ABSTRACT

The existing design languages and methods available to the computer systems designer are critically examined in this report. A new language, which is considered to be flexible, expansible and more akin to the designers' natural methodology, is presented. A command structure and an implementation technique for use with a Honeywell DDP 516 computer with disc backing storage for developing an abstract definition of combinational networks, of upto 20 input and 20 output variables, on an interactive basis, is also presented.

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRONICS

Master of Philosophy

A LANGUAGE FOR COMPUTER AIDED LOGIC SYSTEM DESIGN

by

Dinesh Pai, B.Sc.(Eng.)

<u>CONTENTS</u>

ACKNOWLEDGEMENT

I am indebted to my project supervisor, Professor Douglas Lewin, Brunel University, formerly of Southampton University, for his guidance, helpful discussions and his continuous encouragement. I also wish to acknowledge the valuable assistance given by Dr. R.G. Bennetts with this project.

I wish to thank International Computers Limited and the Science Research Council for the financial support.

My gratitude extends to Catherine Quither for the timely assistance in converting my illegible writing into a typed manuscript and to Sharon Conlin for her patience and encouragement throughout the preparation of this report.

1. Introduction to Digital System Design Specifications

1.1 Introduction

A digital system is conventionally divided into hardware, i.e., the part which implements the basic characteristics of the system using electronic or mechanical building blocks and which is relatively difficult to modify, and software which forms a superstructure on the hardware and assigns a set of different characteristics to the digital system for the final applications. The design processes of the two parts are physically different in that they rely on different building blocks and consequently employ different criteria. For example, hardware design is influenced by the types of electronic switching elements available, fan-in and fan-out factors of gates, i.e., the number of inputs and the number of outputs that may be connected to a gate, packaging of the switching elements, interconnection methods and problems of fabrication of sub-units and units; whereas the software design is based on the repertoire of instructions executable by the hardware, memory accessing and information management techniques and the input-output device handling techniques employed by the hardware, and the communication between various sections of software.

Conceptually however, the design phase for both hardware and software is identical and can be characterized by the following steps:

a. Define the system in a natural language describing its overall characteristics, such as input-output behaviour, performance, etc.

b. Convert the description in a. to formal specifications.

c. Implement the formal specifications in terms of appropriate building blocks with due regard to physical constraints such as speed, cost, reliability, testability and to a lesser extent future modifiability.

Despite the identical nature of the design procedures for hardware and software. It is a current practice to treat the two aspects completely differently, especially in steps b. and c. Often step b. is completely bypassed in the design process. This is due to several reasons, the main ones being the designers' reluctance to conform to any formalization of the design process since it could be regarded as reducing the scope for exercising their skill and ingenuity, and the distinct lack of standard formal techniques which could cover a wide range of problems. Obviously then, the design process relies heavily on the designers' past experience and ingenuity, is extremely time consuming and prone to errors. Such a process is also subject to inaccurate and inadequate documentation. In many cases the documentation is based on the final design with no trace of the intermediate steps taken by the designer.

This means that there is a proliferation of many different techniques currently employed in the design of a digital system. While this practice does not cause many problems when small systems are being implemented it presents more and more acute problems when digital systems of large sizes, such as modern computers are designed. Here, of necessity, the design process must be divided up and the need for suitable formalization of overall design techniques and good documentation, for intercommunication between numerous designers and for the subsequent manufacturing, becomes more urgent.

1.2 Design Automation

In the last application mentioned above, the data required particularly for large system implementation is very large indeed and in most cases certainly too large for manual handling. Fortunately, however, large and powerful computers have recently become available and these can be efficiently employed to handle the mechanical tasks in system implementation. In fact, most manufacturing concerns already use digital computers to perform component layout, back panel wiring, and cable connections - also providing a check on circuit completeness. Additionally, useful tasks of documentation of parts-lists and drafting are also relegated to the computer. Both these factors assist and improve the production process.

It is, therefore, natural to extend the scope of design automation and consider the possibility of employing the digital computer in the design process.

Apart from the obvious advantages of documentation facilities and automatic logic generation, as required in step c, the digital computer can also provide to the designer some powerful facilities, which, in most cases, otherwise would be beyond the time and effort available. These are:

1. minimization, i.e., removing redundancies;
2. simulation to check the design completeness and to obtain performance figures;
3. generation of test sets which would allow the detection and location of faults if and when they arose.

The results obtained by invoking the above facilities can provide very valuable feedback to the designer allowing him to modify the design specification or the design itself as necessary, and to reinitiate the design cycle until an optimal, i.e., economically satisfactory, solution is reached - a process which normally should be executed before any expensive manufacture is initiated. If the designer had facilities to communicate directly with the computer, e.g., via a teletype unit or a visual display unit, the feedback cycle could be made much shorter. The designer then would be in a position to experiment with various designs, increasing considerably his scope for ingenuity and exercising his skill.

Unfortunately, however, in view of the current state of the art, the above procedure has major drawbacks.

The data generated during the design phase is large and the resulting computation is very complex even when performed on large computers. Switching theory, [1], [2], [3] the only tool available for rigorous design, is still mainly applicable to small systems and its application to large systems' design, both hardware and software, is still at an infancy stage. Nevertheless, the potential advantages of the above approach are unquestionable.

The designer then,must be provided with a communication interface with the computer, i.e., a language. This language must be such that it is of a high enough level so that too much time is not spent specifying routine duties, yet at the same time must be of a low enough level to be flexible. It must also be relatively easy to learn to be of practical value, e.g., in documentation, teaching its use to new designers etc. And of course, the language must be translatable into a format so that tools provided by switching or similar theory may be applied.

In the following chapters we examine the various languages proposed so far and discuss their relative merits and disadvantages. A comprehensive set of examples is also provided in the appendix to complement the discussion.

2. REGULAR EXPRESSIONS

2.1 Introduction

Regular expressions [4 - 30] describe the input-output behaviour of a clocked or pulse mode system in a way which is independent of its internal structure. As such, regular expressions provide a method of representing a system as an abstract automaton and of deriving a mathematical model for it. Also, since all clocked or pulse mode systems are covered, regular expressions can handle a large class of sequential systems. The language of regular expressions is precise and since the description is in a single-line type of format it is much easier to process than, say, state tables or state diagrams. Furthermore, because of their characteristics, regular expressions sometimes closely resemble natural language description. It appears, therefore, that the language of regular expressions is a very useful tool for analysis. However, the regular expression describing a system can vary considerably depending on the way it is derived and to the author's knowledge, no satisfactory methods yet exist to discover the identities of equivalent regular expressions.

The limitation of the language of regular expressions is that it can only apply to a finite state system. A computer is essentially a finite state machine with a separate large memory and, therefore, regular expressions cannot be used for synthesis of computers. Secondly, the regular expression representation is such that when the expression becomes valid, i.e. when the system "accepts" the regular expression the ouput is made equal to 1; otherwise the output remains at 0. Therefore, for multiple outputs the only way to use this language is to consider each output separately and derive the relevant regular expression for each. Thus this method is mostly suited for single output systems.

These disadvantages restrict the use of regular expressions and designing digital computers using regular expressions only would be an extremely long and laborious, if not an impossible process. The mathmematical nature of regular expressions, however, has roused considerable interest and a wealth of papers have appeared since Kleene[18] first introduced their use in connection with automata. The following discussion, therefore, is included as an illustration of the language of regular expressions and a rigorous and complete coverage is not included.

2.2 Historical Survey

The theory of regular expressions dates back to 1943 when McCulloch and Pitts [21] developed a logical theory to describe the behaviour of nerve nets. In 1956 Kleene [18] extended the ideas to describe abstract automata by regular expression and also showed that every finite state deterministic automaton can be defined by a regular expression and that every regular expression can be realised by a finite state, deterministic automaton. The theory he developed, however, was mainly in terms of nerve nets and was rather complicated. Later Copi, Elgot and Wright [12], in their expository paper, simplified the discussion but restricted themselves to instantaneous logic. In 1960 McNaughton and Yamada [22] added to the theory by providing algorithms for deriving regular expressions from state diagrams and vice versa. Some other treatments of regular expressions were also developed by Lee [20], Arden [4], Mayhill [25] and Rabin and Scott [28]; but their terminology and presentation varied widely. In 1962 Brzozowski [5] published an expository paper giving a unified account of all the theory published until then; and around the same time Ghiron [15] independently published a correspondence enumerating rules to manipulate regular expressions. Since then Brzozowski has published a number of papers on this subject. He and McClusky [6] furthered the ideas of Arden [4] and applied signal flow graph techniques to regular expressions. He also overcame one of the major disadvantages of the technique by McNaughton & Yamada which requires very lengthy manipulation, by developing the concept of derivatives of regular expressions [7] and the techniques to obtain state diagrams from regular expressions using derivatives. Spivak [29] also independently developed these techniques of derivatives, but he referred to a derivative as "the quotient of division". Udagawa et al [30], in 1965, unified the derivative approach and Arden's linear equation method into a matrix form.

The more recent work in this field has been mainly on the algebra rather than applications of regular expressions [9,11,27].

2.3 Definitions and Properties

Consider a set of n inputs to a machine M as shown in Figure 1, such that each input can take up a value of logical 0 or 1. These binary variables are called input signals. A particular ordered arrangement of the input signals is called an input configuration. Assuming the input configuration represents a binary string with a_{n-1} as the most significant bit and a_0 as the least significant bit, the value of the string is called an input symbol, and the set of input symbols is called an input alphabet. It follows that the input symbols can take values between 0 and 2^n-1 and the input alphabet contains 2^n symbols. Only synchronous machines are considered and the values assumed by the input symbols at successive clocking times denote an input sequence.

For the present discussion we restrict ourselves to a limited set of regular operators containing +,.,*,(,) namely the disjunction, concatenation and star operators and parenthesis. The regular expressions are recursively defined as follows:

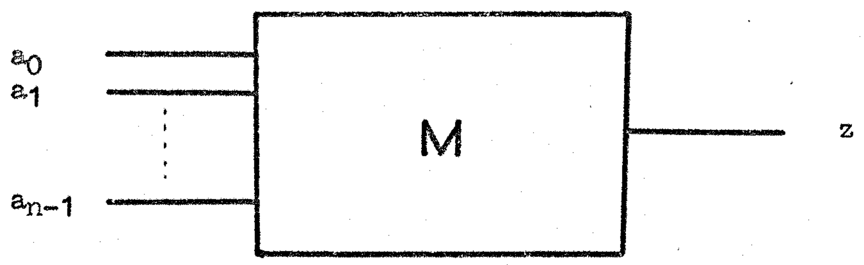


FIGURE I. MACHINE M

1) Any symbol of the input alphabet, a $\emptyset$ or a λ is a regular expression.

2) If $\underline{A}$ and $\underline{B}$ are regular expressions then $\underline{A}+\underline{B}$, $\underline{A}.\underline{B}$ (sometimes written $\underline{AB}$) and A^* are also regular expressions.

3) Only expressions derived by application of rules 1 and 2 a finite number of times are regular expressions.

Parenthesis are used to group sequences of regular expressions. The symbol λ is an input sequence of zero length and $\emptyset$ is the null or empty set of sequences, the difference being that λ is a set with one symbol and $\emptyset$ is a set with no symbols. The star operator is defined as follows:

$$\begin{aligned} A^* &= \lambda + \underline{A} + \underline{AA} + \underline{AAA} + \underline{AAAA} + \dots \\ &= \lambda + \underline{A} + \underline{A}^2 + \underline{A}^3 + \underline{A}^4 + \dots \end{aligned}$$

An automaton realises a regular expression or it is said to accept a regular expression if when a valid sequence contained in that regular expression is applied to the machine an output of 1 is produced, and such a regular expression defines the machine. Before attempting to derive any regular expressions for a given machine and vice versa, it will be useful to consider some of the basic properties which are enumerated below.

If $\underline{A}$, $\underline{B}$ and $\underline{C}$ are regular expressions, then

i)	$\underline{A} + \underline{B}$	$=$	$\underline{B} + \underline{A}$	Commutative
ii)	$(\underline{A} + \underline{B}) + \underline{C}$	$=$	$\underline{A} + (\underline{B} + \underline{C})$	Associative
iii)	$(\underline{AB})\underline{C}$	$=$	$\underline{A}(\underline{BC})$	Associative
iv)	$\underline{AB} + \underline{AC}$	$=$	$\underline{A}(\underline{B} + \underline{C})$	Distributive
v)	$\underline{AC} + \underline{BC}$	$=$	$(\underline{A} + \underline{B})\underline{C}$	Distributive
vi)	$\underline{A} + \emptyset$	$=$	$\emptyset + \underline{A} = \underline{A}$	
vii)	$\underline{A}\emptyset$	$=$	$\emptyset\underline{A} = \emptyset$	Properties of $\emptyset$
viii)	$\emptyset^*$	$=$	λ	

ix) $\underline{A}\lambda = \lambda\underline{A} = \underline{A}$ Properties of λ

x) $\lambda^* = \lambda$

xi) $\underline{A} + \underline{A} = \underline{A}$

xii) $(\underline{A} + \underline{B})^* = (\underline{A}^*.\underline{B}^*)^*$

In some cases the knowledge of sequences from time zero is not required or available. In such cases a don't care symbol is useful. It is called $\underline{i}$ meaning any symbol of the alphabet and $\underline{i}^*$ is a don't care sequence.

2.4 Regular Expression from Natural Language Description

As we stated in the preceding section, a regular expression is essentially a sequence of inputs accepted by an automaton. Thus the language of regular expressions can be used for describing sequence recognisers and it is this kind of description that the language suits most. If the set of input strings accepted by the automaton is known or alternatively if an automaton has to be designed with a known set of input strings, then it is a simple matter to convert this description into a regular expression. The task of discovering the set of all-input strings accepted by an automaton, however, is a very complex one and in practice, except in a few cases, is impossible.

Suppose that it is necessary to generate an output if the input string contains the sequence 1011 then the regular expression describing this automaton would be simply

$$\underline{R} = i^*1011(i^*1011)^*, \qquad i = 1 + 0.$$

A better example would be one containing the Boolean operators & (AND) and ' (negation). For example, an automaton accepting an input sequence containing groups of 11 followed by groups of 00 but not ending in 01 or accepting an input sequence containing groups of 101 would have the regular expression

$$R = (i^*11(11)^*00(00)^*) \,\&\, (i^*01)' + i^*101(101)^*$$

Another, useful, example is a divide-by-two automaton which accepts all sequences containing an even number of 1's. This automaton is defined completely and precisely by the regular expression

$$\underline{R} = 0^*10^*1(0^*10^*1)^*$$

2.5 Regular Expressions for Combinational Logic

As was stated before, a regular expression decribes a sequence of input symbols at successive clock times necessary to produce an output of 1. It follows, therefore, that the regular expression for a unit delay is

$$\underline{R} = \underline{i}^*1\underline{i} \qquad (1)$$

This expression is valid for a machine containing instantaneous logic. If, however, a unit delay is inherent in the logic then the required expression is

$$\underline{R} = \underline{i}^*1 \qquad (2)$$

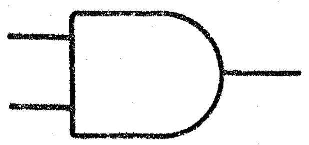
Regular expressions for combinational logic devices can be similarly derived and some examples are given in figure 2.

2.6 Regular Expressions from State Diagrams

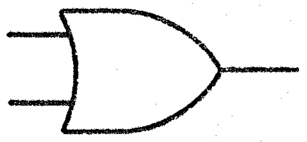
The technique illustrated below is due to Arden [4].

Each state has a regular expression associated with it which describes all the sequences necessary to bring the machine into that state from a starting state. This regular expression is obviously equal to all the regular expressions associated with the adjacent states, i.e. the states from which the state under consideration can be reached by inputting a single symbol, followed by the symbols which will cause the transitions. This equation can be written as:

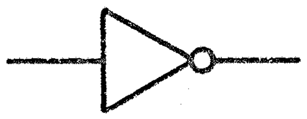
$$\underline{D}_{s1} = \underline{D}_{s1}a_{11} + \underline{D}_{s2}a_{21} + \dots + \underline{D}_{sn}a_{n1} \qquad (3)$$



AN AND GATE $R = i^* 3$



AN OR GATE $R = i^* (1+2+3)$



AN INVERTOR $R = i^* 0$

FIGURE 2

where $\underline{D}_{s1}$ is the regular expression describing all the sequences taking the automaton from the starting state q_s to the state q_1 and a_{21} is the input symbol causing a direct transition from state q_2 to state q_1, etc.

Regular expressions associated with other states can be written down similarly:

$$\begin{aligned}
\underline{D}_{s2} &= D_{s1}a_{12} + \underline{D}_{s2}a_{22} + \dots + \underline{D}_{sn}a_{n2}. \\
&= \\
&= \\
\underline{D}_{ss} &= D_{s1}a_{1s} + \underline{D}_{s2}a_{2s} + \dots + \underline{D}_{sn}a_{ns} + \lambda \\
&= \\
&= \\
\underline{D}_{sn} &= \underline{D}_{s1}a_{1n} + \underline{D}_{s2}a_{2n} + \dots + \underline{D}_{sn}a_{nn}. \qquad (4)
\end{aligned}$$

where λ is the starting symbol.

These can then be solved as simultaneous equations.

As an example consider the state diagram shown in figure 3, of a machine with only one input. If $\underline{A},\underline{B},\underline{C}$, and $\underline{D}$ represent the regular expressions associated with the states A,B,C and D respectively, then the relevant equations are

$$\underline{A} = D1 + \lambda \qquad (5)$$

$$\underline{B} = \underline{A}0. \qquad (6)$$

$$\underline{C} = \underline{B}1 + \underline{A}1 + \underline{C}0 \qquad (7)$$

$$\underline{D} = \underline{C}1 + \underline{B}0 + \underline{D}0 \qquad (8)$$

Then substituting for $\underline{B}$ in (7)

$$\underline{C} = \underline{A}(01+1) + \underline{C}0 \qquad (9)$$

This is an equation of the type

$$\underline{X} = \underline{XA} + \underline{B}$$

which suggests that a sequence $\underline{B}$ is required to arrive at state X and any further occurrence of sequence $\underline{A}$ will cause transition back

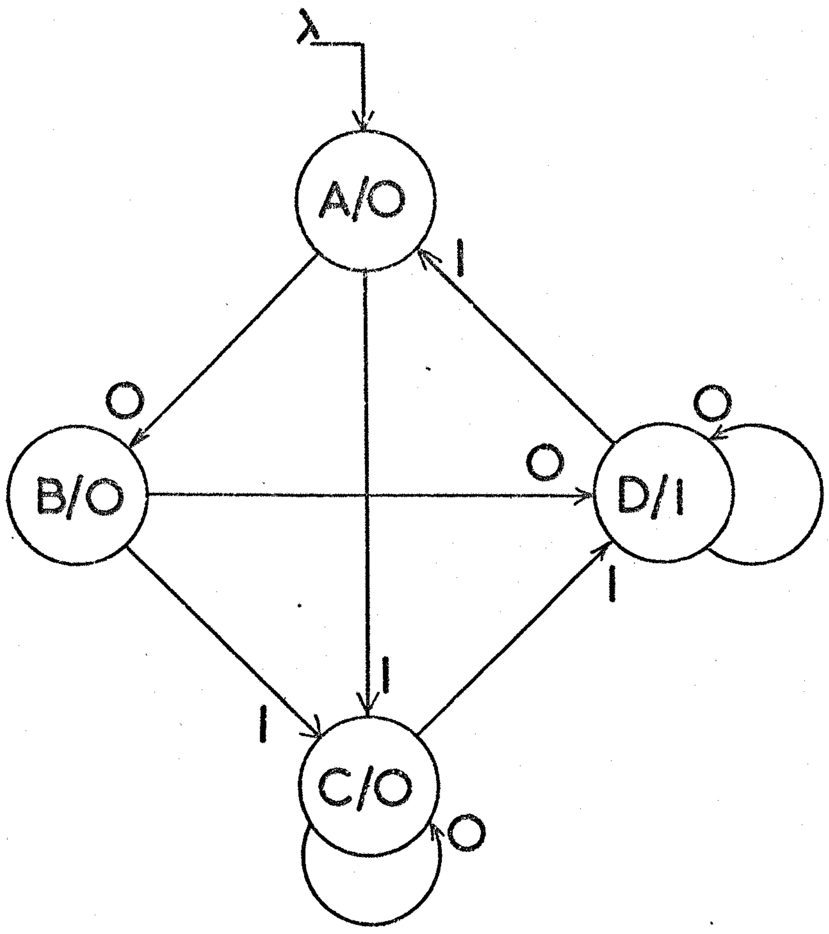


FIGURE 3

to X, i.e. the solution to the equation is

$$X = \underline{B}\underline{A}^*$$

In fact it can be shown that this is the only solution to this type of equation providing $\underline{A}$ does not contain λ[†]

Thus the solution to (9) is

$$\underline{C} = \underline{A}(01+1)0^* \tag{10}$$

From (10) and (6)

$$\begin{aligned}\underline{D} &= \underline{A}((01+1)0^*1 + 00) + \underline{D}0 \\ &= \underline{A}((01+1)0^*1 + 00)0^* \end{aligned}\tag{11}$$

and from (5) and (11) we get

$$\begin{aligned}\underline{A} &= \underline{A}((01+1)0^*1 + 00)0^*1 + \lambda . \\ &= \lambda(((01+1)0^*1 + 00)0^*1)^*. \\ &= (((01+1)0^*1 + 00)0^*1)^*. \end{aligned}\tag{12}$$

Hence

$$\underline{B} = (((01+1)0^*1 + 00)0^*1)^*0. \tag{13}$$

$$\underline{C} = (((01+1)0^*1 + 00)0^*1)^*(01+1)0^*. \tag{14}$$

$$\underline{D} = (((01+1)0^*1 + 00)0^*1)^*((01+1)0^*1+00)0^* \tag{15}$$

If the machine produces an output in state D then the regular expression defining the machine is $\underline{D}$.

2.7 State Diagrams from Regular Expressions

The method described below is due to McNaughton and Yamada [22]. It is illustrated with a running example which uses the regular expression obtained in the last section.

Step 1. Associate a position 1 with the leftmost symbol in the regular expression. Associate a position 2 with the next occurrence of the same symbol to the right and so on until the last occurrence is suitably identified. Repeat this procedure for all

† see Appendix III.

the other symbols in the alphabet. These identifications appear as subscripts to the symbols in the regular expression.

Applying this step to the expression D we get

$$\underline{D} = (((0_1 1_1 + 1_2) 0_2^* 1_3 + 0_3 0_4) 0_5^* 1_4)^* ((0_6 1_5 + 1_6) 0_7^* 1_7 + 0_8 0_9) 0_{10}^* . \quad (16)$$

A position is termed <u>initial</u> if a valid sequence is contained in the regular expression which begins with that position and similarly a position is <u>terminal</u> if a valid sequence can terminate in that position. In the above regular expression these positions are

Initial	0_1, 1_2, 0_3, 0_6, 1_6, 0_8.
Terminal	1_7, 0_9, 0_{10}.

Step 2. In this step, we determine all the <u>allowable transitions</u>. These are ordered pairs of positions which a valid sequence can follow. The meaning should be clear from the ordered pairs in the example which are

$(0_1,1_1)$;

$(0_2,0_2)$, $(0_2,1_3)$;

$(0_3,0_4)$;

$(0_4,0_5)$, $(0_4,1_4)$;

$(0_5,0_5)$, $(0_5,1_4)$;

$(0_6,1_5)$;

$(0_7,0_7)$, $(0_7,1_7)$;

$(0_8,0_9)$;

$(0_9,0_{10})$;

$(0_{10},0_{10})$;

$(1_1,0_2)$, $(1_1,1_3)$;

$(1_2,0_2)$, $(1_2,1_3)$;

$(1_3,0_5)$, $(1_3,1_4)$;

$(1_4,0_1)$, $(1_4,1_2)$, $(1_4,0_3)$, $(1_4,0_6)$, $(1_4,1_6)$, $(1_4,0_8)$;

$(1_5,0_7)$, $(1_5,1_7)$;

$(1_6,0_7)$, $(1_6,1_7)$;

$(1_7,0_{10})$;

Step 3. The state diagram is then built up using the following procedure. Assume a present state q_i corresponding to position set

$$\left\{ P_i \right\} = \left\{ P_{ik} \middle| k \text{ is an integer} \right\}$$

of the symbol i. Suppose a symbol j is received then the next state q_j is the largest set $\left\{ P_j \right\}$ such that there is at least one allowable transition to each position of the set $\left\{ P_j \right\}$ from the set $\left\{ P_i \right\}$. If there is no such set, i.e. it is an unallowable transition, then the next state is a fault state and all the transitions from this state terminate in this state. This process is continued until all positions are covered. An initial starting state S is also assumed.

Applying this procedure to the example we obtain the state diagram shown in figure 4. This appears quite different from the state diagram in figure 3, for which the regular expression $\underline{D}$ was derived; however, using usual minimisation techniques the diagram in figure 4 reduces to the same as in figure 3.

2.8 Derivatives of Regular Expressions

A far more elegant method to obtain the minimal state diagram is the use of derivaties of regular expressions, a method developed by Brzozowski [7] and independently by Spivak [29]. The derivatives simply give an indication whether a particular sequence is contained in the regular expression or not. They also handle multiple occurrences simultaneously; hence repeats, corresponding to loops in the state diagram are recognised and identical loops merged. The state diagrams thus obtained, therefore, are minimal.

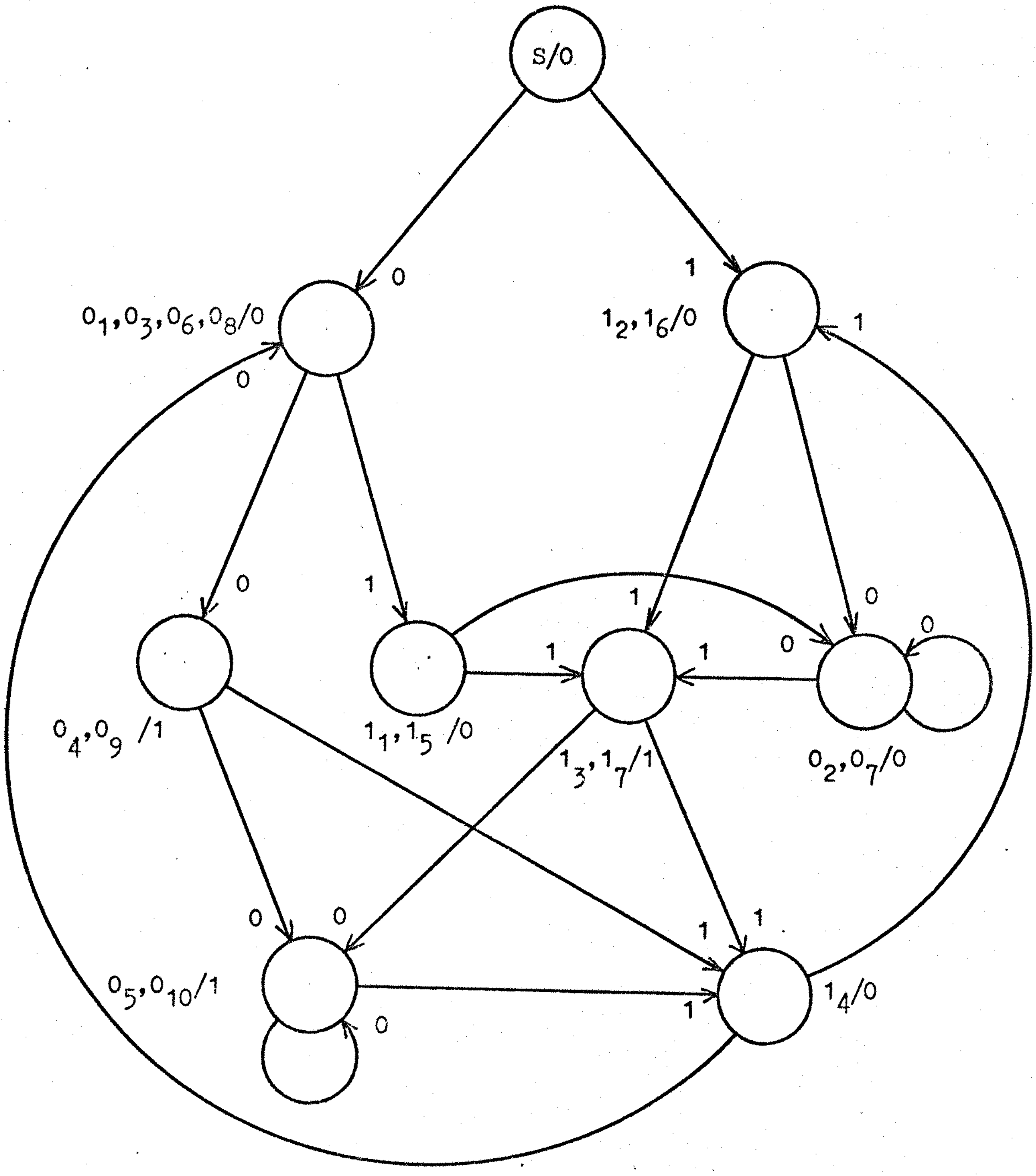

S/0
0
1
0₁,0₃,0₆,0₈/0
1₂,1₆/0
0
1
0
1
1
1
1
0
0
0
0₄,0₉ /1
1₁,1₅ /0
1₃,1₇/1
0₂,0₇/0
0
0
1
1
0₅,0₁₀/1
1
1₄/0
0

FIGURE 4

There are two kinds of derivatives: a) the left derivative denoted by $D_s^L[\underline{R}]$ where $\underline{R}$ is the regular expression whose derivative is taken with respect to the sequence s, and b) the right derivative which is denoted by $D_s^R[\underline{R}]$. They both can be used identically to develop state diagrams. For the discussion below, we restrict ourselves to the left derivative and omit the superscript.

The derivative of a regular expression $\underline{R}$ with respect to a sequence s is defined as

$$D_s[\underline{R}] = \{t \mid st \in \underline{R}\}$$

Before going into the details of this method, a function δ has to be defined and rules of derivatives given. The δ function is defined by

$$\begin{aligned} \delta[\underline{R}] &= \lambda && \text{if } \lambda \in \underline{R} \\ &= \emptyset && \text{if } \lambda \notin \underline{R} \end{aligned}$$

and the rules of derivatives, given without proof, are listed below.

$$\begin{aligned} D_{a_1}[a_2] &= \lambda && \text{if } a_1 = a_2 \\ &= \emptyset && \text{otherwise} \end{aligned} \qquad (19)$$

where a_1 and a_2 are symbols of the input alphabet.

If a is a symbol of the input alphabet, f is any function of the two regular expressions $\underline{R}$ and $\underline{Q}$, then

$$Da[\underline{RQ}] = (Da[\underline{R}])\underline{Q} + \delta[\underline{R}]Da[\underline{Q}] \qquad (20)$$

$$Da[\underline{R}^*] = Da[\underline{R}]\underline{R}^* \qquad (21)$$

$$Da[\underline{R}'] = Da[\underline{R}]' \qquad (22)$$

$$Da[f(\underline{R},\underline{Q})] = f(Da[\underline{R}], Da[\underline{Q}]) \qquad (23)$$

and finally $D_\lambda[\underline{R}] = \underline{R}$

where ' is used to indicate negation.

From the above rules it follows that

$$D_{a_1a_2}[\underline{R}] = D_{a_2}[D_{a_1}[\underline{R}]] \qquad (24)$$

and

$$D_{a_1 \cdots a_n}[\underline{R}] = D_{a_n}[D_{a_1 \cdots a_{n-1}}[\underline{R}]] \qquad (25)$$

Also from the definition of derivatives it follows that a regular expression can be written in the form

$$\underline{R} = \delta\,\underline{R} + \sum_{a\epsilon A} aD_a\,\underline{R} \qquad (26)$$

where $\delta\,\underline{R}$ is introduced if $\underline{R}$ contains λ.

2.9 State Diagrams from Regular Expressions using Derivatives [7], [29].

In section 2.7, an elementary state diagram was obtained from a regular expression and then switching theory was used to reduce it to a minimal form. The algebra of regular expressions can also be used to obtain a minimal state diagram directly. To do this, first the rules and properties of identical, or more correctly indistinguishable, states must be noted. Indistinguishability is defined as follows: two states of an automaton are said to be indistinguishable if the behaviour of the automaton is identical in each of the two states.

Assume an automaton M, defined by a regular expression $\underline{R}$. It follows from the definition of regular expressions that if the automaton is in the starting state q_λ then a valid sequence s, contained in $\underline{R}$ will be accepted by M. Similarly a state q_i is said to accept a sequence s if M is in state q_i and if the sequence s is applied to M, an output of 1 is produced at the end of s. Quite clearly then, two states q_i and and q_j are indistinguishable if all the sequences accepted by one are also accepted by the other and vice versa.

Now, if a sequence s_i takes the automaton from the starting state q_λ to a state q_i, it follows from the definition of derivatives that the derivative of the regular expression $\underline{R}$ with respect to s_i is a regular expression which contains all the sequences accepted by q_i. Therefore the definition of indistinguishability can be modified to read "that two states q_i and q_j are indistinguishable if the derivatives with respect to s_i and s_j are equivalent where s_j is the sequence taking M from starting state to the state q_j, and s_i is also similarly defined."

This provides the criterion for minimality.

The state diagram then is obtained by the following procedure, which applies to an automaton M defined by the regular expression $\underline{R}$ and whose input alphabet is A_k containing the input symbols $a_1, a_2 \ldots\ldots a_k$.

Step 1. Begin by taking $D_\lambda \underline{R}$ which will be $\underline{R}$.

Step 2. Determine all $D_{a_i} \underline{R}$ and associate a new state with each distinct $D_{a_i} \underline{R}$. This will give all the derivatives to sequences of length 1.

Step 3. Continue step 2 for sequences of length 2 and beginning with each a_i for which $D_{a_i} \underline{R}$ were different.

Step 4. Repeat step 3 for higher length sequences until no further distinct derivatives are obtained.

Step 5. Determine the outputs associated with each of the states generated by the above steps. The output is 1 if the δ function of the corresponding derivative is equal to λ. This follows directly from the rules of derivatives and the definition of δ function.

The above function is illustrated by the same example in the previous sections where the input symbols are {0,1} and the output is z.

$$\underline{D} = (((01+1)0^*1+00)0^*1)^*((01+1)0^*1+00)0^* \qquad (15)$$

$$D_\lambda \underline{D} = \underline{D} \qquad \delta(\underline{D}) = \emptyset, \quad z = 0. \qquad (27)$$

$$D_0 \underline{D} = (10^*1+0)0^*1\underline{D} + (10^*1+0)0^* \qquad \delta(D_0 \underline{D}) = \emptyset, \ z = 0 \qquad (28)$$

$$D_1 \underline{D} = 0^*10^*1\underline{D} + 0^*10^* \qquad \delta(D_1 \underline{D}) = \emptyset, \ z = 0. \qquad (29)$$

$$D_{00} \underline{D} = 0+1\underline{D} + 0^* \qquad \delta(D_{00} \underline{D}) = \lambda \quad z = 1. \qquad (30)$$

$$D_{01} \underline{D} = 0^*10^*1\underline{D} + 0^*10^* = D_1 \underline{D} \qquad (31)$$

$$D_{10} \underline{D} = 0^*10^*1\underline{D} + 0^*10^* = D_1 \underline{D} \qquad (32)$$

$$D_{11} \underline{D} = 0^*1\underline{D} + 0^* = D_{00} \underline{D} \qquad (33)$$

$$D_{000} \underline{D} = 0^*1\underline{D} + 0^* = D_{00} \underline{D} \qquad (34)$$

$$D_{001} \underline{D} = \underline{D} = D_\lambda \underline{D} \qquad (35)$$

Thus there are only four distinct states corresponding to $D_\lambda \underline{D}$, $D_0 \underline{D}$, $D_1 \underline{D}$, and $D_{00} \underline{D}$ and the state diagram is as in figure 5 which is the same as in figure 3 with states A,B,C,D replaced by states q_λ, q_0, q_1 and q_{00} respectively.

In the examples so far the outputs are associated with states only, i.e. only Moore machines are considered. Another type of machine, called a Mealy type, has its outputs associated with transitions, i.e. they depend on the present state and the input. The above procedure is easily amended to produce Mealy type machines.

In the Moore type of machines a distinction is made between two derivatives differing only by λ as one of these has an output associated with it and the other one does not. In deriving Mealy machines this distinction is omitted and the outputs are associated with transitions. The Mealey machine diagram corresponding to the example is shown in figure 6.

2.10 The State Characteristic Equation

From its definition, a derivative of a regular expression with respect to a sequence s is a regular expression accepted by the state q_s, where the sequence s takes the automaton from the starting state q_λ to q_s. Thus, it follows that a technique similar to Arden's can be applied with derivatives to state diagrams to obtain regular expressions. Udagawa et al [30] unified Brzozowski's derivative method and Arden's simultaneous equations method into a matrix form to do this giving the state characteristic equation.

Consider a set of states $\{q_1, q_2, \ldots, q_n\}$. We define a matrix D

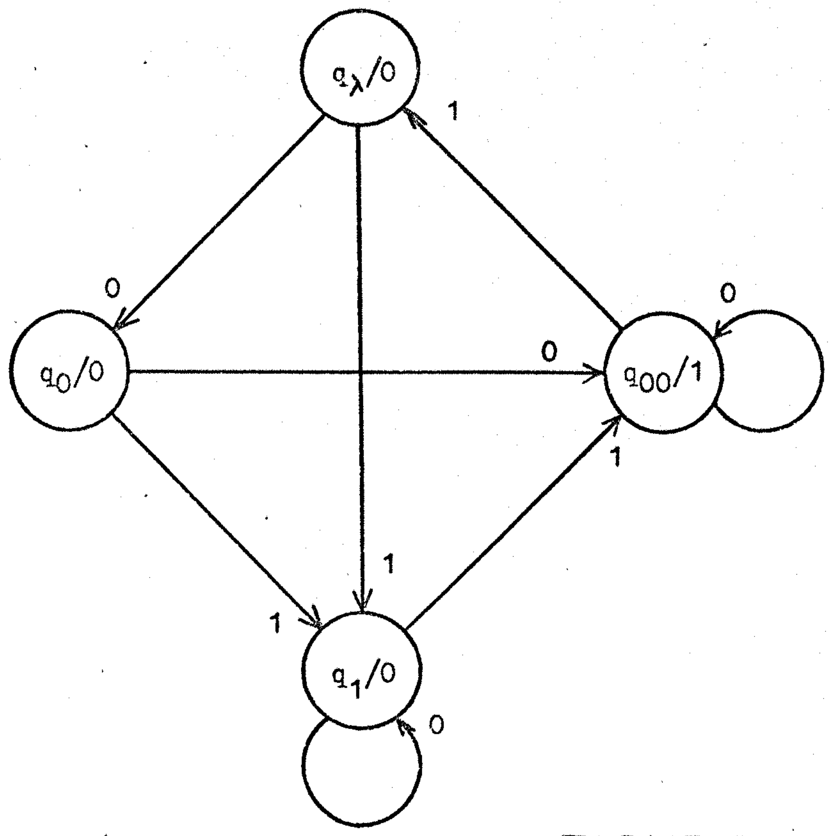


FIGURE 5

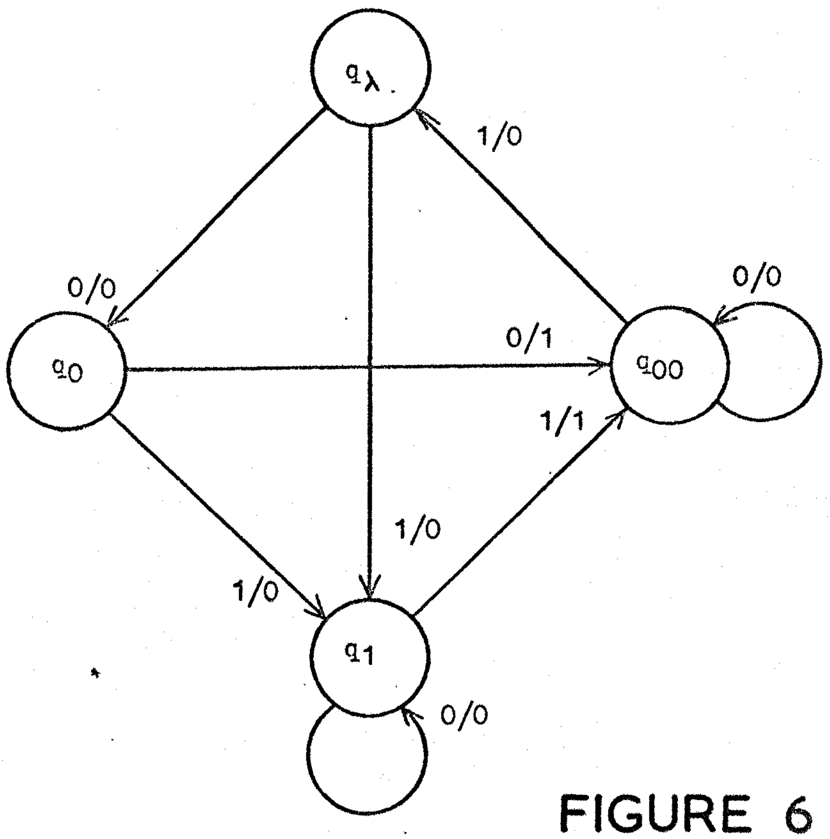


FIGURE 6

$$D = \begin{bmatrix} D_1 \\ D_2 \\ \cdot \\ \cdot \\ D_n \end{bmatrix} = \begin{bmatrix} d_{11} & d_{12} & d_{13} & \cdots & d_{1n} \\ d_{21} & d_{22} & d_{23} & \cdots & d_{2n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ d_{n1} & d_{n2} & d_{n3} & \cdots & d_{nn} \end{bmatrix} \qquad (36)$$

such that d_{ij} is a regular expression which describes the class of sequences causing a transition from the state q_i to q_j. We also define a matrix A

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \cdot \\ \cdot \\ A_n \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} \qquad (37)$$

where a_{ij} is an input symbol causing a transition from the state q_i to q_j. And finally we define an n by n matrix E whose diagonal elements are λ and all the other elements are $\emptyset$.

