Computer Aided Design Techniques Applied to Logic Design

B.I. Dervişoğlu.

Thesis presented for the Degree of Doctor of Philosophy of the University of Edinburgh in the Faculty of Science, July 1973.

ABSTRACT OF THESIS

Name of Candidate Mr Bulent Dervisoglu

Address 127 Bruntsfield Place, EDINBURGH.

Degree Ph.D. *Date* 26th July, 1973.

Title of Thesis Computer Aided Design Techniques Applied to Logic Design.

Computer Aided Design techniques are essential for the development of complex digital systems. The existing methods of CAD approach the problem in two ways. The first one is to use a formal language to describe system behaviour and then use special algorithms to turn it into hardware. The second approach is to use flow-charts to specify the flow of information within a digital system and then to realise the flow-chart in hardware.

A different approach is presented in this thesis. Components are described (or provided by the design system) which realise the required operations. These are displayed on a graphics screen and connections are made by drawing lines between their terminals. A simulator is used to test the design. To illustrate the use of the CAD system, a simple computer was designed. Methods for realising the idealised components in practical versions are also discussed. Part of this involves the well-known state assignment problem in sequential circuit synthesis, and a new method is presented.

Use other side if necessary.

Summary.

Computer Aided Design techniques are essential for the development of complex digital systems. The existing methods of CAD approach the problem in two ways. The first one is to use a formal language to describe system behaviour and then use special algorithms to turn it into hardware. The second approach is to use flow-charts to specify the flow of information within a digital system and then to realise the flow-chart in hardware.

A different approach is presented in this thesis. Components are described (or provided by the design system) which realise the required operations. These are displayed on a graphics screen and connections are made by drawing lines between their terminals. A simulator is used to test the design. To illustrate the use of the CAD system, a simple computer was designed. Methods for realising the idealised components in practical versions are also discussed. Part of this involves the well-known state assignment problem in sequential circuit synthesis, and a new method is presented.

Table of Contents

Chapter 1: Introduction.

The growing complexity of digital systems make it imperative to use automatic processing in their design to the greatest possible extent. Application of existing digital computers to the production stages have so far been successful. For example component placement and conductor routing of printed circuit boards, back board wiring, generation of test sequences,...etc. have all been mechanised though some of the methods need improvements to cope with more complex situations. In general, problems at this end do not require creative thinking and the main objective is to optimise the solutions according to some criteria. However, the problems encountered in the initial stages of a digital system design are very different in nature. Here the objective is not to optimise but to eliminate unnecessary detail which tends to obstruct creative thinking. Before automatic processing can be introduced into these stages, one question has to be answered. 'What are the conditions under which a designer can apply creative thinking?' The answer to this question should form the underlining philosophy to any successful high level design method. It is no use finding ingenious techniques which can be easily programmed if they obstruct the designers' thought processes.

The design system should provide the user with the basic building blocks that he would like to work with and allow him to arrange them in any way to form more complex structures. Obviously, it is not possible to foresee all types of building blocks that the designer may wish to use. Therefore, he should be allowed to describe new ones. The new blocks can be of similar complexity to the existing ones or a number of existing blocks can be brought together to form a higher level block. Whatever the case is, he should be allowed

to describe them easily and in a natural way. "The development and employment of complex concepts doubtless is basic to most intellectual activity. This must be especially true for creative thought.... An essential ingredient of creative thought is the ability to move quickly and easily from one concept to a related one".* The designer can be helped by providing useful facilities which may act as 'suggestions' but his line of thought should not be restricted due to some artificial rules. He should be allowed to express himself in a natural way and the CAD system should be able to detect his mistakes.

Chapter 2 of this thesis describes some of the better known CAD methods for digital systems design. Since it is essential to know the 'state of the art' before attempting to improve it, an extensive study was made of this area. It is hoped that this chapter forms a good source of reference without being too long. Chapter 3 describes the author's philosophy in designing a digital system and Chapter 4 presents a CAD system based on this philosophy.

Application of this CAD system to the design of a relatively simple digital computer is illustrated in Chapter 5 followed by a description of the techniques used in the internal (computer) representation of logic designs in Chapter 6. Chapter 7 discusses some of the problems encountered in realising the components used in the design and presents a new state assignment technique for sequential circuits. Some proposals are made in Chapter 8 to improve the programs described, followed by the general conclusions drawn from this research.

(*) D. E Wooldridge. 'The machinery of the brain'. McGraw-Hill 1963, page 225.

Chapter 2 : Existing CAD Techniques.

This chapter discusses some of the better known Computer Aided Design methods used in digital systems design. Names of such methods can be found in the references. The methods discussed in this chapter have been selected not because they are better than the others but because they form a good cross-reference and show the general techniques used. In general, CAD techniques fall into two categories.

(i) - Methods in which special computer languages* are used to specify both the structure and the behaviour of digital systems. Programs written in the "Design Language" are then translated into Boolean equations which realise the digital system.

(ii) - Methods in which the operation of a digital system is described by a flow-chart and then realised by using special components which have a one-to-one correspondence with the flow-chart symbols.

Below, some examples from each group are given followed by a general criticism of the methods in each group. Finally an overall criticism of the general principles employed in each group is presented.

(*) Here, "language" is used to infer a linguistic representation, with a fixed alphabet and grammar.

2.1. Design Methods Using Formal Languages.

The following four sections describe some of the design methods that use a formal language in describing a digital system. Apart from differences in the languages themselves, the four methods discussed below differ from each other by the way in which the structure of a digital system is defined and also by the way in which descriptions are turned into hardware.

2.1.1. Logic Design Translator.

LDT (42), (74) is a design system composed of three major programs and can be described as "a three-pass, special purpose compiler, with logic equations rather than machine code as the final output" (42). The three passes of LDT correspond to,

(i) - translation of the input language into an intermediate language,

(ii) - timing analysis,

(iii) - term development and logic equation generation

A language description in LDT consists of two sections. The "Declarative" section provides a linguistic description of the block diagram structure of a digital system. A digital system is seen to consist of "Registers" and "Data Paths" which are defined by using "Declarative Statements". The language uses the Backus Naur Form (BNF) (70) for the representation of declarative statements and the syntax for register declarations is as follows.

⟨register⟩ :: = ⟨identifier⟩ | ⟨identifier⟩ (⟨size⟩)
| ⟨register⟩ (⟨structure list⟩)

⟨size⟩ :: = ⟨unsigned integer⟩ | ⟨unsigned integer⟩ ,
⟨unsigned integer⟩

⟨structure list⟩ :: = ⟨register⟩ | ⟨structure list⟩ ,
⟨register⟩

⟨special register⟩ :: = ⟨register⟩ (⟨delay⟩) | ⟨special
register⟩ (⟨structure list⟩)

⟨delay⟩ :: = ⟨unsigned integer⟩

Thus, a register is defined as a set of storage devices (together with the associated gating) which require only one clock time* to store data and a special register is a register which requires more than one clock time to store data. Special registers are used to represent more complex units such as an adder or a logical unit. Some examples of declarative statements are given below.

1. P (0,12)
2. Q (1,12) (Q1(1,6), Q2 (7,12))
3. SP (1,10) (5)

Statement 1. defines P as a 13-bit register with bits numbered from zero to twelve. Statement 2. defines Q as a 12-bit register and the bits are numbered from one to twelve. Furthermore, Q1 refers to the first half of Q and Q2 refers to the second half. Statement 3. defines SP as a 10-bit special register having a delay of five units. Its bits are numbered from one to ten.

(*) One clock time is the time required to set a flip-flop.

Since the full description of the block diagram structure of a digital system must include the interconnections between the registers, these are declared by using "Transfer Statements" which name a "Destination Register" and give a list of the "Source Registers" which can transfer data to it. The BNF representation of this type of statement is shown below.

⟨transfer⟩ :: = ⟨destination register⟩ = (⟨source register⟩)

⟨destination register⟩ :: = ⟨register⟩

⟨source register⟩ :: = ⟨register⟩ | ⟨source register set⟩ ,
⟨register⟩

for example, the statement

P = (Q,R)

indicates that register P can accept data from either of the registers Q and R.

In the "Operational" section of the language description, register transfer operations are described using ALGOL like statements. The general form of such statements is:

register "operation" register → register

which specifies that the result of the operation on the contents of the registers to the left of the arrow is to be placed into the register on the right hand side of the arrow. Statements without "operation" indicate simple transfers (P → Q). Other types of statements include subroutine calls, conditional statements, memory access statements, ... etc. A total of thirteen statement types can be specified. Some examples are given below.

Statement Type	Example
Transfer	1. P → Q
Arithmetic	2. P + Q → R
Subroutine	3. GETBUS
Memory Access	4. MA* → INSTREG
Control Transfer	5. GO TO 1
Logical	6. P∧Q → R
Shift	7. P MOVE RIGHT. OFF 3 → P

Statements are combined to form micro programs which represent the instructions to be executed by a digital system. The translator converts the input language into an intermediate language. This intermediate language is not a linguistic representation of the input description, but a table called the "Design Table" which lists the micro operations to be performed. The design table has twelve columns and as many rows as there are operations to be performed within a micro program. When translating the input description into the design table, the translator consults the declarations about the digital system structure and checks to see if the transfers specified are legal. In doing so, the translator may detect, for example, that the common BUS is required to perform a transfer, if the declarative section has specified such a structure. Furthermore, temporary storage may also be detected.

e.g. When translating the following statement

$(P \wedge Q) \vee R \rightarrow P$

the translator generates a new register to hold the result of the AND operation and the OR operation will be performed on the contents of the temporary storage and the R-register. In such cases, the translator assigns a name to the temporary register and notifies the

designer. The columns of the design table contain the following information

Column	1	Source register name.
"	2	MSB* of the field of the source register
"	3	LSB+ of the field of the source register.
"	4	Destination register name.
"	5	MSB of the field of the destn. register.
"	6	LSB of the field of the destn. register.
"	7	Equipment used for the transfer.
"	8	MSB of the field of the equipment used.
"	9	LSB of the field of the equipment.
"	10	Control conditions to be satisfied.
"	11	Relative time at which the micro step is to occur.
"	12	Delay value of the destination register.

The operators (e.g. $\wedge$, +) are entered in the control column (column 10) and the time column (column 11) contains a flag value which will be used by the Timing Analysis Routine. A "0" flag indicates that the following row is a part of the operation (the present row and the next can be considered as one) and a "1" flag indicates that the design table is partitioned below that row.

(*) Most significant bit.
(+) Least significant bit.

The timing analysis routine determines the time at which an operation may begin. Unless a delay value is specified, the transfers take place in one clock time. Since the control transfer statements (GO TO, SUBROUTINE CALL, etc.) alter the sequence of operations, the design table has to be partitioned at that point to enable the timing analysis of each branch individually. The relative starting time of each operation is calculated by observing the following rules.

1- The first operation in the design table may begin immediately.

2- The first operation in any partition of the design table may begin at time $t_p + 1$ where t_p is the highest clock time assigned to any previous operation in the table.

3- The start time of all other operations in any partition is found by calculating the earliest time when the registers and additional equipment required by the operation, will be free. The operation may begin immediately if all devices are free, but otherwise has to wait until the previous operations are finished and the devices become available.

Because of the third rule, LDT causes a digital system to be realised as a "Pipe-Line" (38) system and to ensure its correct operation the design table is partitioned whenever a control transfer statement is encountered. Finally, to each micro program, an implicit subroutine called "Micro Sequence Completed" is added so that control may be given to the next micro program in sequence. This is achieved by inserting the following information after the last row of the design table.

Column	1	0
"	4	CLOCK
"	10	RESET
"	11	Tmax
"	12	Delay value of CLOCK.

(Tmax is the highest clock time at which any previous operation will be completed).

The following example, taken from (42) shows a micro program and the design table after timing analysis. It is assumed that the declarations about the structure of the system have been previously given and that they include the two bus lines BUS1 and BUS2.

1. P $\longrightarrow$ Q
2. K + 1 $\longrightarrow$ P
3. S + T $\longrightarrow$ W
4. X MOVE RIGHT. OFF 3 $\longrightarrow$ X
5. IF K = P

   THEN (6. W $\longrightarrow$ X)

   ELSE (7. W $\longrightarrow$ Y)

   (8. $R_{6-10} \longrightarrow W_{2-6}$)
9. K-1 $\longrightarrow$ P

Table 2.1 A micro program input to LDT.

Table 2.2 gives the design table generated for this micro program.*

(*) The table has been copied from page 257 of ref. (42).

Microstep Number k	Row i	Source Register 1	Begin 2	End 3	Destination Register 4	Begin 5	End 6	Eq. Used 7	Begin 8	End 9	Control 10	Time 11	Delay 12
1	1	P	1	n	BUS1	1	n					1	0
	2	BUS1	1	n	Q	1	n					1	1
2	3	K	1	n	BUS1	1	n					2	0
	4	BUS1	1	n	COUNT	1	n				UP	2	1
	5	COUNT	1	n	BUS1	1	n					3	0
	6	BUS1	1	n	P	1	n					3	1
3	7	S	1	n	ARITH (A)	1	n				ADD	1	5
	8	T	1	n	ARITH (B)	1	n				ADD	1	5
	9	ARITH	1	n	W	1	n					6	1
4	10	X	1	n	SHIFT	1	n				RIGHT.OFF	1	1
	11	SHIFT	1	n	SHIFT	1	n				RIGHT.OFF	2	1
	12	SHIFT	1	n	SHIFT	1	n				RIGHT.OFF	3	1
	13	SHIFT	1	n	X	1	n					4	1
5	14	K	1	n	BUS1	1	n					4	0
	15	BUS1	1	n	LOGICAL (A)	1	n				EQL	4	3
	16	P	1	n	BUS2	1	n					4	0
	17	BUS2	1	n	LOGICAL (B)	1	n				EQL	4	3
	18	LOGICAL	1	n	FF1	1	1					7	1
6	19	W	1	n	X	1	n	FF1	1	1	FF1	8	1
7	20	W	1	n	Y	1	n	FF1	1	1	~FF1	8	1
8	21	R	6	10	W	2	6	FF1	1	1	~FF1	9	1
9	22	K	1	n	BUS1	1	n					4	0
	23	BUS1	1	n	COUNT	1	n				DOWN	4	1
	24	COUNT	1	n	BUS1	1	n					7	0
	25	BUS1	1	n	P	1	n					7	1
999	26	"0"	-	-	CLOCK	-	-				RESET	10	1

Table 2.2 Design Table.

The final step in the design process is to convert the design table into application equations for each bit of the registers. The term development routine scans each micro program and extracts the terms for the bits of the registers used within the micro program. A final sort run combines these terms and forms the application equations which may be further processed to fit practical requirements. Proctor (74) reports that the equations generated for a part of the Burroughs D825 computer, correspond closely to the actual equations used in the earlier design of D825. Some of the equations generated by LDT for the example given above, can be found in the reference.

2.1.2. Schorr's Register Transfer Language.

Schorr (80) describes a CAD system for the analysis and synthesis of digital systems. The language used is an adaptation of Reed's language (12). Using Schorr's approach, a digital system is described by a set of register transfers and the two reverse processes of design synthesis and analysis are tackled by starting with the register transfers and deriving the Boolean equations to realise them, when doing synthesis, and starting with the Boolean equations and obtaining a set of register transfers for the analysis of a digital system. The register transfer statements are written in a language which can be translated into Boolean equations by a syntax oriented compiler. By changing the syntax table of the compiler, it can be made to translate the Boolean equations into register transfers. The simplified syntax tables for both processes can be found in the reference (80).

(i) - Design Synthesis

Instructions to be executed by a digital system are expressed as micro programs using a simple register transfer notation. Each

statement is preceded by a "Conditional Expression" (CE) enclosed within vertical bars. The CE's can be timing signals or logical expressions formed from timing signals or control signals. Concurrent operations can be written on the same line, separated by semi-colons. The last register transfer expression on a line indicates the next timing signal to be activated and hence defines the next set of statements to be executed.

e.g.

$| t_1 | : P \longrightarrow Q;\ X \longrightarrow Y;\ 1 \longrightarrow t_2$

Individual bits of registers are identified by subscript expressions and logical operations, such as AND, OR, EXCLUSIVE OR, NEGATION are indicated by the connectives " $\cdot$", "+", "$\oplus$", "'", respectively. Other symbols used are "," for the concatenation of two or more registers, "L_j" to indicate a left-shift by j places and "v" to indicate the OR of the individual bits of a register.

Table 2.3 shows the micro program of an adder. This table is taken from ref. (80) and the operations of the adder are as follows.

Register M contains the augend which will be replaced by the sum of the contents of the M and A registers. During the operation of the adder, the contents of M are first transferred to the N register and a bit by bit addition is performed between the N and A registers. Any carries generated are shifted left and stored in N. When there are no more carries (indicated by D = 0) the addition is complete and the adder waits for an ADD signal and then repeats the operation. Any overflow condition is indicated by lighting a lamp attached to register L.

1. $\lvert t_1 \rvert$:	$M \rightarrow N;$		
	$0 \rightarrow D;$		
	$0 \rightarrow OF;$	$1 \rightarrow t_2$	
2. $\lvert t_2 \rvert$:	$N \oplus A \rightarrow A;$		
	$L_1(N.A) \rightarrow OF,N;$		
	$v(N.A) \rightarrow D;$	$1 \rightarrow t_3$	
3a. $\lvert t_3.D' \rvert$:	$A \rightarrow M;$		
	$OF \rightarrow L;$		
	$0 \rightarrow ADD;$	$1 \rightarrow t_4$	
3b. $\lvert t_3.D \rvert$:	$0 \rightarrow D;$	$1 \rightarrow t_2$	
4a. $\lvert t_4.ADD' \rvert$:		$1 \rightarrow t_4$	
4b. $\lvert t_4.ADD \rvert$:	$0 \rightarrow L;$	$1 \rightarrow t_1$	

Table 2.3 Micro operations of adder.

The control unit for the adder is synthesized separately, by first extracting the micro operations for the control signals.

$\lvert t_1 \rvert$:	$1 \rightarrow t_2$
$\lvert t_2 \rvert$:	$1 \rightarrow t_3$
$\lvert t_3.D \rvert$:	$1 \rightarrow t_2$
$\lvert t_3.D' \rvert$:	$1 \rightarrow t_4$
$\lvert t_4.ADD \rvert$:	$1 \rightarrow t_1$
$\lvert t_4.ADD' \rvert$:	$1 \rightarrow t_4$

Table 2.4 Control unit signals of adder.

The timing signals (t_i) can then be generated by using any of the following synthesis techniques.

1- Delay line synthesis.

2- Using a timing counter controlled by the external signals ADD and D.

3- Using the control unit design introduced by Wilkes (88).

Schorr gives the realisation of Table 2.4 by using the first and second methods mentioned above. He also notes that since a CU may have to generate more than one set of control signals, the delay line synthesis cannot be used in general.

Translation of the register transfer statements into Boolean equations is done by a compiler and the techniques used can be found in the reference.

(ii) - Design Analysis

The analysis programs can translate Boolean equations into a set of register transfers characterised by those equations. The translation is possible since the Boolean equations are in BNF form. Further requirements are that, the Boolean equations

1- must be in the sum-of-products form,

2- must be set-reset equations for RS flip-flops,

3- must have all literals appearing in them identified.

The mechanics of translating the Boolean equations into register transfers can be found in (80). However, the problem of sorting register transfers into appropriate micro programs representing the instructions to be executed by a digital computer is unsolved, as reported by Schorr.

2.1.3. The ALERT System.

ALERT (39) is a CAD system designed to convert logic descriptions into hardware. As input, ALERT takes the architecture of a digital system described in Iverson notation and produces the necessary Boolean equations to realise the described logic. Iverson notation

is used in the APL language (4) and has also been used to describe the architecture of some computers (37). This notation is an algorithmic language that makes it possible to represent sequential logic as programs of micro events. Further processing (25), (78) of the Boolean equations generated by ALERT, turns them into standard computer circuits. Because of the algorithmic structure of the notation it is also possible to simulate the description before realising it. Several modifications have been made to the Iverson notation to use it in ALERT. The most significant change is the inclusion of declaration statements to define the class and dimension of variables. Whereas, in the original notation arrays are not declared, in ALERT a constraint was put which causes all variables to be taken as a single "bit" unless otherwise is stated in a declaration statement. Also, the special Iverson operators such as $\perp$, $\leftarrow$, w, have been replaced by 'VALUE', = , 'SUFFIX', respectively, so that standard input devices can be used. ALERT goes through eight major steps to output the Boolean equations. These steps are briefly described below.

1- Translation

During this step the input format, syntax and consistency is checked. Implied variables are generated and the specification is reformulated.

2- Selection Decoding

During this step the necessary decoding to address subscripted variables (registers) is generated.

3- Macro Generation

This step replaces some MACRO block definitions with actual logic circuits obtained from the system library.

4- Sequence Analysis

During this step the micro events specified by the description algorithm are assigned to time periods. This is done in such a way as to minimize the number of periods to be used. A micro program is partitioned into event groups by splitting it at the following points.

(i) The beginning of each micro program.

(ii) Statements which are the destinations of "GO TO" statements.

(iii) Statements after each conditional ("IF") statement.

(iv) The second and subsequent assignment statements which assign a value to a variable receiving input more than once within an event group.

5- Identification of Flip-Flops

Variables which must retain their values after their inputs have changed are identified in this step and are assigned to flip-flops. The user may also explicitly declare flip-flops by typing (=FF) after the name of the variable. For use with such variables, new statement types to assign values to the SET or RESET inputs of flip-flops are included in the language.

6- Control Provision

This routine generates a control counter and decodes its outputs for use as the timing signals required by step 4.

7- Consolidation

During this step duplicated logic blocks are eliminated, inefficiently connected arrays are re-arranged and associated elements are tied together.

8- Expansion

This routine expands each connection where an array is implied,

generating an individual copy for each bit in the array and produces a point-to-point connection list.

The following example is given in (39).

Example:

Assume that there are eight 3-bit registers forming an array called X. There is another 3-bit register called the K-Register and an 8-bit register called the M-Register. The task is to load the last three bits of the M-Register into the k-th register in X (k is the value stored in the K-Register) if k is not zero. If k is zero, the 0-th register in X will be cleared.

The corresponding statement coded in the original Iverson notation is

$$X^{k} \leftarrow (k \neq 0) \wedge (w^{3}/m).$$

This statement is input to ALERT by the following micro program.

```
D   X(DIM = 8,3)    (=FF).   X is declared as eight 3-bit registers.
D   K(DIM = 3).              K is declared as a 3-bit variable.
D   M(DIM = 8).              M is declared as an 8-bit variable.
M   LIX                      This is the Load Index micro program.

          X('VALUE'K) = ('OR'/K)'AND'('SUFFIX'(3)/M).
          END OF SYSTEM.
```

Table 2.5 A micro program in ALERT.

ALERT processes the above given micro program and generates the necessary logic to realise it, in the form of Boolean equations to be processed by other programs. The layout of the logic generated by ALERT is given in Figure 2.1. It shows how the i-th row of X is

selected and how the SET and RESET inputs are driven. The "OR" gate tests the contents of the K-Register and its output is used to gate the last 3-bits of the M-Register. The decoder selects the appropriate row of X and its outputs are gated with the S1(1) signal, which is the timing signal.

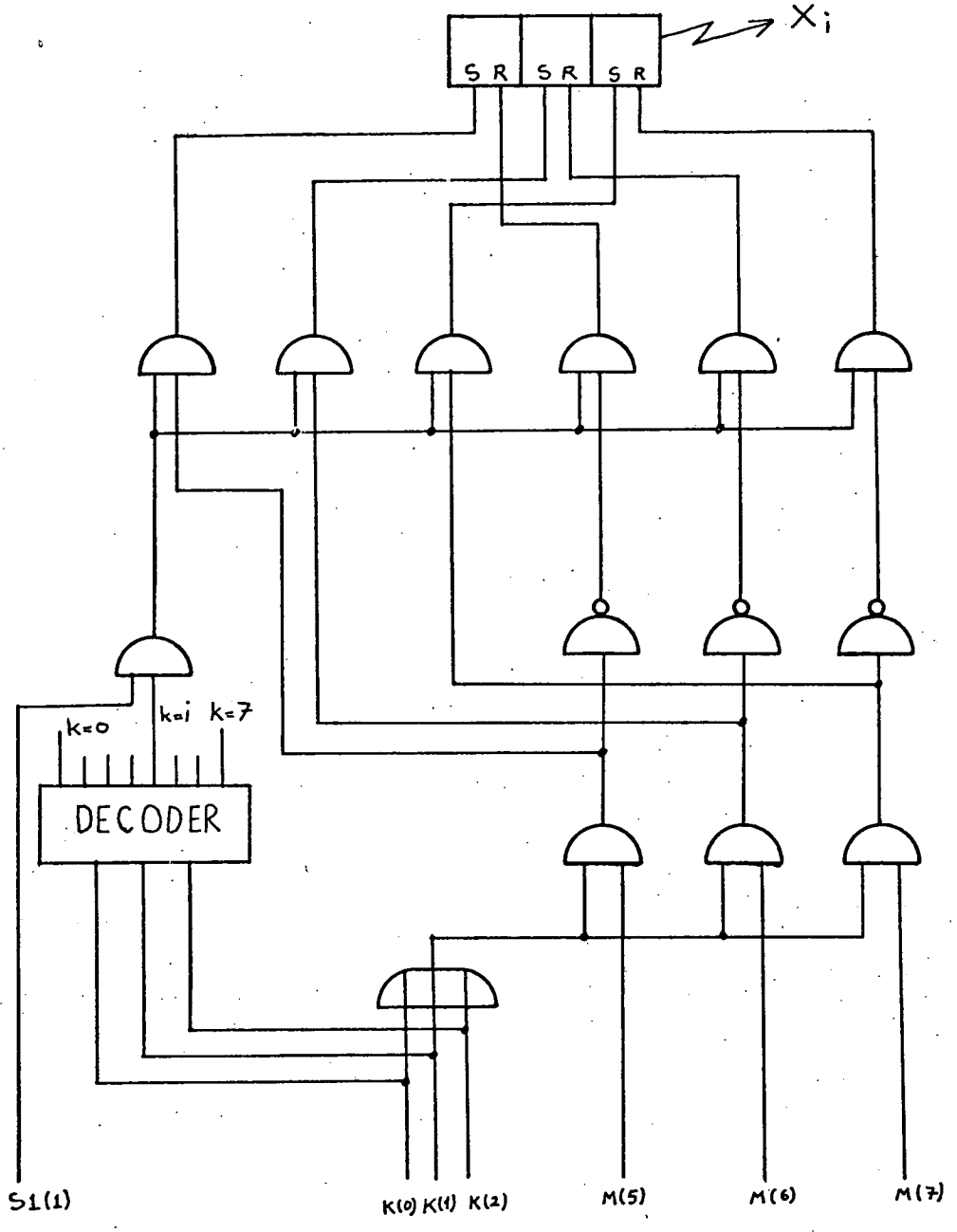


Figure 2.1 Design generated by ALERT.

2.1.4. Digital Systems Design Using Sequential Circuits.

Gerace (40) describes a method for designing a digital system as a network of interconnected sequential circuits. Basic to the method is a register transfer language to specify the behaviour of a digital system and a special type of sequential circuit, called "Level-Input, Level-Output, Clocked" (LLC) sequential circuit. The final structure of the digital system is largely undetermined until the flow-tables to realise the register transfers are formed. This is because the registers do not appear on their own but are 'buried' within the LLC circuits that form the digital system. The steps involved in designing a digital system using this approach can be summed up as follows.

(i)- The operation of a digital system is defined by means of a formal language using register transfer statements.

(ii)- The linguistic description is translated into flow-tables representing LLC circuits.

(iii)- The flow-tables are turned into sequential circuits using realistic components.

It is observed that the final structure of digital systems designed using this method consists of two parts which are called the "Operation Part" (Sub-system O) and the "Control Part" (Sub-system C).

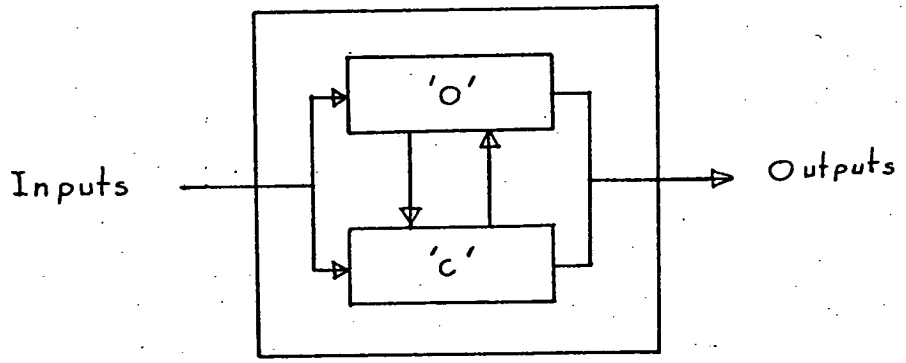


Fig. 2.2 System Organisation.

The language used to define the register transfers is a simple register transfer language. A "Sentence" can be formed from "Transfer Relations" and "Control Expressions" (CE). Transfer relations can be "Register Transfer Relations" which are called "Operation Words" (OW) since they are executed in sub-system O, or they can be "Control Transfer Relations" which are called "Control Words" (CW) since they are executed in sub-system C. For example, the OW

$$A_i \longrightarrow B_i \qquad i = 0, 1, \ldots, n$$

means that the contents of B are to be replaced by the contents of A for bit values i = 0 to n. B_i is called the "Transfer Register". The CW

$$S_1 \longrightarrow S_2$$

indicates a jump from the present micro instruction (S_1) to another micro instruction (S_2). The label of any micro instruction (S_i) is enclosed in vertical bars and precedes the instruction. Logical operations such as AND, OR, NOT, EXCLUSIVE OR are allowed and are represented by the symbols ".", "+", "'", "$\oplus$", respectively. Also, carry digits may be indicated by using square brackets. The register transfers may be conditioned by using CE's which are enclosed in parantheses. Register transfers which are executed simultaneously can be written on the same line, separated by commas. A typical micro instruction may look like

$$|S_1| \ (\text{ADD,GO:11}) \quad A_i \oplus M_i \oplus [r_{i-1}] \longrightarrow M_i,\ S_1 \longrightarrow S_2 \qquad i = 0,1,\ldots.n$$

which represents the micro instruction to add the two digits of the A

and M registers and the carry digit from the previous bit and then put the result into the corresponding bit of the M-Register. When this is done control will be given to another micro instruction whose label is S_2. These operations will be executed only if both ADD and GO signals are equal to logical one. When a sentence is conditioned as above, it is called a "Conditional Sentence" (CS). Micro instructions can be grouped to form micro programs which represent instructions to be executed by a digital system. It is assumed that one of the micro programs is a special one which fetches the operation code of the instruction to be executed and causes a jump to the proper micro program to execute it.

It was mentioned earlier that the linguistic descriptions are later turned into flow-tables representing LLC circuits. An LLC circuit is a sequential circuit with the following characteristics.

(i)- When the clock pulse is not present the internal state of the circuit is stable. However, changes in the level inputs may produce changes in the outputs.

(ii)- When the level inputs are stabilised the clock pulse is applied. The present state and the outputs do not change during the presence of the pulse but all changes may occur only after the pulse is removed.

To see how LLC circuit flow-tables are constructed, consider the flow-table of a sequential machine SM_1, having two inputs (X_1,X_2), two outputs (Z_1,Z_2) and three internal states (all inputs and outputs are levels).

Present State	Next-State and Outputs			
	00	01	11	10
1	1,00	2,01	3,01	1,10
2	3,00	1,11	1,11	3,10
3	2,00	1,11	1,10	3,10

Table 2.6 Flow-table of SM_1.

To realise SM_1 as an LLC circuit, first a pulse input is added to it. This input will be the clock pulse controlling the operation of the LLC circuit. The flow-table of SM_1 is then expanded such that all next-state entries are stable states when the clock input is zero and are the same as those for SM_1 when the clock input is one. The outputs of SM_1 are duplicated on the two halves of the flow-table such that outputs are the same for input conditions which differ only in the value of the clock pulse. Hence the LLC flow-table of SM_1 is

Present State	Next-State and Outputs							
	00	01	11	10	00	01	11	10
1	1,00	1,01	1,01	1,10	1,00	2,01	3,01	1,10
2	2,00	2,11	2,11	2,10	3,00	1,11	1,11	3,10
3	3,00	3,11	3,10	3,10	2,00	1,11	1,10	3,10
	Clock = 0				Clock = 1			

Table 2.7 LLC Circuit flow-table of SM_1.

Gerace describes three different types of flip-flops and shows how an LLC circuit can be realised by using these special flip-flops. The complexity of realisation is seen to be dependent upon the particular type of flip-flop used.

The method of converting the language description of a digital system into a set of flow-tables for LLC circuits, is fully explained in the reference (40). Below, a brief description of the technique used is given, followed by a simple example.

A list "L" of OW's is called "Convertible" if in all the OW's in L, the transfer register elements have the same index. The first step in converting a convertible OW is to define the inputs and outputs of the LLC circuit which will realise it. A simplified version of the rule by which the inputs and outputs of the LLC circuit are defined, is as follows. The register names appearing ONLY on the left hand side of the arrow form the inputs and the transfer register name forms the output of the LLC circuit realising an OW. If the convertible list L contains more than one OW, then the inputs and outputs of the LLC circuit are a conjunction of the inputs and outputs for the individual OW's, respectively. Once the inputs and outputs are determined, the equations defining the behaviour of the LLC circuit can be obtained by the following procedure.

Let 'k' be the number of OW's in L. Then, the transfer register name in any OW, say OW_j, is replaced by the next-state variable "Y_j". If the same register name also appears on the left hand side of the arrow in any of the OW's in L, then they are replaced by the present-state variable "y_j". Finally, the arrows are replaced by "equal" signs such that k expressions defining the behaviour of the LLC circuit which realises L, will be obtained. As an example, assume that a digital system is characterised by a single micro program.

$$|S_1|\ A_i + B_i \rightarrow A_i,\ B_i \cdot C_i \rightarrow C_i\ ,\ S_1 \rightarrow S_1 \qquad i = 0, 1, \ldots, n$$

Since all transfer registers (A and C) have the same index (i) this list of OW's in S_1 is convertible. The LLC circuit realising this micro program has only one input (B_i) and two outputs (A_i and C_i). Replacing the register names by the corresponding state variables and the arrows by the "equal" signs, the following equations are obtained.

$$y_1 + B_i = Y_1$$

$$B_i \cdot y_2 = Y_2$$

The internal variable map of the flow-table obtained from these equations, is given below.

