keyswitches are interfaced with DBUS by conventional DBUS source decode logic. Since the data keyswitches are a static source, module busy logic is not required. The DBUS source response time for the data keyswitches, t_W is 36nS..

6.6.1 The MI-module and the CI-module

These modules are constructed on a common wire-wrap circuit and although their individual operation is distinct, common control gates are shared between the modules. A circuit diagram for the combined MI- CI-module is shown at Figure 6.32

6.6.1.0 The MI-Module

The MI-module provided an interface between DBUS and the modified network hardware of Section 6.1. The MI-module is designed to be compatible with the same input/output connections of the network hardware as the D.E.C. PDP11/40 DR11C interface. However there are major differences between network operation using the modular processor and the PDP 11/40. Separate network control inputs are used for the network micro-orders using the MI-module instead of the multiplexing of the data channel using the PDP 11/40. The individual network register load micro-orders are generated by the MI-module such that the network device registers appear as DBUS source and destination addresses. A control literal instruction is decoded by the MI-module to enable up to two concurrent network control micro-orders to be sent to the network. The network provides two DBUS source addresses corresponding to the 16 bit response registers for layers 1 and 2 of the network labelled MRR1 and MRR2 respectively. Adjacent source addresses are allocated to MRR1 and MRR2 to simplify the decoding logic. The outputs of MRR1 and MRR2 are multiplexed by the network hardware using the least significant bit of the CBUS source field to select the appropriate register.

Since there are comparatively long propagation delays associated with sending a response register request to the network hardware and then receiving the response this extends the DBUS source response time for the network response registers to a minimum of 85nS. for t_w. The MRR1 and MRR2 DBUS source decode logic is inhibited when MR is low. MR goes low after the loading of the network data register, N.D.R. while propagation takes place through the net elements. This time is about 3μS. and represents a maximum value, t_w for inappropriately timed network response register DBUS requests.

Four consecutive DBUS destination addresses are used to load the network data and control registers NDR, CAR, TDR and TMR enabling powerful and concise micro-programming capability. On decoding a DBUS destination reference to a network register the data is received by a temporary register in the MI-module and the DBUS read acknowledge signal, DBRA-, generated. The DBUS destination response time, t_R, is 40nS. minimum for network registers. The DBUS destination decode logic then goes busy for a period of 300nS. to allow the transfer to the network register to take place. If NDR is the destination then the 300nS. delay signal is replaced by the MR signal generated by the network hardware. The transfer of the data from the MI logic to the network register takes place in two stages. A delay of 120nS. is allowed for the data signal propagation to the network hardware before the load minerva interface register micro-order, LDMIR, is sent to the network. A further 200nS. delay is made to allow for propagation delays in the network hardware before the appropriate micro-order, selected by the two least significant bits of the CBUS destination field is sent to the network hardware.

The control micro-orders for the network are generated by the control literal micro-instruction CLIT which is distinguished by having the five most significant bits in the function field of the micro-instruction set, i.e. 76_8. The remaining ten bits of the micro-instruction word are free to be decoded arbitarily by any module. The MI-module uses the six least significant bits of the control literal micro-instruction, the four unused bits are required to be zero. The six active bits are decoded as two fields of three bits each enabling the simultaneous generation of two micro-orders selected from two groups of 8. The 0_8 micro-order in each group is unconnected to enable only one micro-order from each field to be significant. The control micro-orders AUTO, SAC, RAC2, ICAR, CCAR, CTDR and LTMR generated by the most significant field. The last named micro-order duplicates the direct loading of the net teach mask register by a DBUS destination reference but its inclusion is justified because connection to TMR is made from the most significant byte of the data word, and connection to CAR and TDR by the least significant byte. This enables close packing of teach mask data to be obtained with micro-instruction economy. The control orders generated by $CBUS_{10 \rightarrow 12}$ are TE2, TE1 , RAC1 and CTDR . Three micro-orders are unassigned.

The net micro-order signals are pulses of 400 nS. duration. A DBUS read acknowledge signal is generated after 40nS. from the receipt of the CLIT micro-instruction. The CLIT micro-instruction decode logic is disabled for the duration of the micro-order signal.

6.6.1.1 The CI-module

The CI-module permits data transfers between the modular processor and a DEC PDP 11 series computer with a DR11C general purpose 16 bit parallel interface. The CI-module has one DBUS source address to read data, one DBUS destination address for outputting data to the computer. Both are referred to as the computer interface register, CIR.

Data transfers from the PDP 11/40 to the modular processor take place as follows:

A signal REQB is sent to the PDP 11/40 when MMPP is available to accept data. The DBUS source address decoder is disabled during the REQB signal.

On performing an output instruction a new data ready NDR signal is output from the PDP 11/40 which resets the signal REQB and enables the CIR DBUS source address decoder. The data is statisized in a register on the DR11C interface by the NDR signal and sent to the CI-module. Data is transferred from the DR11C interface to any DBUS destination during an appropriate D-μ instruction. At the completion of the D-μ instruction the REQB signal is regenerated.

The DBUS source acknowledge time, t_W, has a minimum value of 30nS. when the computer data is ready but it is determined by external factors otherwise.

Data transfers from the modular processor to the computer interface follows a similar pattern.

During the execution of an appropriate D-μ instruction data is read from DBUS to a staticizing register in CI-module. On completion of the micro-instruction the signal modular processor data ready, MPDR is sent to the DR11C interface and the DBUS destination decoder is disabled. When the PDP 11/40 has completed an input instruction a signal computer data received, CDR is sent to the modular processor resetting the signal MPDR and enabling the DBUS destination decoder for further output data transfers.

The DBUS destination acknowledge time, t_R, for CIR has a minimum value of 30nS. assuming previous data has been input by the computer.

6.7 Micro-programming MINIC Target Code

The following section presents examples of the micro-code implementation of some MINIC statements and discusses the preformance advantages obtained. Micro-programs are written in a symbolic micro-assembly language the symbols of which have appeared in the text of the preceding section and are summarised in Appendix H.

6.7.0 Micro-program Execution Speed

The execution speed of a modular processor micro-program cannot be predicted as confidently as that of machine code programs written for the DDP516 and presented at Appendix C. The asynchronous nature of DBUS transfers and of micro-instruction passage along the micro-instruction decode pipe-line mean that micro-instruction cycle timings are dependent upon the context in which the micro-instruction appears.

The time required to execute a micro-program segment consisting of consecutive D-μ instructions is given by the recursion:

$$T_{m\rightarrow n} = \sum_{i=m}^{n} t_{Li} + t_{Wi} + t_{Ri} + Y_i + Z_i$$

where $\quad Y_i = 0$ for $t_{RWi} \leq T_{i\rightarrow j}$, $j \leq n$

or $\quad Y_i = t_{RWi} - T_{i\rightarrow j}$ for $t_{RWi} > T_{i\rightarrow j}$

and $\quad Z_i = 0$ for $t_{RR} \leq T_{i\rightarrow k}$, $k \leq n$

or $\quad Z_i = t_{RRi} - T_{i\rightarrow k}$ for $t_{RRi} > T_{i\rightarrow k}$

Empirical values for t_R, t_W, t_{RW} and t_{RR} are given in Appendix H 1.2.2. The time t_L is the latency time between each D-μ instruction and depends upon the time taken to execute the preceding D-μ instruction and the number, if any, of intervening C-μ instructions. The C-module fetch cycle for D-μ instructions is 85nS. which is less than the execution time for most D-μ instructions. It may be assumed therefore that t_L is the micro-instruction pipe-line delay time, 20nS., or the time taken to execute any intervening C-μ instructions minus the time required to execute the previous D-μ instruction which ever is the greater:

$$t_{Li} = pt_c - (t_{L(i-1)} + t_{W(i-1)} + t_{R(i-1)}),\text{subject}$$

to $\quad t_{Li} = 20$nS. minimum.

p is the number of intervening C-μ instructions and t_c is the C-μ instruction cycle time, 138nS..

6.7.1 Systems Organisation

MINIC statements are translated into micro-code segments by a version of the MINIC compiler at present being written for the DEC PDP 11/40 computer. The run-time tasks of the system are separated into two groups, numerical tasks including all arithmetic expressions and subscript address calculations being performed by the PDP 11/40 and all pattern expressions being performed by the modular processor. Storage for real variables and arrays is allocated in the PDP 11/40 core memory and storage for pattern variables, map variables and net variables is allocated in the memory module of the modular processor.

The target code generation routines of the MINIC compiler produce micro-code for each pattern expression segment of a MINIC program together with an ordered string of argument pointers and temporary locations to be placed in the MMPP memory module. During execution of the micro-code register B0 is reserved as a pointer to the argument string. Register A0 is reserved for use as the expression loop counter in a similar manner to the X register of the DDP516 target code.

Initial loading of micro-code is accomplished by the boot-strap loader micro-program given in Appendix H 1.3. Execution of a micro-code segment is initiated by a signal from the PDP 11/40; a signal is returned to the PDP 11/40 by the MMPP at the completion of the micro-code segment. Any results of arithmetic expressions required by the modular processor for indexing subscripted variables are transferred from the PDP 11/40 when requested by the modular processor during program execution.

6.7.2 Micro-code for MINIC Pattern Operators

Boolean operations are directly implemented by single micro-instructions. The comprehensive nature of the ALU-module enables the optimisation of the MINIC, NT operator for each sub-expression of two pattern variables. The mapping operator, BY is implemented by a micro-subroutine utilising the MAP micro-instruction as follows:

A0 - expression loop counter, B0-argument string base,
A3 - argument string index, B3- temporary variable.

```
MAPMS: ADD  A3, B0, MAR    ; Get MAP arg.
       ASR  A0, 4          ; Convert to bit index
       MOV  MDR, B3        ; MAP arg. address
       INC  A3, MAR        ; get PAT arg.
       MOV  SR, A2
       ADD  A2, B3,A2,     ; points to MAP element
       MOV  MDR, B3        ; PAT arg. address
       DLIT 16,A3          ; count for MAP bits
L1:    MOVC A2, MAR        ; get MAP element
       INC  A2, A2         ; bump for next
       MOV  MDR, MR        ; send to mapper
       MOV  MR, A1         ; ÷ 16 for word index
       ADD  A1, B3, MAR    ; get from PAT arg.
       DEC  A3, A3         ; bit count
       MAP  MDR            ; map it
       BCF  ARZ, L1        ; check A3
       RTN                 ; return result in MW
```

The execution time of MAPMS can be calculated by the method described in Section 6.7.3 in conjunction with DBUS source and destination timing given in Appendix H 1.2.2. Using this method gives an execution time of 24µS. for MAPMS, corresponding to an average interconnection time of 1.5µS.per bit.

6.7.3 Examples of MINIC Target Micro-Code

The Target micro-code for a simple MINIC pattern expression is given below. Calls to micro-subroutines for argument manipulation are explained at Appendix H 1.3.0.

LET PAT Z BE PAT X OR PAT Y BY MAP M

```
(1)           JSA  GO             ; wait for host computer
(2)  L1:      DLIT MI, A3         ; MAP variable index
(3)           JSA  MAPMS          ; do mapping
(4)           DLIT XI, A3         ; PAT X index
(5)           JSA  GET            ; get argument
(6)           MOV  MOR, A1        ; PAT X
(7)           MOV  MR, B3         ; map result
(8)           OR   A1, B3, B3     ; OR op
(9)           DLIT ZI, A3         ; PAT Z index
(10)          JSA  PUT            ; Store result B1
(11)          DEC  A0, A0         ; pattern expression count
(12)          WBRR L2             ; wait for D.E.C.
(13)L2:       BCF  ARZ, L1        ; Loop until done
(14)          MOV  A0, CIR        ; done, tell host computer.
```

Argument String:	B0 → 0	Pattern expression size
	MI = 1	MAP M Address
	YI = 2	PAT Y Address
	XI = 3	PAT X Address
	XI = 4	PAT Z Address

The use of the network hardware is illustrated by the following example:

LET PAT R BE NET N IP PAT Z

```
 (1)         JSA  GO            ; wait for host,
 (2)         DLIT NI, A3        ; get NET start
 (3)         ADD  A3, B0, MAR   ; from argument string
 (4)         MOV  MBR, MCAR     ; send to NET
 (5) L1:     ASL  A0, 2         ; index for IP
 (6)         MOV  SR, A0
 (7)         DLIT 4, A3         ; count for NET
 (8)         MOV  A3, B2        ; response word
 (9) L2:     DLIT ZI, A3        ; get PAT Z
(10)         JSA  GET
(11)         MOV  MBR, MDR      ; send to NET
(12)         DEC  A0, A0
(13)         MOV  B2, A3
(14)         DEC  A3, B2
(15)         CLIT INCAR         ; next NET card
(16)         BCF  ARZ, L2
(17)         ASR  A0, 2         ; back to outer
(18)         MOV  SR, A0        ; loop index,
(19)         MOV  MTRR1, B1     ; get NET response
(20)         DLIT RI, A3        ; put at PAT R
(21)         JSA  PUT
(22)         DEC  A0, A0        ; complete outer loop
(23)         WBRR L3            ; wait for BUS
(24) L3:     BCF  ARZ, L1
(25)         MOV  A0, CIR       ; done, tell host.
```

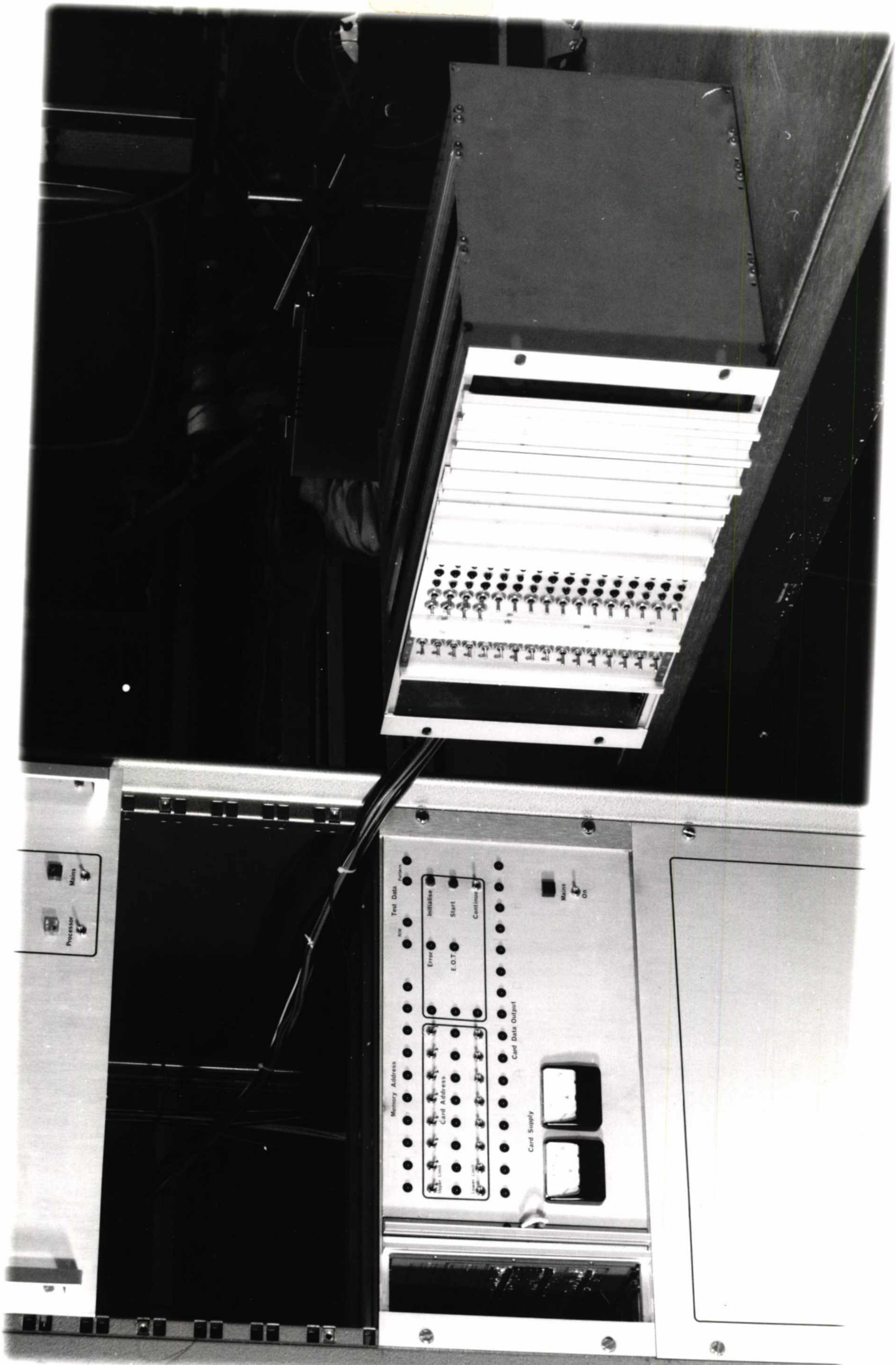


Figure 6.33

Modular Micro-Programmable Processor

6.8 A Summary

The use of available M.S.I. functional building blocks is considered for improving the through-put of net work systems. The separate proposals are considered:

1) A "synthesis of network structures" approach produces a viable hardware design which is of limited flexibility because it has few control states;

2) A serial-parallel implementation utilising shift registers and parallel masking logic offers flexibility and high through-put for some mappings at the expense of memory requirement. The memory requirement is variable for various mappings unlike the sequential mapper and can have a maximum value for an m to n mapping of m x n bits compared to only $n\log_2 m$ bits for the sequential mapper. An implementation with limited storage, ($<n \times m$), could prove useful for restricted classes of mapping such as linear transformations and diameter limited mappings. A general expression for the average storage required over the permutation set of mappings has not been found;

3) The final approach considers the primitive network operators in the MINIC language. Logic modules are used to realise specific MINIC operators under

sequential control. Vertical micro programming is exploited to obtain high speed and flexibility of control.

The processor, (M.M.P.P.), has no fixed order code for instructions obtained from its main memory and in this respect is similar to the Standard Logic CASH-8(1973) and Burrough's B1700 (Wilner, 1972)processors.

The essential features of M.M.P.P. are pipe-lined micro-instruction decoding, concurrent operation of control micro-instructions and data micro-instructions, asynchronous processing enabling overlapped operation of some data processing modules and an extendable architecture to allow future hardware optimisation of costly programmed operations. Data processing modules are described for performing two's complement binary arithmetic, shifting, sixteen logical operations on two words and the inter-connection and bit counting operations peculiar to MINIC

Interface modules are described that connect M.M.P.P. to a host computer and control the network hardware. An operator's keyswitch and display module is also described that allows the manual loading of bootstrap micro-programs. Other modules include a dynamic MOS RAM of 8K words of 16 bits and its controller for the local storage of MINIC operands.

Micro-programs are presented which exhibit a speed-up of 30 times when compared to software simulation for the MINIC interconnection operation. Execution time is limited by the access time of the memory module used for operands. A further speed improvement of 50% could be obtained by using memory with access times comparable to DBUS cycle times. This would cost about 10p per bit compared to 1p per bit for the existing memory module.

Bit counting, required for the MINIC SUM operation, is performed by counting the overflow bit from the shifter-module while executing a complete word rotation. This can be initiated concurrently with other micro instructions; the time taken varies between 100nS. and 900nS. according to programming context. This represents an improvement of two orders of magnitude faster than the software equivalent routine, PS1Z, for patterns of bit density averaging a half.

This section has presented some results of the hardware design of M.M.P.P. Work remains to be completed on the production of a cross-assembler for the development of micro-programs using the symbolic micro-assembly language presented in the examples. The macro facilities of the DEC PDP 11 family assembler program are being used to enable micro-programs to be embedded in PDP 11 assembler source programs.

CHAPTER 7

7.0 Conclusions

This thesis has been concerned with the development of a computing system that realises cellular networks of R.A.M. elements having flexible topologies which would otherwise require impractical computer time to be evaluated by simulation. Networks appear to the Minerva system user as parallel machines although they are implemented by word sequential hardware that is optimised by special purpose hardware and micro-programming with regard to the limitations imposed by the systems peripheral devices.

The Minerva system can be used to design, say, a practical n-tuple pattern recogniser which, after training and optimising procedures are completed, generate tables for programming a P.R.O.M. hardware realisation. At the more speculative end of the systems usefulness it can be used by the brain modeller to study either neurological or psychological behaviour.

The development of the component parts of the Minerva system is summarised in the following three sections.

7.1 The Network Hardware

The design of the network hardware and its associated custom integrated circuit is presented in Chapters 2 and 3. The S.L.A.M.-16 integrated circuit is now clearly superseded through the technological development of the commercial R.A.M. integrated circuit.

The only feature of S.L.A.M.-16 not provided by R.A.M. integrated circuits, the set and reset inputs, were never used in the MINIC language to avoid confusion between their global effect on the network and the specified limits of MINIC NET variables. The practical use of the set and reset inputs is limited to manual initialisation of the network hardware by operator intervention.

The network hardware is operating reliably five years after its commissioning. Mid-term problems were experienced due to the failure of some S.L.A.M.-16 circuits. On visual inspection of failed chips, fusing of the metalisation carrying power supply voltages was noticed. The primary causes, such as junction breakdown, diffusion wandering at points of high field strength or short circuits caused by swarf generated during the encapsulation process could not be positively ascertained.

Few drawbacks within the context of the MINIC system have been found with the structure of the network hardware although increased flexibility could be obtained by hardware to increase the number of inputs to an element by connecting S.L.A.M.-16's in parallel. This would require modifications to the N.E.C. in the form of multiplexer circuits connected to the inhibit inputs of each S.L.A.M.-16 and additional data lines to each N.E.C. It is possible to realise larger S.L.A.M. elements using MINIC software but this is slow and cumbersome involving mapping operations and multiple iterations of network response.

Modifications to the original network hardware not requiring major reconstruction have been described in Chapter 6 and concern improving the matching of data path widths between the network response and the host control processor. Other implementable modifications were considered at this time; one being the hardware loading and outputting of the contents of the S.L.A.M. -16 stores in bit parallel format. Software methods of outputting the contents of S.L.A.M.-16 stores produce an inconvenient format with four store bits from each of the four first layer S.L.A.M.-16's appearing in each of four consecutive output words. Dedicated hardware to re-format the output bits as they are generated was rejected since this operation could be accomplished efficiently by the mapping module of the modular processor.

It is interesting to compare the component cost of the network hardware with one of similar structure built today using updated technology. Five 4K word by 1 bit static MOS, RAM integrated circuits would replace all

256 NEC's, economies are possible because of reduced power supply requirements; higher printed circuit board density would result in economies of edge connectors and racking hardware.

Costs	1971	1978
RAM's	£5150	£50
P.C.'s + connectors	£1000	£100
Racks. etc.	£150	£60
Power supplies	£200	£50
	£6500	£260

The current trend in hardware costs offers two areas for further development. Dedicated machines for low cost applications such as blind reading aids signature verifiers, etc. are now economic. Software development costs could be reduced by hardware to implement high-level language machines. This would be based on the modular micro-programmed techniques presented here.

In an updated implementation of the network hardware advantage can be taken of the large reduction in memory circuit cost to increase the size of the machine. This can be accomplished by either increasing the number of inputs per R.A.M. element or by increasing the total number of individual R.A.M. elements. Ullmann & Kidd's results using the n-tuple method for handwritten character recognition imply an optimum number of inputs per element of about 14. R.A.M. circuits of this size are now comparable to the original cost of S.L.A.M.-16.

Increasing the number of elements is more costly in terms of interconnection hardware than increasing the element size. In n-tuple character recognisers increasing the number of elements makes possible an increase in the number of classes in a multi-way classifier. Alternatively, a finer input retina quantisation from which to select n-tuples may be used. Character recognition by humans is unaffected by increasing the quantisation above 20 x 20. The affect on machine n-tuple recognisers of using finer quantisation is not known although it is suspected that large training sets would be required for significant n-tuple features to be extracted.

Other applications where finer retina quantisation could be useful are in scene analysis and texture analysis; the former as part of a peripheral vision system and the latter to enable measurement of, say, the degree of polish of a bearing surface.

One objection that could now be raised to building network hardware with no greater degree of parallelism than that of its host processor data paths is that the availability of low cost user micro-programmable computers allows the R.A.M. elements to reside in the main memory of the computer to be accessed in a network fashion by high speed micro-programs. This objection is valid to a large degree and was one of the underlying reasons for the development of a micro-programmable processor within the MINERVA system where it is also possible to simulate modified elements if required. However, access times between computer memory and R.A.M. elements being equal the computer micro-code simulation can only generate one element response per main memory cycle whereas the word sequential network hardware generates several parallel responses according to the input data path width divided by the number of inputs per R.A.M. element.

Practical network applications using R.O.M. elements,such as a blind reading aid (Nappey, 1977) are limited by physical size and cost to hardwired construction where a required data throughput can be achieved at lower cost by increasing the parallelism of the network rather than by increasing the instruction rate of a simulation program.

7.2 The MINIC Language

A high level language and its compiler is presented in Chapters 4 and 5 which is shown to be successful in representing existing R.A.M. networks and as a design tool for generating novel network structures.

Networks are limited by having homogenous combinational elements. Non-homogenous networks of elements with varied numbers of inputs can be made by mask patterns that effectively disconnect inputs at the cost of not fully utilising the available element stores. Fully utilised elements could be provided by circuitry to select R.A.M. element address lines as data inputs or system address lines depending upon the number of inputs per element required. Complications of bit packing in the host computer due to variable size data paths could be solved by providing buffer storage for network input and output and causing asynchonous interrupts in the host processor when the buffers are full.

The MINIC language could control the selection of element sizes by an element size vector defined in the DIM statement and associated with every occurrence of the NET variable.

Asynchronous networks of elements having different propagation delays associated with each element can be implemented in MINIC by successively interating a NET expression using a set of masks to select the outputs from active elements. Delays would be quantised. The following program illustrates the procedure:

```
10  FOR A=0 TO T

20  LET PAT R BE (NET N IP PAT X) AN PAT D (A) OR
    PAT R AN NT PAT D (A)

100 NEXT A
```

T is the number of quantisation levels representing the maximum delay of any one element; PAT X is the net input and may be a function of PAT R, the NET state vector.

Asynchronous networks are of interest because of their similarities to neural networks; however, concepts such as transient length and cycle length will not be as useful in analysing asynchronous networks. Such networks offer an important area for future research.

Proposals for extending the MINIC language are presented in Chapter 4.9 These are concerned with providing interfaces between variable types to improve the generality of the semantics.

The dramatic fall in the cost of the external network hardware and the general applicability of the language in areas of logic simulation and coding means the rewriting of the MINIC compiler in a machine independent language would be profitable. The work required to transport the language to any computer system supporting the machine independent language would be reduced to modifying the object code generators to produce target code for the new computer system.

7.3 The Modular Processor

The modular processor is described in Chapter 6. It fulfils three requirements:

1) it provides a micro-programmable implementation of network interconnections giving an improvement in network data-throughput of 30 times that of networks with simulated interconnections;

2) it provides a vertical micro-programmable facility of 80nS. instruction cycle time for writing interpretive programs of MINIC semantics;

3) its modular design and standard control interface allow additional special purpose processing modules where micro-program emulation is not viable.

The enhancement of program execution by micro-programming is now an established programming technique with many applications reported in the literature (see Agrawala and Raucher, Chapter 8, 1976). However the speed improvement reported appears to approximate to the ratio of main memory access time to control memory access time. This suggests that micro-programming does not overcome any architectural shortcomings in a machine structure.

The modular processor is an approach to go beyond this limitation and exploit the availability of low cost, high speed M.S.I. to enable construction of a hardware module for a specific data operation.

Internal processing by modules is under micro-program initiation, is asynchronous and may be concurrent. Micro-instruction fetches and sequence controlling micro-instructions are executed concurrently with data processing.

Since the construction of the C-module (Section 6.4) and the A.L.U.-module (Section 6.5.0) LSI circuits of micro-program sequence controllers and bit-slice ALU's complete with registers have been introduced, (Advanced Micro Devices Inc.). These would simplify and reduce the cost of constructing the C-module and A.L.U.-module however the micro-instruction cycle time would not match that obtained by the descrete component C-module, this being 130nS. for the LSI version compared with 80nS. for the discrete version.

The design of a micro-instruction programming language and translator for the modular processor to enable more general use has been left for future development.

7.4 Further Work

At present an array structured adaptive processor (A.S.A.P.) is at an advanced stage of construction. A.S.A.P. will be controlled using the MINIC language and its internal control orders will be generated using M.M.P.P.

A.S.A.P. is a regularly connected network of 2,404 R.A.M.'s grouped into processing elements (P.E.) of 4 R.A.M.'s together with storage registers and control circuits. Each P.E. has ten data inputs and four outputs. They are arranged on a rectangular torroidal grid, of 24 x 24 elements. Each of the four P.E. outputs maybe connected to its nearest neighbour along perpendicular axis of the rectangle. Interconnections between P.E.'s are hardwired but limited variations of network topology can be realised by multiplexers connected to the address inputs of each R.A.M. These allow five different types of topology as follows:

1) Address inputs and P.E. outputs may be reconnected to common bus connections giving the structure of conventional bulk random access memory to enable initialisation and interrogation of the R.A.M. functions;

2) One address terminal of each element can be connected to an external data source;

3) One address input can be connected to a register containing the current output state of the element or it can be connected to one of four common control lines according to the propagation direction of the element;

4) Address inputs can be connected to the current output state registers of either 4 or 8 nearest neighbour elements on a rectangular grid. When 4 connections are used the remaining address in parts come from individual "program registers" independently setable for each group of 4 elements;

5) When used in the 4 or 8 connected modes alternate rows and columns of the array can be bridged

selectively to realise sizes of torroidal rectangular array between 24 x 24 and 12 x 12.

Early work on two-dimensional pattern processing arrays centers on Unger's S.P.A.C. (1958). S.P.A.C.'s processing elements were controlled by broadcast instructions common to each processor. Algorithms were written using the S.P.A.C. instruction set to perform operations based on topological properties of patterns loaded into the array such as line thining and outline extraction.

More recently Duff (1973) has constructed an asynchronous array of homogeneous logic elements. The array called CLIP3 differs from A.S.A.P. in two important respects: the current output of the CLIP array element is passed to its nearest neighbours asynchronously without any intervening staticizing register; all CLIP elements perform the same function determined by a broadcast instruction.

A.S.A.P. can be used to investigate P.E. functions giving rise to cyclic activity and also it can be trained to compute localised topological properties by allowing independent P.E. functions. A trainable pattern recogniser based on clustering local topological differences in the training set using the A.S.A.P. structure requiring non-homogeneous P.E. functions is presented by Wilson and Aleksander (1977).

Further development of the high-level language MINIC will be required to control A.S.A.P. since some of the array concepts are not present in MINIC. A MINIC assignment expression causes a synchronous mapping of a networks current state and inputs onto its next state. However, A.S.A.P. has the ability to operate autonomously, searching for a particular state to occur.

APPENDIX A

Honeywell DDP 516 Computer system, Electronics Laboratories University of Kent.

A1 The Central Processor

A Honeywell DDP 516 C.P.U. with the following Honeywell supplied options:

1) Real time clock
2) Priority interrupt network, (16 channels)
3) High Speed arithmetic unit.
4) Direct memory access unit, (1 channel expandable to 4)
5) Memory Lock-out

There is 8K of 16 bit 960nS Honeywell and 8K of 16 bit 900nS Ampex magnetic core memory.

Peripherals:

* * 1 A.S.R. Type 33 10 character/sec Terminal
* * 1 Facit 8 hole paper tape reorder 500 or 1000 c.p.s.
* * 1 Facit paper type punch 150 c.p.s.
* * 1 Magnetic Tape Handler and Controller 18 inches/sec 200, 556 bits/inch
* * 1 Dot display: C.R.O. display of 16 x 16 bit pattern
* 1 Honeywell general purpose 16 bit input/output interface channel.
* * 1 Tally paper tape reader, dual spooling bidirectional 300 c.p.s.

* 1 Data Dynamics paper tape punch 46 c.p.s.

* 1 Case Matrix printer, 80 characters/line, 60 lines/minute.

* 1 Data Dynamics line printer, 96 character/line, 300 lines/minute

1 T.V. Camera system

* 1 Minerva learning network

* 1 Writing Pad, 16 x 16 bit pattern input

1 Radio Astronomy Unit - for steerable microwave dish receiver.

1 Solid State Unit - process control for device fabrication

* 1 VDU and Keyboard 20 lines of 80 characters.

* 1 Analogue Output Interface for X, Y plotter or Storage Oscilloscope

* 1 Disc controller - removable cartridge 1M bit, 16 bit words.

* 2 General purpose interfaces

* Directly supported by ADMOS and available with MINIC system.

A2 <u>The Instruction Set</u>

<u>Glossary of Terms</u>

A - Accumulator register 16 bits

B - Extended accumulator register 16 bits

C - Carry register 1 bit

EA- Effective address 14 bits

I - Instruction register 16 bits

M - Memory buffer register 16 bits

P - Programm counter register 14 bits

X - Index register 16 bits

n - Contents of Address n

sign bit = bit 0, most significant bit

A2.1 Memory Reference Instructions

```
 |F|T|X|X|X|X| |S| |Y|Y|Y|Y|Y|Y|Y|Y|Y|
0     op-code 6    address      15
```

F = indirect address flag, T = index address tag,

S = sector bit, 1 = current sector, 0 = base sector

Instruction	Op-code	Speed	Action
JMP	01	1	EA → {P}
LDA	02	2	{EA} → {A}*
ANA	03	2	{A} ∧ {EA} → {A}
STA	04	2	{A} → {EA}*
ERA	05	2	{A} ⊕ {EA} →{A}
ADD	06	2	{A}+{EA}→{A}carry→{C}*
SUB	07	2	{A}−{EA}→{A}carry→{C}*
JST	10	3	{P}+1→{EA},EA +1→{P}
CAS	11	3	{A}<{EA}{P}+1→{P},{A}= {EA} {P}+2→P,{A}>{EA}{P}+3→P
IRS	12	3	{EA}+1→{EA},{EA}= Ø{P}+2→{P}
IMA	13	2	{A}→{MA},{EA}→{A},{MB}→{EA}
STX	15	3	{X} → {EA}
LDX	35	3	{EA} → {X}
MPY	16	5	{A}x{EA}→{A},{B}, overflow→{C}
DIV	17	11	{A}{B}÷{EA}→{A}, remainder{B}

Instructions marked by '*' operate on 32 bit operands when the C.P.U. is in double-precision mode. Add valued effective addresses are illegal in double precision mode.

A2.2 Input/Output Instructions

```
|X|X|1|1|0|0|| | | | | | | | | |
0    3       7       11      15
  op-code        device address
```

Instruction	Op-code	Speed	Action
OCP	14	3	Output control pulse
SKS	34	3	Test control line, if true {P}+2→{P}
INA	54	3	Input to {A}, {P}+2→{P} if successful
OTA	74	3	Output from {A}, {P}+2→{P} if successful
SMK	74	3	Set interrupt mask from {A}

A2.3 Shift Instructions

```
|0|1|0|0|0|0|X|X|X|X|| | | | | |
0    3       7       11      15
          op-code   two's complement
                    of number of places
```

Instruction	Op-Code	Speed	Action
LRL	0400	1+½n	Long Logical Right Shift
LRS	0401	"	Long Arithmetic Right Shift
LRR	0402	"	Long Right Rotate
LGR	0404	"	Logical Right Shift
ARS	0405	"	Arithmetic Right Shift

ARR	0406	"	Right Rotate
LLL	0410	"	Long Logical Left Shift
LLS	0411	"	Long Arithmetic Left Shift
LLR	0412	"	Long Left Rotate
LGL	0414	"	Logical Left Shift
ALS	0415	"	Arithmetic Left Shift
ALR	0416	"	Left Rotate

Long shifts operate on A and B combined.