Now if the starting state of the automaton is q_1, using Arden's method we get

$$\begin{aligned} d_{11} &= d_{11}a_{11} + d_{12}a_{21} + \cdots + d_{1n}a_{n1} + \lambda . \\ d_{12} &= d_{11}a_{12} + d_{12}a_{22} + \cdots + d_{1n}a_{n2}. \\ \cdot &= \cdot \qquad \cdot \qquad \cdot \qquad \cdot \\ \cdot &= \cdot \qquad \cdot \qquad \cdot \qquad \cdot \\ d_{1n} &= d_{11}a_{1n} + d_{12}a_{2n} + \cdots + d_{1n}a_{nn}. \end{aligned} \qquad (38)$$

Then if we write $d_{11}, d_{12}, \ldots, d_{1n}$ as D_1, we get

$D_1 = D_1A + E_1$.

Similarly $D_2 = D_2A + E_2$ etc.

Hence $D = DA + E$. (39)

By similar procedure the derivative form can be written as

$D = AD + E$. (40)

Equations (39) and (40) are called the characteristic equations.

The matrix A is simply another way of stating the state table and the matrix E expresses the output states. Given that an equation of the form $X = AX + B$ has a solution $X = A^*B$† and the equation of the form $X = XA + B$ has solution $X = BA^*$†, usual matrix techniques can be extended to solve the equations (39) and (40).

2.11 Minimal State Diagrams for Multiple Outputs

As was stated before, one regular expression has to be associated with each output; and therefore one way to obtain state diagrams for multiple output automata is to derive a separate state diagram for each output. However, this does not necessarily produce an overall minimal machine. Brzozowski [7] gave the following method which is an extension to the derivative method.

The set of n regular expressions associated with the n outputs is written as a vector

$$\underline{R} = \{\underline{R}_1, \underline{R}_2, \ldots, \underline{R}_n\}. \qquad (41)$$

Then using methods described above a vector of derivatives and another of outputs are generated, i.e.

$$D_{a_i}\underline{R} = D_{a_i}\underline{R}_1, D_{a_i}\underline{R}_2, \ldots, D_{a_i}\underline{R}_n \qquad (42)$$

$$\delta(D_{a_i}\underline{R}) = \{z_1, z_2, \ldots, z_n\}. \qquad (43)$$

As before, the state diagram is built up by associating a new state with each new vector. The output vector is also taken into account if a Moore machine is required.

2.12 Transition Graphs

A state diagram describes a deterministic type of system. By this we mean that if an input is applied to the system in a state, then the next state can be uniquely determined; and also that the system at any given time can exist in only one state. These restrictions are necessary to make a physical realisation of the system possible.

In the preceding sections we developed state diagrams from regular expressions adhering to the above constraints. However, regular expressions can describe very complex sequences and while it is possible to obtain a state diagram of the system to accept a given regular expression it is sometimes easier to lift the restrictions and consider only the sequences or sets of sequences described by the regular expressions. The diagram we then obtain is called a transition graph.

A transition graph consists of suitably identified nodes and directed arcs which are labelled by the input symbols connecting them. At least one of the nodes is termed as a starting node, identified by a short unlabelled arrow going into it, and at least one of the nodes is an accepting or terminal node indicated by a double circle. It is not necessary to have an arrow leading out of a node for every input symbol; also there can be more than one arrow from a node labelled by the same input symbol.

A sequence of directed arcs of a transition graph is called a path and every path describes a sequence of input symbols determined by the symbols associated with the directed arcs. A sequence is said to be accepted if there exists at least one path between a starting node and a terminal node which describes the sequence; otherwise, it is said to be rejected.

A regular expression describes all the sequences accepted by an automaton. Thus, from above, it is clear that to construct a transition graph for a given regular expression, it is only necessary to generate nodes and arcs to contain paths describing the sequences in the regular expression in the simplest way.

For example, consider the regular expression $\underline{R} = 10^*1+00$. To construct a transition graph for this, assume a starting node A.

An input of 1 will cause a transition to a node B. Any number of 0's following this 1 will cause a transition back to B and finally a second 1 will lead to the terminal node C. Similarly, a 0 in the starting node will lead to a node D and a second 0 will lead to the terminal node C. This transition graph is shown in figure 7.

This procedure can be extended to more complicated regular expressions by merely segmenting the sequences in the expression and suitably coalescing their transition graphs. As an illustration the transition graph for the regular expression $\underline{D}$ in (15) is shown in figure 8 which was obtained by straightforward inspection only.

2.13 Conversion to a Deterministic Form

In general a transition graph is non-deterministic and the automaton described by it cannot be directly realised. However, a systematic procedure does exist to convert any non-deterministic graph to a deterministic graph which means that where it is easier and more convenient a non-deterministic graph may be derived with the certainty that a deterministic graph may be obtained. The procedure is given below and is illustrated with the transition graph of figure 8.

Step 1. Begin by establishing a node to represent the set of all starting nodes.

Step 2. Find all the successors of the starting node for each input symbol and create a new node for each distinct set of successor nodes. If a particular (new) node does not have any successor for a particular input symbol then a successor node $\emptyset$ is generated. This node represents the condition when a non-acceptable string is applied to the automaton. In state diagrams this would be equivalent to the "don't care" or "can't happen" conditions. Once the automaton has

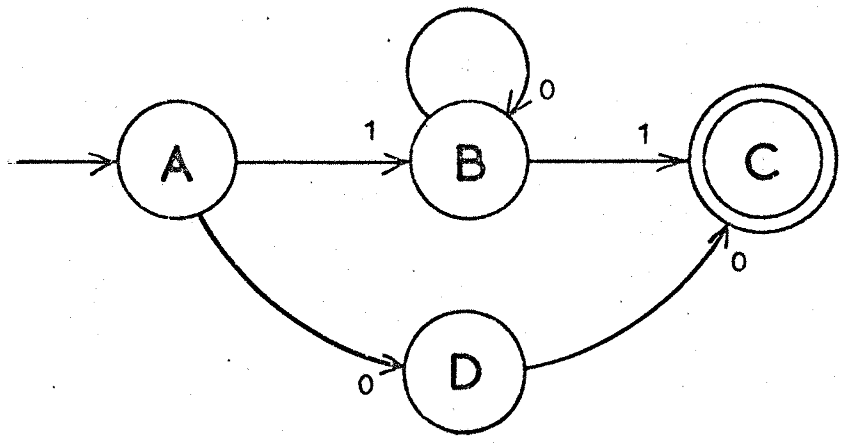


FIGURE 7

TRANSITION GRAPH FOR $R = 10^*1 + 00$

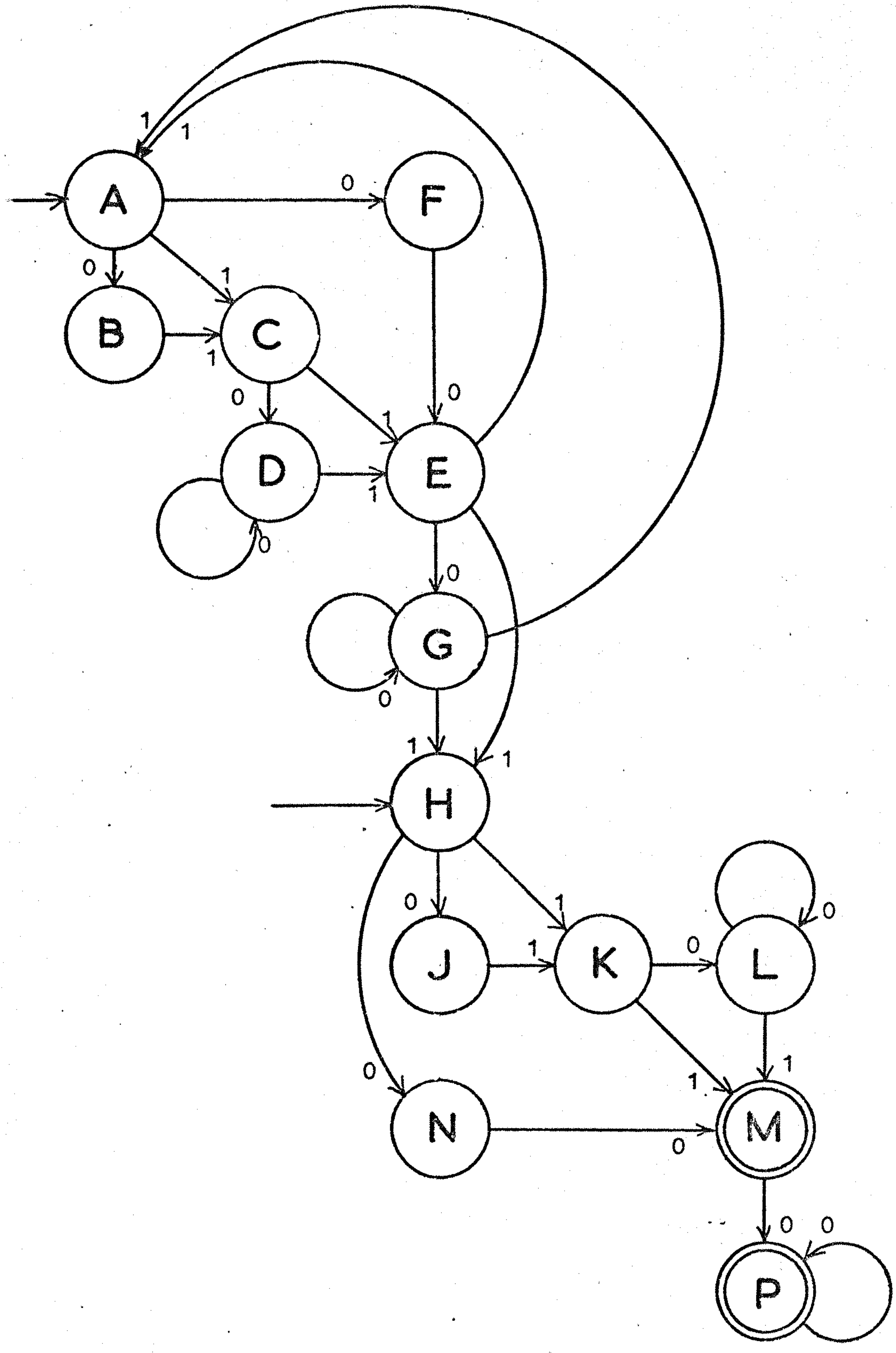


FIGURE 8. TRANSITION GRAPH FOR

$\underline{D} = (((01+1)0^*1+00)0^*1)^*((01+1)0^*1+00)0^*$

reached the $\emptyset$ node any further input sequences cause transitions back to this node. For this reason this condition is sometimes called a fault state.

Step 3. Repeat step 2 for every new node generated until all distinct sets of successor nodes of the non-deterministic graph are covered.

Step 4. Any new node representing a set of nodes which contains a terminal node is also made a terminal node of the deterministic graph.

The above procedure is simplified by building up a successor table using the results of steps 2 and 3, in which the columns represent the input symbols and the rows the nodes of the deterministic graph.

Applying the above procedure to the transition graph in figure 8, we obtain the following:

Step 1. There are two starting states, A, H. We create a node AH to represent the set of nodes $\{A,H\}$.

Step 2. The 0-successors of A are B and F and of H are J and N. Let us name the set of nodes $\{B,F,J,N\}$ as BFJN. Similarly, the 1-successor of the set $\{A,H\}$ is the set $\{C,K\}$ represented by the node CK.

Step 3. By repeating step 2 for nodes BFJN and CK and so on we construct a successor table shown in figure 9.

Step 4. Since M and P are terminal nodes in the non-deterministic graph, the nodes EM and GP of the deterministic graph are also made terminal nodes and this is indicated by making the outputs in these nodes equal to 1.

Clearly, since each node has only 1 successor for each input-symbol and there is only one starting mode, the successor-table defines a deterministic automaton. The description in figure 9, therefore, is identical to a state table, with the node AH representing the state AH, etc. By inspection we note that the nodes CK and DL are equivalent and also that the nodes EM and GP are equivalent. Thus we can derive a state table with 4 states to accept the regular expression $\underline{D}$ in (15) by using the conversion procedure. This state table and the corresponding state diagram are shown in figures 10 and 11 respectively.†

2.14 Conclusions

In this chapter we have briefly introduced the language of regular expressions and discussed its applications to finite state systems. We note that an algorithmic procedure does exist for obtaining a regular expression for a given state diagram [6]. It is obvious that the complexity of the regular expression increases rapidly with the number of states; it increases even more when the size of the input alphabet increases. We also note that the final regular expression depends very much on the intermediate steps taken and several regular expressions seemingly completely different may represent the same system. Some theorems do exist to manipulate regular expressions [12],[13], but since no canonical form is available for regular expressions, no algorithmic procedure exists to prove the identity of equivalent expressions.

In some cases regular expressions for a particular system can be written down directly. However, this is certainly not the general case and we find little justification in statements, such as that by Oglesby [26], "... then the logic designer has only to

† Compare figure 11 with figure 3 from which the regular expression $\underline{D}$ was originally obtained.

	INPUT 0	INPUT 1	OUTPUT
AH	BFJN	CK	0
BFJN	EM	CK	0
CK	DL	EM	0
EM	GP	AH	1
GP	GP	AH	1
DL	DL	EM	0

FIGURE 9

SUCCESSOR TABLE FOR D

STATE \ INPUT	0	1	OUTPUT
1	2	3	0
2	4	3	0
3	3	4	0
4	4	1	1

FIGURE 10

STATE TABLE FOR D

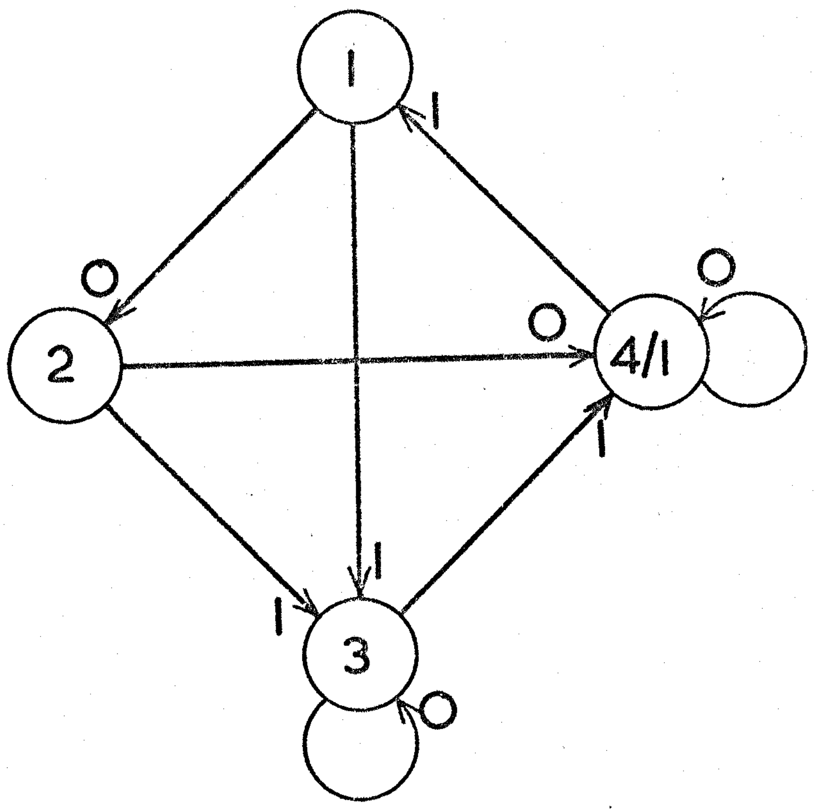


FIGURE II

STATE DIAGRAM FOR D

transform the word statement into a regular expression - an extremely simple task."! The problem of explaining a given regular expression by a word statement is even more difficult and this may be readily verified by examining the regular expressions from this chapter.

The problem of deriving a finite state system to accept a given regular expression has been tackled in three ways. The last technique, that of transition graphs, is the simplest and is algorithmic in nature and consequently may be programmed for a computer fairly easily. This still leaves us with the problem of obtaining a regular expression describing the system; the difficulty becomes more acute if multiple outputs are handled and impossible when the system to be described is non-finite, i.e. where an infinite or very large memory is coupled to a finite state system.

We conclude from the discussion so far that the language of regular expressions, by itself, is inadequate to describe most practical systems. Their best use is when describing sequence detection, and thus may be ideally employed in syntax checking of programmes [13] .

3. Languages Describing Microprogrammed Systems and Their Applications

3.1. Introduction

In general any digital system may be considered as a finite state machine and techniques of switching theory described in [1], [2], [3] or regular expressions discussed in Chapter 2 may be applied in the design of such a system. However, in large systems, such as the present-day computers, the number of states is so large that the theory of finite automata tends not to be very useful. On the other hand, no formal theory similar to switching theory yet exists for a large system which has provisions to exclude unnecessary detail and still be rigorous enough to define the behaviour of such a system concisely and precisely. An attempt can, however, be made towards a formalism by examining the present-day large digital systems.

Large digital systems are essentially instruction execution machines. The design of these systems involves providing for the facilities to store the data and the way the data is manipulated to execute a given set of instructions. The system can, therefore, be partitioned into two parts and we obtain the classic model, figure 1, suggested by Glushkov [56] which consists of an operational part containing the data storage and manipulative facilities, and a control part which provides signals to the operational part in a certain sequential mode to activiate the manipulation within the operational part. The control part can also specify certain tests, the results of which in turn can alter the sequencing of the control part.

The data, which is usually a string of 0's and 1's, is stored in memory units called registers. The operational part contains a collection of registers, and combinational logic to create data paths between the registers and to perform logical functions on this data. The flow of data in such a configuration is referred to by Register

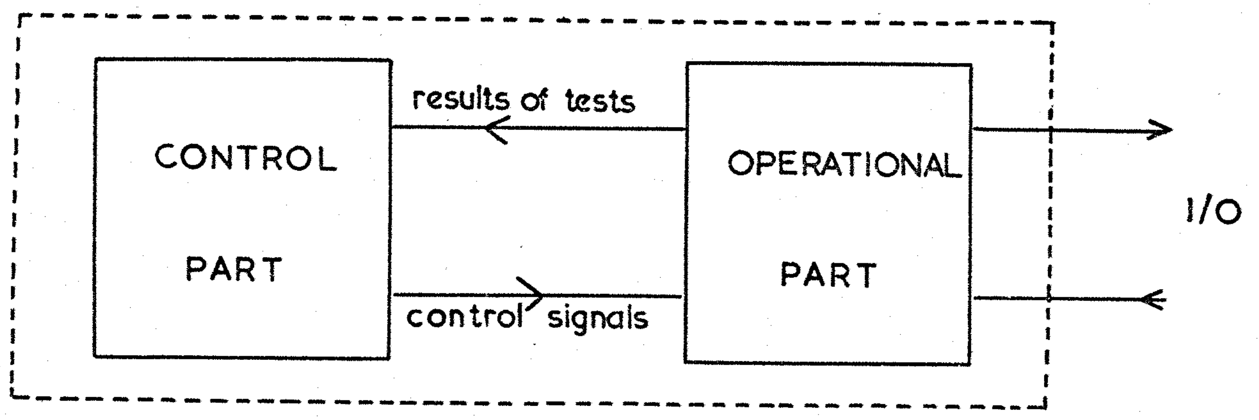


Figure I Glushkov model of a computer

Transfers; and the function of each instruction may be expressed in terms of register transfers in an algorithmic manner.

Obviously it is possible to produce one set of registers and register transfers for each instruction in the machines repertoire; however, this will inevitably result in a large amount of redundancy. The logic designer, therefore, proposes an intuitively derived set of registers sufficient for the instruction execution, and also limits the operations to an optimal number determined by the size and speed requirements and the instructions themselves. These operations are called elementary operations. Elementary operations are also constrained so that once initiated, they do not need further inputs from the control part for completion, and typically reflect the available resources. For example, with integrated circuit hardware technology the elementary operations on data may include logical AND, logical OR, negation, etc. but may not include addition or subtraction, whereas in Large Scale Integrated systems addition and subtraction may easily be treated as elementary operations.

The signals from the control part initiating the operations in the operational part are called micro orders, and the algorithm of an instruction in terms of these micro orders is called a microprogram*. Thus the control unit contains a collection of microprograms for the instructions in the machines instruction set.

---

* Husson's [62, p.20] definition states that a microprogram is a set of micro-orders stored in a control store on a word basis. We remove this restriction and hence, generalise the definition to cover other methods of implementing the control part including the "hard-wired" method.

The design of a digital system with the structure described above consists of defining the storage and manipulative facilities, writing suitable microprograms to interpret the instructions in the instruction repertoire, and obtaining a suitable control part to execute the microprograms. Clearly then, the functions of such a system can be expressed by the microprograms and this suggests a method of formalising the design procedure for a large system.

With the above approach the configuration of the operational part is fixed and microprograms are written in terms of the available facilities. A microprogram can also be viewed from another angle and used to determine the manipulative facilities in terms of elementary operations that are necessary to execute the instructions. In this way the control part and a section of the operational part can be synthesized from the microprogram specifications.

Projecting even further, a microprogram can be assumed to be an algorithm interpreting an instruction. It should then be possible to extract sufficient data to determine what storage facilities are required and how they are manipulated, i.e. a fuller synthesis approach can be taken based on a microprogram type specification.

Microprogram specification, we therefore believe, is an important step in the formalisation of design of large systems.

The next step obviously is to construct a suitable language to specify microprograms in a way that is easy to comprehend, precise and concise. The requirements on the language become more acute in a Computer-Aided-Logic-Design (CALD) environment since the specifications must be sufficiently low level for automatic interpretation, and at the same time, high level yet flexible for the designer to work at his own level without necessitating detailing.

Several languages have been devised to specify microprograms and the associated architecture of the operational part with varying degrees of success. We discuss these languages below.

3.2 Reed's Register Transfer Language

A language to describe the transfers between registers was first proposed by Reed in 1952 [78]. An account of this language is also given in [34]. This language is simple and has a small vocabulary; however, we shall examine it in detail here to elucidate the concepts involved before progressing to the more complex and higher level languages.

In this language a register refers to a hardware block consisting of an array of memory elements each capable of storing one bit of data, i.e. flip-flops. It is identified by an alpha character or a string of alphanumeric characters beginning with an alpha character. The register may be indexed suitably to identify individual flip-flops if necessary and this also provides a facility for using registers of different lengths. Operations are usually specified between the full registers; however, the individual flip-flops may also be selected if required. In the former case, the expansion in the translation process produces the latter form.

Consider a machine consisting of three registers, A, B and C, each 16 bits long. Let the operations to be performed depend on bit 16* of register C: if this bit is 0 then a logical AND is performed with the contents of the registers A and B, otherwise a logical OR is performed. The result then is placed in register C. The register

* The convention adopted here is to consider the contents of each register as a binary representation of a number and to refer to the leftmost bit as the most significant bit. The least significant bit, unless otherwise specified, will always be bit 1.

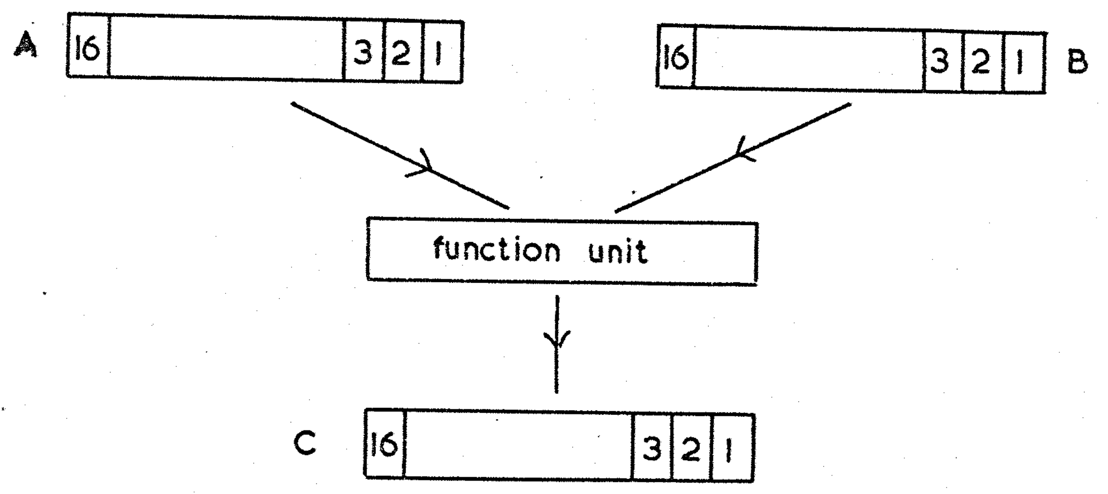


Figure 2 A simple machine

transfer statements to describe this action would be written as

$$|C(16)'| \quad : \quad A \& B \rightarrow C \qquad (1)$$

$$|C(16)| \quad : \quad A + B \rightarrow C \qquad (2)$$

where &, + and ' represent the logical AND, logical OR and negation, respectively. The vertical bars is a shorthand notation to indicate that the action on the right hand side of the colon is to be executed if the logical value of the variables between the bars is 1, i.e. | | : can be translated to the Algol statement <u>if</u> ... <u>then</u>. The variable may be substituted by a boolean function if necessary. The arrow is a short form notation to indicate the replacement of the contents of the register at the head of the arrow by the variable or boolean function specified at the tail of the arrow. Therefore, transfer 1 correctly translated means "provided that bit 16 at register C at a time t is not 1 replace the contents of register C at time t+1 by the AND of the contents or egisters A and B at time t, assuming that the transfer requires a unit time".

The transfers 1 and 2 may be expanded to

$$|C(16)'| \; : A(i) \& B(i) \rightarrow C(i), \quad i = 1,2,\ldots 16 \qquad (3)$$

and $$|C(16)| \; : A(i) + B(i) \rightarrow C(i), \quad i = 1,2,\ldots 16 \qquad (4)$$

In this example all the elements of the registers were involved in the transfers simultaneously, but it is quite possible that only a part of each register is affected. Suppose that only the last three significant bits were used in the transfer and the others were unaffected, then this could be writen as

$$|C(16)'| \; : A(i) \& B(i) \rightarrow C(i), \quad i = 1,2,3 \qquad (5)$$

and $$|C(16)| \; : A(i) + B(i) \rightarrow C(i), \quad i = 1,2,3 \qquad (6)$$

$$C(j) \rightarrow C(j), \quad j = 4,5, \ldots 16 \qquad (7)$$

but it would be sufficient to write only transfers (5) and (6) without losing clarity.

In all the above cases the value of C(16) determined the operation on each element of the registers and it can be considered as <u>a scalar multiplier</u>. For example, transfers (1) and (2) can be rewritten as

$$C(16)'(A \& B) + C(16)(A + B) \rightarrow C \qquad (8)$$

It is easy to see that the operations described above have a direct correspondence with hardware elements. In Reed's original language other operators, such as shifting and addressing were also included again having direct hardware counterparts.

Schorr [82], [93], used this language and developed algorithms to generate the necessary boolean equations for the set and reset terminals of the flip-flops, i.e. the synthesis procedure. This translation process is in two steps: a) each statement is converted to produce the required set and reset equation, and b) all the individual boolean equations for each set and reset terminal are OR'ed together. Thus from transfer (1) we get

$$C(i)/1 = C(16)'.A(i).B(i), \quad i = 1,2,\ldots 16 \qquad (9)$$

and

$$C(i)/0 = (C(16)'.A(i).B(i))', \; i = 1,2,\ldots 16 \qquad (10)$$

where $C(i)/1$ is interpreted as bit $C(i)$ is set to 1 if the logical value of the expression on the right hand side equals 1, i.e. the boolean equation for the set terminal. Similarly from transfer 2 we get

$$C(i)/1 = C(16)(A(i) + B(i)), \quad i = 1,2,\ldots 16 \qquad (11)$$

$$C(i)/0 = (C(16)(A(i) + B(i)))', i = 1,2,\ldots 16 \qquad (12)$$

Grouping these boolean equations we get

$$C(i)/1 = C(16)'.A(i).B(i)+C(16)(A(i)+B(i)), \quad i = 1,2,\ldots 16 \quad (13)$$

and

$$C(i)/0 = (C(16)'.A(i).B(i) +(C(16)(A(i)+B(i)))', \; i=1,2,..16 \quad (14)$$

In Reed's original language the sequencing was implied by the order in which the transfers were written; however this had limitations when branching or repeats had to be specified. Schorr included timing pulses as part of the boolean functions of the conditions and introduced a type of 'goto' transfer. By this method the above example could be written as a sequence of transfers as

$$\begin{array}{lll} \underline{\text{Start}} & : & 1 \rightarrow t_1; \\ |t_1 . C(16)'| & : & A \,\&\, B \rightarrow C,\ 1 \rightarrow t_2; \\ |t_1 . C(16)| & : & A + B \rightarrow C,\ 1 \rightarrow t_2; \end{array} \qquad (15)$$

where start initiates the sequence of transfers in the machine and t_2 is a condition specifying the next transfer. t_1 and t_2 therefore are the outputs of the circuitry controlling the transfer and start and C(16) are the inputs to it. It is a simple matter from the above description to extract the logic for the control circuitry.

Schorr also suggested a method for analysing digital systems by converting boolean equations into register transfers. It requires that all the registers and control variables are declared as such and that the boolean equations specify set-reset conditions and that they are in a sum-of-products (SOP) form. The analysis program makes successive compilation passes, first separating out the control and register transfer expressions and then building up the register transfers. The transfers so obtained, obviously reflect the hardware structure and operation. Schorr did comment on the difficulty of obtaining the transfers in terms of composite events, i.e. involving non-logical operations. Nevertheless, the procedure does allow a concise and formal description to be obtained for an already existing system which then can be used for re-synthesis or simulation.

The language described so far is simple, symbolic and easily learned; and there is a direct correspondence between it and the logic hardware. However, since it has a small vocabulary, a complete description is lengthy. Another disadvantage is that the language is too symbolic to be suitable for communication between the logic designers and members of other disciplines.

Gorman and Anderson [57] enhanced this language slightly by introducing simple arithmetic operators, facility for subroutines and declarations of special hardware blocks, such as a parallel adder or a counter, whose internal functions may not need detailing, particularly if it is to be implemented with L.S.I. circuits. Algol type operators, such as if, then, else, goto, allowed a more concise description and made it more "readable".

The translation process with this language, due to Proctor [77], generated a comprehensive table specifying all the registers involved in each transfer, the operations, any additional components used for the transfers and the timing. The table was filled as far as possible with the data from the register transfer description and then completed, particularly with regards to timing, by the designer. The table was then analysed to achieve the shortest possible execution time. The table then contained data in a similar format to that used by Schorr, from which boolean equations could be generated.

Ilovaiski and Lozowskii [63] described a method to synthesize logic for a computer from a formal specification which was not too unlike Gorman and Anderson's method. The formal description was divided into two parts, a) a declaration part, and b) an operational part. The declaration part declared facilities pertinent to data storage and address mechanism, namely i) storage devices and their bit capacities, ii) methods of representing numbers with their formats, iii) address formats, iv) methods used to modify addresses, v) instruction system, vi) principles used to organise the data, and vii) a table defining the durations of all standard operations expressed in arbitrary units.

The operational part consisted of register transfers and branching information in a similar way to Schorr's and organised with a single elementary operation per line each with a unique label.

The synthesis algorithm first assigns the time durations to each step in accordance with the initial declarations. Consecutive operations are then merged to occur in the same "time slot" unless a variable on the left hand side of one operation is used on the right hand side of the other, or if conditional transfers in some way are affected by the operation to be merged. This results in several micro operations to be grouped together to form new larger micro operations. The subsequent steps are identical to those discussed previously.

3.3 Languages based on Programming Languages

One of the goals of a design language for describing logic is that the language may be used in a computer-aided-design environment. Consequently, a description in such a language is to be processed by a computer and in effect constitutes a program for a computer. It would be natural to ask the question "why not an already existing programming language?" as the designer then would be able to use currently developed software providing good flexibility. Another advantage is that it reduces the overheads of "learning" a new language which also means that the language could be used easily as a standard language for communication between members of different disciplines. On a closer exmination, however, it is found that the capabilities of programming languages tend to be more numerically oriented and features pertinent to logic design, including synthesis and simulation, are not handled efficiently. Nevertheless, it should be possible to augment a programming language to make it suitable for microprogram definitions and still retain its overall structure.

Several languages have been proposed, based on FORTRAN, ALGOL, IVERSON and PL/1. These are discussed below.

3.3.1 FORTRAN

The only language based on Fortran was proposed by Metze and Seshu in 1966 [74]. A computing system is viewed as consisting of separate automata each possessing its own controls and sub-controls. With this type of modelling it is possible to describe parallel and asynchronous operations and as such represents the first real attempt at describing large systems with realistic properties.

The description in this language is given in two blocks. In the first, the system constraints such as the channel capacities, simultaneity of automata, a measure for cost-effectiveness and global constraints and variables are given. The second block defines each automaton with its declarations and register transfers. The transfers themselves were restricted to boolean operations; the other operations such as arithmetic functions were called as subroutines which in turn were detailed as boolean operations.

One of the reasons behind using subroutine call structure is that each subroutine could be detailed in several different ways all producing the same result but having different overheads and which could be stored in a back-up library. The "compiler" then could search the library to choose one of the routines best suited for the application. This concept is very useful and will be explored further later on.

The language allows simultaneous operations and if an interdependence is encountered a method of waiting to allow correct sequencing is defined. Another facility included allows the global conditions to be over-ridden for a particular operation. These suggest that any general asynchronous machine may be defined using this language. A language proposed by Schlaeppi (discussed later) had a limited facility of this type but the concept developed by Metze and Seshu was a significant improvement over the earlier ones.

However, to the author's knowledge, no translator was constructed for this and no further references have been available.

3.3.2 ALGOL

Schlaeppi proposed a language based on Algol in 1964 [81], which was one of the first to be capable of describing a computing system in a hierarchical manner. He introduced four notions for this, namely a step, a sequence, a function and a group. A step was defined as a set of elementary operations which are explicitly declared to be executed in parallel and a succession of steps constitute a sequence. In cases where the internal structure of a particular section was not known or need not be known, then its terminal behaviour was called a function; and finally a group represented a collection of sequences or functions under a common control. Thus a degree of partitioning of the system could be indicated in the description. Secondly, the function facility allowed the machine organisation to be described with broad structural features; the subsequent expansion, as would be necessary in synthesis, could be done by refining the description.

Schlaeppi also introduced time indication in description, firstly by declaring time for standard operations in advance - in a way similar to that adopted by Ilovaiski and Lozowskii - then by timing each step either in units for synchronous operations or by making a transfer conditional upon a ready signal and thus setting up an interlock for asynchronous operations. In addition each group contained an "availability indicator" which if set implied that the group was busy, and which could be used to augment timing interlocks.

The transfers within the steps were written in an Algol-like form and usually were between registers. However, Schlaeppi introduced a distinction between permanent signals such as the contents of registers and transient signals such as busses. The transients could also be used in the transfers without being explicitly declared.

Chu published a language CDL [39] in 1965 which had a closer resemblance to Algol. A description in CDL begins with the usual declaration of registers, sub-registers, memories, terminals and operations. A terminal is useful in describing signals which are not stored in registers but can be accessed from the outside world; it is in some ways similar to a bus mentioned previously.

Labels corresponding to the state of the control part are attached to the transfers. Parallel transfers are indicated by attaching the same label to the relevant transfers. However, no facilities are available for indicating asynchronous operations and this is a drawback.

The language allows the inclusion of special operators whose definitions may be separately detailed and sequences in a similar way to sub-routine calls. The control, however, is still common and the decentralisation of control extensively used in real system cannot be indicated, which also means that a hierarchical type of description is not possible.

This language, therefore, forms only an extension to Reed's language with a different syntax.

CDL was later improved by Chu, McCurdy and Mesztenyi [40-42, 71, 73], who also illustrated methods of boolean translation and simulation. The translation process consists of four phases which are as follows.

i) The design specification is scanned to produce a table with as many rows as microoperations.

ii) The table is analysed to generate in effect two tables, one for control part and the other for operational part showing the input output conditions for each step.

iii) Boolean products are generated from these tables, and finally

iv) These are sorted and combined to produce boolean equations for the register inputs and the outputs.

In 1966, Parnas [76] also published a version of Algol to describe synchronous logic in which he introduced a notion of time block to describe parallelness of operation in a similar way to shared labels in CDL. The methodology of the language views the system to be described for its behavioural properties only and as such neither the structure of the system nor the "how" may be described. The latter restriction could lead to difficulty when synthesizing large systems. Secondly, as with CDL, only synchronous systems may be described with this language.

This language, however, could be used quite well for simulation of synchronous systems.

Darringer [45] modified this language to include structured information. A designer using this language could specify registers of different type such as octal, binary, character, etc. which in some cases would be useful. However, the type declaration of a register fixed its usage and dynamic interpretation was not possible. For example a binary interpretation of a decimal register may be desirable and even necessary in certain cases.

As in CDL, the operations in this language are synchronous and limited to one clock pulse; however, it is possible to indicate an operation over multiple clock pulses. An "if ever" operator similar to the "on condition" operator of PL/1 allowed certain operations to occur asynchronously and in parallel with the main program. The simulation programs, however, did not handle the semantics of these statements correctly. For synthesis, Darringer offered some comments on the translation into hardware but did not suggest any concrete algorithms.

Wilber [87] also gave a version of Algol which was very much similar but in addition provided a facility for implicit timing.

Okada and Motooka [75] proposed a highly hardware oriented language also based on Algol which unlike the languages described so far fully exploited its block structure. The description in this language is divided into five levels. At the lowest level, level 1, a hardware definition in terms of primitives such as gates, flip-flops, and delays is given. It is also possible at this level to include black boxes whose internal structure need not be detailed but which can be used as primitives. The description at level 2 is a functional relation corresponding to the description at level 1.

The description at level 3 shows the system behaviour at each clock pulse including simple explicit sequencing which may be used directly to implement the hardware configuration and control. Finally an algorithmic description may be given at levels 4 and 5, the difference between them being that at level 5 it is more Algol-like and is similar to that by Chu and the description at level 4 is more hardware related as in Reed's language.

The system to be described is modelled as a module containing sub-modules, each of which in turn may contain sub-submodules and so on. This is reflected in the block structure used in the language. In addition a change of block also allows a change of level; thus a desired detailing may be achieved by suitable nesting of blocks.

The above modelling is very useful since it allows the designer to choose to detail the parts he wants specifically defined and leave the rest as default options, and is a good design aid. A limitation, however, is that modules operating in parallel cannot be described.

The proposed translation process involved the changing of levels with the help of library definitions of arithmetic operators, macros, hardware items and modules in an interactive mode. No results were available on the efficiency of this process; however, we feel that the interactive approach with suitable recourses to a library is a right approach and will be exploring this further.

3.3.3 Iverson's APL [49, 59, 64, 66, 70]

A register transfer language essentially describes the hardware layout and the interconnections between them. The micro-programs written in such a language therefore, reflect the hardware constraints placed upon the system. A register-transfer-like language can also be developed for software which will similarly reflect the constraints put on it, namely the capabilities of the hardware processor. However, in both cases the algorithmic description is not sufficiently abstract for "evolving" a design and merely provides a means for mechanisation of routine tasks. A formal description independent of such constraints is required in the design stages.

Iverson proposed a language, APL, which was meant to be universal in a sense that it has a built-in hierarchy to express functions which are usually considered to be hardware oriented as well as those which are usually software oriented. This is in direct contrast with the other programming languages since either they are of too low a level and are strongly machine dependent or they are of too high a level to have sufficient resoltuion for, say, bit operations. The operations in APL can be specified at a bit level, an array level or a matrix level without loss of detail and thus offer facilities for a precise and concise notational description of algorithms which is machine independent, and consequently ideal for a wide range of uses.

A rigorous and full account of the language is given in [66]; however, a brief description of some of the operations, which are likely to be more useful, is given below.

The variables in the language are defined as either scalars or arrays which are one dimensional (vectors) or two dimensional (matrices). The scalar operations are expressed in much the same way as in other programming languages; these are extended on an element by element basis to apply to array operations.

For example,

$c \leftarrow a \;@\; b$ where @ is any operation

is a scalar operation and of course all the variables are scalars; whereas

$\underline{c} \leftarrow \underline{a} \;@\; \underline{b}$

is a one dimensional vector operation and it is interpreted as

$\underline{c}_i \leftarrow \underline{a}_i \;@\; \underline{b}_i \qquad i = 1,2,\ldots, \nu(c)$*

and obviously the dimensions of each of the three variables must be the same and its magnitude is determined by (c). The matrix operation of the same form is written as

$\underline{C} \leftarrow \underline{A} \;@\; \underline{B}$

meaning

$\underline{C}^i_j \leftarrow \underline{A}^i_j \;@\; \underline{B}^i_j \qquad i = 1,2,\ldots, \nu(\underline{A}),\ j = 1,2,\ldots, \mu(\underline{A}).$

where $\nu(A)$ and $\mu(\underline{A})$ are the column and row dimensions respectively of the matrix $\underline{A}$.