		$B_i = 0$		$B_i = 1$	
y_1	y_2	Y_1	Y_2	Y_1	Y_2
0	0	0	0	1	0
0	1	0	0	1	1
1	1	1	0	1	1
1	0	1	0	1	0

Table 2.8 Internal variable map.

The outputs are defined by using the relations $A_i = Y_1$, $C_i = Y_2$. Then, by giving a name to each row of Table 2.8 and expanding it to form the flow-table of an LLC circuit Table 2.9 is obtained.

Present	Next-State and Outputs (A_i,C_i)			
State	$B_i = 0$	$B_i = 1$	$B_i = 0$	$B_i = 1$
1	1,00	1,00	1,00	4,00
2	2,01	2,01	1,01	3,01
3	3,11	3,11	4,11	3,11
4	4,10	4,10	4,10	4,10
	Clock = 0		Clock = 1	

Table 2.9 LLC circuit to realise the micro program.

Note that state 2 is disconnected from the others and can be deleted.

The micro program given above, can then be realised by using n + 1 identical LLC circuits and the structure of the system will be as shown in Figure 2.3.

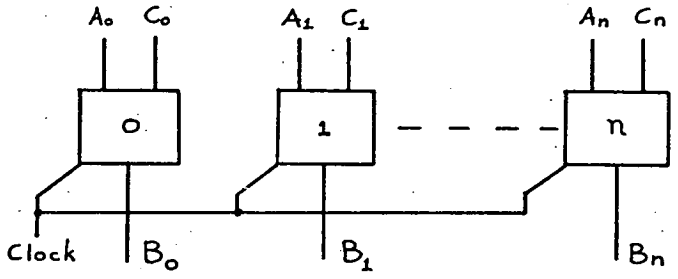


Fig. 2.3 Organisation of the LLC circuits.

Since the digital system given above consists of one micro step, the control part of the system does not exist. The conversion of a digital system description, consisting of more than one micro programs some of which may consist of more than one micro instructions, follows the same lines as above but is more complex and is difficult to do by hand. The full method is described in the reference cited which includes some examples.

The solution to the final step in the design process, where the LLC circuits are realised using actual components, is a problem in Sequential Machine Theory. The reader is referred to the appendix in reference (40) where methods to realise an LLC circuit using any of the flip-flop models introduced by Gerace, are given.

2.1.5. Discussion.

The four CAD methods described in section 2.1. typify the major techniques used in designing digital systems from a language description. Below, various aspects of these methods are compared

with each other.

(i)- Language Aspects.

All four methods use languages which describe the operation of a digital system by register transfer operations. From a language point of view, it is very convenient to treat any component as a register (or a special register). However, with the recent advances in MSI technology, many types of registers and standard components have become easily and cheaply available and this has made it possible to reduce the register transfer operations to, simply, setting the mode bits (control inputs) of a component to correct values. Therefore, writing register transfer equations and obtaining Boolean equations for the individual bits of such standard components is no longer required. With the exception of the Iverson notation, the languages used are simple and easy to learn. Iverson notation however, is a higher level language allowing the use of some very powerful operators and this may cause some problems in its usage. Consider the example given in section 2.1.3. It is required to test the value of a 3-bit register, K. This is achieved by typing ('OR/K) which represents the OR operation of the bits of K. The result will be zero if K is zero and one if K is non-zero. The same test could be achieved by typing ('VALUE'K $\neq$ 0) but this would imply a more complicated testing mechanism. Since K is a 3-bit register and 0 is a scalar variable, a new 3-bit register would be generated, reset to 0 and then a bit-by-bit comparison of the two registers would be done. Clearly, the language is too powerful and one has to have a very sound knowledge of the notation and the way it will be interpreted by the translator so that 'correct' expressions can be written.

(ii)- System structure.

The four methods discussed in section 2.1 show varying amount of importance attached to defining the structure of a digital system. In LDT, all registers and data paths should be declared in the declarative section of the description. In Schorr's language the registers are declared but the data paths are determined by the register transfer statements. In ALERT, declaration statements are used only to define the size and type of storage elements, otherwise default assumptions are made and lengths of variables are matched to each other during operations. Gerace's system leaves the structure undetermined until the flow-tables are obtained. Each approach has its advantages and disadvantages but if the structure is to be fully described (as in LDT), then facilities should exist to input a block diagram structure via graphic input devices. The reason for this is that, even though a block diagram can be fully defined in linguistic terms, such a form of description fails to give an overall view of the structure to the human designer. The approach chosen by Gerace has the disadvantage that since the structure of a digital system is determined as a result of processing the description of its behaviour, changes in the specification of the performance may cause changes in the structure. This may prove unacceptable since maintenance and testing requirements cannot be planned in advance and designing a 'family' of digital systems may not be easily achieved. A similar problem exists in LDT. When a change is made to the language description, since the relative starting times of the micro operations in the design table may be effected, the design table has to be analysed and new time values have to be evaluated again. This may mean a re-design of the timing circuit.

(iii)- Timing and Control.

The input to LDT does not contain any timing information. The timing analysis routine of LDT, determines the execution time of each micro statement on a pipe-line basis. This makes the design of complex concurrent operations difficult since the designer cannot synchronise operations. Schorr's register transfer language requires that each micro statement should specify the next time period to be activated. The control unit is then designed by separating the transfer statements for timing, from the others. However, this results in a control unit which can generate only the timing signals; other control signals are realised together with the register transfer operations. ALERT does implicit timing but the user can gain access to the control counter of a micro program by explicitly declaring it as a register which he can then set to any value both within and from outside that micro program. This ability to control the timing makes it possible to synchronise operations in a parallel processor. The final structure of a digital system designed by using Gerace's techniques naturally divides into a control and an operation part. The flow-table for the control part will have as many internal states as there are different states in the micro program and its next-state entries will be obtained from the state transfer relations written at the end of each micro statement.

(iv)- Modularity of Design.

Schorr's system is applicable to small problems but since there is no provision for dividing a digital system into component parts, a system has to be designed as a whole, thus making it difficult to apply the method to large size problems. Both LDT and ALERT allow

for the description of a digital system as composed of component parts, each with its own specification. However, LDT dissolves the modularity when generating the design table, by replacing the 'CALL's to other micro programs with the entire micro sequence for the called micro program. In Gerace's system, decomposition is done when converting the input description into flow-tables. Since a set of OW's can form a convertible list only if the index of the transfer registers are the same for all OW's in that list, the way in which a digital system is partitioned depends upon this index. Thus, if a change to the input description involves a change of the index of a transfer register, a different organization of the sequential networks forming the system will be necessary.

(v)- Conversion into Hardware.

In LDT, Boolean terms are obtained from the design table and then combined (sorted) to obtain the equations for each bit of the registers. In Schorr's system, equations for register transfers are generated by a syntax oriented compiler which translates register transfers into Boolean equations (or vice versa for analysis). The quality of these equations depend upon the translating and sorting techniques used. ALERT generates Boolean equations by methodically expanding the input specification and performing implied operations (e.g. selection decoding). A consolidation routine attempts to simplify the structure before it is fully expanded and its algorithms strongly influence the optimality of designs generated by the ALERT system. However, even more important than the consolidation routine are those routines which perform the implied operations since they effect the structure. Consider the example given in section 2.1.3. The subscript expression X('VALUE'K) implies the use of a decoder to address array X. On the

right hand side of the statement in the micro program shown in Table 2.5, is the expression 'OR'/K which is used to test the contents of K. Since, which output of the decoder becomes active depends upon the contents of K, no separate testing is necessary. If any register other than the 0-th one is addressed, it can be loaded with the last three bits of the M-Register and should the 0-th register be addressed, it can be reset without using the M-Register. Figure 2.4 gives the logic diagram of this design.

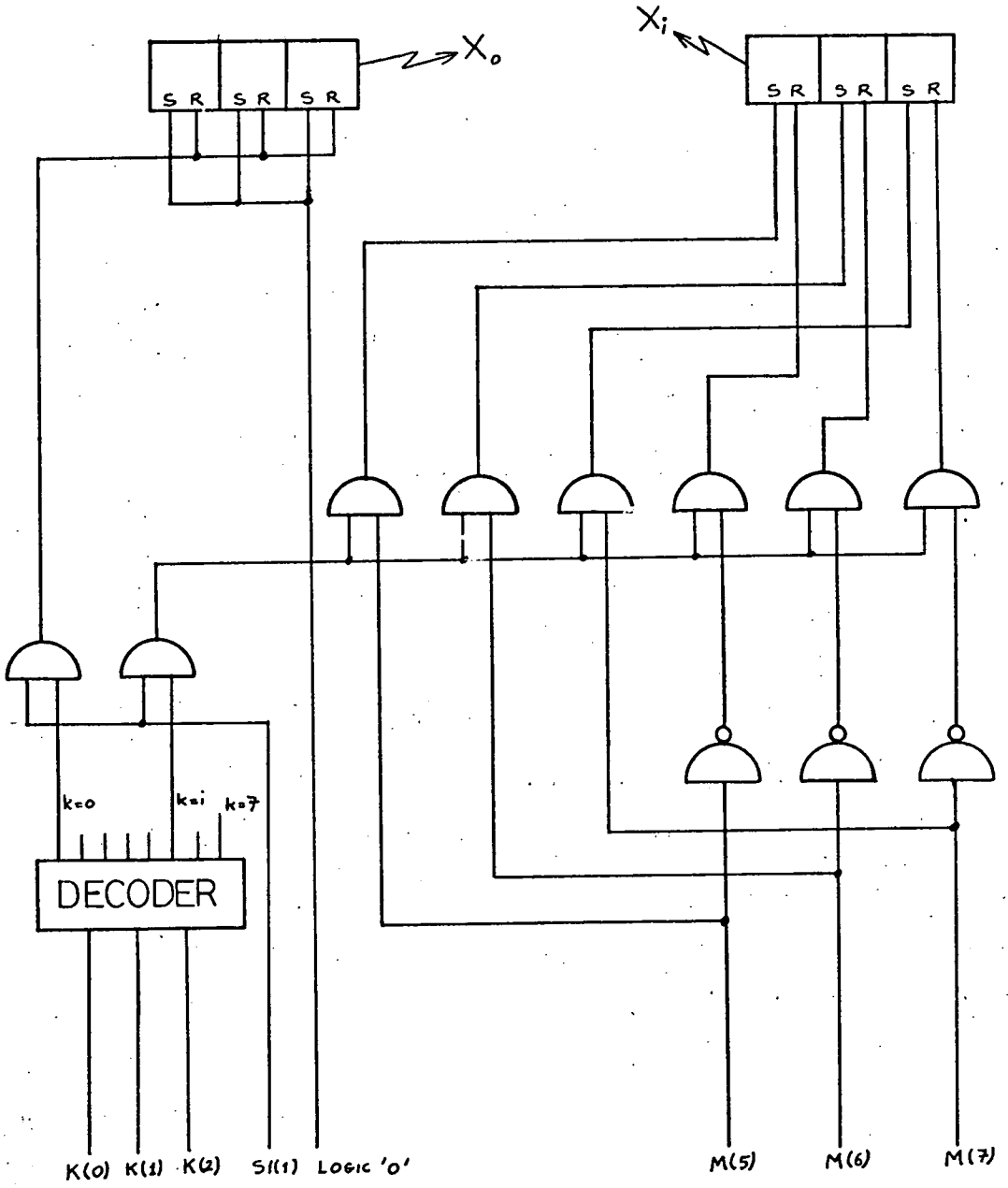


Figure 2.4 A modification of design generated by ALERT.

It can be seen that four gates to do the test and six gates to load data (reset) into the 0-th register are eliminated. In general 3N + 1 gates will be eliminated, where N is the length of the registers in X. Apart from generating redundant logic, ALERT prevents the designer from making full use of the properties of the components used. This is because the designer does not have access to the outputs of implied components. Conversion into hardware using Gerace's method is achieved by obtaining a set of flow-tables and realising them by using special types of flip-flops (F,G and SR flip-flops. See (40)). Once the tables are obtained they can be simplified by using sequential network theory. Gerace also claims that when simplifying the flow-tables (state reduction), redundant registers may be detected. However, it is difficult to accept this claim. The flow-tables are formed such that each bit of a transfer register will occupy a flip-flop in the corresponding table. If state reduction is achieved to such an extent that a flip-flop is eliminated, then the eliminated flip-flop must belong to a transfer register. As each bit of a transfer register can appear only once in a micro statement ($X \rightarrow P_i$, $Y \rightarrow P_i$ is not allowed) this would mean the elimination of an OW, which cannot be possible. On the other hand any elimination of the registers on the left hand side of the arrows may only effect the number of inputs to the LLC circuit and this can be detected much more easily by examining the OW's before converting them into flow-tables.

(vi) Design Evaluation.

All four methods described in section 2.1 are suitable for simulation. However, in the cases of LDT and ALERT, the system

description has to be processed by the corresponding translator to sort out the timing details, before simulation can be possible. Schorr's system also allows for the translation of the Boolean equations into register transfer statements for analysing the design. However, the problem of sorting the resulting transfer statements such that meaningful micro programs can be formed, is still unsolved.

(vii)- Relation to other Methods.

The four CAD methods discussed above are by no means the best linguistic methods but were selected to indicate the general techniques used. Similar methods can be found in the literature. Chu's ALGOL like design language (27) is a satisfactory source for documenting the behaviour of a digital system but structural details cannot be specified using this language. The language uses ALGOL like statements to describe operations, and statements can be conditioned by 'labels' which can be Boolean expressions. The control unit can be specified as a separate section generating the necessary signals to enable the execution of operations. Reserved names are used to indicate specific operations such as ADD, DECODE,..., etc. but when such names are used more than once, it is not clear whether the same adder (for example) or a different one is used to perform the operation.

The language described by Stabler (81) is simple and gives enough structural information to build a digital network. A network is defined in terms of logic gates and registers or storage elements which are treated as 'primitives' since the structure will not be analysed beyond these elements. More complex gate types or sub-machine types (Macro types) can be specified in terms of the

primitives and other macros. Hence a network is specified in a 'bottom-to-top' fashion. Because of the amount of detail involved in making the specification, the language could be used for designing only small size digital systems.

The Digital Design Language (DDL)(35) is a very powerful language but rather complex. The structure of a digital system (or a part of it since the language allows for partitioning) is described by using a variety of 'Facility Declaration' statements and operations are specified with the use of 'Operation' statements which can be conditioned. Parallel operations can be specified by explicitly declaring control counters and setting them to correct values. Duley and Dietmeyer (36) describe the techniques used in translating a DDL description into Boolean equations. The methods used are rather inefficient since a number of passes over the description is required and the entire description has to be re-translated whenever any change is made to the initial specification.

Another design language is proposed by Baray and Su (11). It allows the designer to partition his design into units which are capable of operating independently or with a minimum amount of communication with each other. In this language the structure and control parts of a digital system are separated and expressed in different sections. Operators are used to describe the function of each unit working under the control of control signals. Units yet not designed but whose input-output transfer function can be expressed in another algorithmic language (e.g. PL/I) can also be used. Parallel operations can be specified with the use of BRANCH and JUNCTION elements which are used to activate more than one control branch and bring together more than one control branch, respectively.

The control section of the language is simple and efficient but its structural part is not so because of the unrealistic representation of connections between the operators and links. The interconnection of operators is indicated by using links which can be links having memory (registers) or links with no memory (terminals). Since direct connections between the operators (i.e. Operator outputs → Operator inputs) are not allowed, even a simple feed back has to be made by directing the operator outputs to a link (terminal) and directing the link to the operator inputs.

Schlaeppi's LOTIS (79) language is another example of a design language. In general, it is an adaptation of the ALGOL language with the exception that a machine description in LOTIS does not have the ALGOL block structure since the whole description is made as one block. The hierarchy of the structure of an object machine is achieved by the use of MACRO definitions whereby more complex elements can be defined in terms of the primitive operators (and, or,...). The linguistic constituents of the language correspond to actual elements thus making a specification in LOTIS more realistic than would otherwise be the case. A machine description is made in two parts, namely the declarative part and the procedural part. Transfer statements are used to store data values and the timing of transfers can be specified by either declaring a time interval for each of the operators involved in the transfer, thereby defining a 'transfer duration' (the sum of operator times over the longest path in its structural tree) or else by explicitly declaring 'transfer intervals'. Concurrent assignments and control statements can be grouped together to form a 'Step' and a list of steps forms a 'Sequence'. The time

relations between the steps of a sequence can be expressed in three modes;

A- Asynchronous mode: Step interval is equal to the duration of the longest operation in the step.

B- Fixed-delay mode: Explicit delcarations are used to define the step interval.

C- Synchronous mode: The first step in succession is activated as soon as the step condition (which can be obtained from the output of a clock generator) becomes true.

It is felt that putting the complete description of an object machine into a single procedural block and the detail involved in specifying some of the timing relations are the disadvantages of LOTIS, which otherwise is a simple and useful language.

The Sequence Chart (78) of IBM displays timing and sequencing information in a graphic manner (it is more correct to say lists, since output is not on a graphic display). This can represent only the control part of a digital network and its form is unsuitable for most available input-output equipment. The sequence chart forms only a part of the logical design system as it exists within IBM and on its own is of little value as a design aid.

Bell & Newell (18) describe two descriptive systems for the top computer-system (PMS) and the program level (ISP). However, these two systems are used only for system documentation at the respective level and a digital system design method, accepting as input the architecture of the object machine in the PMS or ISP notation, has yet to emerge.

Finally, it will be proper to conclude this section by mentioning some languages which are used only to simulate digital networks. None of these languages are used to describe digital systems with the purpose of generating the logic to realise them. The level of simulation is not the same in all simulators and some simulate the described logic at the gate and flip-flop level (26), (45), (83) whereas others provide a functional simulation (5), (58). Three valued simulation (50) is another alternative.

2.2. Methods Using Flow-Charts.

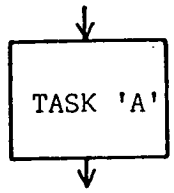
Although some experimental CAD systems have been proposed for inputting logic diagrams into a digital computer (67), graphical approaches to logic design have largely been used only in the area of specifying digital systems by drawing a flow-chart of its operations. The three methods described in the following sections all use the flow-chart approach and in each case design synthesis is achieved by using special components which have a one-to-one correspondence with the flow-chart symbols.

2.2.1. Control Point Design Using Modular Logic.

C. Rey (75) has proposed some general purpose building blocks (modules) to convert a flow-chart design into a logical network. The resulting network does not perform any "Data Operations" but merely generates the correct sequence of control signals to initiate them. Hence, this approach can be used only in designing control circuits. The modularity of the approach comes from the one-to-one correspondence between the flow-chart symbols and the components used to realise them. The four basic flow-chart symbols are described below.

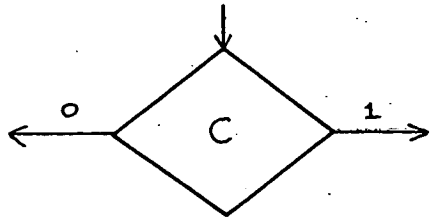
(i)- The "Task Box" is used to represent a set of elementary operations.

e.g.



(ii)- A "Decision Box" is a point in the flow-chart where the control sequence is directed towards one of the two output branches depending on the logical value of a condition.

e.g.



(iii)- The "Circle" has two forms. A normal circle (called the "Circle") is used to merge two or more control signals (i.e. logical OR). A "Dotted-Circle" allows the control flow to continue only when all its inputs are activated (i.e. logical AND).

e.g.

(iv)- The "Fan-Out Point" is a point where control is split into two or more parallel branches.

e.g.

Figure 2.5 shows a typical flow-chart. Since only NAND gates are used in designing the modules, it is assumed that all control signals are normally at logic "1" and they become a negative pulse (1-0-1) when activated.

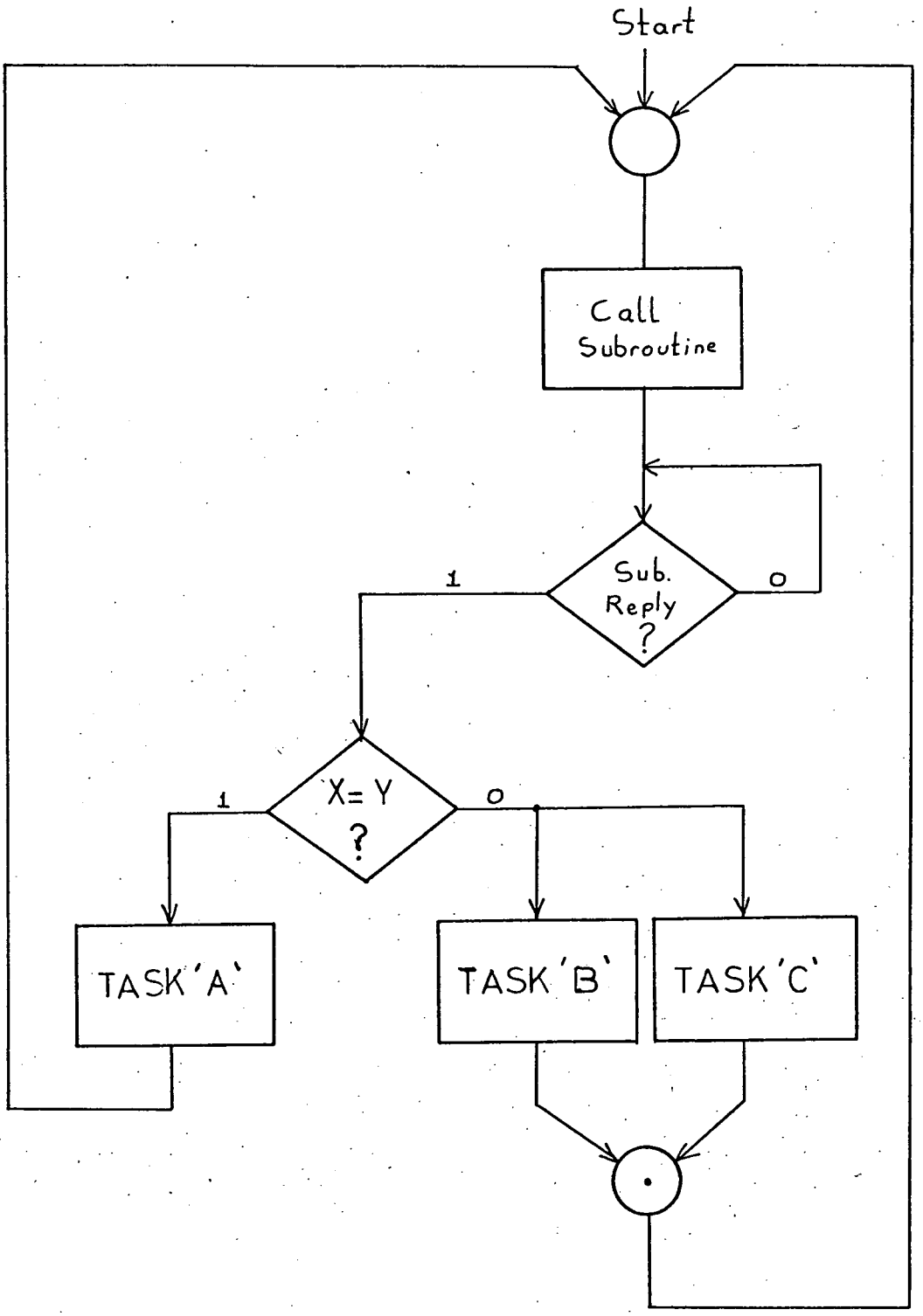
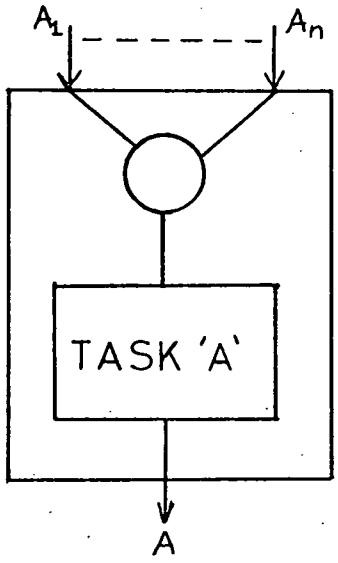


Figure 2.5 Flow-Chart.

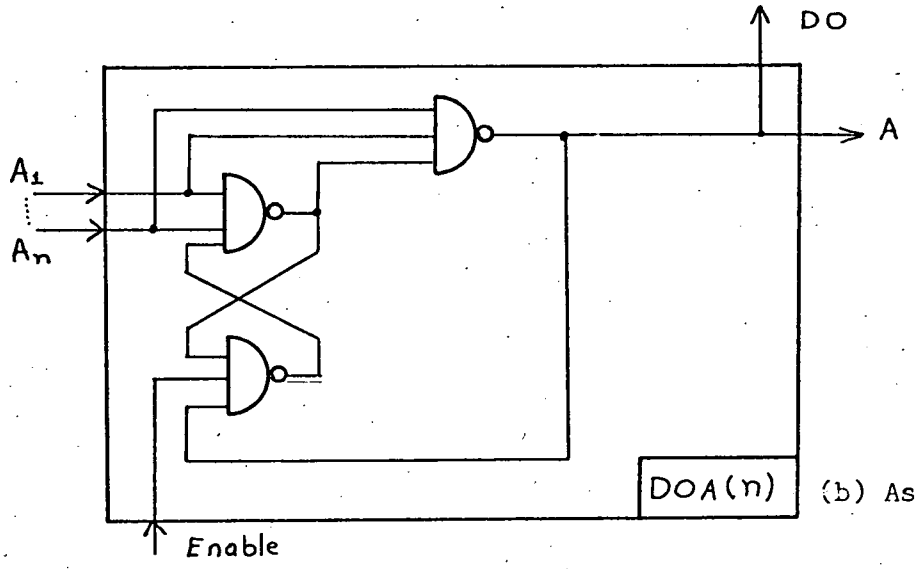
The basic building blocks are described below.

(a)- DO Module.

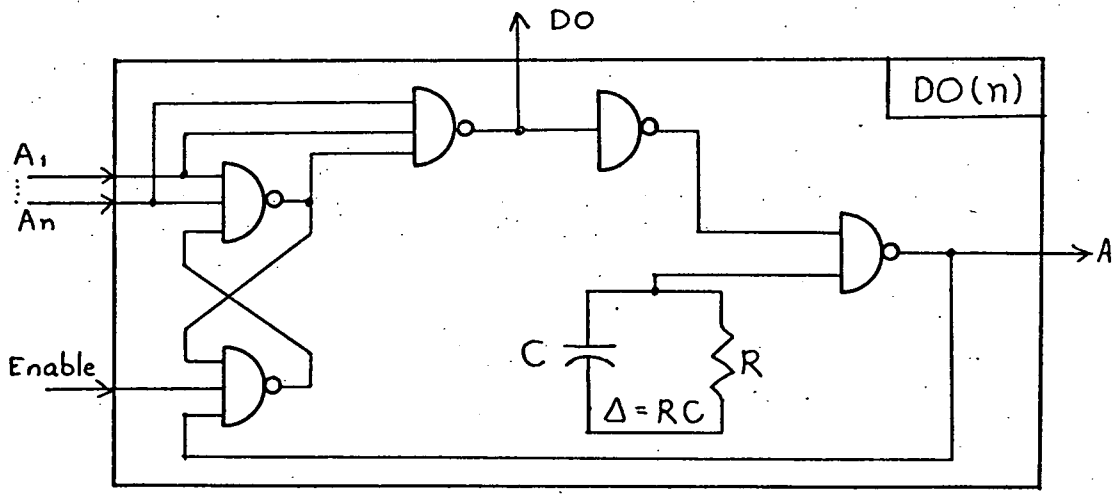
The DO module realises a task-box. Note that it only generates the signals to activate an operation and does not perform the operation itself. There are two forms of this module. The DOA(n) module is used for asynchronous operations. Figure 2.6 (b) shows its design. Assuming that all the A_i inputs are initially at logic "1", its operation is as follows. When one of the inputs, A_i, becomes "0", the flip-flop (cross-coupled NAND gates) is set. When A_i becomes "1" again, a negative pulse is generated which is used to activate the operation to be performed and the A output feeds the next module. The output signal is also fed back into the flip-flop and this resets the system so that the outputs become "1" again. Hence both the inputs and outputs are negative pulses. For "Pseudo-Asynchronous" operations the second form of the component is used. In this form, it is called the DO(n) module and its design is shown in Figure 2.6 (c). Its operation is the same as the DOA(n) module except that the A output appears after a time delay which simulates the duration of the task to be performed.



(a) Flow-Chart



(b) Asynchronous

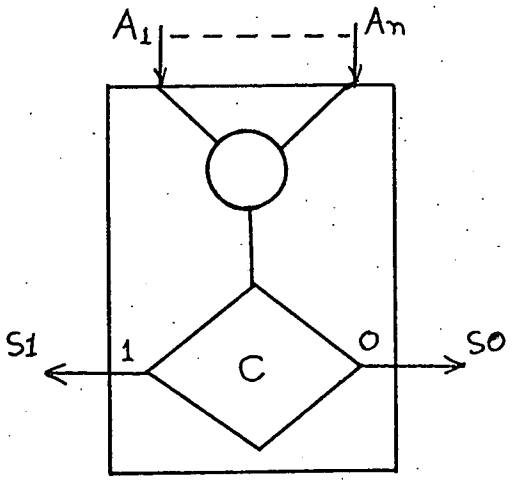


(c) Synchronous

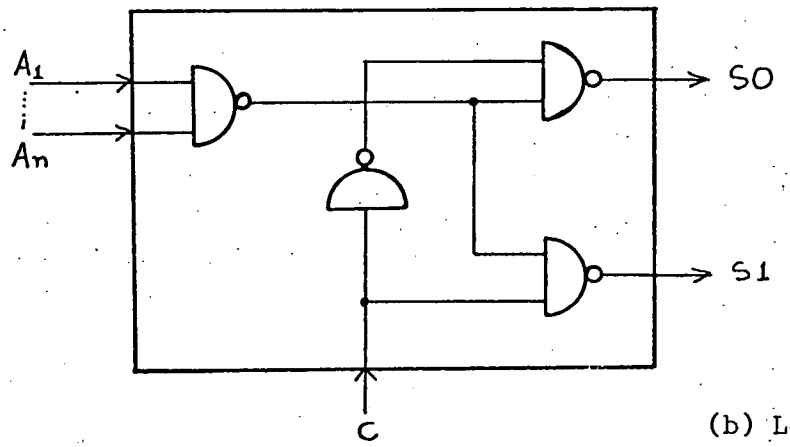
Figure 2.6 The DO Module.

(b)- Sequence Module

The S(n) module realises a two-way branch. Its input conditions are the same as for the DO module and the control flow is directed to the SO or S1 output depending on the value of the C input. Note that C must be set to the correct value before any of the A_i inputs are activated.



(a) Flow-Chart

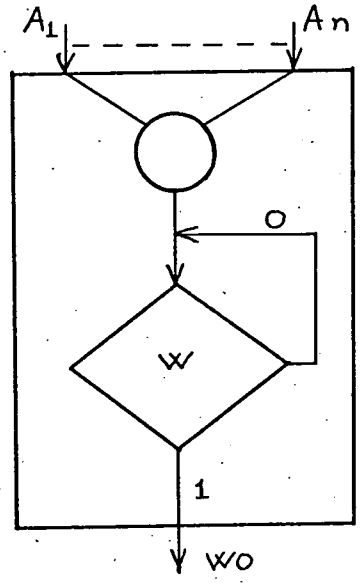


(b) Logic Design

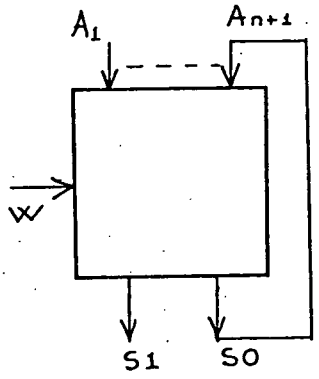
Figure 2.7 The Sequence Module.

(c)- Wait Module

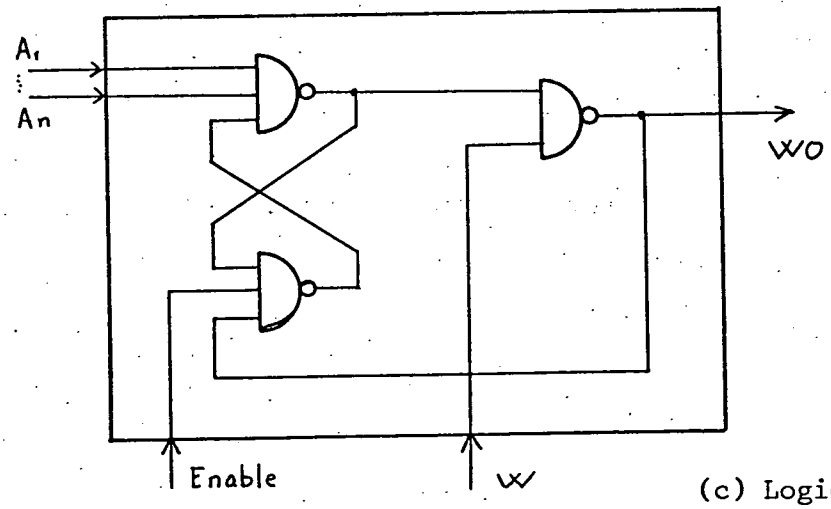
The wait module has two forms. The first one is called Wl(n). In this form, Wl(n) can be realised by using an S(n + 1) module but because of its frequent use, a cheaper design has been proposed. Figure 2.8 shows its realisation and the operation of the component is clear from Figure 2.8 (a).



(a) Flow-Chart.