A2.4 Generic Instructions

```
 |X|X|0|0|0|0|X|X|X|X|X|X|X|X|X|
 0                             15
               op-code
```

Instruction	Op-code	Speed	Action
HLT	000000	1	Halt
SGL	000005	1	Enter Single Precision
DBL	000007	1	Enter Double Precision
SCA	000041	1	Shift Count →{A}
NRM	000101	1+½n	Normalise Floating Point
IAB	000201	1	$\{A\} \rightleftarrows \{B\}$
ENB	000401	1	Enable Interrupts
ERM	001401	1	Enter Restricted Mode
SKP	100000	1	{P}+2 → {P}
SRC	100001	1	Skip{C}= 0
SR4	100002	1	Skip Switch 4 reset
SR3	100004	1	Skip Switch 3 reset

SR2	100010	1	Skip Switch 2 reset
SR1	100020	1	Skip Switch 1 reset
SSR	100036	1	Skip any Switch reset
SZE	100040	1	Skip {A} = 0
SLZ	100100	1	Skip $\{A_{15}\} = 0$
SPL	100400	1	Skip $\{A_0\} = 0$
NOP	101000	1	No operation
SSC	101001	1	Skip {C} = 1
SS4	101002	1	Skip Switch 4 set
SS3	101004	1	Skip Switch 3 set
SS2	101010	1	Skip Switch 2 set
SS1	101020	1	Skip Switch 1 set
SSS	101036	1	Skip any Switch set
SNZ	101040	1	Skip $\{A\} \neq 0$
SLN	101100	1	Skip $\{A_{15}\} = 1$
SMI	101400	1	Skip $\{A_0\} = 1$
CHS	140040	1	$\sim \{A_0\} \rightarrow \{A_0\}$
CRA	140040	1	$0 \rightarrow \{A\}$
SSP	140100	1	$0 \rightarrow \{A_0\}$
RCB	140200	1	$0 \rightarrow \{C\}$
CSA	140320	1	$\{A_0\} \rightarrow \{C\}, 0 \rightarrow \{A_0\}$
CMA	140401	1	$\sim\{A\} \rightarrow \{A\}$
TCA	140407	1	$\sim\{A\} +1 \rightarrow \{A\}$
SSM	140500	1	$1 \rightarrow \{A_0\}$
SCB	140600	1	$1 \rightarrow \{C\}$
CAR	141044	1	$0 \rightarrow \{A_{8-15}\}$
CAL	141050	1	$0 \rightarrow \{A_{0=7}\}$
ICL	141140	1	$\{A_{0-7}\}\rightarrow\{A_{8-15}\}, 0\rightarrow\{A_{0-7}\}$

AOA	141206	1	$\{A\} + 1 \rightarrow \{A\}$
ACA	141216	1	$\{A\} + \{C\} \rightarrow \{A\}$
ICR	141240	1	$\{A_{8-15}\} \rightarrow \{A_{0-7}\}$, $0 \rightarrow \{A_{8-15}\}$
ICA	141340	1	$\{A_{0-7}\} \rightleftarrows \{A_{8-15}\}$

A3 Indirect Addressing

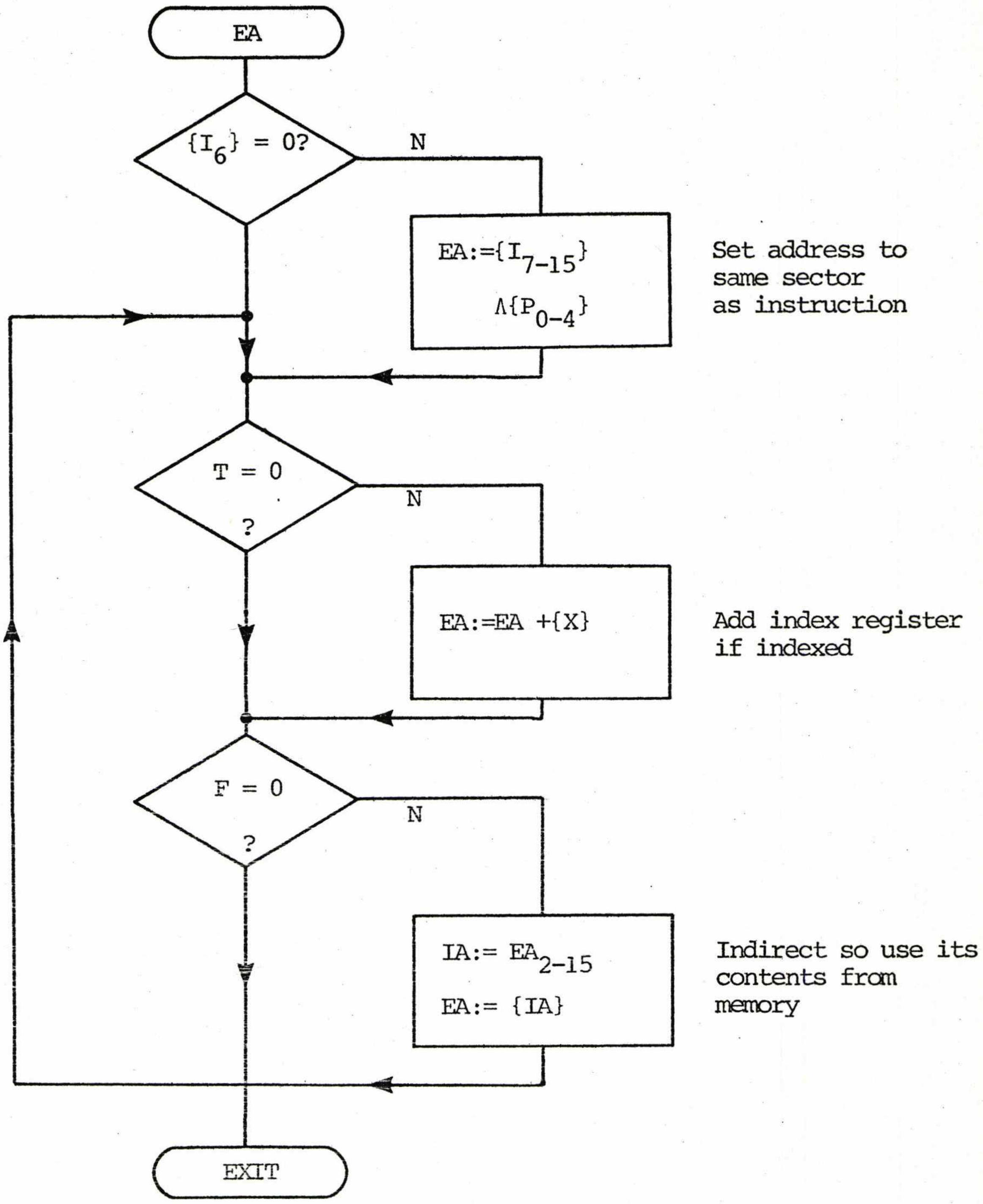
```
|F|T|X|X|X|X|X|X|X|X|X|X|X|X|X|X|
0                                15
              Address
```

Address Constant Format

There is no restriction on the number of levels of indirect addressing. Each level adds 960nS to the total instruction execution time.

The effective address of a machine instruction operand is defined by the following algorithm.

Algorithm for calculating the "effective address":



EA
$\{I_6\} = 0?$
N
$EA:=\{I_{7-15}\} \wedge \{P_{0-4}\}$
Set address to same sector as instruction
$T = 0$?
N
$EA:=EA + \{X\}$
Add index register if indexed
$F = 0$?
N
$IA:= EA_{2-15}$
$EA:= \{IA\}$
Indirect so use its contents from memory
EXIT

A4 The DAP-16 Assembly Language

DAP-16 is a symbolic assembly language based on the mnemonics for identifying machine instructions.

Instruction syntax:

<Label> <op code> <address field> <comments>

Labels: Alphabetic character followed by up to three alpha numeric characters:

L1, GEXC,

Op code : Any instruction mnemonic or DAP pseudo-operation

D.A.P. Pseudo-operations:

DAC	Define Address constant
DEC	Define Decimal literal
OCT	Define Octal literal
BSS n	Reserve n storage locations
PZE	Put zero in op-code
CALL	Call external subroutine

Indirect Addressing: A memory reference or DAC pseudo-operation may be made indirect by a * post-fix to the op code

e.g. LDA* BUFA

Address Field: This may be an expression containing a label, a literal or an absolute address.

```
e.g.        JMP    L1+5          ; Evaluates using address of L1
            LDA    =10           ; Assembler generates a literal =10
            STA    1000          ; Absolute address
```

* when it appears in the address field,is the current value of the location counter.

** Means the address field is undefined.

Indexing: Indexing is denoted by a , 1 post fix to the address field

```
e.g.        JMP*   *, 1       ; Dispatches control to an address
            DAC SUB1          ; Given by an index of the following
            DAC SUB2          ; table
            DAC SUB3
```

A5 Loading

The DAP-16 assembler program requires two passes of the source program during which it defines forward referenced addresses. The output of the assembler program is a corresponding object program in which information about the address field of instructions is encoded so as to be relocatable. A Loader program is used to read in the object program for execution. This calculates absolute addresses for all the labels of the program and automatically converts instructions to indirect ones generating linking addresses in the base sector for all instructions accessing addresses outside of their own sectors. The loader also performs the loading and linking of any external subroutires called by the source program.

APPENDIX B

B1 The ADMOS Operating System

This is a single user operating system designed to provide a device independent environment for the Honeywell DDP516 central processor and the peripheral devices described in Appendix A. A complete description of its design philosophy and implementation is given by Ball (1974).

B2 ADMOS File Structure

There are four channels each for input and output. Channel one is shared with the ADMOS system for the input of commands and the reporting of messages. Files are of two types, character and binary. Character files are organised as strings of 8 bit bytes with the most significant bit of each byte set. They are terminated by a 'cntrl C', 203_8, character.

Binary files consist of strings of 16 bit words and can only be terminated by operator action e.g. program break or under program control e.g. C bit set.

On reading, both types of files ignore leading zero words or bytes. In the case of binary files this means some sort of synchronising convention must be used. When transfering binary files to and from byte orientated devices e.g. paper tape station, the most significant byte of a word is transfered first.

Devices are considered as having structured or non-structured files according to their nature. A non-structured file, such as used by a paper tape station, is created and read incrementally and has no unique file labelling. A structured file, such as a disc or magnetic tape file, is created block by block and is uniquely defined in a directory.

B3 Global Commands

A global command is requested by ADMOS when it has completed the current process or a sub-system is making use of the "input command line", A$IC utility. They are input via Channel 1. If output channel one is assigned to a terminal, an ampersand, &, is output as a command level prompt. Global commands may be requests to use ADMOS utilities or instructions to load and run ADMOS subsystems. They consist of a command word or mnemonic followed by arguments. Arguments may be system parameters or data-result specifications (drspec). Fields in the command line are separated by spaces.

B 3.0 Data-result Specification

A data-result specification consist of a list of channel specification, device and file specification pairs. Non-structural file specifications are simply a mnemonic for the device. A different mnemonic is used to differentiate between binary and character files. The standard device

mnemonics are shown in Table B1. Structured files are specified by a file number in the case of magnetic tape or a four character-user-identifier followed by a file number for disc files. Input Channel 2 and output Channel 2 are also represented by FROM and TO respectively.
Examples of data-result specifications are as follows:

```
I3 DC RJG 10 03 LP

FROM PB TO MC 10 I4 DB TEMP 1

I1 PC                 ; Get commands from paper tape.
```

B 3.1 Command Mnemonics

AC <drspec>	Assign Channels - performs channel assignments as specified in a following drspec.
COPY <drspec>	Initiates a character file transfer between I2 and 02 using the devices specified in the drspec.
BCOPY <drspec>	Initiates a binary file transfer between I2 and 02 using the devices specified in the drspec.
DELDC <disc file spec>	Deletes the specified file from the disc and returns the space to the free area.

DOT <address>	Displays the next sixteen consecutive address from <address> on the C.R.T. display matrix.
JMP <address>	Transfers control to the location given by <address> .
JOB	Resets all default values and clears the user workspace.
PL	Punches leader tape on the high speed punch.
SL	Obtains a self loading systems program from Channel I4.
VFY	Initiates simultaneous input on Channels I2 and I3 verifying frame by frame.

B 3.2 ADMOS Subsystem Calls

Subsystems can be loaded using the SL command but the following have been included as direct ADMOS commands.

DAP	DDP516 assembler program
LOAD	DDP516 object loader and linker program
EDIT	EDIT AGB Text editor
BASIC	BASIC-AGB interpreter
MINIC	Current version MINIC compiler
L4	4k object loader for loading large programs (overwrites ADMOS).

LLST	Object file library editor and linker
PAL	Self loading systems file generator
DEBUG	On line debugging program
PROBE	Disc file directory enquiry program
ML1	General purpose Macro processor.

B 4 ADMOS Utility Routines

Certain of the routines used internally by ADMOS are available to external users programs by a call to a dispatching routine A$ER followed by an argument. The call may be followed by other argument depending upon the nature of the routine.

```
e.g.           JST    A$ER
               DEC     n
```

where n defines the utility routine required.

The utility routines available are listed as follows: ones marked with an asterisk are used in the MINIC system.

Routine	Parameter	
*A$IN	1	Initialize dedicated peripherals
*A$WT	2	Wait until output buffers empty
*A$IC	3	Input command line
*A$DC	4	Decode string matches
*A$OM	5	Output systems message string
*A$MN	6	Output systems message mnemonic
*A$DN	7	Output systems message decimal no.
*A$ON	8	Output systems message octal no.

*A$TF	9	Decode TO-FROM and assign channels
A$AC	10	Assign channels according to calling program
*A$EB	11	Enable and process program breaks
A$MV	12	Locate magnetic tape file
A$CM	13	Set up subsystem for common-mode operation
*A$ID	14	Input data line
*LOGR	15	Force logical error
A$FB	16	Force program break
A$DR	17	Output decimal result
A$OR	18	Output octal result
A$MR	19	Output string-result
A$D	20	Disc block transfer
A$GC	21	Execute global command from calling program.

Routine A$DC is used extensively by the MINIC sub-system for checking string matches. The string to the right of the command pointer A$CP is matched against entries in a user defined table and a numerical decode returned if a match is obtained. Two modes of matching are allowed using the subsets flag A$SF. When this is set, the string to be matched may be the subset of a field, otherwise it must be enclosed by spaces. Decoding of decimal and octal integers is also accomplished by A$DC.

B 4.1 ADMOS Utility Locations

The utility locations enable communication between the user program and the system parameters. Those preceded by an asterisk are used by the MINIC sub-system.

*A$TL	-	Highest location of ADMOS Links
*A$SF	-	Decode string sub-sets flag
*A$CC	-	Command prompt character
*A$CB	-	Command buffer origin
*A$CP	-	Command pointer
*A$DT	-	Dot display
*A$EP	-	Program break return entry point
*A$IF	-	Program breaks inhibit flag
*A$RC	-	Input data line channel
A$UI	-	Address of user interrupt service routine
*A$TF	-	Address of lowest location used by ADMOS
MSK0 MSK1 MSK6	-	Peripheral device interface interrupt masks

B 4.2 Input & Output Routines

There are two routines for input and output, A$IP and A$OP. They are called as follows:

```
INPUT  JST  A$IP
       DEC  <channel no.>
       <argin {A}>

OUTPUT <argin {A}>
       JST  A$OP
       DEC <channel no.>
```

In the case of channels assigned to character devices the transfer takes place between the least significant 8 bit of the accummulator and the peripheral device. Character files are closed by a 203_8 character being transfered and binary files by the C bit being set on entry or exit from the appropriate routine. When a structured file has been closed attempts at further transfer result in an unassigned channel, UC mnemonic, error message.

B 5 ADMOS Links

The ADMOS system occupies the top 9 sectors (4½k) of memory of the DDP516. Communication between ADMOS and the user program is via the ADMOS links which reside in the users workspace or in locations below 104_8. The links form part of the object library to be loaded together with the object of the user program by the LOAD sub-system. They must be the last objects loaded in order that they occupy the highest location of the user system. Locations A$TL and A$TF can then be used to set the limits of the workspace available to the user system. As well as defining the utility locations the links define the entry points to the routines AIP, AOP, A$ER, and A$LG.

Table B1

Device	Character	Binary
Command terminal Teletype	TT	-
Facit paper tape reader	PC	PB
Facit paper tape punch	PC	PB
Magnetic Tape	MC	MB
Line printer	LP	-
Case matrix printer	FP	-
Disc	DC	DB
Vista V.D.U.	VT	-

Table B1 continued

General purpose interface 1	-	G1
General purpose interface 2	-	G2

APPENDIX C

Examples of Target Code produced by the MINIC compiler.

C1 GOTO

C1.1 Backward referenced in same sector:

```
(1)              JMP   BREF     ; direct jump
Locations: 1           Execution time: 1 cycle
```

C1.2 Forward referenced and backward referenced in a previous sector:

```
(1)              JMP*  FRAD     ; indirect jump
(2)      FRAD    DAC   FREF     ; indirect link address
Locations: 2     Execution time: 2 cycles
```

C2 On <exp> GOTO L1, L2, ... Ln

```
                 eval. <exp>
(1)              JST*  IFXT     ; convert to integer
(2)              CRA            ; less than one return
(3)              CAS   =0       ; within range?
(4)              CAS   UB       ; upper bound?
(5)              JMP   OUT      ; No, exit line
(6)              JMP   OUT
(7)              STA   0        ; dispatch to line
(8)              JMP*  *,1      ; via indexed table
                 DAC   L1       ; address of line 1 etc.
                 DAC   L2
```

```
                DAC   Ln
        OUT     ***   **

(9)     UB      DEC   n+1
(10)    ZERO    DEC   0
```

Locations: <exp> + n + 10, Execution time: ~<exp> + 71 cycles.

C3 GOSUB <line number>

```
(1)             JST*  GOSI     ; stack return, dispatch
(2)             DAC   Ln
```

Locations: 2 , Execution time: 18 cycles

C4 RETURN

```
(1)             JMP*  RETI     ; unstack, return
```

Locations: 1 , Execution time: 14 cycles

C5 DISPLAY

C5.1 Unsubscripted:

```
(1)             LDA   {CVA+2}
(2)             STA   A$DT     ; start of display block

                DEC   CVA + 2
```

Locations: 3 , Execution time: 4 cycles

C5.2 Subscripted: <eval. subscript> ; since CVA is required in {A}

```
(1)             ADD   =Z      ; Uses two less locations than normal
(2)             STA   ASDT    ; subscript evaluation
(3)             DEC   2
```

Locations: <subs - 2> + 3 , Execution time: <subs -4>+4 cycles.

C6 FOR

C6.1 FOR X = Y to Z (Default STEP to 1)

```
(1)             DBL           ; calculate initial
(2)             DLD*  XAD     ; values for control
(3)             DST*  XAD     ; variable
(4)             DLD*  ZAD
(5)             DST   TZ
(6)             DLD*  XAD
(7)     LOOP    JST*  FSUB    ; compare with loop limit
(8)             DAC   TZ
(9)             SGL
(10)            SZE
(11)            SPL
(12)            SKP
(13)            JMP   CONT    ; skip out of FOR loop
(14)    YAD     DAC   Y
(15)    XAD     DAC   X
(16)    XAD     DAC   Z
(17)    TZ      PZE
```

Locations: 17 , Execution time:

first time loop = 188 cycles

subsequent loops = 167 cycles

C6.2 FOR X = Y to Z STEP W.

```
(1)              DBL               ; initialise control
(2)              DLD*  YAD         ; variable
(3)              DST*  XAD
(4)              DLD*  ZAD
(5)              DST   TZ
(6)              DLD*  WAD
(7)              DST   TW
(8)              DLD*  XAD
(9)       LOOP   JST*  FSUB
(10)             DAC   TZ
(11)             JST*  FMPY
(12)             DAC   TW
(13)             SGL
(14)             SZE
(15)             SPL
(16)             SKP
(17)             JMP   CONT
                  '
                  '
                  '
(18)      YAD    DAC   Y
(19)      XAD    DAC   X
(20)      ZAD    DAC   Z
(21)      TW     PZE
(22)             PZE
(23)      TZ     PZE
(24)             PZE
```

Locations:- <exp> + 22 , Execution time:-

first time loop = <exp> + 415 cycles.

subsequent loops = 397 cycles

C7. NEXT X

```
                DBL
                DLD*  XAD
                JST*  FADD
                DAC   TW          ; default TW to FONE
                DST*  XAD
                JMP   LOOP
        CONT    ***   **
        FONE    DPB   IEO
```

Locations:- 8 , Execution time:- 168 cycles

C8. LET

C8.1 LET X = Y

```
                DBL               ; simple double precision
                DLD*  YAD         ; assignment
                DST*  XAD
                SGL
```

Locations:- 6 , Execution time:- 10 cycles

C8.2 LET X_1= <expl>, X_2=<exp 2>, ... X_n =<expn>

```
          DBL              ; multiple assignment
          <expl>           ; optimises CPU mode
          DST*   X1AD
          <expn>
          DST*   XnAD
          SGL
```

Locations:- $2 + n + \sum^{n}$<expi> , Execution time:- $2 + 4n + \sum^{n}$<expi> cycles

C8.3 LET X = A+B*(C-D)

```
          DBL              ; no temporary variables
          DLD*   CAD       ; required
          JST*   FSUB
          DAC    D
          JST*   FMPY
          DAC    B
          JST*   FADD
          DAC    A
          DST*   XAD
          SGL
```

Locations:- 12 , Execution time:- 465 cycles

C8.4 LET X = EXP(N*LOG(N) + N)

```
          DBL              ; Calls Mathpak Library
          DLD*   NAD       ; Functions
          JST*   LOG
          JST*   FMPY
```

```
                DAC    N
                JST*   FADD
                DAC    N
                JST*   EXP
                DST*   XAD
                SGL
```

Locations:- 12 , Execution time:- 9309 cycles.

C8.5 LET PAT Y BE PAT X

```
                LDX    PTSZ      ; size of expression
        LOOP    LDA*   PXAD      ; to index
                STA*   PYAD
                IRS    0         ; loop until index = 0
                JMP    LOOP

        PTSZ    DEC    <-n>      ; n=no. of words in pattern variables
        PXAD    DAC    X+n+2,1   ; post-index all pattern addresses
        PYAD    DAC    Y+n+2,1
```

Locations:- 8 , Execution time:- 4+9n

For a 16 x 16 bit pattern expression, n = 16, execution time=148cycles

C8.6 LET PAT Y BE PAT X OR PATZ AN NT PAT F EX PAT G

```
                LDX    PTSZ      ; size of expression to index
        LOOP    LDA*   PFAD
                CMA              ; NT
                ERA*   PGAD      ; EX
                ANA*   PZAD      ; AN
                JST*  ORS       ; OR
                DAC   PATX,1
                STA*  PYAD
                IRS   0         ; loop until index = 0
                JMP   LOOP
                 .
                 .
                 .
        PTSZ    DEC   - exp size
        PFAC    DAC   PATF,1
        PGAL    DAC   PATG,1
        PZAD    DAC   PATZ,1
        PYAD    DAC   PATY,1

Locations:- 15                 , Execution time:- 4 + 40n
```

C8.7 LET PAT Y BE PAT X AN NT PAT Z BY MAP M OR PAT W BY MAP M

```
                LDX   PTSZ      ; size of expression to index
        LOOP    JST*  MPSB      ; BY
                DAC   Z
                DAC   M,1
                CMA             ; NT
                ANA*  PATX      ; AN
                STA   T         ; temporary variable, not indexed
                JST*  MPSB      ; BY
                DAC   W
                DAC   M,1
                JST*  ORS      , OR
                DAC   T
                STA*  YAD
                IRS   0          ; loop until index = 0
                JMP   LOOP
                 .
                 .
                 .
        PTSZ    DEC   <exp size>
        YAD     DAC   Y,1
        T       PZE
```

Locations:- 18 , Execution time:- 4 + 1396n

C8.8 LET PAT X BE NET N IP PAT Y

```
(1)             LDX   PXSZ       ; outer loop expression size
(2)             LDA*  NAD        ; address of first card
(3)             JST*  MOAS       ; send to MINERVA addr. reg.
(4)             LDA   PYSZ       ; net IP expression size
(5)             STA   T1
(6)     L1      STX   T2         ; same outer loop index
(7)             LDA   =-4
(8)             STA   T3         ; counter for net response
(9)             CRA              ; initialise net response variable
(10)    L2      LGL   4          ; prepare for next net response
(11)            STA   T4         ; net response variable
(12)            LDX   T1         ; IP loop count to index
(13)            LDA*  PYAD
(14)            JST*  MINI       ; send to MINERVA, get resp.
(15)            ERA   T4         ; accumulate net response variable
(16)            IRS   T1         ; bump IP loop count
(17)            NOP
(18)            IRS   T3         ; loop until 16 bit response
```

```
(19)            JMP   L2
(20)            LDX   T2        ; complete outer loop
(21)            STA*  PXAD
(22)            IRS   0         ; continue until outer loop done
(23)            JMP   L1
                 .
                 .
                 .
(24)    PXAD    DAC   X,1
(25)    PYAD    DAC   Y,1
(26)    NAD     DAC   N
(27)    PXSZ    DEC   <net exp.size>
(28)    PYSZ    DEC   <ip exp. size>
(29)    T1      PZE
(30)    T2      PZE
(31)    T3      PZE
(32)    T4      PZE
```

Locations:- 33   , Execution time:- 22 + 186n

C8.9   LET NET N IP PAT X BE PAT Y

```
(1)             LDA*  NAD       ; initialise MINERVA
(2)             JST*  MOAS      ; card address register
(3)             LDX   PXSZ      ; IP expression size to index
(4)     L1      LDA*  XAD       ; get training data
(5)             JST*  MINT      ; and send to MINERVA
(6)             JST*  TECS      ; accesses teach data and
(7)             DAC   Y         ; teach clocks
(8)             IRS   0         ; loop until IP expression done
```

```
(9)             JMP   L1
                 ⋮
(10)   NAD      DAC   N
(11)   XAD      DAC   X,1
(12)   PXSZ     DEC   -<ip exp. size>
```

Locations:- 12 , Execution time:- 17 + 83n

C8.10 LET A = SUM PAT X

```
(1)             LDX   PXSZ    ; pattern expression size to index
(2)             CRA           ; initialise SUM temporary variable
(3)    L1       STA   T
(4)             LDA*  XAD
(5)             SZE           ; no. of bits is zero anyway
(6)             JST*  PSZS    ; get no. of bits in {A}
(7)             ADD   T       ; accumulate SUM temporary variable
(8)             IRS   0       ; complete pattern expression
(9)             JMP   L1
(10)            JST*  FLOT    ; integer to floating point
(11)            DST*  AAD
(12)            SGL
(13)             ⋮
(14)   PXSZ     DEC   -<pattern size>
(15)   XAD      DAC   X,1
(16)   AAD      DAC   A
(17)   T        PZE
```

Locations:- 16 , Execution time:- 87 + 116n average.

C8.11 Subscripted Variables

LET X = Y(A)

```
(1)             LDA   YDVA     ; dope vector address of Y
(2)             STA   T
(3)             DBL            ; subscript evaluation
(4)             LDA*  AAD      ; value of subscript
(5)             JST*  IFXT     ; convert to integer
(6)             CRA            ; 0<n<1 return
(7)             CAS*  T        ; normal return
(8)             JMP   OBER     ; Out of Bounds
(9)             NOP
(10)            LGL   1        ; two words per element
(11)            ADD   YAD      ; base address of Y
(12)    S1      STA   YSL      ; sector link for Y element
(13)            DBL
(14)            LDA*  YSL
(15)            STA*  XAD
(16)            SGL
                 :
(17)    YAD     DAC   Y
(18)    YSL     DAC   **
(19)    XAD     DAC   X
(20)    T       PZE
(21)    YDVA    DAC   YDV
```

Locations:- 21 , Execution time:- 87

Generally each singly subscripted real variable requires (12 + subscript expression) locations and increases execution time by (76 + <subscript expression>) cycles.

C8.12 LET X = Y(A, B, C)

```
(1)     LDA   YDVA   ; pointer to Y dope vector
(2)     STA   T1     ; put in temporary
(3)     DBL          ; First dimension
(4)     LDA*  AAD    ; evaluate first subscript
(5)     JST*  IFXT   ; convert to integer
(6)     CRA          ; less than one return
(7)     CAS*  T1     ; compare with first bound
(8)     JMP   OBER   ; out of bounds
(9)     NDP
(10)    STA   T2     ; first subscript
(11)    IRS   T1     ; next dope vector element
(12)    DBL          ; Second dimension
(13)    LDA*  BAD    ; evaluate second subscript
(14)    JST*  IFXT   ; convert to integer
(15)    CRA          ; less than one return
(16)    CAS*  T1     ; compare with 2nd bound
(17)    JMP   OBER   ; out of bounds
(18)    NOP
(19)    IRS   T1     ; next dope vector element
(20)    MPY*  T1     ; offset over 1st bound
(21)    IAB
(22)    ADD   T2     ; sum offsets
(23)    STA   T2
```

```
(24)            IRS   T1      ; next bound
(25)            DBL           ; Third Dimension
(26)            LDA*  CAD     ; evaluate 3rd subscript
(27)            JST*  IFXT    ; convert to integer
(28)            CRA           ; less than one
(29)            CAS*  T1      ; compare with 3rd bound
(30)            JMP   OBER    ; out of bounds
(31)            NOP
(32)            IRS   T1      ; next dope vector element
(33)            MPY*  T1      ; offset for 3rd bound
(34)            IAB
(35)            ADD   T2      ; sum offsets
(36)            LGL   1       ; X2 for red
(37)            ADD   YAD     ; add base
(38)            STA   YSL     ; indirect sector link
(39)            DBL           ; Evaluate expression
(40)            LDA*  YSL     ; get array element
(41)            STA*  XAD
(42)            SGL
                 .
                 .
                 .
(43)     YAD    DAC   Y
(44)     YSL    DAC   **
(45)     XAD    DAC   X
(46)     AAD    DAC   A
(47)     BAD    DAC   B
(48)     CAD    DAC   C
(49)     YDVA   DAC   YDV
(50)     T1     PZE
(51)     T2     PZE
```

Locations:- 51 , Execution time:- 259

Generally each extra subscript dimension requires (10 + subscript expression) locations and increases execution time by (81 + subscript expression) cycles.

C8.13 LET PAT X (A) BE PAT Y BY MAP M (N)

```
(1)      LDA    XDVA   ; pointer to dope vector.
(2)      STA    T      ; put in temporary
(3)      DBL
(4)      LDA*   AAD    ; evaluate subscript
(5)      JST*   IFXT   ; convert to integer
(6)      CRA           ; less than one return
(7)      CAS*   T      ; compare with bound
(8)      JMP    OBER   ; out of bounds
(9)      NOP
(10)     MPY    PXSZ   ; calculate offset
(11)     IAB
(12)     ADD    XAD    ; add base address indexed
(13)     STA    XSL    ; indirect sector link
(14)     LDA    MDVA   ; dope vector to MAP M
(15)     STA    T      ; put in temporary
(16)     DBL
(17)     LDA*   NAD    ; evaluate subscript
(18)     JST*   IFXT   ; convert to integer
(19)     CRA           ; less than one return
(20)     CAS*   T      ; compare with bound
(21)     JMP    OBER   ; out ot bounds
(22)     NOP
```

```
                MPY   MMSZ    ; calculate offset
                IAB
                ADD   MAD     ; add base address indexed
                STA   MSL     ; indirect sector link
                LDX   PESZ    ; initialise pattern size
        LOOP    JST*  MPSB    ; loop to evaluate PAT expression
                DAC   Y
        MSL     DAC   **,1    ; put M sector link haer
                STA*  XSL     ; put result in X array
                IRS   0
                JMP   LOOP
        PESZ    DEC   -<expression size>
        XDVA    DAC   XDV
        MDVA    DAC   MDV
        AAD     DAC   A
        NAD     DAC   N
        XSL     DAC   **
        XAD     DAC   X,1
        T       PZE
        PXSZ    DEC   <PAT Y size>
        MMSZ    DEC   <MAP M size>
```

Locations:- 43 , Execution time:- 169 + 683n

C8.14 LET R2 = SUM NET N (A) IP PAT I BY MAP M

The target code for this subscripted NET variable expression is given in Section 5.3.13

Locations:- 51 , Execution time:- 187 + 2984n

C9 The IF Statement

Generally, after evaluation of the conditional expression control is passed to the location immediately following the evaluation if true, or one location higher if fales.

IF <cond, exp.> THEN <cond true stmt>

for <cond. true stmt.> = GOTO or RETURN

```
                 <cond. exp>
                GOTO or RETURN
                 <next line>
```

for <cond, true stmt.> $\neq$ GOTO or RETURN

```
                 <cond. exp.>
                  SKP
                  JMP  FLSE
                 <cond. true stmt.>
       FLSE      <next line>
```

C9.1 IF SS1 THEN GOTO <N>

$$\begin{tikzcd}
 & \text{SS1} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{JMP N} \arrow[dr, dash] & & \text{JMP* NAD} \arrow[dl, dash] \\
 & \vdots &
\end{tikzcd}$$

; Branch if false

; Indirect if forward ref.

; or out of sector

Locations, 2 or 3 , Execution time:- False: 1

True: 1 or 2

C9.2 IF PAT X = PAT Y THEN RETURN

```
                 LDX    PXSZ
         LOOP    LDA*   YAD
                 ERA*   XAD     ; test for equivalence
                 SZE
                 JMP    FLSE
                 IRS    0
                 JMP    LOOP
                 JST*   RETI
         FLSE    ***    **

         PXSZ    DEC    - <pat X size>
         YAD     DAC    Y,1
         XAD     DAC    X,1
```

Locations:- 11 , Execution time:- 11, min.(false)
18+11n, max.(true)

C9.3 IF <num. expl> <op 1> <num. exp2> THEN <cond true stmt.>
where <opl> is <>, < =, >, or =

```
                 <num expl >
                 DST    T
                 <num exp2>
                 JST*   FSUB
                 DAC    T
                 SGL
         SZE, SNZ,  SPL, SMI       ; = , <> ,< =, >
```

```
                        <true>
                        <false>

                T       PZE
                        PZE

C9.4    <opl> = <,  >=

                        <num. expl>
                        DST   T
                        <num. exp2>
                        JST*  FSUB
                        DAC   T
                        SGL
                        CAS   =0
                        JMP   *+3
                        NOP
                        (SKP)            ; only when  opl  is
                        <true>
                        <false>

C10     PRINT   2; "Y=", Y1; FORMAT $01(16,16) PAT Y

                        LDA   =2
                        STA   OPCH    ; output channel
                        JST*  PQUT    ; print quote
                        DAC   QSTR    ; points to s-string quote
                DBL
                DLD*  Y1AD
                JST*  PRNT    ; print real number
                JST*  PCOM    ; comma, move print-head
                LDX   PYSZ    ; pattern expression
        LOOP    LDA*  YAD
                JST*  PPAT    ; interprets FORMAT
                DAC   FSTR    ; points to FORMAT s-string
                IRS   0
                JMP   LOOP
                JST*  PNL     ; New line

        Y1AD    DAC   Y1
        PYSZ    DEC   -<pat. Y size>
        YAD     DAC   Y,1
```

Locations:- 19 , Execution time:- device dependant.

C11 <u>OUTPUT</u>

The PRINT and OUTPUT target code generators are identical except that calls are made to different sets of run-time routines. The sequence of generated code will therefore be identical for both PRINT and OUTPUT statements.

C12 DRSPEC I3 DB SYS 36 02 LP

```
                JST*  DRSI    ; calls ADMOS assign routines
                DAC   DSRR    ; points to s-string.
```

Locations:- 2 , Execution time:- ADMOS dependant.

C13 END or STOP

```
JST*    END1        ; prints END message

DEC     <line no.>; returns to CMEP
```

Locations 2 , Execution time:- dependant upon command terminal device.

APPENDIX D

MINIC Runtime Routines

D1 'Mathpak' routines

Calling sequence:

```
arg 1 in {A}
JST* <routine link>
DAC <arg 2>
new arg in {A}
```

or for single argument routines:

```
arg in {A}
JST* <routine link>
new arg in {A}
```

A dedicated routine address link is declared in sector zero for each routine entry point. The mode of the c.p.u. e.g. SGL or DBL is set on exit according to the context of the argument.