The elements of the vector can be any numeric or logical quantities or even any alpha-numeric or other characters. The one dimensional vector operations are particularly important in digital system design since they can be used to indicate register operations directly, and the matrix operations can be very useful in data manipulations as in symbol processing applications.

* a 1 origin indexing is used here and the leftmost element is element 1.

The language comprises all the usual arithmetic operators, such as addition, subtraction, multiplication, division and exponentiation, and the logical and relational operators. Shifting type of operations, which are of significant importance in digital system design, are also included. However, the particularly strong powers of the language come from the special operators, such as reduction, masking, expansion and compression of arrays. The reduction of a vector is defined as

$$y \leftarrow @/\underline{x}$$

where y is a scalar quantity and is set to $\underline{x}_1 @ \underline{x}_2 @ \ldots @ \underline{x}_{\nu(\underline{x})}$

Extending this to matrices, we have:

$\underline{y} \leftarrow @/\underline{X}$ columns are reduced

$\underline{y} \leftarrow @//\underline{X}$ rows are reduced.

If the operator is replaced by a binary vector, we get a selection operation by which the elements specified by a 1 in the selection vector are picked out to form a new vector. For example,

if $\underline{x} = (D,I,N,E,S,H,P,A,I)$

and $\underline{u} = (1,0,0,0,0,0,1,0,0)$ and $\nu(\underline{x})$ must be the same as $\nu(u)$

then $\underline{y} \leftarrow \underline{u}/\underline{x}$

will make $\underline{y} = (D,P)$

A more practical example is when certain bits of the instruction word are used as the instruction code etc. this type of operation can be used effectively.

Masking is shown as follows

$$\underline{z} \leftarrow /\underline{x};\underline{u};\underline{y}/$$

and it means $\overline{\underline{u}}/\underline{z} = \overline{\underline{u}}/\underline{x}$ and $\underline{u}/\underline{z} = \underline{u}/\underline{y}$ and obviously

$$\nu(u) = \nu(x) = \nu(y) = \nu(z).$$

A related operation and of considerable importance in file sorting is the meshing operation shown as

$$\underline{z} \leftarrow \backslash \underline{x};\underline{u};\underline{y}\backslash$$

and $\overline{\underline{u}}/\underline{z} = \underline{x}$ and $\underline{u}/\underline{z} = \underline{y}$. It follows that $+/\overline{\underline{u}} = \nu(\underline{x})$ and $+/\underline{u} = \nu(\underline{y})$.

Another very useful operator is the base 2 value operator, which is particularly important in memory addressing type operations. Suppose $\underline{r}$ is a register and contains the address of a word in the main memory $\underline{M}$ and it is necessary to extract this word and put in into the register $\underline{a}$, then this is written as

$$\underline{a} \leftarrow \underline{M}^{\perp \underline{r}}$$

There are some special vectors which are very useful in digital system design and they are listed below.

full	$\underline{\epsilon}(n)$	All elements of the n length vector are 1.
unit	$\underline{\epsilon}^{j}(n)$	jth element is 1 all others 0.
prefix	$\underline{\alpha}^{j}(n)$	the first j elements are 1 the rest 0.
suffix	$\underline{\omega}^{j}(n)$	the last j elements are 1 the rest 0.
interval	$\underline{i}^{j}(n)$	elements are numerically consecutive beginning with j.

These vectors in conjunction with the special operators mentioned before can be used in a very versatile manner allowing a concise yet precise description. An example of a microprogramme in this notation has been included in the appendix.

3.3.4 APL as a Design Language

APL may be used, and has been used [48,59], to specify the microprograms of an existing computer but it becomes just another way of writing register transfers and useful for analysis only. However, the main flexibility of APL lies in being able to describe algorithms in a machine-independent way. Thus it should be possible to use it as design language.

In its full form the language is very general and contains a comprehensive set of operators. A hardware realization of a machine capable of executing all the operations and facilities would be very large indeed; on the other hand, a suitable subset of the language could be easily and directly implemented into hardware and the remaining facilities translated in terms of this subset. Since there is a natural hierarchy in the language, the higher level operators may be expressed in terms of the lower level without much difficulty and the translation process can be fairly straightforward. The system developed [31,86] would be very general and be capable of executing most, if not all, statements written in APL.

One of the drawbacks of APL in a computer environment is the large number of special symbols which are required to express the operations correctly. Iverson [65] suggested a scheme for transliterating these symbols in which one line of APL program is converted into two lines of program written with the more conventional alphabet, with vertical correspondence between them. Obviously this is not only inconvenient and inefficient but also leads to loss of visual clarity.

Friedman and Yang [51,52,53] have developed a design suite, ALERT, which accepts a microprogram description written in a modified subset of APL and converts it into hardware logic design. In this system a physical device, such as a flip-flop, is associated with each variable. Simple logical operators are implemented directly into hardware and the others are converted using library routines in much the same way as suggested by Okada and Motooka. In the subsequent processing the redundancies are removed and hardware expanded where necessary, followed by a sequence analysis. The output of the program is in the form of boolean equations for the input terminals of flip-flops and can be used by synthesize logic with gates.

ALERT also represents the first real attempt of synthesizing hardware logic automatically via a high level design language on a large scale. A synthesis of an existing computer (IBM 1800) was attempted via ALERT and the results then were compared with the "human" design [51]. Using the gate count as a criterion for the "goodness" of design, the initial design obtained via ALERT was very much worse (about 160%) than the human design; however, an approach was suggested which would improve this considerably (about 33% worse).

Another weakness of APL as a design language is that timing of an operation cannot be indicated. Friedman and Yang defined a clock rate outside the main microprograms, thus they were not able to indicate asynchronous operations. Senzig [84] proposed two separate notations to indicate timing with APL. The first, for synchronous operations, is similar to several mentioned earlier. The second for asynchronous operations, uses three timing states, namely idle, active and standby. A statement is normally idle unless activated by a previous statement in the sequence and it is then said to be active. After completion it activates the succeeding statement(s) and goes into a standby state and if the succeeding statement(s) does become active then the current statement reverts to the idle state. The method described here allows asynchronous operations to be indicated with respect to statements rather than quantities. For example, a statement of the type "whenever --- Do ---" cannot be indicated. The method is also unsuitable for showing operations in independent but parallel modules.

Another important consideration of a design language for digital systems is that the designer should be able to specify the choices of hardware, modules and procedures which are available along with their speeds and criteria to be used for optimality of the design. APL does not provide for this.

We may conclude by noting that APL is very effective in expressing algorithms but in its basic form is not suitable as a complete design language. An augmented version, however, may prove a powerful design tool.

3.4 Partitioned Systems

The languages discussed so far tend to use a Glushkov model for a computer, i.e. one having a single control part and a single operational part, which is quite adequate for describing relatively simple systems or subsystems. However, when dealing with a large system it is natural to partition it into several subsystems each of which is characterised by a Glushkov model and all in turn responding to a common control. The complex control mechanism of such an organisation cannot be suitably handled by the earlier languages which were based on a simpler model; only the language proposed by Metze and Seshu had some facilities for this. A formal approach, however, was given by Duley and Dietmeyer in their language DDL [46,47].

In DDL, a system is viewed as a collection of several subsystems or automata each containing "private" facilities and having access to the "public" facilities, the latter being used for intercommunication between the automata. This corresponds almost exactly to the earlier mentioned concept except that the common control is diffused through the subsystems via the "public" facilities and hence is slightly more general. An almost identical approach was also used in CASSANDRE [32,33,58,69,72] which was published (independently) about the same time as DDL and in project CASD [43,44] in 1970. The major differences between these are the use of different base languages: DDL is built upon Reed's language, CASSANDRE is very heavily Algol derived and the CASD language is a version of PL/1. We shall consider these languages in a little more detail below.

A description in DDL is a description of a collection of automata in a block structure format. It begins with an identification of the outermost block, corresponding to the overall system, and the declaration of common highways, global variables and common registers. Each automaton is represented as a block within the outer block and is described in terms of its registers, terminals, segments and states along with the global variables. The notion of segments allows each automaton to nest further sub-automata and the states are used to specify the sequencing.

The statements are written in a way similar to that by Reed but a larger vocabulary is employed and the description tends to be somewhat ideographic; nevertheless it is relatively simple to interpret with a little practice. The automata indicated are usually synchronous but it is also possible to show asynchronous automata. An important omission, however, is that synchronisation of asynchronous events, as in the WAIT facility proposed by Metze and Seshu, cannot be indicated. On the other hand, it is possible to indicate jump to a specified state in a segment and the return state; this could be employed to define a complex control of shared segments.

The translation process is performed in several passes, ultimately producing a set of transfers in a Reed-like form for the whole system subsequent realisation into hardware from which has been described earlier. The segments are "removed" first. Each segment is checked to see if it has any segment calls in it; if it has, then all the states of the called segment are included in the calling segment, with suitable adjustment for next state and return state specification, and the declarations associated with the called segments are also added to the declarations of the calling segment. The states and the transfers are checked to remove duplication, and redundancies, and the remaining states suitably renamed to distinguish

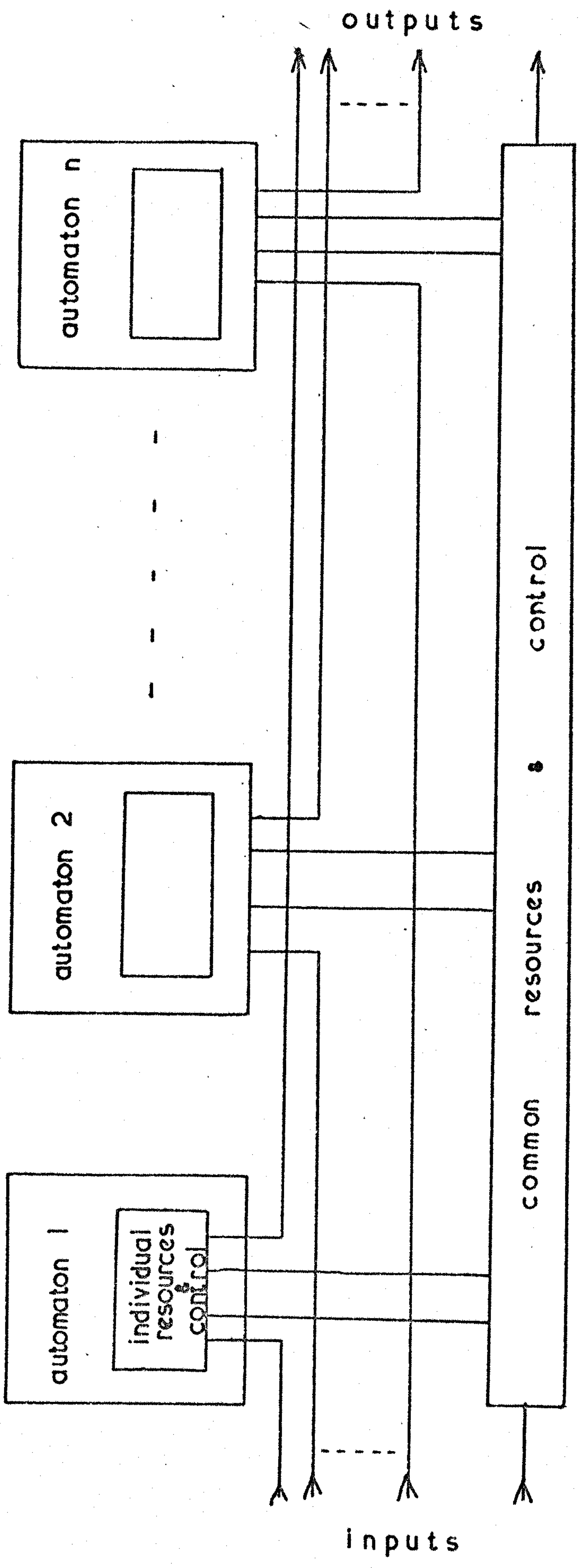


Figure 3 Duley model

between them. The resulting description is then that of an individual automaton.

The next step is to create a state register (unless already declared) such that there is a unique state of the register for every unique state of the automaton. The transfers then can be relabelled to make them conditional upon the contents of the register and the transitions are indicated as a change in the state of the state register. Obviously the size of the register and the coding required to map each state of the automaton into the register will be determined by the mode of operation, i.e. asynchronous or synchronous, etc. Another task at this stage is to express the special operators, including shifting but excluding time shared operations, into a more register-transfer form. The shifts, for example, may be translated into single shift register transfers by associating a counter with it to control the shifting loop. The time shared operators are assumed to be realised only once with suitable gating to control the time sharing. The sequencing logic may be derived by methods already described along with the boolean equations for input terminals of flip-flops and the outputs.

DDL is essentially for hardware representation of partitioned systems. Synchronous systems may be specified precisely using this language; asynchronous systems, however, cannot be very well handled. Another observation is that the philosophy behind DDL suggests that the system to be "designed" has already gone through a design phase and that the language is used merely to describe, in shorthand way, a pre-fixed structure for computer interfacing. Thus the language forms an extension, albeit a complex one, to the Register Transfer Language initially formed by Reed. We feel that the interactive method through which modules may be selected from the backing library is to be strongly recommended.

As stated before, the structure of CASSANDRE is almost identical to that of DDL. The use of standard language, Algol, as a base language, however, makes CASSANDRE far easier to use than DDL. The block structure of Algol is also perfectly fitting to the partitioning concept where each automaton - or unit as it is called in CASSANDRE - can be represented by a block. CASSANDRE also has some minor variations which are discussed below.

A unit in CASSANDRE is assumed to be completely independent from all other units and the communication between the units is done through the inputs and outputs only. Thus a block corresponding to a unit appears similar to a procedure definition with the inputs and outputs as parameters. The declarations following the header contain all the facilities special to the unit as well as any other units used in the description; the latter are declared as external since they are detailed elsewhere. No global variables are employed since such variables may always be included in the input-output list; however this could obviously lead to a long input-output list.

The unit is defined by a set of transfers which may be either boolean connections or synchronous transfers, the latter always being conditioned by a clock-pulse. A repeat operator "for --- equal --- to --- do" is also included to allow iterative arrays to be set up. The sequencing is achieved by labelling the discrete steps as done in DDL, and explicitly indicating a transfer to that label; implicit sequencing is not allowed. The sequencing algorithm extracts this information to set up a table with a correspondence between the labels representing a set of transfers and the conditions necessary to branch to the label, and organising the sequence control to allow execution of the transfers corresponding to all the labels whose conditions are satisfied. Thus explicit and implicit parallelism may be attained. However, it is not very clear how the

```
UNIT:  NAME (INPUT 1, INPUT 2, ... INPUT N;  OUTPUT 1,... OUTPUT M);
       REGISTER        ,     ,  ;
       SIGNAL          ,     ,     ,  ;
         PULSE    C   ;
         CLOCK    C   ;
       EXTERNAL        ,     ,     ;
                  Boolean connections
         ST1 :  C    sequential transfer;  GOTO  ST2;
         ST2 :                             ;

FIN
```

Figure 4 : Structure of a typical CASSANDRE description

sequencing algorithm handles cases where the results of one transfer are directly relevant in the next, particularly if some amount of timing discrepancies occur.

The translation process is quite different from that used in DDL. A table is set up with the declared items along with their scopes. The source description is then converted into a reverse polish notation with pointers replacing the occurrences of variables. An important point about the philosophy of the translation process is that the partitioning defined by the designer is not altered (as was done in DDL). The resulting strings may be used directly for simulation. It is also stated that these may be used for microprogram generation and hardware synthesis; however, no results have been available.

Much of the above description, especially regarding the translation procedure, may also be applied to CASD language. Apart from the change of the base language to PL/1, a few useful additions are also proposed in this system. In particular, these include multi-tasking facilities for explicit parallelism and the WAIT facility acquired from Metze and Seshu. The CASD system, therefore, seems to be more general than the CASSANDRE system. However, at the time of writing this report no results were available regarding the algorithms for translations, especially the translation of asynchronous systems.

3.5 Sequence Chart Analyser

By definition a microprogram is a collection of micro-orders in a particular sequence to utilise the available facilities of hardware or software and also, since it is impossible to achieve instantaneous logic, each micro-order will require a finite time for its execution.

For ease of construction, it is usual to consider a microprogram in terms of a block diagram in which each block represents a micro-order. However, this type of format is difficult to process by a computer and the languages considered so far convert this information to a linear format, which allows ease of processing but loses the visual clarity of sequencing.

Roth [79] published a paper in 1965, giving a method used by IBM which still maintains the visual clarity of a block diagram, and is not as difficult to process. In this method, the block diagram is represented by a sequence chart which is a grid with horizontal divisions as units of time, and the vertical divisions to be used to separate operations. Each transfer is shown as a horizontal bar extending for the length of time of its execution and the corresponding transfer is written over this bar. Conditional transfers are indicated by writing the conditions immediately to the left of the bar; branching is indicated by broken lines and arrows. An entry into the chart is shown a box named chart entry and containing certain conditions. When these conditions are satisfied the chart is initiated. Similarly an exit is shown by a box containing ENDOP.

The sequence chart essentially is a method of presenting a completed design and as such is difficult to be used in design stages. However, in common with the register transfer languages it can be used to syntheisize logic and the associated control, but since it is more difficult to use than the transfer languages, despite its resemblance to block diagrams, it is less likely to be favoured.

3.6 State Tables from Microprograms

It was pointed out earlier that if an abstract description of a digital system can be given or generated within a computer then it would be possible to make use of the well developed switching theory and an overall optimisation can be achieved. But, for even a reasonably sized machine, this is an immense task and consequently some partitioning has to be made. Obviously the abstract description and its subsequent processing will greatly depend on the partitioning used; however, it should be possible to combine the partitioned machines [60] and to try other partitions as a check for optimality.

All the transfer languages considered so far were suitable for an already structured organisation and the translation process generated this boolean equations for the terminals, the outputs and the inputs to flip-flops. The structure assumed is that of several registers, each usually more than one element long, interconnected to allow the various register transfers. Gerace [54] suggested changing this structure to that of several iteratively connected smaller machines and deriving the state-table for each. The structure appears as shown in Figure 5 . An account of his method follows.

To illustrate the method let us consider a simple example where the machine behaviour is expressed by a single transfer only, such as a parallel adder. Let one operand be contained in the machine and when an external operand is input to it let its memory be overwritten by the answer. The usual structure of the machines to achieve this would be as shown in Figure 6 . The operand length is assumed to be 16 bits, register A contains the first operand and B is the second operand. This representation assumes that the outputs of register A remain unchanged despite the changes via the combinational logic; only when the operation is complete that these are allowed to vary. In practice this is done by using clock pulses in the input-output gating or using special flip-flops such as the J-K type. Figure 7 shows the structure in Gerace's method where A(i) is an

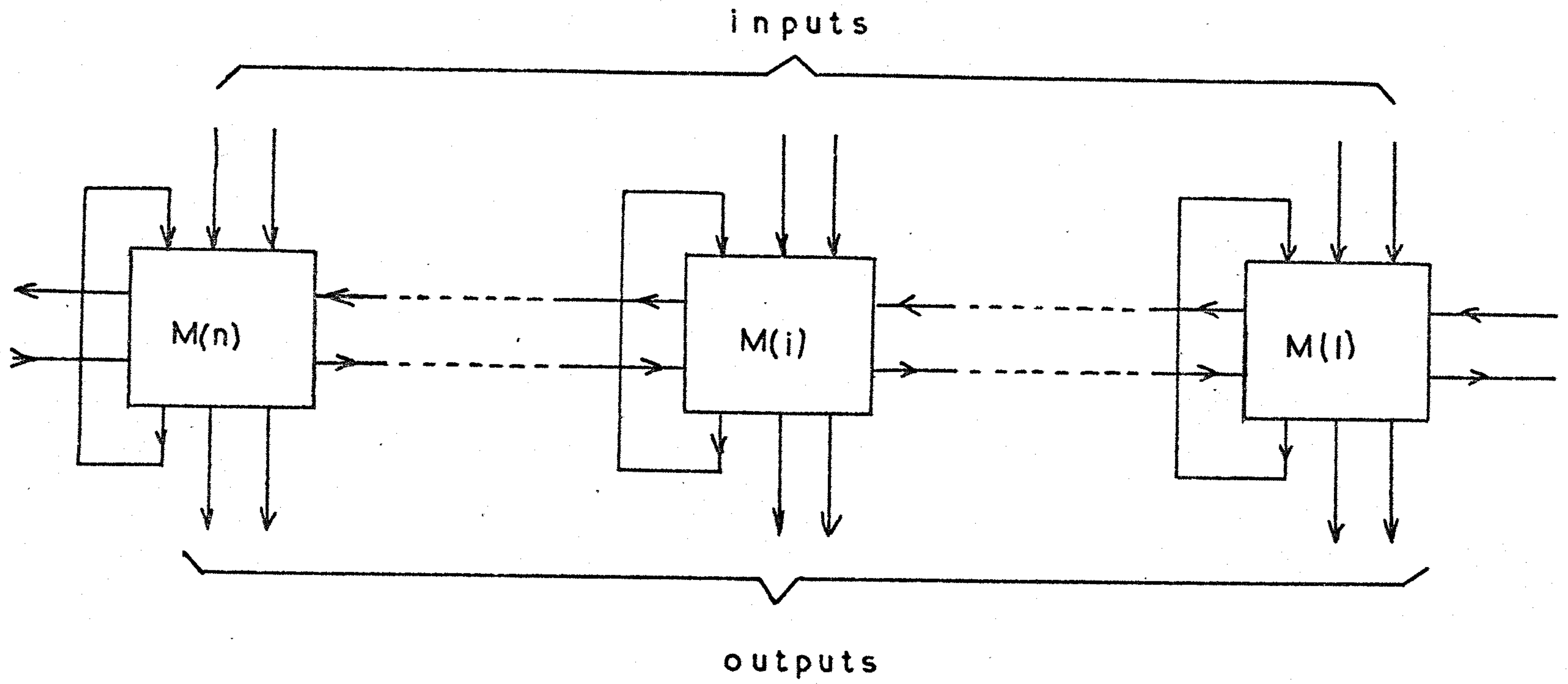


Figure 5 Iteratively connected machine structure

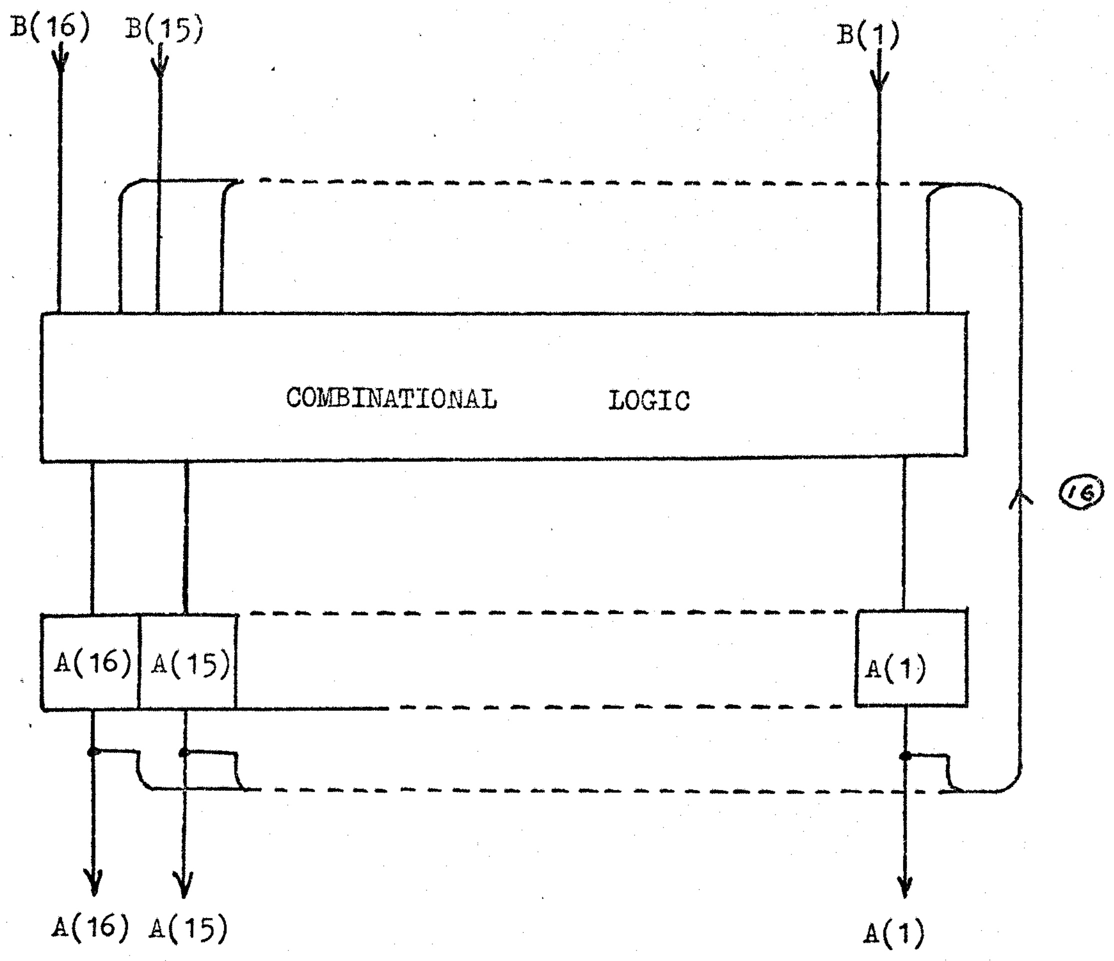


Figure 6. Conventional Structure for an Adder

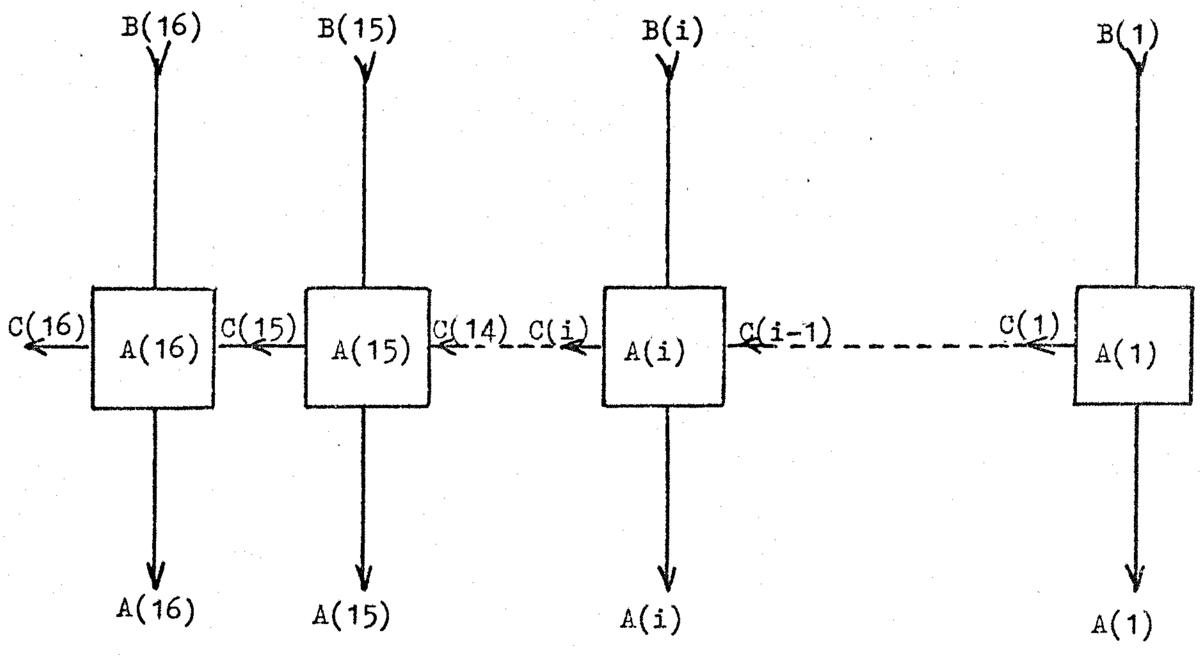


Figure 7. Iterative Structure for an Adder

individual cell of the machine.

Let the register transfer description be

$$A \text{ ADD } B \rightarrow A \tag{16}$$

Expanding this to a bit level, we get

$$A(i) \oplus B(i) \oplus C(i-1) \rightarrow A(i) \tag{17}$$

$$A(i).B(i) + C(i-1)(A(i) + B(i)) = C(i) \tag{18}$$

$i = 1,2,\ldots,16$, and $C(0) = 0$.

Generally each i'th cell will be described by register transfer and boolean equation statements similar to those in (17) and (18). The transfer statement describes the way the memory of the cell is modified, i.e. state variable behaviour, and the boolean equations define the outputs. Thus (a) the variable on the right hand side of all the transfer statements are taken to be state variables; consequently all the variables on the left hand side except those already present on the r.h.s. are inputs to the cell, and (b) the outputs are those defined by the boolean equations and also the state variables.

In the example there is only one state variable, $A(i)$; the inputs are $B(i)$ and $C(i-1)$, the latter being derived from the previous cell, and the outputs are $A(i)$ and $C(i)$. It follows that for a circuit to function satisfactorily all $C(i)$ must be propagated before the operation is completed. Gerace, therefore, imposed two conditions:

1) that the machine will not change its internal state during the absence of the clock pulse but the outputs may change according to the inputs and as defined by the boolean equations,

2) during the presence of the clock pulse the outputs and the state variables will remain unchanged and only when the pulse is removed that the change may be affected.

A next-state table may now be constructed. The right hand side of each transfer statement is replaced by a next-state variable, say Y_i, and any occurrence of the state variable of the l.h.s. is replaced by the corresponding present-state variable y_i. A table is constructed such that the rows correspond to all the combinations of the input variables and the entries are the values of the next-state variable defined by the transfer equations where the arrows are replaced by equal signs and with the above conditions. The output table is similarly constructed.

Applying this procedure to the example, we get,

y_i \ B(i) C(i-1)	Clock = 0				Clock = 1			
	00	01	11	10	00	01	11	10
0	0	0	0	0	0	1	0	1
1	1	1	1	1	1	0	1	0

Y_i

Table 1. Next-state table.

y_i \ B(i) C(i-1)	Clock = 0				Clock = 1			
	00	01	11	10	00	01	11	10
0	00	00	01	00	00	00	01	00
1	10	11	11	11	10	11	11	11

A(i) C(i)

Table 2. Output table.

These two tables give the behaviour of the individual cell in terms of state tables; however, it is not completely abstract as binary values have been assigned to the state variables. By a simple modification and combining the two tables together we obtain a flow table which gives the complete abstract behaviour of the i'th cell.

B(i) C(i-1)

present state	Clock = 0				Clock = 1			
	00	01	11	10	11	01	11	10
1	1/00	1/00	1/01	1/00	1/00	2/00	1/01	2/00
2	2/10	2/11	2/11	2/11	2/10	1/11	2/11	1/11

Next state, outputs A(i) C(i)

Table 3. The abstract state table.

The state behaviour during the presence of the clock pulse is considered unstable owing to the definition of the circuit, stable otherwise, and it is usual to circle the stable entries.

Note that the cell corresponding to i = 1 has only one input and its state table will be much simpler as shown in Table 4.

B(1)

present state	Clock = 0		Clock = 1	
	0	1	0	1
1	1/00	1/00	1/00	2/00
2	2/10	2/11	2/10	1/11

Next state, outputs A(1), C(1).

Table 4. Machine for i = 1.

3.7 Multiple Transfers

Generally a microprogram will consist of several transfers and therefore the system can usually be broken down into two parts,

1) the control unit, and 2) the operational part; the procedure detailed above can be extended to obtain the abstract behaviour of the operational part as well as the control unit.

A typical transfer statement in a set of microprograms would be written as

$$\begin{aligned} S_k: |X(1)|: \; & f_1 \to A(i), \quad f_2 \to B(i); \qquad S_k \to S_1 \\ |X(2)|: \; & f_3 \to A(i), \quad f_4 \to B(i); \qquad S_k \to S_m \qquad (19) \\ S_k+1: \; & \ldots \end{aligned}$$

where S's directly correspond to the state of the control part and govern the transfers in the operational part, X's are conditional expressions, A(i) and B(i) are the state variables in the operational part and the f's are boolean expressions.

The transfers refer to each element of the register arrays; however, it is quite possible and usual that a large number of the elements behave identically and some, especially the terminal elements, require separate description. Thus the first step in obtaining the abstract description from microprograms is to recognise the number of different machines that are described. The next step is to enumerate all the non-simultaneous transfers and identify each with a different label with a view that the control part will generate one signal per each different label and each set of simultaneous transfers will require only one signal.

In fact, the number of separate labels can be far smaller since the same transfer, but identified by a different S label, can be given the same label. The labelling process, therefore, is as follows:

i) separate out each transfer and the associated conditional expressions and the transfers in S,

ii) if two transfers are identical but have different S transfers associated with them they are given the same label,

iii) after (ii) all transfers not labelled and having common S behaviour are given the same label.

iv) provision has to be made to allow no transfers in the operational part.

If we examine the transfers we find that each set of transfers is associated with a state of the control unit and it also gives the transfer of state, i.e. the present state and the next state are defined; since the conditional expressions effectively modify the state transfers, these must be the inputs to the control unit and obviously the outputs of the control unit correspond to the labels obtained above. From this information, it is a fairly straightforward routine to obtain the state table for the control unit.

The state tables for the operational part are obtained in the same way as described in the last section, with the signals corresponding to the labels acting as further inputs.

An example is included in the appendix to illustrate the above procedure.

3.8 Extension to include Read Only Memory (ROM)

In large systems the number of different transfers is quite large and consequently the state table for the control unit of such a system is very complex. To reduce this complexity, a separate memory, which has a non-destructive read-out* and is at least an

* The discussion here is deliberately limited to read only type memories, however, it is accepted that a read-write memory may be successfully employed to achieve a better flexibility.

order faster than the main memory, is used to contain the details about transfers, tests and the sequencing in a coded form. These memories are usually called Read Only Memories (ROM's) or Read Only Stores (ROS's). The control unit behaviour with an ROM can, in many ways, be likened to the state tables as generated in the last section; however, there are some differences.

In a state table, it is possible to have many next states for a present state; whereas when using an ROM it is usual to have only two next-word addresses, an address having a direct correspondence to a state in the state table. The selection of the next address is done by checking the result of the test specified in the word, or if multiple tests are specified, then by collating the results of these tests, and extracting the true-false value from it. The next addresses are, therefore, sometimes called the true address and the false address. The more complex ROM systems have facilities for more alternative addresses.

To implement the state table for the control unit in terms of ROM, the state table is first reorganised to have only two columns, adding if necessary some dummy transfers to allow for multiple tests. The states thus obtained can be coded to give the addresses of the words in the ROM. The outputs defined in the state table are analysed and a coding generated such that the number of bits required in the coding is the smallest without losing the flexibility to indicate parallel transfers where necessary. The contents of the words in the ROM are then determined by this coding, the coding used to specify the tests and the next-word addresses derived from the state table.

3.9 Different types of ROM Implementations

In the last section, we considered a simple implementation of a state table describing a control unit by an ROM in which the number of alternate addresses were restricted to two, corresponding to the true and false results, only. The state table, however, is in an ideal form for manipulation for different types of implementations to allow more efficient utilization of the available resources.

One obvious parameter is the word length. Usually as the word length is increased the length of the microprogram reduces, assuming, of course, that sufficient facilities are available in the operational part to allow the necessary parallelism; but the associated cost, due to increased highway size and decoding networks, increases. In the converse case, the control becomes very much simpler but at the expense of speed measured in terms of number of control cycle per instruction. It is easy to see that a state table may be reorganised to reflect the two* types of implementation requirements.

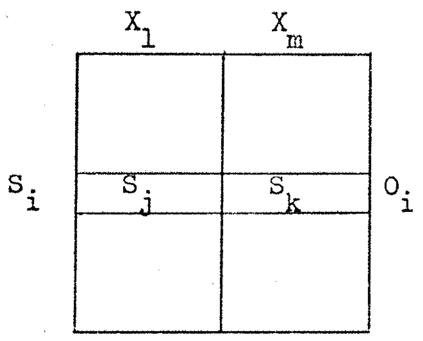
A different type of reorganisation was suggested by Gerace et al [55] in which the change in the state table format rather than the dimensions is utilized. Before exposing the method, however, let us first reconsider the structure of a microprogram description.

We have already noted before that a register transfer type description of a microprogram is a description of a set of control states each of which has associated with it, the operations to be executed when the state is reached, and a branch to the next state determined by the tests. A typical control state description in accordance with this is shown in (20).

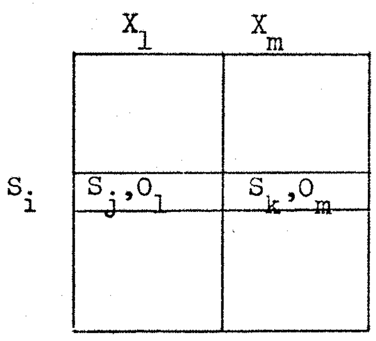
$$\begin{array}{l} S_i : O_i, \underline{\text{if}}\ X_1\ \underline{\text{goto}}\ S_j\ \underline{\text{else}}\ \underline{\text{if}}\ X_m\ \underline{\text{goto}}\ S_k \\ S_j : \end{array} \qquad (20)$$

where the S's refer to the control states, O_i are operations to be

* These two forms of implementations are commonly called horizontal microprograming and vertical microprograming respectively.

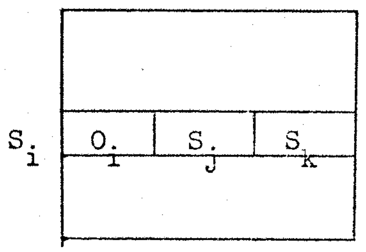


A Moore State Table
(a)

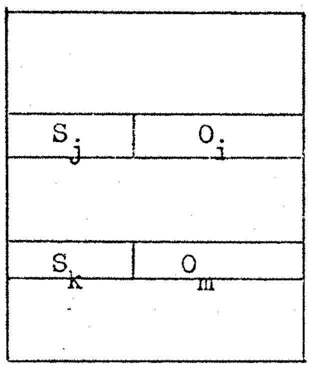


A Mealy State Table
(b)

Figure 8



A Moore type ROM
Corresponding to (a)



A Mealy type ROM
Corresponding to (b)

Figure 9

executed in State S_i and X's are the tests. In state table terms the expression (20) can be restated with S_i as the present state, O_i the outputs, X's as the inputs. Clearly then a description in a form similar to (20) translated into a state table of the Moore type [3, p.107].

We stated earlier that an ROM implementation of a state table may be achieved by creating an image of each row of the state table into a word (or a set of words) into the ROM. For convenience let us call the ROM implementation of a Moore state table as a Moore type ROM. Each word in a Moore ROM must contain the information regarding the operations and branching; therefore the number of conditions tested in a single ROM cycle must be kept down to limit the size of ROM words*. Another important factor associated with a Moore ROM is that the address selection for the next control word is performed by selecting <u>one</u> of the addresses specified by the ROM word. Thus the complexity of the address generation networks increases with the number of alternate addresses**.

A microprogram description may also be written such that the operations performed in a control state are not only functions of the control states but also of the results of tests as, for example, shown in (21)

$$S_i : \underline{\text{if}}\ X_1\ \underline{\text{then}}\ \underline{\text{do}}\ O_1,\ \text{goto}\ S_j;$$
$$\underline{\text{else}}\ \underline{\text{if}}\ X_m\ \underline{\text{then}}\ \underline{\text{do}}\ O_m,\ \text{goto}\ S_k; \qquad (21)$$

where the symbols have similar meanings as before.

---

* The word length may be kept down by restricting the tests to two as suggested in the last section. However, a number of dummy transfers may have to be introduced to effect multiple tests resulting in inefficient usage of memory and a reduced computational speed. The argument here is more concerned with the way the branch information is stored.

** Addressing relative to the present control word address or an address specified by a base register is often employed to reduce the inputs to the address generation networks. Nevertheless, the statement above is still valid.

The state table derived from this type of description may be easily seen to be of a Mealy type [3, p.107]. Now each entry in a Mealy state table figure specifies the outputs (operations to be done) and the next state (address of the next control word). Gerace suggested that this duple may be coded into a single ROM word thereby creating a word image for every entry in the Mealey state table. Obviously then the number of conditions to be tested is not restricted by the length* of the ROM word but only contributes to the complexity of the address generation network. Since the generation network only handles one address data its complexity in general may be shown to be less than the network in an equivalent Moore type of realisation.

Cadden [37] has shown that every Moore state table can be converted into an equivalent Mealy state table and vice versa, and that the number of internal states (rows) in the Moore state table is equal to the number of different pair entries (next state, output) in a Mealy state table. Thus a Moore ROM can always be converted to a Mealy ROM, i.e. an ROM implementation of a Mealy state table, such that the number of words in both is the same. A word in a Moore ROM, however, is longer than in the equivalent Mealey ROM for reasons already discussed. A Mealy relisation, therefore, is to be favoured giving a smaller memory, requiring less complex supporting networks and a higher computational speed.

3.10 Microprogram Transformations

It is usual to consider a microprogrammed system to be characterised by a Glushkov model [56], Figure 1, consisting of an

* This is not strictly true since often the conditions to be tested are specified as a part of a microinstruction, thus reducing the number of inputs to the state table, but increasing the length of the ROM word. The proposition here, however, is valid if this type of coding is not used.

operational part which contains the register structure and a control part, which is a finite state machine controlling the operational part. It is, of course, theoretically possible to merge the two parts and design the system as a single finite state machine; but for practical reasons a division must be made. The position of the dividing line, however, is questionable and usually is set by the designer through his experience, intuition, and the knowledge of available resources. Obviously, in doing so, the designer will experiment with different structures and weigh the relative advantages before deciding upon the final structure.