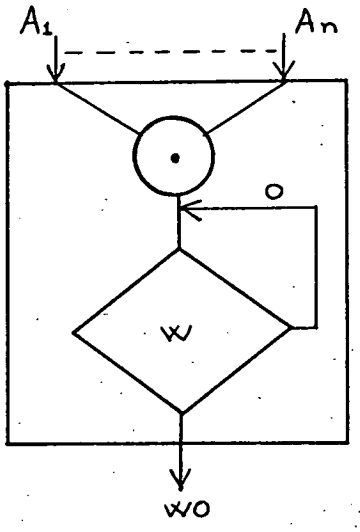
(b) S(n + 1) Realisation



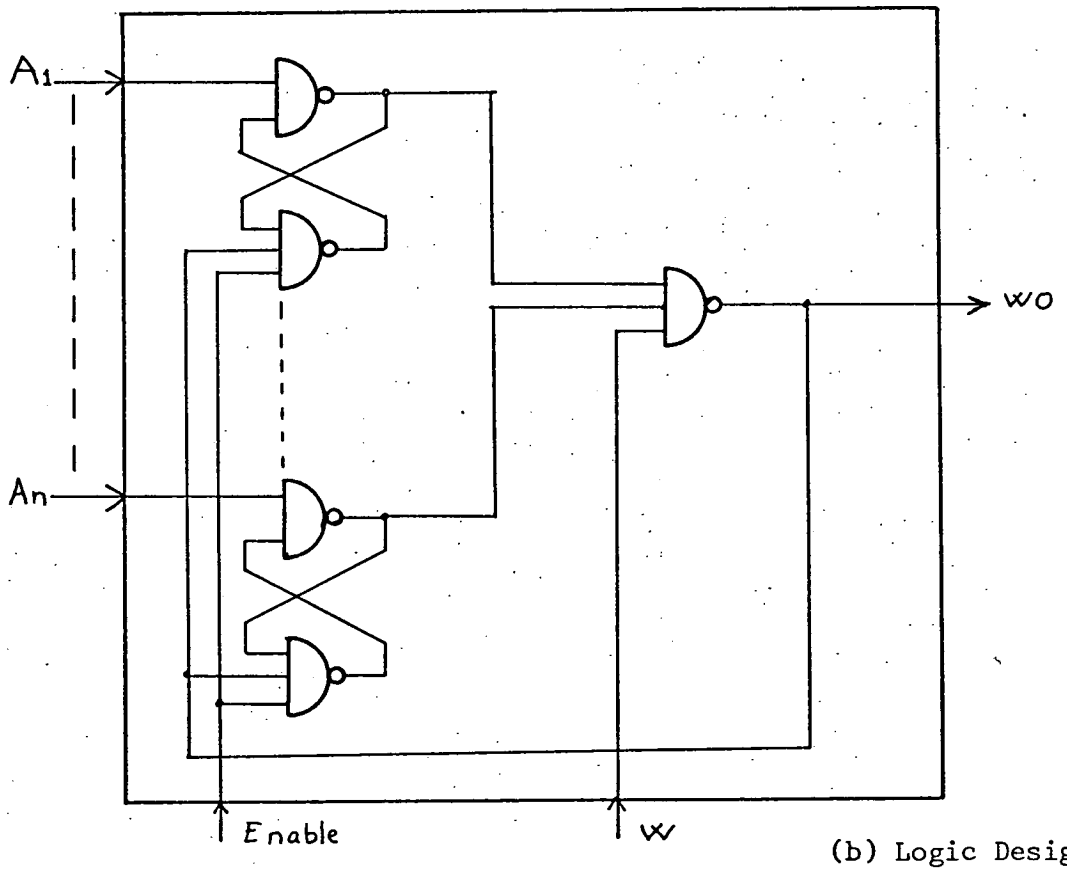
(c) Logic Design

Figure 2.8 Wait Module.

The second form of the wait module is called W2(n) and its operation is similar to that of W1(n) except that a pulse appears at the output only if all A_i inputs are activated and W is at logic "1".



(a) Flow-Chart.



(b) Logic Design

Figure 2.9 Second form of Wait Module.

A very useful characteristic of the modules described above is that the input and output pulses are all of the same polarity (1-0-1) so that the outputs of one can be directly connected to the inputs of another. For asynchronous and pseudo-asynchronous operations the DOA(n) and DO(n) modules are used, respectively. For synchronous operations, a DO(n) followed by a W(1) (note that W1(1) is identical to W2(1)) should be used and the clock pulse should be connected to the "W" input of the wait module.

Figure 2.10 shows the realisation of the flow-chart in Figure 2.5. The one-to-one correspondence between the modules and flow-chart symbols is evident from the realisation. To obtain a cheaper (and faster) design, the flow-chart has to be properly organised. C. Rey gives some basic rules used to simplify a given flow-chart. Using one of these rules, the design given in Figure 2.10 can be simplified by setting the smaller of delay values Δ_B and Δ_C equal to the other and eliminating the W2(2) module by connecting the outputs of the task-boxes B and C to the subroutine call unit.

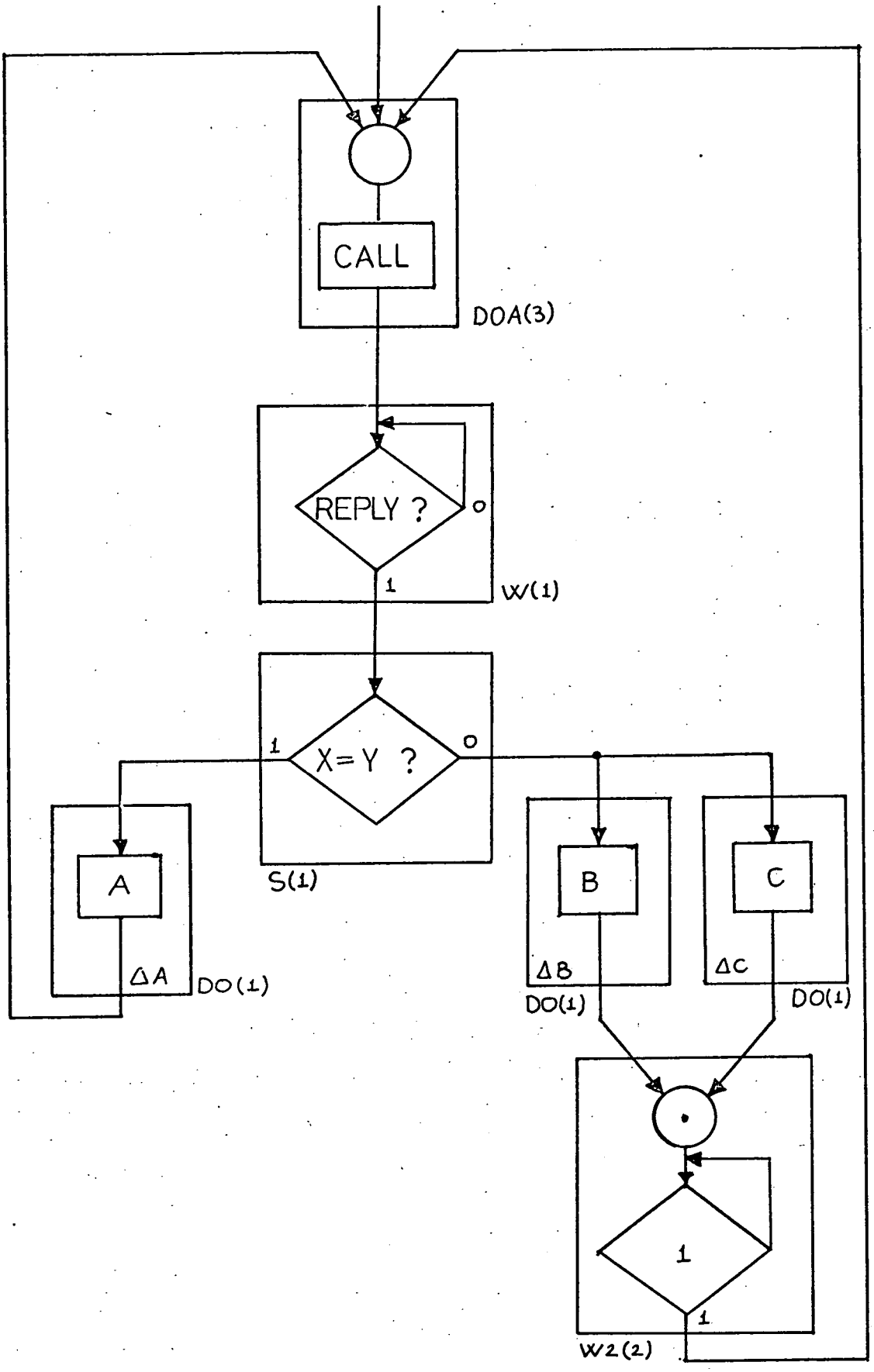


Figure 2.10 Realisation of flow-chart.

2.2.2. The PDP-16 Concept.

PDP-16 is a custom designed general purpose digital computer which is assembled from functional hardware options (modules). Actually, a PDP-16 is a concept rather than a computer since a PDP-16 does not exist until it is designed to a particular specification. The functional options were originally designed by G. Bell using the name "Register Transfer Modules (RTM)" (15). Their production version has been designed by DEC using the name PDP-16 (2). The hardware options fall into two categories.

(i)- Functional (data) Options

The PDP-16 functional options perform arithmetic and logical operations and data transfers on 16-bits of data. Communication between the registers and the General Purpose Arithmetic (GPA) Unit is done through a 16-bit bi-directional data bus. All units operate in asynchronous mode and contain the necessary logic for the timing of operations. Every PDP-16 should contain the KBS16 bus sense unit (the basic bus structure) and the KTM16 bus terminator (resistive network to terminate the data and timing bus lines). All other modules are optional and these include memory, registers, GPA, I/O interface and flags.

(ii)- Flow (Control) Options.

The flow options are used to design the control unit of a PDP-16. The CU is specified by a flow-chart where each symbol has a flow-option counterpart. The basic options are "Branch" units (two-way or eight-way), "OR" units to merge control signals, "Evoke" units which represent one complete data operation and "Subroutine

Return" units to call a group of operations and continue when they are completed.

A PDP-16 is designed by first selecting a suitable set of data options and drawing the flow-chart of the CU which will control their operation. There are certain rules to be observed when drawing the flow-chart. First of all, only the selected data options should be referred to (i.e. if "A + B" is required, the corresponding options which provide two registers and an adder should be selected among the data options). Other restrictions are, no output of a branch unit can be connected directly to its input without first going through an evoke unit, wired -OR connections are not allowed and there must be at least one evoke unit between any two subroutine call elements which refer to the same group of operations. A broader explanation of these rules can be found in the reference (2). When the flow-chart is ready, all data and control options are given a unique number to identify them and then an input file is prepared. This input file, to be processed by DEC's Chartware design package, contains the following information.

(a)- Socket Definitions.

A PDP-16 can be constructed to occupy up to six rows and thirty two columns of sockets. The positions of the sockets that the PDP-16 modules can be plugged in, is defined by giving the row name and the first and last socket numbers. For example the first ten sockets on the last two rows are assigned by typing

E, 1, 10

F, 1, 10

(b)- Functional-option Definitions.

All functional options used should be specified by giving the module name and the number which identifies them.

e.g.

MS16-A, 3, 14

DB16-C, 9

identifies units numbered 3 and 14 as registers and unit 9 as an input interface. Additional space can be reserved for future bus extensions by using the module name MOD-2.

(c)- Flow-option Definitions.

The flow options are identified in exactly the same way as the functional options.

e.g.

KEV16, 5, 7

KB16-A, 10-12

identifies the units numbered 5 and 7 as evoke modules and units numbered 10, 11, 12 are two-way branches.

(d)- OR Gates and other Modules

The same format as above is used to identify OR gates. Also dummy modules to be replaced by future modifications can be specified by using the module name MOD-1.

e.g.

KOR-16B, 1, 2

MOD-1, 20, 21

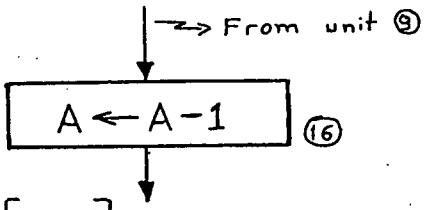
Units 1 and 2 are defined as four input OR gates and units 20 and 21 are dummy modules.

(e)- Wiring Data.

The connections between the flow-chart elements and the functional options are defined by using abbreviated names for the inputs and outputs of components, using the following format.

Output, input, input,.....etc.

For example, if the functional options are numbered such that the GPA is unit 30 (typed as KAC16,30), register "A" is unit 35 (typed as KAR16, 35) then the following section of a flow-chart will be coded as shown below.



9,16,35 [LDA] , 30 [A - 1]

This entry has the following meaning

9, The output of unit 9 is connected to

16, the input of evoke unit, number 16,

35 [LDA] , the "load data into the A register of GPA" input of the register unit, number 35,

30 [A - 1] the "decrement register A by one" input of the GPA unit, number 30.

(f)- Flow-chart Symbol Definitions.

This part of the input file is necessary only if a printout of the flow-chart is requested.

The Chartware programs analyse the input file and generate a set of documentation listings showing the position of each option on a card, the socket to which the cards are to be plugged in, a

pin-to-pin connection listing and the order in which connections are to be made for the back-wiring of sockets. Control tapes for automatic wiring machines can also be obtained.

In summary, a PDP-16 is designed by selecting some functional options and drawing the flow-chart of the control unit. The units used and the global connections between them, are then given as input to the Chartware programs which produce listings showing the positions of cards in the sockets and the back-wiring connections.

2.2.3. The LOGOS Project.

The LOGOS project (41), (46), (47) is an attempt to create a CAD system for the integrated software/hardware design of large scale computer systems. A prime objective in designing the LOGOS system has been "a uniform approach to system design (both software and hardware) at several levels". This implies that software and hardware will be specified in the same manner and whatever level the designer is working, the method of specification will be the same. A further objective has been, to be able to handle modifications, without upsetting the unconcerned groups working on the same problem. To ensure that all designers obey the ground rules, the design environment is an on-line interactive time-sharing computer system with a common data base, under the control of a Primitive Data base Management System (PDMS) (73). Input to LOGOS files is prepared in graphical form, on display units. A database file can be local (unique to a single user) or global (sharable by different users) such that separate units (or the different level design of the same unit) designed by different people, can be linked together. Once the input files are ready, the total system performance can be

checked and the factoring of software and hardware can be done.

In LOGOS, a process is represented by drawing a "Directed Graph" showing the flow of information (77). Separate graphs are drawn to represent the data flow and the control flow and these have been called the "Data Graph" (DG) and Control Graph" (CG), respectively. A graph is drawn by joining the "Cell's" (represented by squares) to "Operator"s (represented by circles) through edges (represented by arrows). The cells and operators must be connected to each other, in alternating order. The start and end points of a graph are indicated by the "Blockhead" and "Blockend" symbols (this is not necessary for the data graph). Any activity is represented by a CG and an associated DG. Figure 2.11. shows a typical CG-DG pair.

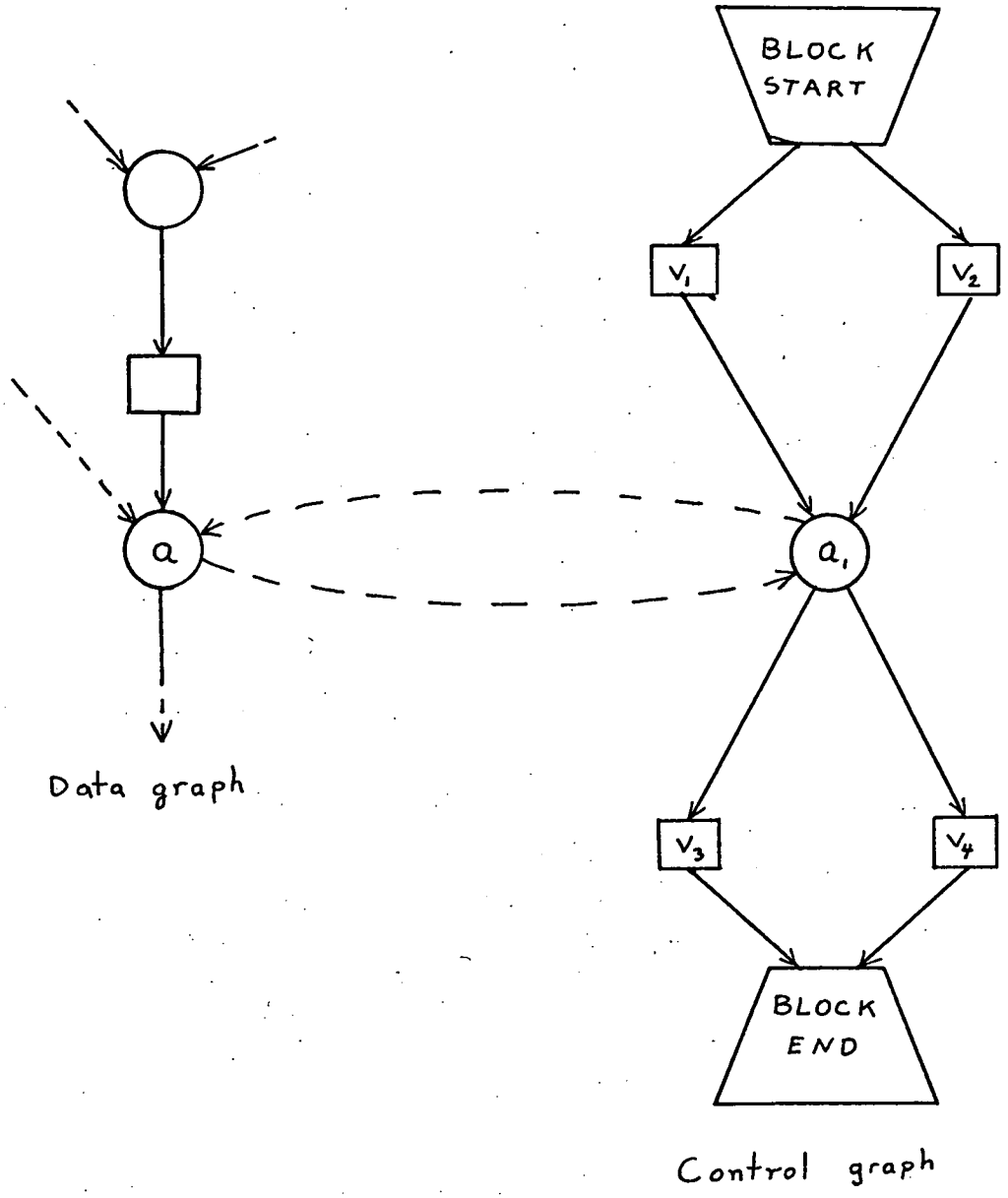


Figure 2.11 A Directed-graph.

Figure 2.11. can be interpreted as follows. Each cell is given a unique number for identification and inside the cell its value is written. For digital systems v_i can be 0 or 1 and a "Cell-Operator-Cell" sequence can be thought of as an "Input terminal-Component-Output terminal" sequence in a component definition. When its input cells contain the correct values, and the output cells contain zeros, operator "a_1" is initiated and activates operator "a"

in the data graph. Upon completion of its operation on the data values, operator "a" sends a termination signal to 'a_1", which then generates new outputs and the control flows downstream. Apart from this downstream flow of signals, there is also an invisible upstream flow since when any operator fires (i.e. changes its outputs), it also changes the values in its input cells. For non-digital systems, the updating of the input cells is done by subtracting one from its value, if it is not zero. For digital systems however, an operator resets its input cells to zero, after it fires. With this upstream flow, the operator prepares itself for new stimuli. The transfer function of each operator can be expressed in vector form. A "State Vector" q is defined as the numbers in each cell written in cell order. Thus, the state vector for operator a_1 might be;

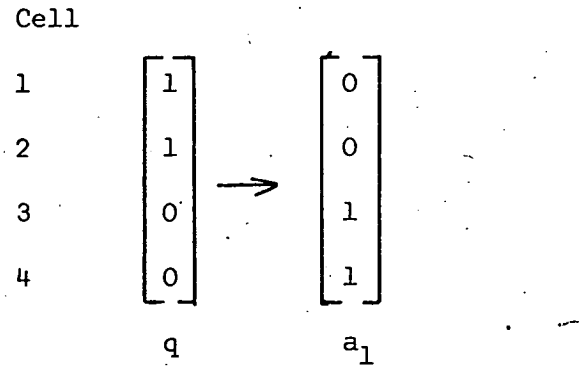


Figure 2.12 State Vector of Operator a_1.

The operators may represent self-contained structures (Blockhead, ..., Blockend) so that at any level of design the operators may be replaced by their detailed representation and vice-versa. The blockhead, blockend structure is analogous to the ALGOL "Begin-End" block. A parameter passing method is used when starting a new block.

The lowest level control operator is called PCON. Figure 2.13 shows a PCON and its transfer function.

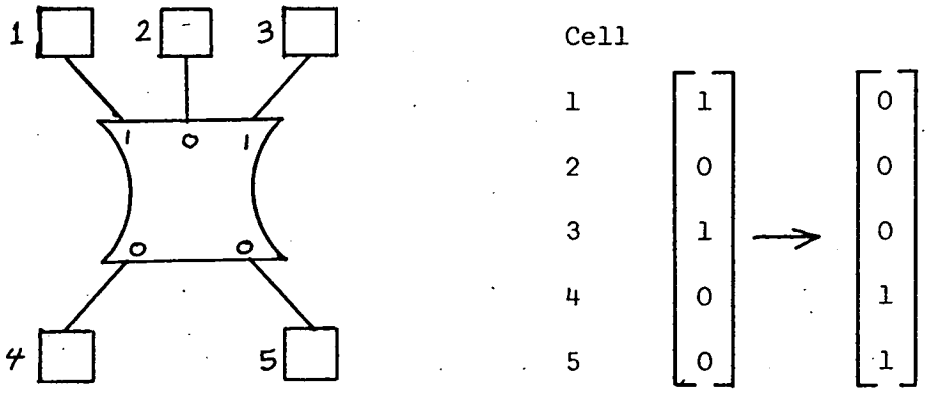


Figure 2.13 PCON and its State-vector.

The PCON's can be realised in hardware by replacing the cells with flop-flops. Figure 2.14 shows the realisation of the PCON, given in Figure 2.13.

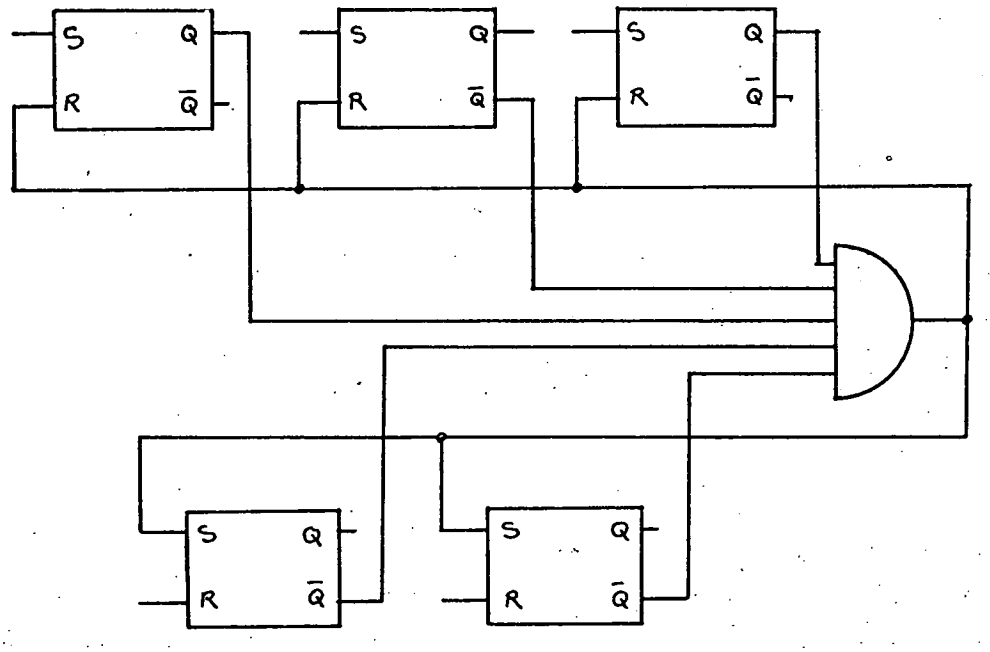


Figure 2.14 Realisation of PCON.

The PCON's may operate in the synchronous or asynchronous mode but communication with other PCON's is done asynchronously. This is essential since any operation may take an arbitrarily long time and others

have to wait until it is completed. Synchronisation between a PCON and its corresponding data operator is accomplished by partitioning a PCON into an "Initiation" part which activates the data operator, and a "Termination" part which is activated by the data operator when its operation is completed. This is illustrated in Figure 2.15.

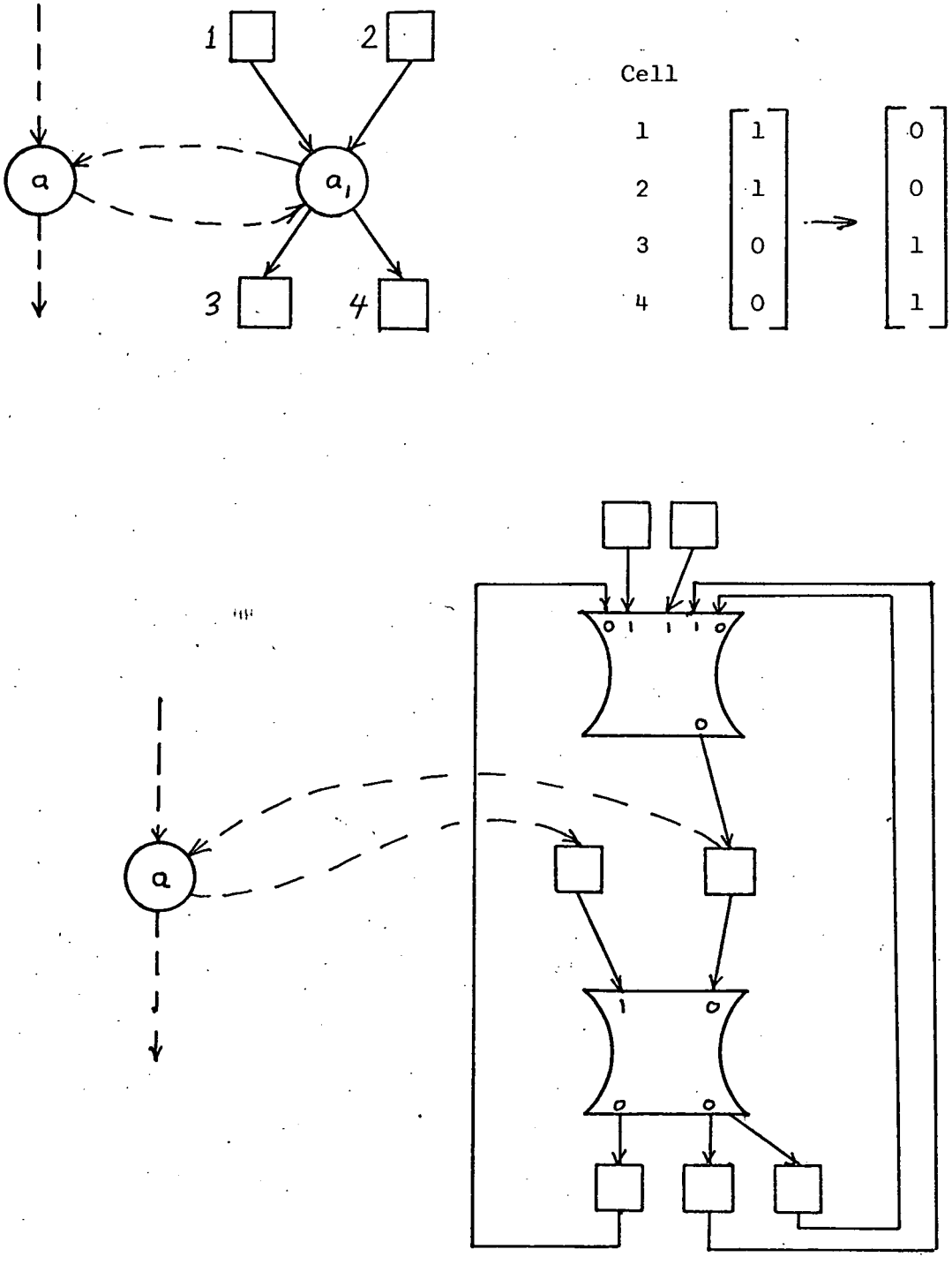


Figure 2.15 Data Synchronisation of PCON.

It is apparent from Figure 2.15 that data dependent operations can be specified by making the necessary link between the control and data operators. Thus, the basic LOGOS functions (AND, OR, BRANCH) can be defined and implemented in hardware. More complex operators can be defined as MACRO operators, using the simpler ones. As the method of representation is same throughout, the designer can work at any level or change to a lower or higher one without difficulty.

In practice, even though designing the control graph is more difficult, the data graph presents more problems in representation. This is because the control variables are mostly single bits whereas data variables are groups of bits (e.g. a register). To prevent too much detail in drawing the data graph, text handling facilities exist whereby certain types of information can be input in textual form. This applies also to specifying the links between the data and control operators although drawing "dotted-lines" to represent these links is also permissible.

When the input files are ready, the design is analysed. Karp & Miller (56) has shown that "a schema is determinate if, given an initial state and an initial set of values, each data location has a fixed sequence of values". This condition is shown to be equivalent to,

(a) no two data operators should be concurrently enabled to store data into the same location.,

(b) no data operator should be enabled to store data into a location while another operator is enabled to read values from the same location.

Instead of an exhaustive simulation to verify a design, in

LOGOS, the system is checked for determinancy and if it is found to be determinate, then it is guaranteed to operate as specified (see section 2.3).

2.2.4. Discussion.

The common philosophy of the three methods described in the previous section is that they all separate the data and control operations from each other and the control unit is specified as a flow-chart. The way in which the data part is designed and linked to the control unit is different in each case. In fact, C. Rey (75) does not describe this side of the design process at all. However, the modules described in section 2.2.1 are quite useful in designing control circuits. The fact that all control inputs and ouputs are pulses of the same polarity (1-0-1) makes these modules ideally suited for combined use. By changing the NAND gates to NOR gates, the reverse polarity (0-1-0) pulses can be obtained. Communication with data elements is done by "level" signals. These include the "wait" and "branch" inputs of the W(n) and S(n) modules, respectively, and DO output of the DOA(n) and DO(n) modules. This latter module is very useful since it simulates the task duration. Various methods of calling 'subroutines' have been illustrated in the reference. When the task duration is not known, a "Wait" module following the DO module is used.

The modules themselves are optimal but there is no guarantee that the flow-chart is so, or even that it is well behaved (determinate). Apart from some basic principles, there is no way of simplifying a flow-chart. The simple rules used to this end can be found in the reference. Since there is no mention in the

reference cited about the computer aids used in describing and simplifying the flow-chart and turning it into hardware, an objective criticism of the techniques described is not possible. Yet, the simplicity and efficiency of the modules justify their being included for discussion.

Going one step further than the modules described by C. Rey, is the PDP-16 concept. The PDP-16 modules form a complete set of control and data components able to communicate with each other. Their switching circuit details can be found in (16). The delay values of the elements are quoted in (2) and for the flow-options, they are between twelve (for the OR gate) and sixty two (for Evoke) nano seconds. The PDP-16 concept is commendable for the set of comprehensive options which from a system designer's point of view are simple and meaningful and it is possible, with the aid of the Chartware design programs, to bring these options together to form a complete digital system. The flow-options are simple and adequate. However, the functional options are rather restrictive. Firstly, a PDP-16 may have only one "bus" through which all data transfers are done. Also, the GPA can perform operations on the contents of A and B registers alone. Operations on any other registers can be done by transferring their contents to the A and/or B register and copying the result back. In a later version (see reference (16)), a new module has been introduced which solves this problem to some extent by allowing operations between the A register and one of the registers from a 16-address scratch-pad memory. In spite of the addition of this new module, the data part of a PDP-16 is very rigid in structure and the designer cannot control the operations; he can only initiate them.

The Chartware programs are very useful but the form of the input file is rather off-putting. The control unit of a PDP-16 is expressed as a flow-chart and then coded for input to the design programs. By coding (i.e. turning the diagram into a set of letters, digits and punctuation marks) the whole essence of the flow-chart is destroyed. This step is laborious, uncreative and very error-prone. Furthermore, the data part of a PDP-16 is hidden very deeply under the specification. The only reference to the functional-options is from the input file as connections to the specific pins of the modules. The designer can draw a block-diagram of the structure of a PDP-16 but that would be more an academic exercise than a creative process.

In spite of its certain draw-backs, the PDP-16 concept is quite useful since with some help from DEC* one can design a small size digital computer in a relatively short time. It is very suitable for educational and some special-purpose computer applications.

Project LOGOS is the most general approach among the flow-chart methods described in the previous sections. The method of specification can be used for defining both hardware and software systems. In fact, the aim is to specify a system as a whole, analyse it and then do the software/hardware split. The control-graph, data-graph pair is a special flow-chart which separates control-flow from data-flow.

(*) DEC used to provide a full PDP-16 design service to generate, process and construct PDP-16 designs, as required. This service no longer continues and the Chartware programs are not supported anymore. However, the PDP-16 modules can still be ordered from DEC.

Operations are carried out by special operators that activate the next set of operators only if they are ready to accept the activity signals. Though the principles are equally applicable to both software and hardware, it seems rather doubtful that software engineers will be willing to use them. Several reasons could be given for this.

(i)- Though in hardware systems, race conditions may occur and therefore it is a sound principle to activate components only when they are ready to receive data, such is not the case for software since computer programs are executed sequentially. Thus, except for multi-processor systems, interrupt handling routines, or in other real-time applications, the (LOGOS) approach will not be beneficial from that point of view.

(ii)- Building and testing software systems is relatively cheaper than hardware systems. A software engineer does not fear too much about his program not working correctly at the first run (in fact he hardly ever expects this to happen). By using simple techniques such as inserting PAUSE or WRITE statements in his program, or by using special debugging programs (e.g. DDT), programs can be corrected easily. In most cases, program bugs will be such things as using a wrong index, forgetting to set a variable to a certain value, passing wrong arguments to a subroutine,.... etc.

(iii)- A computer programmer can express the logic of his program much more easily and elegantly, by using the normal flow-chart diagrams. It is doubtful that he will want to part from such "free style" drawings.

(iv)- Most programmers draw a flow-chart only after the program is written. Even if they were convinced to do otherwise, it is difficult to estimate the balance of the overheads (increased computer usage when preparing a flow-chart, cost of special display terminals,.... etc.) versus the speed and accuracy with which programs may be written.

For hardware systems, however, the LOGOS approach is very promising. As mentioned earlier, the principle that "operators should not fire unless others are ready to accept information", is essential to any LOGOS design. This upstream flow of information goes through the same paths as the downstream flow of control. It may be a good idea to separate these two sets of signals from each other. As the current demand is for "Testable" hardware, providing the test (or maintenance) engineer with a distinct set of signals indicating the status of each operator may be extremely useful. Appendix A describes a proposal for a different representation of LOGOS diagrams and shows how the operators can be designed to incorporate the changes proposed.