Routine	Function	Size	Execution Time (average)
ABS	Returns absolute value $\|A\|$	4	10
ATN	Returns arctangent	54	3710
COS	Returns cosine (requires sine routine)	8	3411
EXP	Returns exponential, e^x	44	4281
FADD	Floating point add	57	195
FDIV	Floating point divide	57	330
FEXP	Raise Floating point to a power	62	9489, 1631*

FNEG	Negate Floating point	10	9
FMPY	Floating point multiply	28	247
FSUB	Floating point subtract (requires FADD)	6	198
IFXT	Floating point to integer conversion	19	57
IFXQ	Convert if integer from floating point (requires IFXT)	11	354
FLOT	Integer to floating point conversion	8	75
LOG	Returns natural logarithm	49	4560
INT	Computes integer part of floating poing	32	86
RND	Pseudo-random sequence generator	18	96
SGN	Returns - IEO or +IEO according to sign	8	8
SIN	Returns sine (arg in radians	47	3209
SQR	Returns square root	40	2376
TAN	Returns tangent, (uses SIN and COS)	9	6970

* Exponent is an integer i, $0 < i < 100$

D2 Pattern Expression Routines

Routine	Function	Size	Av. Speed
ORS	Computes Logical OR of arg. and {A}	13	20
MPSB	Maps next 16 bits from pattern arg. according to {X}	62	692 max^m 660 min^m
MINT	Minerva teach data output routine	5	8
MOAS	Output initial Minerva card address	5	8
TECS	Get teach pattern and teach clock routine	27	60
MINS	Minerva card data input and output routine	10	17
PSZS	Gets no. of bits in {A}	23	100

D3 Input and Output Routines

The execution speed of these routines is determined by the device transfer rate. Since all input andoutput is timeshared by ADMOS quantitative values would be misleading.

Routine	Function	Size
IPNO	Input floating-point number for device buffer	16
IPPT	Input pattern variable from device buffer	46
IPMP	Input map variable from device buffer	41
IPDR	Get next data field from command buffer, input new line if empty	20
PNL	Outputs new-line in PRINT statement	10
OEOF	Closes binary file in OUTPUT statement	4
PQUT	Print quote string	12
OQUT	Outputs ASCII string to binary file	24
PRNT	Print f.p.number as ASCII string	171
OPNT	Convert f.p. to integer and output to binary file	6
PPAT	Print pattern in ASCII format	47
OPAT	Output pattern in binary format	4
PCOM	Format printing into 13 position columns	11
OCOM	Output 5 zero words to binary file	9
TABS	Prints spaces until a specified column	17
OTBS	Outputs specified no. of zero words	13
OPCH	A location which defines output channel	-
IPCH	A location which defines input channel	-

D4 Miscellaneous Runtime Routines

Routine	Function	Size	Execution time
DRSI	DRSPEC interpreter	7	-
GOSI	GOSUB handler	7	14
RETI	RETURN handler	6	12
ENDI	END and STOP handler	8	-
EXFC	Decode ASCII f.p. constant	86	~700/digit +250/expone

Execution times are given as DDP 516 instruction clock cycles.

One clock cycle = 0.96 μS.

APPENDIX E

Network Hardware Control Micro-orders.

CCAR Assembler directive: OCP '1242

Clear Card Address Register.

This sets the card address register to zero and is equivalent to loading it with the address of the first Network Element Card, (N.E.C.).

ICAR Assembler directive: OCP '1342

Increment Card Address Register

Increments the current value of the card address register by one.

LCAR Assembler directive: OCP '1142

Load Card Address Register

This loads the card address register with the least significant 8 bits of the computer output interface register.

LNDR Assembler directive: OCP '1642

Load Network Data Register

This enables the contents of the computer output interface register to the Network Data Register (N.D.R.) and thence to the data inputs of the N.E.C.'s via the Network Data Bus (N.D.B.)

RAC1 Assembler directive: OCP '0242

Reset All Cards layer 1

This simultaneously resets all the layer 1 S.L.A.M.-16 stores.

RAC2 Assembler directive: OCP'0642

Reset All Cards layer 2

Substitute layer 2 as above.

RTDR Assembler directive: OCP '1042

Reset Teach Data Register

This resets the 1 bit Teach Data Register. Subsequent Teach Enable micro-orders cause a 'zero' to be associated with the current input.

SAC1 Assembler directive: OCP '0142

Set All Cards Layer 1

This simultaneously sets all the layer 1 S.L.A.M.-16 stores.

SAC2 Assembler directive: OCP '0342

Set All Cards Layer 2

Simultaneously sets all Layer 2 S.L.A.M.-16 stores.

STDR Assembler directive: OCP '0742

Set Teach Data Register

This sets the 1 bit Teach Data Register. Subsequent Teach Enable micro-orders cause a 'one' to be associated with the current input.

SCCM Assembler directive: OCP '1442

Set Computer Controlled Mode

Sets mode flip-flop such that subsequent data and micro-orders are sourced from the computer interface.

<u>TE1</u> Assembler directive: OCP '0442

<u>T</u>each <u>E</u>nable layer <u>1</u>

This causes the layer 1 S.L.A.M.-16's of the N.E.C. being addressed by C.A.R. to associate the bit pattern present on N.D. bus with the contents of the T.D. register

<u>TE2</u> Assembler directive: OCP '0542

<u>T</u>each <u>E</u>nable Layer <u>2</u>

This causes the layer 2 S.L.A.M. of the N.E. Card currently addressed by C.A. register to associate the responses of the layer 1 S.L.A.M.'s to the bit pattern present on N.D. bus, with the value of the T.D. Register.

APPENDIX F

Internal codes and precedence functions used in the MINIC s-language

Symbol	Code 8
LET	1
FOR	2
PRINT	3
OUTPUT	4
GOTO	5
IF	6
INPUT	7
GOSUB	10
READ	11
DRSPEC	12
DIM	13
DATA	14
DEF	15
DISPLAY	16
PAD	17
RETURN	20
RESTORE	21
REM	22
ON	23
NEXT	24
STOP	25
END	25
THEN	26
STEP	27
TO	30
FORMAT	31

Symbol		Code 8
TAB		32
<>		41
><		41
>=		42
=>		42
<		43
=		44
>		45
<=		46
=<	1)	46
=		47
+		51
-		52
*		53
/		54
↑		55
)		56
BE		60
IP		61
AN		62
EX		63
OR		64
BY		65
SUM		37

Symbol	Code 8
SIN	71
COS	72
TAN	73
ATN	74
SQR	75
EXP	76
ABS	77
INT	100
RND	101
SGE	102
LOG	103
+ 2)	0
- 2)	104

Symbol	Code 8
(	105
NT	106
SS1	121
SS2	122
SS3	123
SS4	124
SSS	125
SR1	126
SR2	127
SR3	130
SR4	131
SSR	132

1) Assignment operator.

2) Unary operators.

APPENDIX G

B.N.F. Description of MINIC

1. <MINIC PROGRAM>::= <immediate statement>|(<statement>)$^{n}_{1}$

2. <immediate statement>::=<statement body>

3. <statement>::=<line no><statement body>

4. <statement body>::=<LET STATEMENT>|<GOTO STMT>|<ON STMT >
   |<PRINT STMT>|<INPUT STMT>|<OUTPUT STMT>|
   <FOR STMT>|<NEXT STMT>|<IF STMT>|<PAD STMT>
   |<DISPLAY STMT>|<GOSUB STMT>|<RETURN STMT>|<REM STMT>
   |<DRSPEC STMT>|<READ STMT>|<DIM STMT>
   |<DATA STMT>|<RESTORE STMT>

5. <GOTO STMT>::= GOTO<line number>

6. <GOSUB STMT>::= GOSUB<line number>

7. <ON STMT>::= ON<num exp> GOTO <no list>

8. <STOP STMT>::= STOP|END

9. <PAD STMT>::= PAD <pat var >

10. <NEXT STMT>::= NEXT <num var >

11. <RETURN STMT>::= RETURN

12. <DISPLAY STMT>::= DISPLAY <pat var>

13. <LET STMT>::= LET <assignment part><MINIC expression>

14. <assignment part>::= <num var >=|(<pat var>|<net part>)BE

15. <MINIC exp >::= <arith exp >|<pattern exp>

16. <arith exp>::= <arith term>[{+/-}<arith term>]$^{n}_{0}$

17. <arith term>::= <arith factor>[{*|/}<arith factor>]$^{n}_{0}$

18. <arith factor>::=<arith primary>[↑<arith primary>]$^{n}_{0}$

19. <arith primary>::= <arith const >|<arith var >|<arith fn>|(<arith exp>)

20. <arith fn>::=<library name><arith primary>|SUM<pat exp>

21. <library name>::= +|-|SIN|COS|TAN|ABS|LOG|ATN|SGN|RND|INT|EXP|SQR

22. <pat exp>::= <net exp>|<pat part>

23. <pat part>::= <pat term>$\Big[$AN<pat term>$\Big]_0^n$

24. <pat term>::= <pat factor>$\Big[$EX<pat factor>$\Big]_0^n$

25. <pat factor>::= <pat primary>$\Big[$OR<pat primary>$\Big]_0^n$

26. <pat primary>::= <map term>|NT<map term>

27. <map term>::= <pattern var>|<pat var> BY<map-variable>|(<pat part>)

28. <pat var>::= PAT<var name>|PAT<var name><subscript list>

29. <map var>::= MAP<var name>|MAP<var name><subscript list>

30. <net exp>::=$\Big[$<pat term>AN$\Big]_0^n$<net term>

31. <net factor>::= $\Big[$<pattern factor EX$\Big]_0^n$<net factor>

32. <net primary>::= <net part>|NT<net part>

33. <net part>::= <net variable>IP<pattern-part>

34. <net variable>::= NET<var name>|NET<var name><subscript list>

35. <arith var>::= <var name>|<var name><subscript list>

36. <subscript list>::= (<arith exp>$\Big[$,<arith exp>$\Big]_0^n$)

37. <var name>::= <alpha>|<alpha><numeric>

38. <IF stmt>::= IF{<arith exp><rel-op><arith exp>|<sense switch>|
<pat exp>=<pat exp>} THEN <cond true stmt>

39. <cond true stmt>::= <LET stmt>|<GOTO stmt>|<GOSUB stmt>|<RETURN stmt>
|<PRINT stmt>|<PAD stmt>|<OUTPUT stmt>
|<INPUT stmt>|<Display stmt>|<ON stmt>
|<DRSPEC stmt>|<END stmt>

40. <sense switch>::= |SS1|SS2|SS3|SS4|SSS|SR1|SR2|SR3|SR5|SSR

41. <rel-op> := <greater than>|<equal to>|<less than>|<not equal to>|
<greater than or equal>|<less than or equal>

42. <greater than>::= >

43. <equal to> ::= =

44. <less than> ::= <

45. <not equal to> ::= <>|><

46. <greater than or equal> ::= =>|>=

47. <less than or equal> ;:= =<|<=

48. <DRSPEC stmt> ::= DRSPEC<ADMOS file descriptor>

49. <FOR stmt> ::= FOR <cntrl var>=<arith exp>TO<arith exp>
STEP<arith exp>

50. <INPUT stmt> ::= INPUT <channel discriptor> {<MINIC var>}

51. <PRINT stmt> ::= PRINT <Format specification>

52. <OUTPUT stmt> ::= OUTPUT<Format specification>

53. <Format specification> ::= <channel discriptor> <output list>

54. <output list> ::= <output separator> |<output item>|
<output item> <output seperator>

55. <output separator> ::= ,|;|<Format list>

56. <output item> ::= <MINIC exp>|<string>|<null>

57. <string> ::= "$[$<alpha numeric>$]_{0}^{71}$ "

58. <alpha numeric> ::= <alpha>|<numeric>

59. <Format list> ::= FORMAT <oset> <dimension>

60. <oset> ::= $<alpha numeric><alpha numeric>

61. <dimension> ::= (<integer>,<integer>)

62. <DIM stmt> ::= DIM<var discription>$[$,<var discription>$]_{0}^{n}$

63. <var discription> ::= <var name>(<size>)|<var type><var name>
<dimension> (<size>)

64. <size> ::= <integer>$[$,<integer>$]_{0}^{3}$

65. <var type> ::= PAT|NET|MAP

66. <channel discriptor>::=#<integer>|<null>

APPENDIX H

The micro-assembly language for the modular processor.

H 1.0 C-μ-instructions

H 1.0.0 Control store reference μ- instructions:

instruction format:

```
|10|0|M|I|XXX|YYYYYYYY|
0        6 8       15
```

M = address mode : 1- relative address, 0 - absolute address.

I = index address : 0- no index, 1- address indexed.

X = micro op. code, Y= address field or offset.

mnemonic	op.code_8	action
BR	4	Branch.
JS	2	Jump to subroutine.
PBR	1	Pop stack and branch.
WBR	0	Wait for DBUS read acknowledge and branch.
WRT	3	Write in W.C.S. at address field.

Syntax:

```
<label>:  <op.code><rel.ad.>(<addr.>|<index>)
<index>::= I   , <relad>::=  A|R
```

example:

```
LOOP:  JSAI
; jumps to subroutine given by the absolute
  address contained in CSIAR
```

H 1.0.1 Conditional Branch micro-instructions

instruction format:

```
|10|1|C|TTTT|FFFFFFFF|
0          8        15
```

T - address of TBUS signal, C -condition true or false,

F - relative offset for branch address

Syntax: <label>: BC<condition><TBUS><offset>

<condition >::= T|F , <offset>::=<label>|<no.>

TBUS - assignments:

Signal No.	Nmemonic	Description
0	DBC	DBUS = 0
1	DBS	DBUS= 177777_8
2	DBL	DBUS 15
3	DBM	DBUS 0
4	ARZ	ALU = 0
5	ARC	ALU carry
6	ARO	ALU overflow
7	ARS	ALU sign bit
8	LMB	Last mapped bit, MRO

9	SHC	Shift register carry
10	-	unassigned
11	-	"
12	-	"
13	-	"
14	-	"
15	-	"

N.B.

Conditional Branch micro-instructions are not synchronised with DBUS operations that may affect the value of TBUS conditions. In order to obtain a stabilised TBUS condition a

```
        WBR     *+1
```

micro-instruction must precede the Conditional Branch micro-instruction.

Alternatively, the program can be optimised by interposing any D-μinstruction that does not affect the TBUS conditions under test in place of the

```
        WBR     *+1
```

micro-instruction.

Example:

```
        MOV     AD    , MAR
        MOV     MBR   , A1
        WBR     *+1
        BCT     DBC   , LOOP
```

WBR must be used here since DBC is changed by all D-μinstructions.

```
SUB    AØ, BØ, AØ
MOV    MBR, BØ
BCT    ARZ, LOOP
```

WBR unnecessary in above example.

H 1.0.2 Generic Micro-instructions

mnemonic - RTN - return from subroutine. op.code - 6.

```
instruction format:     |10|0|0|0|110|dddddddd|
                         0          8      15
```

d - don't care.

mnemonic - GCO - generate control order. op.code -5.

```
instruction format:     |10|0|0|0|101 |eeeeeeee|
                         0          8      15
```

e - evoke control order.

H 1.1 D-μ instructions

H 1.1.0 ALU - micro-instructions

```
instruction format:     |0|M|GGGG|DDDDD|C|BB|AA|
                         0       6     11    15
```

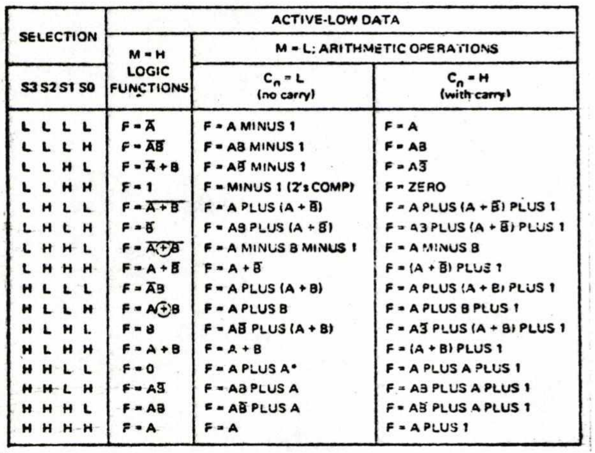
M - A.L.U. mode: 1 - logical mode, 0 - arithmetic mode.

D - DBUS destination address., C - carry in bit,

B - B operand register selector, A - A operand register selector

G - ALU op. code.

<u>A.L.U. op-codes</u>



SELECTION	ACTIVE-LOW DATA		
	M = H LOGIC FUNCTIONS	M = L; ARITHMETIC OPERATIONS	
S3 S2 S1 S0		C_n = L (no carry)	C_n = H (with carry)
L L L L	F = $\overline{A}$	F = A MINUS 1	F = A
L L L H	F = $\overline{AB}$	F = AB MINUS 1	F = AB
L L H L	F = $\overline{A}$ + B	F = A$\overline{B}$ MINUS 1	F = A$\overline{B}$
L L H H	F = 1	F = MINUS 1 (2's COMP)	F = ZERO
L H L L	F = $\overline{A + B}$	F = A PLUS (A + $\overline{B}$)	F = A PLUS (A + $\overline{B}$) PLUS 1
L H L H	F = $\overline{B}$	F = AB PLUS (A + $\overline{B}$)	F = AB PLUS (A + $\overline{B}$) PLUS 1
L H H L	F = $\overline{A \oplus B}$	F = A MINUS B MINUS 1	F = A MINUS B
L H H H	F = A + $\overline{B}$	F = A + $\overline{B}$	F = (A + $\overline{B}$) PLUS 1
H L L L	F = $\overline{A}$B	F = A PLUS (A + B)	F = A PLUS (A + B) PLUS 1
H L L H	F = A $\oplus$ B	F = A PLUS B	F = A PLUS B PLUS 1
H L H L	F = B	F = A$\overline{B}$ PLUS (A + B)	F = A$\overline{B}$ PLUS (A + B) PLUS 1
H L H H	F = A + B	F = A + B	F = (A + B) PLUS 1
H H L L	F = 0	F = A PLUS A*	F = A PLUS A PLUS 1
H H L H	F = A$\overline{B}$	F = AB PLUS A	F = AB PLUS A PLUS 1
H H H L	F = AB	F = A$\overline{B}$ PLUS A	F = A$\overline{B}$ PLUS A PLUS 1
H H H H	F = A	F = A	F = A PLUS 1

<u>Syntax:</u>

<label>: ALU<mode><function code><CIN><A select>,<B select>,

<DBUS destination>,

<mode>::= L|H; representing logical and arithmetic modes

<function code>::= <number> ; representing the row number

in the op.code table.

<CIN>::=c|<null ; (representing an asserted carry-in

<A select>::= A<number> , <B select>::= B<number>

Certain ALU operations of common usage are represented by dedicated mnemonics as follows:

ALUA0	-	DEC	A(n),	; decrement A(n)
ALUA6	-	ADD		; A(n) + B(n)
ALUA9C	-	SUB		; A(n) - B(n)
ALUA15C	-	INC	A(n)	; Increment A(n)
ALUL1	-	OR		; $A(n) \wedge B(n)$
ALUL4	-	AND		; $A(n) \vee B(n)$
ALUL9	-	EX		; $A(n) \oplus B(n)$

H 1.1.1 MOV micro-instructions

```
word format   |1|0|0|M|I|DDDDD|SSSSS|
               0         6     11    15
```

M - memory cycle control; I - auto -increment bit, (1 for increment).

Syntax:

<label>: MOV <mem. cyc.><DBUS source>,<DBUS destination>

(I-bit zero).

<label>: INC<mem. cyc><DBUS source>,<DBUS destination>

(I-bit one).

<mem. cyc>::= C|<null> ; C causes memory cycle

N.B. memory cycle control is only effective when a memory module source or destination reference is made by the same micro-instruction.

H 1.1.2 Shift micro-instructions

```
word format:  |1110|TT|Q|NNNN|SSSSS|
               0       6     11   15
```

T - shift top code ; Q - shift direction ; N - number of places minus one. S - DBUS source address

The result of a shift micro-instruction is obtained from the DBUS destination SR.

Syntax:

<label>: <shift op.code><shift direction><DBUS source> , <no. of shifts>;<shift op.codes> ::= LZ|AS|RO|LO

<shift direction> ::= L|R ; L - Left ; R - Right

For left shift Q is one.

Shift op.code	mnemonic:	Description
0	LZ	Logical shift, zero shift-in.
1	RO	Rotate
2	AS	Arithmetic shift, sign bit or zero shift-in
3	LO	Logical shift, one shift-in.

H 1.1.3 MAP micro-instructions

```
word format:    |111100|L|NNNN|SSSSS|
                 0       6    11   15
```

L - literal mapping flag; N - literal map bit selector, use as map element if L is one ; S - DBUS source.

Syntax:

<label> : MAP<literal><DBUS source>,<map literal>

<literal> ::= L|<null> ; <map literal> ::= <number>|<null>

The result of a MAP micro-instruction is obtained from the DBUS destination MW.

H 1.1.4 Literal micro-instructions

CLIT - control literal micro-instruction

```
Instruction format:  |111110|CCCCCCCCCC|
                      0     6         15
```

C - control order

Syntax:

<label>: CLIT <number>

Control literal assignments:

CLIT 0,0xy ; octal no.

x =	0 - null		
	1 -	CLMTR	Clear MINERVA Teach Register
	2 -	RESET1	Reset layer 1
	3 -	MTC1	MINERVA Teach Clock layer 1
	4 -	MTC2	MINERVA Teach Clock layer 2
	5→7 - unassigned		

y =	0 -	null	
	1 -	LDMTMR	Load MINERVA Teach Mask Register
	2 -	CRMTR	Clear MINERVA Teach Register
	3 -	CRCAR	Clear Card Address Register
	4 -	INCAR	Increment Card Address Register
	5 -	RESET 2	Reset Layer 2
	6 -	SET	Set both layers
	7 -	AUTO	Enter Auto-mode

DLIT - data literal micro-instruction

```
instruction format:  |111111|MMM|AA|LLLLL|
                      0      6   9  11
```

M - Data word most significant bits ;

A - A register destination address ;

L - Data word least significant bits ;

Syntax:

<label> : DLIT <number>,<A reg destination>

H 1.2 DBUS Address Assignments

H 1.2.0 Source Address Assignments

Code	Mnemonic	Description
0	A0	ALU, A operand register
1	A1	ALU, A operand register
2	A2	ALU, A operand register
3	A3	ALU, A operand register

4	B0	ALU B operand register
5	B1	ALU B operand register
6	B2	ALU B operand register
7	B3	ALU B operand register
10	DKS	Data Key switches, KD-module
11	MR	Map element register: 16
12	-	not used
13	MBR	Memory module data Buffer Register
14	SR	Shift Register
15	MW	Mapped result Word
16	CIR	Computer Inter face Register
17	BCR	Bit Counter Register
20	MRR1	MINERVA Response Register 1
21	MRR2	MINERVA Response Register 2
22→37	-	Not used

H 1.2.1 Destination Address Assignments

Code	mnemonic	Description
0	A0	ALU, A operand register
1	A1	ALU, A operand register
2	A2	ALU, A operand register
3	A3	ALU, A operand register
4	B0	ALU, B operand register
5	B1	ALU, B operand register
6	B2	ALU, B operand register
7	B3	ALU, B operand register
10	CSIAR	Control Store Index Address Register
11	CSBR	Control Store Buffer Register
12	MAR	Memory module Address Register

13	MBR	Memory module data Buffer Register
14	-	Not used
15	MR	Map element Register
16	CIR	Computer Interface Register
17	-	Not used
20	MDR	MINERVA Data Register
21	MCAR	MINERVA Card Address Register
22	MTDR	MINERVA Teach data Register
23	MTMR	MINERVA Teach Mask Register
24→37	-	Not used

H 1.2.2 DBUS Source and Destination Timing

mnemonic	tr	tw	trw	trr	(nS.)
A(n)	52	88	-	-	
B(n)	48	88	-	-	
BCR	-	-	-	-	
CIR	36	56	undefined	undefined	
CSBR	46	-	-	-	
CSIAR	36	-	-	-	
DKS	-	36	-	-	
MAR	60	-	-	600 av.	
MBR	60	36	395 av.		
MCAR	46	-	-	-	
MDR	46	-	-	-	
MR	52	38	-	-	
MRR1	-	72	-	-	
MRR2	-	76	-	-	

MTMR	46	-	-	-
MTDR	46	-	-	-
MW	-	32	-	-
SR	-	34	-	-

There is a micro-instruction latency time of 20 nS. minimum and 80 nS. maximum after DBRA has been asserted and SOI being generated. The maximum latency value occurs before a D-μ-instruction following a C-μ-instruction that is immediately preceeded by a D-μ-instruction of 72 nS. (i.e. MOV DKS,CSIAR).

H 1.3 Boot-strap Micro-program for Loading from the Host Computer

```
1       MOV      CIR,AØ         ; word count
2       MOV      CIR,CSIAR      ; start address
3 L1:   MOV      CIR,CSBR       ; copy micro-program
4       WRTAI                   ; put in W.C.S.
5       DEC      AØ ,AØ         ; word count - 1
6       INC         ,CSIAR      ; bump index
7       BCF      ARZ,L1         ; word count zero?
8       MOV      CIR,CSIAR      ; get transfer address
9       BRAI                    ; go execute
```

The micro-program above is used to transfer and execute micro-programs from the host computer. It has to be manually loaded into the WCS using the procedures given in sections 6.4.1.0 and 6.6.0 after powering-up the modular processor hardware.

H 1.3.0 MINIC Argument Access Micro-Routines

MINIC arguments are accessed indirectly using an argument string resident in the memory module. The origin of the argument string pertaining to a particular micro-code segment is placed in register BØ by the micro-routine GO which waits until the argument string pointer is sent from the PDP11/40 computer. (The first entry of the argument string is the pattern expression loop count.) The argument to be accessed from the memory-module is identified by means of an index offset from BØ in A3 using the DLIT micro-instruction.

```
GO:     MOV    CIR,BØ                  ; wait for host
        MOVC   BØ, MAR                 ; fetch loop count
        MOV    MBR,AØ                  ;
        RTN

GET:    ADD    A3,BØ, MAR              ; get argument addr.
        MOV    MBR,B3
        ADD    AØ, B3, MAR             ; result in MDR
        RTN

PUT:    ADD    A3,BØ, MAR              ; Get arg. addr.
        MOV    MBR,B3
        MOV    B1,MBR                  ; Store B1
        ADD    AØ, B3, MAR
        RTN
```

APPENDIX I

ERROR Mnemonics produced by MINIC 19/9/74.

Some errors do not produce mnemonics but force a logical error condition. Any such errors should be noted in the computer log since they may be indicative of unknown error conditions.

CE	Command Error. Command word not recognised.
CN	Channel Number error.
DN	Decimal Number error.
DS	Dimension Subscripts incorrectly formatted.
FR	Forward References overflow. Too many forward referenced line numbers in GOTO, in GOSUB, ON and IF statements.
IO	Indirect address table Overflow
JE	Jump Error. Referenced line number does not exist.
LO	Line number Overflow. Too many numbered statements.
M(	Missing (in expressions.
M)	Missing) in expressions
ML	Map Length too long when inputting map variables.
MN	Map Number missing or wrongly formatted when inputting map variables.
MO	Memory Overflow - no room for more run-time variables.
MR	Map number Range is too large when inputting map variables.
NR	No Room for more program text or object code.
NX	NEXT Statement does not match previous FOR statement.
OB	Subscript Out of Bounds
OP	OPerator missing or unrecognised in an expression.
PL	Pattern Length incorrect when inputting patterns.

QU	QUeer character in program source text.
RT	ReTurn error. Attempt to execute more RETURN statements than previously executed GOSUB statements.
SF	Subscript Format error
SK	StacK overflow - Expression too complex.
ST	Symbol Table overflow - Too many variable labels.
TU	Text Uncompiled - attempt to reference uncompiled text in an immediate statement.
UD	UnDefined variable.
VN	Variable Name missing or wrongly formatted.
VS	Variable Size error due to variables of incompatible sizes in an expression.
VT	Variable Type incompatible.
WU	Word Unrecognised that should begin a MINIC statement.

FLOW-CHARTS OF THE MINIC COMPILER SYSTEM

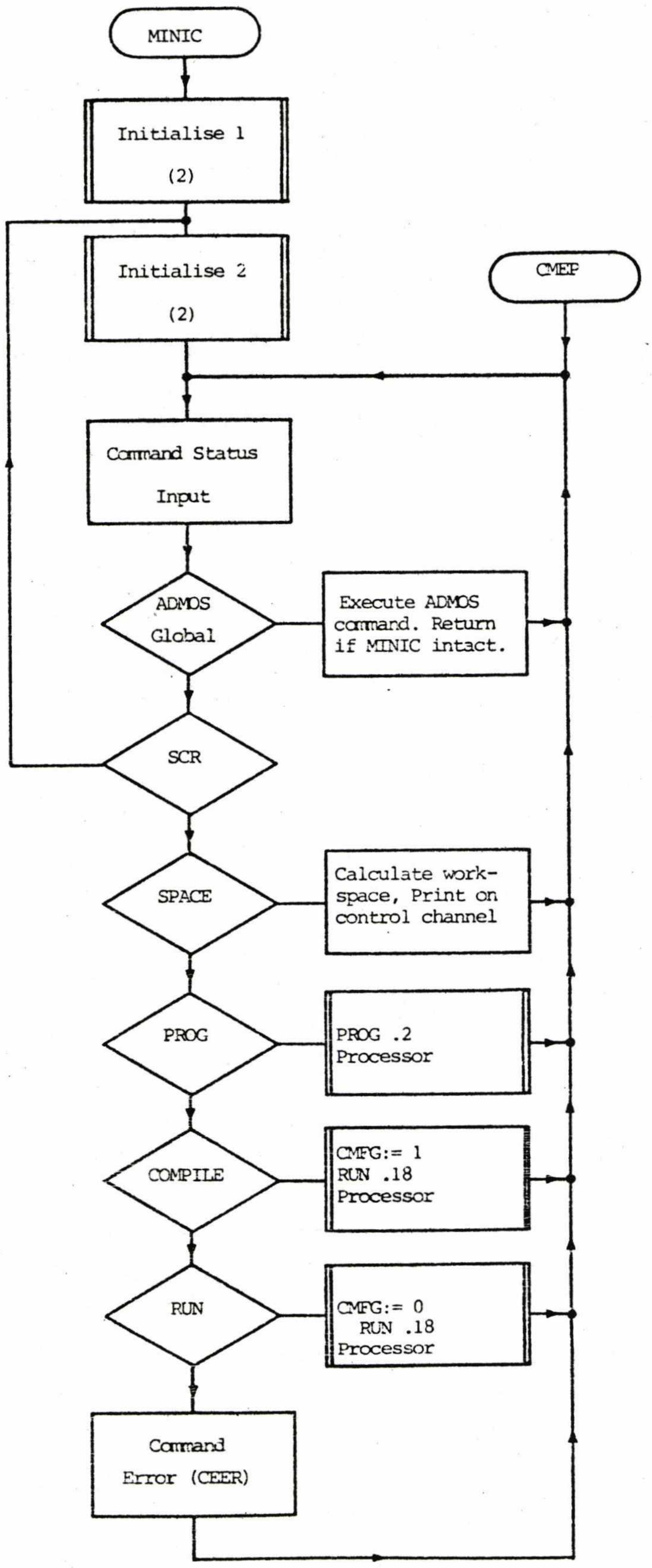
Cross-references in the flow diagrams consist of an entry point identified by a nmemonic or an integer preceded by a comma and the sheet sequence number preceded by a full-stop. Sheet sequence numbers appear at the bottom right hand of each page.

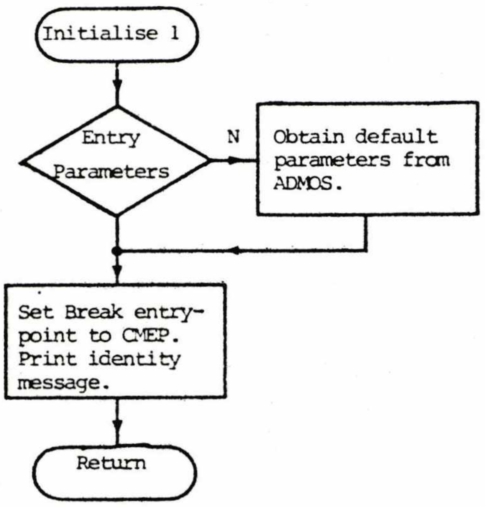
Nmemonics and labels are explained in the Glossary to the thesis.