Berndt [35] suggested a concept of status level diagrams, which he described as functional microprogramming, to help this.The diagrams depict the control states and the sequencing in a diagrammatic form rather like state diagrams. The operations associated with each 'state' will obviously depend upon the resources employed in the operational part and the timing.

An approach similar to this was also used by Franke and Mergler [50] to develop a state table-like description of the control system. This state table description and the status level diagram both provide an overall functional description of the control section which can then be manipulated, say, to merge common control states or to split the states. The resources in the operational part can also be re-examined with regard to the effect on the status level diagrams. The manipulations indicated here are commonly called microprogram transformations.

A formal presentation of microprogram transoformations was made by Stabler [85]. He suggested five goals to achieve this which are as follows.

1) Remove a register from the operational part and adjust the microprograms to allow for this. The latter amounts to adding a register (or an image of it) to the control part.

2) This is the converse of 1, and adds a register to the operational part.

These two goals achieve the shifting of the dividing line between the operational part and the control part.

3) The resources in the operational part can be modified to allow two or more operations to occur in parallel. The microprogram control is then modified to produce one signal for the parallel operations instead of the individual ones before the transformations. Conversely,

4) Split the parallel operations into serial operations thereby reducing the complexity of the operational part.

3 and 4 clearly indicate that a speed/resource trade off is possible. Finally,

5) Reduce the input variables to the control unit and modify the two parts accordingly to preserve the overall behaviour.

The last transformation is particularly important where cable and highway sizes between control part and operational part as well as the ROM size are important. A common application of this was described earlier where the conditions of the operational part to be tested are specified in the microinstruction in a coded form, and the results are returned on a relatively few lines. Obviously decoding delays are introduced causing a loss of performance.

The goals 3 and 4 directly contribute to the number of ROM cycles required to execute a set of instructions which can be related to the execution time. However, an important way of achieving an improvement in the execution time is to overlap ROM cycles and the executions in the operational part. This method is widely used in present day computers and is of particular importance in large computers. Stabler does not deal with this aspect of micro-programming.

3.11 Structure Descriptive Languages

The languages so far considered impose a certain structure on the system and describe its operation in terms of algorithms. This description may be given with varying degrees of conciseness and manipulated using a computer to produce the necessary amount of expansion and/or minimisation, and also generate the logic for the control part governing the system. Thus these languages satisfy certain design* requirements. The important point to note is that a part of the structure is defined and the remaining generated through computer assistance.

There exists another class of languages which are more closely oriented towards describing an already completed design. Using these languages a system may be described in terms of its structure, i.e., the implementational detail, or in terms of its functions, i.e., describe the "what" rather than "how" of the system. The application of these languages for design is rather limited, nevertheless they have a wide range of uses, such as, documentation, input to implementation programs in a design automation suite, structural and functional simulation and fault diagnosis**.

---

* The term design is used here to mean logic (or program) design rather than implementation design.

** See Appendix IV.

Most digital system manufacturers employ design automation techniques in production of the systems for some or all of the application suggested above; and there must be of necessity at least one structure descriptive language associated with each. However, there is very little published material regarding either these languages or the suites; and it is not possible to gauge the proliferation of versions, differing maybe only slightly, of such languages. Examples of typical commercially used languages are in the LOGSIM system developed by the Marconi Company and the RADDS system [100, 107] developed by the Raytheon Company. The usual method of description is to describe each gate or an available primitive module in terms of its inputs, outputs and attributes (e.g. delay)(cf Okada & Motooka [75]). It is also possible to create new blocks out of the available primitives and treat these blocks as primitives, i.e. nesting is possible. Despite this facility however, the description of a large system tends to be very large and consequently the effort required for manually producing these, as is still the normal practice, is also large.

Stabler proposed a System Descriptive Language [106] which was basically on extension of Reed's Language [78]. The main additions were on Algol-procedure-type construct to describe a gating network and on Algol- iterative - type construct to handle iterative net- works - a common feature in digital computer logic.

A/

A serious omission is that neither primitive elemental delays nor explicit delays such as of delay elements and cable delays can be described which would be required in structural simulation. Nevertheless with the constructs suggested and by nesting the description as necessary, it is possible to describe computer logic in a very concise manner. A digital computer can be then employed to remove the linguistic intricacies and produce a structural description in a much more primitive form.

Bell and Newell [89, 90, 91] proposed a much more comprehensive method of describing a computer structure. Using their method, a computer system is described at two levels, namely

i) The PMS (Processor-Memory-Switch) Level and

ii) The ISP (Instruction-Set-Processing) Level.

At the PMS Level the organisation of the whole system is described in terms of its constituent (PMS Level) components and their attributes, including the types, throughput and size, in a diagrammatic form. The main components at PMS level are units such as

(a)	Memory, M	-	component which stores information
(b)	Link, L	-	component that transfer information between two components of a system, i.e. data highway
(c)	Control, K	-	a component that evokes an operation or set of operations in other components - effectively the control state of a system

(d)/

(d)	Switch, S	-	component to switch between links
(e)	Transducer, T	-	component that transforms one type of information into another, e.g. voltage levels into characters on paper, light input voltage levels
(f)	Data operation, D	-	the data manipulation part of a computing system
(g)	Processor, P	-	component to execute a (user) defined program

Each component in turn may be further qualified depending on its rate and application within the system, e.g. Pc to mean central processor, Ms to mean secondary memory.

At the ISP level the processor itself is detailed in terms of its ISP level constituents such as registers, memories, processor control states and data operations. The description is similar to that in Parnas's language in that the effect of the data operations rather than the step taken in achieving them are indicated. This level does not detail the logic structure of the system.

The method proposed by Bell and Newell is quite comprehensive at the PMS level for describing system configuration, particularly since it allows in a natural manner the scope of the language to increase to cover any future concepts. The two levels of description together provide a good means of documenting the architecture of a computing system [90].

It has been stated above that a description in ISP essentially describes the "what" rather than the "how" of/

of the system and in this respect is different from a design langauge specification. The basic approach for design, so far considered, involves extracting the structural information, abstract behaviour of the control part and determining the data and control paths - one of the main aims in this process being the minimization of resources.

Bell et al[92, 96, 103] argue that with the availability of circuit modules implemented in large scale integration (LSI) technology this constraint may not be so relevant. They propose a concept of register transfer modules (RTM) which implement directly the operations evoked by the processor state, e.g. an arithmetic operation between memory bars and a processor register, so that for every different operation a different RTM is employed. Simultaneously, control type RTM's are employed to execute the boolean testing and branching involved <u>without</u> first going through the exercise of extracting the complete behavioural specification of the control part. Obviously as the numbers of different operations, and of different types of controls, increase, the number of different RTM's required increases. However, the approach has the elegance and neatness of being simple for a novice to understand and cutting the time and effort required to implement a design. This concept will certainly prove very useful in teaching.

In/

In SDL1, the language proposed by Gorman [94, 95] and in CDL1, the language proposed by Srinivasan [104, 105], the system may be described at four different levels. These are as follows.

(1) Behavioural - In a behavioural description a system is viewed as a black box with no knowledge of the internal structure and its behaviour is described entirely in terms of the inputs and outputs.

(2) Functional - The black box representing the system is segmented into major functional units such that a behavioural description of each unit is possible. This represents a coarse breakdown of the overall system. The functional description then describes the interconnection between the units and an algorithm in terms of these units to achieve the required behaviour.

(3) Structural - At this level, the authors suggest, the description should be sufficiently precise so that (at least conceptually) the design can be put together by using "off-the-shelf" components, which may be hardware or software.

(4) Implementational - A description of this level defines the method of implementing the system physically either in terms of actual gates and registers for hardware, or machine instructions for software.

The/

The statements within a description at a certain level are grouped together to give a hierarchical description where the hierarchy is determined by the scope of the facilities used in the statements. The scope however, is not limited to within the bounds of the hierarchy and may be extended to a higher level by explicit statements. This concept is slightly better than the global and local variable concept.

These languages contain a comprehensive set of facilities to allow variable interpretation of any entity which is a very useful facility when large systems are considered. The language also allows an extension of the syntax and modification of semantics.

The common syntax for all the levels is particularly useful in system modelling since a common simulator can be constructed to handle description at all levels. As the design progresses it is only necessary to change description to a different level within the same language.

In general however, structure descriptive languages serve an intermediate, and a very useful stage between the design process and the implementation. Their scope, especially when defining the control part of a system, tends to be restricted. Our basic aim to study the possibilities of describing a system without, as far as possible, any structural constraints. To this end, structure descriptive languages are of an indirect interest only.

3.12 Conclusions

Most large digital systems can be regarded as instruction executing systems and consisting of an operational part which contains the data storage facilities, i.e. registers, and the data manipulation, i.e. register transfer, facilities, and a control part which provides the necessary signals to activate the register transfers in a correct manner. We have noted in the discussions in this chapter, that the behaviour of the operational part can be described, in terms of microprograms, in a register transfer language and that it is possible to extract the behaviour of the control part from this description. However, the flexibility offered by the various languages to describe any complex modes at microprograms varies widely.

Earlier register transfer languages were simple and could be directly mapped and thus were good tools for analysis of already designed systems and for automation of implementation. They had their limitation such as, inability to indicate segmentation, multiple operations, mixed synchronous and asynchronous operations etc., and their timing notation was particularly poor. Roth's sequence chart analyser [79] expressed microprograms in a graphical manner which indicated timing and multiple operations, but owing to its graphical nature it is difficult to automate.

Further languages were developed to increase the flexibility and specification ability for which notational and operational conciseness was introduced by using complex/

complex operators and macro calls etc. Some of these languages were based on the structure of existing programming languages, e.g. Metze and Seshn's language based on Fortran [74] and Chu's CDL [39 - 42] and Cassandre [32, 33, 58] based on Algol. Segmentation facilities were introduced in DDL by Duley & Dietmeyer [46, 47] and in Cassandre.

The Iverson notation [64 - 66] provides a means of describing the logical functions of a system at various levels of detail including elemental bit levels, independent of the machine structure and in an algorithmic manner lending itself to a good interpretation in terms of hardware realization. However, the designer usually defines the system in terms of functional blocks first before attempting an algorithmic solution of the problem. The Iverson notation unfortunately, does not have a sufficiently high level of functional descriptive ability. Secondly, at the algorithmic level the language does not contain adequate facilities to express control, particularly timing.

Since all the languages use a predefined register structure, the automatic part is still limited to deriving the controlling circuity and the combinational logic driving the register structure. Gerace [54] described a method by which the register structure implied in the register transfer description may be reformulated into an iteratively-connected-machine structure and obtain a formal abstract definition for each. A more pertinent application of register transfer language/

language description is however, in producing a ROM implementation of the control part. ROM implementations are somewhat more flexible in that it is possible to change the characteristics of a given operational part relatively easily by changing the ROM part of the control part and thus by using, say, plug in ROM modules an effectively different system may be obtained. Gerace, et al [55] have described methods of different ROM implementation and minimisation.

Another interesting, and potentially very useful result noted [85] was that the dividing line between the operational part and the control part is somewhat arbitrary and that certain rules can be applied to shift this line one way or the other. This also exemplifies the artificiality of dividing a system into two arbitrary parts. It should be possible to view a digital system from an overall system view point and describe its behaviour in some manner that is independent of the internal structure and then either algorithmically or via some interaction with the designer evolve the necessary structure.

Finally, the usefulness of a register transfer type language for documentation of system cannot be emphasized too greatly. Its value is further enhanced if a methodology is developed by which a system can be detailed, at the various levels necessary, as it progresses through the design stage. Such a methodology [90] also allows a suitable comparison of various systems to be made in uniform manner.

4. Methods based on Switching Theory and Information Theory

4.1 Introduction

One of the advantages of using switching theory in logic network synthesis is that it provides algorithmic, and hence programmable, techniques for producing logic designs from input-output specifications. These programs may be then used by even a relatively inexperienced designer to produce complex, error free logic designs, providing of course the specifications of the network are input to the program in suitable forms. The latter constraint however, represents a serious disadvantage in that large amounts of data corresponding to truth tables, state tables or flow tables have to be input and obviously this, apart from being tedious, could lead to errors which may be hard to detect. The specifications therefore, have to be input in a way that the chore of the tables may be relegated to the computer.

4.2 Carroll and Mott's Method

An approach to this was suggested by Carroll and Mott [93] in which the inputs and outputs are considered to be related by some continuous function(s) which may be input directly into the program. Carroll and Mott distinguished between 3 types of logic networks. If n is the number of inputs and m is the number of outputs, then these three types are a) those having $n=1, m\geq 1$ b) those with $n\geq 1$, $m\geq 1$ and c) where $n\geq 1$ and $m=1$. A single input network as in type a is a special case in that it represents a counter in which the input itself is the clock input and the outputs are coded in the required form. The input-output relation in such a case is cyclic repeating after p pulses where

$$0 \leq p \leq 2^m - 1.$$

The other two cases are more general froms of logic networks and could represent combinational or sequential networks; the concept of simple input-output functional relations however, is only applicable to combinational networks. Nevertheless if these are known, the production of truth tables is fairly straightforward. A difficulty arises when these functions are to be determined, especially if they are limited to be numerical, as implied by Carroll and Mott, and in many cases it is not possible to determine them. One way to overcome this is to extend the types of functions that may be specified and to include algorithmic descriptions, particularly where iterative relations are involved. Another useful addition is to complement the relational description with the input-output pairs where necessary.

4.3 Smith and Tracy's Method

The specification of a sequential network behaviour introduces another dimension to the problem, i.e. that of time dependence. The method suggested above cannot be used for sequential networks except in the special case of counters. Smith and Tracy [102] proposed a method whereby the behaviour of asynchronous networks may be specified in a short form and converted into normal flow tables.

The method relies on being able to specify the output responses to a series of input sequences as, for example, in pattern recognisers or counters. The series may contain several individual sequences; and the ordering, either to create loops (as in counters) or to indicate branching (tests), is shown by attaching notes goto and follows. The sequences themselves may be defined in terms of either all inputs or a subset of inputs.
As an illustration, consider a network with two inputs a, b and two outputs y, z. y becomes equal to 1 if a=1 and b follows the sequence 10 and providing that a has followed the sequence 010 immediately prior to this. z becomes equal to 1 under the same conditions except that b follows the sequence 01.
The output response type description for the above problem is shown in figure 1.

In the translation* of this type of description to a normal flow table,

---

* Although the steps described here are taken directly from Smith and Tracy's paper, they are slightly modified in the illustration by introducing the restriction that only one input variable may change in a transition.

an intermediate flow table, called a module flow table (MFT) is first generated. This flow table indicates the ordering of the sequences and in effect is a mapping of the goto and follows notes. Next, a preliminary flow table is generated for each sequence such that firstly stable state entries are made where input-output response pairs are specified and then unstable state entries are made corresponding to the next stable states. No unstable entries are made at the tail of the sequences, i.e. at the end of the sequences. The individual flow tables are then concatenated together using the information contained in the MFT and adding unstable entries at the tails of the individual flow tables to correspond to the next stable states. These steps are illustrated in figures 2,3, and 4. The final flow table, obtained algorithmically, is shown in figure 4 and may be compared with the state diagram obtained directly from the initial specification and the corresponding flow table shown in figures 5 and 6 respectively.

The procedure illustrates some interesting points. Firstly the description is almost a state diagram type description but it is in a form which is much more allied to the approach likely to be taken by a designer who is unfamiliar with switching theory methods. Secondly the method does not require a full specification and can be completed in stages; and it seems ideally suited for generating a flow table in an interactive mode. Finally, although the method has been illustrated only with an asynchronous design example, it may be possible to generalize it to include synchronous designs.

		inputs		outputs		notes
		a	b	y	z	
seq 1	1	0	-	0	0	
	2	1	-	0	0	
seq 2	3	0	-	0	0	follows 2
seq 3	4	1	1	0	0	follows 3
seq 4	5	1	0	1	0	follows 4, goto 1
seq 5	6	1	0	0	0	follows 3
seq 6	7	1	1	0	1	follows 6, goto 1
seq 7	8	0	-	0	0	follows 4,6, goto 3
seq 8	9	1	-	0	0	follows 5,7, goto 1

Figure 1. Input-output response specification

	inputs			
seq. number	00	01	11	10
1	-,2	-,2		
2			3	5
3	7	7		4
4	1	1	8	8
5	7	7	6	
6	1	1		8
7			2	2
8	1	1		

The dash indicates the starting inputs of the sequence

Figure 2. MFT corresponding to the example in figure 1

inputs = ab

00	01	11	10	
①/00	①/00	2	2	sequence 1
		②/00	②/00	
③/00	③/00			sequence 2
		④/00		sequence 3
			⑤/10	sequence 4
			⑥/00	sequence 5
		⑦/01		sequence 6
⑧/00	⑧/00			sequence 7
		⑨/00	⑨/00	sequence 8

Figure 3. Individual flow tables corresponding to the example in figure 1.

inputs = ab

00	01	11	10
①/00	①/00	2	2
3	3	②/00	②/00
③/00	③/00	4	6
	8	④/00	5
1		9	⑤/10
8		7	⑥/00
	1	⑦/01	9
⑧/00	⑧/00	4	6
1	1	⑨/00	⑨/00

Figure 4. Concatenated flow table

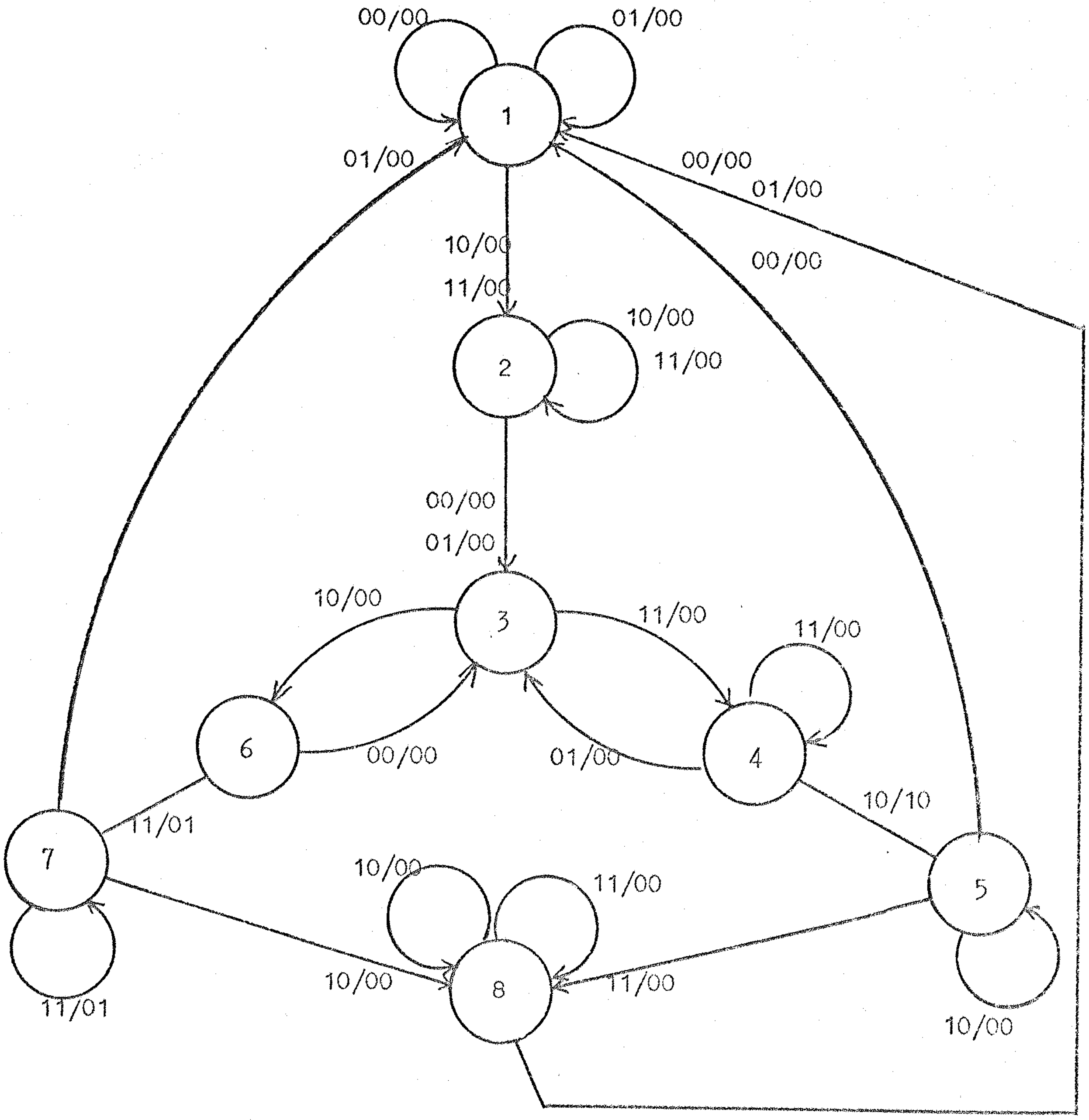


Figure 5. State Diagram for the Network

inputs = ab

00	01	11	10
①/00	①/00	2	2
3	3	②/00	②/00
③/00	③/00	4	6
	3	④/00	5
1		8	⑤/10
3		7	⑥/00
	1	⑦/01	8
1	1	⑧/00	⑧/00

Figure 6. Flow table corresponding to figure 5.

4.4 Petri Nets

The importance of considering the behaviour of the overall system rather than segmenting it prematurely into hardware and software was also emphasized by Holt et al [97,98] in their work on Petri Nets. Using these nets, which were first conceptualized by C.A.Petri as transition nets, it is possible to indicate the behaviour of a system in terms of information flow through hardware processes or software processes in a precise and concise manner. Thus these nets provide a method for describing a system in a unified way and is a significant step towards the required goal.

The basic unit of information flow in a Petri Net is an event. A system can be described by a set of events joined together in a loop,allowing repetition of events. The term event therefore, is generally taken to mean a repeatable event and an individual repetition of an event is called an occurrence.

An event in a Petri Net is represented as a transition and is depicted by a bar with a suitable number attached to it so as to distinguish it. The transitions are connected together by arrows via places or conditions which are depicted by circles. The entries in the places specify the conditions necessary for the transitions.

The arrows establish the relations between the transitions and places: an arrow from a place to a transition means that the place is an input condition for the transition and an arrow from a transition to a place indicates that the transition generates the condition.

A simple example of a repeatable event is a computer in a user environment. Initially let the computer (C) be in an idle state (CI). A user (U) accesses the computer via a teletype unit (T) and inputs a program (P). The computer the computes the program (CP) while the user is waiting at the teletype unit (UTW). When the computer has finished the computation (CP') the results are passed on to the user (UTR) who leaves the computer in an

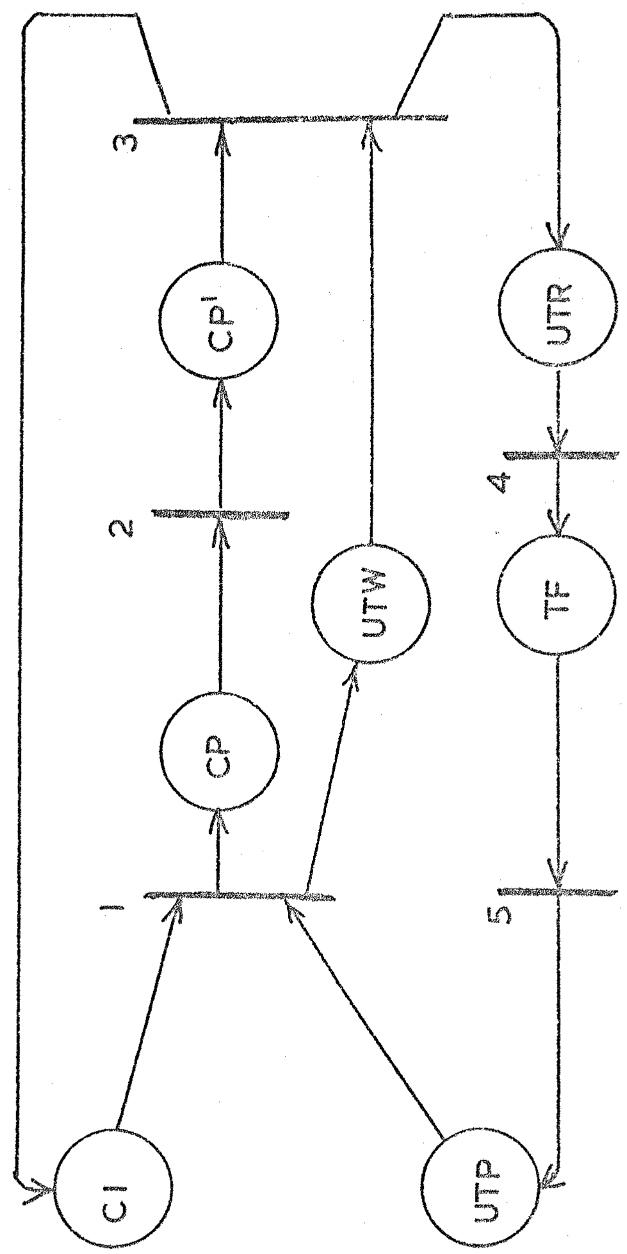


Figure 7. A Petri Net

idle state (CI) and the teletype unit free (TF).

The Petri Net for the above is shown in figure 7. Event 1 in this diagram shows that it will take place only if the two conditions CI and UTP are satisfied. As soon as the event takes place the conditions CP and UTW are generated. The conditions for transition 2 are thereby satisfied and the condition CP' is generated. This in turn, along with UTW, allows transition 3 to take place.

The Petri Net considered here clearly indicates the information flow through the system in a concise and precise way. These nets can also handle concurrent or parallel and independent events particularly well. Thus they can be used to describe the behaviour of a variety of systems with a varying amount of detail.

The history of a system can be recorded by performing a simulation on the Petri Net. Conventionally this would be done as a record of states and the associated conditions generated by them. However, this requires that every distinguishable state be recorded as a separate entry. Holt introduced a notion of occurrence graphs which illustrate the simulation in a graphical way and are able to handle concurrent events more easily than by the conventional approach. An occurrence graph for the Petri Net considered here is shown in figure 8, where the nodes indicate the transitions and the arcs indicate when the conditions specified by the labels associated with the arcs are true.

Petri nets offer some exciting applications. Firstly, concurrency can be relatively easily and concisely depicted. They can be used to describe the input-output behaviour completely, and entirely in terms of its environment without imposing any constraints of implementation technology. The latter application is particularly useful in design. For example, an algorithm may be depicted using Petri Nets containing as much concurrency as is allowed by the constraints due to environment. Now the designer can choose

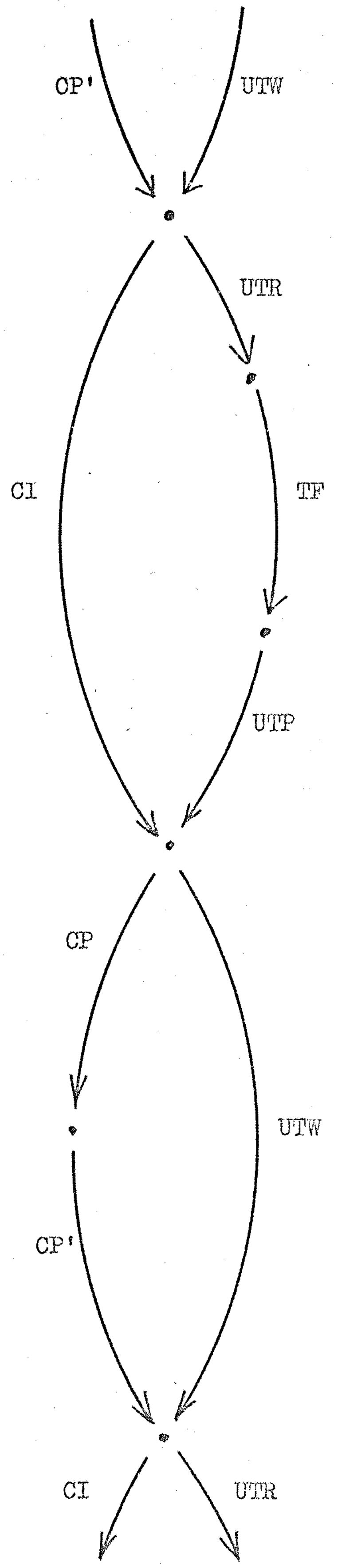


Figure 8. An Occurrence Graph

a particular implementation such that "parts" of the algorithm can correspond directly with the "parts" of implementation. Conversely the designer can modify the algorithm with the constraints of an existing implementation in mind, to best use the implementation.

4.5 Conclusions

Switching theory provides us with methods to describe the behaviour , i.e. the input-output mapping, of a system in a precise manner. Unfortunately, the amount of data required to do so and the data generated in the subsequent phases of design tend to be very large indeed. This does not however, mean that switching theory should be ignored for practical design and in fact switching theory is a very effective means of producing error free designs.

The methods of generating truth tables or state tables described here, go some way towards bridging the gap between the concepts specified by switching theory and practical methods likely to be adopted by a designer. However, there is a strong reluctance among designers to assume switching theory techniques in their design processes and a considerable support is still necessary before these techniques are in general use.

One of the drawbacks in switching theory is that at present it is somewhat inadequate to handle large systems with parallel processing. Petri nets however, handle such systems neatly and also offer some additional useful applications, such as optimization and simulation. The Petri nets seem to offer real potential towards a unified method of system behaviour description and system architecture design.

5. AN APPROACH TO COMPUTER AIDED LOGIC DESIGN

5.1 Development

It is clear from the foregoing discussion that the logic designer acting in a computer aid environment, and especially in an interactive mode, has a language problem. Many languages have been devised and utilised to a varying degree of success. It is also clear that the main drawback comes from the correlation, or the lack of it, between the language and the designer's natural methodology, and also the "design" aspect rather than just the simulation capability of the language. To surmount these drawbacks and to devise a new language, it would be helpful to examine what we are trying to design and how we, as human designers, tackle the problem intuitively.

The system under consideration is a digital processing system, by which we mean that the system will accept information on lines which carry one of only two values and that after processing produce outputs on similar lines. Typical examples of such systems are traffic light controllers and digital computers. It is also envisaged that these systems will, in general, process the inputs in more than one way, i.e. they will have a certain instruction repertoire and that the required instruction would be selected by an external input, such as a program. This definition allows the inclusion a general class of digital processors, since if only one function/

function is executed, the repertoire will include only one instruction and the external input would be, say, an on/off switch.

5.2 The Intuitive Approach

To understand the steps which a designer is likely to follow, it will be useful to consider an example. The example we choose is a simple one, yet adequate to illustrate the steps taken. Two numbers each seven bits long are coded with Hamming distance code * and are accepted in a serial mode. Their parity is checked and a correction is applied if necessary. If the first number is greater than the second then the two numbers are multiplied otherwise they are added together. Finally, the output is correctly coded with Hamming code and put on an output line, again serially.

This itself defines the first step in any design: that of a description of what the overall system is expected to do. There is no mention as to how the parity is checked or how the multiplication or addition is achieved, or for that matter, whether the operation within the system is conducted serially or in parallel. The abstraction we can derive from the above description is that it is a black box with one input line carrying the input data, another one to validate (synchronize) that data and one output line for output data.

* Appendix III

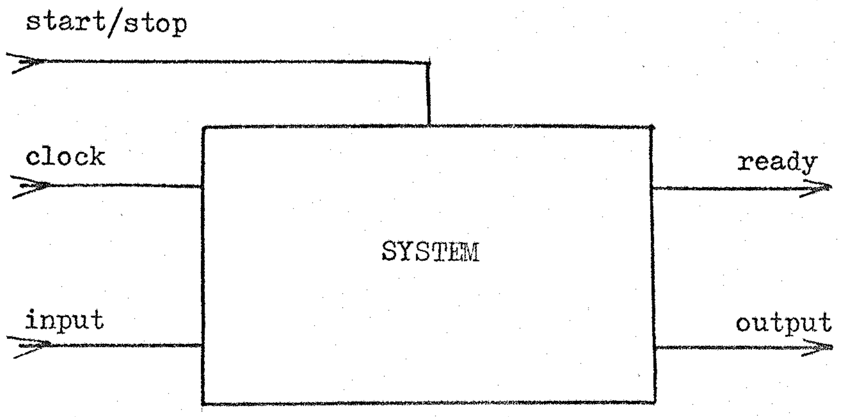


Figure 1. A Small System

Although the clock was not explicitly expressed we deduce that it is necessary. We would also need to provide another output line to indicate when the output is ready and obviously a start/stop line. Since the output is also serial, we would need to know whether a clock line for this has to generate or if the input clock is running continuously and consequently can be used for the output. Let us assume the latter.

It is clear that at this level we are only concerned with outlining our system in terms of the input and the output and the system behaviour is described. We call this type of description a 'behavioral description'. Ideally, we would like to input just this much information into an automation programme and let the design be evolved with respect to some pre-defined cost-effective measures which the designer specifies. But it would be näive to attempt to obtain a solution, let alone an optimal solution.

The/

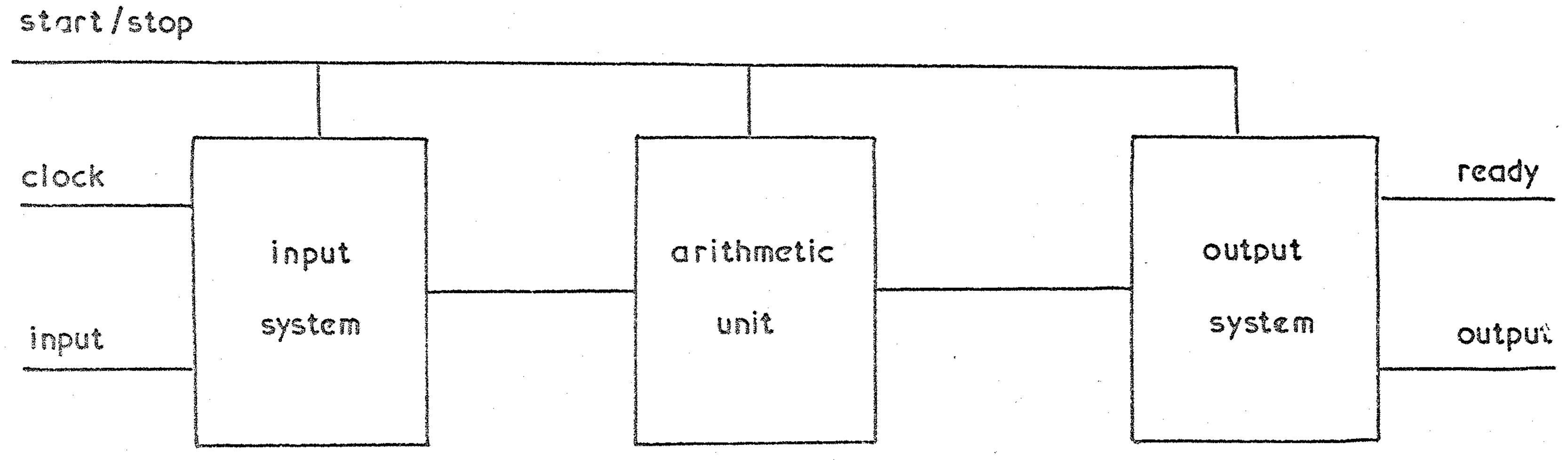


Figure 2 Functional breakdown of the system

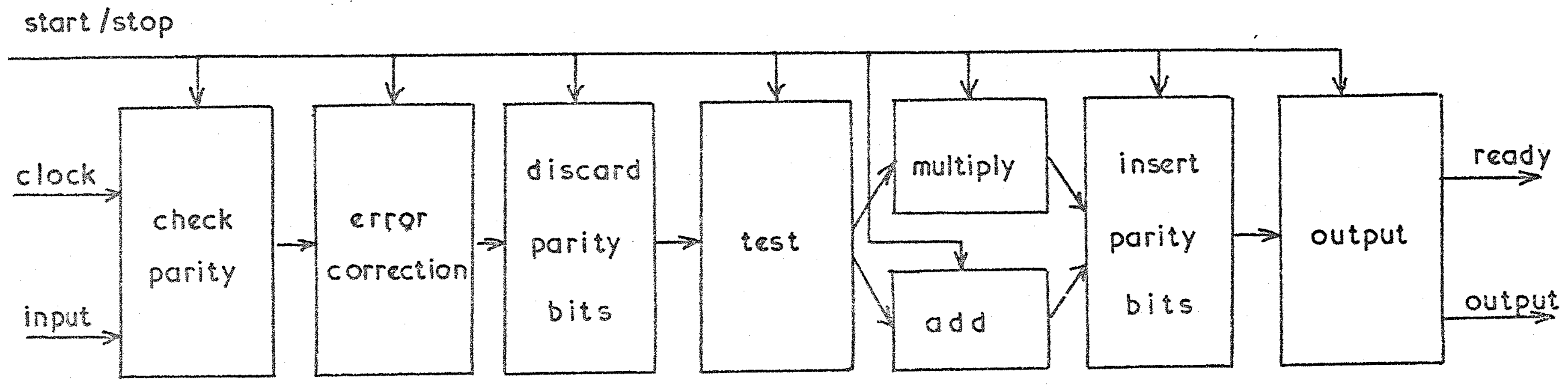


Figure 3 Functional breakdown of the system

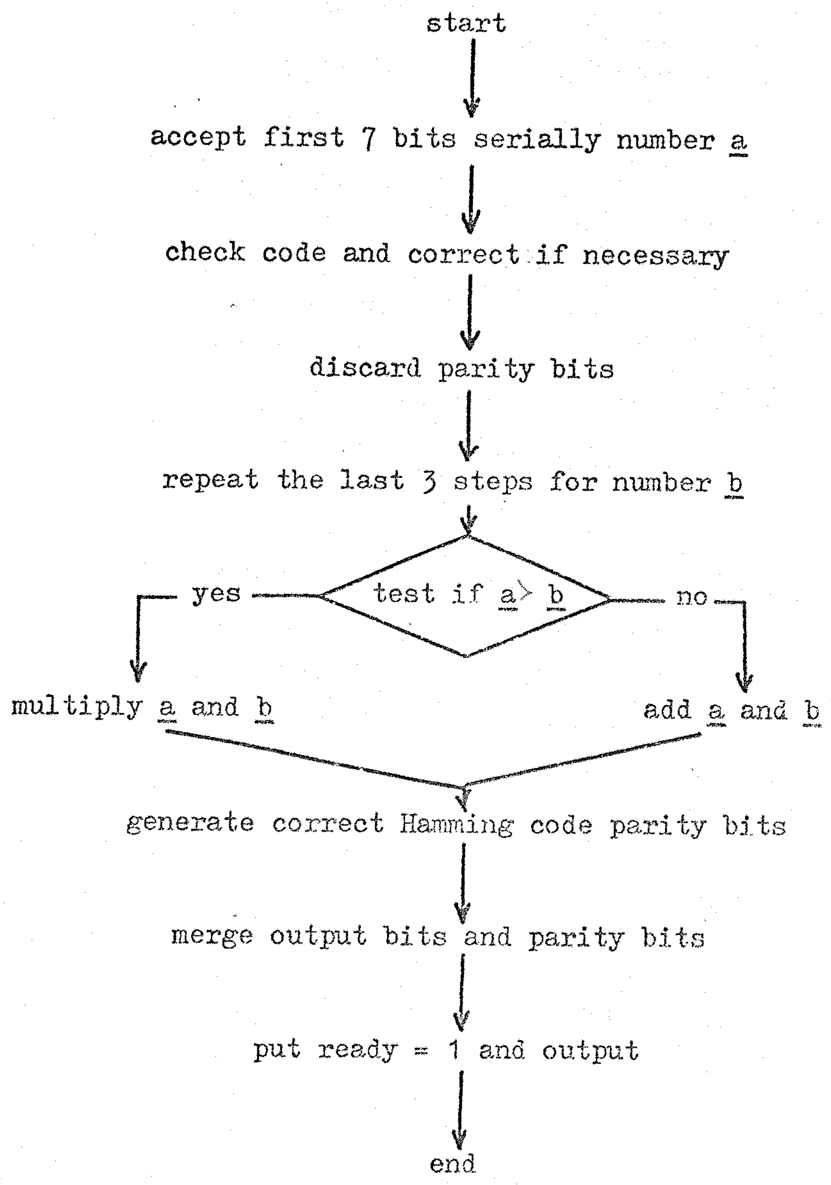

start
accept first 7 bits serially number a
check code and correct if necessary
discard parity bits
repeat the last 3 steps for number b
test if a > b
yes
no
multiply a and b
add a and b
generate correct Hamming code parity bits
merge output bits and parity bits
put ready = 1 and output
end

Figure 4. Functional algorithm

The next step is to break down the black box, called system, into sub-systems, each designated certain functional capabilities. This would start on a coarse breakdown extending to a finer detail as necessary. The functional breakdown for our system is shown in figures 2 and 3. The description of the architecture based on this type of breakdown is called 'functional description' of the system. We would also draw up a flow diagram of how we utilise this architecture and this is shown in figure 4.

The flow diagram gives us the sequence in which each function has to be performed. We still cannot translate this information directly into hardware or software routines until the how of each function is specified. However, since the flow diagram is not related to any machine structure, it is still abstract and independent of the final machine and acts as an overall reference.