Another philosophy adopted in LOGOS is to avoid exhaustive simulation by replacing it with the Karp & Miller algorithms to check if a system is determinate. Even though the upstream flow of information eliminates races along a single branch of the control graph, there may still be races caused by different control flows along separate paths (especially in parallel operations). The use of analytical techniques to detect such cases is appreciated. However, a final check which shows that the specification is correct, is still necessary.

A designer implementing a particular algorithm should be able to see his algorithm in action and make sure that it is correct. Therefore, in addition, to the determinancy test, a simulator is essential.*

2.3. An overall criticism of existing CAD methods.

A great majority of existing CAD methods applied to logic design, use formal languages in specifying digital systems. Probably, the reason for this is the relative ease of designing new languages to better existing ones, compared to finding new methods of describing digital systems. Languages are best suited to describe sequential operations. However, most digital systems operate in parallel. Even in those computers, which seem to operate sequentially, at the micro program level, there is usually more than one activity taking place at a given time. The concept of time, which is alien to languages, has to be introduced to them in the form of variables (or arrays for counters) conditioning the operations and it is here that most design languages fail. There are some design methods using formal languages (e.g. LDT) which eliminate this variable from the initial description, but this is 'hiding' the timing from the user, rather than eliminating it and produces some undesirable side effects. For example, in LDT, if a new operation is inserted into the language description, then all operations coming after it and sharing the same registers, will have their timing signals changed. As LDT generates designs working as a pipe-line system, this may cause a complete re-shuffle of operations.

(*) Through private communications with a member of the LOGOS team (Prof. F.G. Heath), the author has been informed that the problem of writing an equivalent of a simulator for this purpose, is under consideration.

The second point where languages are deficient is in defining the structure of the target digital system. Structural information is best expressed in block-diagram form. Using language statements which name the components and give a list of connections among them, while being suitable for computer processing, is of little value for communicating ideas among designers.

In spite of these shortcomings, formal languages are of great importance in CAD systems because of their descriptive powers, and if carefully designed, their unambiguity.

Design methods using flow-charts are relatively new. This is mainly because special terminals are required to input the flow-charts into a digital computer. The PDP-16 approach does not require drawing diagrams on a display terminal and expresses the flow-chart in language terms. The LOGOS project is the only attempt that the author is aware of, aimed at designing large scale systems by using a directed-graph approach. As the techniques have not yet been fully implemented, it is difficult to make a true criticism of LOGOS but the approach seems to be a viable proposition.

The common characteristic of flow-chart methods is the separation of data and control operations. This is a sound principle since it simplifies the design task and once the overall structure is defined, one can try different control organisation to find the best solution.

Chapter 3: A Design Philosophy.

Most existing CAD programs take a description of the target machine in some sort of language form, and use it to determine the hardware required to realise the operations. The key point to any approach of this type is the method of specification since it must be concise and simple but yet, should carry enough information to realise the description in hardware. It was observed in Chapter 2 that introducing some hardware notions into the design specification simplifies the task. Register transfer languages are a good example of this but they are restrictive since the designer is forced to think only in terms of register structures and yet, other structures exist. Therefore, a method which allows the designer to use various different types of logic structure is required. Extending this idea further, a "Design Space" can be defined which consists only of "Components" and "Connections". All required operations can then be defined by the corresponding components which will realise them. Since the operations to be executed by a digital system can be divided into those of "Data" and "Control", it is logical to specify and realise them separately. Consequently, the design space can be divided into a "Data Space" (called the D-space) and a "Control Space" (called the C-space), each with its own components and connections. The two spaces are then linked together by the "Control Signals" which may originate from either space (so that data dependent operations can be defined). Figure 3.1 illustrates these relations.

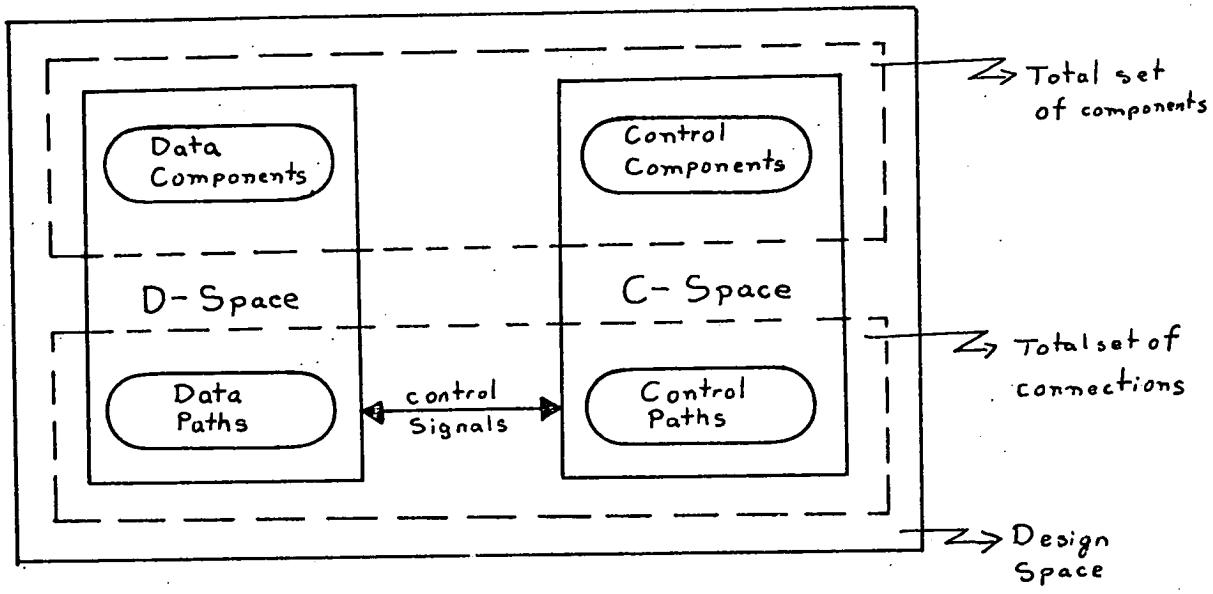


Figure 3.1 The "Design Space"

The problem of designing a digital system is then resolved into the two simpler problems of designing the D-space and C-space. The instruction repertoire and the type of components used will determine the necessary control signals. If this philosophy is accepted, the target machine should be structured so as to allow the specification and realisation of the two spaces separately.

3.1. Structure organisation

In the approach to be described, the process of determining the components and their organisation begins by the extraction of data operations from the design requirements. A set of data components is then chosen to implement them. The nature of both data and control components is described in the next section. Here it suffices to say that data components store (e.g. register), route (e.g. common highway) or modify (e.g. adder) the data values and the control components generate the required signals to activate them. This is

illustrated in Figure 3.2. AC1 and AC2 are two storage registers and ADD is a binary adder. These units share a common highway and their operation is controlled by two control components C1, C2.

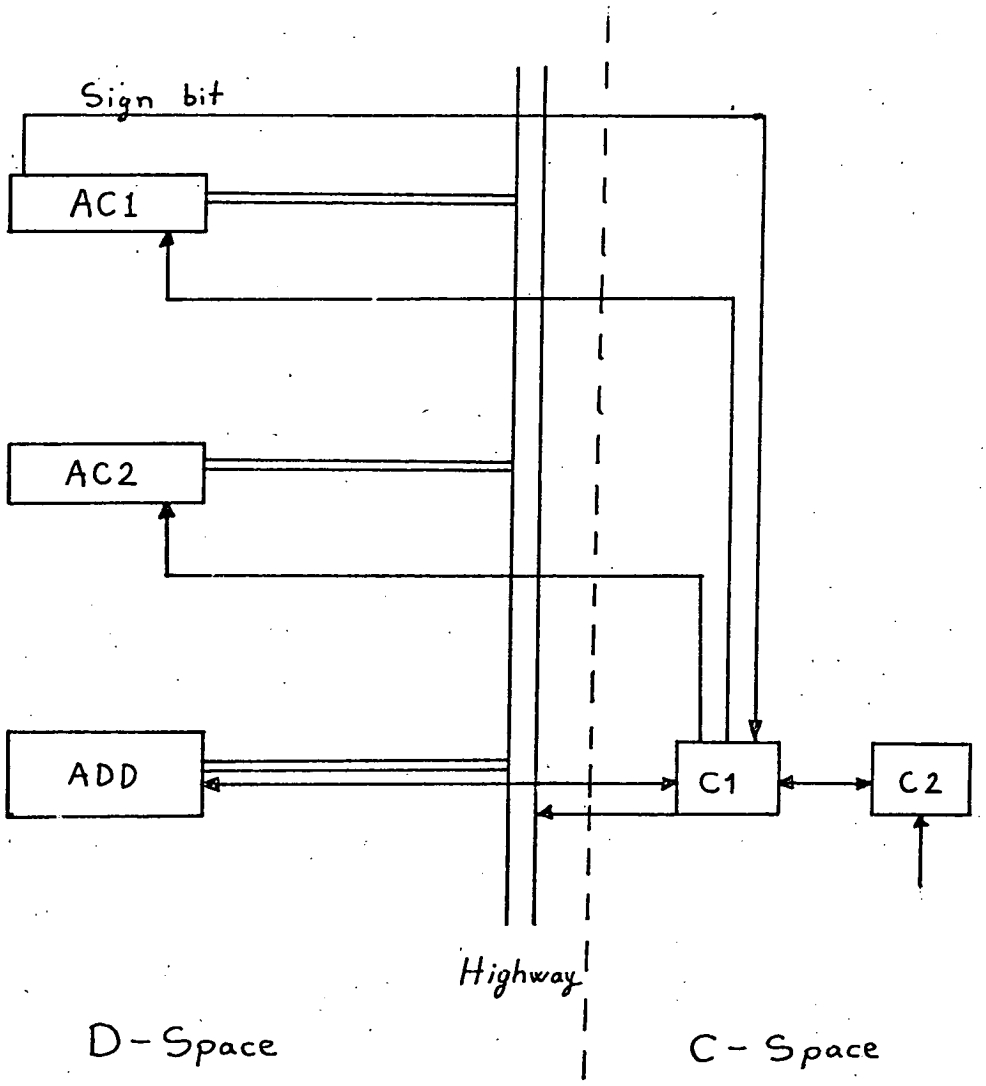


Figure 3.2 A Simple Digital System

The decomposition of the C-space is left to the designer. He may combine C1 and C2 to form a larger component or break them down into

smaller ones. It is better to combine only the related control operations and assign them to one component so that modifications will effect only a small part of the C-space. In the eventual machine the components are likely to be positioned close together and the separation described here is only conceptual.

3.2. Component types

An examination of the structure of some existing computers shows that many of them use very similar data components. The register is the most common of all. So common, in fact, that several design languages have been designed around a data space which consists only of a set of registers (or special registers) and the connections between them. As mentioned earlier, it is desirable to add to this set, components which can store, route or modify data values. This freedom is justified by the advances in integrated circuit technology since many such components are now available as single packages. They usually contain control inputs for external control of their operation, thereby making them consistent with the design philosopy described here. Even if a particular data component may not be directly available, it is a relatively easy task to design one, using the existing components.

The control components however, have to be designed according to the specific control operations required. Their task is to activate the data components and decide on the next operation to be performed. Since in most cases such decisions may depend upon the results of previous operations, control components are mainly sequential circuits. Using sequential circuits, control operations can be synchronised to each other and the different control states

can be represented by the internal-state of such components. On the other hand, some control functions can be implemented by using combinational circuits (e.g. index register selection). In either case the components in the C-space are themselves small logic circuits. Other types of control components can be waveform generators or counters, available as standard components. When implementing the control operations, it is convenient to hide the internal structure of such components from the designer, allowing him to work at a higher level and use only the functions that they realise. Therefore these components will be represented as "black-boxes" which realise switching functions and the designer will be responsible for describing their behaviour.

Memories, often of the read-only variety, can also be used as control components. They can be thought of as sequential components but it is more convenient to give them special treatment. In this case, the general description of the ROM device can be supplied by the design system but the specific memory contents should be specified by the designer.

Chapter 4 : Method of Design.

This chapter describes three computer programs that form an interactive CAD system for designing digital systems. The design process has been broken into three overlapping activities which are called "Component Specification", "Circuit Description" and "Design Evaluation". An overall view of the approach is given below followed by a description of the facilities provided.

4.1 General view of the method

The first step in the design process is to describe the terminal behaviour of some high-level components which will be used to realise a particular digital system. Some of these components are obtained from a component library which contains the definitions of standard data or control components (e.g. register, memory, decoder,...etc.). Other components which realise control operations specific to the particular application, are described by the designer. As mentioned in the previous chapter, in general, control components are themselves logic circuits which realise combinational or sequential switching functions. Hence, the "User Defined" components are divided into two groups and the first program is used to specify their terminal behaviour. Specification is made in terms of truth or state-tables as appropriate. These idealised elements can be realised in particular technologies by standard computer algorithms developed elsewhere (64).

The second program allows the designer to draw circuit diagrams on a display screen. Components are displayed as rectangles with an appropriate number of pins (terminals) and connections are made by drawing 'rubber-band' lines between their terminals. A set of

interconnected components can be grouped together and catalogued as "Macro" blocks. Terminal behaviour of the user defined components can be modified at any time without altering the drawings.

Logic designs prepared in this manner are then tested by using a functional simulator. The simulator program is interactive, allowing the user to stop simulation to examine and/or change logic values. Contents of memory or read-only memory components can also be changed so that the digital system being designed can be micro programmed.

The individual programs of the design system are described in the following sections.

4.2. Component specification

The first program is used to specify the terminal behaviour of user-defined components. The program has two modes of operation. In the 'conversational' mode it accepts instructions typed on the control teletype so that new components can be created or the existing ones can be modified. An example of conversation with the program is given below. Underlined words are program outputs.

READY:	Program is ready.
OPEN	New component definition.
NAME=:	Choose a name.
BLOCK	Component is to be called BLOCK*.
TYPE=:	Which type?
C	Combinational ("S" for sequential).
INPUTS:	Number of input terminals?
2	

(*) Certain names are reserved for library components.

OUTPUTS: Number of output terminals?

2

READY: Data accepted*

START Start display.

Following this conversation, the program enters into a 'graphical' mode and displays a 2 input, 2 output truth-table (initially all outputs are set to zero). This is illustrated in Figure 4.1.

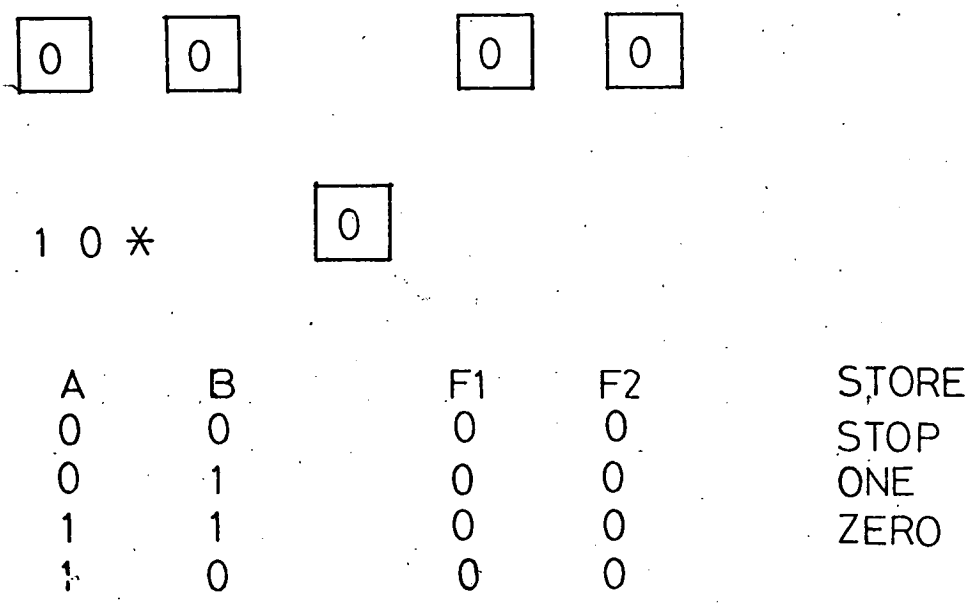


Figure 4.1 Truth-table Display for a Two-input, Two-output component.

The four boxes seen at the top of the figure represent the input and output terminals, in that order. Logic values are shown as 1, 0, and * (don't care condition). By pointing the light-pen to one of the logic values and then to a terminal-box, the contents of the box can be changed. This way, the user can set up input conditions and

---

(*) If the component is a sequential one, the program requests the number of internal-states before typing out this line.

indicates the outputs they affect by placing "1" into the corresponding output-box. Hence, more than one output terminal can be affected for the same input condition. The logic value that the selected outputs should produce is shown inside the centre box. Pointing the light-pen to "STORE" enters the values into the truth-table and mistakes can be detected by visual inspection. Figure 4.2 shows how the second output of the component can be set to realise the Boolean function $F2 = \overline{A}$ (i.e. complement of the first input).

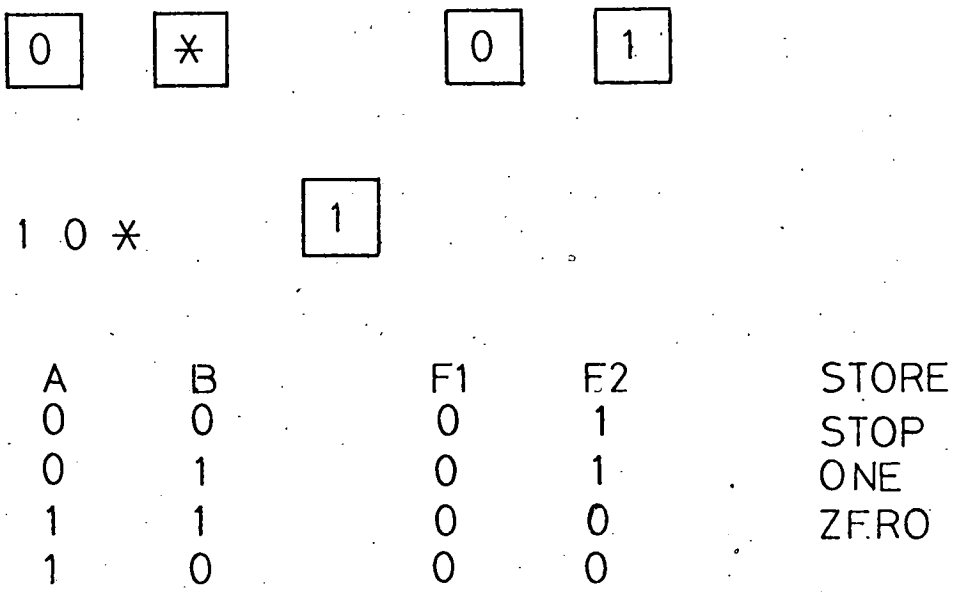


Figure 4.2 Specifying the Values

Outputs can be further modified by changing the input conditions and/or the output value. For example by changing the inputs to "*, 1" and pointing to "STORE" a second time, the function realised by F2 becomes $F2 = \overline{A} + B$. State-tables are specified in a similar way, using additional light-buttons to select the present and/or next-state values. Typing "STOP" on the control teletype terminates the program and the tables are stored in a file on disc. Components

having up to 5 inputs, 5 outputs and 16 internal-states can be defined. These numbers are adequate since only the non-standard control-components are defined in this way. Components which realise functions of a larger number of control variables should be broken into smaller ones so that specifications and modifications can be made easily.

4.3. Circuit description

Circuit descriptions are also made on-line by using a program which enables the designer to draw logic diagrams on the display screen. The user may request a component to be displayed by pointing to "BRING" and typing a name (and the number of bits if it is a library component) on the control teletype. This causes a rectangle to be displayed, which shows the name of the component and has an appropriate number of input, output terminals. Any component can be used more than once and each copy can be positioned, rotated or deleted individually. The mode of operation for a sequential component is defined at this step*. If synchronous operation is required, a clock terminal is also displayed on the component. Separate copies of a sequential component may operate in different modes. This is illustrated in Figure 4.3.

Figure 4.3 Component Display

(*) When a sequential component is requested by the user, the system asks for the mode of operation and the synchronising edge (1 to 0 or vice versa) of the clock pulse (if synchronous mode is desired).

Connections are made by drawing 'rubber-band' lines between the component terminals. This is not simply a matter of graphics since it involves access to the data structure to check, modify or enter some pointers. As it is not always practical to make direct connections (i.e. pin-to-pin), lines may be joined to arbitrary points on existing lines. Also a number of input terminals can be connected to each other and if one of them is later connected to an output terminal, the remaining inputs will also be connected to the same output. However, the connections between the individual input terminals are not lost so that, if the final connection is deleted, the input terminals remain tied to each other. The drawing program offers considerable freedom in making the connections. The only restriction is that each line should have at least one end point connected to a terminal. In addition, a connection will be rejected if it forms a loop or causes a short circuit between two outputs. Figure 4.4 shows some connections.

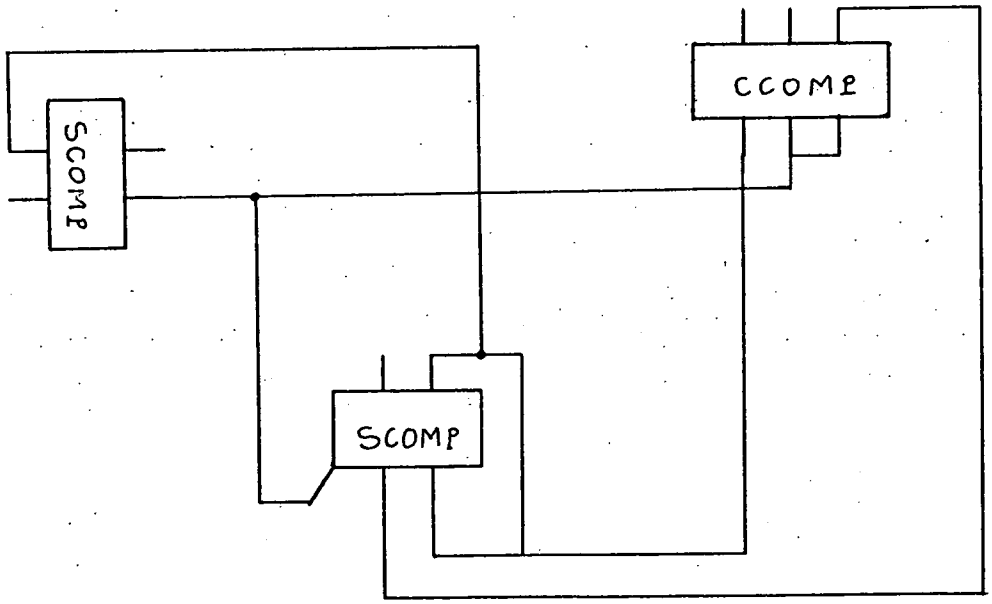


Figure 4.4 Some Connections

When drawing the data-paths, making a separate connection to each terminal of a data-component may become impractical. In such cases the "Multi-Line" facility can be used. A multi-line is a convenient form of representing a number of connections to the same component. It is drawn as a dotted line and a special sign is used to indicate the terminals it covers. The restrictions mentioned above still apply and a multi-line can be joined to another one only if they both represent the same number of bits. Figure 4.5 shows some multi-line connections between two 8-bit universal registers.

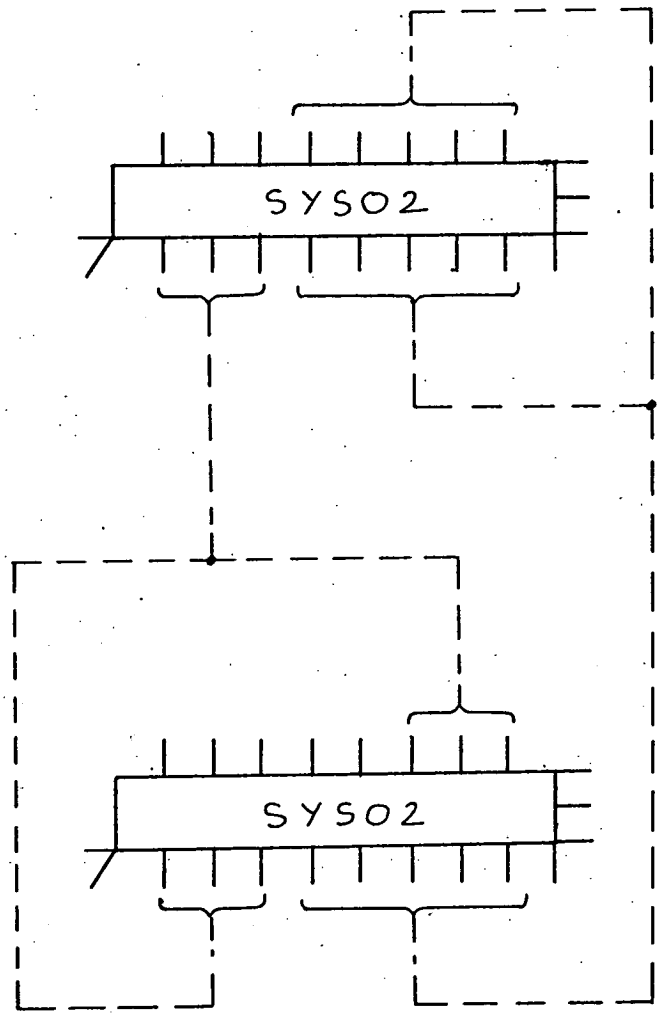
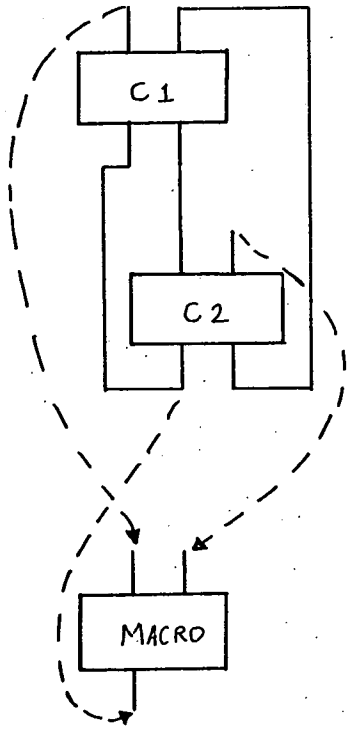


Figure 4.5 Multi-line Connections

Another feature of the drawing program is the "Macro" facility which enables the user to group any set of interconnected components, and use the group as a single 'Macro' component. The external input, output terminals are selected by the user and macro's can be formed from other macro components. Figure 4.6 illustrates a macro formation.



(a) Internal Structure

(b) Circuit Representation

Figure 4.6 Defining Macro Components.

When a macro is defined, its internal components and connections are erased from the screen but can be automatically regenerated and modified.

Logic diagrams can be partitioned using the "PAGE" facility. This enables the designer to work on a new page of drawing when the display screen is full. The program can erase the screen to provide a new page or regenerate the drawings on an existing one. Connections between components on different pages can be made by

displaying the same component on both pages. Note that this is not the same as using the same component twice since it is only a display procedure. Conflicting connections to the same component can be detected easily since the data structure is not partitioned (see Chapter 6). Hard copies of logic diagrams can be obtained on the plotter.

Component specifications can be altered without affecting the circuit diagrams and when both of them are ready, the design can be simulated.

4.4. Design evaluation

The third program is a functional simulator which accepts the data structure prepared by the first two programs. Logic diagrams are displayed on the screen, allowing the designer to 'probe' the components during simulation. All components are assumed to be non-ideal and have at least 1 unit delay. For a more realistic representation of the timing relations, 1 unit is taken as 10 nano seconds.* Simulation is done at 10 ns. intervals for a duration (of simulated time) specified by the user. However, it can be stopped at any time to examine and/or modify the logic or delay values. The present-state of a sequential component or the contents of data-components can also be changed. Hence, if read-only memories are used in the design, different micro programs can be tried.

(*) Although this is not a user controlled feature, the scale can be changed easily.

The simulator generates logic waveforms of selected terminals and outputs them to disc. Hard copies can be obtained by listing the output files on the line printer. The waveforms normally have a linear time axis where values are produced every 10 nano seconds. However, they can also be generated with a non-linear time axis where values are produced only when one of the monitored terminals changes state (1 to 0 or vice versa). This cuts down the amount of output listings produced by eliminating long periods when the signals are all stable. In addition, the simulator can produce warning messages when more than one input or an input and the clock pulse of the same component change state simultaneously. Spikes (pulses with 10 ns. duration) can also be detected.

Another very useful feature of the simulator is the "TRAP" facility. This enables the user to define conditions (e.g. the contents of a register being equal to a certain value) which the program is to detect and stop simulation automatically. After examining or modifying the values, simulation can be resumed. Hence the simulator program can be used as a debugging aid (similar to software debugging programs e.g. DDT).

4.5. Discussion of the method.

The three programs described above form high-level tools for specifying and simulating logic designs. The use of graphical techniques makes it possible to describe the structural aspects of digital systems in a very natural way (i.e. using circuit diagrams). The general characteristics of the design system are as follows:

(i)- Digital systems are described at a high-level and the method of representation is natural to logic designers.

(ii)- Specifications and modifications can be carried out easily. Designs can be partitioned as required and modifications can be made without reiterating the total description.

(iii)- Pre-defined logic blocks can be used in the design.

(iv)- Only the functional characteristics of components are described and implementation details are not considered. However, the user-defined components can be implemented by using standard computer algorithms.

(v)- Components can operate in the synchronous or asynchronous mode.

(vi)- All programs operate on the same data structure so that no additional processing is required to pass data among them.

(vii)- All descriptions (including diagrams) are stored inside the computer. Hard copies can be obtained and all operations are under the user control.

Chapter 5 illustrates the use of the programs on a design example.

Chapter 5: Design Example

The following example illustrates the application of the general philosopy presented in Chapter 3, to the design of a digital system. The Ferranti DISC computer (3) has been selected as the target machine and its design (not complete) was carried out using the three programs described in the previous chapter. The object of the exercise is not to show how this particular computer should be designed nor is it the author's aim to prove his competence or otherwise in designing a digital system. The main objective is to test the viability of the approach and see if a CAD system can be built to implement the proposals. For that reason very little attention was paid to the selection of the particular components in the system library. Some of these components may be found too abstract with no realistic counter part. Others may be too impractical to use in real applications. They were modelled simply to enable the author (or any other user) to design a digital computer. However, it is hoped to show that if the library components are chosen properly, the method of design described in Chapters 3 and 4 can be used in solving practical problems.

A partial specification of DISC is given below (the terminology used is from the DISC literature).

5.1. Machine description.

DISC is a general purpose digital computer to be used in applications where very high speed of execution is not required. It has an 8-bit/word memory for program storage. Each instruction word comprises a 3-bit operation code field and a 5-bit address field. Instruction repertoire consists of 7 'basic instructions' and 17 'additional functions'. More orders are obtained by altering the state of the 'Logical Stat' or 'Double-Length Working Stat' which modify the meanings of some of the orders. Data is stored in 16-bit words in a separate, volatile memory which is divided into 3 sectors of 30 words each. Arithmetic is done in serial binary form using two's complement number representation. The main components of the central processor unit are as follows.

(i)- Accumulator register (AC).

A 16-bit register used during most arithmetic or logic operations on data.

(ii)- P-register (P).

The P-register is another 16-bit register which is used to extend the AC for double-length working. It is also used during multiply and divide routines.

(iii)- Q-register (Q).

A 16-bit register used in multiply and divide routines.

(iv)- Shift counter (SC).

SC is a 5-bit counter used for controlling the shift, multiply or divide operations.

(v)- Program address register (PAR).

The PAR is a 16-bit register which stores a 12-bit program address, 1-bit I/O sector, 2 bits to indicate the volatile memory

sector (1, 2 or 3), and a 1-bit 'master jump stat' to enable jump instructions.

(vi)- Instruction register (IR).

A 16-bit register which accepts 8-bit instruction words from the program memory and stores them during execution. If required by the instruction, two consecutive words can be loaded into IR, prior to transferring its contents to the AC or PAR.

(vii)- Arithmetic and logical unit (ALU).

The ALU handles all data operations and controls inter register transfers.

In the example below, some changes were made to these specifications and only a small sub-set of the instructions were designed. The most important changes are,

(a)- the new machine operates in parallel, asynchronous mode,

(b)- data word length (and the associated registers) is chosen as 8-bits and both memory units have similar characteristics (Memory access times are set to 750 and 30 nano seconds for the program and data store, respectively),

(c)- program memory has only 256 words and correspondingly PAR has 12 bits.

(d)- IR has 8 bits.

5.2. System design.

The following sections describe the design of modified DISC. The circuit diagrams given below were first drawn on the display screen using the light pen and were later reproduced on the plotter. The terminal behaviour of library components used in the design, can be found in Appendix B. Outputs given in the state-tables are in octal.

5.2.1. Control of memory access

Memory operations are realised by using two library components which complement each other. The first one is the memory component. Figure 5.1 shows the circuit representation of an 8-bit/word X 256 word memory. It contains a memory buffer register (inputs I_1 to I_8, outputs O_1 to O_8), an address register (X_1 to X_4 and Y_1 to Y_4), four control inputs, and storage. Access to a word is achieved by setting the read (R) or write (W) input to logic one and applying a positive pulse (0-1-0) to the RX or WX input, respectively. Access time is specified by the user.

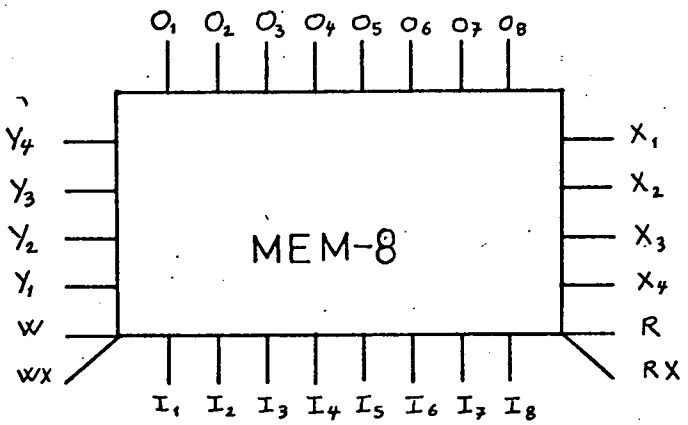


Figure 5.1 The Memory Component.