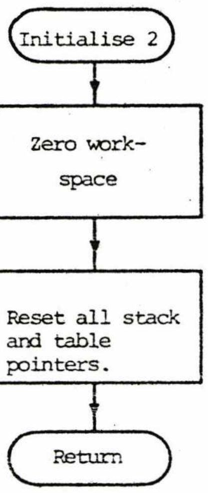
Entry-Point Directory

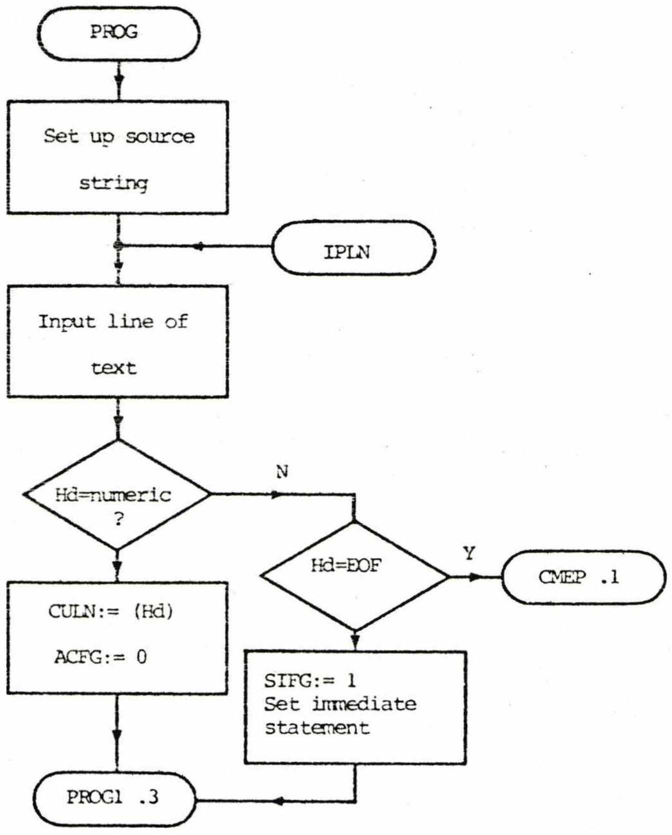
Entry-Point	Sheet Sequence C	Entry-point	Sheet Sequence C
CMEP	1	GSB7	42
COMPILE	18	GSGL	48
DISPLAY	12	GSMI	47
DISPLAYG	25	GTVL	49
DRSG	24	GTVR	49
ENDG	24	GVAD	47
EXPX	13	GVAR	49
EXV	16	GVLK	47
EXVR	17	GVNL	49
EXXP	12	IF	5
FOR	9	IFG	21
FORG	25	IF1	20
GDBL	48	Initialise 1	2
GESZ	45	Initialise 2	2
GEXG	29	INPUT	5
GFIC	42	INPUTG	24
GFPV	46	IPLN	2
GJMP	27	ISBS	43
GLD	50	LET	8
GOSUB	4	LETG	25
GOSUBG	22	MINIC	1
GOTO	4	NEXT	3
GOTOG	22	NEXTG	27
GPOG	45	NTIC	48
GRO	44		
GSBC	40		

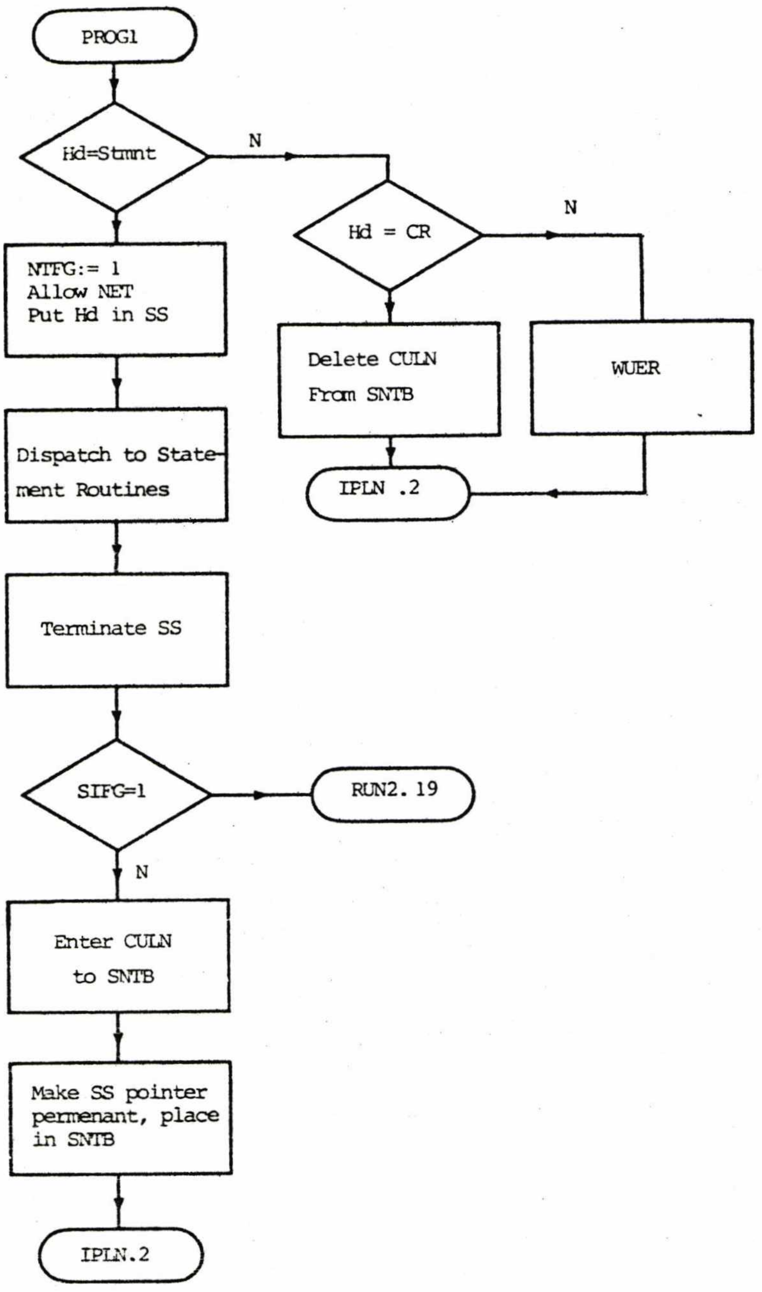
Entrypoint	Sheet Sequence
ON	4
ONG	24
OUTPUT	10
OUTPUTG	28
PAD	12
PRINT	10
PRINTG	28
PROG	2
PROG1	3
RETG	23
RUN	18
RUN2	19
SAVA	44
STOPG	24


MINIC
Initialise 1
(2)
Initialise 2
(2)
CMEP
Command Status
Input
ADMOS
Global
Execute ADMOS
command. Return
if MINIC intact.
SCR
SPACE
Calculate work-
space, Print on
control channel
PROG
PROG .2
Processor
COMPILE
CMFG:= 1
RUN .18
Processor
RUN
CMFG:= 0
RUN .18
Processor
Command
Error (CEER)

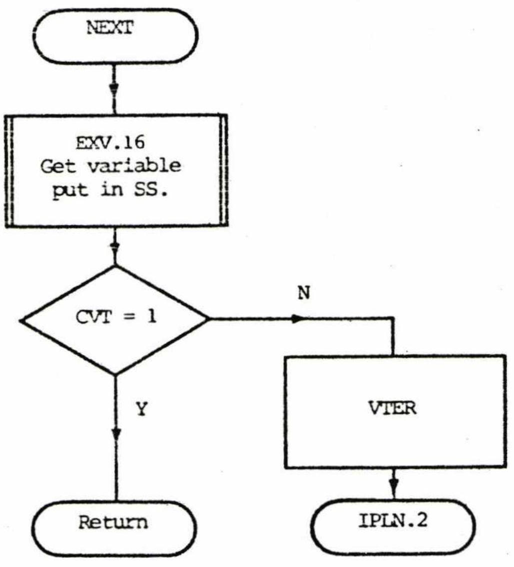

Initialise 1
Entry Parameters
N
Obtain default parameters from ADMOS.
Set Break entry-point to CMEP. Print identity message.
Return

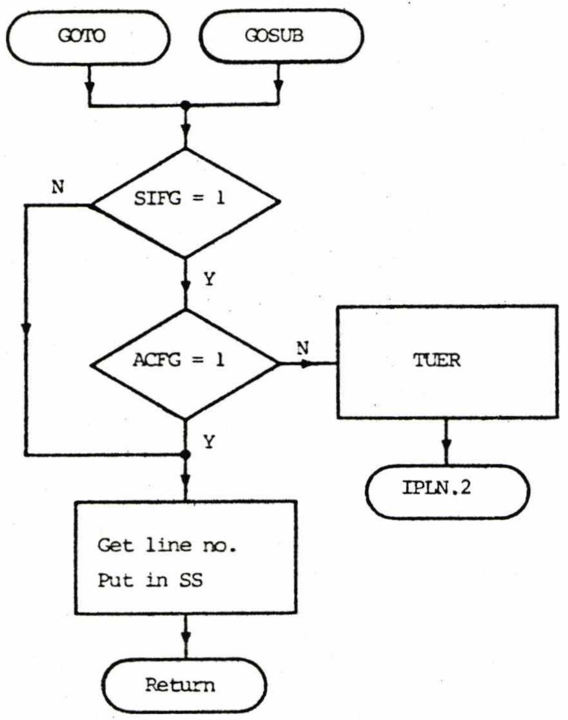

Initialise 2
Zero work-space
Reset all stack and table pointers.
Return

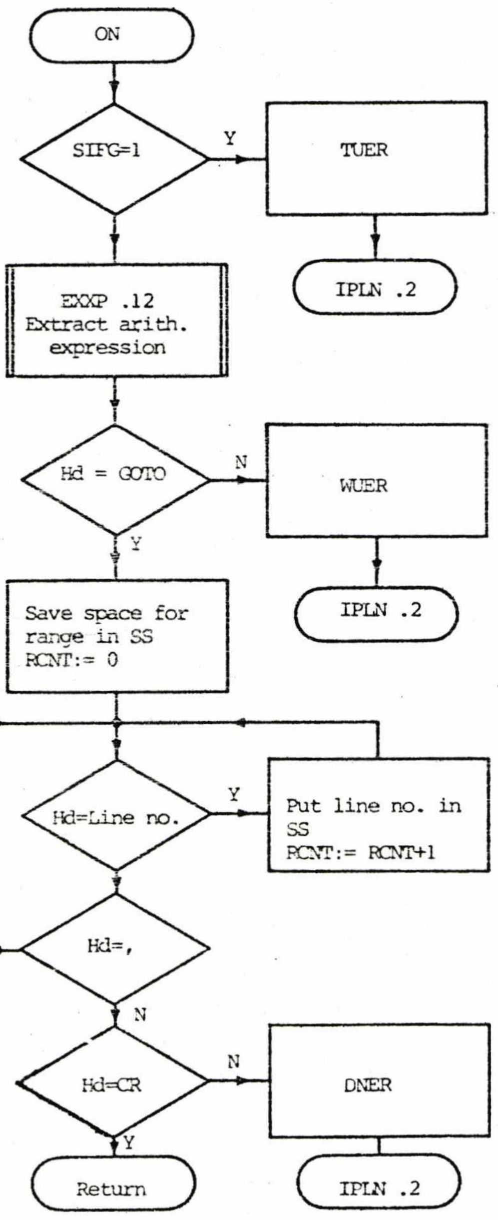

PROG
Set up source string
IPLN
Input line of text
Hd=numeric ?
N
Hd=EOF
Y
CMEP .1
CULN:= (Hd)
ACFG:= 0
SIFG:= 1
Set immediate statement
PROG1 .3

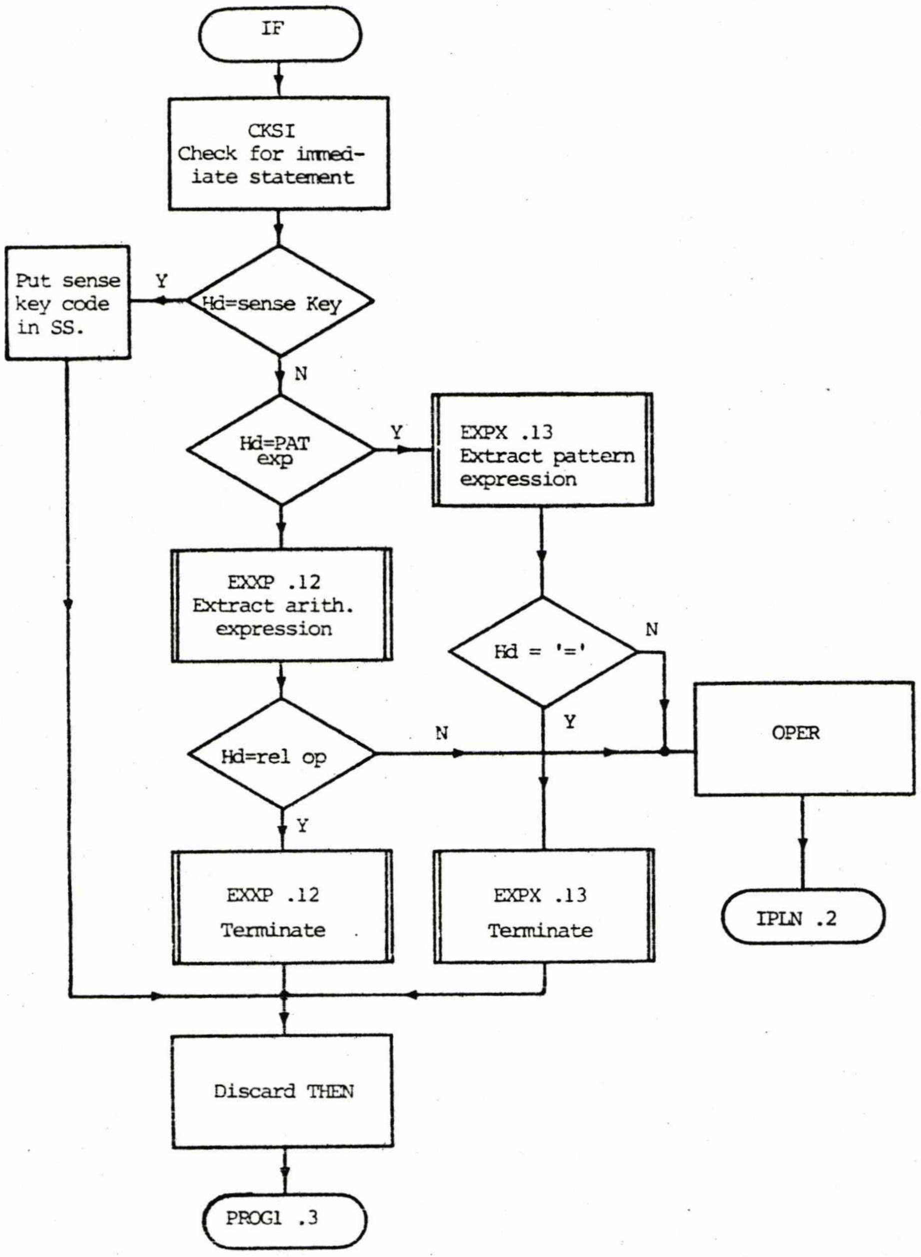


Statement Routines

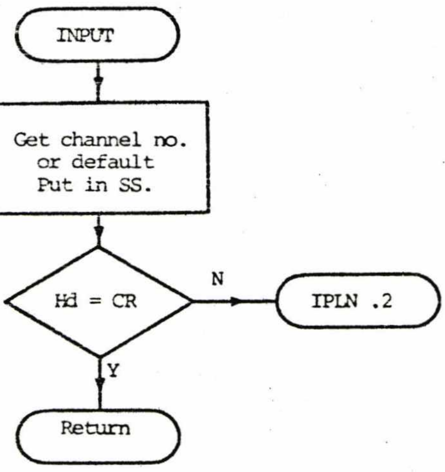



GOTO
GOSUB
SIFG = 1
N
Y
ACFG = 1
N
TUER
Y
IPLN.2
Get line no.
Put in SS
Return

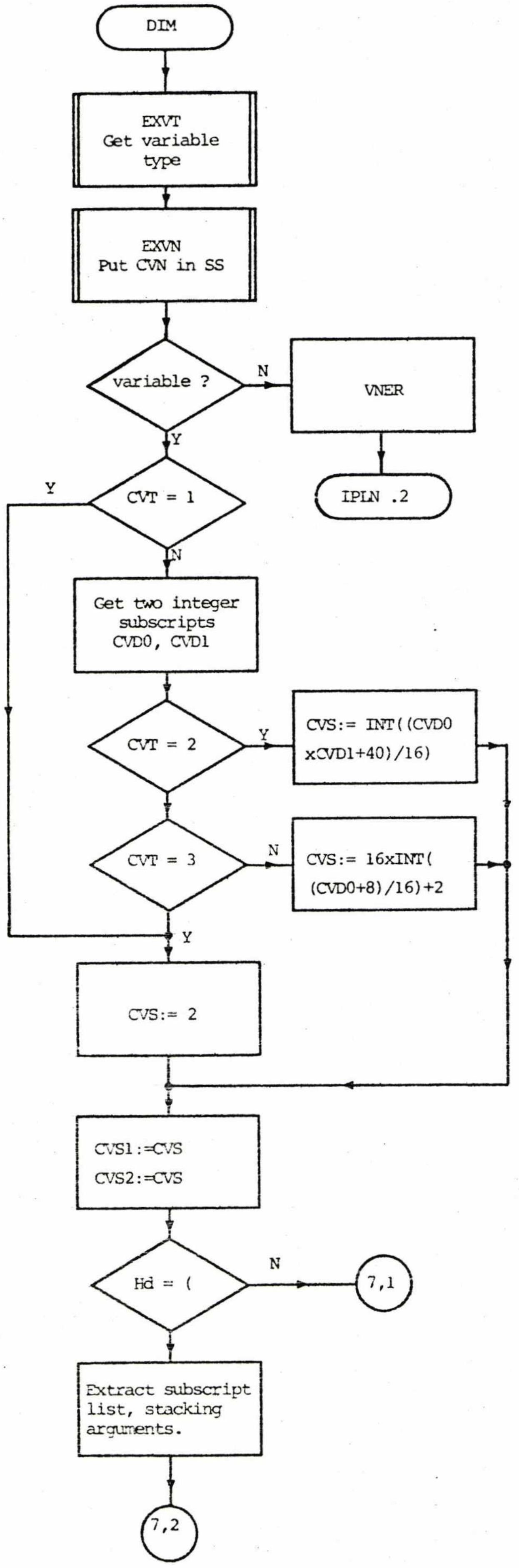

ON
SIFG=1
Y
TUER
IPLN .2
EXXP .12
Extract arith.
expression
Hd = GOTO
N
WUER
Y
IPLN .2
Save space for
range in SS
RCNT:= 0
Hd=Line no.
Y
Put line no. in
SS
RCNT:= RCNT+1
Y
Hd=,
N
Hd=CR
N
DNER
Y
Return
IPLN .2



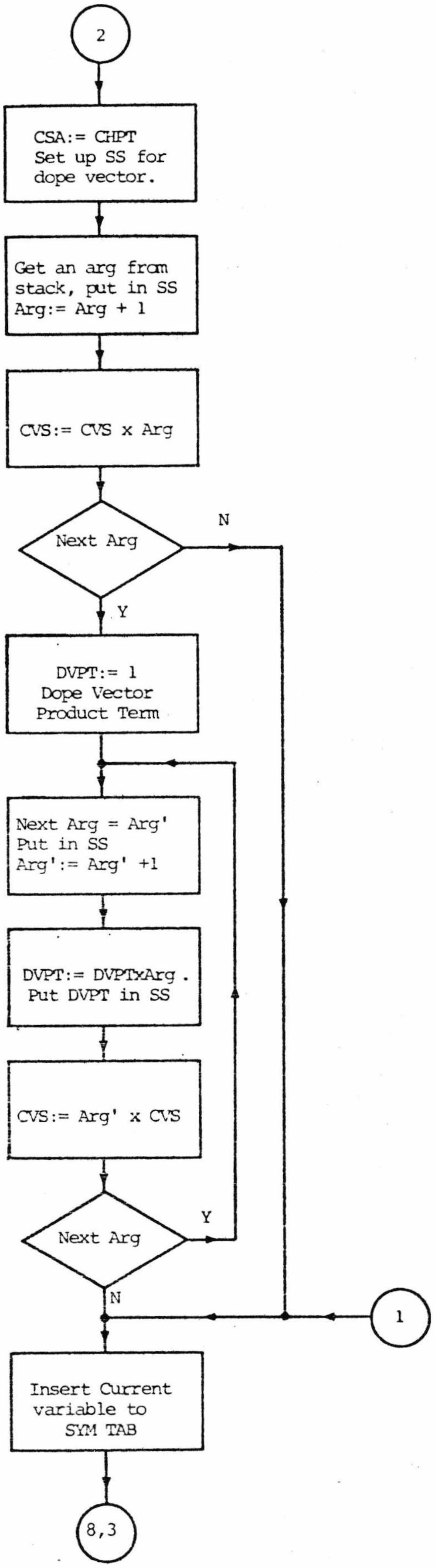
IF
CKSI
Check for immed-
iate statement
Hd=sense Key
Y
Put sense
key code
in SS.
N
Hd=PAT
exp
Y
EXPX .13
Extract pattern
expression
EXXP .12
Extract arith.
expression
Hd = '='
N
Y
Hd=rel op
N
OPER
Y
EXXP .12
Terminate
EXPX .13
Terminate
IPLN .2
Discard THEN
PROG1 .3

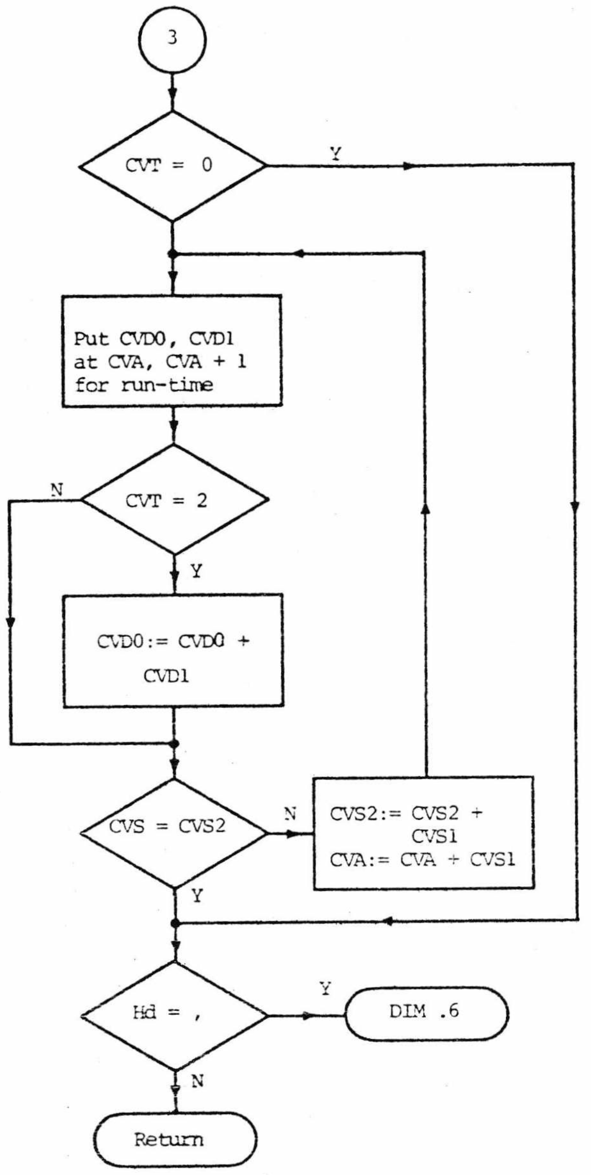


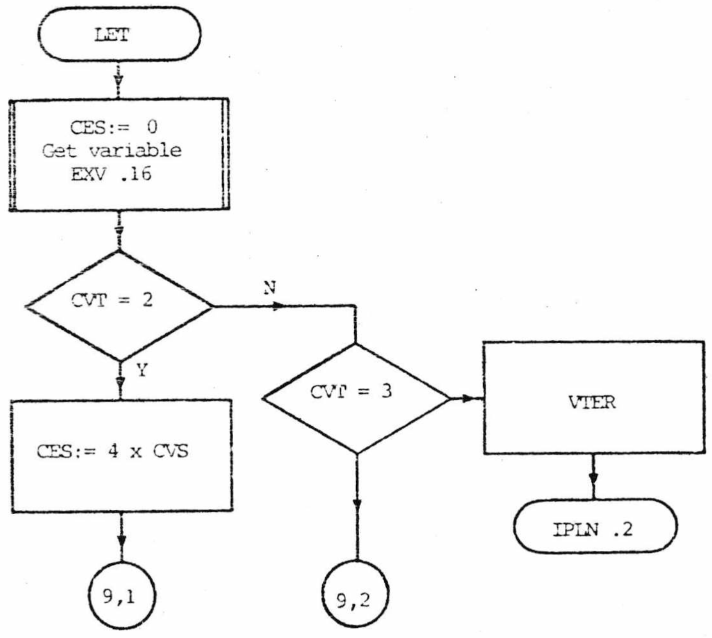
INPUT
Get channel no.
or default
Put in SS.
Hd = CR
N
IPLN .2
Y
Return

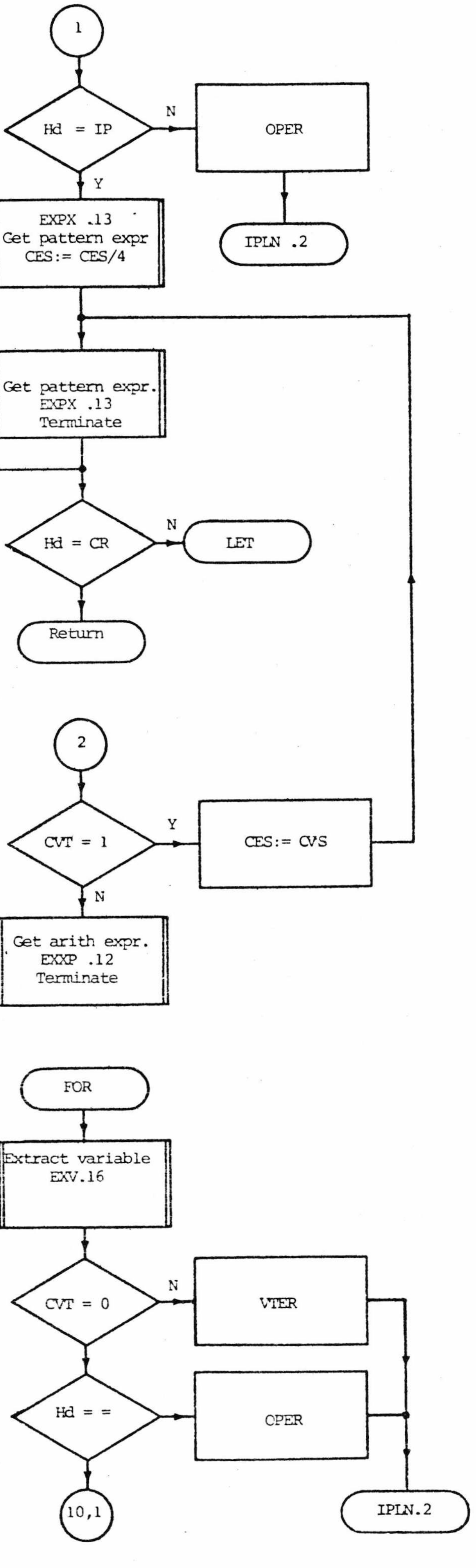


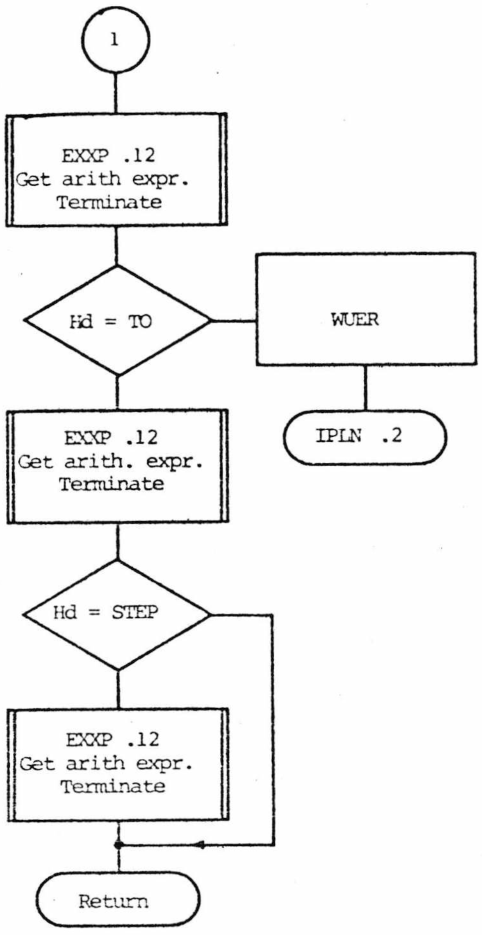
DIM
EXVT Get variable type
EXVN Put CVN in SS
variable ?
N
VNER
IPLN .2
Y
CVT = 1
Y
N
Get two integer subscripts CVD0, CVD1
CVT = 2
Y
CVS:= INT((CVD0 xCVD1+40)/16)
CVT = 3
N
CVS:= 16xINT((CVD0+8)/16)+2
Y
CVS:= 2
CVS1:=CVS
CVS2:=CVS
Hd = (
N
7,1
Extract subscript list, stacking arguments.
7,2

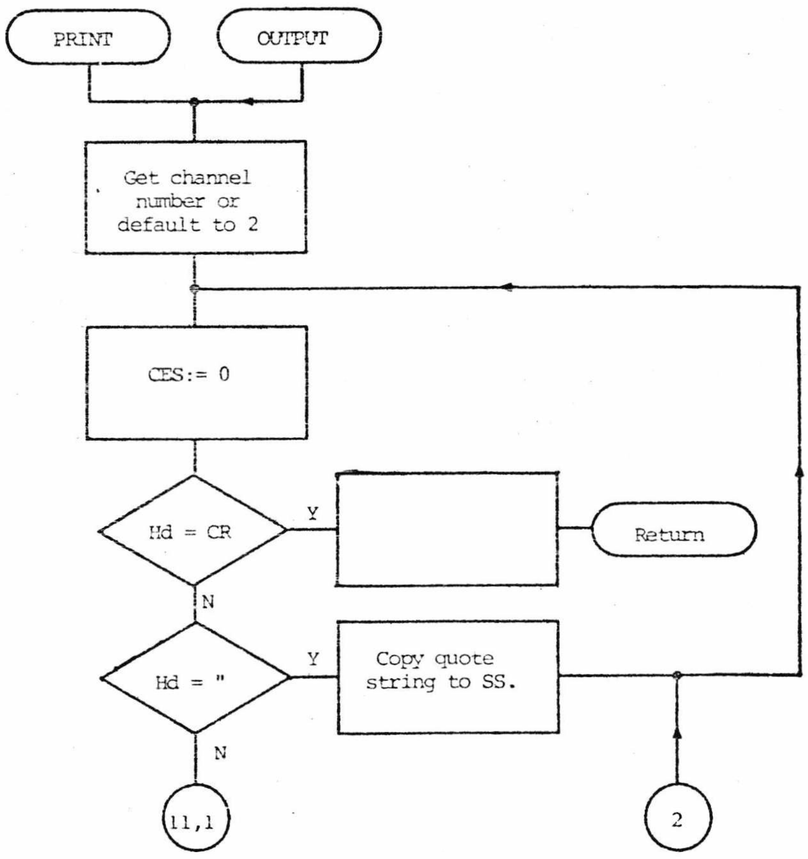

2
CSA:= CHPT
Set up SS for
dope vector.
Get an arg from
stack, put in SS
Arg:= Arg + 1
CVS:= CVS x Arg
Next Arg
N
Y
DVPT:= 1
Dope Vector
Product Term
Next Arg = Arg'
Put in SS
Arg':= Arg' +1
DVPT:= DVPTxArg .
Put DVPT in SS
CVS:= Arg' x CVS
Next Arg
Y
N
1
Insert Current
variable to
SYM TAB
8,3

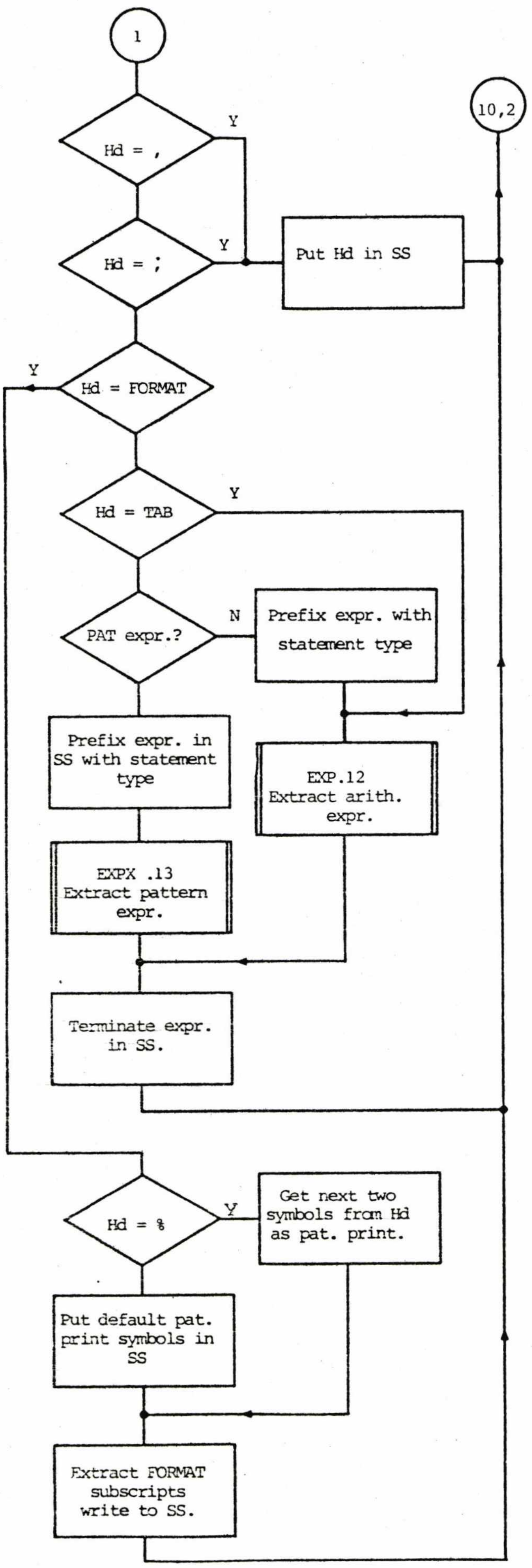

3
CVT = 0
Y
Put CVD0, CVD1 at CVA, CVA + 1 for run-time
CVT = 2
N
Y
CVD0:= CVD0 + CVD1
CVS = CVS2
N
CVS2:= CVS2 + CVS1
CVA:= CVA + CVS1
Y
Hd = ,
Y
DIM .6
N
Return

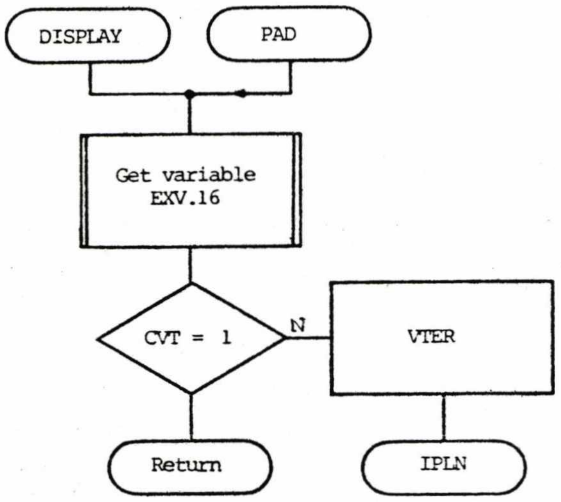

LET
CES:= 0
Get variable
EXV .16
CVT = 2
N
Y
CES:= 4 x CVS
CVT = 3
VTER
IPLN .2
9,1
9,2

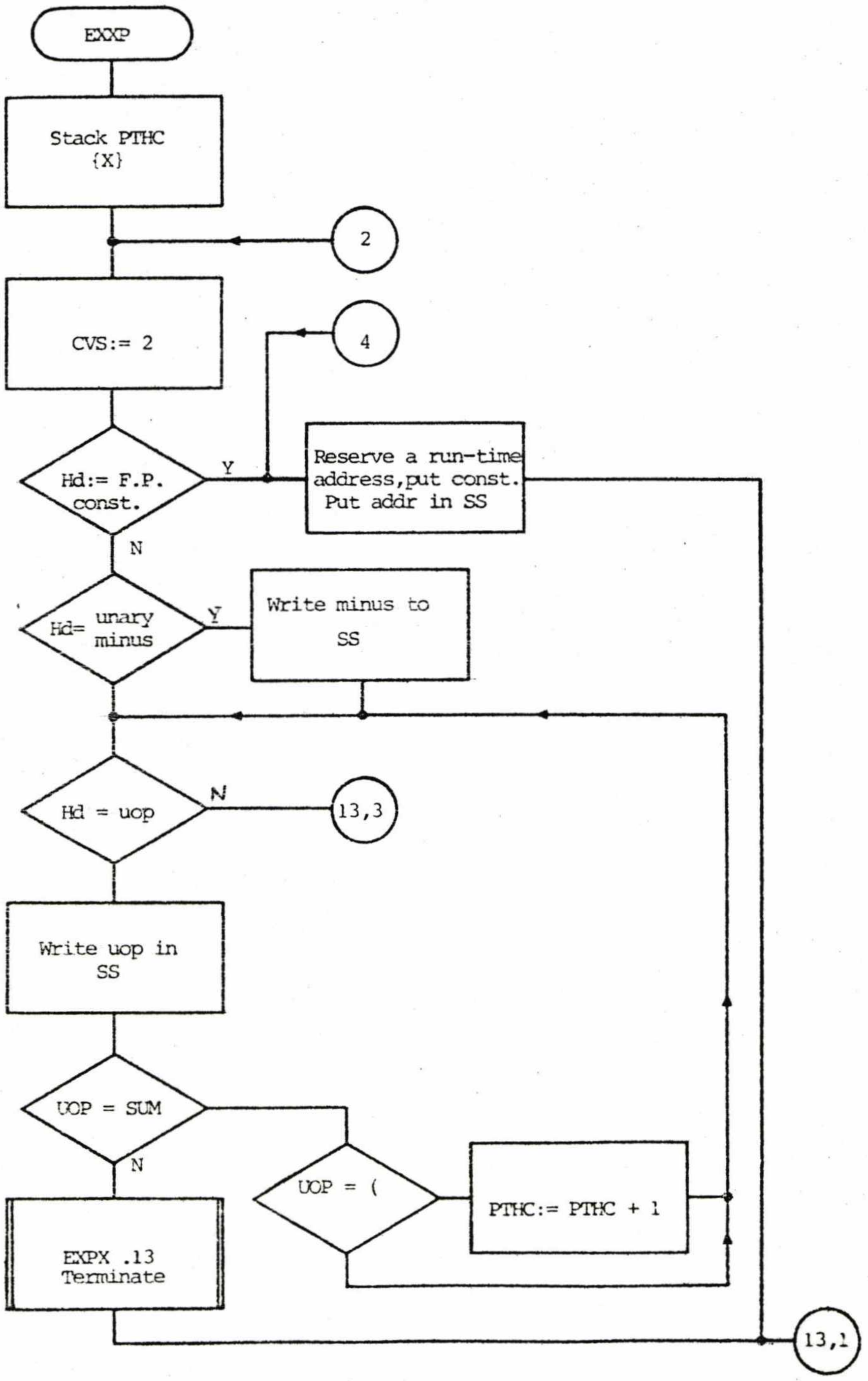

1
Hd = IP
N
OPER
IPLN .2
Y
EXPX .13
Get pattern expr
CES:= CES/4
Get pattern expr.
EXPX .13
Terminate
Hd = CR
N
LET
Return
2
CVT = 1
Y
CES:= CVS
N
Get arith expr.
EXXP .12
Terminate
FOR
Extract variable
EXV.16
CVT = 0
N
VTER
Hd = =
OPER
10,1
IPLN.2