The next step is to detail each function in an algorithmic manner. The human designer at this stage, owing to his experience and intuition, may resort to hardware blocks and express the algorithms with these hardware constraints. However, we feel that this is "jumping the gun", as this process may lead to quick hardware realisation but will not allow any logical process of overall minimization. For example, the designer may allocate J-K flip-flops for memory elements to minimize hazards due to asynchronous signals but the overall system may be such that only R-S flip-flops, which are cheaper, may be adequate. Another example/

example is that the designer may allocate separate registers and a parallel adder whereas a serial circuitry may be sufficient.

It can be safely said that a designer will normally derive a flow diagram similar to the one in figure 4, from the description of the system in a natural language which is sufficiently formal for the logic designer yet it is quite comprehensible to the members of other disciplines. Since we wish to devise a language that can be used as a general purpose design language, we feel that at the highest level the language should incorporate information of this type. It must be remembered, however, that the statements in the flow diagram indicate the flow of data and the operations performed upon it, and that the logical operations for each statement have still to be defined. Therefore, it will be useful to think of these steps as macro functions and each of these is detailed in a logic design language.

Returning to the example, we consider the required translation of the functional macros. We have established a data flow through the functional boxes, the data being a collection of strings or bit patterns. The input data may be sustained long enough for the functional boxes to perform the necessary operations; on the other hand, especially in the caseof serial transfers, it may not be present long enough and the whole of the data may require "memorizing".

The Hamming code used here has 3 check bits and 4 information bits and since the validity of the number cannot/

cannot be checked until all the 7 bits are present, all these will have to be memorized. Let this function be denoted by a register and since seven bits of each number have to be registered and that they appear serially, a counter has to be introduced. The functional breakdown, then would be as follows.

1. set counter to 0
2. increment counter by 1
3. if the clock pulse is present then register input into a vector, the position being determined by the value of the counter.
4. if the counter has a value 7 then go to 5 else go to 2.
5. ...

This is an algorithmic description of the functional breakdown and the Iverson notation is most useful here. The algorithm is re-written below using this notation, where the counter is k, the clock is c, and the first seven bit number is $\underline{a}$.

```
1.  $k \leftarrow 0$
2.  $k \leftarrow k + 1$
3.  $c:0, (=) \rightarrow 3; \quad (\neq)\ \underline{a}_k \leftarrow$ input;
4.  $k:7, (=) \rightarrow 5; \quad (\neq) \rightarrow 2;$
5.  ...
```

The error checking and correcting steps are expressed algorithmically as follows.

```
5.  $e1 \leftarrow \not\vee/((7)\top 85)/\underline{a}$        ($\not\vee$ = exclusive or)
6.  $e2 \leftarrow \not\vee/((7)\top 51)/\underline{a}$
7.  $e4 \leftarrow \not\vee/((7)\top 15)/\underline{a}$
8./
```

```
8.  n ← ⊥ e4,e2,e1
9.  n:0,   (=) → 10;  (≠) $\underline{a}_n$ ← $\overline{\underline{a}_n}$
10. ...
```

e1,e2 and e4 are three scalar quantities corresponding to the three error bits. Statement 5 is interpreted as: mask the vector $\underline{a}$ by a binary pattern whose value is 85, i.e. select the odd bits of $\underline{a}$ and if the sum is odd then there is an error; similar interpretation is used on statements 6 and 7. The statements 8 and 9 define a single error correction. The masking patterns can be generated using special Iverson operators and 5,6 and 7 can be re-written as

```
5.  e1 ← ≠/(2 | $\underline{i}^1(7)$)/$\underline{a}$
6.  e2 ← ≠/(4 | $\underline{i}^1(7)$)/$\underline{a}$
7.  e4 ← ≠/($\underline{i}^1(7)$ > 3)/$\underline{a}$
```

Without going through the remaining steps it is easy to see the general format of the algorithmic description and that it can be similarly applied to the remaining functions of the machine.

The hierarchy in the description is already apparent as the functional macros are at a higher level than the algorithmic description, and it can be extended so that each operation is further simplified to a lower level and so on. In an intuitive approach, the hierarchy is extended until the description has almost a one to one correspondence with some structural elements. Thus the vectors are immediately translated into registers, the steps into timing cycles and the remaining operations performed by clever/

clever manipulation of interconnecting logic to minimise delays, elements and, in the case of parallel processing, hazards. This process can be largely automated and has indeed been demonstrated by Friedman et al [53]. The input to their program, ALERT, is in the form of Iverson statements and the outputs define the excitation equations for the flip flops, and these are subsequently processed to obtain the logic diagrams, wiring diagrams etc.

It may be recognised that the Friedman approach is to assign hardware blocks to achieve the various operations; however, this is the same as in the case of register transfer languages and the only functions that the computer provides is to assign these blocks automatically, and to remove redundancies. We feel that a better approach is to derive the behaviour in terms of, say, truth tables and state tables from the functional description and then process this by a logic assignment programme.

There are two ways of obtaining this information, the first is to use the functional description and converting this directly to state tables by a method similar to Gerace's [54] and the second is to use the allocations obtained by a programme similar to Friedman's and then from the excitation equations obtain the state tables.

Clearly then, a library of available and usable physical objects has to be created and for this a flexible and comprehensive declaration facility is needed. As new objects/

objects become available they should be readily added to this library without affecting either the flow or the structure of the language using this library and hence the declaration facility should be expandable naturally. It should include sufficient information to determine its applicability completely. For example, a software routine will require in its declaration, its name, input and output parameters, how it is called in the main programme, its size and speed. Further information which may be necessary is how the routine functions and any illegalities either in operation or interaction with another routine. Another useful parameter would be a cost figure, which is particularly useful in cases where the designer wishes to trade cost with speed or vice versa.

5.3 The Computer Aided Approach

From the previous discussion we deduce that the designer needs to specify a system at three different levels. At the first, the system is defined entirely in terms of its input and output behaviour, i.e. the specification at this level describes the system as a whole without any indication as to its internal structure. At the second level, the system is decomposed into several sub systems, each as which may be defined

(a) in terms of its input-output behaviour, or

(b) by algorithm specifying its functions

Finally at the third level the designer may specify a structural detail and the operations constrained by this structure.

To apply any of the minimization programs, the data in the computer must be obtained either in terms of truth tables or the equivalent forms thereof, or in terms of state tables or their equivalent forms. If the designer inputs the data in terms of behavioural specification then the subsequent manipulation is straightforward. However, the data to be input becomes enormous and a short form method must be considered. Such a method for inputting behavioural specification for combinational networks is proposed later.

The structural definitions can be given by register transfer languages or in a form of Iverson notation [53]. This type of description is very useful for analysis work and, in the design process, can be used to generate the boolean equations for the logic interconnecting the registers. The boolean equations can be manipulated to minimize the combinational logic; however, the registers themselves are not minimized, mainly because no formal methods yet exist to minimize sequential logic without returning to a state-table-type specification. Furthermore, a large part of the design is already complete before use of computers is sought. A method to automate the earlier parts of design, namely the functional detailing, must be considered.

The Iverson notation is very useful here since it can be used to detail design information at this functional level, i.e. without resorting to structural constraints, and has sufficient flexibility to detail at different levels of parallelness at operations. It also can be easily extended to describe the operations within a structural definition but has no provision for defining this structure nor for any explicit timing.

6. THE LOGIC DESIGN LANGUAGE

6.1 Introduction

A logic design language is primarily a language to describe the algorithms for logical processing of system. Its main uses are in the design of logic systems; however, the language should also be capable of documenting existing systems. Furthermore, it is to be used by members of other disciplines also, as a common reference language and thus should be lucid, sufficiently descriptive, yet without too much detail. Conversely however, a description in this language must be interpretable by a computer as a program to produce abstract data for subsequent manipulation, and this requires that the language is highly structural, highly symbolic and that the description contains a considerable amount of detail.

As noted in the previous discussions a program in a conventional programming language tends to define a set of processes to be executed sequentially, where as a logic system in general contains facilities to execute processes in parallel and the necessary synchronization. The language must reflect this clearly. Also, the structure of the language should be such that undue restrictictions are not imposed on the designer's mode of design, but rather is adaptable to the different methodologies used by different designers and cater for the different aspects of a design process.

In/

In the following sub-sections we discuss the various facilities demanded of the language, how they are catered for, and the structure of the language.

6.2 Structure of the Description of a System

The system under consideration must be of a nature such that a set of abstract data, the level of which is decided by the designer, may be generated from its description. It should therefore be either constrained to a certain size (in terms of, say, an algorithm) or segmented down to produce manageable sub-systems. It is suggested that this segmentation is based on a functional division within the system, as discussed earlier.

The system can then be described in the following ways:

(1) Entirely by its input-output behaviour

(2) In terms of the inputs, outputs and an algorithm or algorithms defining the functions within the system

(3) In terms of the inputs, outputs, a predefined structure and the data flow

Despite the distinctions in the different ways however, the basic structure of the description must necessarily be common. A designer may wish to use any one or more of the above methods to describe a system depending on the size of the system and the detail available.

The description therefore, is organized in a block structure similar to Algol; however, there are some important differences. In Algol, a block introduces a new level of variables,/

variables, labels etc., or it may be an independent entity, in the form of a procedure, which may be accessed by program with actual parameter substituting the dummy parameters of the procedure. Owing to the sequential nature of Algol, as any other programming language, only one copy of each procedure needs to be maintained. In the logic design language however, a 'procedure', or in the general terms, a system or a sub-system, may be one of two types, namely, one which is shared, in the same way as an Algol procedure, and one which is duplicated.

We define FACILITY as being a system or a sub-system which is shared, with different arguments as necessary, and a MODULE as being a system or a sub system which is duplicated for each separate use. Of course, a module or a facility in turn may contain, within it, additional modules or facilities.

In the logic design language (LDL) therefore, a system is a Module containing various other modules and facilities and the description in the LDL in a program defining the interrelation between the inputs, outputs and any facilities and modules contained in the System Module. In a programming language this interrelation is always defined by an 'algorithm; however, in LDL it could be in one or more forms as selected by the designer. For example, the description may be a truth table, a state table, flow table, wave form description, functional algorithm or an algorithm in terms of predefined structure. The designer specifies the/

the type of data involved by succeeding the BEGIN at the start of the block by the appropriate type name and when the type of information is to be changed, this is done by introducing a new block.

A module or a facility in general must be declared before it is used. However, two other forms are also allowed. A block may be declared as FORWARD in which case the declaration is expected later on in the description. Alternatively a module or a facility may be declared as LIBRARY where a library of previously designed modules or facilities has been set up and is to be used when completing the description. The library facility is particularly useful when a team of designers design different sections of a system separately and compile a library in the process which is then accessed to complete the overall design.

As mentioned earlier a block introduces a new level of variables, label etc. thus an identifier declared in an outer block is accessible to an inner block except when an identifier with the same name is declared in the inner block. The identifiers declared in the inner blocks are never accessible to the outer block. Similar restrictions also apply to labels.

The/

The syntax for a 'program' in the LDL is given below in the Backus-Naur form

```
<program>               :=  <block> FINISH.
<block>                 :=  <unlabeled block> | <label> : <block>
<unlabeled block>       :=  <block type> <block head> <description>
                            END. |
                            <block type> <block head>  FORWARD |
                            <block type> <block head>  LIBRARY
<block type>            :=  MODULE | FACILITY
<block head>            :=  <block name> ( <input list>.<output list> )
                            <value part> <specification part>
<block name>            :=  <identifier>
<input list>            :=  <identifier list> | <empty>
<identifier list>       :=  <identifier>. | <identifier>.
                            <identifier list>
<output list>           :=  <identifier list> | <empty>
<value part>            :=  VALUE  <identifier list> | <empty>
<specification part>:=  <type> <identifier list>. | <type>
                            <identifier list>.
                            <specification part>
<type>                  :=  SCALAR | VECTOR | MATRIX | CLOCK | PULSE
<description>           :=  <description head> . <description tail>
<description head>      :=  BEGIN  <description type>.
                            <declaration>
<description type>      :=  TRUTH TABLE | STATE TABLE | FLOW TABLE |
                            BOOLEAN EQ. |
                            WAVEFORM | REGULAR EXPRESSION |
                            FUNCTIONAL | STRUCTURAL
```

```
<declaration>        ::=  <empty> |<type> <identifier list>.
                          <declaration>
                           GLOBAL CONDITION <Boolean expression> .
                          <declaration>
<description tail>  ::=  <block>.<description tail> |

                          {description in the appropriate format}
```

A typical example of an adder would be as follows

```
MODULE ADDER (A.B.K1.I..C.K.).
VECTOR A.B.C [0:I]. SCALAR K1.I.K .

BEGIN STRUCTURAL .
      SCALER J. VECTOR KP [0:I + 1] .
      LABEL  L .
      MODULE ADD (A.B.K1..C.K).
      SCALAR A.B.K1.C.K.
      BEGIN TRUTH TABLE .

            ABK1    CK
            000     00
            001     10
            010     10
            011     01
            100     10
            101     01
            110     01
            111     11

      END .
```

```
0        K [0] ← 0.
1  L:    J   ←   0.
2        ADD (A [J] . B[J].KP [J]..  C [J] . KP [J + 1]  ).
3        J ← J + 1 .
4        IF   J≤I    GOTO     L.
5        K ← KP [J + 1].
   END .
```

6.3. Description

The description of a system may be in an abstract form, e.g. when the system is defined entirely in terms of its input-output behaviour. Common forms of such descriptions are truth tables, state tables, flow tables, boolean equations, regular expressions etc. A description in one of these forms has neither a provision of any kind to include an algorithmic type of description nor to introduce named modules or facilities other than those determined by the subsequent manipulation programs. The advantage of such a description is that the full power of automation may be applied to produce an optimal design. The disadvantage however is that the description tends to be very lengthy; and an interactive mode of operation to develop it is preferred and this in turn requires a versatile command structure. A suitable command structure to develop an abstract description of a combinational network is given in the following section.

A major part of the description in LDL however, will be in the form of an algorithm either at an abstract level or in terms/

terms of structural constraints of the system. In both cases the description is given by statements. At an abstract level a statement may be a data transfer as in a programming language, or at a structural level it may be a register transfer type of statement.

In general a statement defines a sequence of actions to be performed and once initiated, the execution of the statement cannot be interrupted. A set of statements may be grouped together to form a compound statement where again once initiated the execution of the statement cannot be interrupted unless a global condition declared within the compound statement becomes false. A compound statement is distinguished by enclosing statements between BEGIN and END. The enclosed statements themselves may be any statements including compound statements. A compound statement may also introduce new global (global to the compound statement) conditions and new variables.

Sequencing is implicit in the order in which the statements are presented except when modified by either explicit or implicit parallelness or by branching. Labels may be associated with each statement for branching.

Each statment can also be made subject to a condition or a set of conditions, in the same way as in Algol as long as the evaluation of the conditions produces a logical value of true or false. These conditions can be any relational tests. A statement may also contain several sets of conditions and the corresponding actions for each condition similar to the Algol conditional statements. In addition we introduce a notion of global conditions which are tested prior to commencement of execution of each statement. For example a clock signal or an interrupt signal from a peripheral unit may be a global condition. The scope of global signals may also be controlled by declaring it at the appropriate level that is if a signal X is declared as global in block A then it influences all the statements and blocks contained in block A but if block B contains A then signal X will not influence the execution of block B.

We also introduce additional constructs to indicate synchronism and parallel execution, namely the until statement, when statement, the while statement and the in parallel statement. In the first three cases a condition, as defined by a boolean expression or a relational test is monitored continuously. In the until statement the statement following the test is executed, repeatedly if necessary, until the condition becomes true , the converse is true in the while statement if when the condition become false control is passed to the next statement after the while statement. The when statement effectively requires the system to halt until the condition tested becomes true.

The in parallel statement initiates the execution of all the statements defined in the scope of the in parallel statement together. The statements defined to be executed in parallel may themselves be any statements including in parallel statements. This facility we feel is particularly important when asynchronous processes are executed in parallel.

The/

The operations within a statement are evaluated using right to left (N.B.) procedure as required by the Iverson notation. This however, may be modified by parenthesis.

The syntax for a description is given below in the Backus-Naur form.

```
<description>          ::=  <statement>. <description>|
                            <statement>
<statement>            ::=  <unconditional statement>|<condition>
                            <unconditional statement> <alternative>
<unconditional         ::=  <until statement> | <while statement>|
  statement>                <when statement> |
                            <in parallel statement>|<simple statement>|
                            <compound statement> |
                            <facility call statement>|
                            <module call statement>
<until statement>      ::=  UNTIL  <boolean expression>
                            DO  <statement>
<while statement>      ::=  WHILE  <boolean expression>
                            DO  <statement>
<when statement>       ::=  WHEN  <boolean expression>
                            DO  <statement>
<in parallel           ::=  IN PARALLEL DO BEGIN  <description>. END
  statement>
<simple statement>     ::=  <branch statement> | <assignment>
<branch statement>     ::=  GOTO  <branch point>
<branch point>         ::=  <label>
<assignment>           ::=  {assignment written in Iverson notation}
<compound statement>   ::=  BEGIN  <declaration>  <description>  END
```

```
<boolean expression> ::= {an expression when evaluated returns
                           a true or false value}
<facility call        ::= <name of facility> ( <input parameters>.
  statement>              <output parameters> )
<name of facility>    ::= <identifier>
<input parameters>    ::= <parameter>. | <parameter>
                          <input parameters>
<output parameters>   ::= <parameter> . | <parameter>
                          <output parameters>
<parameter>           ::= <identifier> | <expression>
<module call          ::= <name of module> ( <input parameters>.
  statement>              <output parameters> )
<name of module>      ::= <identifier>
<condition>           ::=  IF <boolean expression> THEN
<alternative>         ::=  ELSE <statement> | <empty>
```

6.4 Variables

The variables in the description are interpreted as in Iverson notation, i.e. they can be logical, integer or real variableseither in a scalar form or vector form. Matrix manipulation is not envisaged at present but a reference may be made to any vector (row or column) of an array. A variable must be declared (at the head of the block) before it is used. However, it is not distinguished by any particular terminology as is inherent in the Iverson notation but is implied in the usage. For example, a vector quantity when used as a scalar will refer to the right most scalar quantity. Similarly a numerical quantity used as a logical quantity will be interpreted as trueif it is non-zero or false if zero.

7. COMMAND STRUCTURE OF THE TRUTH TABLE GENERATOR

In the following section the facilities and the command structure which will be used to input and complete a truth-table in an interactive mode are presented. The account is divided into three subsections: the types of combinational networks and their requirements from a designer's viewpoint are given in the first, and the command structure and the proposed method of implementation using the Honeywell 516 computer in the department are given in the second and third subsection respectively.

7.1 The Requirements

The behaviour of a combinational network may be known to the designer in different forms. These are broadly categorised into the following types which are not necessarily exclusive but provide convenience of detailing.

i) The full truth table. The designer knows and wants to input the output behaviour for each input configuration*.

ii) The truth table with incompletely specified I.C. This is similar to i) above except that the designer only knows a subset of the I.C. and the remaining are either 'don't cares' or a fault condition.

In both the above cases the designer can specify each input or output variable as being one of three values, viz.

* Here after referred to by I.C. and similarly an output configuration will be referred to by O.C.

on or a 1 condition, off or a 0 condition and a 'don't care' condition. If the entries are specified in binary notation, value of each variable can be explicitly indicated as a 1, a 0, or a - respectively. However, with up to 20 input and 20 output variables the full configurations in binary form are tedious and lengthy to input. A shortened version is often used where three bit groupings from the least significant end (the right hand end) are expressed by their equivalent octal value. A slight difficulty arises when the don't cares have also to be specified and to overcome this the following two* methods are often used.

1) Each configuration is specified as an octal duple with the first element set equal to the octal value when all the non-on conditions, i.e. the off's and the don't care's are set to 0 and the second element similarly specifying the off conditions.

2) The second method is similar but specifies the on and the don't care conditions. The choice between the two is arbitrary and entirely depends on the designer.

In some cases, it is easier to input the entries with their equivalent decimal forms. The don't care conditions are then treated in the same way as above.

iii) Routing Networks. This does not actually involve real 'design' but as it is one of the commonly used types it is included in the discussion here. It/

* Others are possible but they are only different combinations of the ON, OFF, and DON'T CARE conditions and are not considered here.

It is characterized by the fact that inputs can be divided into controlling variables and the routed variables.

iv) Functional Relation. Not all truth tables are known in their abstract form and in fact, the most common mode is when the I.C.'s have a mapping into the O.C.'s and this mapping is known. The designer may wish to specify this mapping as a logical or numerical function.

v) Iterative Combinational Networks. These fall between the combinational and sequential networks. In practice sequential techniques are often used to solve problems of such networks and a combinational treatment tends sometimes to be academic. However, these networks will be included in the programmes where they may be specified by DO loops similar to the FORTRAN DO loops.

7.2 The Usage of The Programme

The first essential set of parameters required for the programme is the input output size. The present minimization programmes at the Southampton University operate on up to 20 input and 20 output variables, and this same limit will be adhered to in the truth table generation programme. The variables can be separate identifiers or members of arrays or a combination of both, subject to the condition that the input and output names may not be common. These will be declared in response to requests generated by the programme immediately on initiation.

During the process of generating the truth table in an interactive mode, it may be necessary to be able to type headings/

headings out and the truth table filled in a column form. Since there are only 72 character positions per line available on the tele-type unit, a severe restriction has to be placed on the length of each identifier. A maximum of two characters, both alpha characters, per identifier and in the case of array identifiers the first character will be assumed to be the name of the array and the second character, a digit, will specify the relative address. The latter constraint allows only 10 variables in an array; however, it is felt that this limit will still be quite adequate and if larger arrays are necessary they can be specified as two arrays.

The generation of truth tables is achieved in two ways:

a) by inputting a truth table via the console by an interactive process or

b) inputting a functional description and letting the program generate the truth table.

These two methods are distinguished by the directives immediately following the input-output declaration. If a functional description is put in, the program will fill up as much of the truth table as possible and the remainder will need to be completed by method a).

The completion process, a), is executed in two modes:

i) the program cycles through each unspecified I.C. and the designer fills in the appropriate O.C. and

ii) the designer specified both I.C. and the corresponding O.C.

A/

A mode indicator is used to identify each mode: it is set to zero, also the default mode, for the former and set to one for the latter.

a) Fully specified truth table in binary form.

In this case, immediately after the input output declaration, and when the programme is in an awaiting state, the designer inputs a command @GO. Since there are no other commands the programme will recognize this as a fully specified truth table input in mode O, and will print out two headings INPUTS and OUTPUTS followed by a list of the input and output variables in the same order as in the declarations. It will then print out the I.C. OOO...O and invite the user to type by displaying a question mark (?). The designer then enters the corresponding O.C. with 1's, O s or a blank or hyphen to denote a don't care. The teletype will be automatically aligned for columnizing but if the inputting is prematurely terminated by a carriage return, line-feed or a semicoln, the remaining entries are assumed to be don't care's. After carriage return the programme will line feed and print out the next O.C. and so on. Any additional carriage returns or line-feeds will be ignored.

b) Fully specified truth table in octal or decimal form.

If a decimal or octal print out or if the designer wishes to input in decimal or octal the following commands are used:

	@TY OC,OF	carriage return	for octals with OFF's specified see 7.1.ii
	@TY OC,DC	"	for octal with DON'T CARE'S
	@TY DE,OF	"	for decimal with OFF'S
and	@TY DE,DC	"	for decimal with DON'T CARE'S

These/

These commands instruct the programme to type out in the appropriate format and also inform the programme as to which type of inputting should be expected. They can be used before the initial @GO command in which case the list of variables as headers will be suppressed or if in the middle of the programme, the last line will be reprinted and the subsequent output will be in the required format. A return to the binary format is made by the command

```
        @TY BI
```

the headers then will be displayed again.

The designer on the other hand can override the specified format while inputting by typing in OO (the letters O), OD, DO, DD and BI before the actual inputting to mean octal with off's, octal with don't care's, decimal with off's, decimal with don't care's and binary respectively.

c) Truth table with incompletely specified I.C.'s

As indicated above a set mode command, @MO = can be used to set the mode to 1 to allow the designer to input the whole or the required amount of the truth table himself. A more common usage, however, is when a subset of the inputs need to be given a value and the others are cycled through. The set variable commands, @SV are used for this, the format of which is shown below.

```
      @SV   variable list   =   logical value
```

For example, in a 4-input, 2-output combinational network the first two inputs never occur together. A possible method for this is as follows:

```
    INPUTS ? a,b,c,d          The programme messages are in
    OUTPUTS ? e,f             capitals
    @sv a=0
    @sv b=1
    @go

          INPUTS              OUTPUTS
        A  B  C  D             E  F
        0  1  0  0     ?       1  -
        0  1  0  1     ?       0  0
        0  1  1  0     ?       1  1     etc. followed by
    @sv a=1
    @sv b=0
    @go
```

and filling in the next part. The other O.C.'s are set to DON'T CARE'S by entering

```
    @ot = -
    @go
```

and finally terminating the input process by

```
    @fi
```

In the above example the 'others' were set to DON'T CARE'S, but they could just as well have been set to a required O.C. to indicate a fault condition. If the input variable assignation was to be done in decimal or octal then the VALU operator, corresponding to the Iverson operator, could be used as

```
    @sv valu  <variable list>   =  <decimal value>
and @sv valu  <variable list>   = '<octal value>
```

respectively where a variable list is a list of variables separated/

separated by commas acting as concatenation* operators.

d) Routing Networks

Here the inputs are divided into controlling signals and routed signals. In theinput specification the routed signals are set to DON'T CARE's and the remaining cycled through. The O.C.'s then could be inserted as in formal programming languages, i.e. by enclosing literals in quote marks or preceding them by 'equal to' signs; however, as the number of characters in a line is necessarily limited, a non-printing character, CTRL L, will be used. The choice of the character is such that it does not conflict with any of the control functions of the tele-type unit; the letter L is chosen to stand for literal.

e) Functional Relation

A @FUNCTION directive is used to instruct the controlling program to accept the subsequent input in functional format; however, since the translation on the HONEYWELL 516 computer is to in conjunction with the Fortran Compiler, differences have to be introduced. The usage of the format is given below.

i) No segment declaration is made since the functional specification will consist of only one segment; however, this segment may be processed in several steps each initiated by @GO directive, providing that the specification until then is complete within itself, i.e. it does not refer to a non-existing label etc.

* defined later

ii) For every new instruction, the controlling programme will type out a sequence number which may be used for deletion etc.

iii) All quantities will be assumed to be of either logical or unsigned integer type and the operations between them will determine their type. The operations for the time being will be limited to those listed below with their trans-literation, but it is hoped that the entire vocabulary of the Logic Design Language connected with combinational networks will be implemented.

iii.a.) Each single variable may have two logical values: True and False represented by a 1 and a 0 respectively. Arrayed variables will be considered to be strings with logical values for each element. If in an assignment the quantity on the right hand side is greater than the capacity of the variable on the left hand side, then only the right hand portion will be preserved and the rest will be lost; on the other hand if reverse is the case then the left hand side will be filled by 0's. Similar arrangements will be employed in arithmetic operations.

iii.b.) In string operations the variables may be

1) whole arrays in which case the arrays are referenced by their names only, i.e. without indices,

2)/

2) parts of arrays in which case the first and the last indices are given in parenthesis separated by a comma and following the names of the arrays, and

3) set up using concatenation operators (,). A new variable may be introduced to assume the value of the concatenated array.

Examples of 2 and 3 are A (1,4) and A1,A2,A3,A4 where each refers to the sub-array formed out of the first four elements of the array A. In the latter case an assignment

N = A1,A2,A3,A4

may be used where N is a new variable which may be used in subsequent manipulations.

iii.c.) The processing order will be from the right to the left unless modified by parenthesis.

iii.d.) Unpredictable results will occur if any of the variables on the right hand side have been set to DON'T CARE's before the instruction is executed.

iv) The following operations will be implemented.

Function	Symbol	Example
Logical Not 1' complement	'	A'
Logical Or	↑	A ↑ B
Logical And	.	A.B
Logical Nor	(↑)'	(A ↑ B)'
Logical Nand	(.)'	(A.B)'
Addition	+	D+E
Subtraction/		

Function	Symbol	Example
Subtraction	-	D-E (D $\geqslant$ E)
Equals or assignation	=	A = D+E
Catenation	,	A = D,E
Greater than	>	IF (A >B) A= B↑ C
Greater than or equal to	>=	IF (A)= B) A = B'
Equal to	=.	IF (A=.B) GOTO 10
Not equal to	='	IF (A='B) GOTO 20
Less than or equal to	<=	IF (A <=B) D= D'
Less than	<	IF (A <300) B=0
Conditional	IF	IF(G) A=B'
Branch	GOTO	IF(A='B) GOTO 20

v) Transfer of control from normal execution is achieved by GOTO, IF and DO statements. In the present case only ordinary GOTO statements will be implemented, i.e. computed GOTO and ASSIGN statements will not be considered. The DO loops will be the same as in Fortran except that the terminal statement for each DO loop must be a GO (without the @) statement.

vi) There will be no DATA statements.

vii) When typing the specifications in, the first characters following the sequence numbers may be a C to indicate a comment line or a digit to start a label which should be all numeric, or a form character to skip the label field. There will be no column for continuation lines but instead a delimiter (;) is to be used to indicate the end of a program line. Comments may be introduced following this semicolon and the next carriage return.

viii)/

viii) There will be no instructions similar to the FORTRAN input-output instructions and all outputting will have to be done in the format specified elsewhere. However, a print-out of the programme written so far may be obtained by introducing a 1 after the terminal @GO statement and it takes the form

```
@GO 1 CR for a print out
@GO   CR for no print out
```

f) Iterative Functions

This is a special category of e) above and the DO loop format defined above will be used for this type of function.

The above instruction define the commands used to partially or completely fill the truth table. The remaining instructions deal with modifications or subsequent manipulations such as displaying or paper-tape output etc. Present plans do not include visual display using the graphics terminal, but since this will provide a very rapid and useful means of checking the truth table contents serious consideration will be given to its use later on.

i) Set up a mask. An assignment is used to setup a mask for deleting purposes. Upto 10 masks will be allowed at any one time and are set up by the command

@SMn = ⟨mask pattern⟩

where n is a decimal number 0 to 9 inclusive. The mask pattern can be of any of the five types used in input output/

output specification discussed earlier. If the pattern is specified in binary and not all entries (each entry corresponding to an input variable) are specified, the mask will necessarily be left justified and in the same order as the input declaration; the unspecified entries will be set to DON'T CARE's.

ii) The delete instruction. The delete instruction is @DL and can take one of the following forms.

a) @DL Mn where n is a decimal number 0 to 9 inclusive and Mn specifies a mask previously set up. On execution the instruction causes the O.C.('s) corresponding to the I.C. specified by the mask to be deleted.

b) @DL Mn_1,Mn_2 where Mn_1 and Mn_2 specify a mask each as before. This instruction causes all the O.C.'s corresponding to the I.C.'s between those speeified by the masks to be deleted. Mn_2 may be replaced by a decimal number in which case this instruction will be executed as in a), and repeated for the entries in the table the number of which is specified by the decimal number. Note that if VALUE Mn_1 = VALUE Mn_2 + 1 the deletion process will cycle until all the truth table is deleted. A better method is to use the following form.

c) @DL AL This instruction deletes all, i.e. effectively restarts the programme.

d)/

d) Using the format of a) and b) above the masks can be set at the time the delete instruction; the Mn's then are replaced by binary, octal or decimal patterns.

e) If the input format is of functional type a delete instruction should refer to a line by its sequence number in the functional specification. The format for the delete instruction then is

    @DL n

where n is the sequence number

A line following this command may be

1. another delete command
2. start of additional functional specification, in which case the updating specification until the next delete command or @GO command will be inserted after the delete line, or a
3. @GO directive to execute the updating and reprocessing. A 1 is introduced after the @GO directive if a listing of the updated file is also required.

iii) The entries may be changed instead of being deleted by the @CH (change) instruction. It is used as follows.

    @CH Mn,Mm

Mn specifies a mask as in ii.a.) above, and Mn is a binary, octal or decimal pattern of the usual format which should replace the O.C.('s) corresponding to the I.C. specified by the mask Mn.

iv)/

iv) Since the programme is on an interactive basis, it is quite possible that part-way through a need for a new input or a new output may arise or that an input or output may be found redundant. Rather than starting the programme all over again the following four instructions may be used.

```
@RM IP, <input variable list>
@RM OP, <output variable list>
@NW IP, <input variable list>
@NW OP, <output variable list>
```

where RM, NW, IP and OP refer to remove, new, input and output respectively. Usually an input should be removed only if it is redundant; however, if on its removal conflictions are encountered then these will automatically be brought to the designer's attention. Similarly if a new input is introduced then the two O.C.'s 'distinguished' by this input will be set equal to the same value as when it did not exist and the subsequent entries of course will be correctly treated. The output entries corresponding to the new outputs prior to their introduction will be set to DON'T CARE's.

For convenience of implementation, problems with upto 12 input or output variables are treated differently from those with greater inputs or outputs. Thus care should be taken to see that these boundaries are not crossed with the above instructions.

v)/

v) The @EQ instruction. This instruction is used to equate the O.C.'s corresponding to two or more I.C.'s and may have two, three or four arguments according to the function required. Each argument specifies an I.C. and can be defined directly or by masks set previously set as with @DL or @CH instructions. If two arguments are specified then the O.C. corresponding to the second I.C. is set equal to the O.C. corresponding to the first; if three are specified then the second two refer to the limits between which the equate operation has to be repeated; and if four arguments are specified then the block specified in the last two is equated to the block specified by the first two. Errors such as conflictions or unequal length blocks will be brought to the designer's attention.

vi) Mode setting. The mode is set by the directive

@MO =

A '1' or a '0' is entered on the right hand side to set or reset the mode respectively.

vii) Inputting via the paper tape reader. To enable the paper tape reader for command and data input the directive @PR will be used. The last instruction on the tape must be @AK to return the control back to the tele-type unit key-board.

viii) File* Input. If a file of the truth table is to be input the instruction @FI n will be used, where n is a decimal number identifying the file or if it is preceded by ' then it is an octal number.

* see following subsection

ix) Output. At present only two output media are considered, namely the tele-type unit and the paper tape punch; however, it is hoped that the graphics terminal could also be used at a later data. The corresponding instructions are

@AP PL_1, PL_2 for output on the tele-type unit and

@PP PL_1, PL_2 for output on the paper tape punch

PL_1 is a parameter list to define the scope of output which can be one of the following three:

a) A small section of the truth table whose start and finish are specified in the same way as in @DL instruction.

b) A file is output in which case PL_1 is specified as FN= followed by a decimal or octal number as in viii) above.

c) The entire truth table is output in which case a hyphen (-) is written for PL_1.

PL_2 is a parameter list to indicate the format of output and same abbreviations as in the @TY instruction will be used.

Note 1: The paper tape output is to be compatible with the input requirements of the subsequent minimization programmes and it should be remembered that the binary format is not used.

Note 2: If the file number is entered as a hyphen (-) then the file currently being processed will be output.

x)/

x) Return to B.O.S. The programme will normally be run under the auspices of the operating system B.O.S. controlling the computer. A return to the operating system will be made if at any time @SB is typed in.

xi) Error Corrections

   i) Errors while typing in. The same conventions as those being currently used with B.O.S. will be employed, viz.

      a) Delete the last character. This is done by one CTRL H per character to be deleted with the modification that spaces will be ignored and one deletion per character other than space should be input.

      b) Delete the whole line. A left pointing arrow is input to delete the whole line.

   ii) Interruption during execution. It will be necessary to include a facility to interrupt the execution phase; however the exact format will be defined at a later date.

xii) Comments. Comments may be included any time between quote marks (""). These will only be useful at input time as these will not be stored and cannot be retrived except in the case of functional specification, where they are introduced by a C in the first column of a new instruction line or following the terminating semicolon and the subsequent carriage return.

7.3/

7.3 Proposed Implementation

The programme is required to deal with upto 20 input and 20 output variable problems. If a full truth table is generated for a problem of this complexity then it would contain of 2^{20} rows and 40 columns or putting it another way 2^{20} words of storage, assuming that each word can hold all the 40 columns, will be required. The constraint demands that the word length be 40 if only 0's and 1's are to be stored or 80 if the DON'T CARE's have also to be stored. Using a 16 bit word therefore, $2^{20} \times 5$ words of storage space will be necessary.

However, in practice, no designer is likely to generate a truth table of such a size or if he does not all the entries are likely to be completely distinct and this could lead to a saving of storage space. In any case, the storage and the manipulation has to be severely scrutinized to keep the problem within manageable size. Various schemes for storage are considered below.

1) The address of each computer word is made to match with an input configuration. This immediately has an advantage that the input configurations do not have to be stored thus on an average the storage space is halved. This also has the disadvantage that if the inputs contain any DON'T CARE's then the corresponding outputs have to be repeated and this means that for every DON'T CARE input two identical output entries have to be defined. Thus if there are a large number of DON'T CARE input configurations a large amount of redundancy results.

2)/

2) The I.C. is stored along with its mask* and the corresponding output entries which will require only the conditions specified by the designer to be stored; however, again DON'T CARE conditions have to be expanded and secondly, since this data will be stored sequentially as it is input the order of the I.C.'s will be lost and consequently, no indication will be available as to which I.C.'s are not specified other than by placing an end marker and cycling through the memory to test for an I.C.

3) A third method is to store the I.C.'s in the same way as above, but in an order according to their values. To keep a tab on the relative position of the entries they could be either

   a) stored consecutively in an ascending order but in which case a later addition or deletion means pushing down or raising the later entries or

   b) attaching a link word to point to the successor, i.e. to use a list structure.

   The list structure method requires one more word per entry; however, it offers two major advantages:

      i) it is very flexible since the size of the list can be altered very easily by altering the links and

      ii) it offers a concise and precise way of storing data.

   It is proposed, therefore, that a type of list structure be adopted.

---

* defined overleaf

Each block of data, or cell, will require to hold three items of data, namely the I.C. in an expanded form, the O.C. and a mask. The mask defines which of the entries in O.C. contain valid OFF's or ON's and which are to be taken as DON'T CARE's

Link	Input Config.	More sig. output	Mask for M.S.O/P.	Less sig. output	Mask for L.S.O/P.

Figure 1. A cell in a list structure.

The I.C.'s are the same as inputted by the designer if they have been specified in full, or fully expanded by the programme based on the specification provided by the designer. The full expansion is necessary since for subsequent manipulation the truth table must contain all the input configurations for which a non-trivial output configuration exists. In storage the two trivial output cases which will be omitted are as follows:

1) when a large number of I.C.'s exist for which the outputs are all DON'T CARE's or

2) when a large number of I.C.'s exist for which the outputs are either all 0 or some other specified O.C.

Both these cases are defined by the directive @OT = and the programme will check against this before outputting the truth table.

The Data Words. The programme will handle upto 20 input and 20 output variables. Since the computer word is only 16 bits long at least 2 words will have to be used for each output configuration/

configuration, however, allowing for the relevant masks 2 more words will be required. Hence in a full sized problem each block of data must contain at least 4 computer words per O.C. Similar considerations for the inputs show that 2 words per input configuration are necessary. Thus, 7 words including the link word per block corresponding to each row of the truth table will be necessary, i.e. in 16k store (that of the Honeywell 516 computer) only about 2k entries will be possible without leaving much room for the programme itself. This is overcome by dividing the data into files, the number of each file is determined by the value of the more significant bits (the first inputs during declaration), and the address within a file by the value of the less significant bits. The division between the less and the more significant bits is, therefore, dictated by the constraint that within each file each entry is directly addressable. This in the worst case means that all 20 outputs are specified for each I.C. and the whole file can be held in the computer store or within about 12k, allowing the number of I.C.'s per file of upto about 2k or 12 input bits. The less significant half will therefore be with 12 input variables and the remaining 8 inputs will generate upto 256 files which also will be linked in a list format and stored in backing store. Access to a file in the backing store is obtained by dumping the file present in store into the backing store and reloading the store with the named file.

For economy of storage the programme in core will be limited to instructions to call the relevant routines from the disc store/

store and the current programme operating on the data. Thus the area in store will appear as a small executive to which the user communicates.

Most of the routines will be in DAP, the low-level language for the Honeywell 516 computer but for functional specification the Fortran Library and some subroutines in Fortran will also be used.

8. CONCLUSIONS

8.1 Summary

In the design of any logical system , the behaviour is usually expressed in a natural language first, the designer then extracts the relevant information and puts it into formal terms and then proceeds to the final design. Various techniques of abstraction are investigated here; and their applications to large system design are studied and the conclusions summarized below. A pertinent factor involved is that the human designer has relatively little patience to learn new techniques and abandon his usual methods, particularly if the new techniques are rather remote from his way of thinking.

For a small scale design, the natural language specification is easily converted into a flow table, state table or a state diagram, and switching theory can be used extensively to obtain an optimal design. However, large amounts of store are used in the process especially in the last case where graphical inputting is required.

The algebra of regular expressions has been developed to express the above information in a linear form and in mathematical terms allowing easier computation. It is precise and can apply to all synchronous and pulse mode systems; it is also closer to a natural language description than, say, the state table approach. However, the use of regular expressions as a design tool has several problems. The major ones are

1) It/

1) It is highly mathematical and as such the designer will have to be educated specially.

2) Different methods used to obtain the regular expressions tend to produce different answers and which usually bear little or no resemblance to each other, despite the advances in the algebra, their identity is still cumbersome to prove.

3) As in the case of the state table approach, it is only applicable to finite state machines.