The second component is used to generate the necessary signals to initiate a memory access. When a positive pulse is applied to the P input, X becomes logic one, a positive pulse appears on PX, and after a time delay, another positive pulse appears on SX. The X and PX outputs are connected to the R (or W) and RX (or WX) inputs of the memory component. SX is used to indicate the completion of memory access.

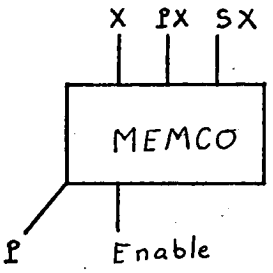


Figure 5.2 The Memory Controller Component.

5.2.2. Designing the fetch/execute cycle.

DISC does not use instruction pipe-lining so that the two control states of 'instruction fetch' and 'execution' become mutually exclusive. Since an asynchronous design is attempted, the end of each instruction execution is indicated by setting a flip-flop which will be cleared when the computer goes into the fetch state. Even though these two control states can be represented by a single logical variable, it was decided to use separate variables for each one*. Figure 5.3 gives the circuit diagram of the control circuit to realise the fetch/execute cycle. The diagram is explained below.

(*) This has no particular reason but it was felt that individual fetch and execute signals would be more useful.

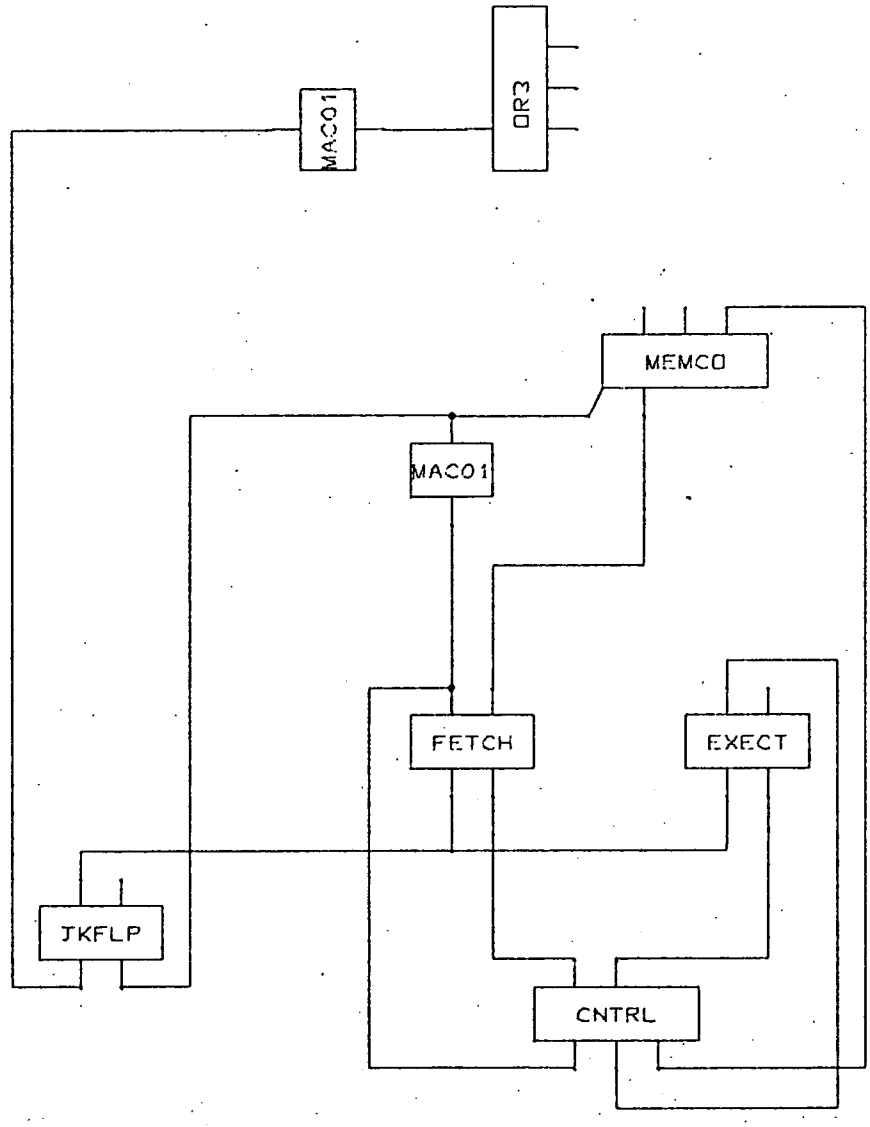


Figure 5.3 Control of Fetch/execute States.

The 3-input OR gate carries the 'execution complete' signal which sets a flip-flop (JKFLP). FETCH and EXECT take the OR of the two inputs and generate both true and complemented outputs. MACO1 is a macro component which transforms a zero-to-one change at the input into a positive pulse (i.e. a 'one-shot'). When the flip-flop is set, both FETCH and EXECT become one. This enables MEMCO which controls the read access to program memory. The output from MACO1 which drives MEMCO, is also used to reset JKFLP. CNTRL is a sequential component which sets FETCH = 1, EXECT = 0 after JKFLP has been reset. When the memory access is complete, CNTRL complements the values of FETCH and EXECUTE and waits until another 'execution complete' signal is received. Table 5.1 gives the state-table of CNTRL.

Present	Next-state & present outputs.							
State	000	001	011	010	110	111	101	100
1	1,2	1,2	1,2	1,2	1,2	1,2	1,2	2,2
2	2,2	2,2	2,2	2,2	2,2	2,2	3,1	2,2
3	3,1	3,1	3,1	4,1	3,1	3,1	3,1	4,1
4	4,1	4,1	4,1	4,1	1,2	4,1	4,1	4,1

Table 5.1 State-table for CNTRL.

In state 1, the component waits for the outputs to settle down and then goes into the 'fetch' state (state 2). The 'access completed' signal from MEMCO causes a transfer into state 3 and then into state 4, which is the 'execute' state. CNTRL waits in this state until EXECT becomes one (causing a momentary FETCH = EXECT = 1) and then transfers back into state 1.

MAC01 could also be designed as a sequential circuit*. However, a simpler solution is obtained by making use of the propagation delay through an inverter. This is shown in Figure 5.4.

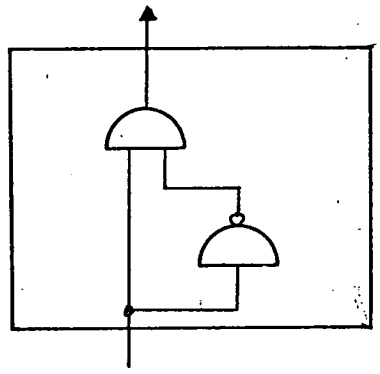


Figure 5.4 The MAC01 Component.

Duration of the pulse at the output of MAC01 is equal to the propagation delay of the inverter.

5.2.3. Instruction fetching and decoding.

Figure 5.5 shows the connections to the program memory which has been called MAIN8.

(*) The obvious solution is to make it a library component.

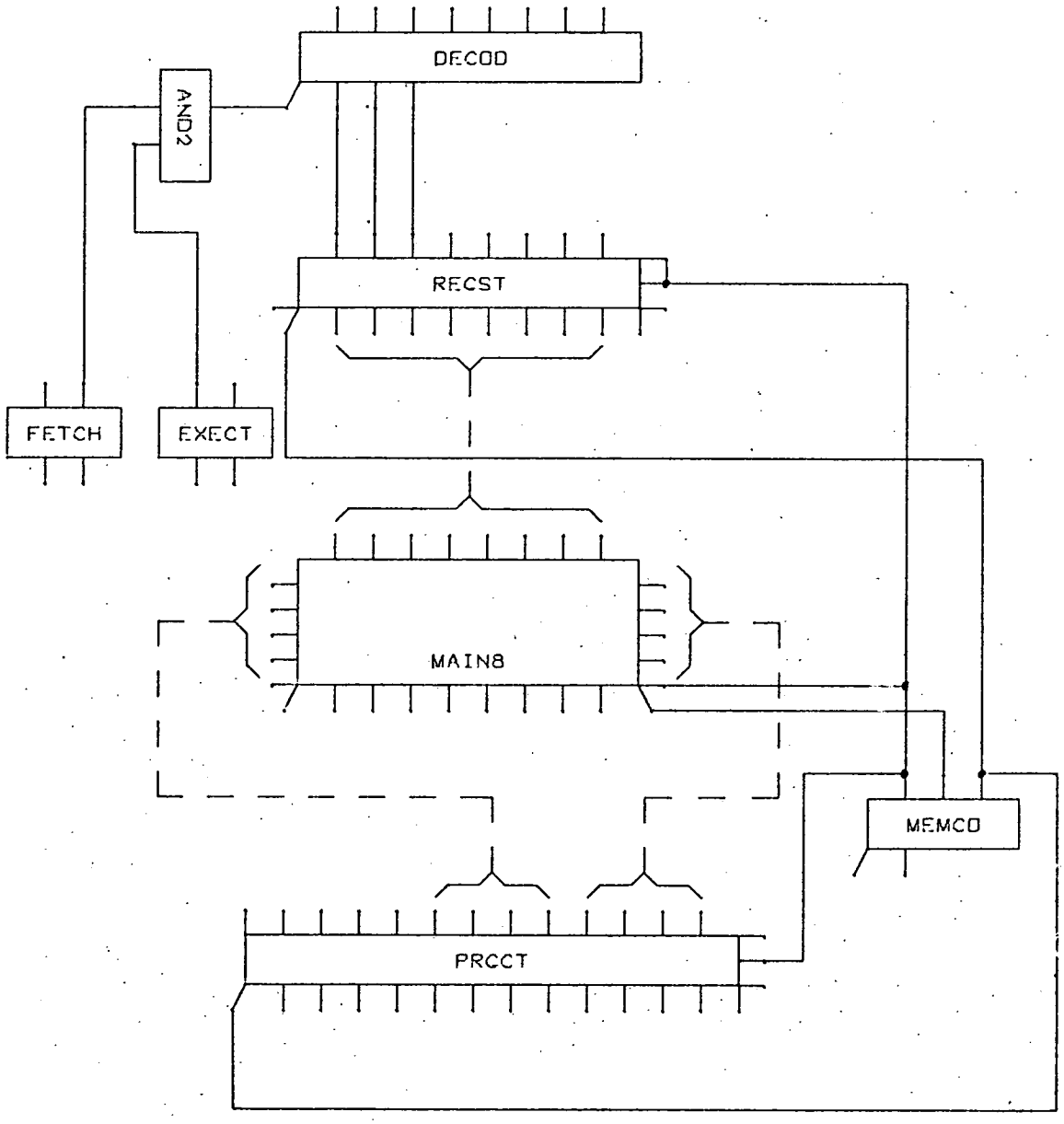


Figure 5.5 Connections to Program Memory.

The program address is obtained from the lower order bits of the 12-bit (plus overflow) program counter, PRGCT. Outputs of the memory buffer are connected to the parallel data inputs of an 8-bit universal register, REGST, which represents the instruction register. The 3-to-8 decoder, DECOD, is another library component which is enabled when FETCH is zero and EXECT is one. MEMCO initiates the memory access. The 'access complete' pulse is used to clock the two registers so that REGST accepts the parallel data and PRGCT is incremented by one. Other connections on the registers set the mode of operation (see Appendix B).

5.2.4. Arranging the data paths.

Figure 5.6 shows the main data paths. HGHWY represents an 8-bit common highway which is shared among four components. It has four 'highway request' inputs (inputs 1, 3, 5 and 7) and four data ports (each 8-bit wide). When one of the request inputs is set to logic one, the 8-bit data on the adjacent data port is loaded to the highway and the 'highway busy' output (9th output) is set. The unit has priority logic such that if simultaneous requests are made, the one on the leftmost input wins the highway. Further attempts to get the highway are ignored until the last request is cancelled. The component also has storage so that the latest data values are retained even after the current request is cancelled.

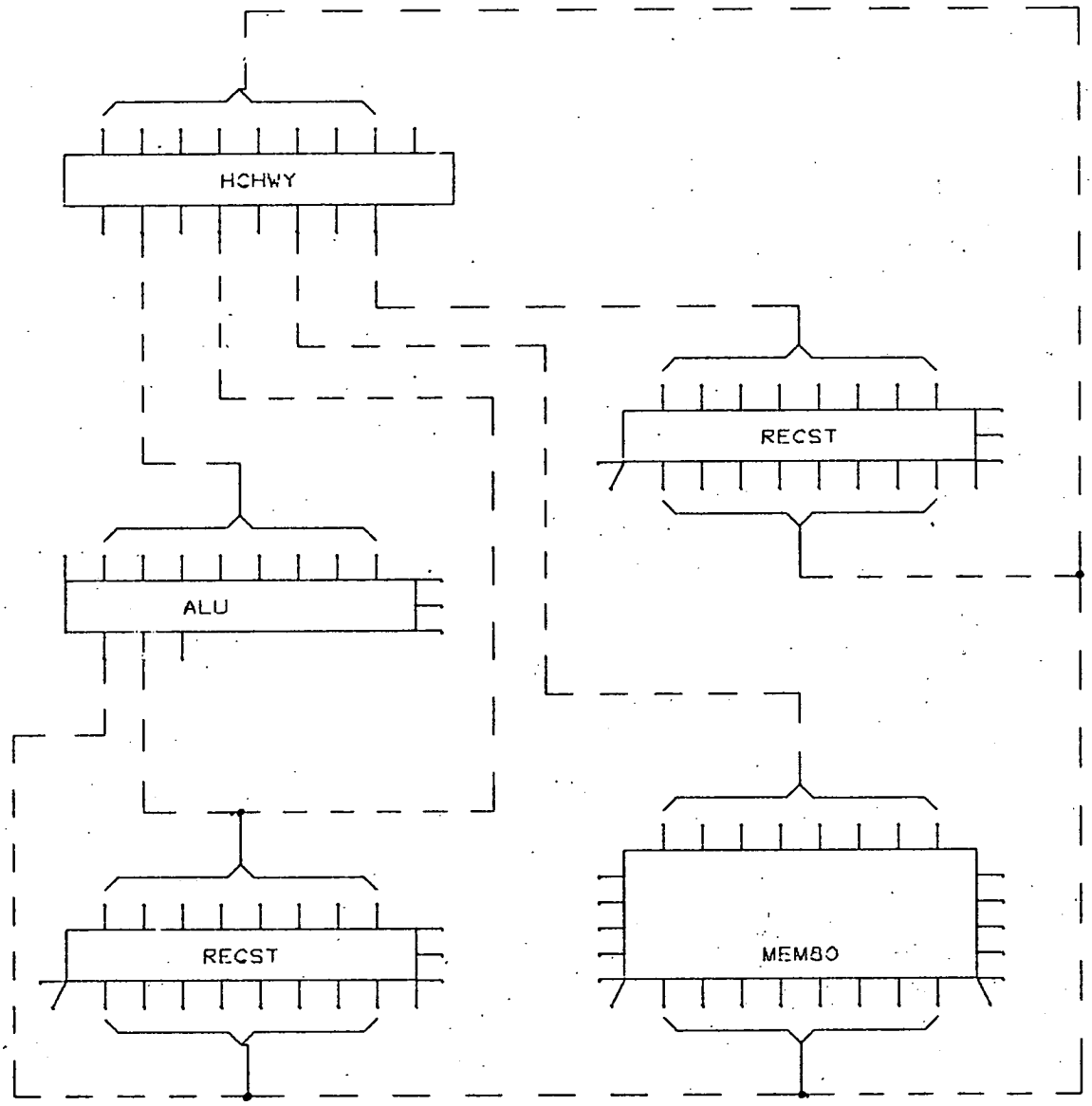


Figure 5.6 Highway Organisation.

The four components which share this unit are described below, in the order of priority they receive from the highway.

(i)- ALU is the arithmetic and logical unit which performs data operations (defined by the mode inputs) between the contents of the highway and the accumulator register.

(ii)- REGST is an 8-bit universal register which is used as the accumulator.

(iii)- MEM80 is the data memory which is similar to MAIN8 (Figure 5.5), but much faster.

(iv)- REGST is another register which is used as the Q-register. Each of these components have an associated component to load their outputs to the highway. Figure 5.7 shows their connection to the highway.

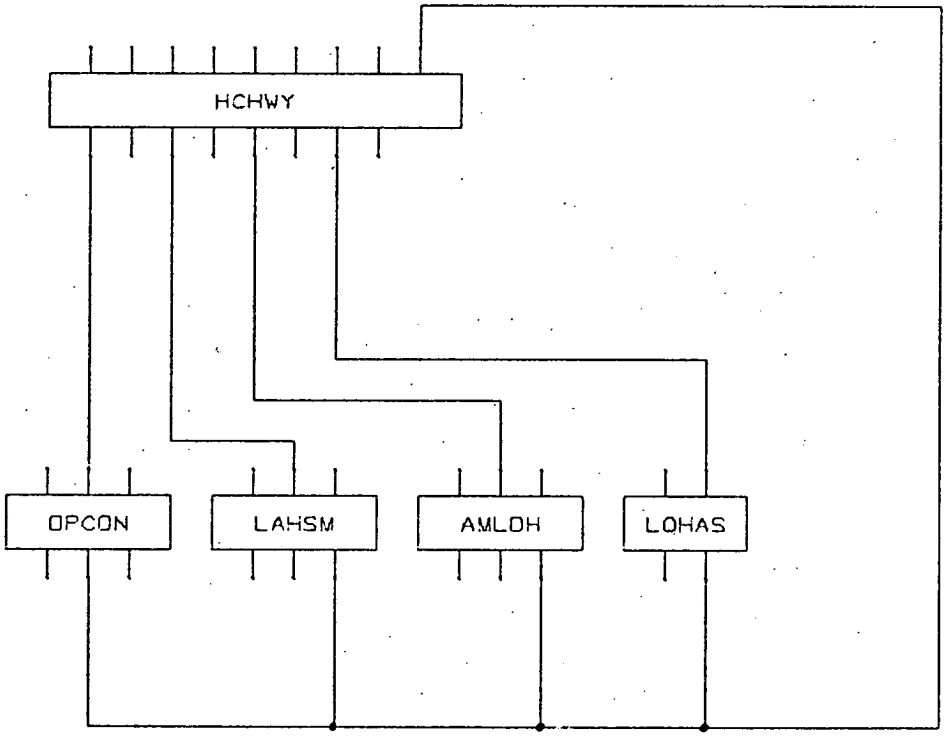


Figure 5.7 Highway Request Components.

For example, the state-table for LQHAS (Load Q-register to Highway And Signal) is given below.

Present	Next-state & present outputs			
State	00	01	11	10
1	1,0	1,0	1,0	2,1
2	1,0	1,0	3,2	2,1
3	1,0	1,0	3,2	3,2

Table 5.2 State-table for LQHAS.

Tables for the other components are given later.

5.2.5. Access to data memory.

Two memory controller components are used to control the read and write operations in the data memory. Memory address is obtained from the 5 least significant bits of the instruction register and the two sector bits stored in the program counter. These connections are shown in Figure 5.8.

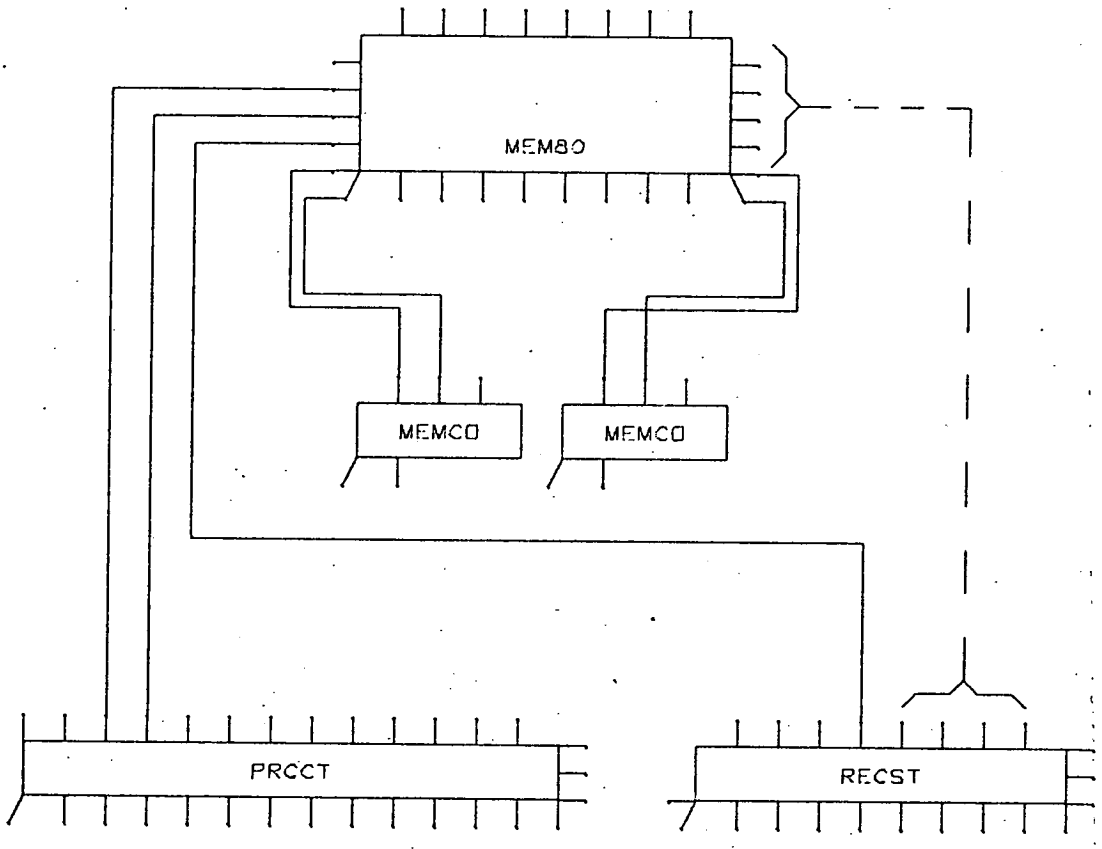


Figure 5.8 Connections to the Data Memory.

In addition, two components are used to load the AC or the memory buffer to highway and to initiate the memory access. Table 5.3 gives the state-table of a sequential component, LAHSM (Load Accumulator to Highway and Store in Memory), which first loads AC to the highway and then activates the memory controller component handling the write operations. A level output from LAHSM indicates that data has been stored in memory. The inputs to the component indicate 'begin', 'memory access done', 'highway received' and its outputs mean 'start memory access', 'get highway', 'operation complete'. Its second input and first output are connected to the component (MEMCO) which handles write operations in the data memory.

Present	Next-state & present outputs							
State	000	001	011	010	110	111	101	100
1	1,0	1,0	1,0	1,0	1,0	1,0	1,0	2,2
2	2,2	2,2	2,2	2,2	2,2	2,2	3,6	2,2
3	3,6	3,6	3,6	3,6	4,0	4,0	3,6	3,6
4	4,0	4,0	4,0	4,0	4,0	4,0	5,1	5,1
5	1,0	1,0	1,0	1,0	5,1	5,1	5,1	5,1

Table 5.3 State-table for LAHSM.

The state-table of a similar component, AMLDH, which is used to 'Access Memory and Load Data to Highway', is given below. Its second input and first output are connected to the component (MEMCO) which handles read operations from the data memory.

Present	Next-state & present outputs							
State	000	001	011	010	110	111	101	100
1	1,0	1,0	1,0	1,0	1,0	1,0	1,0	2,4
2	2,4	2,4	2,4	2,4	3,0	3,0	2,4	2,4
3	3,0	3,0	3,0	3,0	3,0	3,0	3,0	4,2
4	4,2	4,2	4,2	4,2	4,2	4,2	5,1	4,2
5	1,0	1,0	1,0	1,0	5,1	5,1	5,1	5,1

Table 5.4 State-table for AMLDH.

5.2.6. Designing load and store instructions.

The 'load AC from memory' and the 'store AC in memory' instructions (operation codes 001 and 010, respectively) are implemented using the last two components describe above. Because of the similarity between the two operations, a single component was designed to control both of them. The component is called S1S2E and receives two inputs from the decoder (one for each instruction). A third input is obtained by passing the 'operation complete' outputs (output 3) of AMLDH and LAHSM through an OR gate. It generates two outputs to activate AMLDH and LAHSM, respectively, and the third output is connected to the OR gate which sets the JKFLP (see Figures 5.3 and 5.9). The state-table for S1S2E is given below.

Present	Next-state & present outputs							
State	000	001	011	010	110	111	101	100
1	1,0	1,0	1,0	3,2	1,0	1,0	1,0	2,4
2	1,0	1,0	1,0	1,0	2,4	2,4	4,1	2,4
3	1,0	1,0	4,1	3,2	3,2	3,2	3,2	3,2
4	1,0	1,0	4,1	4,1	4,1	4,1	4,1	4,1

Table 5.5 State-table for S1S2E.

The load instruction also requires a clock pulse to be sent to the AC so that the contents of the highway can be transferred into the accumulator register. This pulse is obtained from the first output of S1S2E. Also, the mode inputs of AC should be set to 'accept parallel data'. The circuits for these operations are given later. Figure 5.9 shows the complete control circuitry to execute the first four instructions (load, store, add, subtract). Add and subtract instructions are described in the next section.

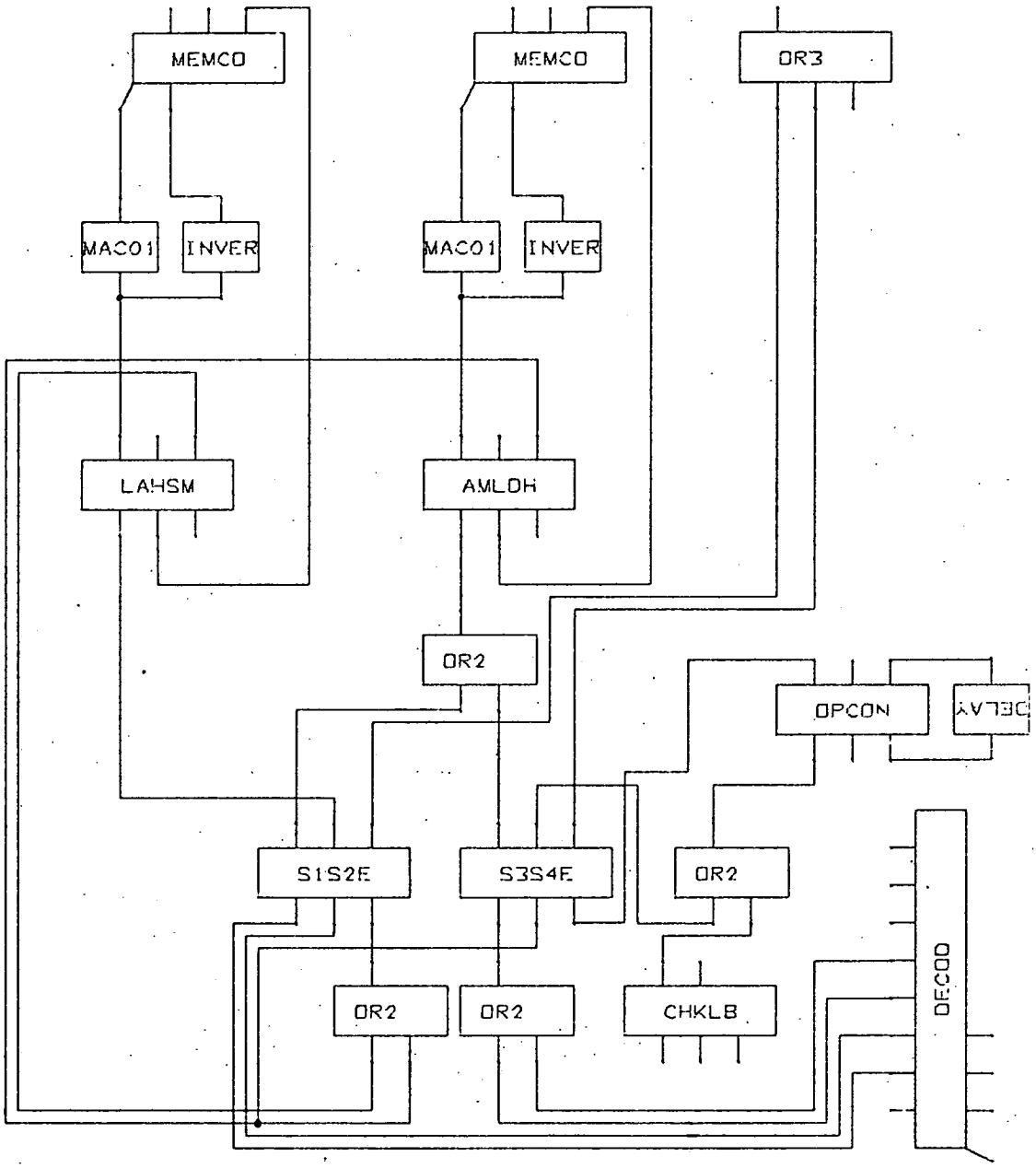


Figure 5.9 Control Circuit for the First Four Instructions.

5.2.7. Add and subtract instructions

The add and subtract instructions (operation codes 011 and 100, respectively) perform arithmetic operations between the contents of the AC register (the lower REGST in Figure 5.6) and a memory location, placing the result into the AC. As seen from Figure 5.6, ALU can perform data operations on the contents of AC and the highway. Therefore, first a memory access is needed to get the data word and place it on the highway. Table 5.6 gives the state-table of a component, S3S4E, which activates AMLDH to perform this operation. When this is done, S3S4E activates another component, OPCON, which waits for the ALU to perform the arithmetic, places the result on the highway and informs S3S4E that the operation is complete. The same signal is sent as clock pulse to AC so that it accepts the result from the highway. S3S4E then sends a signal through the OR gate, setting JKFLP.

Present	Next-state & present outputs.							
State	000	001	011	010	110	111	101	100
1	1,0	1,0	1,0	1,0	1,0	1,0	1,0	2,4
2	1,0	1,0	1,0	1,0	3,2	2,4	2,4	2,4
3	1,0	1,0	1,0	1,0	3,2	4,1	4,1	3,2
4	1,0	1,0	1,0	1,0	4,1	4,1	4,1	4,1

Table 5.6 State-table for S3S4E.

The particular arithmetic operation (addition or subtraction) to be performed by the ALU is selected by setting the mode inputs of ALU as required. The circuit to achieve this will be described later. OPCON simulates the task duration of ALU by generating an output, which is delayed and fed back. The delay value should be set to half the

time needed by ALU to perform the operation. The result is put on the highway by placing a request through OPCON (see Figure 5.7). The state-table of OPCON is given below.

Present	Next-state & present outputs							
State	000	001	011	010	110	111	101	100
1	1,0	1,0	1,0	1,0	2,1	1,0	1,0	2,1
2	1,0	1,0	1,0	1,0	2,1	3,0	3,0	2,1
3	1,0	1,0	1,0	1,0	3,0	3,0	3,0	4,2
4	1,0	1,0	1,0	1,0	5,4	5,4	4,2	4,2
5	1,0	1,0	1,0	1,0	5,4	5,4	5,4	5,4

Table 5.7 State-table for OPCON.

Component CHKLB seen in Figure 5,9, is used during multiply operations and is described in the following section.

5.2.8. Shift and multiply instructions

The shift and multiply operations are treated together since multiplication is done by successive addition (or subtraction) and shifting. Operation code 110 indicates the instruction to shift the contents of the AC to the right by the number of places indicated in the address field of the instruction. However, this number is ignored if the shift counter is pre-loaded beforehand. Therefore, first the contents of the counter is checked and the number in the address field is transferred there if the counter is empty. During a multiply instruction, the number to be loaded is octal 10 (8 shifts). Figure 5.10 shows how the data is selected. G1 and G2 are combinational components defined by Table 5.8 and AND5M is an AND gate which produces a one output during multiply operations.

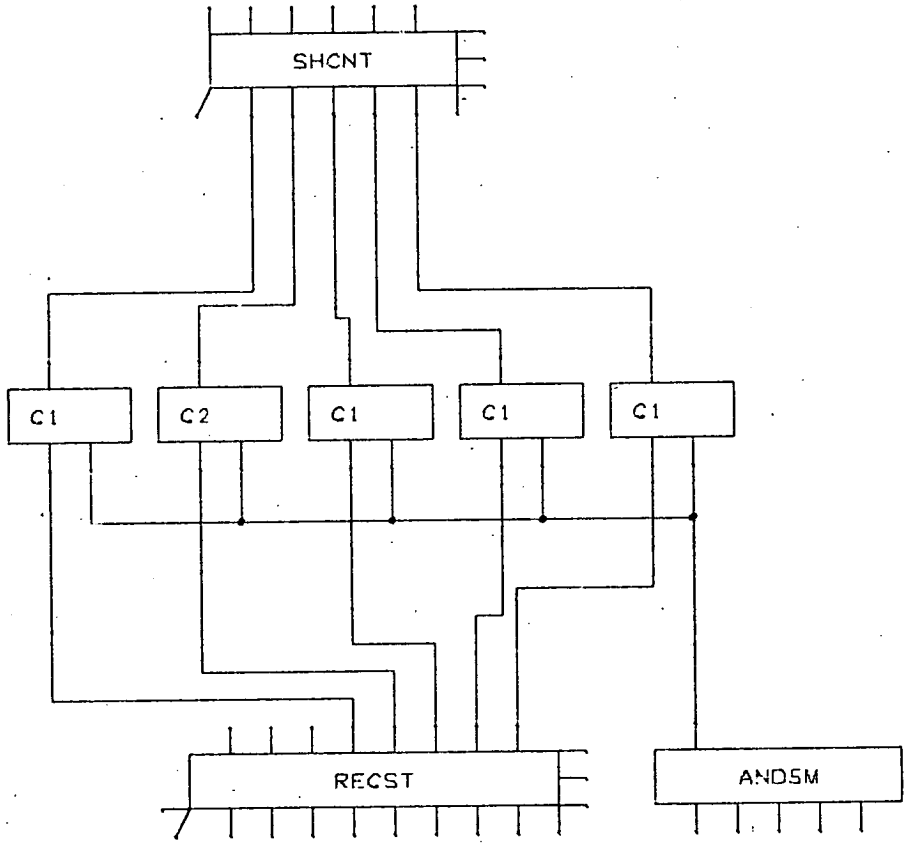


Figure 5.10 Selecting Data for the Shift Counter.

I1	I2	G1
0	0	0
0	1	1
1	1	0
1	0	0

(a) G1

I1	I2	G2
0	0	0
0	1	1
1	1	1
1	0	1

(b) G2

Table 5.8 Truth-tables for G1 and G2.