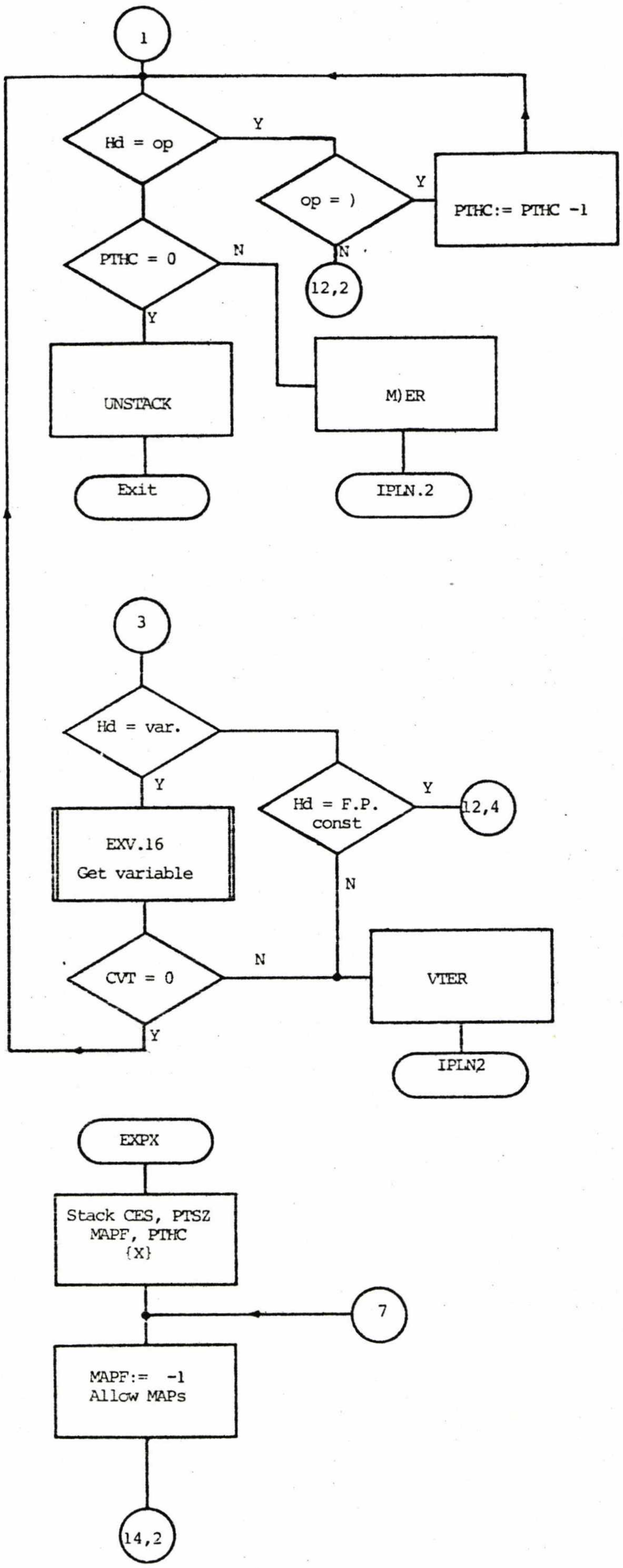

1
EXXP .12
Get arith expr.
Terminate
Hd = TO
WUER
IPLN .2
EXXP .12
Get arith. expr.
Terminate
Hd = STEP
EXXP .12
Get arith expr.
Terminate
Return

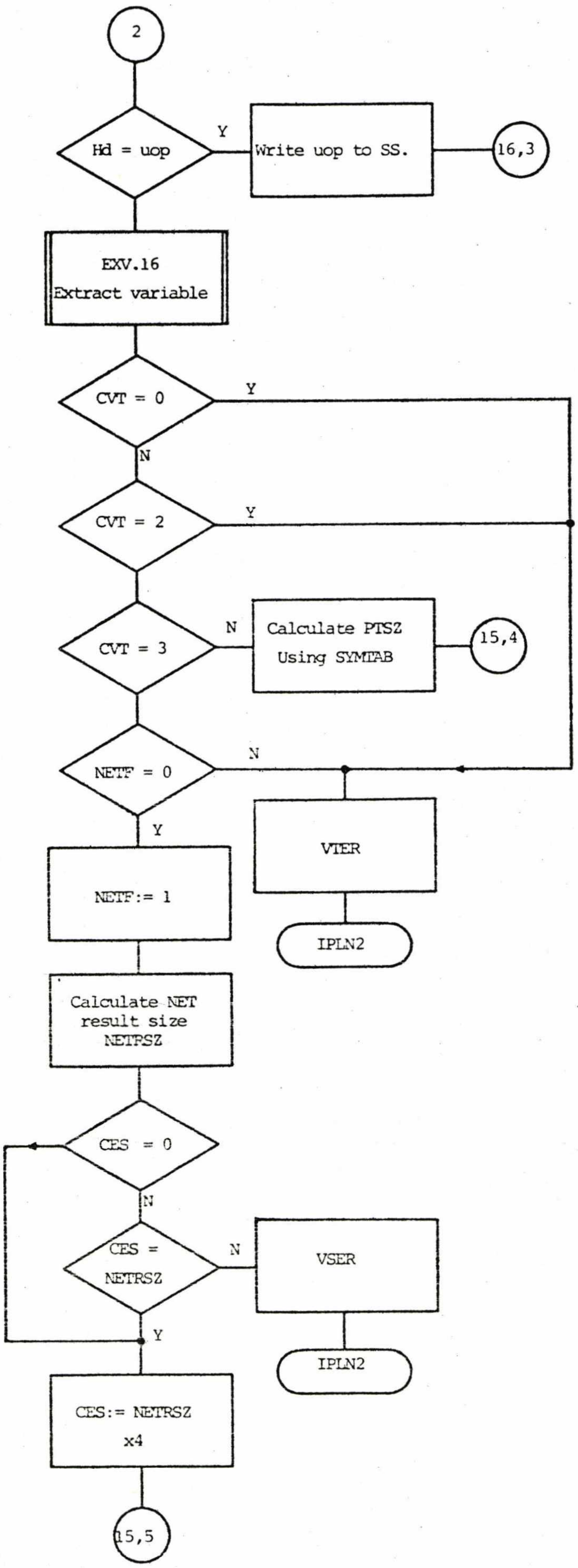

PRINT
OUTPUT
Get channel number or default to 2
CES:= 0
Hd = CR
Y
N
Return
Hd = "
Y
Copy quote string to SS.
N
11,1
2

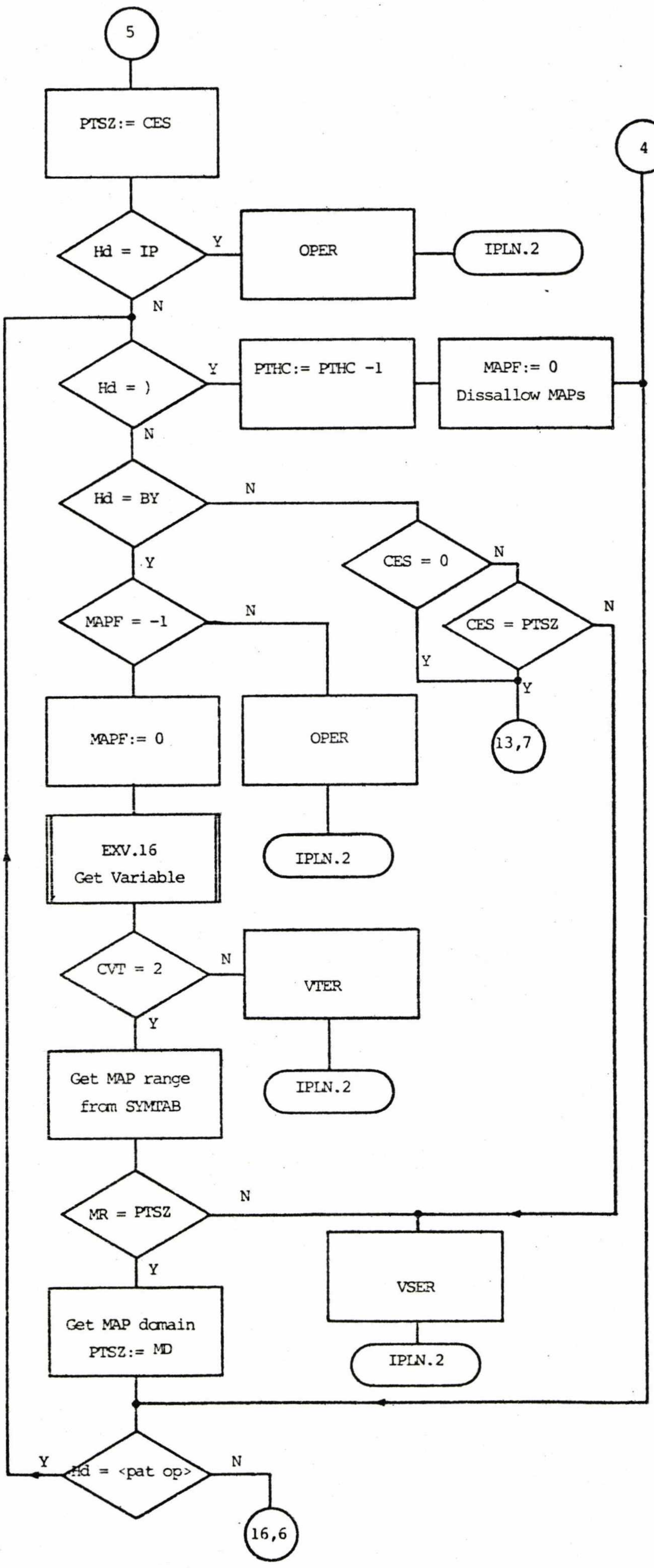

1
10,2
Hd = ,
Y
Hd = ;
Y
Put Hd in SS
Y
Hd = FORMAT
Hd = TAB
Y
PAT expr.?
N
Prefix expr. with statement type
Prefix expr. in SS with statement type
EXP.12 Extract arith. expr.
EXPX .13 Extract pattern expr.
Terminate expr. in SS.
Hd = %
Y
Get next two symbols from Hd as pat. print.
Put default pat. print symbols in SS
Extract FORMAT subscripts write to SS.

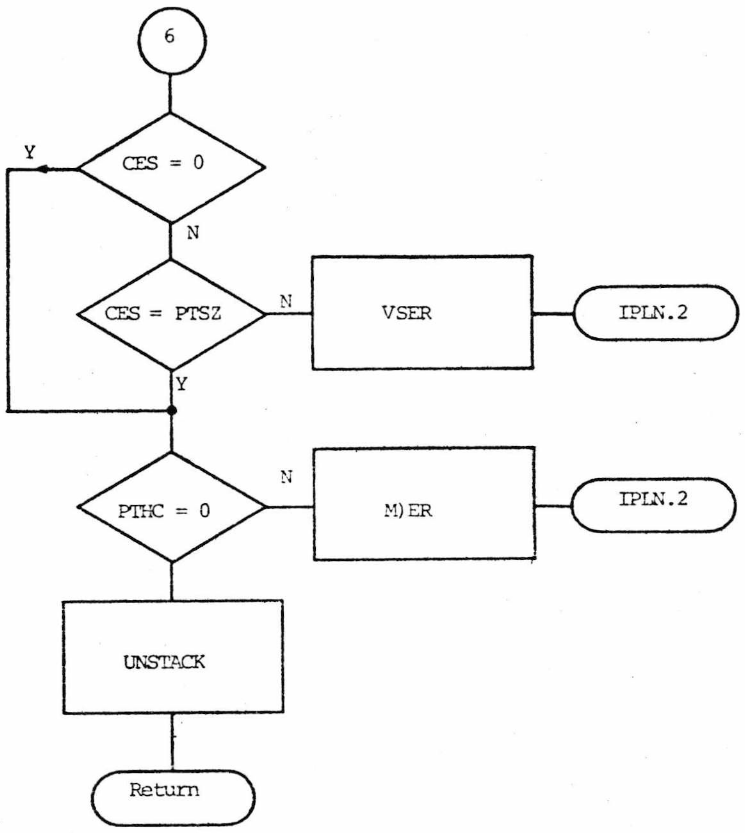

DISPLAY
PAD
Get variable
EXV.16
CVT = 1
N
VTER
Return
IPLN

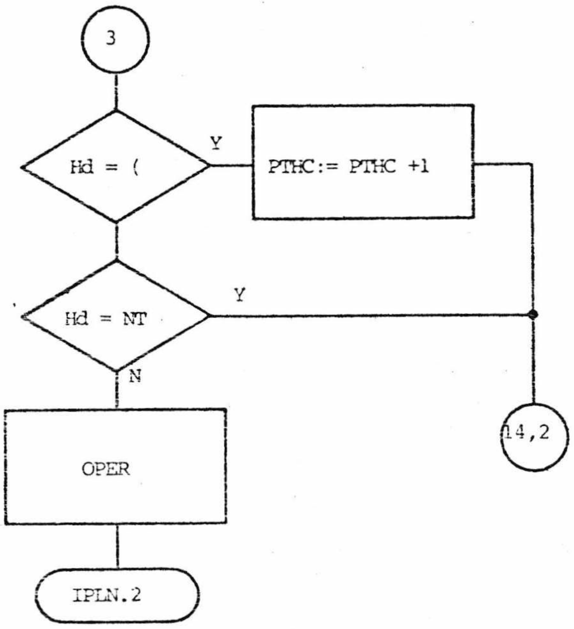

EXXP
Stack PTHC
{X}
2
CVS:= 2
4
Hd:= F.P.
const.
Y
Reserve a run-time
address,put const.
Put addr in SS
N
Hd= unary
minus
Y
Write minus to
SS
Hd = uop
N
13,3
Write uop in
SS
UOP = SUM
N
UOP = (
PTHC:= PTHC + 1
EXPX .13
Terminate
13,1

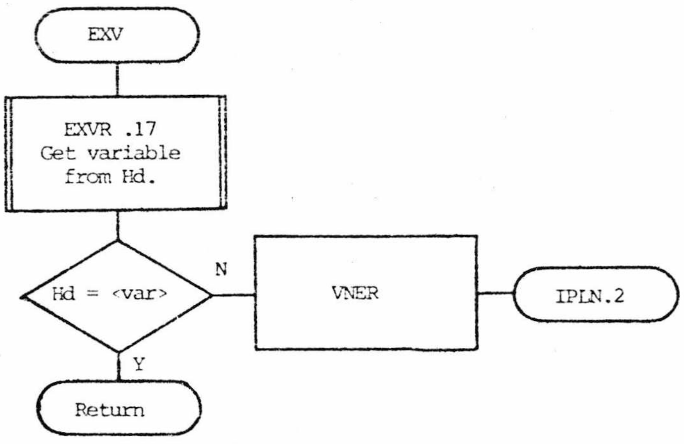

1
Hd = op
Y
op =)
Y
PTHC:= PTHC -1
N
12,2
PTHC = 0
N
Y
UNSTACK
M)ER
Exit
IPLN.2
3
Hd = var.
Y
Hd = F.P. const
Y
12,4
N
EXV.16
Get variable
CVT = 0
N
VTER
Y
IPLN2
EXPX
Stack CES, PTSZ MAPF, PTHC {X}
7
MAPF:= -1 Allow MAPs
14,2

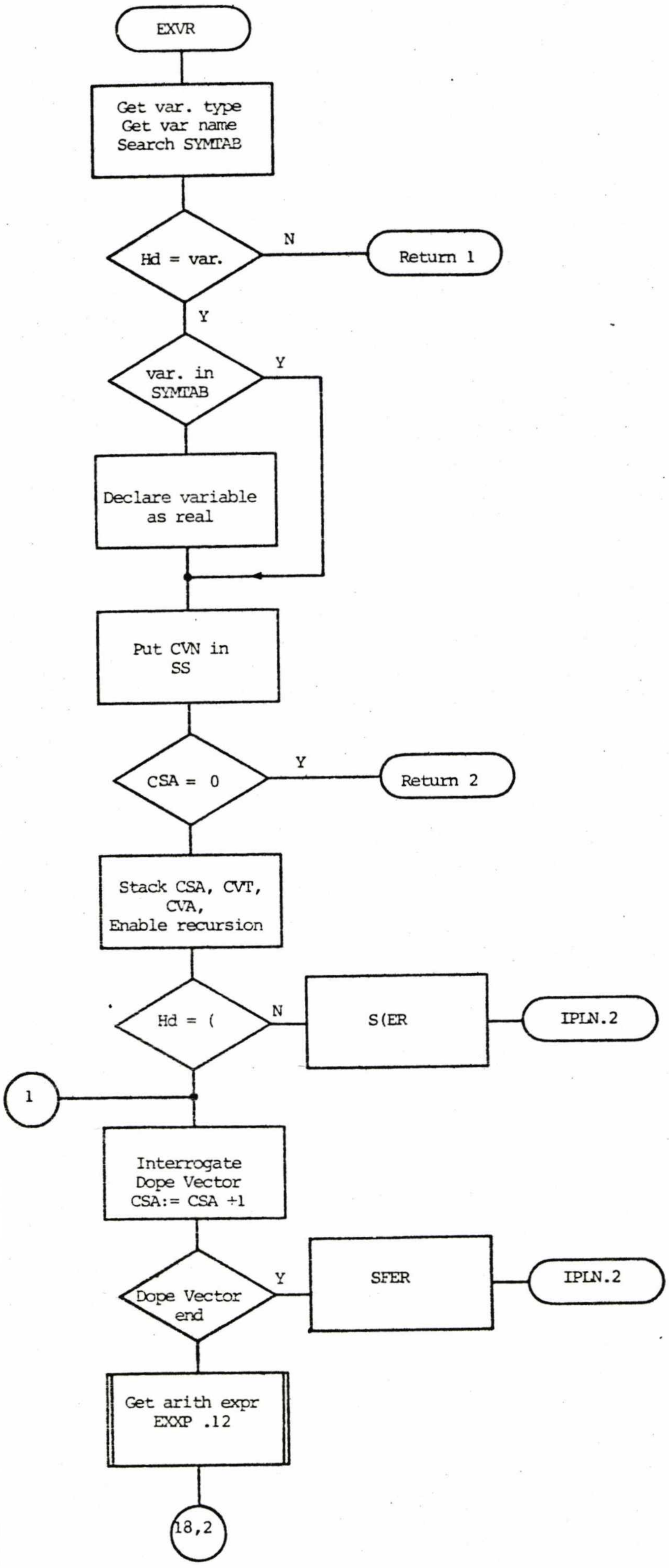

2
Hd = uop
Y
Write uop to SS.
16,3
EXV.16
Extract variable
CVT = 0
Y
N
CVT = 2
Y
CVT = 3
N
Calculate PTSZ
Using SYMTAB
15,4
NETF = 0
N
Y
VTER
IPLN2
NETF:= 1
Calculate NET
result size
NETRSZ
CES = 0
N
CES =
NETRSZ
N
VSER
IPLN2
Y
CES:= NETRSZ
x4
15,5

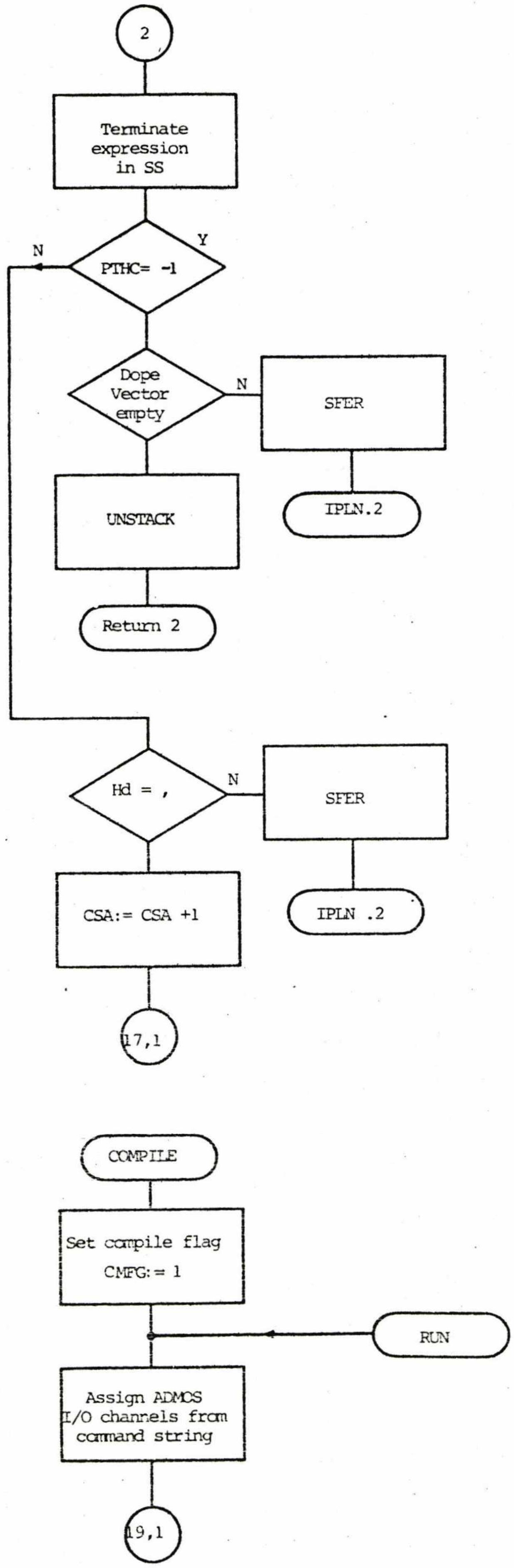

5
PTSZ:= CES
4
Hd = IP
Y
N
OPER
IPLN.2
Hd =)
Y
N
PTHC:= PTHC -1
MAPF:= 0
Dissallow MAPs
Hd = BY
N
Y
CES = 0
N
Y
CES = PTSZ
N
Y
13,7
MAPF = -1
N
MAPF:= 0
OPER
IPLN.2
EXV.16
Get Variable
CVT = 2
N
Y
VTER
IPLN.2
Get MAP range
from SYMTAB
MR = PTSZ
N
Y
Get MAP domain
PTSZ:= MD
VSER
IPLN.2
Hd = <pat op>
Y
N
16,6

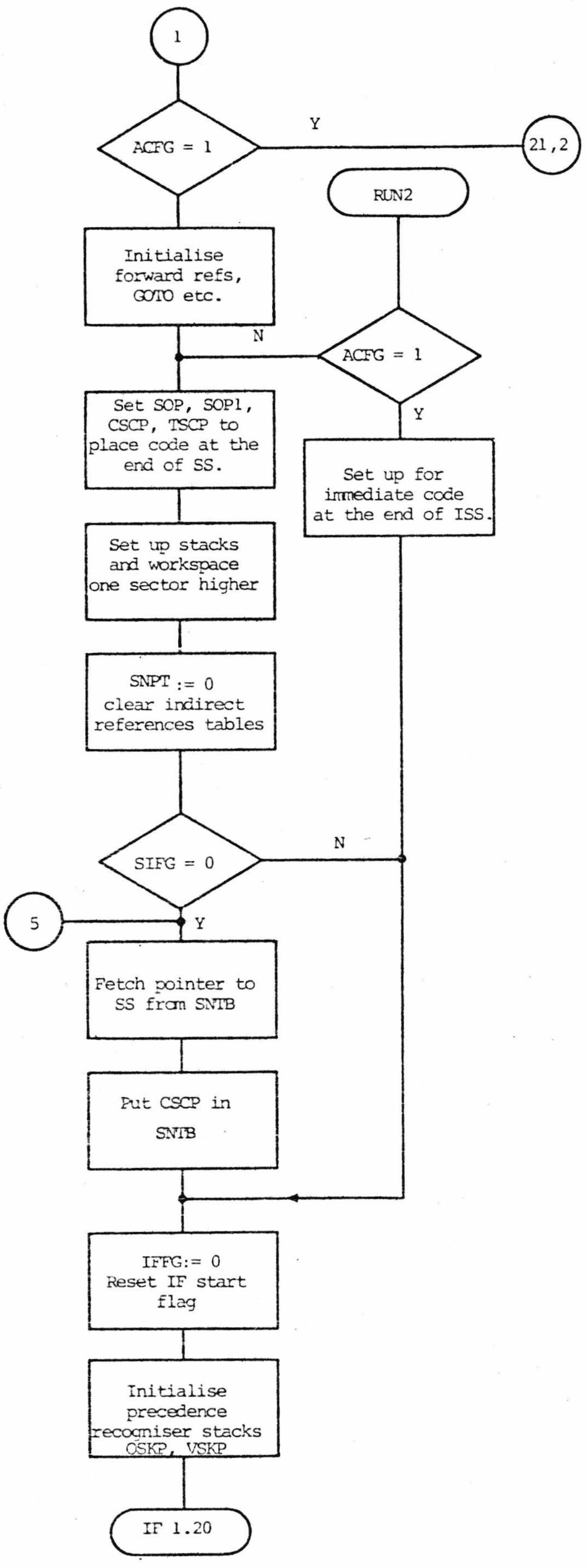

6
CES = 0
Y
N
CES = PTSZ
N
VSER
IPLN.2
Y
PTHC = 0
N
M)ER
IPLN.2
UNSTACK
Return


3
Hd = (
Y
PTHC:= PTHC +1
Hd = NT
Y
N
OPER
14,2
IPLN.2

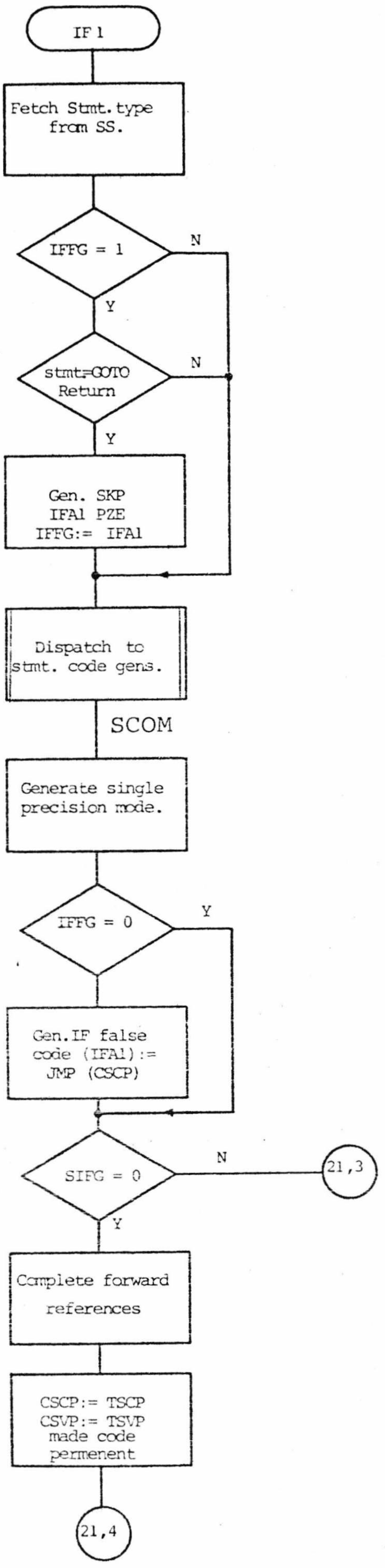

EXV
EXVR .17
Get variable
from Hd.
Hd = <var>
N
VNER
IPLN.2
Y
Return

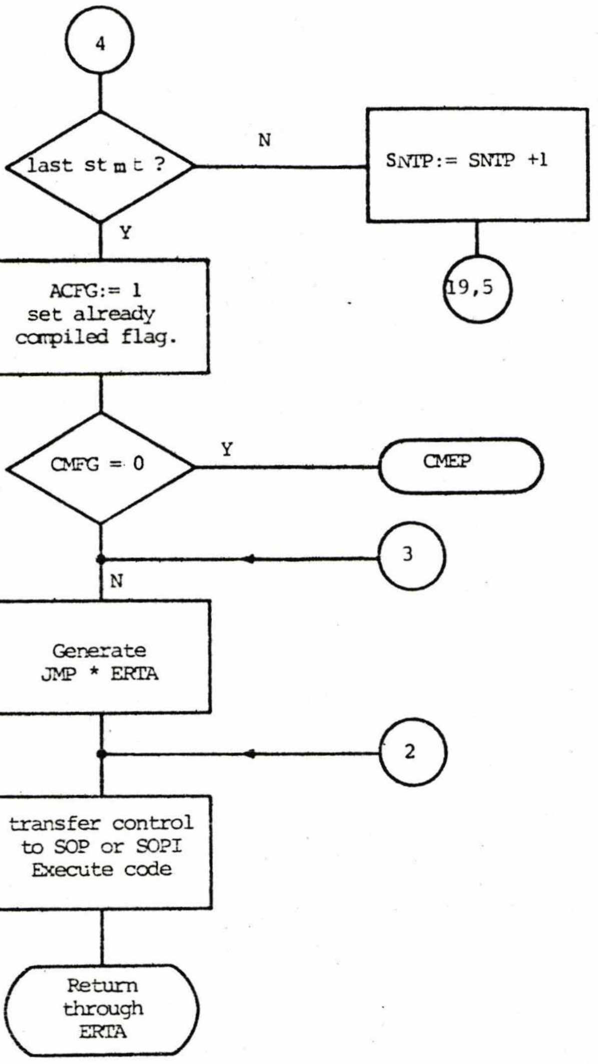

EXVR
Get var. type
Get var name
Search SYMTAB
Hd = var.
N
Return 1
Y
var. in SYMTAB
Y
Declare variable as real
Put CVN in SS
CSA = 0
Y
Return 2
Stack CSA, CVT, CVA, Enable recursion
Hd = (
N
S(ER
IPLN.2
1
Interrogate Dope Vector CSA:= CSA +1
Dope Vector end
Y
SFER
IPLN.2
Get arith expr EXXP .12
18,2


2
Terminate expression in SS
PTHC= -1
N
Y
Dope Vector empty
N
SFER
IPLN.2
UNSTACK
Return 2
Hd = ,
N
SFER
IPLN .2
CSA:= CSA +1
17,1
COMPILE
Set compile flag
CMFG:= 1
RUN
Assign ADMOS I/O channels from command string
19,1

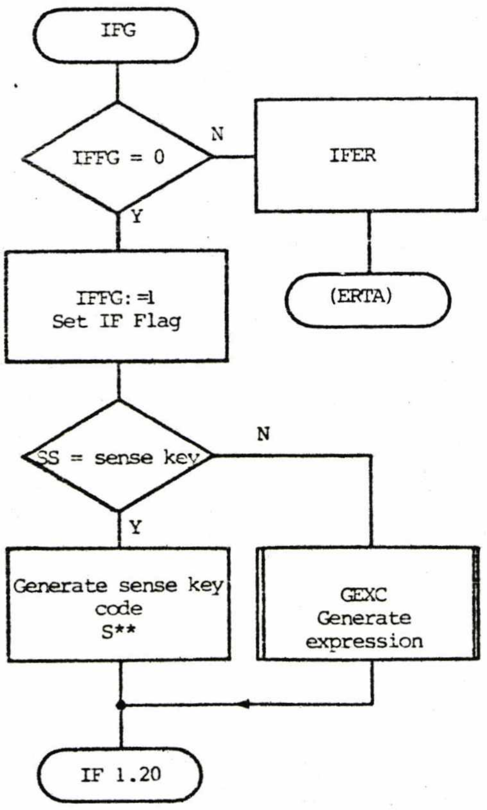


1
ACFG = 1
Y
21,2
RUN2
Initialise forward refs, GOTO etc.
N
ACFG = 1
Set SOP, SOP1, CSCP, TSCP to place code at the end of SS.
Y
Set up for immediate code at the end of ISS.
Set up stacks and workspace one sector higher
SNPT := 0 clear indirect references tables
N
SIFG = 0
5
Y
Fetch pointer to SS from SNTB
Put CSCP in SNTB
IFFG:= 0 Reset IF start flag
Initialise precedence recogniser stacks OSKP, VSKP
IF 1.20

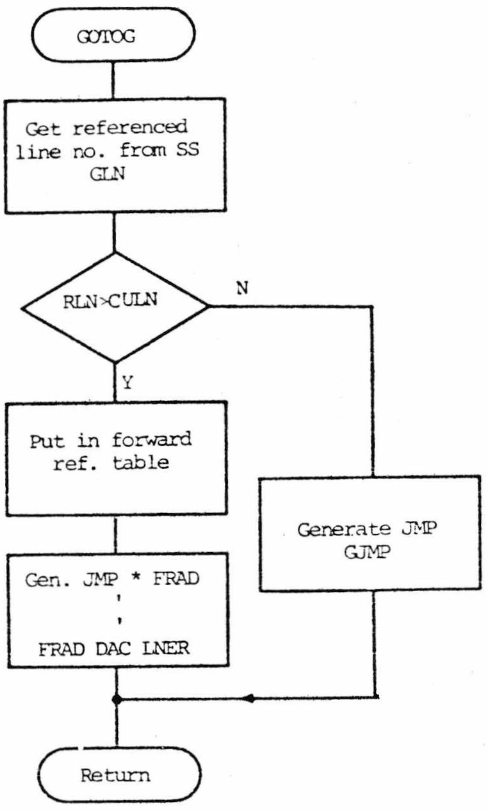

IF 1
Fetch Stmt. type from SS.
IFFG = 1
N
Y
stmt=GOTO Return
N
Y
Gen. SKP IFA1 PZE IFFG:= IFA1
Dispatch to stmt. code gens.
SCOM
Generate single precision mode.
IFFG = 0
Y
Gen. IF false code (IFA1):= JMP (CSCP)
SIFG = 0
N
21,3
Y
Complete forward references
CSCP:= TSCP
CSVP:= TSVP
made code permenent
21,4

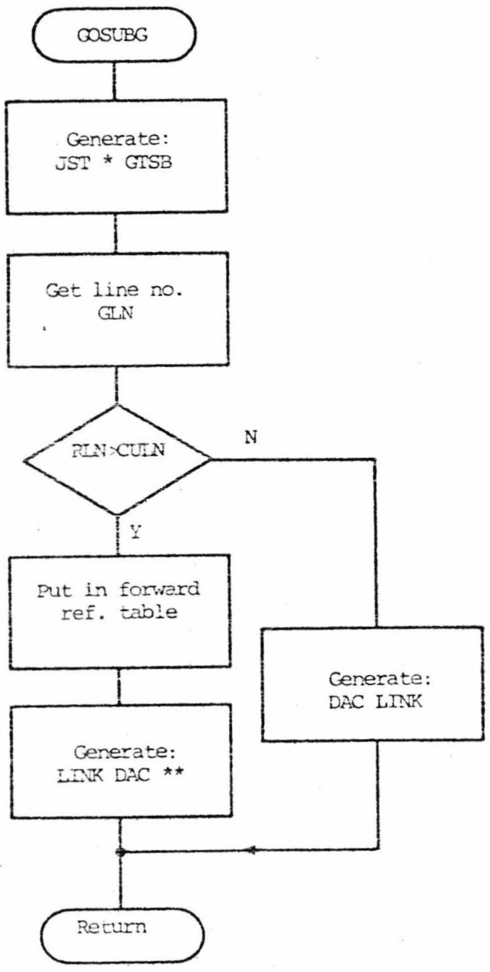


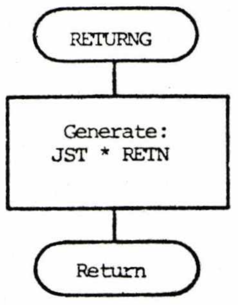
4
last stmt ?
N
SNTP:= SNTP +1
19,5
Y
ACFG:= 1
set already
compiled flag.
CMFG = 0
Y
CMEP
3
N
Generate
JMP * ERTA
2
transfer control
to SOP or SOPI
Execute code
Return
through
ERTA