The last objection is particularly relevant, since large scale systems cannot, in general, be represented as finite state machines, or conversely, if they are so specified, the description in terms of, say, state tables would be astronomical in size.

The large scale systems of interest to us are essentially instruction execution machines; the instructions may be known at a high level but their detailing, if any, is not known. The designer, designing intuitively, defines a structure with known capabilities and limitations. He then decomposes the instructions into low level commands which lie within the scope of this structure. The decomposition merely defines the way data is transferred between registers and the necessary control for it.

Earlier register transfer languages, devised to express micro program were simple and could be directly mapped into/

into hardware; and thus were good tools for analysis of already designed systems and for the automation of implementation. They were however, limited in their scope of specification which tended to be lengthy.

Further languages were developed to increase the flexibility and the specification ability to which notational and operational conciseness was introduced by using complex operators and macro cells. Some were usefully developed based on the structure of existing programming languages, such as DDL based on Reed's language [78] and Cassandre based on Algol.

Since all the languages used a predefined structure, the automatic part was still limited to deriving the controlling circuitry and the combinational logic driving the register structure. Gerace [54, 55] gave methods by which the register structure implied in the register transfer description may be reformulated into an iteratively - connected - machines structure and the formal abstract definition for each may be derived.

He also gave methods of implementing the control part of a system using read-only-memories. Another useful technique was given by Stabler [85] for microprogramme transformation, that is, to modify the structure and shift the line dividing the register structure and control structure.

The Iverson notation [66] provides a means of describing the logical functions of a system at various levels of detail, including elemental bit levels, independent of the machine/

machine structure and in an algorithmic manner, lending itself to a good implementation in terms of hardware realization. While providing excellent facilities for the description of an algorithm however, Iverson notation is particularly lacking in high level functional description and in timing.

At the other end of the spectrum, some langauges were specially developed to describe the structure of a system. The application of these languages in the early stages of design is limited, nevertheless they have a wide range of applications, including implementation in a design automation suite, structural simulation, documentation and fault diagnosis.

Ideally we would like to employ techniques offered by switching theory in our design, since only these are rigourous enough to produce error free designs and also allow us to interface with the other aspects of design such as fault diagnosis in a consistent manner. Unfortunately however, switching theory is still at an infancy stage as far as large scale system design is concerned.

The Petri Nets and occurrence graphs [97] show a promise of dealing with large scale systems in an abstract manner, with these it may be possible to produce a uni ed theory of system behaviour description and system architecture design.

8.2 Current Work

One of the most important aspects of a design language is that it should cater for the different methods of design used by designers in a consistent and natural manner. It also should be easy to learn and be concise and precise yet flexible.

The structureof such a language has been proposed. We feel that this language allows a designer to express the design specification in a manner similar to his own thinking. It is block structured so that at the system level the blocks in the language correspond closely to the functional blocks making up the system. The control and timing interrelation between the functional blocks can be expressed at the block level. The blocks in turn can be detailed into further blocks as necessary.

At the low level the description normally would be in an algorithmic form; alternatively the designer may choose to detail in a different form and use an interactive process to develop this detail. The library facility in the language allows this to be done without modifying the general structure of the description.

A command structure to develop a description of combinat- ional networks is defined to be used with the Honeywell 516 computer at the Southampton University. The main description however is related to the ICL 1907 at the University.

The/

The language also provides a means of uniformly describing the processes of a logical system for design, simulation and documentation.

8.3 Future Work

The LDL language can be used to describe the design specification of a logical system. However, it is largely biased towards hardware systems, but is sufficiently flexible to include software specifications. The necessary extensions need to be defined.

The command structure described herein also is limited to combinational network design. Additional command structure needs to be defined, say, similar to the one developed by Smith & Tracy [102], and in particular for using a graphics terminal for this.

The current scope of the language as a whole is necessarily limited for batch processing type of operation on the ICL 1907 computer at the University. However, techniques need to be developed to suggest alternative functional blocking to the designer which he may choose to accept or ignore. This necessarily means that a suitable system design theory needs to be developed and the language used in context of this.

REFERENCES and BIBLIOGRAPHY

[1] Hill, F.J., and Peterson, G.R.,
Introduction to Switching Theory and Logical Design,
J. Wiley & Sons Inc., 1968.

[2] Kohavi, Z.,
Switching and Finite Automata Theory,
McGraw Hill Book Company, 1970.

[3] Lewin, D.W.,
Logical Design of Switching Circuits,
ThomasNelson and Sons Ltd., 1968.

[4] Arden, D.W.,
Delayed Logic and Finite State Machines,
Theory of Computing Machine Design,
University of Michigan, 1960 Summer Session, pp. 1-35.

[5] Brzozowski, J.A.,
A Survey of Regular Expressions and Their Applications,
IRE Transactions on Electronic Computers, June 1962, pp. 324-335.

[6] Brzozowski, J.A., and McCluskey, E.J. Jr.,
Signal Flow Graph Techniques for Sequential Circuit State Diagrams,
IEEE Transactions on Electronic Computers, April 1963, pp. 67-76.

[7] Brzozowski, J.A.,
Derivatives of Regular Expressions,
Journal of the A.C.M., Vol. 11, October 1964, pp. 481-494.

[8] Brzozowski, J.A.,
Regular Expressions from Sequential Circuits, Short Note,
IEEE Transactions on Electronic Computers,
Vol. EC-13, December 1964, pp. 741-744.

[9] Brzozowski, J.A.,
Roots of Star Events,
Journal of the ACM, Vol. 14, July 1967, pp. 466-477.

[10] Burks, A.W., and Wright, J.B.,
Theory of Logical Nets,
Proceedings of the IRE, Vol. 41-10, 1953.

[11] Cohen, R., and Brzozowski, J.A.,
On Decomposition of Regular Events,
Journal of the ACM, Vol. 16-1, January 1969, pp. 132-144.

[12] Copi, I.M., Elgot, C.L., and Wright, J.B.,
Realisation of Events by Logical Nets.
Journal of the ACM, Vol. 5, April 1958, pp. 181-196.

[13] Fujino, Kiichi,
A Method of Automatic Generation of Compilers,
NEC Research and Development, No. 16, January 1960, pp. 86-95.

[14] Gelenbe, S.,
Regular Expressions and Checking Experiments,
Polytechnic Institute, Brooklyn, N.Y.
U.S.G.R.D.R. AD 666 696, September, 1967.

[15] Ghiron, H.,
Rules to Manipulate Regular Expressions of Finite Automata,
IRE Transactions on Electronic Computers,
Correspondence, August 1962, pp. 574-575.

[16] Ginzburg, S.,
A Procedure for Checking Equality of Regular Expressions,
Journal of the ACM, Vol. 14, April 1967, pp. 355-366.

[17] Johnson, M.D., & Lackey, R.B.,
Sequential Machine Synthesis using Regular Expressions.
Computer Design, September 1968, pp. 44-47.

[18] Kleene, S.C.,
Representation of Events in Nerve Nets and Finite Automata,
Rand Research Memorandum, No. RM.704, 1951.

[19] Langholz, G.,
Regular Expressions and Their Analysis and Synthesis of Automata.
2nd B.C.S. Symposium on Logic Design, Reading University,
28th March, 1969.

[20] Lee, C.Y.,
Automata and Finite Automata,
Bell System Technical Journal, Vol. 39, September 1960,
pp. 1267-1295.

[21] McCullock, W.S. & Pitts, W.,
A Logical Calculus of the Ideas Immanent in Nervous Activity,
Bell Math. Biophysics, Vol. 5, 1943, pp. 115-133.

[22] McNaughton, R. & Yamada, H.,
Regular Expressions and State Graphs for Automata,
IRE Transactions on Electronic Computers, Vol. EC-9, March 1960,
pp. 39-47.

[23] McNaughton, R.,
Techniques for Manipulating Regular Expressions,
Systems and Computing Science, Eds. J.F. Hart & S. Takusu,
University of Toronto Press, 1967, pp. 27-41.

[24] Mirkin, B.G.,
The Language of Pseudo Regular Expressions,
Kibernetica, Vol. 2, No. 6., 1966, pp. 8-11.

[25] Myhill, J.,
Finite Automata and Representation of Events,
WADC Report, 1957.

[26] Oglesby, R.A.,
A Computer-Aided-Logic-Design using Regular Expressions,
Computer Design, August 1970, pp. 79-84.

[27] Paz, A. & Peleg, B.,
On Concatenative Decomposition of Regular Events,
IEEE Transactions on Computers, Vol. C-17, March 1968, pp. 229-237.

[28] Rabin, M.O. & Scott, D.,
Finite Automata and Their Decision Problems,
IBM Journal of Research & Development, Vol. 3, April 1959,
pp. 114-125.

[29] Spivak, M.A.,
A New Algorithm for Abstract Synthesis of Automata,
Proceedings of Scientific Seminar on Theoretical and Applied
Problems in Cybernetics and Theory of Automata,
Vol. 1, No. 3, Kiev, 1963.

[30] Udagawa, K., Inagaki, Y., & Tange, H.,
The State Characteristic Equations of Finite Automata and
Their Regular Expressions,
Electronics & Communications in Japan, Vol. 48-9, September
1965, pp. 25-36.

[31] Abrams, P.S.,
An APL Machine,
AD 706 741 SU-SEL-70-017,
February 1970.

[32] Anceau, F., Liddell, P., Mermet, J., Payan, C., & De Pollignac, K.,
CASSANDRE for Logical System Modelling,
Prepared for the Australian Computer Conference, August 1969.

[33] Anceau, F., Liddell, P., Mermet, J., Payan, C., & Doussey, J.,
CASSANDRE; Langage et Systeme,
Institute Mathematiquee Appliquees, Grenoble, May 1970.

[34] Bartee, T.C., Lewbow, E.L., & Reed, I.S.,
Theory and Design of Digital Machines,
McGraw Hill Book Company, 1962,

[35] Berndt, H.,
Functional Microprogramming as a Logic Design Aid,
IEEE Transactions on Computers, Vol. C-19, October 1970, pp. 902-907.

[36] Breuer, M.A.,
General Survey of Design Automation of Digital Computers,
Proceedings of the IEEE, Vol. 54, No. 12, December, 1966, pp. 1708-1721.

[37] Cadden, W.J.,
Equivalent Sequential Circuits,
IRE Transactions on Circuit Theory, Vol. CT-6, March 1959,
pp. 30-34.

[38] Cain, J.T., Mickle, M.H., & McNamee, L.P.,
Simulation of a Digital System Using a Compiler Level Language,
Proceedings of the 20th SWIEEESCO, April 1968, pp. 1301-1308.

[39] Chu, Y.,
An Algol-like Computer Design Language,
Communications of the ACM, Vol. 8, No. 10, 1965, pp. 607-615.

[40] Chu, Y.,
A Higher Order Language for Describing Microprogrammed Computers,
Maryland University Technical Report, TR 68-78, September 1968.

[41] Chu, Y.,
Design Automation by the Computer Design Language,
NASA-CR-100566, N.69-22155, March, 1969.

[42] Chu, Y., Pardo, P.R. & Yen, J.,
A Methodology for Unified Hardware Software Design,
NASA-CR-110445, University of Maryland Technical Report
TR-70-107, January 1970.

[43] Crocket, E.D., Capp. D.H., Frandeen, J.W., Isberg, C.A.,
Bryant, P., Dickinson, W.E., & Paige, M.R.,
Computer-Aided System Design.
Proceedings of the Fall Joint Computer Conference 1970,
pp. 287-296.

[44] Crocket, E.D., Capp, D.H., Frandeen, J.W., Isberg, C.A.,
Bryant, P., Dickinson, W.E., Paige, M.R.,
Computer-Aided System Design,
IBM Report 16.198, July 1970.

[45] Darringer, J.A.,
A Language for the Description of Digital Computer Processors,
SHARE-ACM-IEEE Design Automation Workshop, July 15-18th 1968,

[46] Duley, J.R., & Dietmeyer, D.L.,
A Digital System Design Language (DDL),
IEEE Transactions on Computers, Vol. C-17, No. 9, September 1968,
pp. 850-861.

[47] Duley, J.R., & Dietmeyer, D.L.,
Translation of a DDL Digital System Specification to Boolean
Equations,
IEEE Transactions on Computers, Vol. C-18, No. 4, April 1969,
pp. 305-313.

[48] A Formal Description of System 3/60,
IBM Journal of Research & Development, Vol. 3, No. 3, 1964,
pp. 198-261.

[49] Foster, G.H.,
APL: A Perspicuous Language,
Computers and Automation, November 1969, pp. 24-27.

[50] Franke, E.A. & Mergler, H.W.,
Computer Aided Functional Design of Digital Systems,
Proceedings of the 20th SWIEEESCO, April 1968, pp. 1301-1304.

[51] Friedman, T.D., & Yang, S.C.,
Quality of Designs from an Automatic Logic Generator,
IBM Computer Applications Report RC 2068 (#10536), 25 April 1968.

[52] Friedman, T.D.,
ALERT; A Program to Compile Logic Designs of New Computers,
Digest of the 1st IEEE Computer Conference, 6-8th September 1968,
pp. 128-130.

[53] Friedman, T.D., & Yang Sih-Chin,
ALERT: Methods of Logic Generation,
IEEE Transactions on Computers, Vol. 18, No. 7, July 1969 pp. 593-614.

[54] Gerace, G.B.,
Digital System Design Automation - A Method for Designing a Digital System as a Sequential Network System,
IEEE Transactions on Computers, Vol. C-17, No. 11. November 1968, pp. 1044-1061.

[55] Gerace, G.B., Vanneshi, M., & Casaglia, G.F.,
Equivalent Models and Comparison of Microprogrammed Systems.
Presented at the International Advanced Summer Institute on Microprogramming, St. Raphael, France, August-September 1971.

[56] Glushkov, V.,
Automata Theory and Formal Microprogram Transformations,
Kibernetica, Vol. 1, No. 5, 1965.

[57] Gorman, D.F., & Anderson, J.P.,
A Logic Design Translator,
Proceedings of the Fall Joint Computer Conference 1962, pp. 251-261.

[58] Harrand, Y., Anceau F., Liddell, P., Mermet, J. & Payan, C.,
CASSANDRE - Langage pour la Conception Assistee des Ensemble Logiques.
L'Onde Electrique, Vol. 49, f.1, Janvier 1969, pp. 120-126.

[59] Hellerman,
Digital Computer System Principles,
New York, McGraw Hill Book Company, 1967.

[60] Hennie, F.C.,
Finite State Models for Logical Machines,
J.W. Wiley, 1968, pp. 14-20.

[61] Harnbuckle, G.D., Thomas, E.L. Spann, R.N. & Diehuis, R.J.,
Computer-Aided Logic Design on the TX-2 Computer,
SHARE-ACM-IEEE Meeting Proceedings, 1969, pp. 357-369.

[62] Hussan, S.S.,
Microprogramming Principles & Practices,
Prentice Hall, 1970.

[63] Ilovaiskii, I.V., & Lozowskii, V.S.,
Using the Address Language to Automate Synthesis of Digital Computers,
AD 679 519, March 1968.

[64] Iverson, K.E.,
A Common Language for Hardware, Software & Applications,
Proceedings of the Fall Joint Computer Conference, 1962, pp. 121-129.

[65] Iverson, K.E.,
A Transliteration for Keying and Printing Microprograms,
IBM Report, 12th April 1962.

[66] Iverson, K.E.,
A Programming Language,
John Wiley, 1962.

[67] Knuth, D.E., & McNeeley, J.L.,
SOL - A Symbolic Language for General Purpose System Simulation,
IEEE Transactions on Electronic Computers, 1964, pp. 401-414.

[68] Lazarev, V.G.,
On the Synthesis of Microprogramming Automats,
AD 674 217, 29th September 1967.

[69] Lustman, F.,
Simulation d'une Machine Digitalè a Partir d'une Description en Langage Cassandre,
R.I.R.O., Vol. 3, No. B-2, 1969, pp. 77-91.

[70] McCracken, D.D.,
Whither APL,
Datamation, 15th September 1970.

[71] McCurdy, B.D., & Chu, Y.,
Boolean Translation of a Macro Logic Design,
IEEE 1st Computer Conference, 6-8 September 1968, pp. 124-127.

[72] Mermet, J. & Lustman, F.,
CASSANDRE: Un Langage du Description de Machine Digitales,
Revue Francaise d'Informatique et de Recherche Operationelle,
Vol. 2, No. 15, 1968, pp. 3-35.

[73] Mesztenyi, C.R.,
Computer Design Language Simulation and Boolean Translation,
N-68-31483 (TR-68-72), June 1968, Maryland University.

[74] Metze, G., & Seshu, S.,
A Proposal for a Computer Compiler,
Proceedings of the Spring Joint Computer Conference, 1966,
pp. 253-263.

[75] Okada, Y., & Motooka, T.,
Logical Design Language,
Electronics & Communications in Japan, Vol. 50, No. 12, pp. 109-117.

[76] Parnas, D.L.,
A Language for Describing the Functions of Synchronous Systems,
Communications of the ACM, Vol. 9, No. 2, February 1966, pp. 72-77.

[77] Proctor, R.M.,
A Logic Design Translater Experiment Demonstrating Relationship of Languages to Systems,
IEEE Transactions on Electronic Computers, August 1964, pp 421-430.

[78] Reed, I.S.,
Symbolic Synthesis of Digital Computers,
Proceedings of the ACM. September 1952, pp. 90-94.

[79] Roth, J.P.,
Systematic Design of Automata,
Proceedings of the F.J.C.C. 1965, pp. 1096-1100.

[80] Salum, Kh.,
A Language for Describing and Modelling Digital Structural Systems (TSMIOD),
AD 673 811, 1967.

[81] Schlaeppi, H.P.,
A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS),
IEEE Transactions on Electronic Computers, August 1964, pp. 439-448.

[82] Schorr, H.,
Towards the Automatic Analysis and Synthesis of Digital Systems,
Ph.D. Dissertation, Princeton University, 1963.

[83] Schorr, H.,
Computer-Aided Digital System Design and Analysis using a
Register Transfer Language,
IEEE Transactions on Electronic Computers, Vol. EC-13, December
1964, pp. 730-737.

[84] Senzig, D.N.,
Suggested Timing Notation for the Iverson Notation,
IBM Report, 20th July 1962.

[85] Stabler, E.P.,
Microprogram Transformations,
IEEE Transactions on Computers, Vol. C-19, No. 10, October 1970,
pp. 908-916.

[86] Thurber, K.J. & Myrna, J.W.,
System Design of a Cellular APL Computer,
IEEE Transactions on Computers, Vol. C-19, No. 4, April 1970,
pp. 291-303.

[87] Wilber, J.A.,
A Language for Describing Digital Computers,
University of Illinois, Department of Computers Report 197,
February 1966.

[88] Barbacci, M., Bell, C.G., Newell, A.,
ISP: A Language to Describe Instruction Sets and Other Register Transfer Systems,
Digest of the Sixth Annual IEEE Computer Society International Conference, September 1972. p.p. 219-222

[89] Bell, C.G., Newell, A.,
The PMS and ISP Descriptive Systems for Computer Structures,
Proceedings of the Spring Joint Computer Conference, 1970.
p.p. 351-374

[90] Bell, C.G., Newell, A.,
Computer Structures, Readings and Examples,
McGraw Hill, 1970

[91] Bell, C.G., Knudsen, M., Siewiorek, D.,
PMS: A Notation to Describe Computer Structures,
Digest of the Sixth Annual IEEE Computer Society International Conference, 1972 p.p. 227-230

[92] Bell, C.G.,
Eggert, J.L., Grason, J., Williams, P.
The Description and Use of Register Transfer Modules,
(RTM's), Short Notes, IEEE Transactions on Computers,
May 1972 p.p.495-500

[93] Carroll, C.C., M tt, H.,
Procedures for the Automated Synthesis of Logical Networks
IEEE Transactions on Education, Vol. E-10 No.2 June 1967
p.p.77-81

[94] Gorman, D.F.,
Systems Level Design Automation: A Progress Report on the System Descriptive Language (SDL II)
<u>IEEE 1st Computer Conference</u>, 1968 p.p.131-134

[95] Gorman, D.F.,
<u>A Systems Descriptive Language and its Uses</u>,
Ph.D. Thesis, University of Pennsylvania, 1968

[96] Grason, J., Bell, C.G., Eggert, J.,
The Commercialization of Register Transfer Modules
<u>Computer</u> October 1973 p.p.23-27

[97] Holt, A.W., et al,
<u>Information System Theory Project</u>
Contract AF 30 (602) - 4211, Project 4594, Task 459403
22nd October 1968

[98] Johnson, R.R.,
Measures and Evaluations,
Transcript of Lecture given at Grenoble 1969 p.p.11, 22-39

[99] Patil, S.S., Dennis, J.B.,
The Description and Realization of Digital Systems,
<u>Digest of the Sixth Annual IEEE Computer Society International Conference</u>, September 1972 p.p.223-226

[100] Scheff, B.H., Kronstadt, E., Young, S.,
The Role of a Computer Machine Aids System in the Digital Design Process
<u>Joint Conference on Methematical and Computing Aids to Design</u>, 1969

[101] Sedlak, J.,
Language for Modelling Logical Sequential Circuits (SELOB),
Information Processing Machines No.14, 1968 p.p.193-212

[102] Smith, R.J.II, Tracely, J.H.,
A New Method for Sequential Circuit Specification,
Proceedings of the SW IEEESCO, p.p.458-462

[103] Spruel, A.H.,
The Construction of Digital Computers using Register
Transfer Modules, AD 733 201, Sept., 1971

[104] Srinivasan, C.V.,
CDL1: A Computer Description Language,
AD 693555, July 1969

[105] Srinivasan, C.V.,
Introduction to CDL1, A Computer Description Language,
AFCRL-67-0565, Report 1, September 1967

[106] Stabler, E.P.,
System Description Languages,
IEEE Transaction on Computers, Vol. C-19 No.12, p.p.1160-1173

[107] Varian, H., Kronstadt, E., Scheff, B.H., Young, S.,
RDDL: A Versatile Computer Design Language based on a
Precedence Grammar Compiler
Joint Conference on Mathematical and Computing Aids to
Design 1969

[108] Wendt, S.,

A Method for the Design of Synchronous Digital Hardware Systems,

Elektron Rechenanl, Vol.12, December 1970, p.p.314-323

[109] Wendt, S.,

On Structures of Micro Program Control Units,

Elektron Rechenanl, Vol.13, February 1971, p.p.22-26

APPENDIX I

A1-1

Several languages have been described in the preceding sections and a comparison made; however, it is felt that an example of each would be helpful to illustrate their differences. Strictly speaking an example should be included for each language, but in some cases where the differences are small it would be pointless to do so; also for ease of comparison the same example is used throughout.

Most of the languages only apply to digital computers and as such the example taken is a small, fictitious, 12 bits/word digital computer. It is not meant to be exhaustive of the capabilities of the languages but will be used to bring out any pertinent features. The block diagram of the computer and the instruction formats along with the instructions are shown in figs. A1 and A2 respectively. The description below, however, is limited to the multiplication algorithm only, and its flow diagram is depicted in fig. A3.

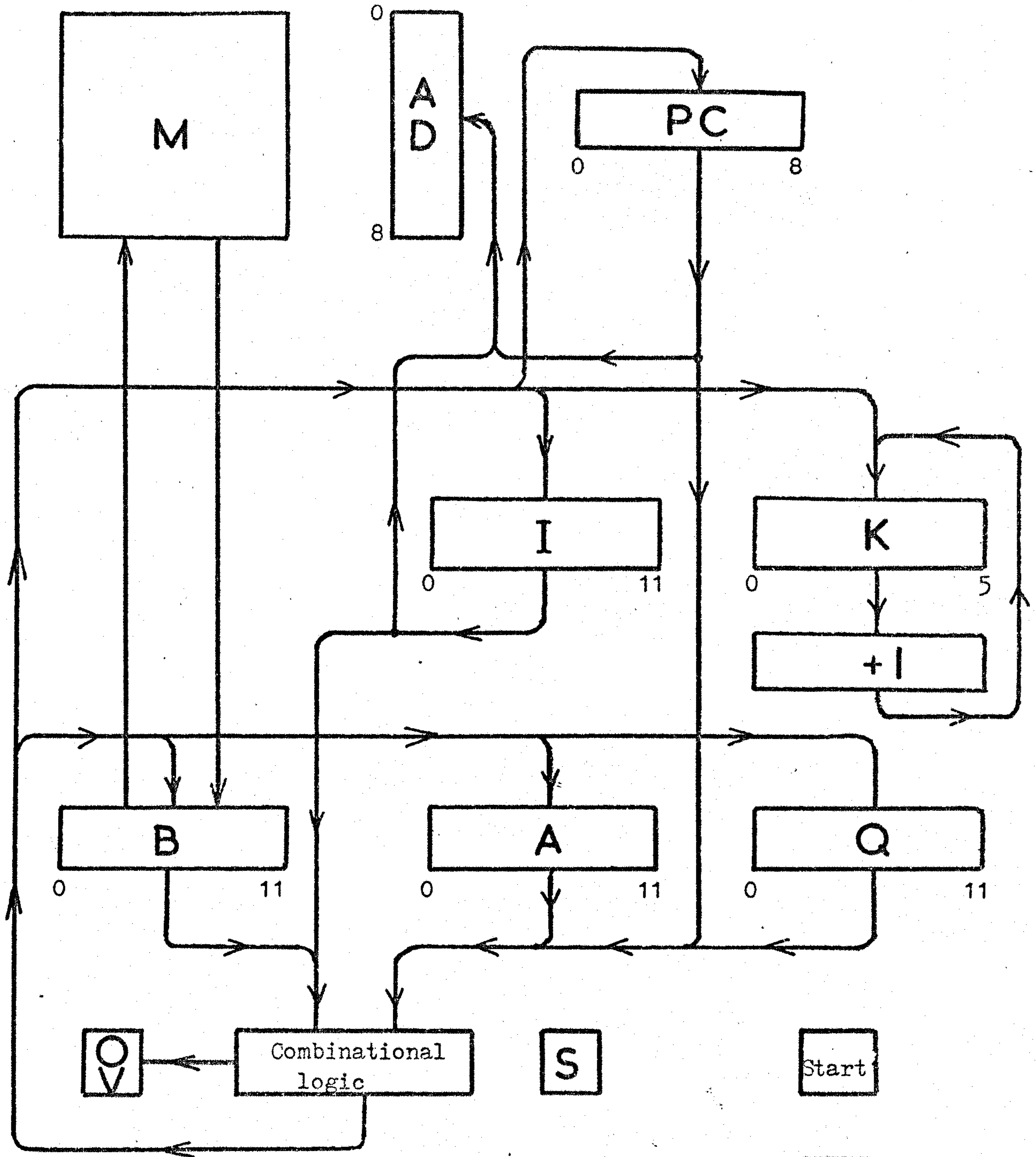
A1-2 Regular Expressions

The computer described here cannot easily be represented as a finite state machine and hence regular expression techniques cannot be applied. On the other hand if the multiplier was represented as a finite state machine then a description would be, albeit large, possible and this can be illustrated fairly simply.

Regular expressions essentially describe the valid sequences to produce an output; if the two twelve bits are available in parallel then the minimum sequence length is one and the input alphabet will consist of 2^{24} symbols and there will be 23 regular expressions for the 23 bits of the answer. This then becomes a straightforward table look-up method. On the other extreme, if the multiplier is a serial multiplier then the input alphabet will consist of two symbols only but the minimum sequence length will be 24.

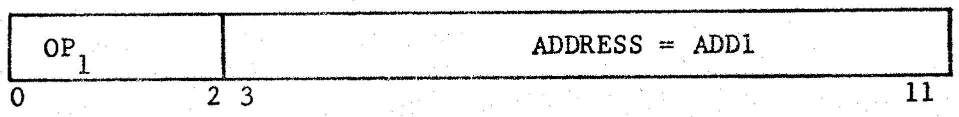
Since each output symbol can assume only one of two values, its regular expression will contain all the sequences of length n, where n is the length of the smallest sequence producing an output, which do produce the output as well as all the sequences of length n which do not produce an output followed by any of the sequences of length n producing an output. Hence, if $\underline{P}$ contains all the sequences producing an output but does not contain the star operator then the regular expression describing the machine is

$$\underline{R} = (\underline{P})^* \, \underline{P} \, ((\underline{P}')^* \, \underline{P})^*$$

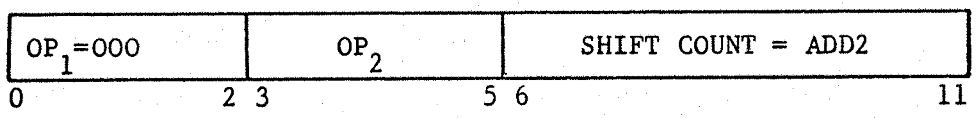


M : Memory 512 12-bit words, B : Memory buffer,
A : Accumulator, Q : Multiplier register,
I : Instruction register, K : Counter,
AD: Address register, PC: Programme counter,
OV: Overflow, S : Sign register.

The Register Structure of the Machine **FIGURE A-1**



Group A	Operation Code (OP_1)	Instruction
	001	Add to Accumulator
	010	Jump Unconditionally
	011	Jump if Accumulator zero or positive
	100	Store Accumulator
	101	Multiply Accumulator
	110	Load Accumulator Indirectly
	111	Decrement Store by 1.



Group B	OP_1 = 000	OP_2	
		000	Halt
		001	Clear Accumulator
		010	Complement Accumulator
		011	Spare
		100	Right Circulate
		101	Left Shift
		110	Right Circulate Double Length
		111	Left Shift Double Length

FIGURE A-2

The Order Code Formats

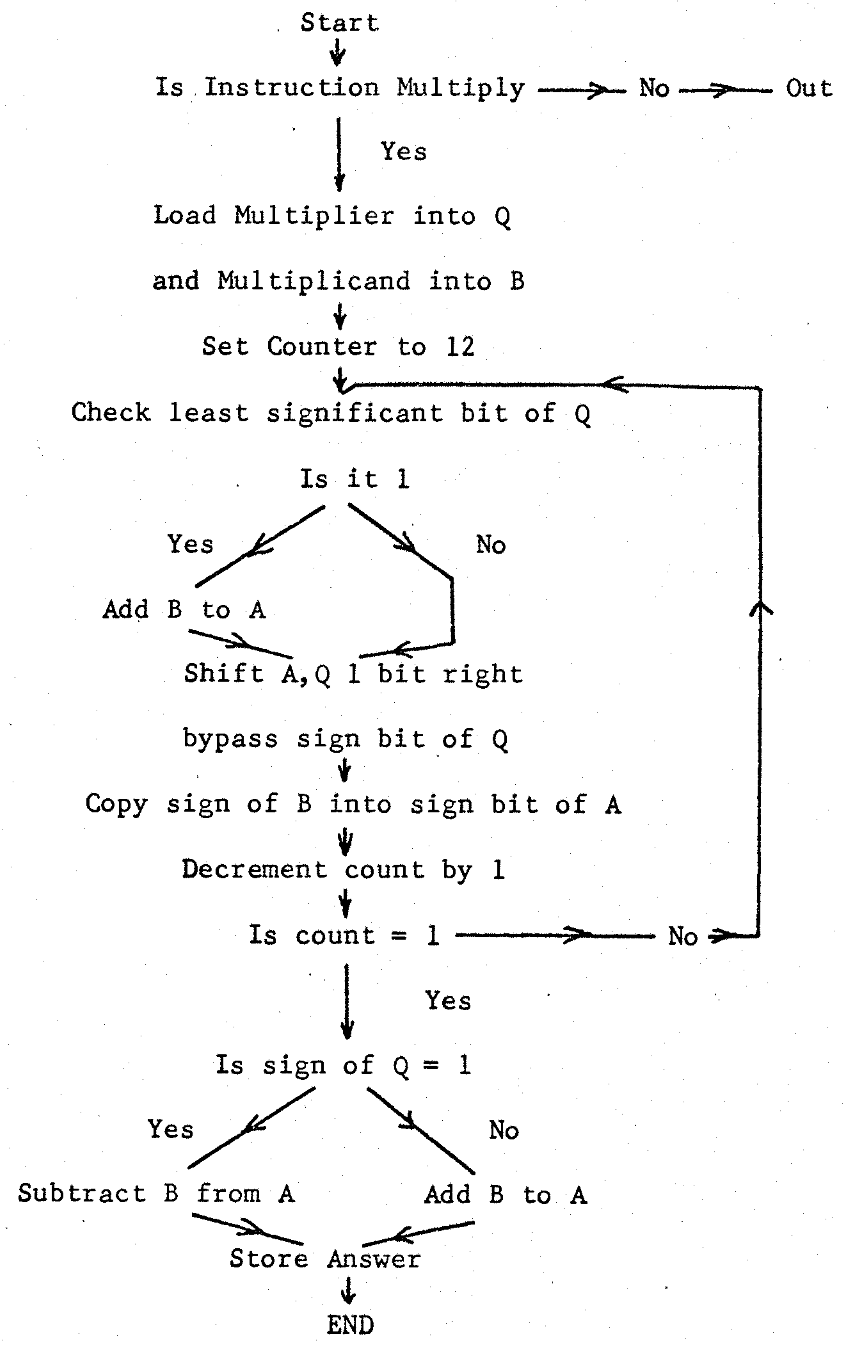


FIGURE A-3

The Multiplication Algorithm

This expression will realise a Moore machine and the corresponding Mealey machine is described by

$$\underline{R} = ((\underline{P}')^* \, \underline{P})^*$$

As an example the regular expressions for a two bit multiplier, without the sign bit, are obtained as follows. The expressions for each $\underline{P}$ derived from fig. A4 are

$$\begin{aligned}
\underline{P}_1 &= 0101 + 0111 + 1101 + 1111 \\
\underline{P}_2 &= 0110 + 0111 + 1001 + 1011 + 1101 + 1110 \\
\underline{P}_3 &= 1010 + 1011 + 1110 \\
\underline{P}_4 &= 1111
\end{aligned}$$

The state diagrams for each machine or a composite machine can be obtained using the techniques shown in section 2; however, to complete the illustration here, the state diagram for the Mealey machine corresponding to $\underline{P}_3$ is derived.

State	$\underline{R}_3 = (((\underline{P}_3)')^* \, \underline{P}_3)^*$	Output
1	$D_\lambda[\underline{R}_3] = \underline{R}_3$	Z = 0
2	$D_0[\underline{R}_3] = ((\emptyset)'(\underline{P}_3')^* \, \underline{P}_3 + \emptyset)^* \, \underline{R}_3$	Z = 0
3	$D_{00}[\underline{R}_3] = (00+01+10+11)(\underline{P}_3')^* \, \underline{P}_3\underline{R}_3$	Z = 0
3	$D_{01}[\underline{R}_3] = D_{00}[\underline{R}_3]$	Z = 0
4	$D_{000}[\underline{R}_3] = (0 + 1)(\underline{P}_3')^*\underline{P}_3\underline{R}_3$	Z = 0
4	$D_{001}[\underline{R}_3] = D_{000}[\underline{R}_3]$	Z = 0
5	$D_{0000}[\underline{R}_3] = (\underline{P}_3')^* \, \underline{P}_3\underline{R}_3$	Z = 0
5	$D_{0001}[\underline{R}_3] = D_{0000}[\underline{R}_3]$	Z = 0
6	$D_1[\underline{R}_3] = (010+011+110)'(\underline{P}_3')^*\underline{P}_3\underline{R}_3 + (010+011+110)\underline{R}_3$	Z = 0
7	$D_{10}[\underline{R}_3] = (10+11)'(\underline{P}_3')^* \, \underline{P}_3\underline{R}_3 + (10+11)\underline{R}_3$	Z = 0
8	$D_{11}[\underline{R}_3] = (10)'(\underline{P}_3')^* \, \underline{P}_3\underline{R}_3 + (10)\underline{R}_3$	Z = 0
4	$D_{100}[\underline{R}_3] = (0+1)(\underline{P}_3')^* \, \underline{P}_3\underline{R}_3 = D_{000}[\underline{R}_3]$	Z = 0
9	$D_{101}[\underline{R}_3] = (0+1) \, \underline{R}_3$	Z = 0
4	$D_{110}[\underline{R}_3] = (0+1)(\underline{P}_3')^*\underline{P}_3\underline{R}_3 = D_{000}[\underline{R}_3]$	Z = 0
10	$D_{111}[\underline{R}_3] = 1(\underline{P}_3')^*\underline{P}_3\underline{R}_3 + 0\underline{R}_3$	Z = 0
1	$D_{1010}[\underline{R}_3] = \underline{R}_3$	Z = 1
1	$D_{1011}[\underline{R}_3] = \underline{R}_3$	Z = 1
1	$D_{1110}[\underline{R}_3] = \underline{R}_3$	Z = 1
5	$D_{1111}[\underline{R}_3] = (\underline{P}_3')^*\underline{P}_3\underline{R}_3 = D_{0000}[\underline{R}_3]$	Z = 0
2	$D_{00000}[\underline{R}_3] = D_0[\underline{R}_3]$	Z = 0
6	$D_{00001}[\underline{R}_3] = D_1[\underline{R}_3]$	Z = 0

The corresponding state diagram is shown in figure A5.

Input Sequence	Outputs			
0 0 0 0	4	3	2	1
0 0 0 1	0	0	0	0
0 0 1 0	0	0	0	0
0 0 1 1	0	0	0	0
0 1 0 0	0	0	0	0
0 1 0 1	0	0	0	1
0 1 1 0	0	0	1	0
0 1 1 1	0	0	1	1
1 0 0 0	0	0	0	0
1 0 0 1	0	0	1	0
1 0 1 0	0	1	0	0
1 0 1 1	0	1	1	0
1 1 0 0	0	0	0	0
1 1 0 1	0	0	1	1
1 1 1 0	0	1	1	0
1 1 1 1	1	0	0	1

FIGURE A-4

Input/Output Behaviour for the Two-Bit Multiplier

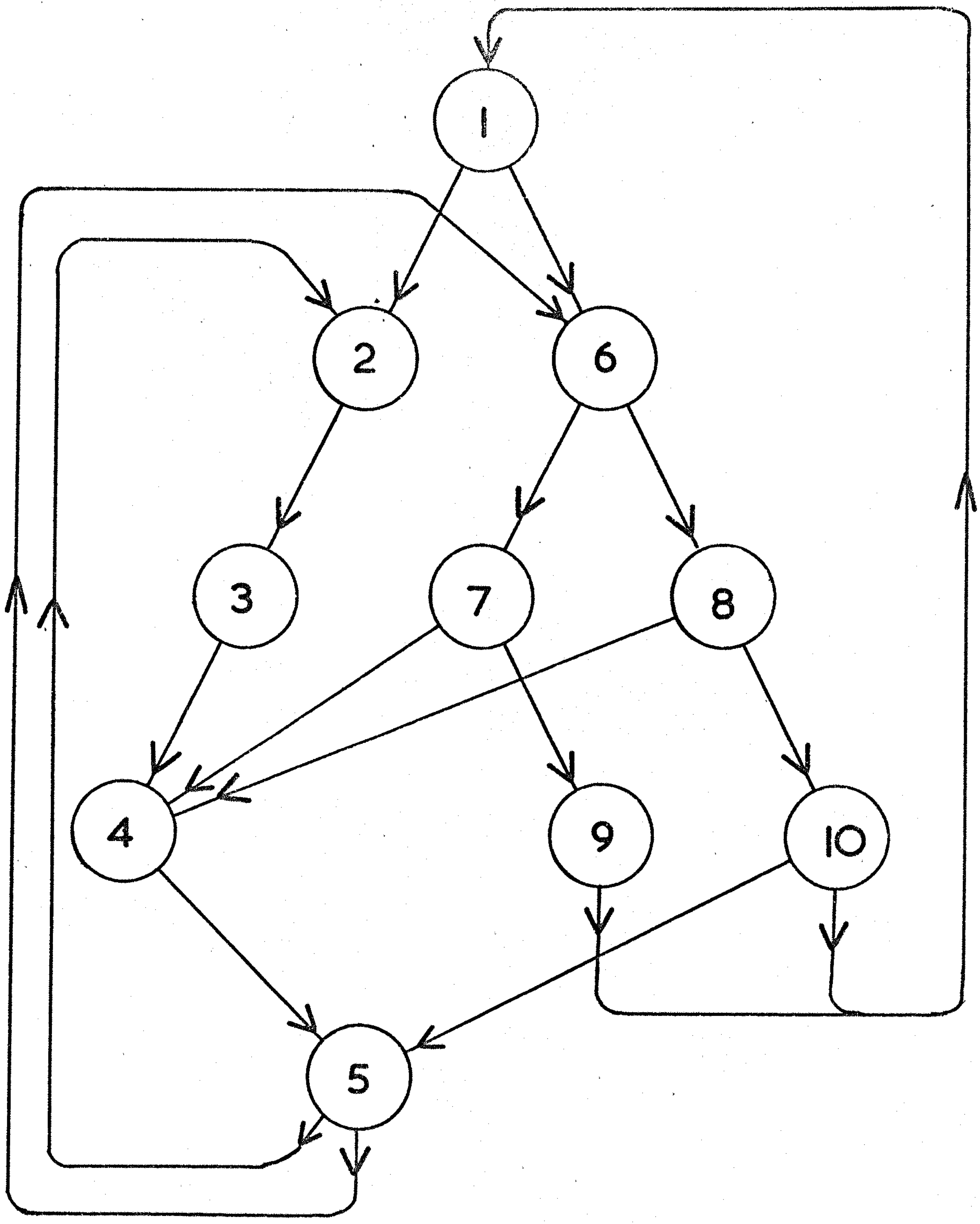


FIGURE A-5

STATE DIAGRAM FOR THE

TWO-BIT MULTIPLIER

A1-3 Reed - Schorr Language

Reed's language was basically an algorithm description language and had no formal declaration facilities; let the various registers be somehow declared in the programme as per fig. A-1. Schorr used a notation

$$(A) \qquad B \qquad\qquad \text{A1-3-1}$$

to mean that the contents of A were transferred to B and

$$(\langle A\rangle) \qquad B \qquad\qquad \text{A1-3-2}$$

to mean that the contents of the register specified by A were transferred to B. Here to avoid a large number of brackets, the brackets in a transfer of type A1-3-1 will be omitted as done by Reed and leave out the angular brackets from the second type of transfer. Thus the transfers A1-3-1 and A1-3-2 will be written as

$$A \rightarrow B \qquad\qquad \text{A1-3-3}$$

and

$$(A) \rightarrow B \qquad\qquad \text{A1-3-4}$$

respectively.