Data is accepted by the counter when the count/load mode input becomes zero. The complete circuit for shifting and multiplication is shown in Figures 5.11 and 5.12.

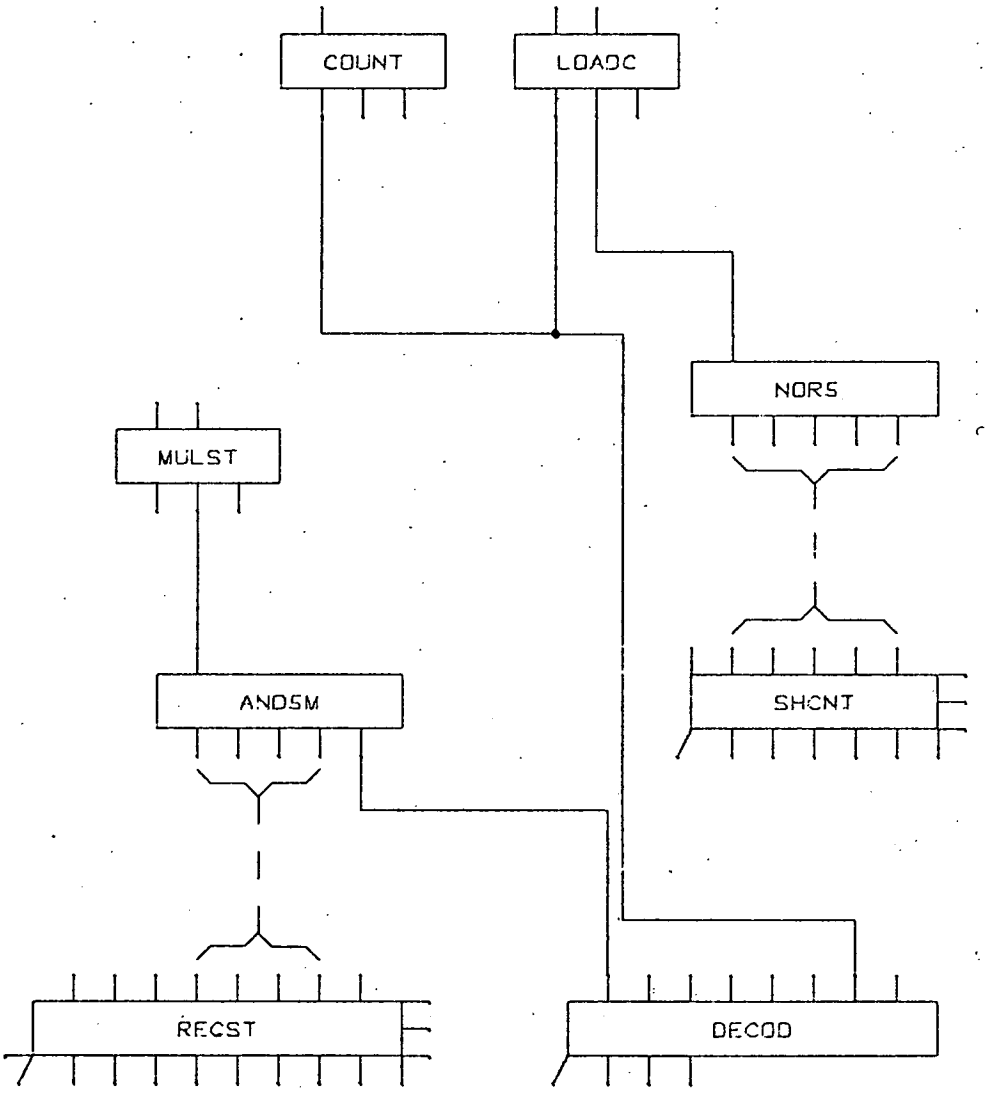


Figure 5.11 Enabling Shift or Multiply Operations.

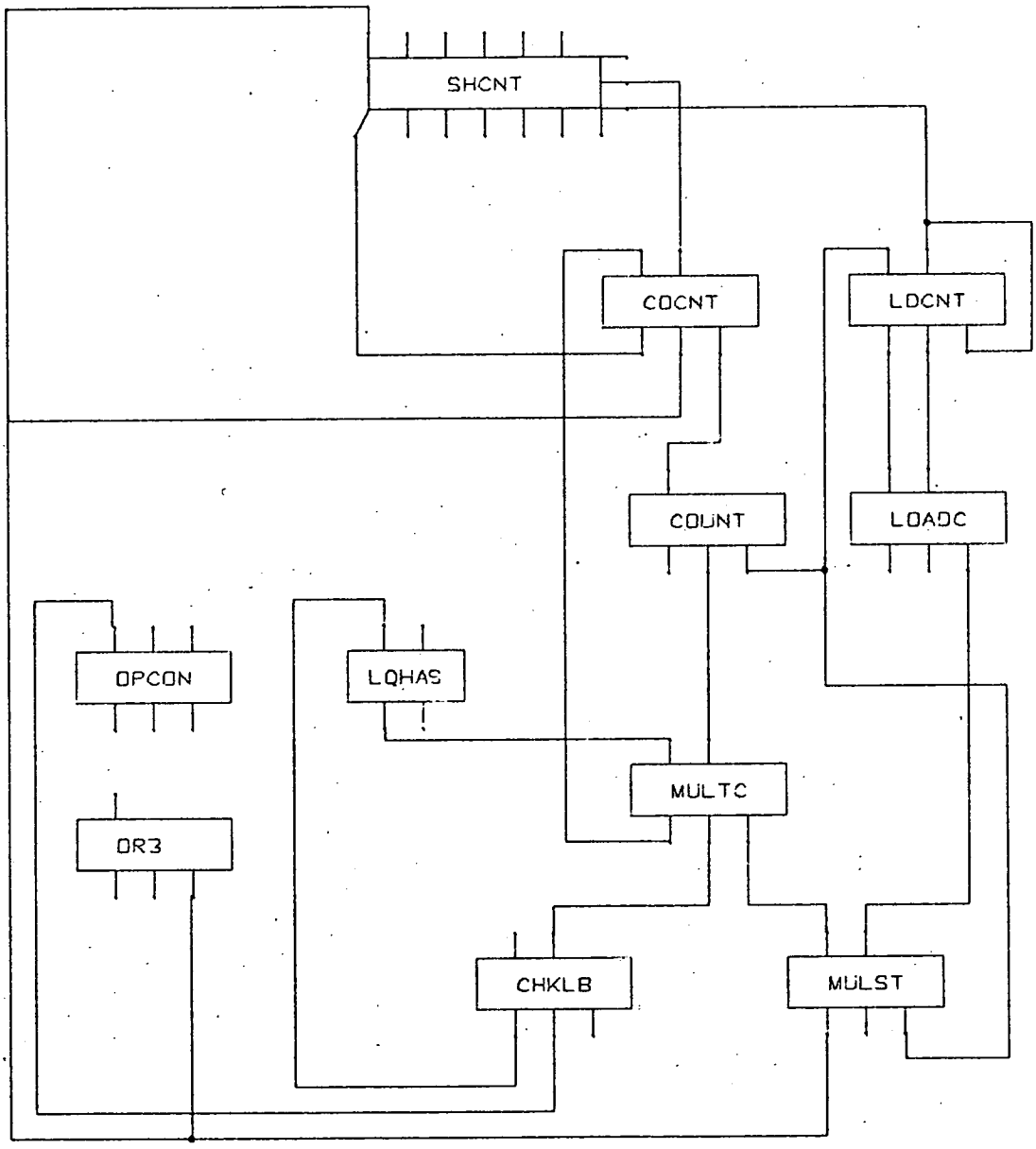


Figure 5.12 Control of Shifting and Multiplication.

LOADC ('Load Counter') is a combinational component with 3 inputs and 2 outputs. Its first input is taken from the decoder output which indicates shift operation. The second input is obtained by passing the counter output through a NOR gate to test its value. The third input comes from another component which controls multiply operations. The first output indicates that shift (or multiply) operation is enabled and the second output indicates whether the counter should be loaded with a number. Table 5.9 gives the truth-table for LOADC.

I1	I2	I3	O1	O2
0	0	0	0	0
0	0	1	1	1
0	1	1	1	1
0	1	0	0	0
1	1	0	1	1
1	1	1	1	1
1	0	1	1	0
1	0	0	1	0

Table 5.9 Truth-table for LOADC.

The outputs of LOADC are connected to the inputs of LDCNT ('Load Controller') which is a sequential component. It generates two outputs. The first one shows if the counter is ready for operation and the second output is connected to the load/count mode input of the counter. This output is normally kept at logic one (i.e. count mode). The second output of LDCNT is also fed back to the third input of the component. This enables LDCNT to check if loading has been done. Table 5.10 gives the state-table of LDCNT.

Present	Next-state & present outputs							
State	000	001	011	010	110	111	101	100
1	1,1	1,1	1,1	1,1	1,1	2,0	3,3	1,1
2	1,1	1,1	1,1	1,1	3,0	2,0	2,0	3,0
3	1,1	1,1	1,1	1,1	3,3	3,3	3,3	3,3

Table 5.10 State-table for LDCNT.

These components prepare the shift counter for operation. The counter receives a continuous clock pulse (not shown in the diagram) but its 'count enable' input is controlled by two other components. COUNT is a 3 input, 1 output combinational component. It receives its inputs from the decoder (shift instruction), MULTC (a component used during multiplication) and LDCNT (counter ready signal). Its truth-table is given below.

I1	I2	I3	O1
0	0	0	0
0	0	1	0
0	1	1	1
0	1	0	1
1	1	0	1
1	1	1	1
1	0	1	1
1	0	0	0

Table 5.11 Truth-table for COUNT.

This component activates COCNT ('Control Counting'), which is a sequential component. COCNT receives the clock pulse applied to the counter, as a pulse input and passes it on to the AC (and P)

register. Counting is done towards zero and when that value is reached the overflow pulse generated by the shift counter stops COCNT. The overflow pulse is also connected to the OR gate which sets JKFLP. Table 5.12 gives the state-table of COCNT.

Present	Next-state & present outputs							
State	000	001	011	010	110	111	101	100
1	1,0	2,1	3,1	1,0	1,0	1,0	1,0	1,0
2	1,0	2,1	3,1	1,0	1,0	5,0	3,3	1,0
3	1,0	2,1	3,1	1,0	1,0	5,0	3,3	1,0
4	1,0	5,0	5,0	1,0	1,0	5,0	4,1	1,0
5	1,0	5,0	5,0	1,0	1,0	5,0	5,0	1,0

Table 5.12 State-table for COCNT.

The multiply instruction is an additional function (operation code 000) which is written as '00011110' (octal 36). It places the double-length product of two signed integers (initially in P and Q) into the AC and P registers (least significant bits in P). Booth's algorithm (66) is used to perform this operation and in accordance with the requirements of this algorithm, the P-register is 9 bits long. During multiplication, AC and P are shifted together, with the last bit of AC feeding the first one of P. The last two bits of the P-register are checked before each shift operation and an addition or subtraction is performed between the AC and the Q-register (result in AC) depending upon their values. Table 5.13 shows the action taken for each possible combination. Bit 9 should be zero before the multiplication routine is started.

Bit 8	Bit 9	Action
0	0	Do nothing.
0	1	Add Q to AC.
1	1	Do nothing.
1	0	Subtract Q from AC.

Table 5.13 Booth's Algorithm for Multiplication.

The control components for the multiply routine are shown in Figure 5.12. The operations are started when the second input of the 'Multiplication Start' component, MULST, becomes one (see Figure 5.11). MULST first activates LOADC (Table 5.9) and then MULTC ('Multiply Control'). The overflow pulse stops multiplication. Table 5.14 gives the state-table of MULST.

Present	Next-state & present outputs.							
State	000	001	011	010	110	111	101	100
1	1,0	1,0	1,0	2,1	2,1	1,0	1,0	1,0
2	1,0	1,0	3,3	2,1	2,1	2,1	1,0	1,0
3	1,0	1,0	3,3	3,3	4,0	4,0	1,0	1,0
4	1,0	1,0	4,0	4,0	4,0	4,0	1,0	1,0

Table 5.14 State-table for MULST.

MULTC does the actual control of the multiplication routine. It loads Q to the highway by activating LQHAS (Table 5.2) which in turn, informs CHKLB ('Check Last Bits') that data is ready. Bits 8 and 9 of the P-register are passed through an EXCLUSIVE-OR gate whose output is connected to CHKLB which decides if addition or subtraction is needed. If so, then OPCON is activated (see Figure 5.9). When the arithmetic is done (or if none is required) CHKLB sends a signal to

MULTC which enables counting for one pulse period and stops it again to repeat these operations. When the overflow pulse is received, MULST disables MULTC and the multiplication is completed. The state-tables for MULTC and CHKLB are given below.

Present	Next-state & present outputs.							
State	000	001	011	010	110	111	101	100
1	1,0	2,2	1,0	1,0	1,0	1,0	1,0	1,0
2	1,0	2,2	3,1	1,0	1,0	2,2	2,2	1,0
3	1,0	3,1	3,1	1,0	1,0	4,1	4,1	1,0
4	1,0	2,2	1,0	1,0	1,0	4,1	4,1	1,0

Table 5.15 State-table for MULTC.

Present	Next-state & present outputs							
State	000	001	011	010	110	111	101	100
1	1,0	1,0	1,0	1,0	2,1	1,0	3,2	2,1
2	1,0	1,0	1,0	1,0	2,1	2,1	2,1	2,1
3	1,0	1,0	1,0	1,0	4,0	4,0	3,2	3,2
4	1,0	1,0	1,0	1,0	4,0	4,0	4,1	4,1

Table 5.16 State-table for CHKLB.

An arithmetic shift (i.e. sign extension) is needed for the multiplication routine. However, the shift instruction may define an arithmetic shift or rotation depending whether the address field is written as '0XXXX' or '1XXXX', respectively. Hence, the data to be placed into the topmost bit of AC is determined by using a combinational component called SHIFT. Its truth-table is given below. Figure 5.13 shows its connections to the accumulator.

I1	I2	I3	I4	O1
0	0	0	0	0
0	0	0	1	0
0	0	1	1	0
0	0	1	0	0
0	1	1	0	1
0	1	1	1	0
0	1	0	1	0
0	1	0	0	0
1	1	0	0	1
1	1	0	1	1
1	1	1	1	1
1	1	1	0	1
1	0	1	0	0
1	0	1	1	1
1	0	0	1	1
1	0	0	0	1

Table 5.17 Truth-table for SHIFT.

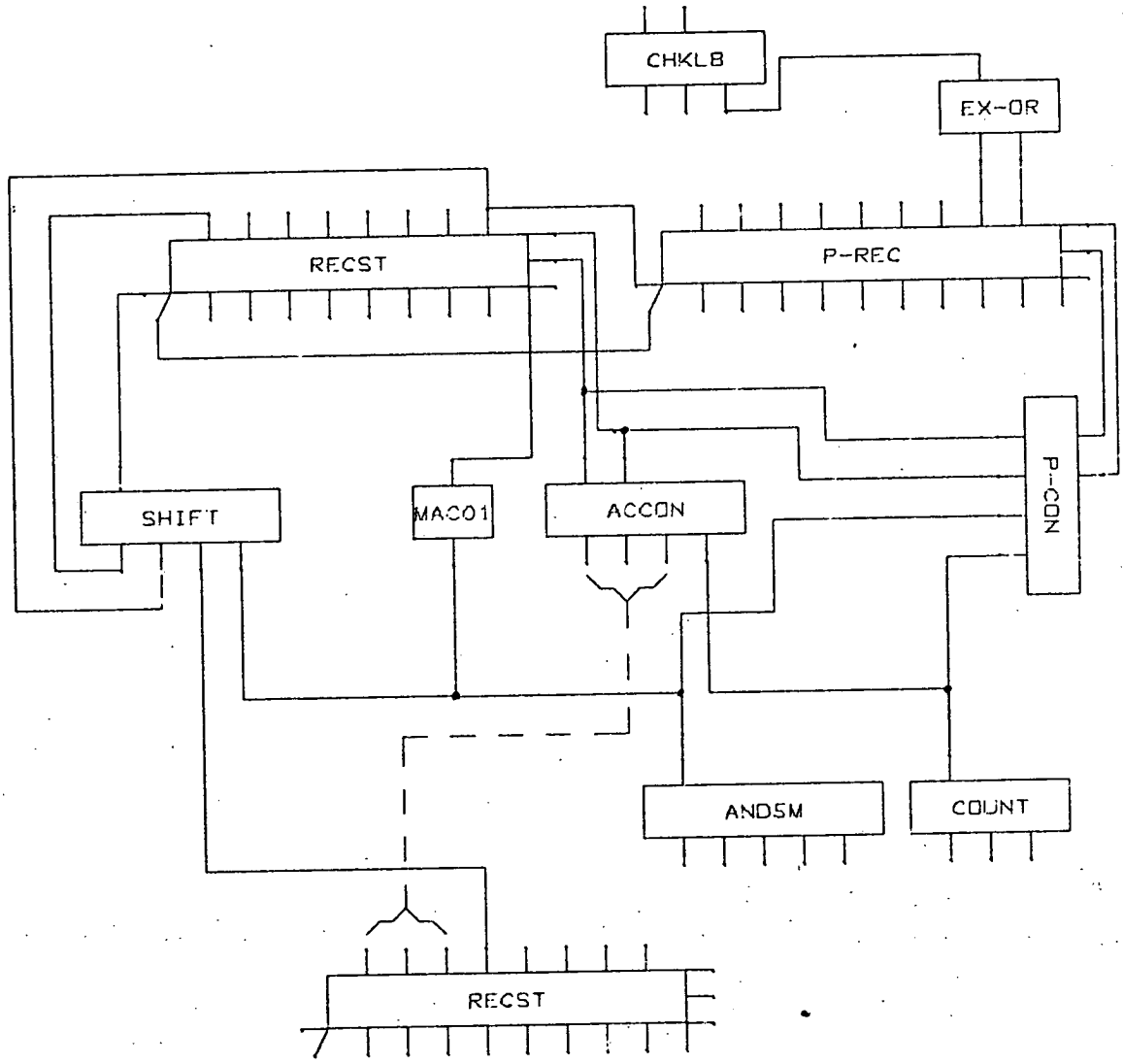


Figure 5.13 AC and P-register.

Figure 5.13 also shows how the mode bits of the AC and the P-register are set. The truth-tables for ACCON and P-CON are given below. MACO1 generates a short pulse at the beginning of a multiply routine and is used to clear the accumulator.

I1	I2	I3	I4	O1	O2
0	0	0	0	1	1
0	0	0	1	0	1
0	0	1	1	0	1
0	0	1	0	1	1
0	1	1	0	1	1
0	1	1	1	0	1
0	1	0	1	0	1
0	1	0	0	0	0
1	1	0	0	0	1
1	1	0	1	0	1
1	1	1	1	0	1
1	1	1	0	0	0
1	0	1	0	0	0
1	0	1	1	0	1
1	0	0	1	0	1
1	0	0	0	1	1

Table 5.18 Truth-table for ACCON.

I1	I2	I3	I4	O1	O2
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	1	0	0
0	0	1	0	0	0
0	1	1	0	0	0
0	1	1	1	0	1
0	1	0	1	0	0
0	1	0	0	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	1	1	1
1	1	1	0	0	0
1	0	1	0	0	0
1	0	1	1	1	0
1	0	0	1	0	0
1	0	0	0	0	0

Table 5.19 Truth-table for P-CON.

Figure 5.14 shows how the mode bits of the ALU are set. The truth-table for ALCON is given in Table 5.20.

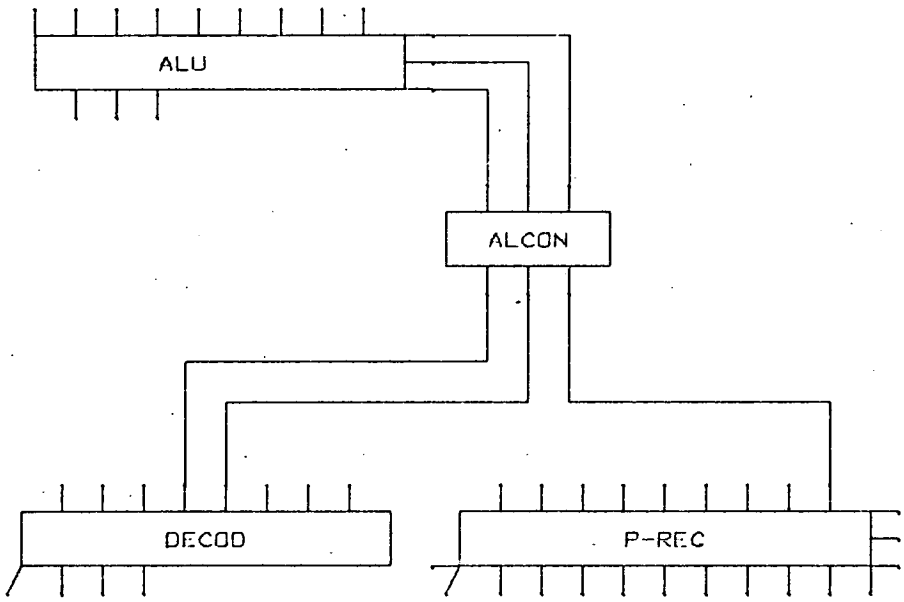


Figure 5.14 Controlling the ALU.

I1	I2	I3	O1	O2	O3
0	0	0	0	0	1
0	0	1	0	1	0
0	1	1	0	0	1
0	1	0	0	0	1
1	1	0	0	0	0
1	1	1	0	0	0
1	0	1	0	1	0
1	0	0	0	1	0

Table 5.20 Truth-table for ALCON.

Finally, Figure 5.15 shows how the clock input of AC is obtained.

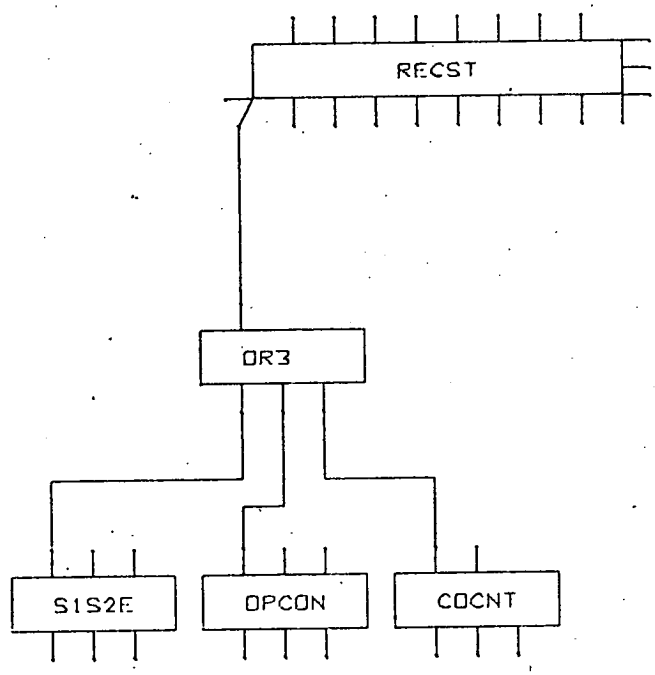


Figure 5.15 Clock Pulse for Accumulator.

5.3. Performance evaluation.

The design presented above has been tested using the simulator program. A number of modifications were made to the initial component and/or circuit specifications until the final form was obtained. Then the total design was checked by placing a simple test program into the program memory. Data values were placed into the register (and the data memory) and a large scale simulation was made.* The results of the operations were compared with hand worked solutions and were all correct. A number of spurious spikes were detected and some of them were eliminated. The remaining ones do not cause any malfunction. The access time of the two memory units were set to 750 and 30 nano seconds for the program and data store, respectively. The ALU performs operations in 40 nano seconds and the continuous clock to the shift counter has a frequency of 12.5 MHz. The propagation delays and the state transfer times of the remaining components are all 10 nano seconds. With these values, the fetch cycle is 850 nano seconds long. The duration of the execute cycle for different operations is given below.

Instruction	Execution time
Load AC	270 ns
Store	250ns.
Add	380 ns.
Subtract	380 ns.
Shift (3 times)	360 ns.
Shift (5 times)	530 ns.
Multiply (requires 5 additions)	2210 ns.
Rotate (21 times)	1810 ns.

(*) See pages 203-205 for samples of simulator outputs.

Multiplication can be made faster with a slight modification to the circuits such that the Q-register is loaded to the highway only when an addition or subtraction is required (at the present, Q is loaded to the highway before each shift operation). The total cpu time taken to simulate 14 instructions (19600 nano seconds) was approximately 28 minutes using 59980 kilo-core-seconds (i.e. approximately 59980/800 = £75 running cost).

5.4. Component realisation.

This section describes how the design components can be realised. As an example, a highway component with three data ports (i.e. sharable among three units) is implemented using gates and flip-flops. A sequential component is used for the priority logic. Therefore the implementation of user defined components is also illustrated. Table 5.21 gives the state-table of a sequential component that determines which of the three request inputs (R_1, R_2, R_3) will be given the control of the highway (indicated by the outputs O_1, O_2 and O_3, respectively).

Present	Next-state & present outputs							
State	000	001	011	010	110	111	101	100
1	1,0	4,1	3,2	3,2	2,4	2,4	2,4	2,4
2	1,0	4,1	3,2	3,2	2,4	2,4	2,4	2,4
3	1,0	4,1	3,2	3,2	3,2	3,2	2,4	2,4
4	1,0	4,1	4,1	3,2	2,4	4,1	4,1	2,4

Table 5.21 State-table for Priority Controller.

As the component is to operate in asynchronous mode, a state assignment free from critical races (may have non-critical ones) is necessary. Table 5.22 gives a modified state transition-table and shows the binary assignments for the states. There are race conditions under the 1st, 2nd, 4th and the 8th columns but none of them are critical. Output values are the same as those shown in Table 5.21.

Binary	Present	Next-state							
Code	State	000	001	011	010	110	111	101	100
00	1	1	4	2	2/3/4	2	2	2	2
01	2	1	1/3/4	3	3	2	2	2	2
11	3	1/2/4	4	3	3	3	3	2	2
10	4	1	4	4	3	1	4	4	1/2/3

Table 5.22 Modified Transition Table

Next, an internal variable map is obtained by replacing the next-state entries with their state assignments. When there is a multiple choice (e.g. 1/2/4 entry in the 1st column) some of them are eliminated by examining the table so that the number of adjacent entries is increased.

y_1y_2	000	001	011	010	110	111	101	100
0 0	00	10	01	-1	01	01	01	01
0 1	00	1-	11	11	01	01	01	01
1 1	0-	10	11	11	11	11	01	01
1 0	00	10	10	11	00	10	10	0-

Table 5.23 Internal Variable Map

Then the flip-flop type is selected. Here, J-K flip-flops are used. Table 5.24 gives the transition table for a J-K flip-flop.

Q_i	Q_{i+1}	J	K
0	0	0	-
0	1	1	-
1	1	-	0
1	0	-	1

Table 5.24 Transition Table for a J-K Flip-flop.

Excitation maps are obtained for each flip-flop using tables 5.23 and 5.24.

y_1y_2	000	001	011	010	110	111	101	100
0 0	0-	1-	0-	--	0-	0-	0-	0-
0 1	0-	1-	1-	1-	0-	0-	0-	0-
1 1	-1	-0	-0	-0	-0	-0	-1	-1
1 0	-1	-0	-0	-0	-1	-0	-0	-1

(a) 1st flip-flop

y_1y_2	000	001	011	010	110	111	101	100
0 0	0-	0-	1-	1-	1-	1-	1-	1-
0 1	-1	--	-0	-0	-0	-0	-0	-0
1 1	--	-1	-0	-0	-0	-0	-0	-0
1 0	0-	0-	0-	1-	0-	0-	0-	--

(b) 2nd flip-flop

Table 5.25 Excitation Map.

The excitation equations are then obtained from Table 5.25 and simplified.

$$J_1 = \underline{\bar{R}_1\bar{R}_2R_3} + y_2\bar{R}_1R_2$$

$$K_1 = \bar{y}_2R_1\bar{R}_3 + y_2R_1\bar{R}_2 + \bar{R}_2\bar{R}_3$$

$$J_2 = \underline{\bar{R}_1R_2\bar{R}_3} + \bar{y}_1R_2 + \underline{\bar{y}_1R_1}$$

$$K_2 = \bar{R}_1\bar{R}_2$$

An output map is formed to obtain the output equations.

y_1y_2	000	001	011	010	110	111	101	100
0 0	000	001	010	010	100	100	100	100
0 1	000	001	010	010	100	100	100	100
1 1	000	001	010	010	010	010	100	100
1 0	000	001	001	010	100	001	001	100

Table 5.26 Output Map

$$O_1 = \underline{\bar{y}_1R_1} + y_2R_1\bar{R}_2 + \bar{y}_2R_1\bar{R}_3$$

$$O_2 = y_1y_2R_2 + \bar{y}_1\bar{R}_1R_2 + \underline{\bar{R}_1R_2\bar{R}_3}$$

$$O_3 = y_1\bar{y}_2R_3 + \underline{\bar{R}_1\bar{R}_2R_3}$$

Figure 5.16 gives the logic diagram of the complete circuit.

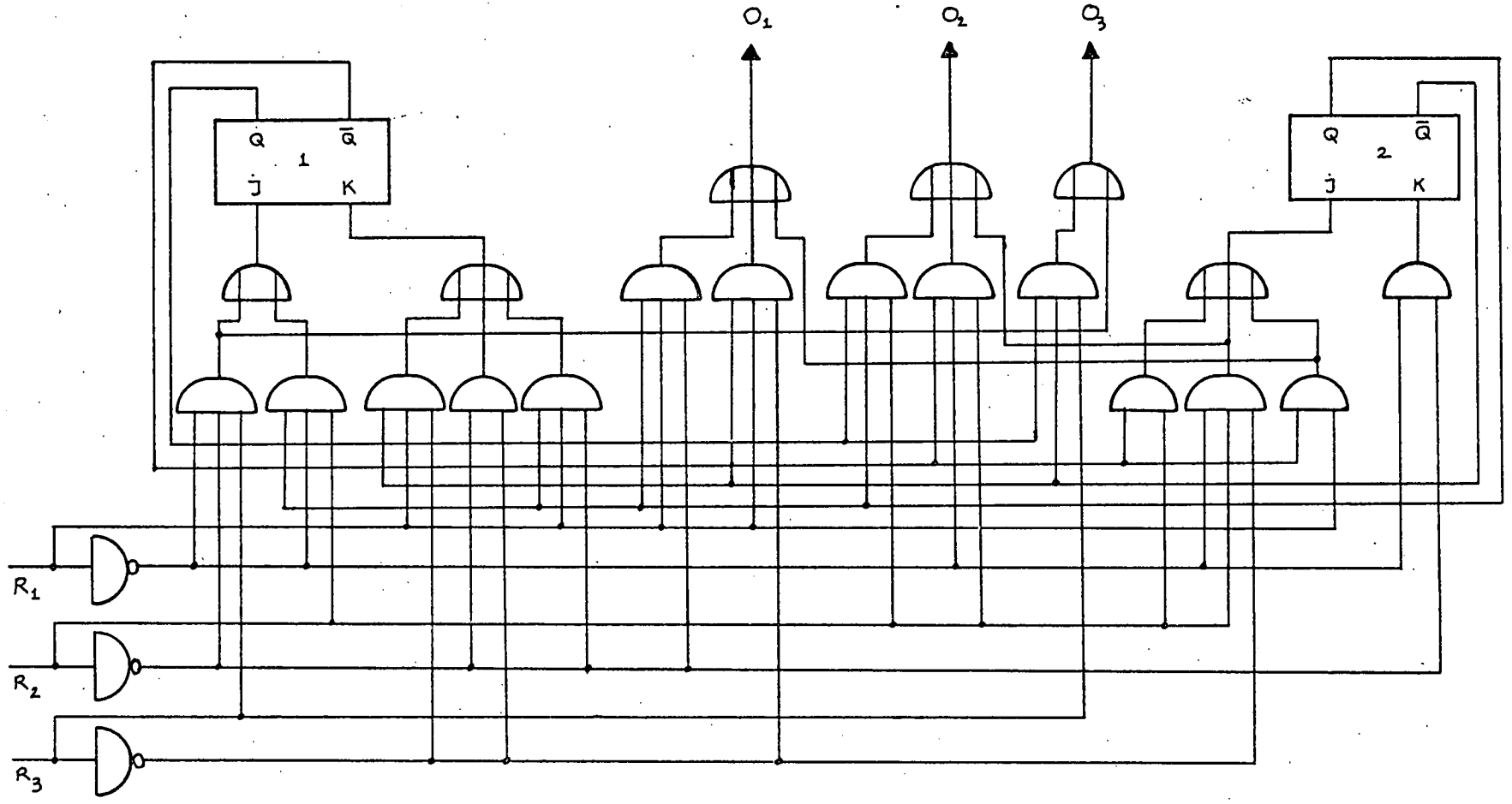


Figure 5.16 Logic diagram of the priority circuit.

The outputs 0_1, 0_2, 0_3 are used to select the data to be loaded into the highway flip-flops. This is shown in Figure 5.17.