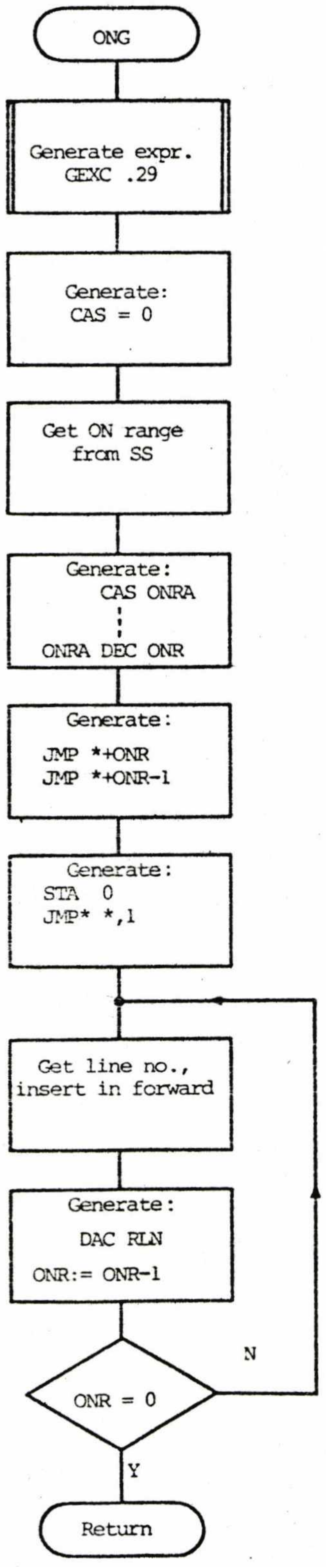


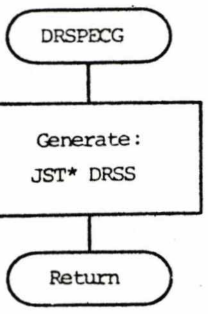
IFG
IFFG = 0
N
IFER
(ERTA)
Y
IFFG: =1
Set IF Flag
SS = sense key
N
Y
Generate sense key
code
S**
GEXC
Generate
expression
IF 1.20

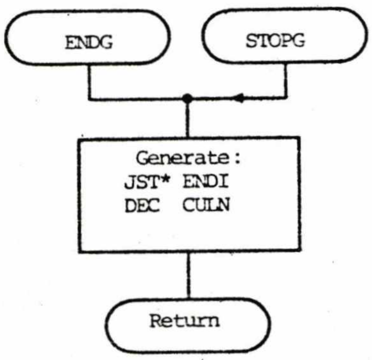

GOTOG
Get referenced line no. from SS GLN
RLN>CULN
N
Y
Put in forward ref. table
Generate JMP GJMP
Gen. JMP * FRAD
FRAD DAC LNER
Return

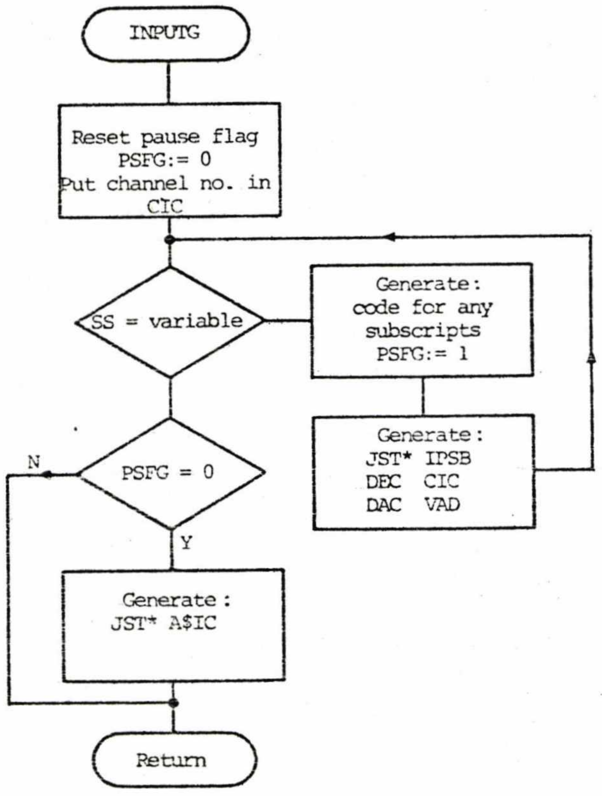

GOSUBG
Generate: JST * GTSB
Get line no. GLN
RLN>CULN
N
Y
Put in forward ref. table
Generate: DAC LINK
Generate: LINK DAC **
Return

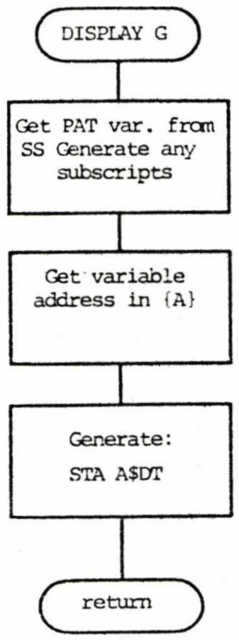

RETURNG
Generate:
JST * RETN
Return

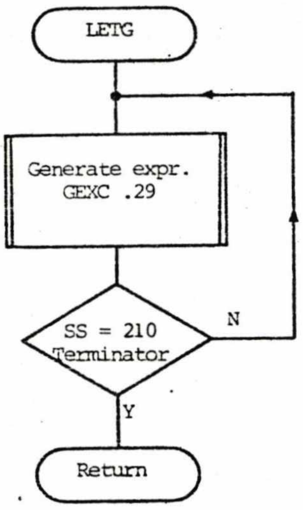

ONG
Generate expr.
GEXC .29
Generate:
CAS = 0
Get ON range
from SS
Generate:
CAS ONRA
:
ONRA DEC ONR
Generate:
JMP *+ONR
JMP *+ONR-1
Generate:
STA 0
JMP* *,1
Get line no.,
insert in forward
Generate:
DAC RLN
ONR:= ONR-1
ONR = 0
N
Y
Return

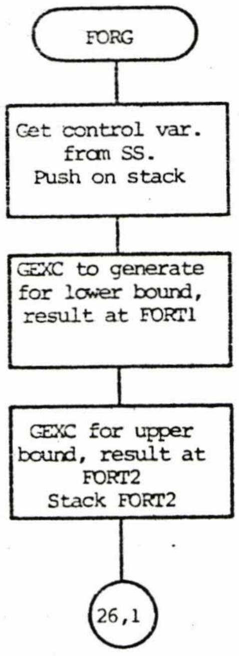

DRSPECG
Generate:
JST* DRSS
Return

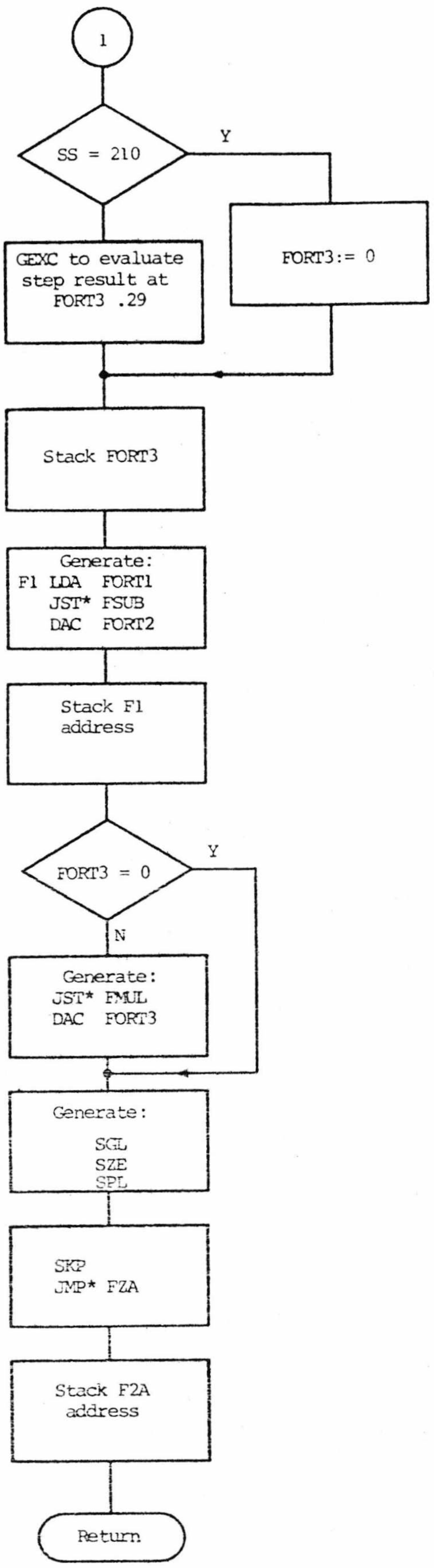

ENDG
STOPG
Generate:
JST* ENDI
DEC CULN
Return

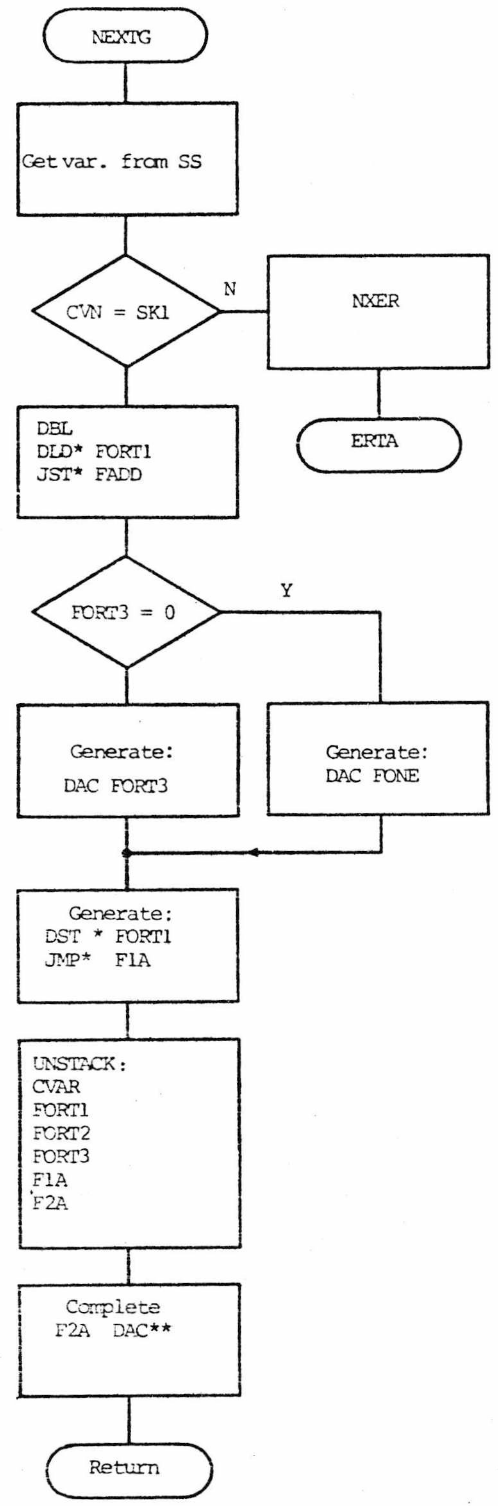

INPUTG
Reset pause flag
PSFG:= 0
Put channel no. in
CIC
SS = variable
Generate:
code for any
subscripts
PSFG:= 1
Generate:
JST* IPSB
DEC CIC
DAC VAD
PSFG = 0
N
Y
Generate :
JST* A$IC
Return

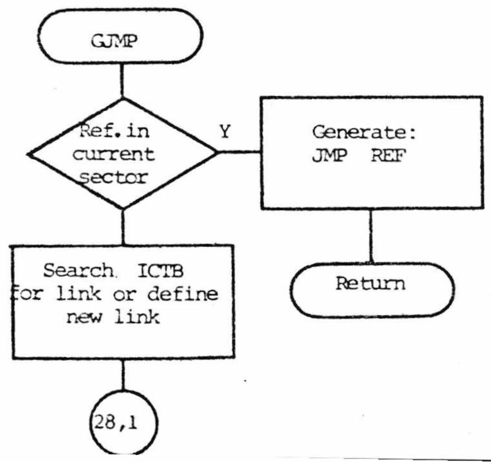

DISPLAY G
Get PAT var. from SS Generate any subscripts
Get variable address in {A}
Generate: STA A$DT
return

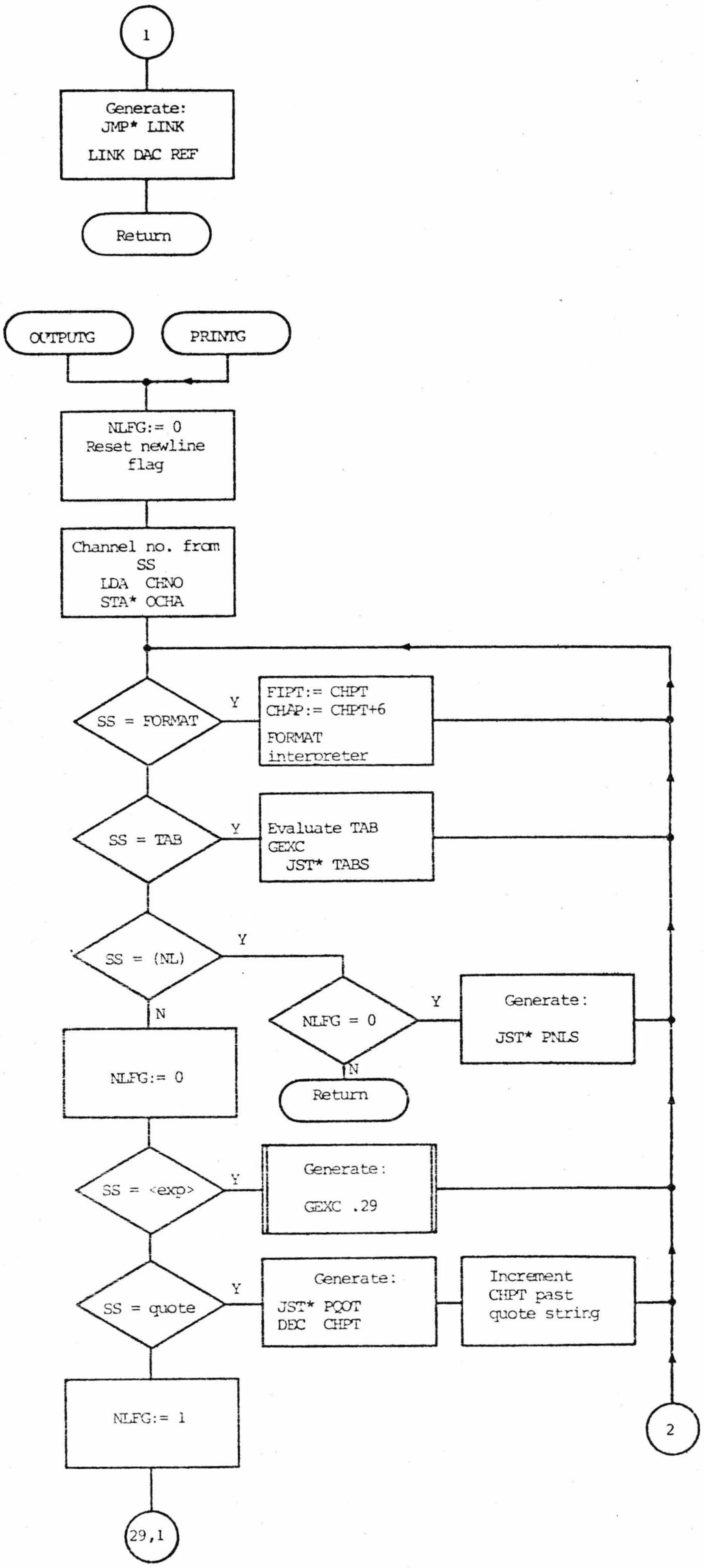

LETG
Generate expr. GEXC .29
SS = 210 Terminator
N
Y
Return


FORG
Get control var. from SS. Push on stack
GEXC to generate for lower bound, result at FORT1
GEXC for upper bound, result at FORT2 Stack FORT2
26,1

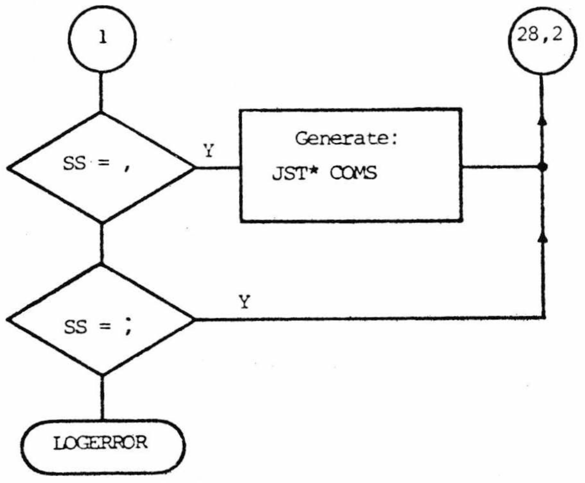

1
SS = 210
Y
FORT3:= 0
GEXC to evaluate step result at FORT3 .29
Stack FORT3
Generate:
F1 LDA FORT1
JST* FSUB
DAC FORT2
Stack F1 address
FORT3 = 0
Y
N
Generate:
JST* FMUL
DAC FORT3
Generate:
SGL
SZE
SPL
SKP
JMP* FZA
Stack F2A address
Return

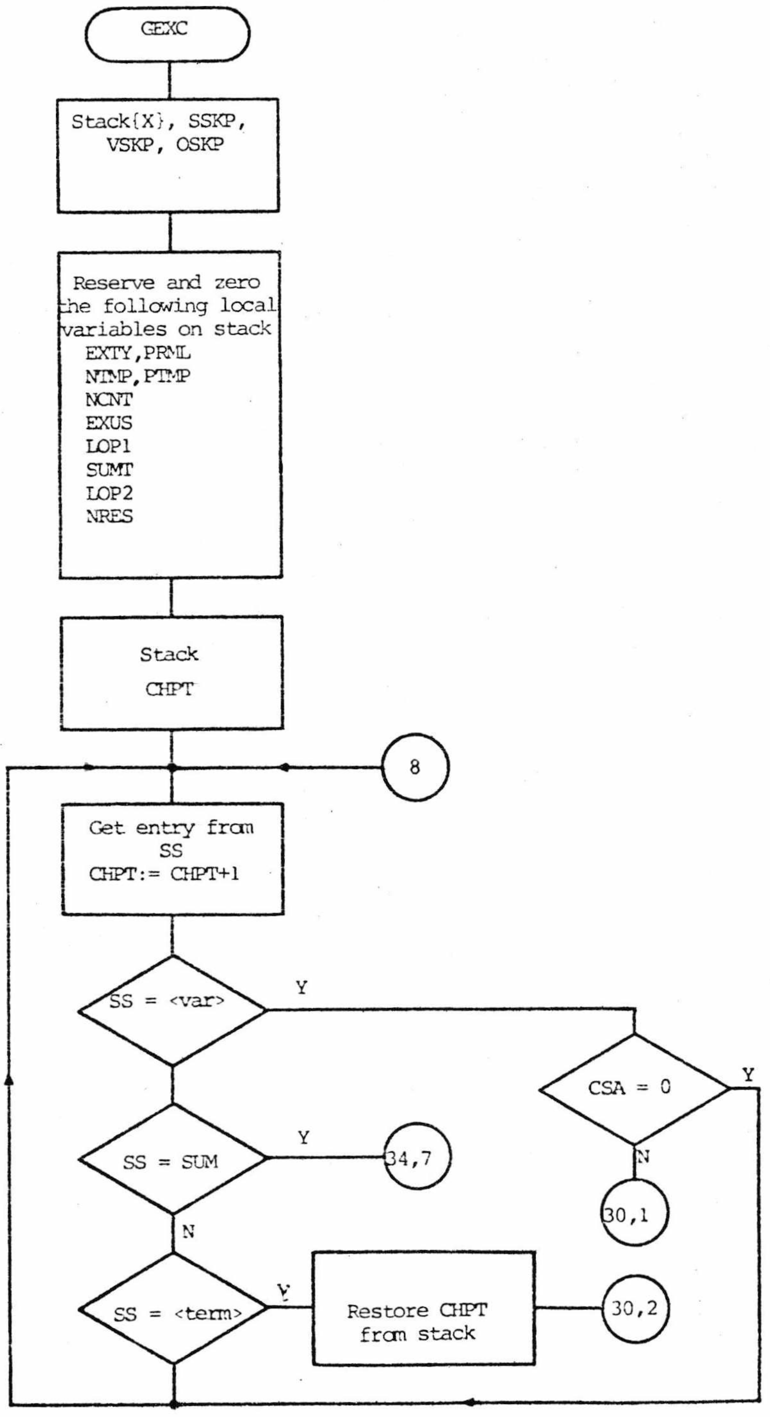

NEXTG
Get var. from SS
CVN = SK1
N
NXER
ERTA
DBL
DLD* FORT1
JST* FADD
FORT3 = 0
Y
Generate:
DAC FORT3
Generate:
DAC FONE
Generate:
DST * FORT1
JMP* F1A
UNSTACK:
CVAR
FORT1
FORT2
FORT3
F1A
F2A
Complete
F2A DAC**
Return

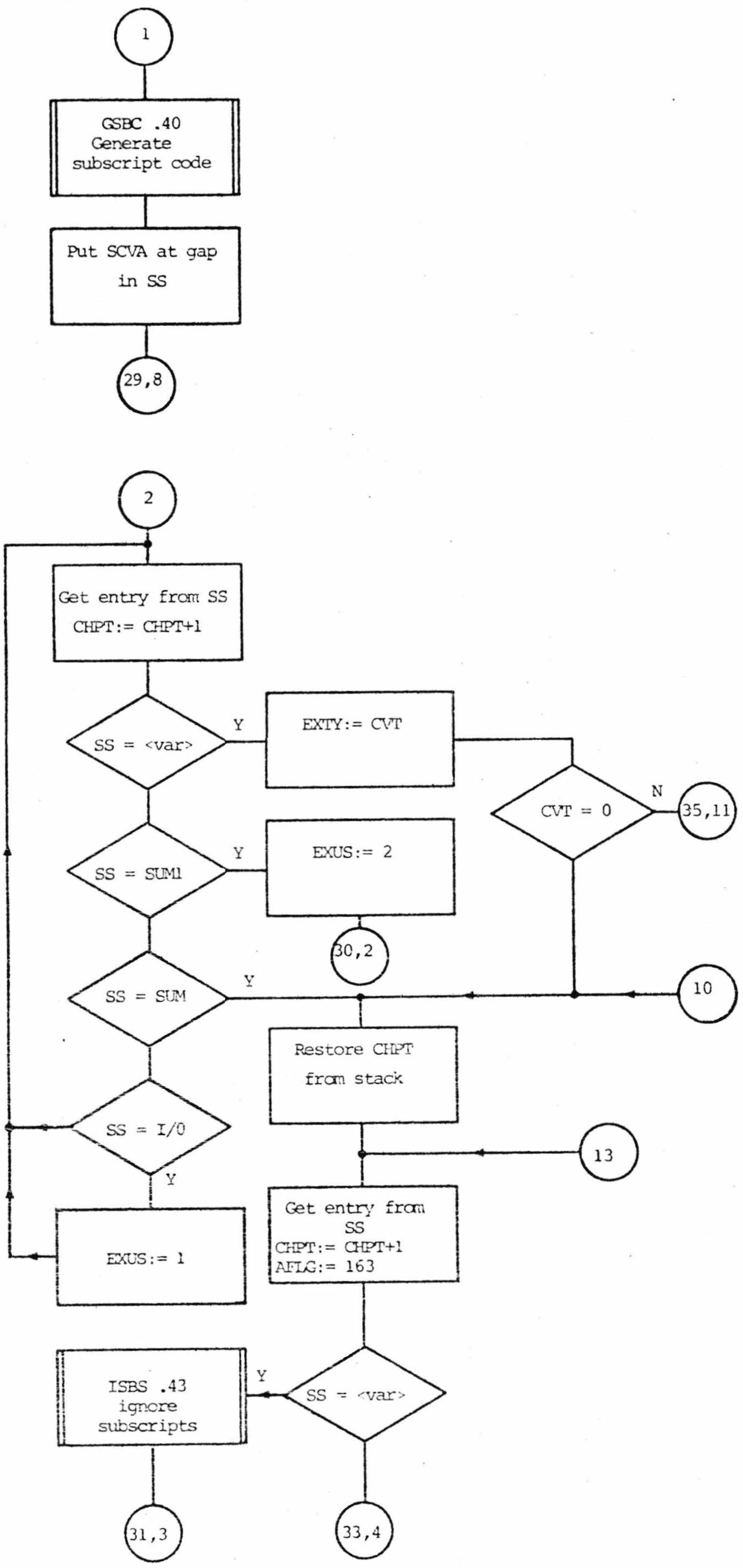

GJMP
Ref. in current sector
Y
Generate:
JMP REF
Return
Search ICTB for link or define new link
28,1

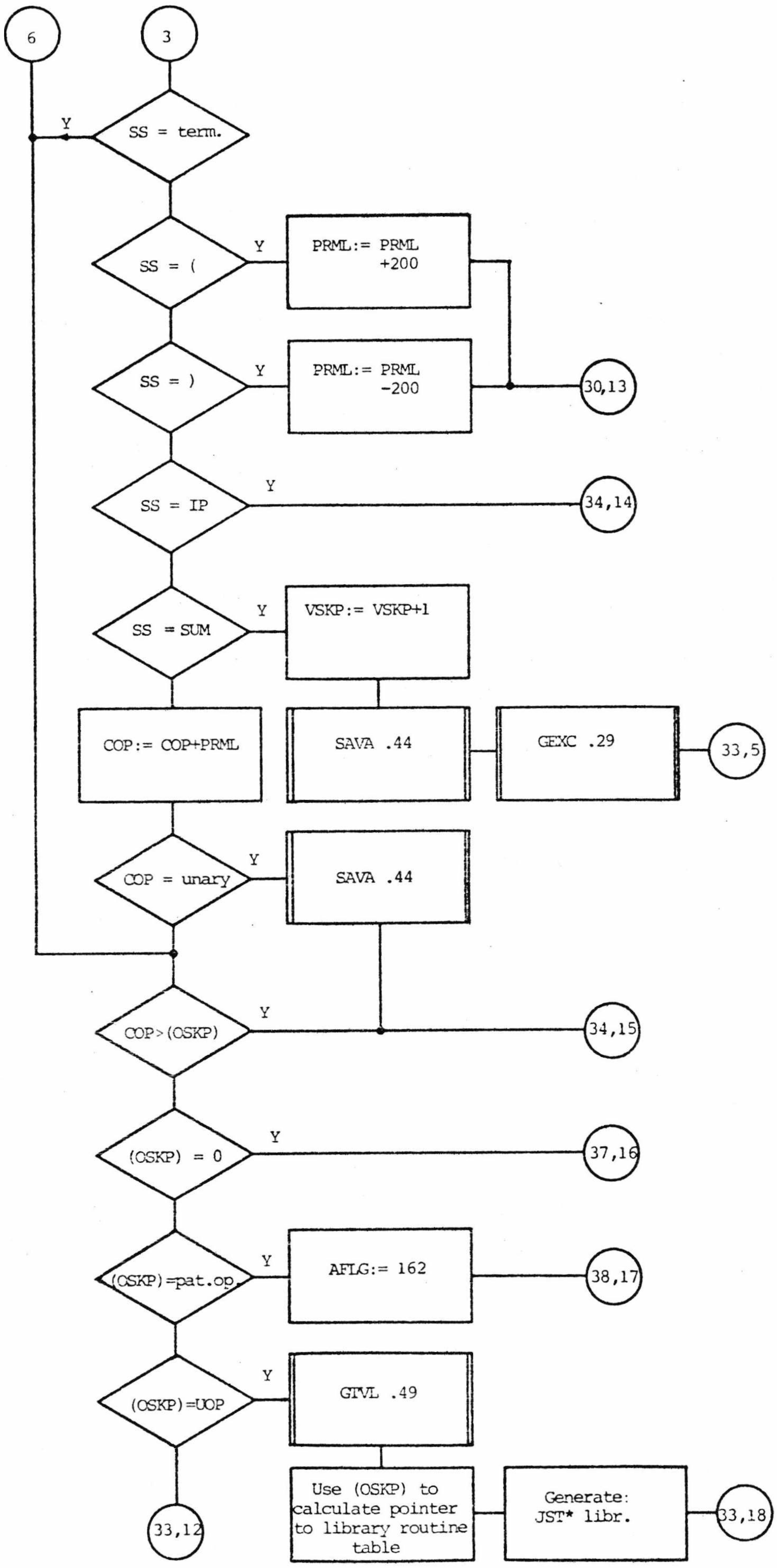

1
Generate:
JMP* LINK
LINK DAC REF
Return
OUTPUTG
PRINTG
NLFG:= 0
Reset newline flag
Channel no. from SS
LDA CHNO
STA* OCHA
SS = FORMAT
Y
FIPT:= CHPT
CHAP:= CHPT+6
FORMAT interpreter
SS = TAB
Y
Evaluate TAB
GEXC
JST* TABS
SS = (NL)
Y
N
NLFG = 0
Y
N
Generate:
JST* PNLS
Return
NLFG:= 0
SS = <exp>
Y
Generate:
GEXC .29
SS = quote
Y
Generate:
JST* PQOT
DEC CHPT
Increment CHPT past quote string
NLFG:= 1
2
29,1

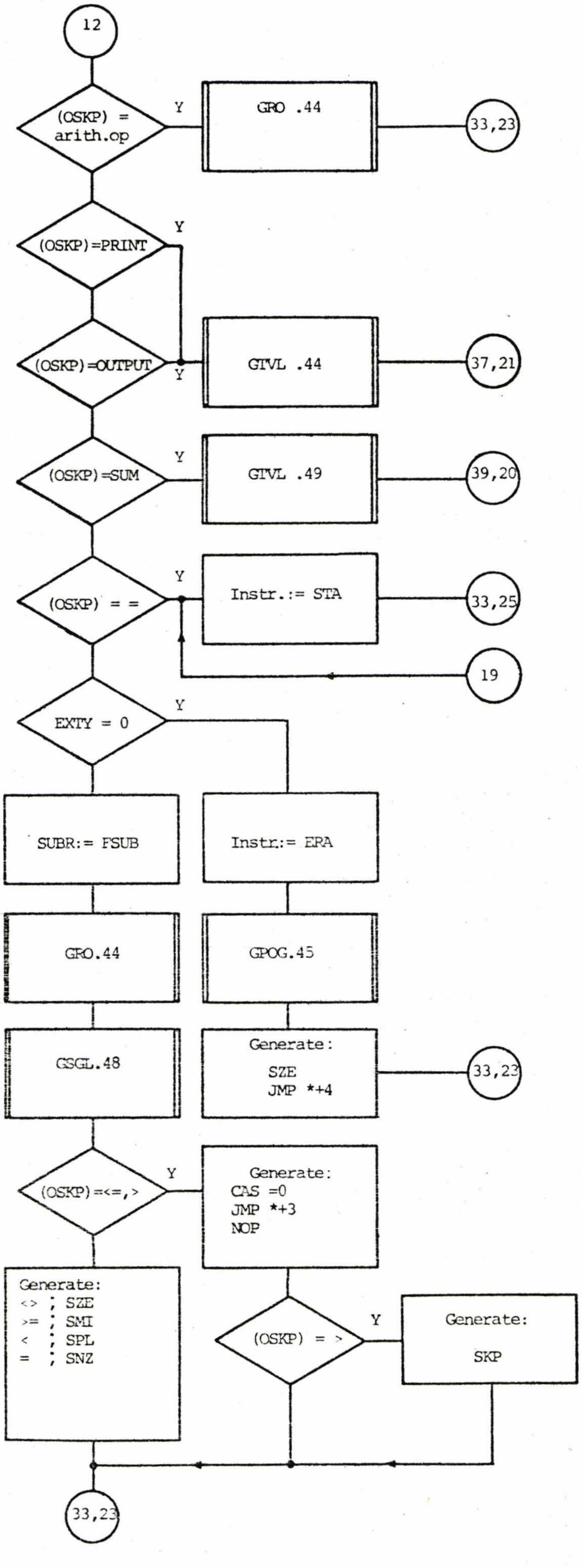
C28

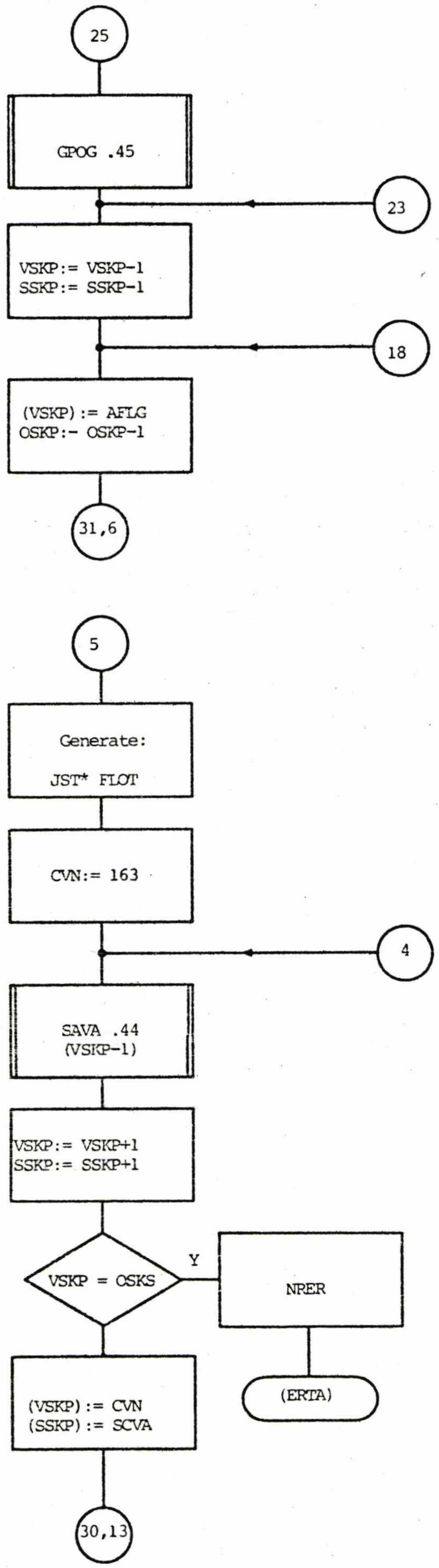

1
SS = ,
Y
Generate:
JST* COMS
28,2
SS = ;
Y
LOGERROR

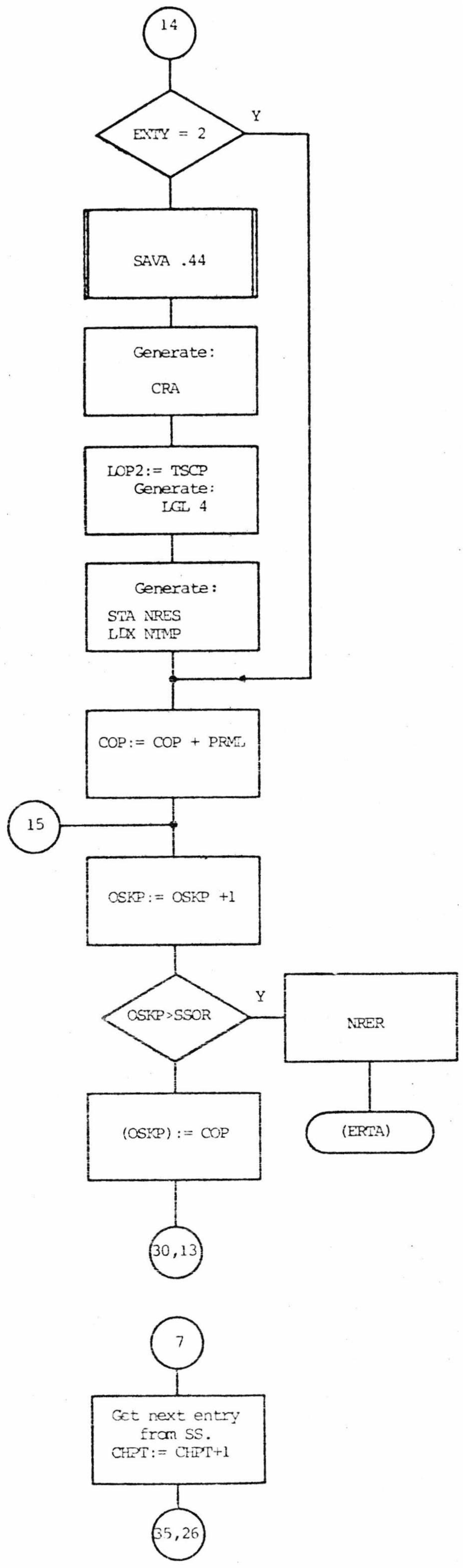

GEXC
Stack{X}, SSKP, VSKP, OSKP
Reserve and zero the following local variables on stack
EXTY,PRML
NTMP,PTMP
NCNT
EXUS
LOP1
SUMT
LOP2
NRES
Stack
CHPT
8
Get entry from SS
CHPT:= CHPT+1
SS = <var>
Y
CSA = 0
Y
N
30,1
SS = SUM
Y
34,7
N
SS = <term>
Y
Restore CHPT from stack
30,2