$\lvert \text{Start}.t_1 \rvert$	:	$1 \rightarrow t_2$	
$\lvert \text{Stop} \rvert$	:	$1 \rightarrow t_1$; stop	
$\lvert \text{Reset} \rvert$	:	$0 \rightarrow A;\ 0 \rightarrow B;\ 0 \rightarrow Q;\ 0 \rightarrow AD;\ 0 \rightarrow OC;$	
		$0 \rightarrow I;\ 0 \rightarrow K;\ 0 \rightarrow S;\ 0 \rightarrow OV;\ 1 \rightarrow t_2$	
$\lvert t_2 \rvert$	:	$PC \rightarrow AD;\ 1 \rightarrow t_3;$	
$\lvert t_3 \rvert$	:	$(AD{:}M) \rightarrow B;\ PC+1 \rightarrow PC;\ 1 \rightarrow t_4$	
$\lvert t_4 \rvert$	:	$B \rightarrow I;\ 1 \rightarrow t_5$	

Multiply

$\lvert t_5.OP.(0).OP.(1)'\circ OP.(2) \rvert$	:	$\rightarrow L$	
$\lvert L \rvert$	:	$ADD1 \rightarrow AD;\ A \rightarrow Q;\ 1 \rightarrow t_6$	
$\lvert t_6 \rvert$	:	$(AD{:}M) \rightarrow B;\ -11 \rightarrow K;\ 1 \rightarrow t_7$	
$\lvert t_7 Q(11) \rvert$	:	$B + A \rightarrow A;\quad 1 \rightarrow t_8$	
$\lvert t_7 Q(11)' \rvert$	:	$1 \rightarrow t_8$	
$\lvert t_8 \rvert$	:	$R_1(A,Q(1{:}11));\ B(0) \rightarrow A(0);$	$1 \rightarrow t_9$
$\lvert t_9 \rvert$	:	$K + 1 \rightarrow K;$	$1 \rightarrow t_{10}$
$\lvert t_{10}.K(0)$	:		$1 \rightarrow t_7$
$\lvert t_{10}.K(0)'.Q(0) \rvert$	:	$A - B \rightarrow A;$	$1 \rightarrow t_{11}$
$\lvert t_{10}\circ K(0)'Q(0)' \rvert$	:		$1 \rightarrow t_{11}$
$\lvert t_{11} \rvert$	:	$A \rightarrow B;$	$1 \rightarrow t_{12}$
$\lvert t_{12} \rvert$	:	$B \rightarrow (AD{:}M);\ ADD1 + 1 \rightarrow ADD1\ ;$	$1 \rightarrow t_{13}$
$\lvert t_{13} \rvert$	:	$Q \rightarrow B;\ ADD1 \rightarrow AD;$	$1 \rightarrow t_{14}$
$\lvert t_{14} \rvert$	:	$B \rightarrow (AD{:}M);$	$1 \rightarrow t_1$
$\lvert t_1 \rvert$	:	start next instruction	

The addition and subtraction operations are specified by the use of, what Schorr calls, a virtual register, which is used to represent the carry bits.

Hence the addition operation A + B $\rightarrow$ A is written as

$A(i) \oplus B(i) \oplus C(i) \rightarrow A(i) \qquad i = 0.1,\ldots,11$

$A(i).B(i) + A(i).C(i) + B(i).C(i) \rightarrow C(i-1)$

$0 \rightarrow C(11)$

A1-4 Schlaeppi's Language LOTIS

This is more a simulation language than a synthesis language and some figures for timing are introduced which are all in microseconds. Let the memory access time be 2 units and the cycle time 5 units.

```
CPU    DP/
M(9b,12); AD(12); B(12); I(12) = OP$_1$(3),ADD1(9) = OP$_1$(3), OP$_2$(3), ADD2(6)
K(5); A(12); Q(12); PC(9); OV(1); S(1); ready(1); +(2); -(2);
Comment The store cycle is asynchronous and when the cycle is finished
a Ready signal is produced.
fct Read, Memory/
1. 2: Ready : = 0; B : = M(AD)/
2. 3: Ready : = 1/fin
fct Store, Memory/
1. 2: Ready : = 0; M(AD) : = B/
2. 3: Ready : = 1/fin
seq begin, Control/
1. Start: if not (Stop or Reset) then call fetch else goto fin/
2. Stop : goto fin/
3. Reset: A,B,AD,I,Q,PC,OV,S,K : = 0/fin
seq fetch, Control/
1. Ready: AD : = PC; PC : = PC + 1/
2. Memory: call Read/
3. Ready : I : = B/
4. Goto CP/fin
seq Multiply, Arit/
1. AD : = ADD1; Q : = A; K : = -11/
2. Memory: call Read/
3. Ready : if (Q(11)=1) then A : = A + B else goto 4/
4. A : = (B(0),A(0,10)); Q : = (Q(0);A(11),Q(1,10));
   K : = K + 1/
5. if K(0)) then goto 3 else if (Q(0)) then A : = A - B/
6. B : = A/
7. Memory: call store/
8. Ready : ADD1 : = ADD1 + 1/
```

```
     9. AD : = ADD1; B : = Q/
     10 Memory: call Store/fin
A1-5 Language of Chu et al
     Register AD(0-11),                  £ Address register for the memory
              A (0-11),                  £ Accumulator Register
              B (0-11),                  £ Memory interface and arithmetic reg.
              I (0-11).                  £ Instruction Register
              Q (0-11),                  £ Multiplier Register
             PC (0-11)                   £ Instruction Counter
              K (0-4),                   £ Counter
             OV (1),                     £ Overflow Register
              S (1),                     £ Sign Register
     Sub-register OP1 (0-2)=I(0-2)       £ Group A Operation Code Bits
              OP2(0-2) =I(3-5)           £ Group B Operation Code Bits
              ADD1(0-8)=I(3-11)          £ Group A Address Bits
              ADD2(0-5)=I(6-11)          £ Group B Address Bits
     Memory   M (0-511, 0-11)            £ Main Memory
     Switch   Start
              Stop
              Reset
     Clock    T
    /Start/   T ← 1
    /Stop/    T ← 0
    /Reset/   AD ← 0, A ← 0, B ← 0, I ← 0, Q ← 0, K ← 0, OV ← 0,
              S ← 0, T ← 0,
     Sequence fetch
```

Comment begin when the clock has been set to 1 it automatically steps itself at the end of each step in the sequence unless reset at the end of an instruction or by external switches. end;

```
     /P1/        AD ← PC
     /P2/        B ← M(AD), PC ← PC ADD 1
     /P3/        I ← B, end of fetch sequence
     /P4.(OP1=5)/        AD ← ADD1, Q ← A, K ← -11
     /P5.(OP1=5)/        B ← M(AD),
     /P6.(OP1=5)/        do ADDM
     /P7.(OP1=5)/        do RIGHTSHIFT
     /P8.(OP1=5)/        if K ≠ 0 then P ← 6 else if Q(0) = 1 then
                         B ← A SUB B else B ← A
     comment begin ADD and SUB are addition and subtraction routines. end;
     /P9.(OP1=5)/        M(AD) ← B, ADD1 ← ADD1 ADD 1
     /P10.(OP1=5)/       AD ← ADD1, B ← Q
```

```
/P11.(OP1=5)/       M(AD) <- B, P <- 1
ADDM : if Q(11) = 1 then A <- A ADD B;
RIGHTSHIFT:  A&Q(1-11) <- shr B(0) & A & Q(1-10),
             if K ≠ 0 then K <- K ADD 1;
```

A1-6 Okada & Matooka

The language proposed by Okada and Motooka has five levels of descriptions; the 5th level corresponds to the algorithmic description and is quite similar to Chu's language. At level 4 the sequencing is shown more formally as is done for the multiplication sequence below.

Level 4

```
M1 : AD := I(3-11), Q := A, K :=-11;
M2 : B  := M(AD)   : M2(READY'), M3(READY);
M3 :               : M4 (Q(11)), M5(Q(11)');
M4 : A  := A + B;
M5 : A(0):=B(0), A(1-11):=A(0-10), Q(1-11):=A(11)&Q(1-10);
M6 : K  := K + 1
M7 :               : M3(K(0)), M8(K(0)');
M8 :               : M9(Q(0)), M10(Q(0)');
M9 : B  := A - B;
M10: M(AD) := B    :M10(READY'), M11(READY);
M11: I(3-11):= I(3-11) + 1;
M12: AD := I(3-11), B := Q;
M13: ,(AD) := B,   :M13(READY'), END(READY);
END;
```

At level 3 the sequencing is described with single unit timings and it is more explicit. Therefore operations such as additions have to be detailed. At level 2 the operations are shown as in level 3 but the sequencing is omitted and at level the interconnections of gates etc. along with declarations of delays of the gates are enumerated. For the present these are omitted from here.

A1-7 Metze and Seshu

```
C     Declaration of the name of the system
      MACHINE COMPUTER DP
C     Global Headers
      SYN (WL, 12), (DWL, 23), (AL, 9)
      PARALLEL (MEMORY, CP)
      OPTIMISE (SPEED)
C     These headers declare global quantities such the word length, WL,
C     Double word length, DWL, and Address Length, AL, as well as the modules
C     which can operate simultaneously and the criterion for optimality.
      MACRO READ (M, AD, B)
C     This declares the read routine of the main-memory which is assumed to have an
```

```
C      independent control within itself.  The control unit activates an access line
C      ACC and waits till a READY signal becomes true.
       CALL MEMORY (ACC)
       WAIT (READY=1)
       B= (AD)
       ENDC
       ENDM
       MACRO WRITE (M, AD, B)
       CALL MEMORY (ACC)
       WAIT (READY=1)
       (AD)=B
       ENDC
       ENDM
       CONTROL CP
C      Start Description of main computer
       REGISTER A(WL), B(WL), Q(WL), I(WL), PC(AL), AD(AL), K(4), OV(1) , S(1)
       EQUIV (OP1)= I(0,2)),(OP2 = I(3,5)), (ADD1 = I(3,11)),
     1 (ADD2=I(6,11))
       INTERFACE(MEMORY) AD,B
       DECODE(OP1) DEC,ADD,JMC,JMZ,STO,MPY,LA1,DS1
DEC    DECODE(OP2) HLT,CLA,COA,  ,RSC,LSS,RCD,LSD
C      Main Programme
MPY    AD= ADD1
       CALL READ
       Q=A
       A=B
       K=-11
L1     IF(Q(11)=0)L2
       .ADD(A,B,OV)
C      The prefix . requests a library routine
L2     Q(2,11)=Q(1,10)
       Q(1)=A(11)
       A(1,11)=A(0,10)
       A(0)=B(0)
       .ADD(K,1, )
       IF(K(0)=1) L1
       IF(Q(0)=0) L3
       .SUB(A,B, )
L3     B = A
       CALL WRITE
       .ADD(ADD1,1, )
```

```
      AD = ADD1
      B = Q
      CALL WRITE
      GOTO NEXT
C     This fetches the next instruction
      ENDM
```

A1-8 Duley and Dietmeyer DDL

In the description using DDL the system model is assumed to be a collection of automata normally functioning independently and communicating via a common highway. In our system let the memory unit be one automaton, switches and the cetral processor itself being the other automata Let the timing be a global variable and be controlled by the switches.

```
  <SY>      Computer
  <TE>      START, STOP, RESET, SW[1:3]
  <EL>      THREE SWITCHES (S[1:3]).
  <BO>      START = SW[1], STOP = SW[2], RESET = SW[3].
  <AU>      INTERLOCK
  <ST>      A0:⌈S[1]. S[2]. S[3].              → A0. |1 → A1.
                                 |0,3,5,6,7,
                   → A2.      → A3 ..
               |2          |4

            A1: SW[3]= 1..
            A2: SW[2]= 1..
            A3: SW[1]= 1...
  <AU>      CP:START:
  <RE>      A[0:11], B[0:11], Q[0:11], I[0:11], AD[0:8], PC[0:8], K[0:4],
            OV, S.
  <TI>      P(1E-6)
  <OP>      ADD(A,B)[0:11]
  <TE>      A[0:11], B[0:11], C[0:11], OV
  <BO>      ADD = A ⊕ B ⊕ C,
            C[0: 10]  = A[1:11].B[1:11]V(A[1:11] V B[1:11]).C[1:11],
            C[11]     = 0,
            OV        = A[0]. B[0]V(A[0]VB[0] ) .C[0]..
  <OP>      SUB (A,B)[0:11]
  <TE>      A[0:11], B[0:11], C[0:11], OV
  <BO>      SUB = A'⊕B ⊕ C,
            C[0:10]   = A[1:11]'.B[1:11]V(A[1:11]'V B[1:11]).C[1:11],
            C[11]     = 0,
            OV        = A[0]'.B[0]V(A[0]VB[0]).C[0]..
  <SEG>     FETCH
  <ST>      F0: AD ← PC,  ↑¢  PC, → F1
            F1 : |READY'| ⇒ MEMORY(READ=1) → F2;  → F1;
            F2 : |READY | I ← B, → F3; → F2..
```

```
   F3  : ⌈I[0:2]⌉₅ ⇒ MPY(⇒F0);  →F3...
  <CO>       The above step assumes that multiplication is the only instruction
             to be interpreted and the others cause a restart of instruction
             fetching sequence.
  <SEG>      MPY :
   M0  :     AD ← I[3:11], Q ← A, K ← 11D5, → M1.
   M1:       READY' :⇒ MEMORY (READ=1), →M2.
   M2  :     READY  : ↑¢  I[3:11],          →M3.
   M3  :     |Q[11]|  A ← ADD(A,B)          →M4;   →M4.
   M4  :↳    B[0]oA[0:11]oQ[1:11],  ↓¢  K,→M5;
   M5  :    |K=0 |→M6;   →M3.
   M6  :    |Q[0]| B ← SUB(A,B), →M7;  B ←  A,  →M7.
   M7  :     READY': ⇒ MEMORY(WRITE=1), →M8.
   M8  :     READY :  AD ←  I[3:11], B ← Q, →M9.
   M9  :     READY': ⇒ MEMORY(WRITE=1), → M10.
   M10:      READY : ⇒    ..
  <AU>       MEMORY:P:
  <EL>       MEM(RD[12], READY: READ,WRITE,WD[12], AD[9]).
  <RE>       AD[9], B[12],
  <DE>       DLY(2E-6).
  <ST>       L0 : |READ| → RD0;    |WRITE| →WR0; → L0.
             RD0 : READY ← 0, DLY = 1, →RD1.
             RD1 : B ← RD,  READY ← 1, ⇒    ..
             WR0 : READY ← 0,      →  WR1..
             WR1 : WD ← B,  DLY = 1,→WR2..
             WR2 : READY ← 1,  ⇒  ...
  .(END OF SY)
```

A1-9 Cassandra

This language is in many ways similar to DDL and is based on Algol.

```
UNIT     Computer (INPUT (0:11), START, STOP, RESET; OUTPUT(0:11))
         REGISTER A(0:11), B(0:11), Q(0:11), I(0-11), PC(0:8),
                  AD(0:8), OV(0:0), S(0:0), K(0:4);
         SIGNAL   READ, READY, START,STOP, RESET;
         EXTERNAL ADM(AD(0:9), B(0:11), READ; READY(1:1), OUT(0:11))
                  AD (A(0:11), B(0:11); C(0:11), OV(0:0));
         COMMENT  These external units are memory addressing and addition;
         CLOCK    P;
                  OUTPUT := A;
         S1:      <P>         AD ← OC;
         S2:      <P>         AD(PC, 1; PC, ), READ := 1;
```

```
          S3:     <P>        ADM(AD,  , READ; READY,B);
          S4:     <P>        IF READY THEN I ← B;
          S5:                IF OP(0:2) EQUAL 5 THEN GOTO MP ELSE GOTO S1;
          MP:     <P>        AD ← I(3:11);
         MP1:     <P>        ADM(AD ,  , READ; READY,B);
         MP2:     <P>        Q ← A, K ← -11;
         MP3:     <P>        IF Q(11) THEN AD(A,B;A,OV);
         MP4:     <P>        AD(K,1; K, );↓A(0:11)&Q(1:11), A(0) ← B(0);
         MP5:                IF K NOT EQUAL 0 THEN GOTO MP3;
         MP6;     <P>        IF Q(0) THEN AD(1,B';B, ) ELSE GOTO MP8;
  COMMENT  This complements B;
         MP7:     <P>        AD(A,B; A,OV);
         MP8:     <P>        B ← A, READ :=0;
         MP9:     <P>        ADM(AD,B,READ; READY, );
        MP10:     <P>        IF READY THEN AD(AD,1;AD, ), B ← Q;
        MP11;     <P>        ADM(AD,B,READ;READY, );
        MP12:     <P>        IF READY GOTO S1;
  END
  UNIT     ADM(P,AD(0:8), IN(0:11),READ; READY, OUT(0:11));
           REGISTER M(0:11,0:511);
           SIGNAL   AD(0:8), IN(0:11), READ(1:1), READY(1:1), OUT(0:11);
           PULSE    P; CLOCK P;
                    A1:     <P> READY:=0, IF READ NOT EQUAL 1 THEN M( ,⊥AD) ← IN
                                ELSE OUT ←  M(,⊥AD);
                    A2:     <P> READY:=1;
  END
  UNIT     AD(A(0:11), B(0:11), C(0:11), OV(1:1));
           SIGNAL   A(0:11), B(0:11), C(0:11), D(0:11), OV(1:1);
                    C := A⊻B⊻C;
                    C(1:11) & OV := A∧B∨(A∨B)∧C,
                    C(0) := 0;
  END
```

A1-10 Iverson

The Iverson notation is capable of describing algorithms only and has no formal declaration facilities for registers etc. Assume these are declared as in fig. A-1.

```
    1        start∧stop'∧reset'   : 1      (≠ , =)  →  (1,6)
    2        start                : 1      (≠ , =)  →  (2,6)
```

```
3    $\underline{stop}$   : 1                                   $(\neq , =) \rightarrow (4,3)$
4    $\underline{reset}$   : 1                                  $(\neq , =) \rightarrow (6,5)$
5    $\underline{a}, \underline{b}, \underline{q}, \underline{k}, \underline{ad}, \underline{i}, \underline{ov}, \underline{s} \leftarrow 0$              $\rightarrow 2$
6    $\underline{ad} \leftarrow \underline{pc}$
7    $\perp \underline{pc} \quad 2^{12} | (\perp \underline{pc} + 1)$
8    $\underline{b} \leftarrow \underline{M}^{\perp \underline{ad}}$
9    $\underline{i} \leftarrow \underline{b}$
10     $\perp (\alpha^{3}/\underline{i})$ : 5                        $\neq \rightarrow$ (other instructions)
11   $\underline{k} \leftarrow 2(5)\top 11$
12   $\underline{ad} \leftarrow \omega^{9}/\underline{i}$
13   $\underline{b} \leftarrow \underline{M}^{\perp \underline{ad}}$
14   $\underline{q} \leftarrow \underline{a}$
15   $\omega^{1}/\underline{q}$ : 1                                 $(\neq , =) \rightarrow (17,16)$
16   $\perp \underline{a} \quad 2^{12} | (\perp \underline{a} + \perp \underline{b})$
17   $\perp \underline{k} \leftarrow \perp \underline{k} - 1$
18   $\underline{a}, \omega^{11}/\underline{q} \leftarrow \alpha^{1}/\underline{b}, \underline{a}, (1 \downarrow \alpha^{10})/\underline{q}$
19   $\underline{k}$ : 0                                            $(\neq , =) \rightarrow (15,20)$
20   $\alpha^{1}/\underline{q}$ : 0                                 $(\neq , =) \rightarrow (21,22)$
21   $\perp \underline{a} \leftarrow \perp \underline{a} - \perp \underline{b}$
22   $\underline{M}^{\perp \underline{ad}} \leftarrow \underline{b}$
23   $\perp \omega^{9}/\underline{i} \leftarrow \perp \omega^{9}/\underline{i} + 1$
24   $\underline{ad} \leftarrow \omega^{9}/\underline{i}$
25   $\underline{b} \leftarrow \underline{q}$
26   $\underline{M}^{\perp \underline{ad}} \leftarrow \underline{b}$          $\rightarrow 3.$
```

Al-11 GERACE's method

Gerace's method converts register transfer type expressions to state tables, but this description must be written to indicate bit by bit operations. The multiplication algorithm, thus, should be written as follows.

The indices are $i = 1,2...,10.$ $j = 2,3,...,10.$ $m = 0,1,2,3.$

```
$|t_0|$   $(I(0)I(1)I(2):101)$                                                   $t_0 \rightarrow t_1;$
          $(I(0)I(1)I(2):101)$                                                   $t_0 \rightarrow t_5;$
$|t_1|.$        $A \rightarrow Q,\ M \rightarrow B,\ C \rightarrow K$                 $t_1 \rightarrow t_2;$
$|t_2|$   $(Q(11):1)$   $A(i) \oplus B(i) \oplus s(i+1) \rightarrow A(i),$
                        $A(11) \oplus B(11) \oplus [s(12)=0] \rightarrow A(11),$    $t_2 \rightarrow t_3;$
          $(Q(11):0)$                                                           $t_2 \rightarrow t_3;$
$|t_3|$                 $A(i-1) \rightarrow A(i),\ B(0) \rightarrow A(0),$
                        $A(11) \rightarrow Q(1),\ Q(j-1) \rightarrow Q(j),$
                        $K(m) \oplus r(m+1) \rightarrow K(m),$
                        $K(4) \oplus [r(5)=1] \rightarrow K(4),$                   $t_3 \rightarrow t_4;$
```

x	y	s(i+1)	s(i)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

x	r(i+1)	r(i)
0	0	0
0	1	0
1	0	0
1	1	1

Figure A-6

Definitions of the carry functions s(i) & r(i)

$|t_4|$ (K(0)Q(0):01) $A(i) \oplus B'(i) \oplus s(i+1) \rightarrow A(i)$,

. . . . . . $A(11) \oplus B'(11) \oplus [s(12)=1] \rightarrow A(11)$

$A(0) \oplus B'(0) \oplus s(1) \rightarrow A(0)$ $t_4 \rightarrow t_0$;

(K(0)Q(0):00) $t_4 \rightarrow t_0$;

(K(0) : 1) $t_4 \rightarrow t_2$;

Note C contains the constant -11, and the functions s and r are defined by the tables in fig. A-6.

There are four sets of machines corresponding to A,B,Q and K and it would be possible to derive the state tables for each separately; however, the machines corresponding to A and Q are clearly connected and the partition

$$\pi = \{\overline{A,Q}\ \ \overline{B}\ \ \overline{Q}\}$$

would be preferable. The first step is to identify the distinguishable submachines of each of the machines and list the transfers associated with them. There are four distinguishable sub-machines of the A,Q machine corresponding to bit 0, bit 1, bit j and bit 11 and their lists are as follows.

$(L_0)\overline{A,Q}$

a_1	$t_1 \rightarrow t_2$	$A(0) \rightarrow Q(0)$;
a_2	$(Q(11):1)t_2 \rightarrow t_3$	$A(0) \oplus B(0) \oplus s(1) \rightarrow A(0)$;
a_3	$t_3 \rightarrow t_4$	$B(0) \rightarrow A(0)$;
a_4	$(K(0)Q(0):01)\ t_4 \rightarrow t_0$	$A(0) \oplus B'(0) \oplus s(1) \rightarrow A(0)$;

$(L_1)\ \overline{A,Q}$

a_1	$t_1 \rightarrow t_2$	$A(1) \rightarrow Q(1)$;
a_2	$(Q(11):1)\ t_2 \rightarrow t_3$	$A(1) \oplus B(1) \oplus s(2) \rightarrow A(1)$;
a_3	$t_3 \rightarrow t_4$	$A(0) \rightarrow A(1)$;
		$A(11) \rightarrow Q(1)$;
a_4	$(K(0)Q(0):01)\ t_4 \rightarrow t_0$	$A(1) \oplus B'(1) \oplus s(2) \rightarrow A(1)$;

$(L_j)\ \overline{A,Q}$

a_1	$t_1 \rightarrow t_2$	$A(j) \rightarrow Q(j)$;
a_2	$(Q(11):1)\ t_2 \rightarrow t_3$	$A(j) \oplus B(j) \oplus s(j+1) \rightarrow A(j)$;
a_3	$t_3 \rightarrow t_4$	$A(j-1) \rightarrow A(j)$;
		$Q(j-1) \rightarrow Q(j)$;
a_4	$(K(0)Q(0):01)\ t_4 \rightarrow t_0$	$A(j) \oplus B'(j) \oplus s(j+1) \rightarrow A(j)$;

$(L_{11})\ \overline{A,Q}$

a_1	$t_1 \rightarrow t_2$	$A(11) \rightarrow Q(11);$
a_2	$(Q(11):1)\ t_2 \rightarrow t_3$	$A(11) \oplus B(11) \oplus [s(12)=0] \rightarrow A(11);$
a_3	$t_3 \rightarrow t_4$	$A(10) \rightarrow A(11);\ Q(10) \rightarrow Q(11);$
a_4	$(K(0)Q(0):01)\ t_4 \rightarrow t_0$	$A(11) \oplus B'(11) \oplus [s(12)=1] \rightarrow A(11);$

Similarly the lists for B and K are:

(L_n) B $\qquad n = 0,1,\ldots,11$

a_1	$t_1 \rightarrow t_2$	$M(n) \rightarrow B(n);$

(L_m) K

a_1	$t_1 \rightarrow t_2$	$C(m) \rightarrow K(m);$
a_3	$t_3 \rightarrow t_4$	$K(m) \oplus r(m+1) \rightarrow K(m);$

(L_4) K

a_3	$t_3 \rightarrow t_4$	$K(4) \oplus [r(5)=1] \rightarrow K(4);$

and finally

a_0	$(I(0)I(1)I(2):101)\ t_0 \rightarrow t_1$ $(I(0)I(1)I(2):101)'t_0 \rightarrow t_5$ $(Q(11):0)\ t_2 \rightarrow t_3$ $(K(0)Q(0):00)\ t_4 \rightarrow t_0$ $(K(0):1)\ t_4 \rightarrow t_2$	

From the listing above it is apparent that the inputs to the control part are Q(11), K(0), Q(0) and the three instruction bits I(0), I(1) and I(2); hence the state table in fig. A7 for the control unit can be derived. For the sake of simplicity some combinations of the inputs have been omitted as these do not provide any additional information.

The entries in this table correspond to the next state of the control unit and the outputs which initiate the transfers in the operational part; it is in an abstract form and can be synthesized in terms of hardware or software as necessary.

Seven different state tables have to be generated to specify the operational part completely; however since this example is for illustrative purposes only, the state table for A(0), Q(0) machine only will be derived here.

The external inputs to this machine are s(1), B(0) and the a outputs from the control unit; the present state variables y_1, y_2 replace A(0) and Q(0) respectively on the left hand side of the transfer expressions and Y_1, Y_2 similarly on the right hand side. The state table derived using the procedure described in the main text is shown in fig. A-8 In fig A-9 the corresponding output table is depicted. Finally the abstract state table including the output behaviour is shown in fig A-10.

Present states \ Inputs	Q(11)	Q'(11)	K(0)	K'(0)Q(0)	K'(0)Q'(0)	X	X'
t_0	-	-	-	-	-	t_1, a_0	t_5, a_0
t_1	$t_2 a_1$	t_2, a_1	t_2, a_1	t_2, a_1	t_2, a_1	t_2, a_1	t_2, a_1
t_2	t_3, a_2	t_3, a_0	-	-	-	-	-
t_3	t_4, a_3	t_4, a_3	t_4, a_3	t_4, a_3	t_4, a_3	t_4, a_3	t_4, a_3
t_4	-	-	t_2, a_0	t_0, a_4	t_0, a_0		

Figure A-7. The State Table for the Control Unit of the Multiplier

inputs			B(0)s(1)				B(0)		B(0)s(1)			
y_1y_2			00	01	11	10	0	1	00	01	11	10
00	00	00	00	10	00	10	00	10	10	00	10	00
01	01	00	01	11	01	11	01	11	11	01	11	01
11	11	11	11	01	11	01	01	11	01	11	01	11
10	10	11	10	00	10	00	00	10	00	10	00	10
	a_0	a_1	a_2				a_3		a_4			

Figure A-8, The assigned state table for the machine A(0)Q(0)

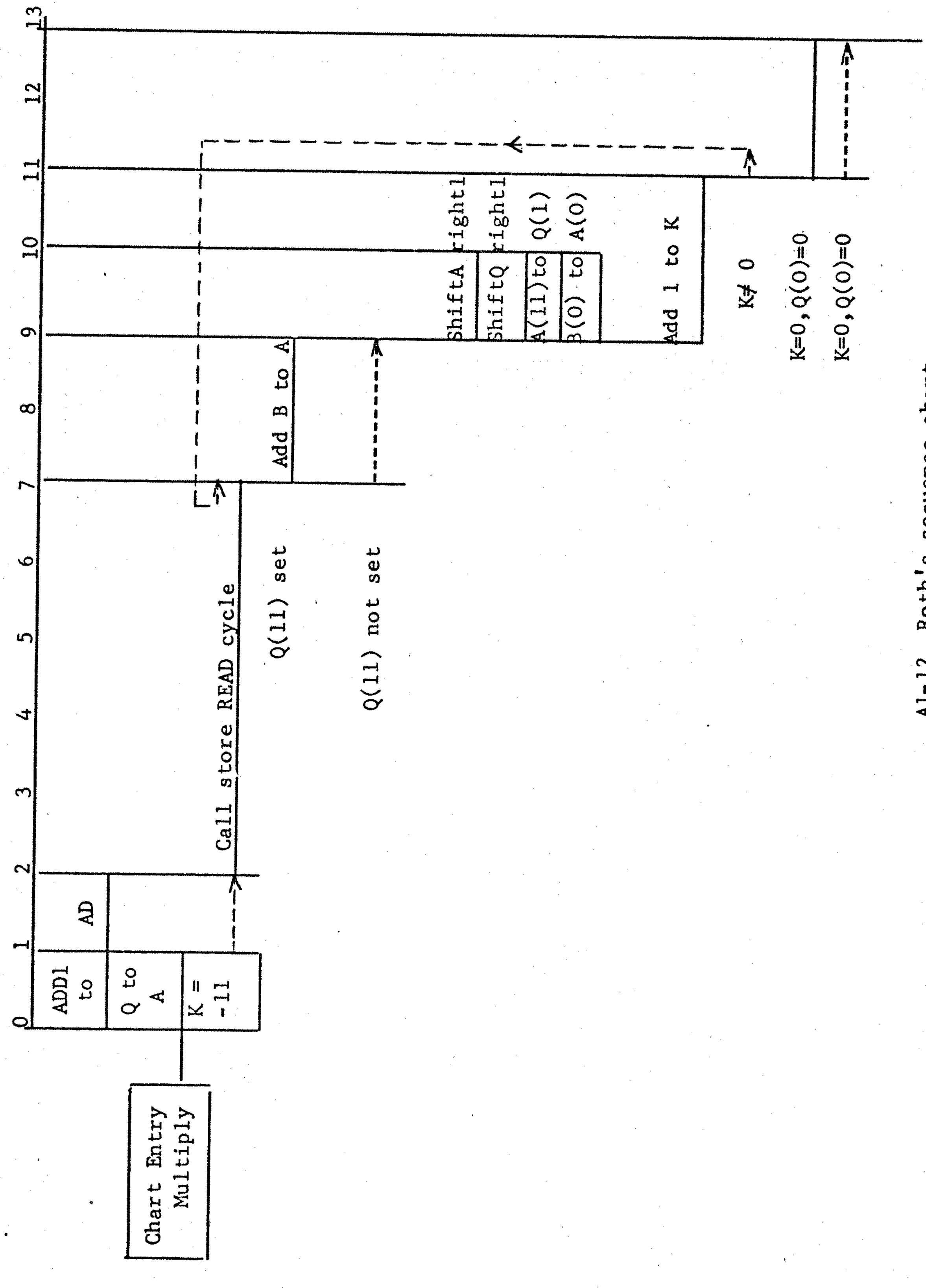
inputs			B(0)s(1)				B(0)		B(0)s(1)				
y_1y_2			00	01	11	10	0	1	00	01	11	10	A(0)Q(0)
00	-	-	0	0	1	0	-	-	0	1	1	1	00
01	-	-	0	0	1	0	-	-	0	1	1	1	01
11	-	-	0	1	1	1	-	-	0	0	1	0	11
10	-	-	0	1	1	1	-	-	0	0	1	0	10
	a_0	a_1	a_2				a_3		a_4				
			S(0)						S(0)				

Figure A-9, Output Table for the machine A(0)Q(0)

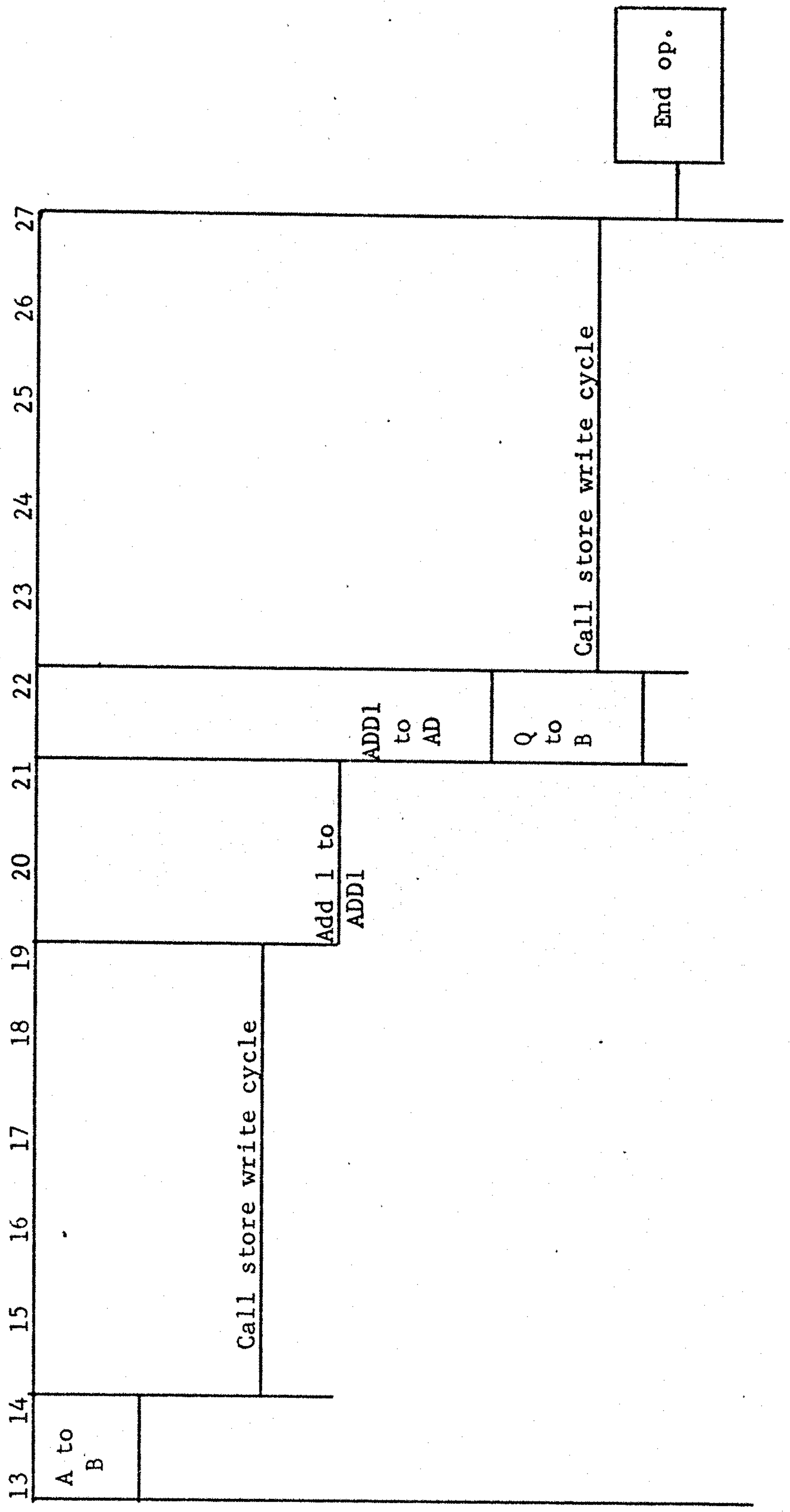
Present State \ inputs			B(0)s(1)				B(0)		B(0)s(1)				A(0)Q(0)
			00	01	11	10	0	1	00	01	11	10	
	a_0	a_1	a_2				a_3		a_4				
1	1,-	1,-	1,0	4,0	1,1	4,0	1,-	4,-	4,0	1,1	4,1	1,1	00
2	2,-	2,-	2,0	3,0	2,1	3,0	2,-	3,-	3,0	2,1	3,1	2,1	01
3	3,-	3,-	3,0	2,1	3,1	2,1	2,-	3,-	2,0	3,0	2,1	3,0	11
4	4,-,	4,-	4,0	1,1	4,1	1,1	1,-	4,-	1,0	4,0	1,1	4,0	10

Next state s(0)

Figure A-10, The state table for the machine A(0)Q(0)



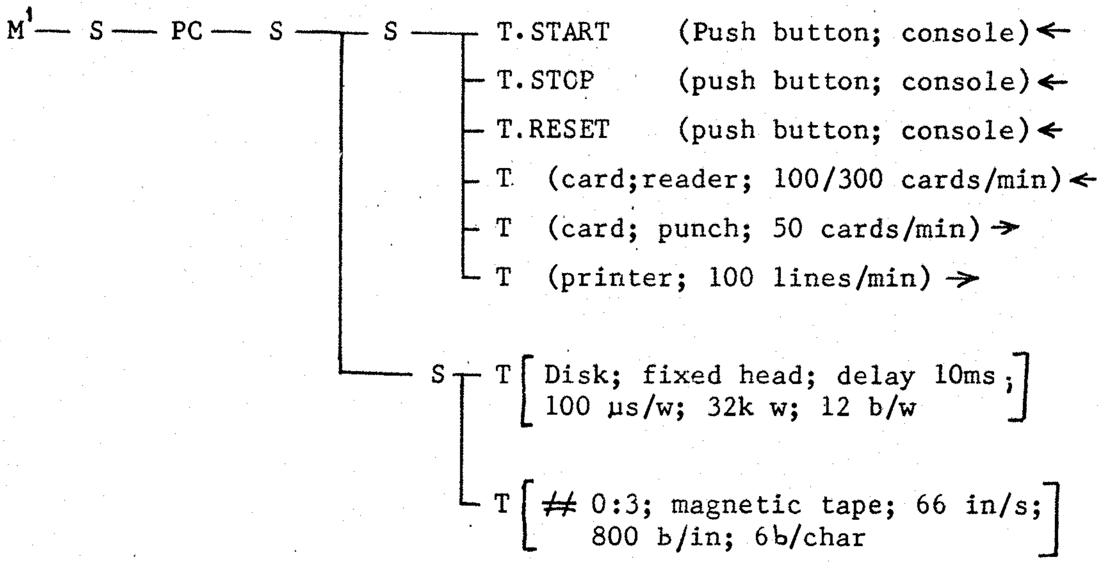
A1-12. Roth's sequence chart.



Al-12 (continued)

A1-13 PMS Level Description

In the machine considered here, only the register structure is shown. The PMS level description is more concerned with the way the system is configured. Let us therefore assume that there are two peripheral controllers on the system, first one handling some magnetic devices and the second one handling devices such as printers and card equipment.


M'
S
PC
S
S
T.START (Push button; console)
T.STOP (push button; console)
T.RESET (push button; console)
T (card;reader; 100/300 cards/min)
T (card; punch; 50 cards/min)
T (printer; 100 lines/min)
S
T [Disk; fixed head; delay 10ms; 100 µs/w; 32k w; 12 b/w]
T [# 0:3; magnetic tape; 66 in/s; 800 b/in; 6b/char]

A1-14 ISP Level Description=

Pc State

AD ⟨0:8⟩	memory address register
PC ⟨0:8⟩	program counter
I ⟨0:11⟩	Instruction register
K ⟨0:5⟩	Multiplication counter
B ⟨0:11⟩	Memory buffer
A ⟨0:11⟩	Accumulator
Q ⟨0:11⟩	Multiplier
OV	overflow registers
S	sign register

Mp state

M [0:511]⟨0:11⟩	main memory

Pc Console State

START	start switch
STOP	stop switch
RESET	reset switch

```
Instruction format
          OP ⟨0:2⟩        :=  ⟨0:2⟩        opcode
          MAD⟨0:8⟩        :=I ⟨3:11⟩       address
Start process
          START ∧¬ (STOP ∨ RESET) → fetch;
Fetch process
          fetch := (B ← M[AD] ; next I ← B;   PC ← PC + 1); → execute
execute process
          execute := (
          multiply (:= OP = 5)→(Q ← A; B ← M  MAD ; K ← 12; → Loop);
          Loop := (Q⟨11⟩ = 1 → A ← A + B;    next
                   A⟨0:11⟩□ Q ⟨1:11⟩ ← A⟨0:11⟩□ Q⟨1:11⟩ /2;
                   K ← K - 1; next K ≠ 1→ Loop;   K = 1 → fin);
          fin :=   (Q⟨0⟩ = 1 → A ← A + B;   Q ⟨0⟩ = 0 →A ← A - B; next
                    M [AD] ← A; next AD ← AD + 1; next  M[AD] ← Q)
                   ); → fetch
```

APPENDIX II

A2-1 The Hamming Code

The Hamming Code is a special form of parity checking and is used for single error correction. The number of check bits is determined by the number of data bits; if there are m data bits, k check bits will be required such that

$$2^k \geqslant m + k$$

and these check bits are placed in the positions corresponding to the powers of2, the lowest, 2^0, being the leftmost.