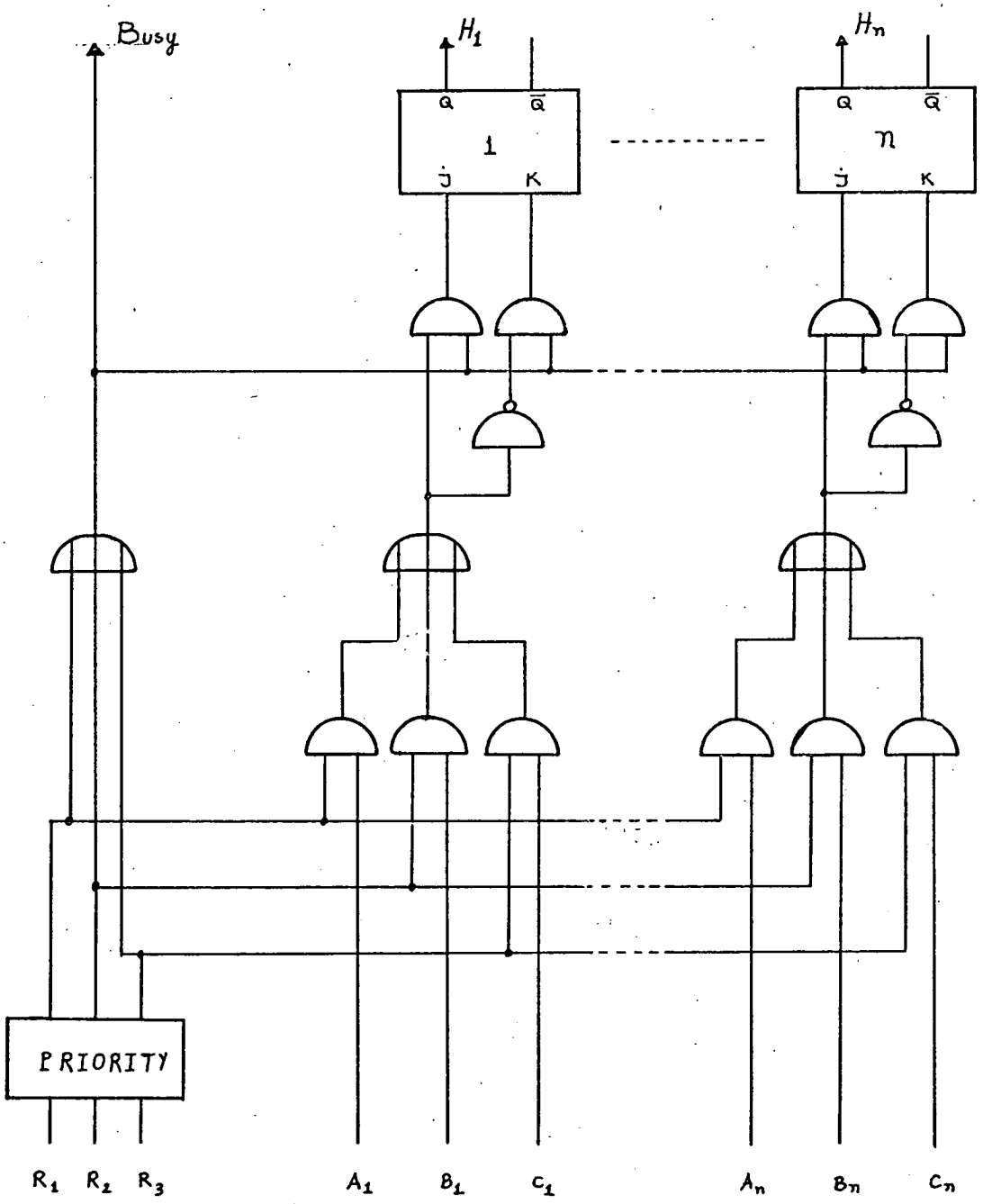


Figure 5.17 Realisation of the Highway Component.

Chapter seven discusses some of the techniques used in realising sequential and combinational components.

Also, a new state-assignment method for synchronous sequential machines is presented.

Chapter 6: Programming Considerations

This chapter describes the way in which logic designs prepared by the user are stored inside the computer. It also gives a closer view of the three programs described in Chapter 4 and discusses some ergonomic considerations.

The data structure is not of the generalised 'ring-structure' type. Such arrangements were examined and taken into consideration but the structure implemented in the programs is a different one. There are two main reasons for this. Firstly, when the research was started, it was not known what type of information should be linked together. For example, should there be a separate ring for display information or should the coordinates and the display file names of components be placed on the same ring as the connections. Furthermore, the type of information to be stored was not fully determined. 'Page','multi-line', 'macro' facilities were added on at a later stage. Secondly, the data structure was expected to be (and is) of a moderate size so that no special techniques would be required for 'garbage collection' or modifications. However, in the light of the experience gained, a more general type of data structure could be built for the future versions of the programs.

6.1 Data structure organisation

The data structure consists of a number of tables all of which are prepared on-line as the programs are instructed to carry out the required operations. The user is unaware of these 'behind-the-scene' operations but, undoubtedly, their efficiency affects the performance of the programs.

A table, called the "MASTER" table, is prepared (or modified) by the component specification program. In this table each component is represented by a master-block which carries some global definitions. These definitions include the name, type, number of inputs, number of outputs and the number of internal states of the component. The tabular descriptions (state-table or truth-table) of user defined components are also included in this block. Master-blocks of library components are entered into the table only when they are used in a logic diagram so that the table will not be crowded by unused components. Figure 6.1 shows a master-block.

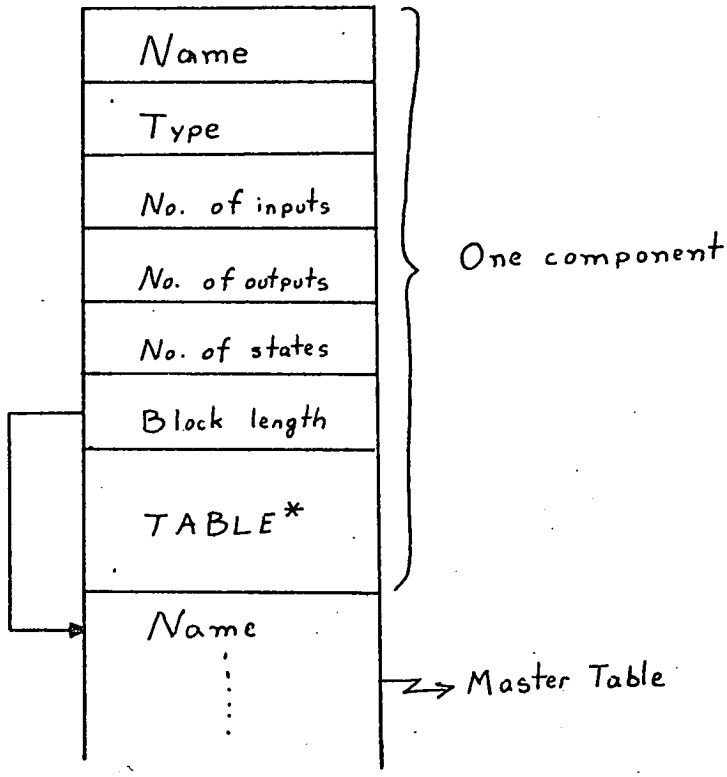


Figure 6.1 A MASTER-block

The drawing program prepares an "INSTANCE" table where each component in the logic diagram is represented by a block. Every instance-block points to a master-block and components which are used more than once are represented by multiple instance-blocks, all pointing to the same master-block. The master-instance relations are illustrated in Figure 6.2.

(*) Not needed for library components.

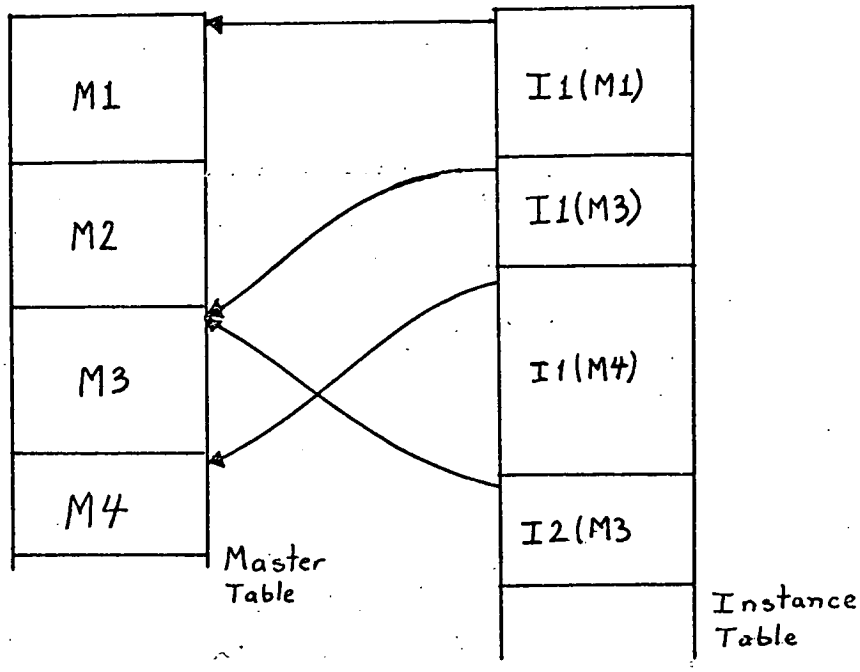


Figure 6.2 Master-Instance Relations

An instance-block stores certain values used during simulation and also indicates the connections to other components. Each terminal of a component has an associated word in the instance-block. These words are called "output-words" or "input-words" depending upon the type of terminal associated with them. Figure 6.3 shows an output-word.

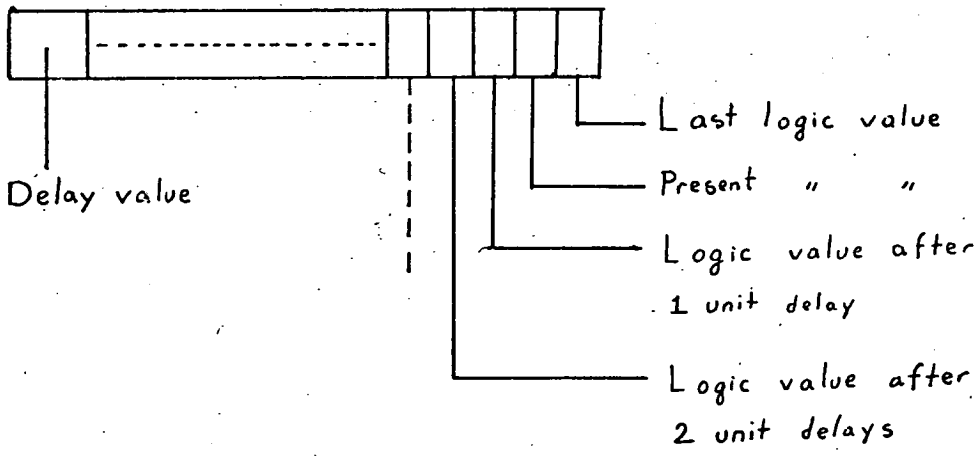
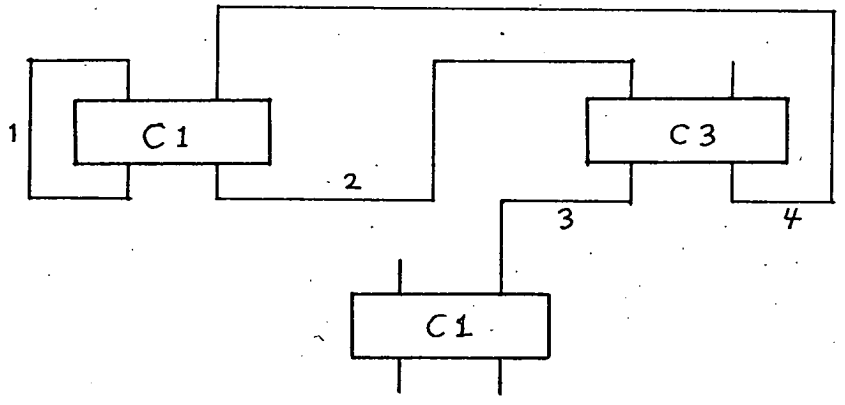
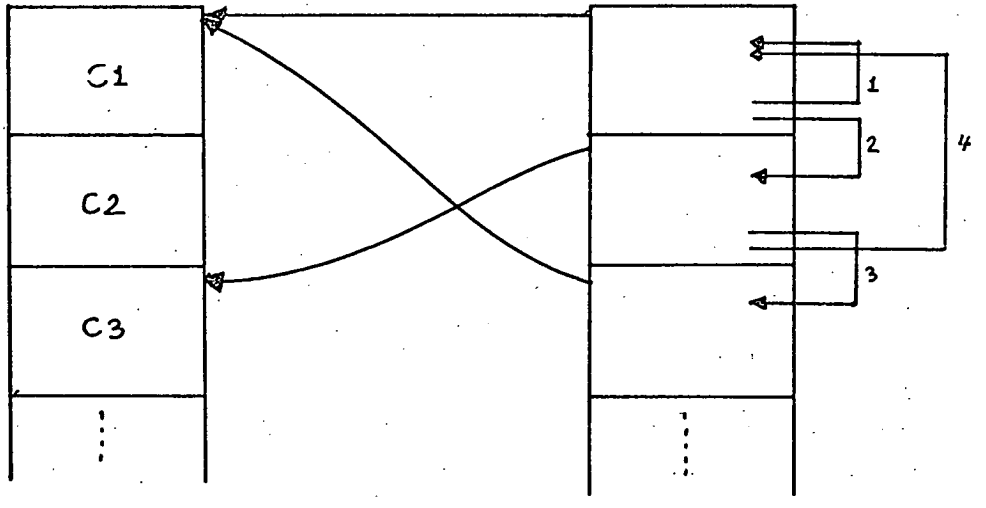


Figure 6.3 An Output-word

All components and lines (conductors) used in logic diagrams are given internal names and connections between components are represented by the "input-pointers" stored in input-words. The input-pointers can indicate not only which terminals are connected to each other, but also which line (or lines) makes the connection.



(a) Logic diagram



(b) Internal representation.

Figure 6.4 Pointers in the Instance Table

An input-pointer has five fields.

(i)- "V" field shows whether the pointer is directed at an input terminal (V = 1) or an output terminal (V = 0).

(ii)- "L1" field gives the internal name of the line which makes the connection.

(iii)- "L2" field gives the internal name of another line which is needed to complete the connection. L1 and L2 fields will be the same if the connection is directly between two terminals (i.e. only one conductor ties the two terminals together). However, if the connection is from a terminal to a line, L2 is the internal name of that line.

(iv)- "C" field identifies the component to which the connection is made.

(v)- "T" field identifies a terminal of the component in the "C" field.

Figure 6.5 shows some connections and input-pointers.

C_1, C_2 and the numbers on the lines are internal names and the pointers are written as "V, L1, L2, C, T".

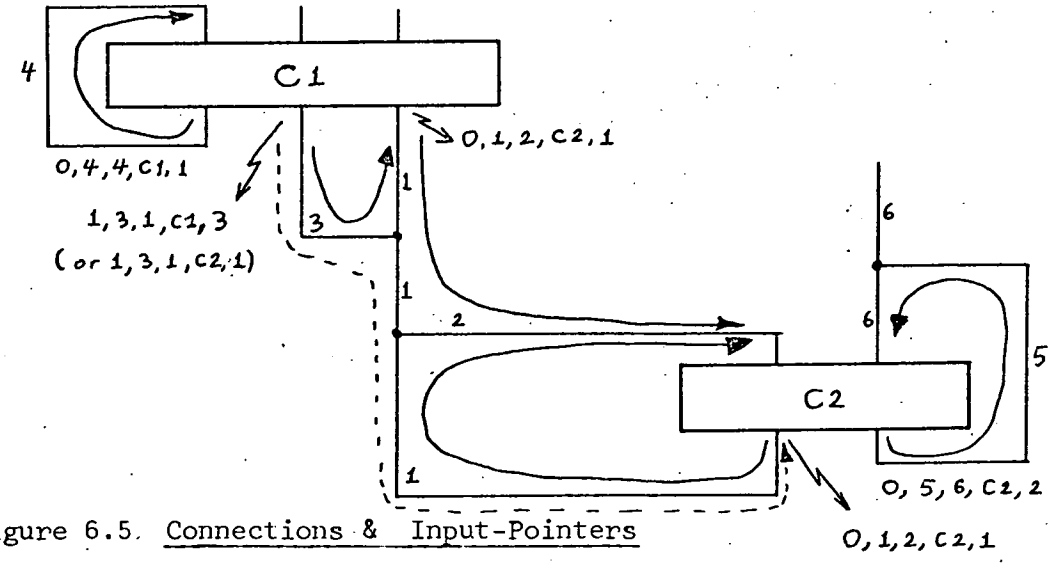


Figure 6.5 Connections & Input-Pointers

Note that the 2nd input of C1 does not directly point to the 1st output of C2. Instead it is seen to be connected to another input terminal which points to the required output. This is because the connection requires three lines (lines 3, 1, and 2) but the pointer has only two levels (L1 and L2 fields). Therefore, some inputs may have to point to other inputs and the connection to the output terminal (if any) can be found by tracing the pointers. It is necessary to retain the connectivity of the inputs to each other so that if the connection to the output terminal is deleted, the remaining terminals stay tied together. Also, the 2nd input of C1 could be made pointing to the 1st input of C2, as indicated by the dotted arrow. Either way, the same connections are represented and which one will be used depends upon the relative positions of C1 and C2 in the instance table.

When connecting a number of inputs to each other or an interconnected set of inputs to an output, the direction of the input-pointers may have to be changed. The technique used in sorting out the directions, is described below.

If two input terminals S_1 and S_2 are connected to each other, their pointers are arranged such that a 'loop' is formed.

Figure 6.6 Connection Loop

If S_1 is then connected to another input terminal, say S_3, the pointers along the path which starts from S_1 are followed until a loop is found. The loop is eliminated, directions of pointers along the path are reversed and a new loop is formed by pointing S_1 and S_3 to each other. This is illustrated in Figure 6.7

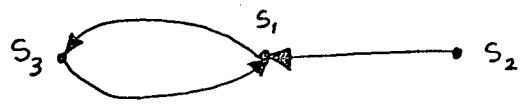
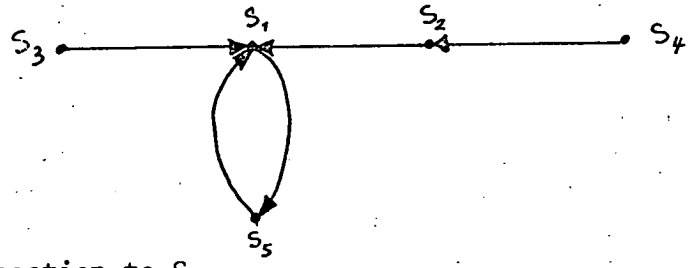


Figure 6.7 Forming a New Loop

The process is applied to any terminal which is connected to a new input. Figure 6.8 shows the situation if S_2 is connected to S_4 and S_1 to S_5.

(a) Connection to S_2



(b) Connection to S_1

Figure 6.8 Adding New Connections

It can be seen that after each connection the pointers are arranged such that all inputs point to the last terminal added to the

connections. Hence a 'directed-graph' is formed which terminates with a loop. Making a connection to an output terminal eliminates the existing loop but no new loop is created since output terminals do not have pointers. This is illustrated in Figure 6.9 where S_2 has been connected to the output terminal O.

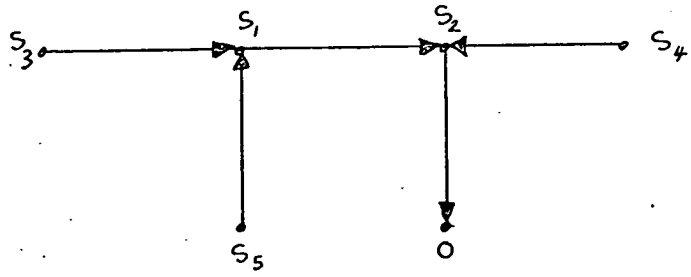


Figure 6.9 Connection to an Output Terminal

When connecting two terminals to each other, this procedure is applied to each of them separately (if it is an input terminal). The direction of pointers along one path can be changed only if the path terminates with a loop (i.e. no connection to an output terminal). If both terminals are outputs or no paths have been found for either of them, the connection is rejected since it will short-circuit two output terminals. When a connection is deleted, the recovery routines will insert a loop (loops) into the graph provided that the remaining path (paths) is not connected to an output terminal. Figure 6.10 shows the remaining pointers if the connection between S_1 and S_2 is deleted. The loop can be between S_1 and S_3 or between S_1 and S_5 depending upon the lines used in the connections.

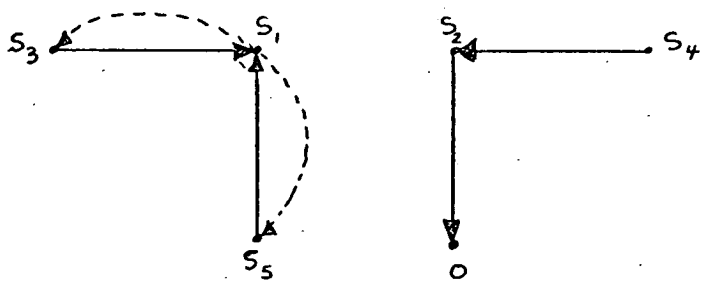


Figure 6.10 Deleting a Connection

Changing the direction of a pointer or inserting a loop is made such that the pointers always indicate the line (or lines) used in the connection. Figure 6.11 shows a logic diagram and the pointers stored.

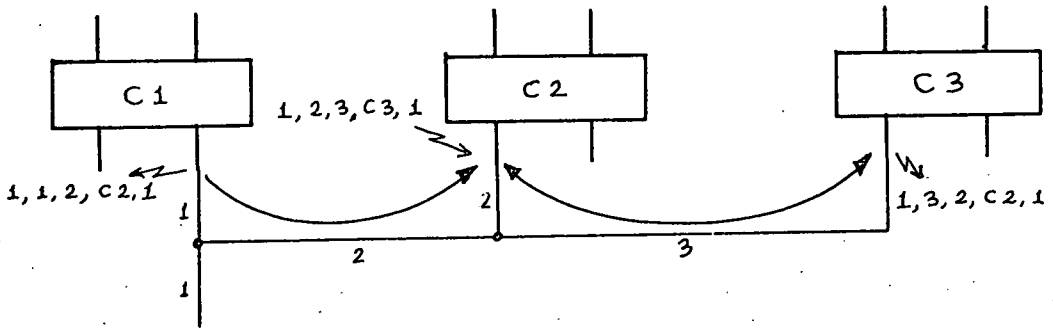


Figure 6.11 Inputs Connected to Each Other

The modified pointers are shown in Figure 6.12 where line 1 has been connected to an output terminal.

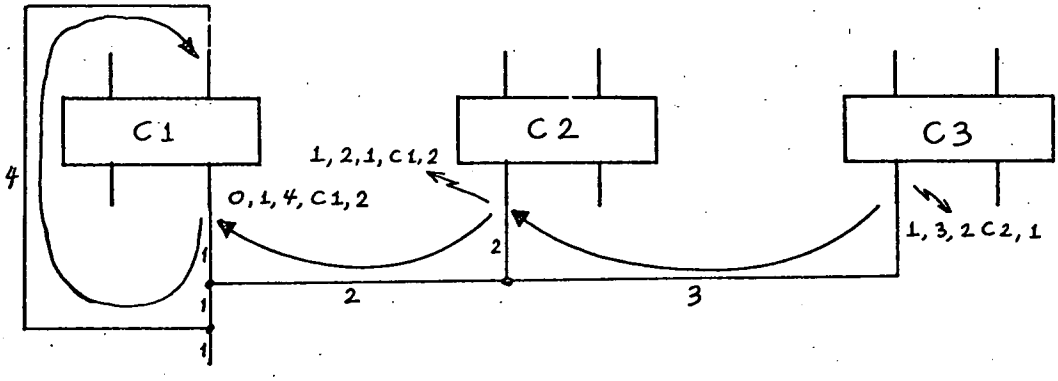


Figure 6.12 Modified Pointers After an Output Connection

If line 4 is deleted, a loop will be inserted between the 2nd input of C1 and the 1st input of C2. This is achieved by changing the pointer of the 2nd input of C1 so that it becomes "1, 1, 2, C2, 1". It can be seen that a line which is directly connected to an input terminal, appears in the "L1" field of the pointer on that terminal. An input-pointer such as "0, L1, 0, 0, 0" indicates that line L1 is connected to an input terminal which does not have any other connections.

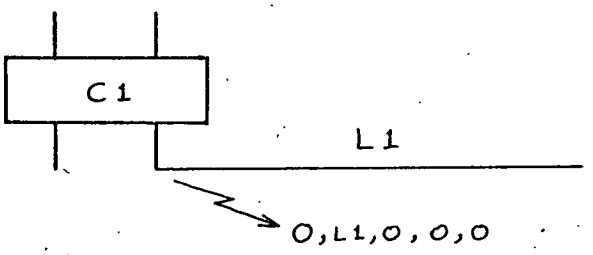


Figure 6.13 Free Input Terminal

This illustrates the efficiency of the pointer-structure in representing the connections. Even though, some types of connections (i.e. Figure 6.13) will not exist in the final logic diagram, it is necessary to be able to handle them since such connections may appear while modifying the diagrams. A "Multi-Line" connection is handled in the same way and the manipulations on pointers are applied to each pair of terminals of the connections implied by the multi-line.

The MASTER-INSTANCE relations separate the component specifications from the circuit descriptions so that either could be altered without effecting the other. The situation becomes somewhat complicated when MACRO components are used since their behaviour is determined by the components and their connections which form the body of the MACRO. Consider Figure 6.14 where a MACRO is defined.

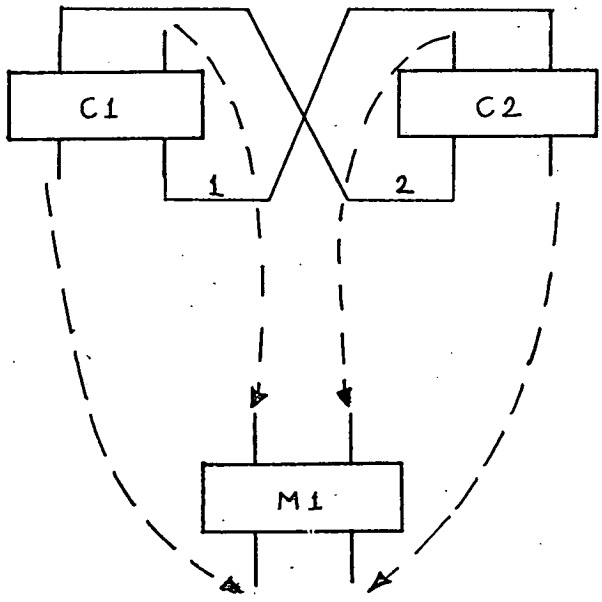


Figure 6.14 A MACRO Component.

C1, C2 and M1 are each represented by a separate instance-block so that the operations described above can be performed on either of them. Since the numbers of inputs and outputs of a component are given by its master-block, a new master should be prepared for M1. Figure 6.15 illustrates this.

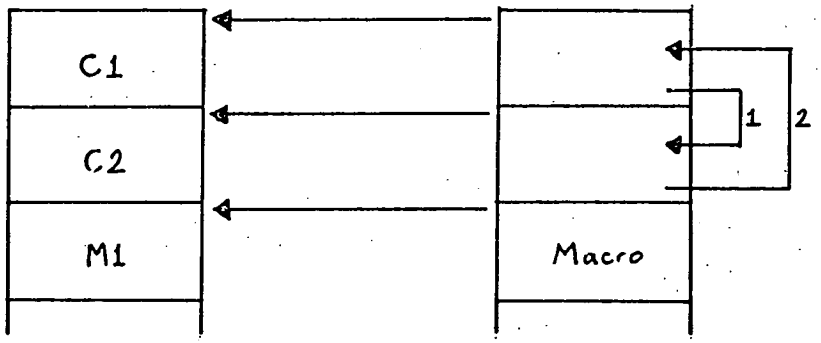


Figure 6.15 Macro Components.

A 'dummy' instance-block is used to identify the components which are used in the macro definition. The instance-block of the macro points to the dummy and the dummy points to the individual components.

A master-block is also needed for the dummy and this has been embedded in M1.

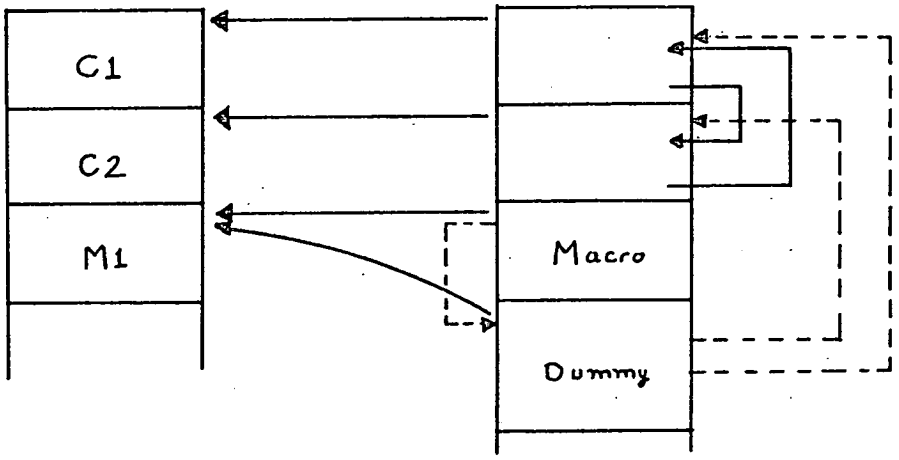


Figure 6.16 Dummy Block for Macro.

The dummy block identifies the 'internal' components of a macro and also shows which of their terminals are external to the macro (see Figure 6.14). If a macro component is used more than once, new instance-blocks will be created and they will all point to the same dummy.

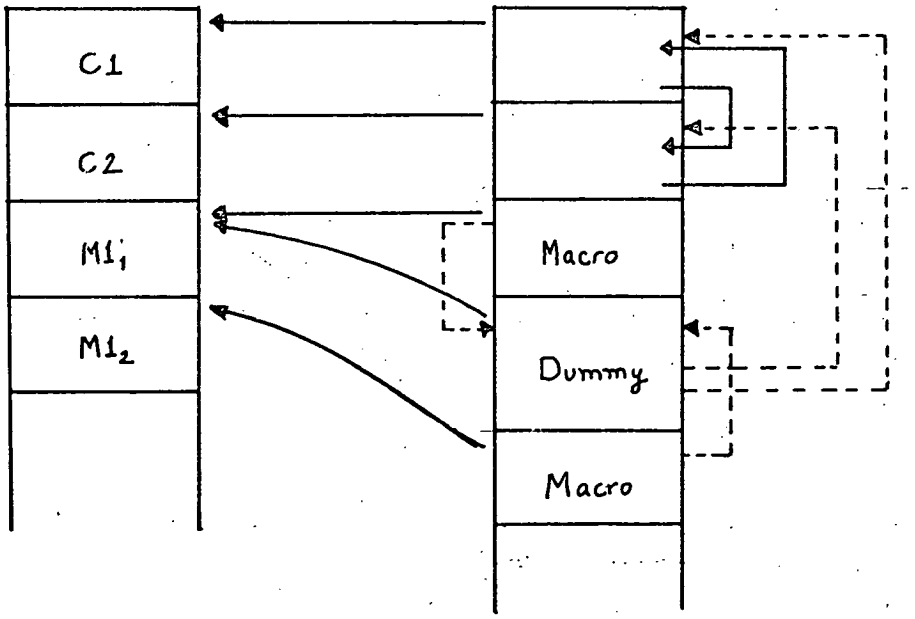


Figure 6.17 Multiple Instances of a MACRO.

Figure 6.17 shows that a new master-block has been generated for the second instance of the macro. This arrangement is required by the simulator. Ml_1 and Ml_2 have the same parameters (i.e. number of inputs,... etc.) and also have a storage area where the logic values of the outputs of the components used in the macro definition, are stored. To simulate a macro component, the simulator first carries the values from the master-block to these components, evaluates new outputs and stores them back in the master. Therefore, the structure of the macro in the instance-table, is "Re-entrant" and any component which shares that structure provides its own data. Since all instances of a macro component use the same structure, any modification to the latter will be automatically reflected in all. Connections to a macro are made on their instance-blocks. The simulator consults the dummy-block to find out which of the internal components should receive the external inputs and also which of the internal outputs should be carried to the external outputs of the macro. This section of the simulator has been written in a 'recursive' fashion so that the internal components of a macro can themselves be macro components and this can be done to any depth.

As mentioned in Chapter 4, partitioning a logic diagram does not require doing the same to the data-structure. A "PAGE" table is prepared which, for each component, gives the page number and the coordinates on that page. The program consults this table to see if (and where) a component should be displayed on the current page. Components which appear on more than one page are entered twice (or more) into this table and can be positioned or rotated individually. A similar arrangement has been made for the lines except that lines can appear only on one page.

6.2. Implementation details.

The three programs described in Chapter 4 have been written to provide the logic designer with a convenient tool which he can use to try out various designs and select the one which is best suited for the purposes in mind. Consequently, the data-structure has been organized such that component and circuit descriptions can be modified individually without affecting each other. Since most of the communication between the user and computer is done using the display screen, certain links are established between the objects on display and their internal representations. This has been achieved by dividing the display file (see Appendix C) into segments for each component or line and using the same internal name to identify a segment or its corresponding block in the data-structure. When a light-pen hit is recorded on any segment, the graphics software returns the internal name of that segment and the program does a search to find a data block with that name. Alternately when a new object is to be displayed, a new data block is created, the display segment is prepared and the same internal name is given to both of them.

To keep the interactions at a high level, all operations concerned with the internal representation of a logic design are hidden from the user. For instance, deleting a line between two terminals involves identifying the line, checking each of the terminals to find out where they are connected, modifying some pointers and finally, eliminating the line from the display. As far as the user is concerned, the modification has been fully specified by selecting the "ERASE" mode and pointing the light-pen to anywhere on the unwanted connection. In effect, he has issued an instruction to "delete that line". It is the responsibility of the program to find out which

line is to be deleted and which pointers are to be modified. When the required operations have been successfully carried out, visual feed-back is provided by causing the connection to disappear from the logic diagram. System response to such requests should be as quick as possible so that the user is not frustrated. In the current version of the program, the response time varies between 1 and 5 seconds depending upon the complexity of the operations to be performed. However, when the time-sharing computer is heavily loaded, delays of up to 30 seconds may be expected. It should be noted that most of that time will be spent in bringing the user's job into core.

In an interactive system, it is essential to know what the program is currently doing or what instructions the user can give to it. Therefore, when a particular mode is selected, the light-button corresponding to that mode is displayed at a larger scale than the others. Meaningful words have been selected for the light-buttons so that they can be used both to give instructions and also to indicate the "present-state" of the programs. It is also useful to give some of the light-buttons different meanings depending upon the current mode but care must be taken to ensure that the particular meaning is obvious. This technique is used in the drawing program. In the "ERASE" mode no action is taken until the user points the pen to a component or line to be deleted. However, if the program is in the "LINE" mode and the user has started making a connection, pointing to "ERASE" deletes the last part of the line being drawn and the program automatically returns to "LINE" mode. Similarly "FIX" is used to fix the end point of a line segment to allow changing direction. If the end point lies on a terminal (or a line) the

connection is finalised, otherwise the program returns to "LINE" mode. Hence, communications take the form of a simple graphic language which the computer can interpret according to the current state of the conversation, i.e. it is context dependent. Statements which have no meaning are ignored (e.g. pointing to a line and not specifying an operation). There is no need to say "exit from the current mode" since this is done automatically and the corresponding light-button returned to its original scale when the required operations are completed. Since logic drawings play an important role in the total design system, it was found worthwhile spending some effort to improve the quality of diagrams displayed on the screen. The components are displayed at a slightly higher intensity than the lines. This helps in recognising each component and makes it easier to understand the diagram. Also, if a line is connected to another line, a bright dot is displayed to distinguish it from a cross-over of connections. The end points of terminals have similar bright dots. As described in Chapter 4, multi-line connections are represented as dotted-lines. The program calculates a suitable length for small line segments and the gaps between them. This is done in such a way that the dotted-line starts and ends with a solid segment so that a visible corner is formed if the line changes direction. This is shown in Figure 6.18.