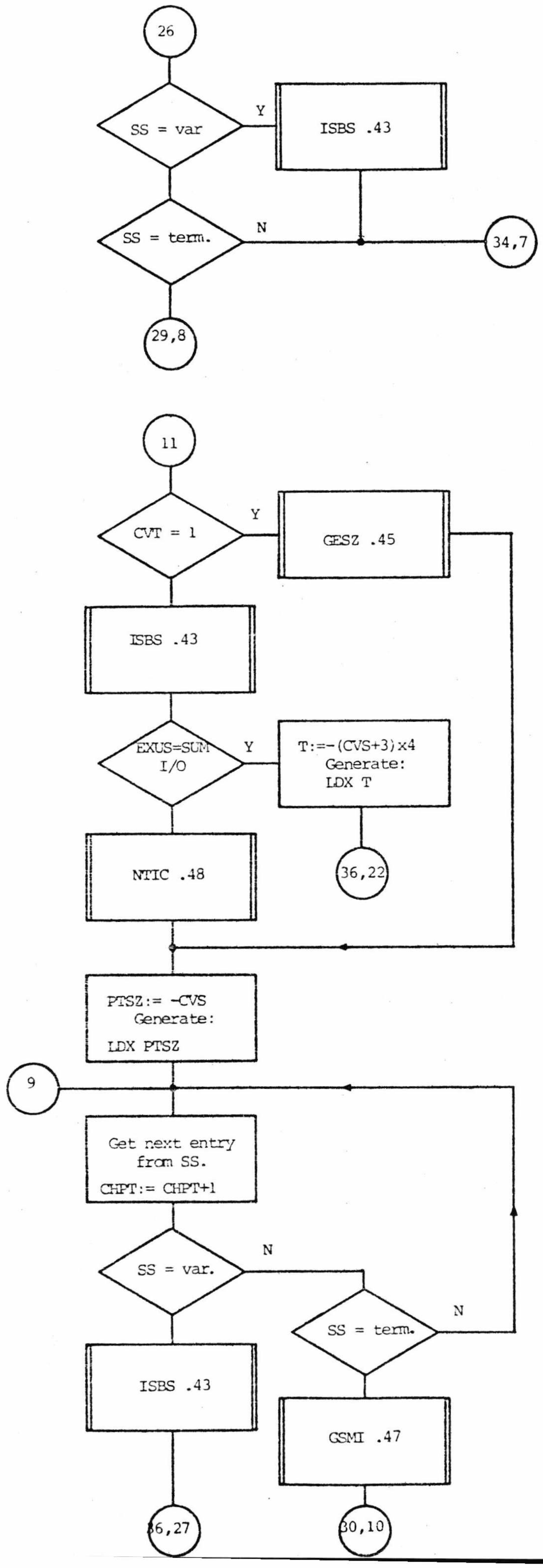

1
GSBC .40
Generate
subscript code
Put SCVA at gap
in SS
29,8
2
Get entry from SS
CHPT:= CHPT+1
SS = <var>
Y
EXTY:= CVT
CVT = 0
N
35,11
SS = SUM1
Y
EXUS:= 2
30,2
SS = SUM
Y
10
Restore CHPT
from stack
SS = I/0
Y
13
EXUS:= 1
Get entry from
SS
CHPT:= CHPT+1
AFLG:= 163
ISBS .43
ignore
subscripts
Y
SS = <var>
31,3
33,4


6
3
SS = term.
Y
SS = (
Y
PRML:= PRML +200
SS =)
Y
PRML:= PRML -200
30,13
SS = IP
Y
34,14
SS = SUM
Y
VSKP:= VSKP+1
COP:= COP+PRML
SAVA .44
GEXC .29
33,5
COP = unary
Y
SAVA .44
COP>(OSKP)
Y
34,15
(OSKP) = 0
Y
37,16
(OSKP)=pat.op.
Y
AFLG:= 162
38,17
(OSKP)=UOP
Y
GTVL .49
Use (OSKP) to calculate pointer to library routine table
Generate: JST* libr.
33,18
33,12


12
(OSKP) = arith.op
Y
GRO .44
33,23
(OSKP)=PRINT
Y
(OSKP)=OUTPUT
Y
GTVL .44
37,21
(OSKP)=SUM
Y
GTVL .49
39,20
(OSKP) = =
Y
Instr.:= STA
33,25
19
EXTY = 0
Y
SUBR:= FSUB
Instr.:= ERA
GRO.44
GPOG.45
GSGL.48
Generate:
SZE
JMP *+4
33,23
(OSKP)=<=,>
Y
Generate:
CAS =0
JMP *+3
NOP
Generate:
<> ; SZE
>= ; SMI
< ; SPL
= ; SNZ
(OSKP) = >
Y
Generate:
SKP
33,23

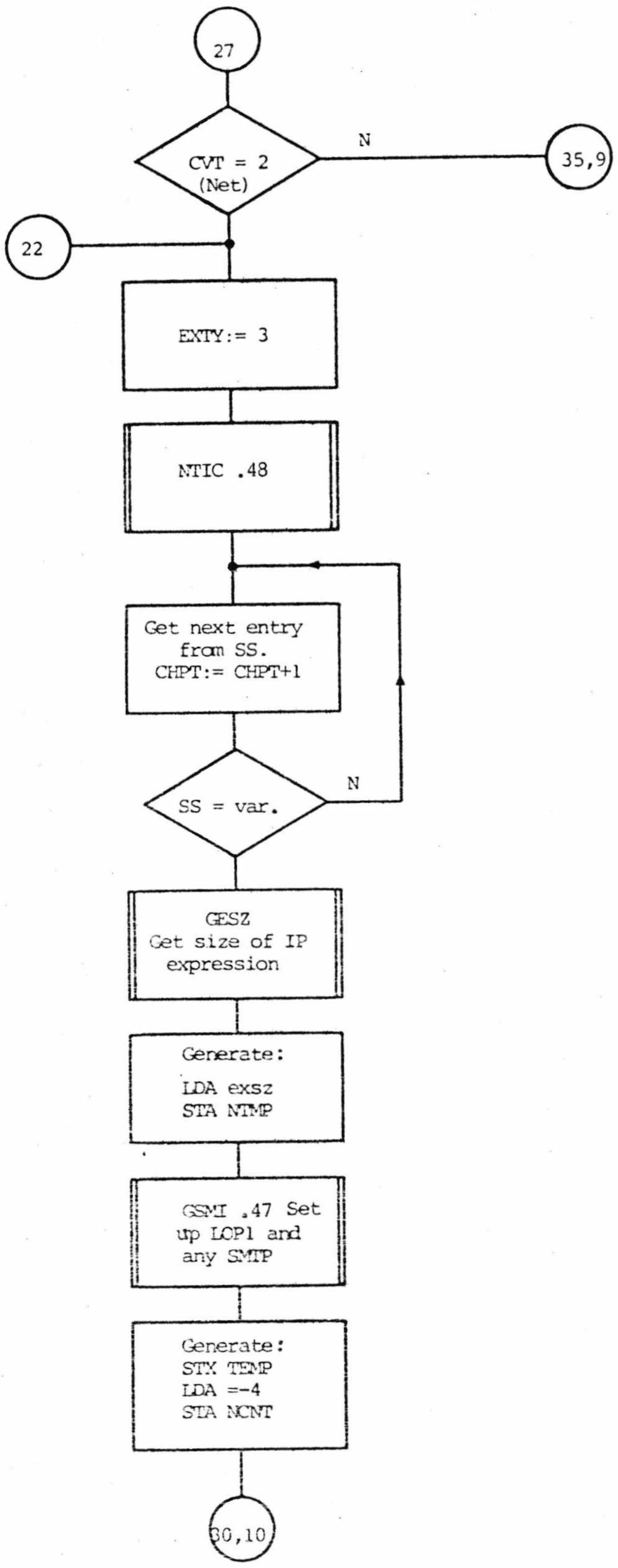

25
GPOG .45
23
VSKP:= VSKP-1
SSKP:= SSKP-1
18
(VSKP) := AFLG
OSKP:- OSKP-1
31,6
5
Generate:
JST* FLOT
CVN:= 163
4
SAVA .44
(VSKP-1)
VSKP:= VSKP+1
SSKP:= SSKP+1
VSKP = OSKS
Y
NRER
(ERTA)
(VSKP) := CVN
(SSKP) := SCVA
30,13

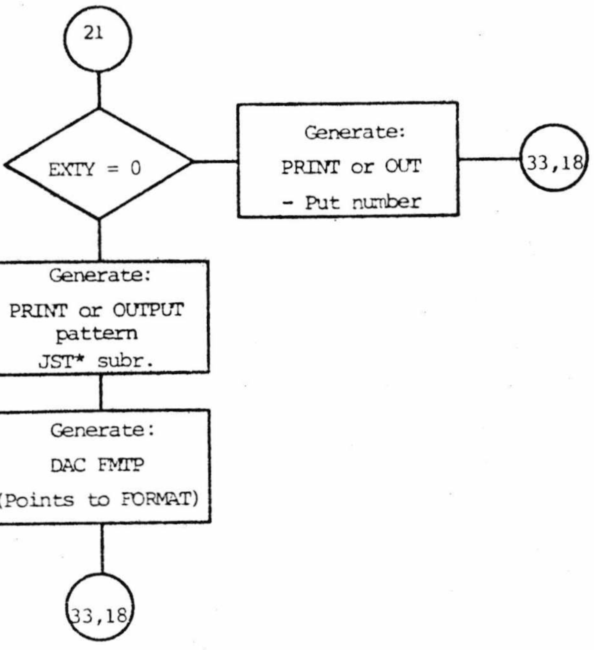

14
EXTY = 2
Y
SAVA .44
Generate:
CRA
LOP2:= TSCP
Generate:
LGL 4
Generate:
STA NRES
LDX NTMP
COP:= COP + PRML
15
OSKP:= OSKP +1
OSKP>SSOR
Y
NRER
(ERTA)
(OSKP) := COP
30,13
7
Get next entry
from SS.
CHPT:= CHPT+1
35,26

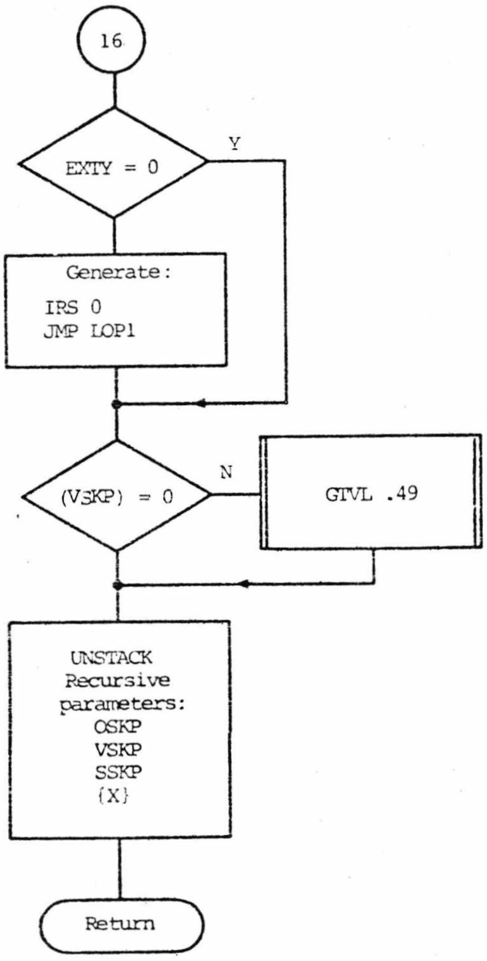


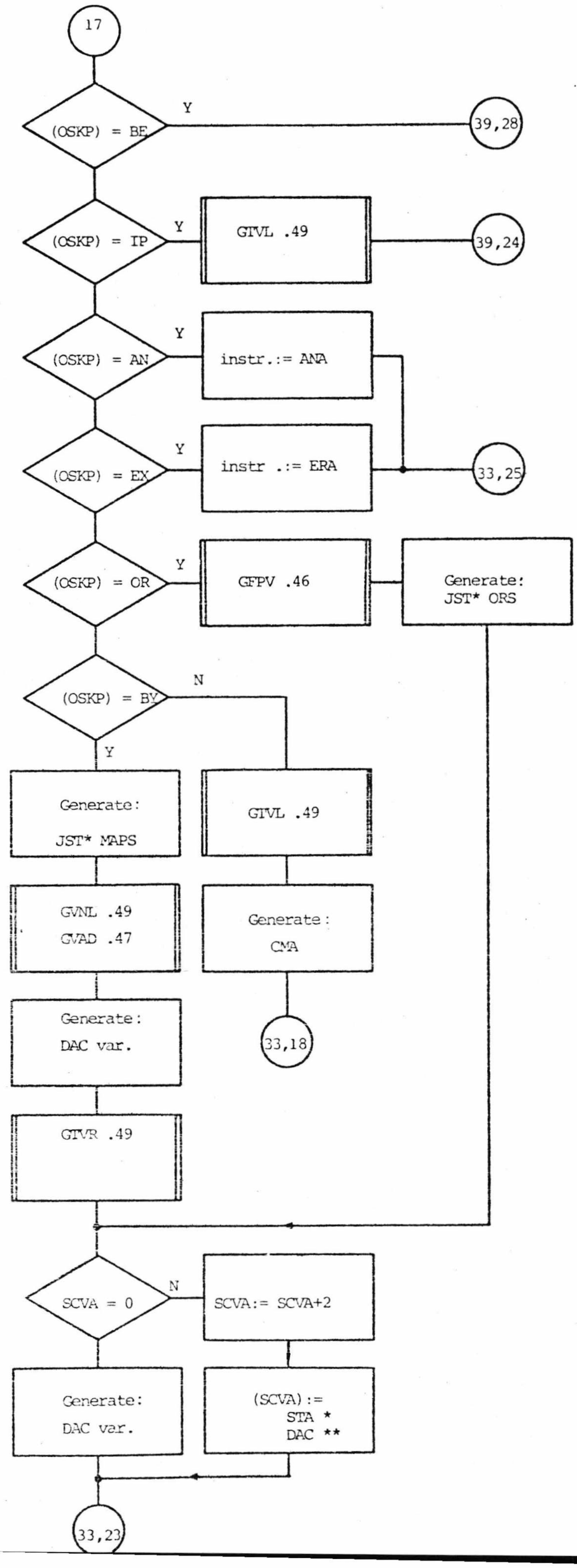
26
SS = var
Y
ISBS .43
SS = term.
N
34,7
29,8
11
CVT = 1
Y
GESZ .45
ISBS .43
EXUS=SUM I/O
Y
T:=-(CVS+3)x4 Generate: LDX T
36,22
NTIC .48
PTSZ:= -CVS Generate: LDX PTSZ
9
Get next entry from SS. CHPT:= CHPT+1
SS = var.
N
SS = term.
N
ISBS .43
GSMI .47
36,27
30,10

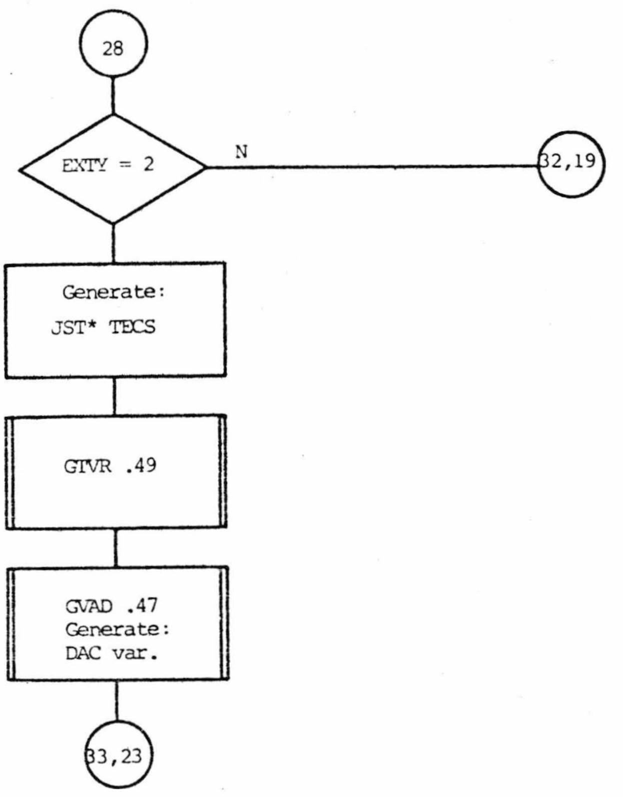
C35

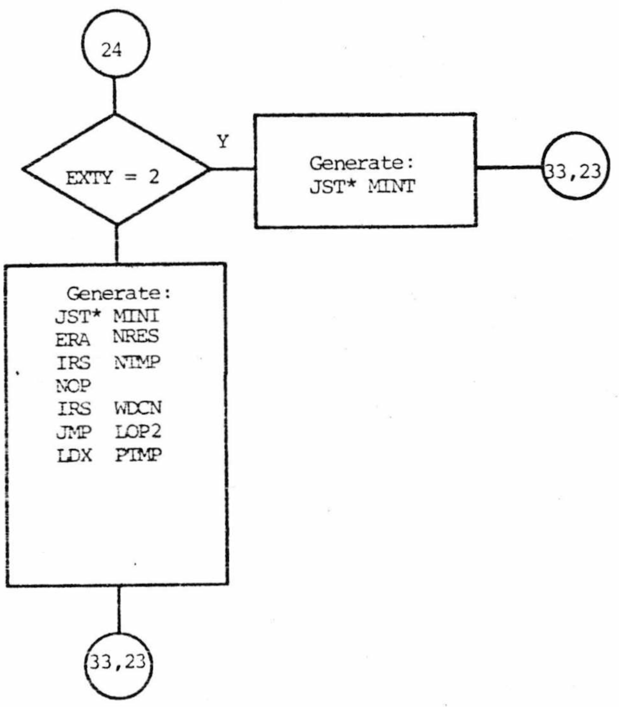

27
CVT = 2 (Net)
N
35,9
22
EXTY:= 3
NTIC .48
Get next entry from SS. CHPT:= CHPT+1
SS = var.
N
GESZ Get size of IP expression
Generate: LDA exsz STA NTMP
GSMI .47 Set up LCP1 and any SMTP
Generate: STX TEMP LDA =-4 STA NCNT
30,10

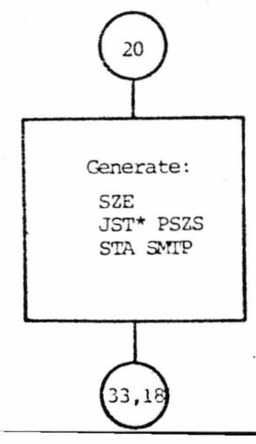

21
EXTY = 0
Generate:
PRINT or OUT
- Put number
33,18
Generate:
PRINT or OUTPUT
pattern
JST* subr.
Generate:
DAC FMTP
(Points to FORMAT)
33,18

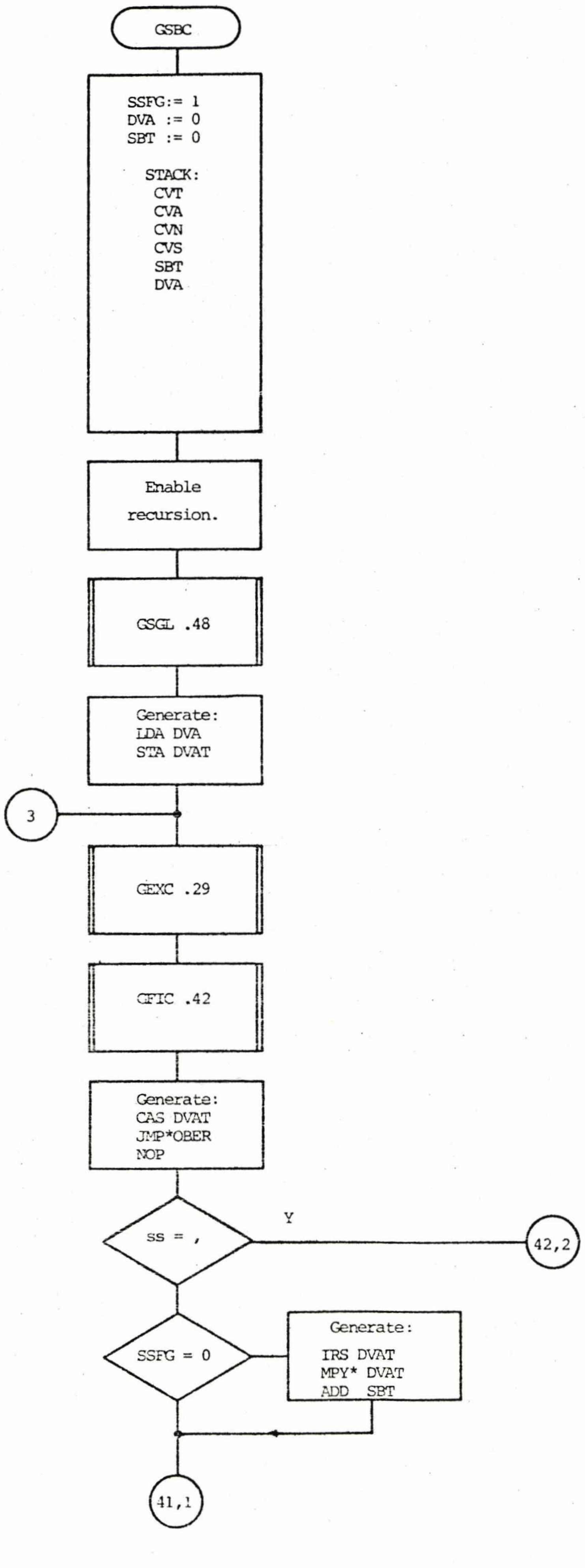

16
EXTY = 0
Y
Generate:
IRS 0
JMP LOP1
(VSKP) = 0
N
GTVL .49
UNSTACK
Recursive
parameters:
OSKP
VSKP
SSKP
(X)
Return

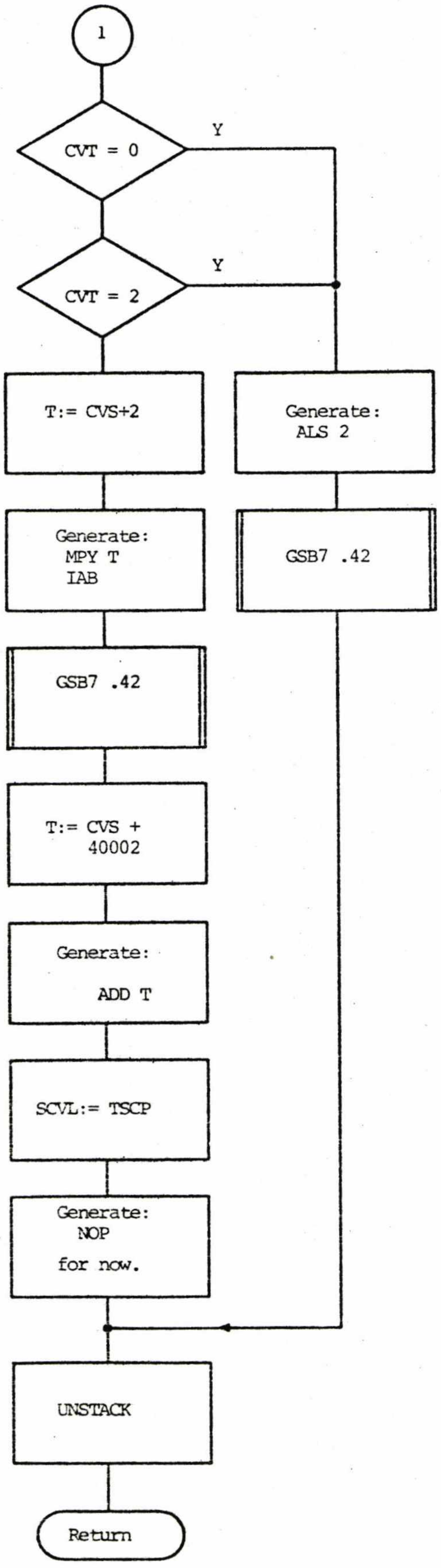

17
(OSKP) = BE
Y
39,28
(OSKP) = IP
Y
GTVL .49
39,24
(OSKP) = AN
Y
instr.:= ANA
(OSKP) = EX
Y
instr .:= ERA
33,25
(OSKP) = OR
Y
GFPV .46
Generate: JST* ORS
(OSKP) = BY
N
Y
Generate: JST* MAPS
GTVL .49
GVNL .49
GVAD .47
Generate: CMA
Generate: DAC var.
33,18
GTVR .49
SCVA = 0
N
SCVA:= SCVA+2
Generate: DAC var.
(SCVA):= STA * DAC **
33,23

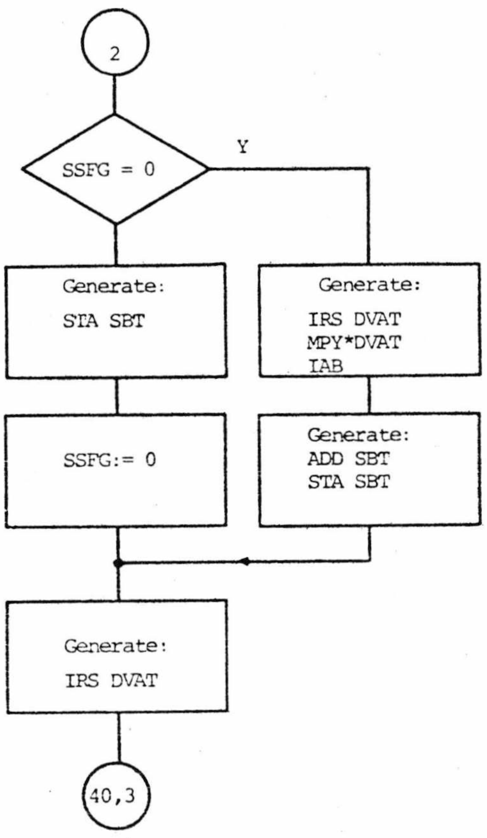

28
EXTY = 2
N
32,19
Generate:
JST* TECS
GTVR .49
GVAD .47
Generate:
DAC var.
33,23

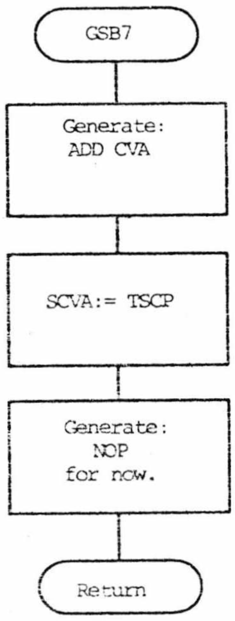

24
EXTY = 2
Y
Generate:
JST* MINT
33,23
Generate:
JST* MINI
ERA NRES
IRS NTMP
NOP
IRS WDCN
JMP LOP2
LDX PTMP
33,23

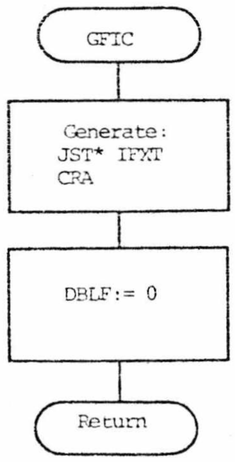

20
Generate:
SZE
JST* PSZS
STA SMTP
33,18

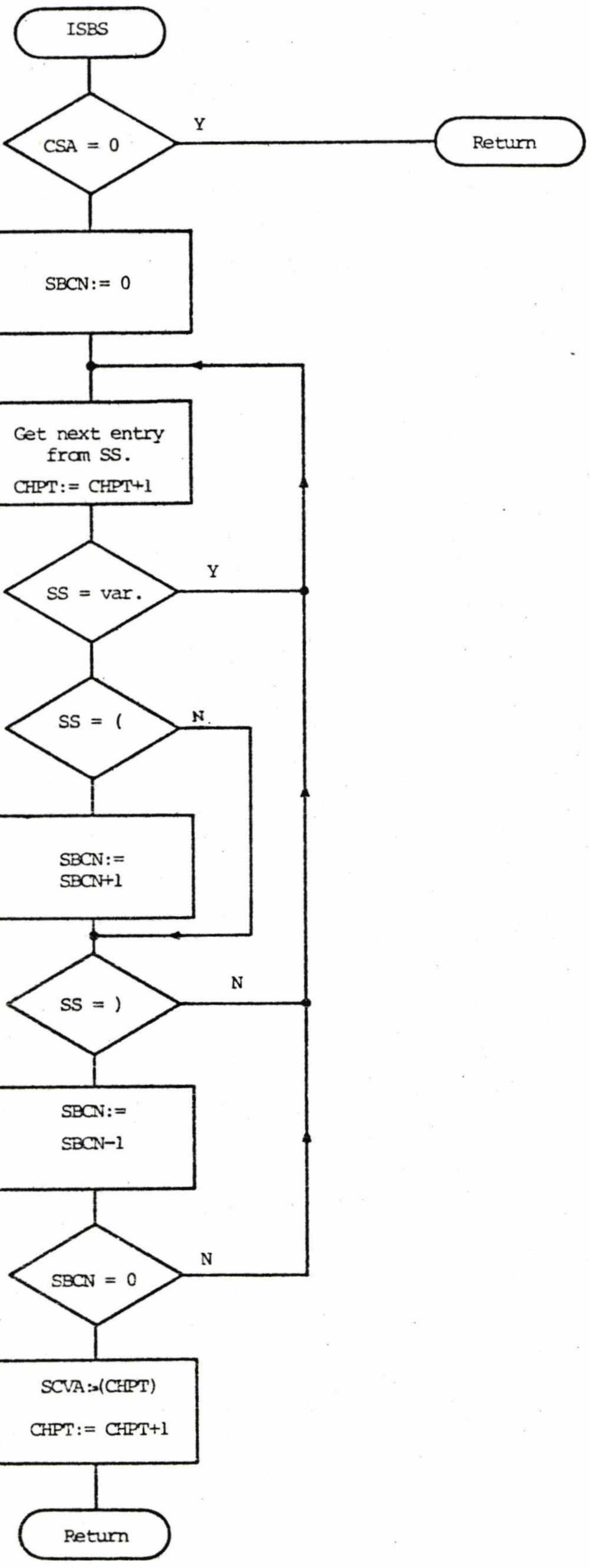

GSBC
SSFG:= 1
DVA := 0
SBT := 0
STACK:
CVT
CVA
CVN
CVS
SBT
DVA
Enable recursion.
GSGL .48
Generate:
LDA DVA
STA DVAT
3
GEXC .29
GFIC .42
Generate:
CAS DVAT
JMP*OBER
NOP
ss = ,
Y
42,2
SSFG = 0
Generate:
IRS DVAT
MPY* DVAT
ADD SBT
41,1

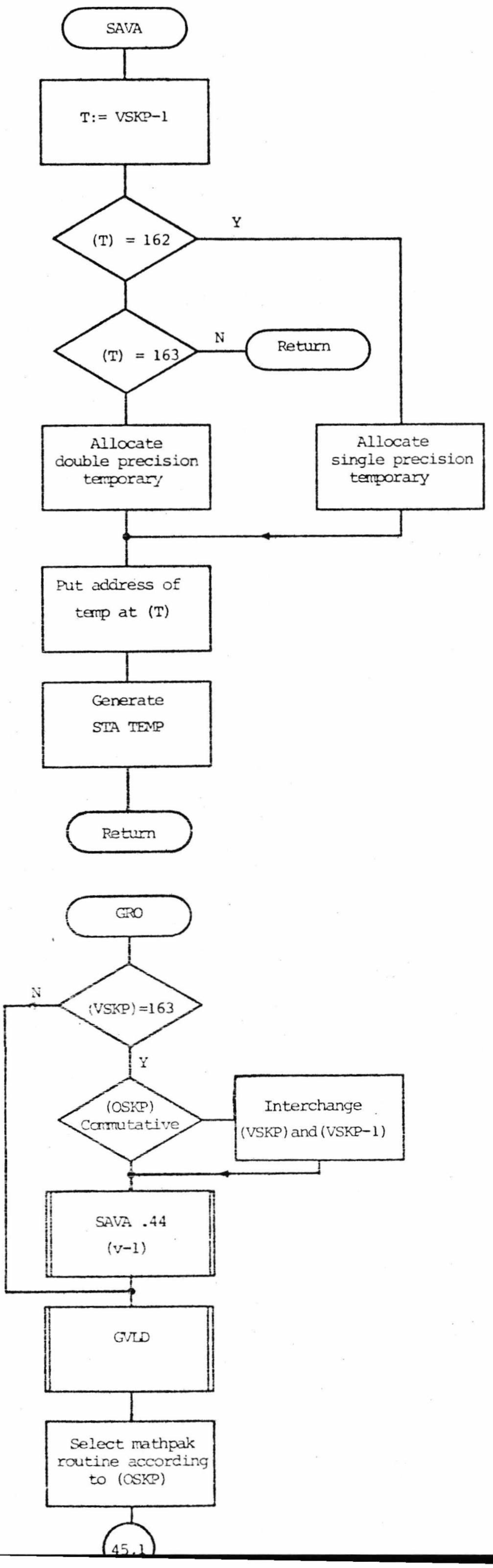

1
CVT = 0
Y
CVT = 2
Y
T:= CVS+2
Generate:
ALS 2
Generate:
MPY T
IAB
GSB7 .42
GSB7 .42
T:= CVS +
40002
Generate:
ADD T
SCVL:= TSCP
Generate:
NOP
for now.
UNSTACK
Return