The 2^ith check bit is used as a parity check, for even parity, on those positions whose checking numbers contain a 1 in the 2^ith column. For example, the 2^0 check bit is used to check the parity of positions 1,3,5,7,..., the 2^1 check bit is used to check the parity of positions 2,3,6,7,10,11... and so on.

When error detecting and error correcting, if the check is successful, then a 0 is placed in the column corresponding to the check bit and a 1 if it fails. For single error correction, the bit in the position indicated by these check bits is inverted.

For example, consider a 4 bit data message 1011, which requires 3 check bits and the encoded message is 0110011. Let us suppose that during transmission bit four is inverted and the received message is 0111011. Applying a parity check to the positions 1,3,5,7, we get an even parity and therefore the check bit 0 is set to zero. Parity check on bits 2,3,6,7, is also successful and the check bit 1 is also set to zero. The final check, however, is unsuccessful and the check bit 3 is set to 1. Thus the bit corresponding to the position 100, i.e. bit 4 is in error. Therefore the corrected message is 0110011.

Appendix III

A) Proof that $\underline{X} = \underline{BA}^*$ is the solution of the equation

$$\underline{X} = \underline{XA} + \underline{B}, \lambda \notin A \qquad (1)$$

This proof was given in a theorem by Arden [4] and is reproduced below.

The fact that $\underline{X} = \underline{BA}^*$ is a solution of equation (1) can be verified by direct substitution, and we get

$$\begin{aligned} \underline{XA} + \underline{B} &= \underline{BA}^*\underline{A} + \underline{B} \\ &= \underline{B}(\underline{A}^*\underline{A} + \lambda) \\ &= \underline{BA}^* \\ &= X \end{aligned}$$

Now suppose $\underline{X} = \underline{BA}^*$ is not the only solution of equation (1) and there exists a solution $\underline{X} = \underline{BA}^* + \underline{C}$ such that $\underline{C} \cap \underline{BA}^* = \emptyset$

Then

$$\begin{aligned} \underline{XA} + \underline{B} &= (\underline{BA}^* + \underline{C})\underline{A} + \underline{B} \\ &= \underline{BA}^*\underline{A} + \underline{CA} + \underline{B} \\ &= \underline{B}(\underline{A}^*\underline{A} + \lambda) + \underline{CA} \\ &= \underline{BA}^* + \underline{CA} \end{aligned}$$

$$\text{but} \quad \underline{XA} + \underline{B} = \underline{X} = \underline{BA}^* + \underline{C}$$

$$\text{therefore} \quad \underline{BA}^* + \underline{C} = \underline{BA}^* + \underline{CA} \qquad (2)$$

Intersecting both sides of equation (2) by $\underline{C}$ we get

$$\underline{BA}^* \cap \underline{C} + \underline{C} \cap \underline{C} = \underline{BA}^* \cap \underline{C} + \underline{CA} \cap \underline{C}$$

$$\text{therefore} \quad \underline{C} = \underline{CA} \cap \underline{C}$$

$$\text{implying} \quad \underline{C} \subset \underline{CA}$$

But since the assumption is that $\underline{A}$ does not contain λ, the shortest sequence of $\underline{CA}$ must be longer than the shortest sequence of $\underline{C}$ unless $\underline{C}$ is empty, thence $\underline{C} \not\subset \underline{CA}$. Therefore $\underline{X} = \underline{BA}^* + \underline{C}$ is not a solution of equation (1), and since this is true for all cases at $\underline{C}$ when $\underline{C}$ and $\underline{BA}$* are disjoint, the only solution of equation (1) is $\underline{X} = \underline{BA}$*.

B) Proof that $\underline{X} = \underline{A}^*\underline{B}$ is the solution of the equation

$$\underline{X} = \underline{AX} + \underline{B} \; , \lambda \notin \underline{A} \qquad (3)$$

This proof follows from an identical procedure used in the last proof.

APPENDIX IV

Proof copy of

Analysis of Sequential Logic Circuits

to be published in

Computer Journal, February, 1974.

ANALYSIS OF SEQUENTIAL LOGIC CIRCUITS

D. Pai* B.Sc.

and

Professor Douglas Lewin** B.Tech., M.Sc., C.Eng., M.I.E.R.E.

* Burrough Machines Ltd., formerly
Dept. of Electronics, Southampton University.

** Dept. of Electrical Engineering and Electronics,
Brunel University.

Index Terms

Feedback loops, secondary variables, asynchronous sequential circuits, logic circuit analysis.

Abstract

In the analysis and simulation of sequential circuits, and in particular asynchronous sequential circuits, the automatic location of feedback loops within the network often presents serious problems.

This paper presents an algorithm, based on an analytical approach, which will isolate the true feedback loops in a network, that is those paths which correspond to the actual secondary variables of the circuit.

1. Introduction

A logic circuit can usually be defined in a formal mathematical manner using truth-tables, state or flow-tables or some such model [1]. An abstract definition of this type is often used in digital systems design, for example:

(a) for the economical implementation and re-configuring of circuits;

(b) to obtain a true logical simulation;

(c) to enable fault testing and diagnosis procedures to be evaluated;

(d) for the concise documentation of logic circuits, etc.

Often, however, especially if the circuit has been designed intuitively, this type of description is not available; the circuit then has to be analysed in order to derive a formal model.

The problem of analysing cominational circuits (in order, for example, to generate a truth-table) is relatively simple, and can be solved by using conventional simulation techniques or by tracing the paths between the inputs and the outputs. When analysing sequential circuits, however, the presence of feedback loops in the network means that these techniques are no longer applicable. The normal method of proceeding in these cases is to isolate the feedback loop in some manner (often intuitively) and then apply the standard combinational techniques. In the case of clocked sequential networks or relay circuits the problem is trivial, since the feedback loops are clearly distinguishable. The/

The major problem lies with asynchronous networks, that is, circuits containing interconnected NAND or NOR gates, and it is this aspect of analysis which is considered in this paper.

Sequential circuits can be divided into two main categories (i) synchronous and (ii) asynchronous. Synchronous circuits are characterised by the fact that in the absence of a sampling signal, i.e. the clock signal, changes in the inputs do not alter the internal state of the circuits (although of course the outputs may change. To achieve this, storage elements (bistables) with pre-defined feedback loops (i.e. secondary variables) are employed in the circuit and driven by combinational logic[1]; thus all the feedback loops are consequently restricted to these storage elements. The analysis of synchronous sequential circuits therefore reduces to an analysis of combinational circuits and is a straightforward procedure[2].

Asynchronous circuits in many cases are implemented using relays which act as the storage elements. The analysis of these circuits is similar/

---

1. The outputs of the storage elements may be fed back to the inputs of the storage elements. In this case these storage elements are such that the outputs do not change during the presence of the clock signal; hence, for the purposes of analysis they may be considered as independent variables and the circuit feedback-free. If the outputs do change during the presence of the clock pulse the circuit will malfunction.

2. The algorithm to be described in this paper is equally applicable to combinational circuits.

/similar to the analysis of synchronous sequential circuits and it is only necessary to derive the excitation equations for the combinational circuits driving the relay coils.

In the more usual case however, when the circuit is implemented using standard logic modules (such as NAND gates), the feedback loops are not so clearly defined. The method adopted so far [1], [2], [3], [4] is to assume a feedback loop, break this loop and through simulation find out if it is possible to fully define the behaviour of the circuit. This method, though usable, is not algorithmic and does not lend itself to computer programming for automatic analysis.

In this paper we present a more systematic approach for locating these feedback loops and hence the secondary variables.

2. Algorithm

The analysis of asynchronous sequential circuits involves

(i) detecting the feedback loops, and

(ii) selecting only those feedback loops which correspond to the secondary variables.

Before we proceed with the description of the algorithm let us examine the condition implied in the second step. If

$$\underline{Y} = \{\text{set of all the secondary variables}\}$$

then/

then for all i if y_i is the value of the ith secondary variable at time t and if Y_i is the value of the same variable at time $t+\delta t$ where δt is a function of the logic delays then it is a necessary condition [1] that

$$Y_i = f_i(y_i) \qquad \ldots\ldots\ldots\ldots 2.1$$

such that $f_i(y_i)$ contains at least one positive y_i term and that this term is not redundant. If this condition is not met y_i is redundant and the corresponding loop can be removed. We shall not concern ourselves with the proof of this statement which can be found in [1].

The behaviour of a general logic circuit can be expressed as

$$Z_j = g_j(\underline{X},\underline{y}) \qquad \ldots\ldots\ldots\ldots 2.2$$

where Z_j is the jth variable in the set $\underline{Z}$, the set of all outputs, and $\underline{X}$ is the set of all inputs; if the circuit is combinational then the set $\underline{y}$ is empty. In the algorithm described below the circuit being analysed is assumed to be combinational until found otherwise.

The algorithm requires a topological description of the circuit in which each gate is defined in terms of its inputs, output[3] and the function [3]. It is also necessary to distinguish the external/

---

3. It is assumed that each gate produces only one output. If gates generating multiple outputs, e.g. ECL gates with complementary outputs, are employed, then each of these outputs must be specified by a separate gate with identical inputs but with different functions and different outputs. If wired functions are used it is necessary to also specify these by additional gates with wired outputs acting as inputs to these gates and their outputs feeding the next stages.

/external inputs and outputs, i.e. through which the circuit is accessed, from the connections internal to the circuit. A convention adopted here is to label outputs by Z_j, inputs by X_k and the internal connections by C_n where j, k and n are all integers. Thus for a circuit containing k inputs, j outputs and n internal connections the description is given as

$$Z_a = f_a(\underline{X},\underline{C},\underline{Z}) \qquad a = 1,2,\ldots,j$$

$$\text{and} \quad C_b = f_b(\underline{X},\underline{C},\underline{Z}) \qquad b = 1,2,\ldots,n$$

where $\underline{C}$ is the set of all internal connections.

In the following discussion we shall refer to the inputs, internal connections and the outputs by X-types, C-types and Z-types respectively.

The algorithm is based on tracing the logic path of a Z-type backwards, i.e. towards the inputs, so as to finially obtain an equation for Z in terms of $\underline{X}$ and the secondary variables (if any), only. Thus, starting from the topological description of a circuit, the terms in an output equation Z are expanded (unless it is a primary input) by substituting the inputs of the corresponding gate which generates that term; we shall call the equation produced in this way a Z-equation.

Further/

Further substitutions are made in successive passes for each of the C-types and Z-types in the Z-equation. Clearly this will either lead to a Z-equation in terms of X-types only, or feedback loops will be encountered; in the latter case the process will never terminate. During the iteration process a note is kept in a list, called the C-list, of each C-type and Z-type encountered during the substitutions. The presence of a feedback loop is detected by noting if during the iteration the Z-equation contains a C-type or a Z-type for which a substitution was already made in the previous iteration(s), since this implies that the particular <u>signal is a function of itself</u>. Any variables which are detected in this way are entered into a feedback variable list, the F.V. list.

If during a pass one or more new F.V's are detected then the C-list and the Z-equation so far generated are deleted and the prcedure restarted with the modification that substitutions are not allowed for any variables contained in the F.V. list (except when it is necessary to obtain an initial equation) and that these variables are not entered into the C-list. The sequence is repeated until all the feedback variables between the Z and the inputs are located and a Z-equation is obtained in terms of $\underline{X}$ and the feedback variables only.

At the conclusion of the algorithm the F.V. list contains those variables which re-occurred after an initial substitution was made, thereby implying that feedback loops may be present. However, it is necessary to ascertain that all the variables in the F.V. list do in fact correspond to loops (and hence to secondary variables) that is their characterising equations must satisfy the condition specified in 2.1. The next step therefore is to obtain an excitation equation for each F.V. and the procedure for this is identical to that used to obtain a Z-equation. The resulting equation is checked to see that condition 2.1 is met. If the condition is not met then the corresponding variable is deleted from the F.V. list and the whole procedure restarted.

It/

It then only remains to apply this procedure to the remaining Z-types and any other feedback variables that are detected. The final F.V. list corresponds to the list of secondary variables, the equations for the F.V.s to the excitation equations and the Z-equations to the output equations. The flow diagram for the above procedure is depicted in figure 1.

The output equations and the excitation equations obtained from the algorithm completely define the asynchronous circuit, and may be expended to generate the flow tables.

3. Examples

The above procedure will now be illustrated through a number of examples. First we will consider a Texas Instrument D-type bistable the circuit for which is given in figure 2 and the corresponding topological description in table 1[4].

Let us start by taking Z_1.

C-list	F.V.list	Equation
Z_1	-	$Z_1 = \overline{X_1} + \overline{C_2} + \overline{Z_2}$
Z_1, C_2, Z_2	-	$= \overline{X_1} + X_2 \cdot C_1 \cdot X_3 + X_2 \cdot C_3 \cdot Z_1$

Now Z_1 is already in the C-list; hence we add Z_1 to the F.V.list.

-	Z_1	$Z_1 = \overline{X_1} + \overline{C_2} + \overline{Z_2}$
C_2, Z_2	Z_1	$= \overline{X_1} + C_1 \cdot X_2 \cdot X_3 + X_2 \cdot C_3 \cdot Z_1$
C_2, Z_2, C_1, C_3	Z_1	$= X_1 + (\overline{X_1} + \overline{C_4} + \overline{C_2}) \cdot X_2 \cdot X_3$
		$+ (\overline{C_2} + \overline{X_3} + \overline{C_4}) \cdot X_2 \cdot Z_1$

C_2 is therefore added to the F.V.list.

-	Z_1, C_2	$Z_1 = \overline{X_1} + \overline{C_2} + \overline{Z_2}$
Z_2	Z_1, C_2	$= \overline{X_1} + \overline{C_2} + C_3 \cdot X_2 \cdot Z_1$
Z_2, C_3	Z_1, C_2	$= \overline{X_1} + \overline{C_2} + (\overline{C_2} + \overline{X_3} + \overline{C_4}) \cdot X_2 \cdot Z_1$
Z_2, C_3, C_4	$Z_1, C_2,$	$= \overline{X_1} + \overline{C_2} + (\overline{C_2} + \overline{X_3} + X_2 \cdot X_4 \cdot C_3) \cdot X_2 \cdot Z_1$

C_3 is added to the F.V.list.

-	Z_1, C_2, C_3	$Z_1 = \overline{X_1} + \overline{C_2} + \overline{Z_2}$
Z_2	Z_1, C_2, C_3	$= \overline{X_1} + \overline{C_2} + X_2 \cdot C_3 \cdot Z_1$ ....(3·1)

Applying the procedure to C_2 and C_3 we get

-	Z_1, C_2, C_3	$C_2 = \overline{C_1} + \overline{X_2} + \overline{X_3}$
C_1	Z_1, C_2, C_3	$= X_1 \cdot C_4 \cdot C_2 + \overline{X_2} + \overline{X_3}$

---

4. The algorithm has already been programmed. The inputs to this programare in Polish form; however, a standard form is used here for illustration purposes.

C-list	F.V.list	Equation	
C_1,C_4	Z_1,C_2,C_3	$= X_1\cdot(\overline{X}_4 + \overline{X}_2 + \overline{C}_3)\cdot C_2 + \overline{X}_2 + \overline{X}_3$	......(3·2)
-	Z_1,C_2,C_3	$C_3 = \overline{C}_2 + \overline{X}_3 + \overline{C}_4$	
C_4	Z_1,C_2,C_3	$= \overline{C}_2 + \overline{X}_3 + X_4\cdot X_2\cdot C_3$	..........(3·3)

and finally

-	Z_1,C_2,C_3	$Z_2 = \overline{X}_2 + \overline{C}_3 + \overline{Z}_1$	..........(3·4)

Z_1, C_2 and C_3 are the secondary variables and equations 3·1,3·2 and 3·3 represent the corresponding excitation equations. The output equation for Z_2 is given in 3·4 and since Z_1 is an output as well as a secondary variable a dummy output equation is generated for Z_1, i.e.:

$$Z_1 = Z_1 \qquad \text{..........(3·5)}$$

<u>Example 2.</u>

Consider the circuit given in figure 3 the topological description for which is given in table 2.

Starting with Z_1 we get

C-list	F.V.list	Equation
Z_1	-	$Z_1 = \overline{X}_1 + \overline{C}_3$
Z_1,C_3	-	$= \overline{X}_1 + C_2\cdot Z_1\cdot C_4$
-	Z_1	$Z_1 = \overline{X}_1 + \overline{C}_3$
C_3	Z_1	$= \overline{X}_1 + C_2\cdot Z_1\cdot C_4$
C_2,C_3,C_4	Z_1	$= \overline{X}_1 + (\overline{C}_1 + \overline{X}_2)\cdot Z_1\cdot(\overline{C}_3 + \overline{X}_2)$
-	Z_1,C_3	$Z_1 = \overline{X}_1 + \overline{C}_3$

Now, Z_1 is in the F.V.list and the substitution for it is thus complete; however, since:

$$Z_1 \neq g(Z_1).$$

Z_1 is therefore removed from the F.V.list.

Z_1	C_3	$Z_1 = \overline{X}_1 + \overline{C}_3$	..........(3·6)
-	C_3	$C_3 = \overline{C}_2 + \overline{Z}_1 + \overline{C}_4$	

C-list	F.V.list	Equation
C_2, Z_1, C_4	C_3	$= C_1\cdot X_2 + X_1\cdot C_3 + X_2\cdot C_3$
C_2, Z_1, C_4, C_1	C_3	$= \overline{X}_1\cdot X_2 + X_1\cdot C_3 + X_2\cdot C_3$ ....(3·7)

Therefore C_3 is the only secondary variable, and the corresponding excitation and output equations are given by 3·7 and 3·6 respectively.

Example 3.

We finally consider a circuit which Unger [5] has analysed by identifying and breaking feedback loops using a trial and error process. The circuit and the topological description are given in figure 4 and table 3 respectively.

C-list	F.V.list	Equation
Z	-	$Z = \overline{C}_6$
Z, C_6	-	$= X_1\cdot C_{10}$
Z, C_6, C_{10}	-	$= X_1\cdot(\overline{C}_3+\overline{C}_4+\overline{C}_5+\overline{C}_6)$
Z	C_6	$Z = \overline{C}_6$
-	C_6	$C_6 = \overline{X}_1+\overline{C}_{10}$
C_{10}	C_6	$= \overline{X}_1+C_3\cdot C_4\cdot C_5\cdot C_6$
C_{10}, C_3, C_4, C_5	C_6	$= \overline{X}_1+(\overline{C}_7+\overline{C}_9)\cdot(\overline{X}_2+\overline{C}_7)\cdot(\overline{C}_{10}+\overline{C}_9)\cdot C_6$
-	C_6, C_{10}	$C_6 = \overline{X}_1+\overline{C}_{10}$

C_6 is removed from the F.V.list. The equation for Z now reads

$$Z = X_1\cdot C_{10} \qquad \text{..........(3·8)}$$

Restarting the substitution process for C_{10} we get

-	C_{10}	$C_{10} = \overline{C}_3+\overline{C}_4+\overline{C}_5+\overline{C}_6$
C_3, C_4, C_5, C_6	C_{10}	$= C_7\cdot C_9+X_2\cdot C_7+C_{10}\cdot C_9+X_1\cdot C_{10}$
$C_3, C_4, C_5, C_6, C_7, C_9$	C_{10}	$= \overline{X}_1\cdot(\overline{C}_1+\overline{C}_2+\overline{C}_3+\overline{C}_4)+X_2\cdot\overline{X}_1+ C_{10}\cdot(\overline{C}_1+\overline{C}_2+\overline{C}_3+\overline{C}_4)+X_1\cdot C_{10}$
-	C_{10}, C_3, C_4	$C_{10} = \overline{C}_3+\overline{C}_4+\overline{C}_5+\overline{C}_6$
C_5, C_6	C_{10}, C_3, C_4	$= \overline{C}_3+\overline{C}_4+C_{10}\cdot C_9+X_1\cdot C_{10}$
C_5, C_9, C_6	C_{10}, C_3, C_4	$= \overline{C}_3+\overline{C}_4+C_{10}\cdot(C_1+C_2+C_3+C_4)+X_1\cdot C_{10}$
C_5, C_9, C_1, C_2, C_6	C_{10}, C_3, C_4	$= \overline{C}_3+\overline{C}_4+C_{10}\cdot(C_9\cdot C_8+X_2\cdot C_8+\overline{C}_3+\overline{C}_4)+X_1\cdot C_{10}$

C-list	F.V.list	Equation
-	C_{10}, C_3, C_4, C_9	$C_{10}= \overline{C}_3+\overline{C}_4+\overline{C}_5+\overline{C}_6$
C_5, C_6	C_{10}, C_3, C_4, C_9	$= \overline{C}_3+\overline{C}_4+C_{10}\cdot C_9+X_1\cdot C_{10}$
-	C_{10}, C_3, C_4, C_9	$C_3= \overline{C}_7+\overline{C}_9$
C_7	C_{10}, C_3, C_4, C_9	$= X_1+\overline{C}_9$

C_3 is also removed from the F.V.list. Substituting for C_3 in the equation for C_{10} we get

$$C_{10}= \overline{X}_1\cdot C_9+\overline{C}_4+C_{10}\cdot C_9+X_1\cdot C_{10}$$

Next we obtain an equation for C_4.

	C_{10}, C_4, C_9	$C_4= \overline{X}_2+\overline{C}_7$
C_7	C_{10}, C_4, C_9	$= \overline{X}_2+X_1$

eliminating C_4 also from the F.V.list. The equation for C_{10} now reads

$$C_{10}= \overline{X}_1\cdot C_9+X_2\cdot\overline{X}_1+C_{10}\cdot C_9+X_1\cdot C_{10} \quad \text{........(3·9)}$$

Similarly for C_9 we get

-	C_{10}, C_9	$C_9= \overline{C}_1+\overline{C}_2+\overline{C}_3+\overline{C}_4$
C_1, C_2, C_3, C_4	C_{10}, C_9	$= C_9\cdot C_8+X_2\cdot C_8+C_7\cdot C_9+X_2\cdot C_7$
$C_1, C_2, C_3, C_4, C_7, C_8$	C_{10}, C_9	$= C_9\cdot\overline{C_{10}}+X_2\cdot\overline{C_{10}}+\overline{X_1}\cdot C_9+X_2\cdot\overline{X_1}$ ....(3.10)

The circuit shown in figure 4 therefore is characterized by equations 3·9 and 3·10 which are the excitation equations for the two secondary variables and the output equation 3·8[5].

---

5. The equations obtained here are idential to those obtained by Unger [5] where y_1 and y_2 refer to C_9 and C_{10} respectively.

4. Conclusions

The algorithm presented here detects feedback loops analytically from a topological description. However, the following points should be noted.

a) The procedure concerns itself only with the terminal behaviour of the circuit. Hence, variables which have no effect on the external behaviour of the circuit, e.g. a redundant feedback loop, will be ignored.

b) The resulting excitation equations may be different to those used during the design of the circuit. In this case the behaviour obtained using this procedure will be equivalent to the original behaviour.

c) The algorithm does not accept explicit delays. It is assumed that the logic circuit being analysed is made up using real gates with inherent delays and that the circuit functions correctly. It is envisaged that the algorithm will be extended to include explicit delays and predefined gate delays.

d) The algorithm is equally applicable to the analysis of combinational circuits, in which there are no feedback loops. Thus the method is quite general and useful in general logic network analysis.

A preliminary version of this algorithm has already been programmed using a list processor imbedded in FORTRAN. We hope to include this algorithm as part of the facilities offered by the Computer-Aided-Logic-Design suite currently being developed at Brunel and Southampton Universities.[6]

References

[1]. S.H.Unger, Asynchronous Sequential Switching Circuits, J.Wiley & Sons, New York, 1969.

[2]. A.A.Kaposi and D.R.Holmes, Logic Network Analysis, Computer Aided Design, Autumn 1970 pp 9-18.

[3]. D.Cunningham, The Generation of Diagnostic Testing Procedures for Sequential Circuits, IEE Colloquium on Computer Applications to Design, Simulation and Testing of Logic Circuits and Systems, 16 November, 1971.

[4]. H.Y.Chang,C.G.Manning and G.Metze, Fault Diagnosis of Digital Systems, J. Wiley and Sons, 1970, p.95.

[5]. S.H.Unger, ibid, pp. 174-177.

[6] D.W.Lewin, E.Purslow and R.G.Bennetts.
Computer Assisted Logic Design - The CALD System
I.E.E. Conf. on C.A.D. Pub.No.86 p. 343-351 1972.

Acknowledgement

One of the authors (Pai) wishes to acknowledge the financial support given by the Science Research Council and International Computers Limited for part of the work.

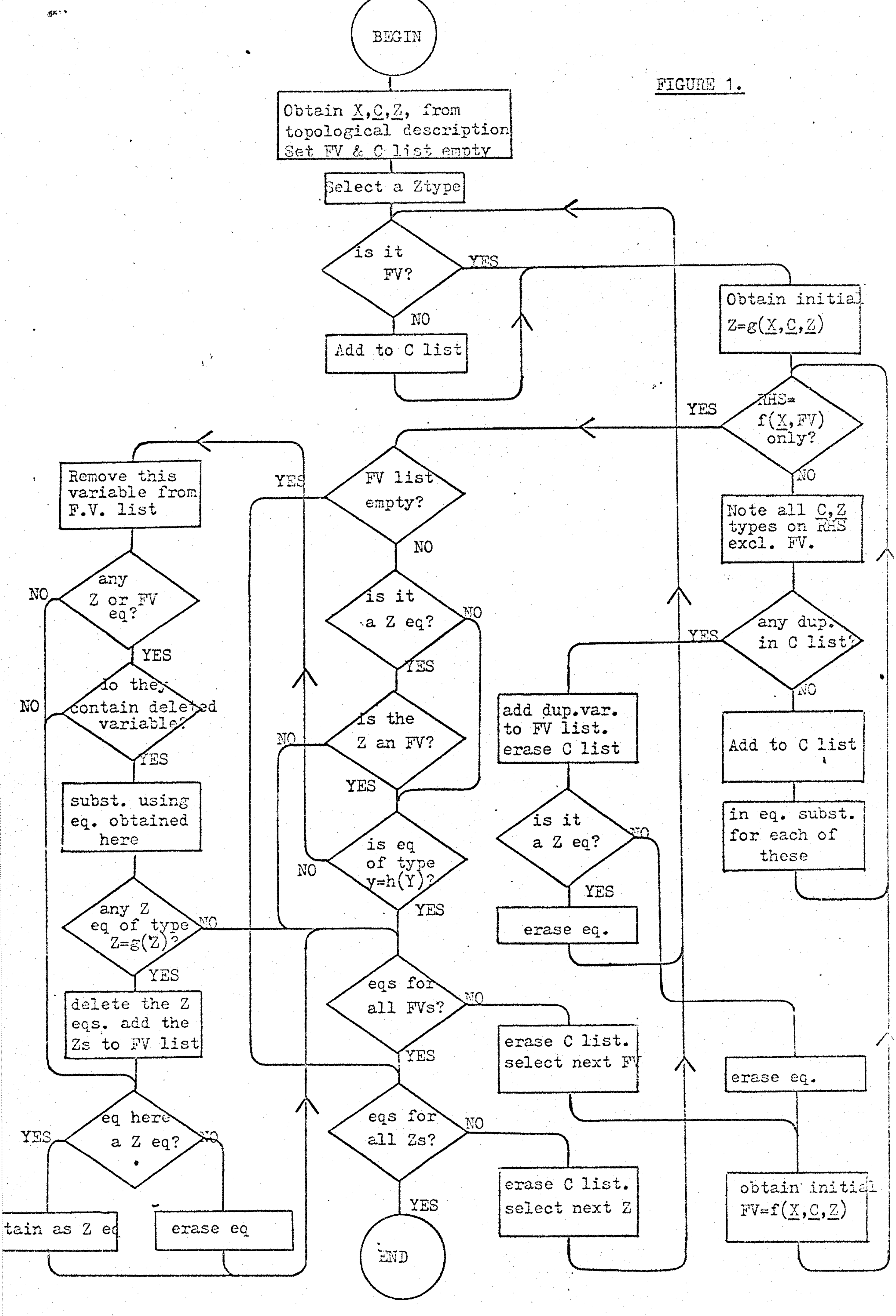

BEGIN
Obtain X,C,Z, from topological description Set FV & C list empty
Select a Ztype
is it FV?
YES
NO
Add to C list
Obtain initial Z=g(X,C,Z)
RHS= f(X,FV) only?
YES
NO
Note all C,Z types on RHS excl. FV.
any dup. in C list?
YES
NO
Add to C list
in eq. subst. for each of these
add dup.var. to FV list. erase C list
is it a Z eq?
NO
YES
erase eq.
erase eq.
obtain initial FV=f(X,C,Z)
FV list empty?
YES
NO
is it a Z eq?
NO
YES
is the Z an FV?
NO
YES
is eq of type y=h(Y)?
NO
YES
eqs for all FVs?
NO
YES
erase C list. select next FV
eqs for all Zs?
NO
YES
erase C list. select next Z
END
Remove this variable from F.V. list
any Z or FV eq?
NO
YES
do they contain deleted variable?
NO
YES
subst. using eq. obtained here
any Z eq of type Z=g(Z)?
NO
YES
delete the Z eqs. add the Zs to FV list
eq here a Z eq?
YES
NO
tain as Z eq
erase eq

FIGURE 1.

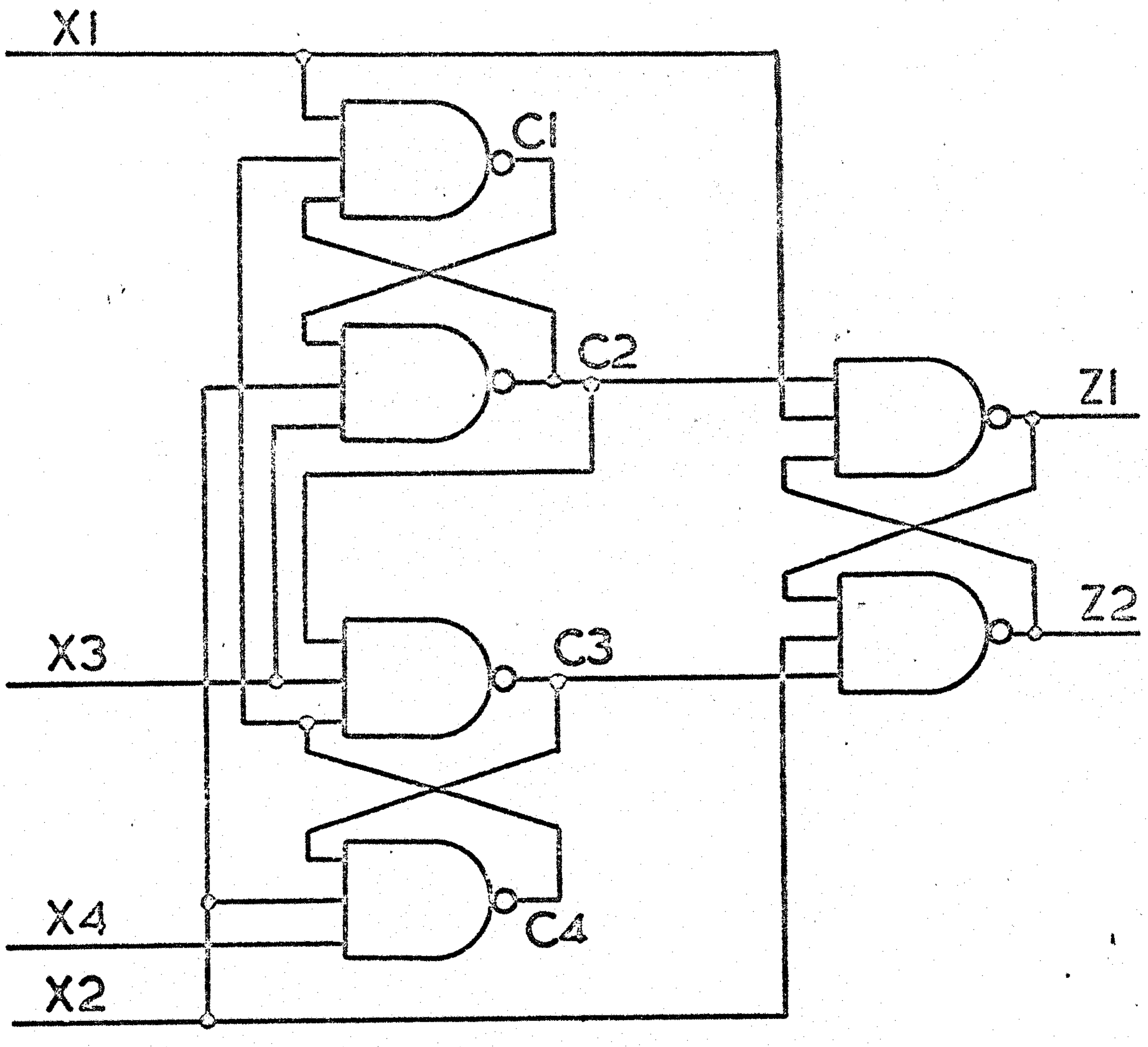


FIGURE 2

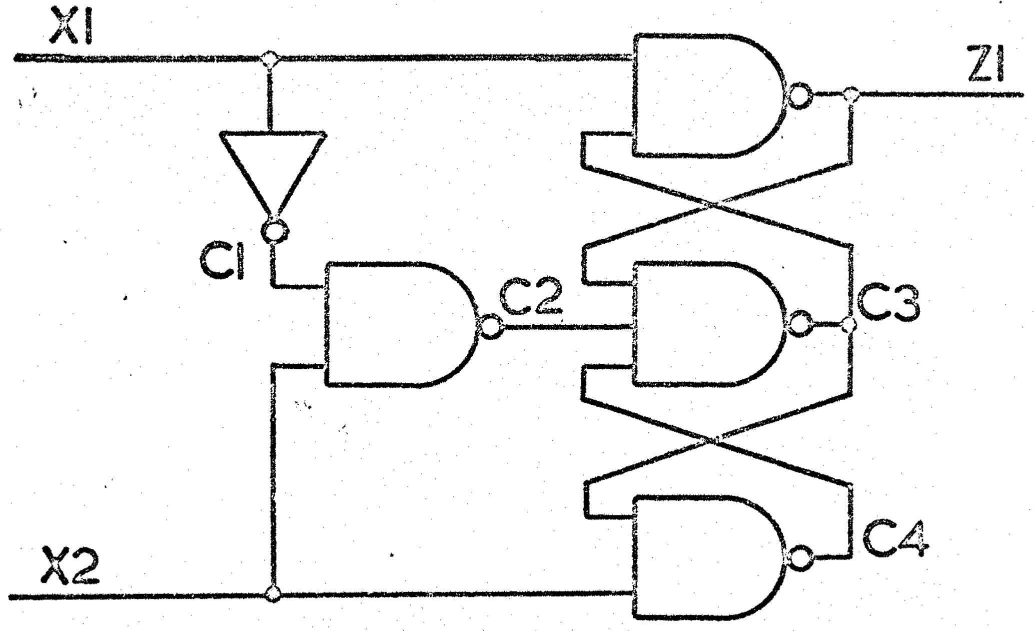


FIGURE 3

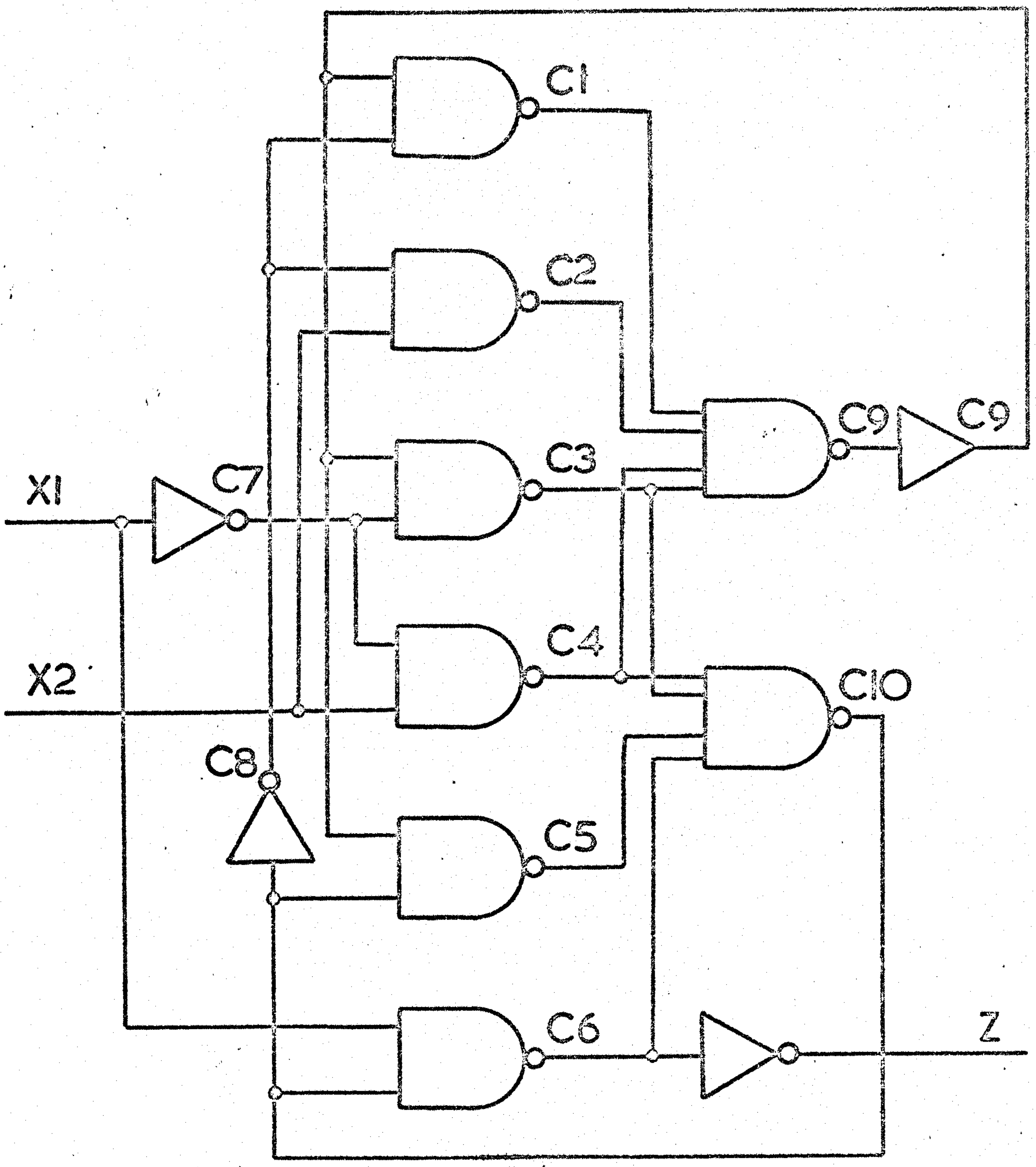


FIGURE 4

$$C_1 = \overline{X_1 \cdot C_4 \cdot C_2}$$

$$C_2 = \overline{C_1 \cdot X_2 \cdot X_3}$$

$$C_3 = \overline{C_2 \cdot X_3 \cdot C_4}$$

$$C_4 = \overline{X_4 \cdot X_2 \cdot C_3}$$

$$Z_1 = \overline{X_1 \cdot C_2 \cdot Z_2}$$

$$Z_2 = \overline{X_2 \cdot C_3 \cdot Z_1}$$

Table 1

$$C_1 = \overline{X_1}$$

$$C_2 = \overline{C_1 \cdot X_2}$$

$$C_3 = \overline{C_2 \cdot Z_1 \cdot C_4}$$

$$C_4 = \overline{C_3 \cdot X_2}$$

$$Z_1 = \overline{X_1 \cdot C_3}$$

<u>Table 2</u>

$$
\begin{aligned}
Z &= \overline{C_6} \\
C_1 &= \overline{C_9 \cdot C_8} \\
C_2 &= \overline{X_2 \cdot C_8} \\
C_3 &= \overline{C_7 \cdot C_9} \\
C_4 &= \overline{X_2 \cdot C_7} \\
C_5 &= \overline{C_{10} \cdot C_9} \\
C_6 &= \overline{X_1 \cdot C_{10}} \\
C_7 &= \overline{X}_1 \\
C_8 &= \overline{C}_{10} \\
C_9 &= \overline{C_1 \cdot C_2 \cdot C_3 \cdot C_4} \\
C_{10} &= \overline{C_3 \cdot C_4 \cdot C_5 \cdot C_6}
\end{aligned}
$$

Table 3

Titles for Figures