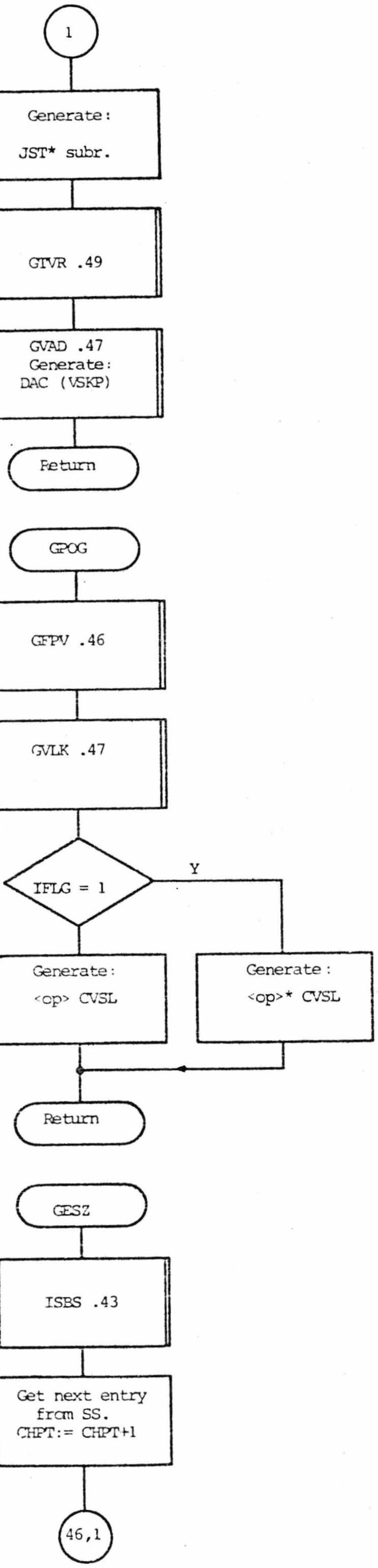

2
SSFG = 0
Y
Generate:
STA SBT
Generate:
IRS DVAT
MPY*DVAT
IAB
SSFG:= 0
Generate:
ADD SBT
STA SBT
Generate:
IRS DVAT
40,3


GSB7
Generate:
ADD CVA
SCVA:= TSCP
Generate:
NOP
for now.
Return

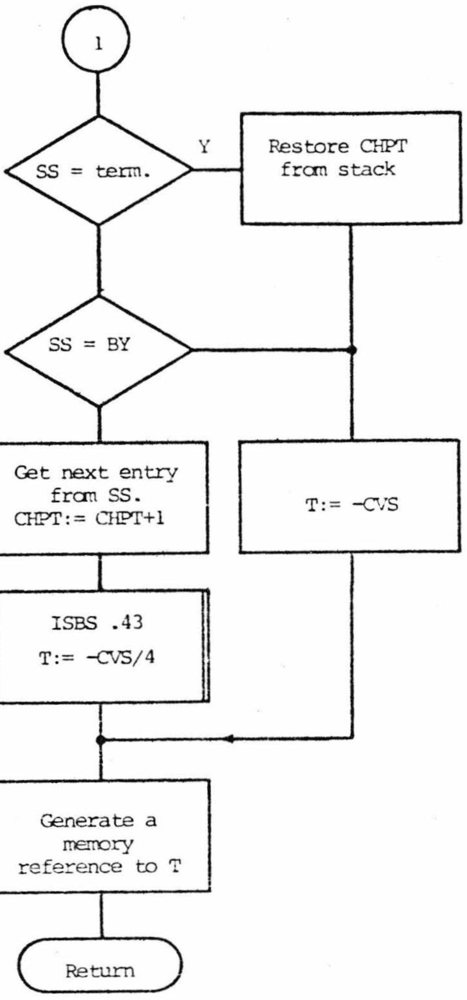

GFIC
Generate:
JST* IFXT
CRA
DBLF:= 0
Return

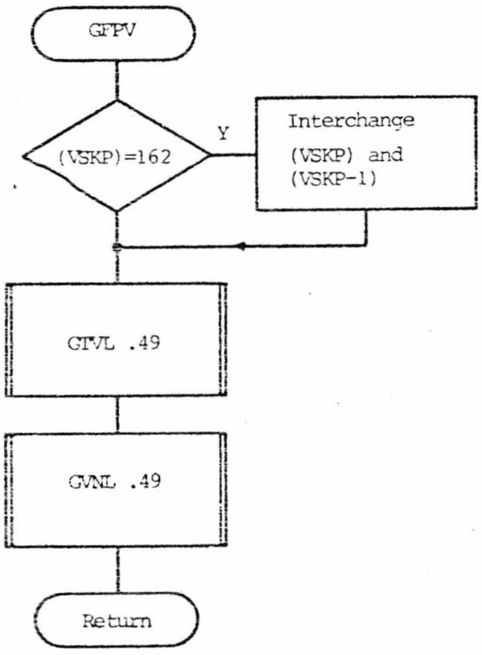

ISBS
CSA = 0
Y
Return
SBCN:= 0
Get next entry from SS.
CHPT:= CHPT+1
SS = var.
Y
SS = (
N
SBCN:= SBCN+1
SS =)
N
SBCN:= SBCN-1
SBCN = 0
N
SCVA:=(CHPT)
CHPT:= CHPT+1
Return

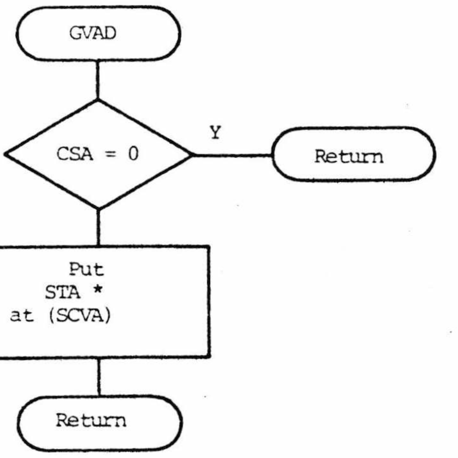

SAVA
T:= VSKP-1
(T) = 162
Y
(T) = 163
N
Return
Allocate double precision temporary
Allocate single precision temporary
Put address of temp at (T)
Generate STA TEMP
Return
GRO
(VSKP)=163
N
Y
(OSKP) Commutative
Interchange (VSKP) and (VSKP-1)
SAVA .44 (v-1)
GVLD
Select mathpak routine according to (OSKP)
45.1

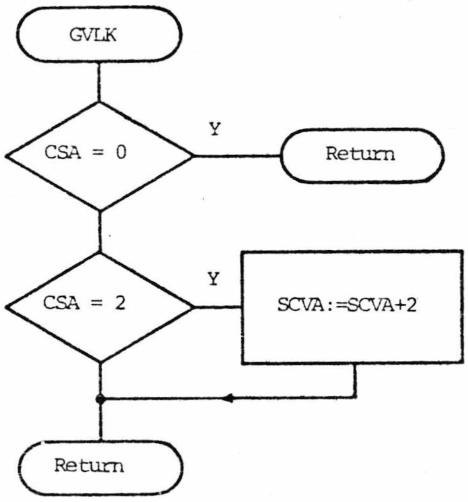


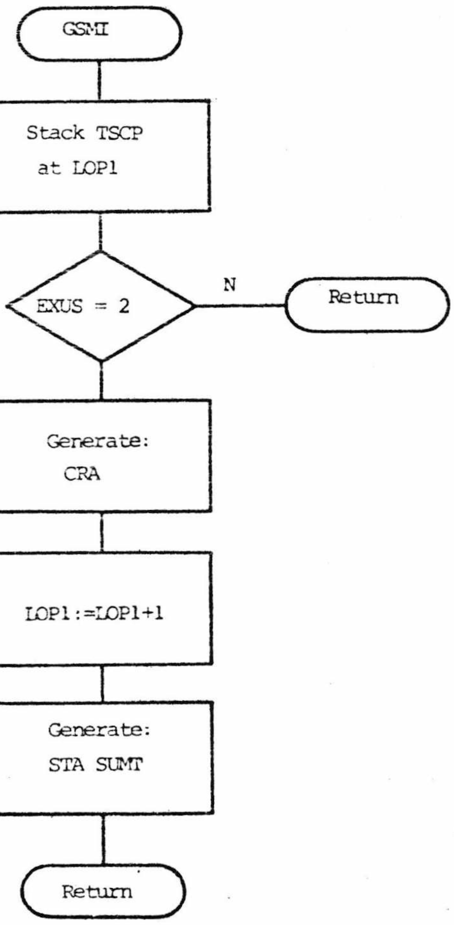
1
Generate:
JST* subr.
GTVR .49
GVAD .47
Generate:
DAC (VSKP)
Return
GPOG
GFPV .46
GVLK .47
IFLG = 1
Y
Generate:
<op> CVSL
Generate:
<op>* CVSL
Return
GESZ
ISBS .43
Get next entry
from SS.
CHPT:= CHPT+1
46,1

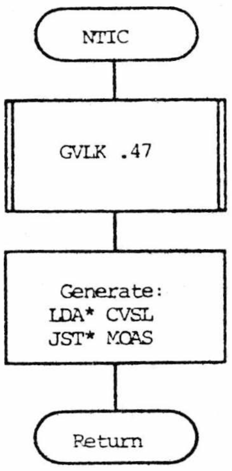

1
SS = term.
Y
Restore CHPT from stack
SS = BY
Get next entry from SS. CHPT:= CHPT+1
T:= -CVS
ISBS .43
T:= -CVS/4
Generate a memory reference to T
Return

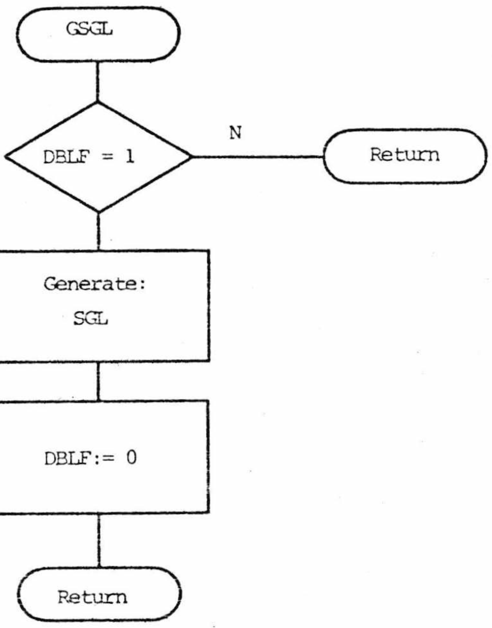

GFPV
(VSKP)=162
Y
Interchange (VSKP) and (VSKP-1)
GTVL .49
GVNL .49
Return

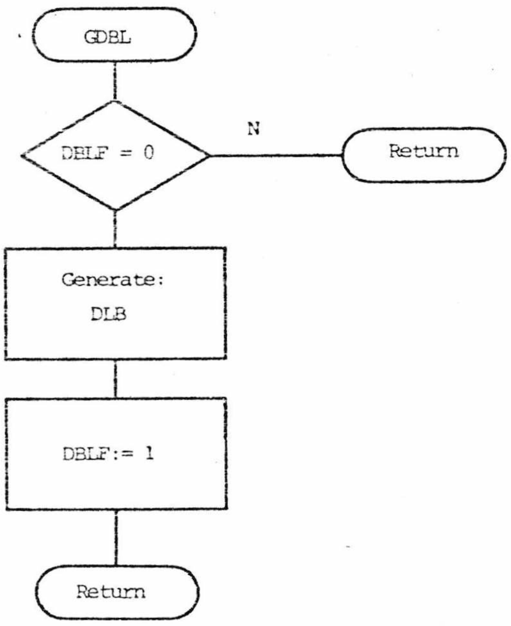

GVAD
CSA = 0
Y
Return
Put STA * at (SCVA)
Return

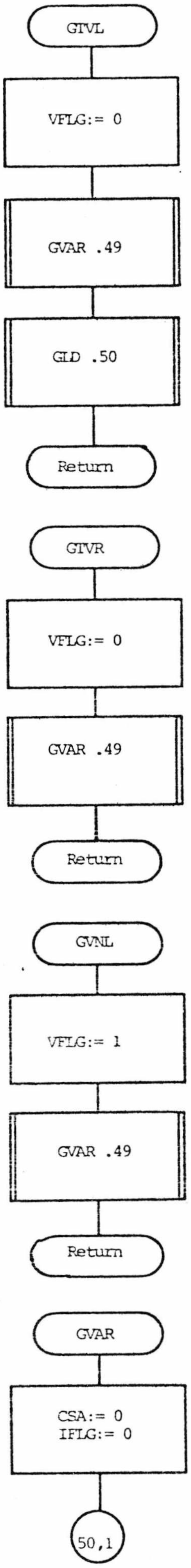

GVLK
CSA = 0
Y
Return
CSA = 2
Y
SCVA:=SCVA+2
Return

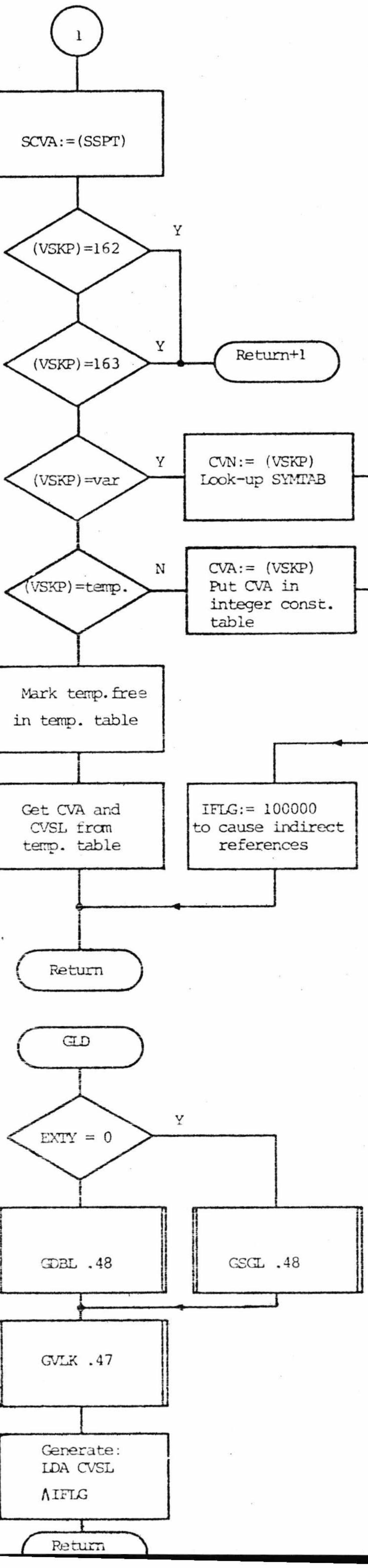

GSMI
Stack TSCP at LOP1
EXUS = 2
N
Return
Generate: CRA
LOP1:=LOP1+1
Generate: STA SUMT
Return


NTIC
GVLK .47
Generate:
LDA* CVSL
JST* MOAS
Return


GSGL
DBLF = 1
N
Return
Generate:
SGL
DBLF:= 0
Return


GDBL
DBLF = 0
N
Return
Generate:
DLB
DBLF:= 1
Return


GTVL
VFLG:= 0
GVAR .49
GLD .50
Return
GTVR
VFLG:= 0
GVAR .49
Return
GVNL
VFLG:= 1
GVAR .49
Return
GVAR
CSA:= 0
IFLG:= 0
50,1


1
SCVA:=(SSPT)
(VSKP)=162
Y
(VSKP)=163
Y
Return+1
(VSKP)=var
Y
CVN:= (VSKP)
Look-up SYMTAB
(VSKP)=temp.
N
CVA:= (VSKP)
Put CVA in
integer const.
table
Mark temp.free
in temp. table
Get CVA and
CVSL from
temp. table
IFLG:= 100000
to cause indirect
references
Return
GLD
EXTY = 0
Y
GDBL .48
GSGL .48
GVLK .47
Generate:
LDA CVSL
∧IFLG
Return

GLOSSARY

AAC	-	Address all cards
ACFG	-	Already compiled flag
ADMOS	-	A disc and magnetic tape operating system
AFLG	-	Accumulator type flag
ALC	-	Adaptive logic circuit
ASR	-	Address same register
CAR	-	Card address register
CCAR	-	Clear card address register
CEER	-	Command syntax error
CES	-	Current expression size
CHPT	-	Character string pointer
CIC	-	Current input channel
CKSI	-	Check if statement is immediate
CMEP	-	Command entry point
CMFG	-	Compile only flag
COMS	-	Comma interpreter routine
COP	-	Current operator
CR	-	Carriage return
CREG	-	Control register (same as FNREG)
CSA	-	Current subscript address
CSCP	-	Current sector, code pointer
CSIAR	-	Control store index address register
CSIR	-	Control store instruction register
CSVP	-	Current sector, variables pointer
CTDR	-	Clear teach data register
CTMR	-	Clear teach mask register
CULN	-	Current line number

CVA	-	Current variable address
CVD	-	Current variable dope vector
CVN	-	Current variable name
CVS	-	Current variable size
CVSL	-	Current variable sector link
CVT	-	Current variable type
DBLF	-	Double precision flag
DNER	-	Decimal number error
DRSS	-	Data/result specification string
DVA	-	Dope vector base address
DVAT	-	Temporary for dope vector address
DVPT	-	Dope vector pointer
ENDI	-	End interpreter routine
ERTA	-	Error return address
EXPX	-	Extract pattern expression
EXTY	-	Expression type e.g. real, pattern, net, teach
EXUS	-	Expression use e.g. SUM, PRINT, OUTPUT
EXV	-	Extract variable
EXVN	-	Extract variable name
EXVT	-	Extract variable type
EXXP	-	Extract arithmetic expression
FLOT	-	Converts integer to floating point
FMUL	-	Floating print multiply
FRAD	-	Forward reference table address
GESZ	-	Get expression size routine
GEXC	-	Generate expression code routine
GFIC	-	Generate floating-point to integer code
GFPV	-	Generate for pattern variables getting top two arguments

GLD	-	Generate to load current argument into accumulator
GLN	-	Get line number and its run-time address
GPOG	-	Generate pattern operator reducing stacks
GRO	-	Generate and reduce for arguments calling a library routine
GSBC	-	Generate subscript evaluation code
GSGL	-	Go single precision mode
GSMI	-	Generate SUM and set up loop
GTVL	-	Get top variable, loading accumulator
GTVR	-	Get top variable, not loading
GVAD	-	Get address of variable or array element
GVAR	-	Get penultimate variable, loading accumulator.
GVLK	-	Generate an intersector link for a variable
GVNL	-	Get penultimate variate, not loading
Hd	-	Head of string to lexical or syntax analyser
HIAD	-	Top limit of workspace
HRAD	-	Half-rack address
ICAR	-	Increment card address register
ICTB	-	Integer constant table
IFFG	-	If statement flag
IFLG	-	Indirect instruction generation flag
IFXT	-	Integer to floating-point conversion routine
INT	-	Integer
IPLN	-	Input program line entry-point
IPSB	-	Input runtime subroutine
ISBS	-	Ignore subscript blocks in s-string
ISS	-	Immediate statement s-strong.
LCAR	-	Load card address register
LETG	-	LET statement generator
LNDR	-	Load network data register

LNER	-	Line number error
LOP1	-	Pattern expression loop
LOP2	-	Net result loop
LOS	-	Layer one set
LSI	-	Large scale integration
LTDR	-	Load teach data register
LTMR	-	Load teach mask register
LWAD	-	Low limit of workspace
M)ER	-	Missing) error
M(ER	-	Missing (error
MAKS	-	Manual/Auto keyswitch
MAP	-	Map operator allowed flag
MAS	-	Manual/Auto select
MD	-	Map domain
MINI	-	MINERVA initialisation routine
MINT	-	MINERVA teach routine
MNAT	-	Manual/Auto toggle
MR	-	Map range
MRR1	-	MINERVA response register layer 1
MRR2	-	MINERVA response register layer 2
MSI	-	Medium scale integration
NCNT	-	Net loop counter
NDR	-	Net data register
NEC	-	Network element card
NETF	-	Net variable allow flag
NETRSZ	-	Net variable result size
NRER	-	No room
NRES	-	Net response temporary variable
NTMP	-	Net loop counter temporary variable
NXER	-	Unmatched NEXT statement

OBER	-	Subscript out of bounds
ONRA	-	ON range address
ONR	-	ON range
OPER	-	Operator error
ORS	-	OR subroutine
OSKP	-	Operator stack pointer
PRML	-	Precedence multiplier
PSFG	-	Pause flag
PSIZ	-	Pattern size routine
PSZS	-	Pattern size routine link
PTHC	-	Parenthesis counter
PTMP	-	Pattern expression loop counter temporary
PTSZ	-	Pattern variable size
RAC1	-	Reset all cards layer 1
RAC2	-	Reset all cards layer 2
RCNT	-	ON stmt: range counter
RELOP	-	Relational operator
RETN	-	RETURN interpreter routine
RLN	-	Referenced line number
SAVA	-	Save accumulator in a temporary variable
SAC1	-	Set all cards layer 1
SAC2	-	Set all cards layer 2
SBCN	-	Subscript dimension counter
SCOM	-	Statement compiled entry-point
SCR	-	SCRATCH
SCVA	-	Subscripted current variable address
SFER	-	Subscript format error
SIFG	-	Immediate statement flag.

SLAM	-	Stored Logic Adaptive Microcircuit
SNPT	-	Statement number table pointer
SNTB	-	Statement number table
SOP	-	Start of compiled program
SOPI	-	Start of immediate compiled program
SOSK	-	Start of operators stack
SS	-	Source-string (S-string)
SSKP	-	System's stack pointer
SSOR	-	S-string origin address
SUMT	-	SUM temporary for accumulating SUM result
SVSK	-	Start of variables stack
TE1	-	Teach enable layer 1
TE2	-	Teach enable layer 2
TECS	-	Teach card subroutine
TSCP	-	Temporary sector code pointer
TSVP	-	Temporary sector variable pointer
TTL	-	Transistor-Transistor Logic
TYER	-	Text uncompiled
ULC	-	Universal logic circuit
UDP	-	Unary operator
VFLG	-	Variable flag
VNER	-	Variable name error
VSER	-	Variable size error
VSKP	-	Variables stack pointer
VTER	-	Variable type incompatibility
WCS	-	Writeable control store
WUER	-	Reserved word not found

BIBLIOGRAPHY

Advanced Micro Devices Inc.,	'Am2901, Am 2909 Technical Data' Advanced Micro Devices Inc., 901 Thompson Place, Sunnyvale, Calif. 94086., 1975.
Agrawala, A.K., and Raucher, T.G.,	'Foundations of Microprogramming: Architecture, Software and Applications'. New York: A.C.M. Monograph Series, Academic Press Inc, 1976.
Aleksander,I.,	'Design of Universal Logic Circuits'. Electronics Letters, Vol. 2, No.8., August, 1966.
Aleksander, I.,	'Microcircuit Learning Nets: Hamming Distance Behaviour'. Electronics Letters 6, p.134, 1970.
Aleksander, I.,	'Introduction to Logic Circuit Theory'. London: George G. Harrap & Co. Ltd., 1970.
Aleksander, I.,	'Some Psychological Properties of Digital Learning Nets'. Int. J. Man-machine Studies. Vol.2, pp.189-212, 1970.
Aleksander, I.,	'Micro-circuit Learning Computers'. London: Mills and Boon, 1971.
Aleksander, I.,	'The Human Machine.' St. Saphorin, Switzerland: Georgi Publishing Co., 1977.
Aleksander, I.,	'Action-Oriented Learning Networks'. Kybernetes, Vol.4, pp.39-44, 1975
Aleksander, I., and Al-Bandar, Z.,	'Adaptively Designed Test Logic for Digital Circuits.' Electronics Letters. Vol.13, p. 466, 1977.
Aleksander, I., Albrow, R.C.,	'Adaptive Logic Circuits'. Computer Journal, Vol.11, No.1, May 1968.

Aleksander, I., Albrow, R.C., & Noble, P.W., 'A Universally Adaptive Monolithic Module'. Electronic Communicator, July-August 1967.

Aleksander, I., and Atlas, P., 'Cyclic Activity in Nature: Causes of Stability'. Int. J. of Ieuroscience, 6, pp.45-55, 1973.

Aleksander, I., & Mandani, E.H., 'Microcircuit Learning Nets: Improved recognition by means of Pattern Feedback'. Electronics Letters, 4.,No.20,p.425, 1968.

Aleksander, I., Stonham, T.J., & Wilson, M.J.D., 'Adaptive Logic for Artificially Intelligent Systems'. IERE Journal Vol.44, pp.39-44, 1974.

Allan, A., 'An Interactive Simulation Program for Networks of Memory Elements' M.Sc. dissertation, University of Kent, Canterbury, 1972.

Arbib, M.A., 'The Metaphorical Brain'. New York: Wiley-interscience, 1972.

Backus, J.W., 'The FORTRAN Automatic Coding System' Proc. Western Joint Comp. Conf. Vol.11 pp. 188-198, 1957.

Ball, A.G., 'An Operating System for a Small Computer" M.Sc. Thesis, University of Kent, Canterbury, 1974.

Bell, C.G., Eggert, J.L., Grayson, J., & Williams,P., 'The description and use of register transfer modules'. IEEE. Trans. Comput., Vol.C-21, pp.495-500, May 1972.

Bledsoe, W.W. & Bisson, C.L., 'Improved Memory Matrices for the n-tuple Pattern Recognition Method'. IRE Trans. on Electronic Computers, Vol. E.C.-11 No.3, pp.414-415, June 1962

Bledsoe, W.W., & Browning, I., 'Pattern Recognition and Reading by Machine'. Proc Eastern Joint Comp. Conf. pp. 225-232, 1959.

Block, H.D. and Levin, S.A.,	'On the Boundedness of an Interactive Procedure for Solving a System of Linear Inequalities' Proc. Amer. Math. Soc. Vol. 26, No.2., pp. 229-235, 1970.
CASH-8,	'CASH-8 Reference Manual', Standard Logic Inc., 3841 South Main, Santa Ana, Calif. 92707, June 1973.
Cheung-Yueng-San, C.Y.E.,	'Some Aspects of Adaptive Logic for Pattern Recognition'. Ph.D. Thesis, University of Kent, Canterbury, 1973.
Crane, H.,	'The Neuristor - A Novel Device and system concept.' Proc. IRE., Vol.50 pp 2048-2060 Oct. 1962.
Crawford, R.H.,	'MOSFET in Circuit Design' New York: McGraw-Hill Book Co., 1967.
Dawson, C.,	'Aspects of Simple Scene Analysis with Learning Networks' Ph.D. Thesis, University of Kent, Canterbury, 1966.
Dreyfus, H.,	'What Computers Cannot Do'. New York: Harper & Row, 1972.
Duda, R.O., and Fossum, H.,	'Pattern classification by interactively determined linear and piecewise linear discriminant functions.' IEEE, Trans. Electronic Computers, Vol. EC15, pp 221-232, April 1966.
Duff, M.J.B., Watson, D.M., Fountain, T.J., & Shaw, G.K.,	'A Cellular Logic Array for Immage Processing'. Pattern Recognition, Vol. 5., pp.229-247, Pergamon 1973.
Enayat, A.,	'A Pattern Recognition Method for the Interpretation of Mass Spectra of Mixtures of Compounds'. M.Sc. Dissertation, University of Kent, Canterbury, 1978.
Fairhurst, M.C.,	'The Dynamics of Learning in Some Digital Networks' Ph.D., Thesis, University of Kent, Canterbury, 1973.

Floyd, R.W.,	'Syntactic Analysis and Operator precedence'.JACM, 10, pp.316-333, July 1963.
Fukushima, Kuwihido,	'Visual Feature Extraction by a multilayered Network of Analog Threshold Elements'. IEEE Trans. on Systems Science & Cybernetics, Vol. SSC-S, No.4, Oct. 1969.
Gavrilov, M.A., & Zakrevskii, A.D.,	'LyaPas: A Programming Language for Logic and Coding Algorithms'. Nadler,M., Trans. New York: Academic Press, 1969.
Gerace, G.B.,	'Microprogramming Control for Computing Systems'. IRE Translations, EC-12 p.733, 1963.
Glennie, A.E.,	'On the Syntax Machine and the Construction of a Universal Compiler'.Carnegie Tech. Computation Centre Report, No.2. July 1960.
Gregory, R.L.,	'Eye and Brain', London: Weidenfeld & Nicholson, 3rd Edn., 1977.
Gries, D.,	'Compiler Construction for Digital Computers'. New York: John Wiley & Sons, Inc., 1971.
Hanna, F.K.,	'APL: An Evaluation with Particular Reference to Research in Automata Theory', Electronics Laboratories, University of Kent, Canterbury, July 1973.
Harmon, Deon. D.,	'Studies with Artificial Neurons'. Kybernetik, Ban 1, Heft 3, Dec. 1961.
Hassitt, A., Lageshulte, J.W., & Lyon, L.E.,	'Implementation of a High Level Language Machine'. C.A.C.M. 16, No.4, pp.199-212, April 1973.
Hebb, D.O.,	'The Organisation of Behaviour'. John Wiley & Sons Inc. New York 1959.
Herscher, M.B., & Kelly, T.P.,	'Functional Electronics Model for the Frog Retina'. IEEE Trans. on Military Electronics, Vol.7., pp.98-103, 1963.

Hopgood, F.R.A., 'Compiling Techniques'. London: Macdonald, 1969.

Hubel, D.H., & Weisel, T.N., 'Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex.' J. Physiol., 160, pp. 106-154, 1962.

Husson, S.S., 'Micro programming Principles and Practices'. Prentice-Hall, New Jersey, 1970.

Intel Corporation, 'Device type 1101 and 1103 Data Sheets'. Santa Clara, Calif.: Intel Corp. 1970.

Iverson. K.E., 'A Programming Language'. John Wiley & Sons, Inc, New York, London 1962.

Kauffman, S.A., 'Metabolic Stability and Epigenesis in Randomly Constructed Genetic Nets'. J. Theoret, Biol,, 22, pp. 437-467, 1969.

Kilmer, W.L., McCullock, W.S., & Blum, J., 'Some Mechanisms for a Theory of the Reticular Formation'. In M. Mesarovic, Ed., "Systems Theory and Biology". New York: Springer-Verlay, pp. 286-375, 1968.

Kornerup, P., 'Concepts of the MATHILDA System'. 2nd Annual Symp. on Comp. Architecture Proceedings, IEEE, pp. 159-164 Jan. 1975.

Kulkarni-Kohli, C., & Newcomb, R.W., 'An Integrable M.O.S. Neuristor Line.' Proc. IEEE, Vol.64 (11), pp.1630-2, Nov.1976.

Kurtz, T.E. & Kemmeny, J.G. 'Basic Programming' New York: Wiley, 1967.

Lawrie, D.H., 'Access and Alignment of Data in an Array Processor'. IEEE, Trans. Compt., Vol. C-24, pp. 1145-1155, Dec. 1975.

Lettvin, J.Y., Maturana, H., McCulloch, W.S., & Pitts, W.H., 'What the Frog's Eye Tells the Frog's Brain'. Proc. IRE, 47, p.1940-1951, 1959.

Lewis, P.M., & Coates, C.L., 'Threshold Logic'. New York: John Wiley & Sons Inc., 1967.

McCullock, W.S., & Pitts, W.,	'A logical calculus of the ideas imminent in nervous activity' Bull. Math. Biophys. Vol.5, pp.115-133, 1943.
McDonald, J.F., Sustman, J.E., & Harris, R.G.	'Fast Register-Transfer-Module Writable Control Store for Microprogrammed Computer Design'. Proc. IEEE Vol.61 No.11 pp.1538-1543 Nov. 1973.
Minsky, M. and Papert, S.,	'Perceptrons: An Introduction to Computational Geometry'. The M.I.T. Press, 1969.
NANODATA Corporation,	'QM-1 Hardware Level User's Manual'. 2nd Ed. NANODATA Corp, 2457 Wehrle Drive, Williamsville, New York 14221, March 31, 1974.
Nappey, J.A.,	'Aspects of N-Tuple Character Recognition for a Blind Reading Aid'. Ph.D. Thesis Brunel University, Uxbridge 1977.
Nilsson, Nils. J.,	'Learning Machines'. McGraw-Hill Book Co., New York, 1965.
Noton, D., & Stark, L.,	'Eye Movements and Visual Perception'. Scientific American, pp.34-43, June 1971.
Novikoff, A.B.J.,	'On Convergence Proofs for Perceptrons' Proc, Symp. on Mathematical Theory of Automata, pp.615-622, 1962.
Raj Reddy, D.,	'Some Numerical Problems in Artificial Intelligence'. in "Complexity of Sequential and Parallel Numerical Algorithms". Traub, J.F., Ed. pp.131-147, London Academic Press, 1973.
Pease, C.,	'The Indirect Binary n-Cube Microprocessor Array', IEEE, Transactions on Computers, Vol. C-26, No.5, p.458-473, 1977.
Reeves, A.P.,	'A Digital Learning System for Tracking Pattern Features'. Ph.D. Thesis University of Kent, Canterbury, 1973.

Reigel, E.W., Faber, U., & Fisher, D.A., 'The Interpreter - a Micro programmable Building Block System'. Spring Joint Computer Conference Proceedings, AFIPS Press, New Jersey, pp 705-723, 1972.

Roberts, L.G., 'Pattern Recognition with an Adaptive Network'. IRE International Convention Record, pp.66-70, 1960.

Rocha, V.C., 'Error Correcting Codes'. Ph.D., Thesis, University of Kent, Canterbury, 1976.

Rochester, M., Holland, J., Haibt, L., & Duda, W., 'Test on a Cell-Assembly Theory of the Action of the Brain, using a Large Digital Computer'. IRE Trans on Info. Theory, Vol.IT6, pp.80-93, Sept. 1956.

Rosenblatt, F., 'Principles of Neurodynamics'. Washington D.C., Spartan Books, 1962.

Rosenblatt, F., 'A Comparison of Several Perceptron Models'. in "Self-Organising Systems" Yovits, Jacobi & Goldstein, Eds. Washington D.C.: Spartan Books, 1962.

Rosenblatt, F., 'Recent Work on Theoretical Models of Biological Memory'. in "Computer and information Sciences II"Ton, Ed. pp.33-56 New York: Academic Press 1967.

Rosin, R.F., 'Contemporary Concepts of Micro-Programming and Emulation'. A.C.M. Computing Surveys, Vol.1, p.197, Dec 1969.

Runge, R.G., Uemura, M., & Viglione, S.S., 'Electronic Synthesis of the Neural Network in the Pigeon Retina'. In 'Cybernetics Problems in Bionomics', Oestricher, H.L., & Moore, D.R. Eds., Bionomics Symposium. pp.791-810, 1966.

Smith, M.H., Sobek, R.P., et al., 'The System Design of JASON, a Computer-Controlled Robot'. Proc. Int. Conf. Cybernetics & Society, IEEE, pp.72-75, September 1975.

Stonham, T.J.,	'Automatic Classification of Mass Spectra'. Pattern Recognition, Vol.7, p.235, 1975.
Stonham, T.J.,	'Improved Hamming-Distance Analysis for Digital Learning Networks'. Electronics Letters, Vol.13, No.6, pp. 155-156 17th March 1977.
Stonham, T.J., & Enayat, A.,	'A Pattern Recognition Method for the Interpretation of Mass Spectra of Mixtures of Compounds'. to be published 1978.
Sutro, L.L.,	'Proposed Electronics to Represent Properties of the Frog's Eye'. In "Cybernetic Problems in Bionomics"., Oestricher, H.L., & Moore, D.R., Eds. Bionomics Symposium, pp. 811-819, 1966.
Texas Instruments,	'The TTL Data Book for Design Engineers'. Texas Instruments CC-411, 1974.
Tollyfield, A.J.,	'Aspects of Training and Connection in Some Cellular Learning Networks'. Ph.D., Thesis, University of Kent, Canterbury 1975.
Tomé, A.B.,	'Some Aspects of Simple Cellular Arrays'. Ph.D. Thesis, Brunel University, Uxbridge 1976.
Ullmann, J.R.,	'Experiments with the n-tuple Method of Pattern Recognition, IEEE. Trans Computers, Dec. 1969.
Unger, S.H.,	'A Computer Oriented Towards Spatial Problems'. Proc. IRE. Vol.46, pp.1744-1750 Oct. 1958.
Wilkes, M., & Stringer, C.,	'Micro-programming and the design of control circuits in an electronic digital computer. Proc. Camb Phil. Soc. 49, pp.230-238. 1953.
Wilner, W.T.,	'Design of the Burroughs B1700'. Fall Joint Comp. Conf. Proc. , AFIPS, pp.489-497 1972.

Wilson, J.D., & Aleksander, I.,	'Adaptive Arrays Parts I & II'. Brunel University, Uxbridge, Technical Memorandum N/R/043 & N/R/044 1976 & 1977.
Winograd, T.,	'Understanding Natural Language by Computer'. Edinburgh University Press, 1972.