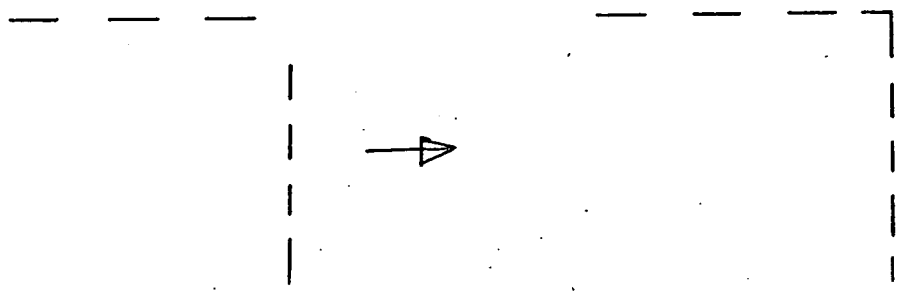


Figure 6.18 Multi-line Cornering

All programs have been written in FORTRAN. The size of the data-structure is about 5K for the design example given in Chapter 5. This does not include the display file which requires an additional 3K. The drawings are made on a 10 point, invisible grid such that the coordinates of the corners and the terminals of components are rounded off to the nearest 10. This helps in alligning the components such that connections can be made by horizontal or vertical lines. If a component is rotated by 180 degrees, its name is displayed with zero rotation (not upside down) so that it can be easily read. However, to avoid confusion in determining which terminals are inputs, in such situations the name is displayed closer to the input terminal side of the component. When the orientation is 90 or 270 degrees, the name of the component is also rotated.

Chapter 7: A General Discussion on Component Realisation.

The logical design of one of the high level components used in the design example, was given in Chapter 5. Logical design is a major problem area and has attracted the attention of many researchers. The following sections present a general discussion on the problems encountered in this field and on the techniques suggested by others to solve them. The author's contribution to two particular problems (finding clearing sequences and state-assignment of sequential machines) is described in sections 7.2.1 and 7.2.2, respectively.

7.1. Combinational components

Algorithms to find the prime implicants and select an irredundant set which covers all the 'one' outputs of a Boolean function can be found in references (60), (65), (90). Most of these algorithms can also be applied for the synthesis of multi-output functions. Hence, the combinational component specifications can be directly fed into some other programs to simplify and realise them. The outputs from such programs are usually in a two-level, sum-of-products (or product-of-sums) form and they should be reformulated so that special logic types (e.g. NAND/NOR) or multi-level synthesis can be used to implement them. Multi-level realisations may result in slower circuits but may be necessary because of fan-in limitations. When such constraints are satisfied the gates are assigned to IC packages and placed on printed circuit boards. Programs to perform such operations are reported in (25) and (78). It may sometimes be necessary to include redundant expressions in the final equations so that the logic circuits become hazard-free. A hazard is said to exist in a combinational circuit if transitions between a pair of

adjacent (differing in only one variable) input combinations cause a momentary spurious output. If the outputs are same for both of the input combinations, then the hazard is 'static', otherwise it is 'dynamic'. One way to prevent such faults is to use all of the prime implicants in the implementation of a function. This method guarantees a hazard-free realisation but may not be practical due to its expense. Hazards may also exist for multi-input-variable changes and these cannot be cured by using all of the prime implicants. Therefore, analytical techniques should be used to detect and prevent such cases (60).

It was mentioned in Chapter 4 that the user-defined components are limited to 5 inputs and 5 outputs. These figures are just about right if standard algorithms are to be used to perform the above mentioned operations. When the numbers of input and output variables are increased the problems become quite difficult to solve, even with the largest of computers. This is mainly because of the large number of terms generated by the algorithms when searching for the prime implicants. For large variable (e.g. 10 inputs, 10 outputs) problems, the core store and execution time required to obtain the theoretical minimum solution exceeds beyond acceptable limits. Even if the prime implicants can be obtained using a reasonable amount of core store and execution time, the problem of finding an irredundant set to realise the given function, is a formidable one. It is reported in (63) that to find an irredundant set of prime implicants for a 6 input, 10 output combinational network, 12000 covering terms were generated in 28 minutes on an ICL 1907 computer. Therefore, many researchers have turned their attentions to "approximate" algorithms which provide 'near optimum' solutions. Using such

techniques, first the largest (i.e. the simplest) prime implicant which will cover an input combination is found. Other inputs covered by the same prime implicant are deleted and the process is continued until all inputs have been covered. Description of an algorithm working in this manner can be found in a paper by A.R. Meo (19). Similar techniques are used in the CALD system (64). Unfortunately, such techniques are sensitive to the ordering of the inputs in the truth-table and different results may be obtained by simply reformulating the problem. However, the differences will be only in the 'non-essential' prime implicants since all 'essential' prime implicants can always be found (otherwise some inputs cannot be covered at all). Results obtained with such approximate algorithms indicate that considerable savings can be made in the cost of finding solutions with reasonable sacrifice in minimality.

7.2. Sequential components

The first step in realising a sequential component is to simplify the state-table by eliminating redundant states and merging the remaining ones (state reduction). This may reduce the number of memory elements (flip-flops) necessary to realise the component. The number of flip-flops required for a sequential machine with "N" states, is $K = \text{Log}_2(N)$. Hence, if the number of states is reduced from five to four, only two flip-flops will be needed, instead of three. However, reducing N from 8 to 5 does not reduce the number of flip-flops. Even then, it may be useful to simplify the state-table since eliminating some of the states introduce don't care values into the table and may lead to simpler excitation equations for the flip-flops. State reduction can be achieved by first constructing an "Implication Table" to find the set of all "Maximum

Compatibility Classes" and then finding a "Minimum Closed Set of Compatibility Classes" (90). This is a complex procedure and the execution time increases rapidly with the number of variables. There may be more than one minimum closed sets which require different states to be merged and at this stage, there is nothing to indicate that one of them will yield simpler equations than the others. Furthermore, while it is possible that some of the states can be eliminated and don't care values can be introduced into the table, there is also the very likely probability that any existing don't care entries will be destroyed due to the merging of states. Therefore, apart from the difficulty in obtaining a fully reduced state-table, there is no real guarantee that the excitation equations will be much simpler for a fully reduced table. Consequently faster working, approximate methods can be used for state reduction without any apparent loss of optimality (64).

The next step is to assign binary codes to the internal states such that the excitation equations for the flip-flops are minimised to the greatest possible extent. In fact, for the minimality of total cost, it is necessary to consider the excitation and output equations together. Asynchronous components require further attention to eliminate 'critical race' conditions. A race occurs if a change in the internal state of the component requires a change in more than one state variable. Since it cannot be guaranteed that all state variables will change at precisely the same moment, the internal-state may temporarily become different from either the present or the desired next-state. A race is called 'critical' if the next-state entries of these temporary states are such that the component may settle in one of them or branch to an

unwanted state. To date, there is no known method of state assignment which will give the theoretical optimum solution with or without treating the excitation and output equations together, for either the synchronous or asynchronous case. Complete enumeration of all possible assignments is totally out of the question since the number of possibilities are far beyond any acceptable limits*. Therefore again, approximate methods are used to solve the state assignment problem (34), (64), (82), (84), (86). Although some of these methods may sometimes give better results than the others, it is difficult to label any one of them as being the 'best' of all. This is because, the techniques used in a method may produce better results if applied to a certain type of sequential component. Unfortunately, some logic designers are not fully up to date with switching theory. They regard sequential circuits simply as circuits with memory that can operate in the synchronous or asynchronous mode and built according to the Moore or Mealy models (60). This is probably because the first attempts to apply the algorithms of classical switching theory to the simplification and realisation of logic circuits have not been very successful due to the inefficiency of these algorithms in handling large variable problems. However, recent advances in this field seem very promising and modern switching theory should prove to be an invaluable asset for logical design. Sequential circuits may be classified in a variety of ways; by examining the properties of state-partitions (55), (82), the existence or otherwise of homing sequences or distinguishing sequences (60), the length of the memory span (90), the amount of

(*) For a 9 state, sequential component, there are more than 10 million distinct assignments. The number of different assignments goes beyond 100 million.

feed-back (24), the linearity or otherwise of the next-state and output functions (28), (43),...etc. Research has been done in these areas and some of the results have been put into practice to obtain suitable state assignments for real problems. One new method is described in section 7.2.2.

When a suitable state-assignment is found, the flip-flop type is selected* and the excitation or output equations are extracted. These equations can then be realised using the techniques mentioned in the previous section. For sequential components operating in the synchronous mode, static or dynamic hazard detection for the excitation equations is not necessary. However, asynchronous circuits may suffer from yet another hazard condition, called 'essential hazard' which may be caused by the differences in the propagation delays of input and state variables. They can be eliminated if the state variables are slowed down such that all inputs propagate fully before the state variables start changing.

It can be seen that the correct operation of sequential circuits depends upon a number of factors. Hence, it is highly desirable that they can be tested easily. Fault detection in sequential components is not discussed here. However, an algorithm to find 'clearing sequences' which can be very useful in that area, is presented in the next section.

(*) It may be possible to consider the type of the memory element when looking for a suitable state assignment. However the author is not aware of any method doing this.

7.2.1. Finding clearing sequences

The formal definition of a sequential machine can be found in the literature (90). However, to clarify the following text, the definition of the next-state function is also given here.

The next-state function "g" of a sequential machine SM_1* with "n" inputs ($x_1, x_2, \ldots, x_n$) and "m" states ($S_1, S_2, \ldots, S_m$) is characterised by,

$$S_t = g(S_p, X_j)$$

where X_j is a min-term of inputs. S_p is called the present-state and S_t is the next-state.

SM_1 is said to be NON-CYCLIC if there exists an input sequence $I(k) = X_1, X_2, \ldots, X_q$ such that

$$S_k = g(\ldots g(g(S_i, X_1), X_2), \ldots X_q)$$

for all $i = 1, 2, \ldots, m$ and one k, $1 \leq k \leq m$.

I(k) is called a "Clearing Sequence" (CS) and brings SM to state S_k no matter which state (S_i) SM_1 starts from. SM_1 is CYCLIC if it is not non-cyclic. It is easy to show that for a 'strongly connected' (90), non-cyclic sequential machine, there exists at least one CS for any state S_k and no two states S_k, S_r have identical clearing sequences.+

At a given time, the set of states that SM_1 can be in, is called a "Present State Set" (PSS). "Next State Sets" (NSS) can be generated from a PSS such that

---

(*) The term SM_1 is used to distinguish this example from several others described later on.

(+) See next page.

Since SM_1 is non-cyclic there exists at least one state S_k such that

$$S_k = g(\ldots g(g(S_i, X_1), X_2), \ldots X_q)$$

where

$$I(k) = X_1, X_2, \ldots X_q$$

is a clearing sequence. Also, if SM_1 is strongly connected, by definition any other state S_j can be reached from S_k. If $T(k-j)$ is the sequence of inputs which brings SM_1 to state S_j when the initial state is S_k, then

$$I(j) = I(k), T(k-j)$$

is a clearing sequence for state S_j. It follows that there exists at least one clearing sequence for any state S_j.

Assume that two states S_k and S_r have identical clearing sequences (i.e. $I(k) = I(r)$). Since the starting state can be any state, it may include S_k. Then,

$$S_k = g(\ldots g(g(S_k, X_1), X_2), \ldots Xq)$$

and $$S_r = g(\ldots g(g(S_k, X_1), X_2), \ldots Xq)$$

which is possible only if $S_k = S_r$. Hence no two states may have identical clearing sequences.

$$\text{if } PSS_i = \{S_1, S_2, \ldots, S_r\}$$

$$\text{then } NSS_{i,j} = |\{g(S_1,X_j), g(S_2,X_j), \ldots g(S_r,X_j)\}|$$

$$= G(PSS_i, X_j).$$

where $|\{\ldots\}|$ is used to indicate that the set contains each element only once.

A state set with one element (state) alone, is called a "Terminal Set". The behaviour of a sequential machine can be described by constructing a "State Set Table" which, for each PSS, shows the corresponding NSS under a given input condition. Figure 7.1 gives the flow-chart of an algorithm to construct this table.

The algorithm starts by taking the total set of states $(S_1\ S_2, \ldots, S_m)$ as the first present-state set and generates the next-state sets from it for each input condition. Then the next-state sets are taken one by one and new sets are generated from them. The process is continued until no new state-sets can be found. Figure 7.1 gives the flow-chart of this algorithm.

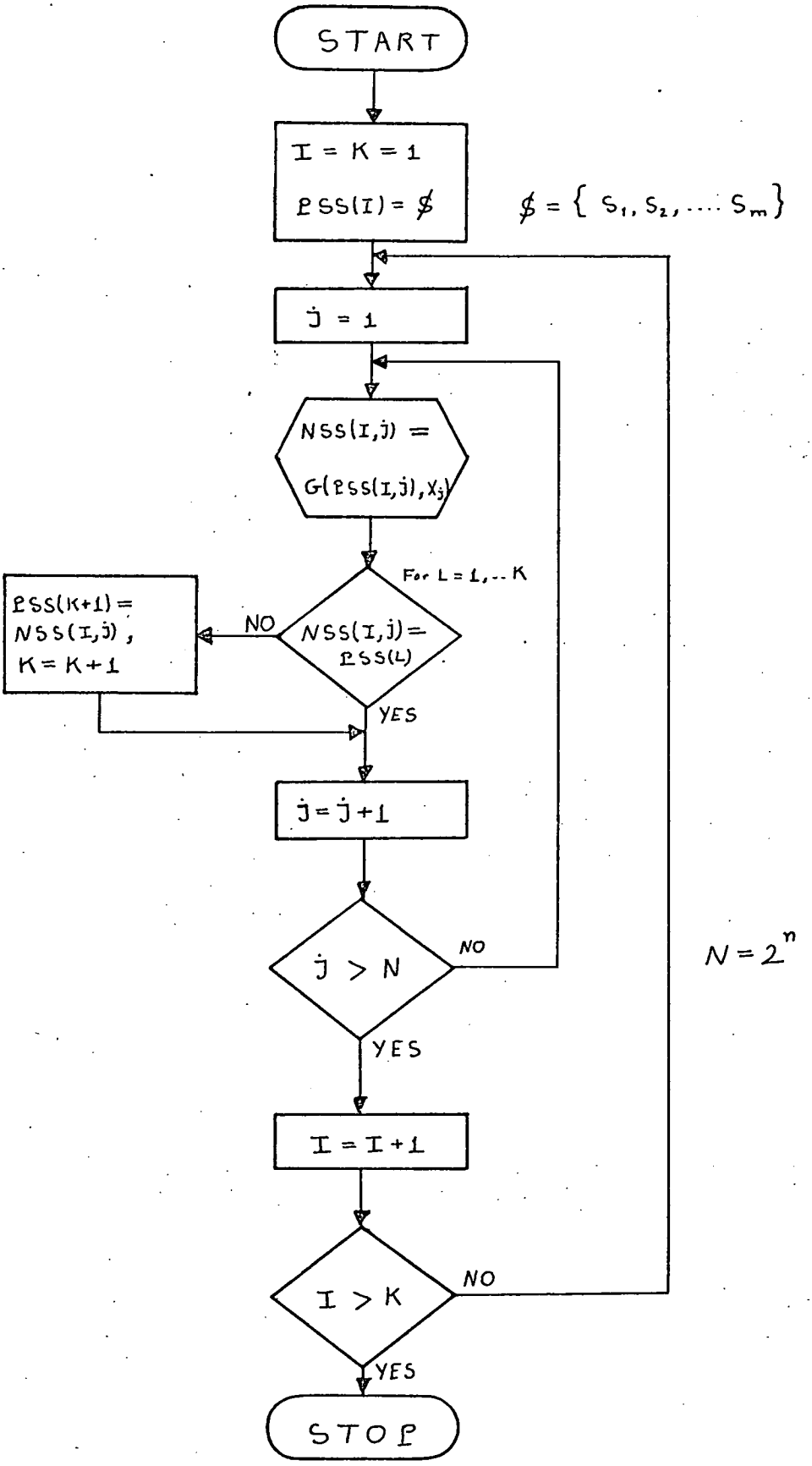


Figure 7.1 Constructing the "State Set Table"

Example The state-table of a single input sequential machine, SM_2, is given in Table 7.1

Present State	Next State X=0	Next State X=1
1	4	3
2	1	3
3	4	4
4	2	2

Table 7.1 State Table of SM_2.

The state-set table obtained by the application of the above described algorithm is given in Table 7.2.

PSS	NSS X=0	NSS X=1	Group Number
1,2,3,4	1,2,4	2,3,4	1
1,2,4	1,2,4	2,3	2
2,3,4	1,2,4	2,3,4	2
2,3	1,4	3,4	3
1,4	2,4	2,3	4
3,4	2,4	2,4	4
2,4	1,2	2,3	5
1,2	1,4	3	6
3	4	4	7
4	2	2	8
2	1	3	9
1	4	3	10

Table 7.2 State-set Table for SM_2.

Each PSS is given a group number as follows. Row one is group one. Any other PSS which is the NSS of a group preceeding it (say G_i), forms a new group (G_{i+1}). Note that a new PSS is formed only if it was not found before.

To find a CS for each internal state, first the terminal sets are found and underlined. If there are more than one terminal sets for any state, only those in the lowest group (smallest group index) are underlined. If all states of a sequential machine can be found in terminal sets, then the machine is non-cyclic and strongly connected. If some but not all states can be found in terminal sets, then it is non-cyclic but not strongly connected. Otherwise, it is cyclic. SM_2, is non-cyclic and strongly connected. Its clearing sequences can be found by starting from an underlined set and tracing its origin to the total set of states (S_1, S_2, S_3, S_4).
e.g. Clearing sequence for state 3 is found as follows.

State set 3 can be reached from state set 1,2 by x=1
" 1,2 " " " " " " 2,4 " x=0
" 2,4 " " " " " " 1,4 " x=0
" 1,4 " " " " " " 2,3 " x=0
" 2,3 " " " " " " 1,2,4" x=1
" 1,2,4 " " " " " 1,2,3,4" x=0

Thus I(3) = 0,1,0,0,0,1 is a clearing sequence for state 3.
Note that if a next-state set can be reached from more than one PSS, the PSS which is in the lowest group is selected. This ensures that the shortest CS is found. In the example given above, the set 2,3 can be reached from 1,2,4 or 1,4 or 2,4. Since 1,2,4 is in the lowest group, it is selected. However, the set 2,4 can be reached

from 1,4 or 3,4 both of which are in the same group. Hence, either of them can be selected without affecting the length of the CS.

It is possible to illustrate all of the shortest CS for each state of SM_2 by a "Distinguishing Tree".

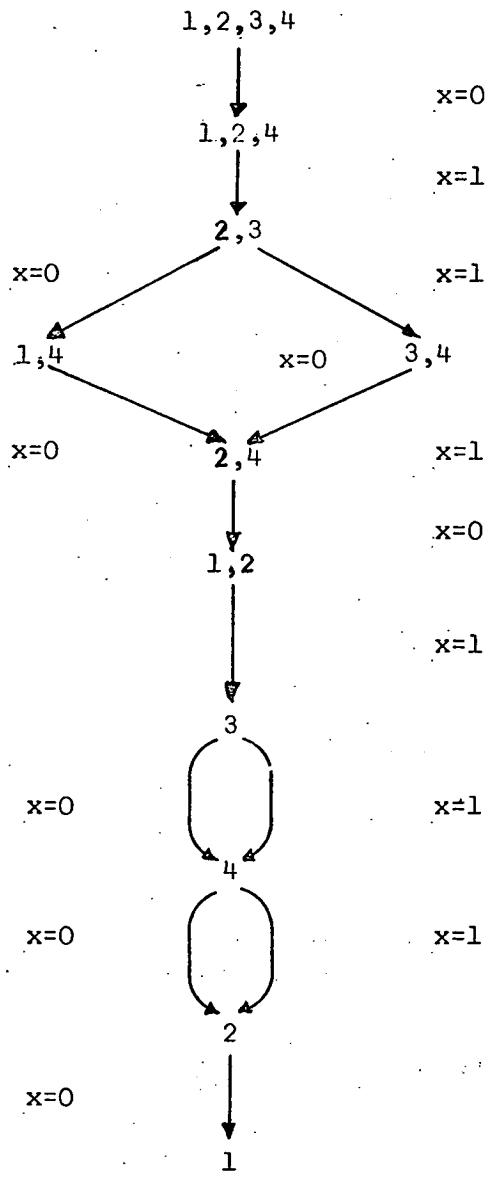


Figure 7.2 Distinguishing Tree for SM_2.

7.2.2. Application of clearing sequences to state assignment

This section describes a new method of state assignment for synchronous sequential machines. The method aims at assigning each state to a binary code derived from its shortest clearing sequence. It has been called the "Shortest Path Algorithm" and can be used only for strongly connected, non-cyclic sequential machines operating in the synchronous mode.

It was shown in the previous section that the clearing sequence for any state, say S_j, of a sequential machine SM_1 with "n" inputs, can be written as

$$I(S_j) = X_1, X_2, \ldots, X_q$$

where each X term is a min-term of "n" variables. When "n" is equal to one, X_i becomes a binary digit and $I(S_j)$ can be considered as a binary number with "q" bits. Assume now, that $I(S_j)$ is applied simultaneously to SM_1 and to a "q" bit shift register SR_1 which accepts serial data. SM_1 responds to the input data by changing its internal state and SR_1 does the same by shifting the data on the register. Since $I(S_j)$ is a clearing sequence for state S_j, at the end of the input sequence, SM_1 will be in state S_j and the contents of SR_1 will be the sequence $I(S_j)$. Therefore, the binary code assignment of S_j in SR_1 becomes $I(S_j)$. If the same is done to all states of SM_1, then the state assignments for a shift register realisation of SM_1 will be found. If all possible input sequences of length "q" are clearing sequences (not necessarily the shortest), then a perfect shift register realisation of SM_1 will be obtained. If some of the possible input sequences do not represent any clearing sequence at all, then the realisation of SM_1 will not be a perfect shift register but will still show such characteristics. Consider a very simple

example. Table 7.3 gives the state table of SM_2.

Present State	Next-State x=0	x=1
1	1	2
2	1	3
3	1	3

Table 7.3 State-table for SM_2.

The clearing sequences are

I(1) = 0

I(2) = 0, 1

I(3) = 1, 1

To make all clearing sequences have the same number of digits, the sequence for state one can be written as I(1) = -, 0. There are 4 possible input sequences with 2 bits and it can be seen that each one represents a clearing sequence. Hence, if every state of SM_2 is assigned to the binary code represented by its clearing sequence, a perfect shift register realisation of SM_2 will be obtained. Note that the state with the shorter clearing sequence (State 1) is assigned to two adjacent codes (00 and 10). The excitation equations are

$$Y_1 = Y_2 \; ; \; Y_2 = X.$$

The number of flip-flops required for the realisation is equal to the length of the longest one of the clearing sequences (i.e. two). This number may sometimes be much larger than the minimum number of flip-flops to implement a given sequential machine. Since a shift register realisation cannot be guaranteed, there is not much point in

using too many flip-flops with the hope that a good solution can be obtained. In such cases, the top bits of the clearing sequences can be dropped. This can be described as an 'overflow' of the shift register. Then the problem of finding the state assignments becomes more complex since two different sequences may become identical if their top digits are ignored. Thus, conflict situations may arise as more than one state has to be assigned to the same binary code. Before explaining how such conflicts are resolved, the case for sequential machines with more than one input will be discussed.

When "n" is greater than one, each digit (X_i) of a clearing sequence will be a binary number with "n" bits. For the clearing sequence itself to represent a binary number, it must be decoded such that each digit becomes "1" or "0". Since X_i can have 2^n values, a decoder with that many number of inputs is necessary. This may create some conflict situations since different sequences may become identical if they are decoded. There will not be an explicit decoder present in the final realisation but all decodings will be examined to find the one which causes the least number of conflicts. Table 7.4 lists all possible decoding functions for n = 2 (i.e. inputs x_1, x_2).

$F_1=\emptyset$	$F_5=X_2$	$F_9=X_1'$	$F_{13}=X_1'X_2'+X_1X_2$
$F_2=X_1X_2'$	$F_6=X_1+X_2$	$F_{10}=X_1'+X_2'$	$F_{14}=X_1+X_2'$
$F_3=X_1$	$F_7=X_1'X_2+X_1X_2'$	$F_{11}=1$	$F_{15}=X_2'$
$F_4=X_1X_2$	$F_8=X_1'X_2$	$F_{12}=X_1'+X_2$	$F_{16}=X_1'X_2'$

Table 7.4 Decoding Functions for Two Variables

For instance, F_7 transforms the sequence "00, 01, 11, 10" into the binary number "0101" and F_{10} transforms the same sequence into "1101".

A computer program has been written which finds all of the shortest clearing sequences for each state (there may be more than one minimal length clearing sequence for any state)and then resolves the conflicts caused by truncation or decoding. Since all possible decodings are examined, only sequential circuits with less than 4 inputs can be handled*. The following example describes how the program works.

Example:

The state-table of a sequential machine SM_3, with 5 states and 2 inputs, is given in Table 7.5.

Present State	Next-State 00	01	11	10
1	3	1	4	2
2	1	5	4	2
3	3	4	3	5
4	5	1	4	2
5	5	4	3	5

Table 7.5 State-table of SM_3.

First the clearing sequences are found. Table 7.6 gives the shortest clearing sequence for all states.

(*) Number of decoding functions is equal to 2^R where $R=2^n$ (n = number of inputs).

State	Clearing Sequence
1	00,01,01, or 01,01,01, or 11,01,01
2	00,01,10 or 01,01,10 or 11,01,10
3	00,00,11 or 01,00,11 or 11,00,11
4	00,00,01 or 01,00,01 or 11,00,01
	or 00,01,11 or 01,01,11 or 11,01,11
5	00,00,10 or 01,00,10 or 11,00,10
	or 10,01,00

Table 7.6 Clearing Sequences for SM_3.

Since all sequences have 3 digits and SM_3 has 5 states, a 3 flip-flop realisation will be derived without truncating any of the sequences. To examine the 16 possible decodings an "Assignment Table" is prepared. This table has 16 rows, each corresponding to a different decoding function, and 8 columns, each corresponding to a different binary code. On each row of the table, the states are mapped into the columns which represent the binary numbers obtained by decoding the clearing sequences with the decoding function of the row. For example, on row 7, state 1 is mapped into the column for '111' since F_7 transforms I(1)=01,01,01 into that number. On the same row, state 1 is also mapped into the column for '011' since F_7 transforms the other two clearing sequences into that number. Conflict situations appear when more than one state is mapped into the same column of a row. Table 7.7 shows the assignment table for SM_3. When the assignment table is ready, a score is given to each row by considering such factors as

(i)- the total number of mappings in a row,

(ii)- the number of non-conflicting entries (i.e. only one state mapped into a column),

Function	000	001	011	010	110	111	101	100
F_1	1,2,3,4,5	-	-	-	-	-	-	-
F_2	1,3,4	2,5	-	-	-	-	-	5
F_3	1,4	2,3,4,5	-	-	-	-	2,3,4,5	1,4,5
F_4	1,2,4,5	3,4	-	-	-	-	3,4	1,2,4,5
F_5	5	3,4	1,4	2,5	2	1,4	3,4	5
F_6	-	3,4,5	1,2,4	-	5	1,2,4	3,4,5	-
F_7	3	4,5	1,2	4	4,5	1,2	4,5	3
F_8	3,5	4	1	2,4,5	2,4	1	4	3,5
F_9	-	-	1,4,5	2,3,4,5	2,3,4,5	1,4	-	-
F_{10}	-	-	1,2,4,5	3,4	3,4	1,2,4,5	-	-
F_{11}	-	-	-	-	-	1,2,3,4,5	-	-
F_{12}	-	-	5	-	2,5	1,3,4	-	-
F_{13}	1,2	4,5	3	4,5	4,5	3	4	1,2
F_{14}	1	2,4	3,5	4	4	3,5	2,4,5	1
F_{15}	1,4	2	5	3,4	3,4	5	2,5	1,4
F_{16}	1,2,4	5	-	3,4,5	3,4,5	-	-	1,2,4

Table 7.7 Assignment Table of SM_3.

(iii)- the number of different states which appear as single entries,

(iv)- the complexity of the decoding function,

(v)- the score given to each column (calculated from the number of "1"s in the binary code of that column and the columns adjacent to it).

Then, the non-conflicting entries are selected from the highest scoring row, and assigned to the binary code of those columns. Assigned states and used columns are deleted, other rows are given an additional score if they contain similar entries to the selected ones and the process is repeated until all states are assigned. The algorithm works such that if in any row, there are more than one possible assignments for a state, all of them are taken. If at a later stage, one of the columns is required by another state, that column is released and the others are kept for the original state.

The program found the following assignments for SM_3.

Internal state	Binary code	Iteration step	Selected from row
5	100	1	2
2	010	2	5
1	011	3	8
3	000	3	8
4	001	3	8

Table 7.8 Assignments for SM_3.

The excitation equations for J-K flip-flops require 13 gates with a total of 41 gate inputs. For the same problem, results obtained by using other methods (34), (64), (84), (86) all require 13 gates and the number of gate inputs varies between 31 and 36. For this particular example, the results are no better than those obtained by other methods. This is not always the case.

The quality of the results depend upon the strength of the shift register relations inside the sequential machine. Table 7.9 gives the state-table of another sequential machine which shifts the data on one input to the left or right, depending upon the data on the second input.

Present State	Next-state 00	01	11	10
1	1	2	8	1
2	4	3	8	1
3	5	6	7	2
4	8	7	7	2
5	8	7	6	3
6	5	6	6	3
7	4	3	5	4
8	1	2	5	4

Table 7.9 Sequential Machine SM_4.

The realisation obtained by the method described above requires 18 gates and 36 gate inputs for the J-K flip-flops. These figures compare favourably with the results from another method (64) which require 27 gates and 60 inputs. Another state-table is given in Table 7.10. The realisation obtained by (64) requires 13 gates and

36 inputs. The shortest path algorithm found a perfect shift register realisation using 4 flip-flops (no additional logic is needed).

Present State	Next-State 00	01	11	10
1	2	1	2	1
2	3	1	3	1
3	4	1	4	1
4	5	1	5	1
5	5	1	5	1

(a) State-table

State	Code
1	---1
2	--10
3	-100
4	1000
5	0000

(b) State assignment

Table 7.10 Shift Register Realisation of a State-table

Don't care entries in the state-table can be ignored without causing any problem. e.g. Table 7.11 shows part of a state-table with a don't care entry.

Present State	Next-state x=0
5	3
6	-
7	5

Table 7.11 Part of a State-table with Don't Care Entry.

The next-state set generated from the present-state set 5, 6, 7 will be 3, 5. However, if the clearing sequences are to be used for testing purposes, the don't care entry should be changed to state 3 or 5 or a different branch of the distinguishing tree should be used.

7.3. Library components

Library components are, to a large extent, modelled on available integrated circuits. The main difference between a model and its IC counterpart is that, whereas IC's have a fixed number of bits, the size of the model can be changed by the user. Therefore, in most cases the realisation of a library component requires designing a larger circuit with the same input-output characteristics, from the smaller ones available on the market. For example, a 12 bit counter can be obtained by simply cascading three, 4-bit counters and connecting their control inputs in parallel. However, it may not always be possible to do this. A 12-bit universal register can be built from three, 4-bit ones but it is not so simple to realise a 10-bit register in this fashion. This is because the unused bits must be by-passed when performing shift operations. Hence, if the number of bits on the model is not a multiple of the number of bits on the IC, additional logic may be necessary to realise the library component.

One solution to this problem is to manufacture IC components with a small number of bits (e.g. two bits) so that different size components can be easily obtained. However, this approach is against the IC philosophy since it means a move towards using discrete components again. Therefore, the solution lies in manufacturing IC's which can be easily adopted to fit certain requirements. Additional inputs and/or outputs can be provided to give deeper access to the component.

An alternative and better way is to manufacture IC's as cellular arrays and custom-design the interconnection masks. These problems are for the component manufacturers to solve. Unfortunately, there is not a noticable movement in this direction. The main fault lies in the current design philosophies since the types of components to be used are considered at a very late stage. This point will be discussed further in Chapter 8 .

More complex library components may require a separate design cycle to realise them. For instance, a logical unit which can perform various logic operations (e.g. AND, OR,...etc) can be modelled as a single component. Since operations performed by such components are well defined (i.e. standard) their design is not a major task. In most cases the logic for one bit is repeated for all others and conventional techniques can be used to realise them. The 'highway' component deserves further comments. In many design circles a common highway is treated as a complicated data path. On its own, it is not seen as a component but rather as the necessary gating to pass data among several units. This thinking should be changed and the common highway should be treated as a component in its own right since it performs data operations under external control. The design example given in Chapter 5 illustrates this point. No claim is made about the usefulness of the particular highway component which was used but the concept is found to be very helpful in representing and designing digital systems.

Chapter 8: Proposals For Future Work.

The CAD programs described in Chapter 4 have been successfully applied to the design of a simple digital computer (see Chapter 5). Many of the facilities provided were found to be extremely useful. The most useful ones were 'Page' (including the facility to display components on more than one page), 'Multi-Line', and 'Trap' (see Chapter 4). However, several improvements can be made to the programs to make them more efficient and easy to use.

8.1. Changes to the component library

The example given in Chapter 5 indicates that digital systems can be designed by using a small number of different data components. However, the problem of selecting the particular components which should be included in the system library needs further investigation. Programming a whole range of IC's might appear to be a solution but this would create an undesirable side effect by introducing implementation details at a very early stage in the design process. Furthermore, currently available IC components are not ideally suited for all applications. It was mentioned in Section 7.3 that even a simple construction such as cascading two identical registers to increase the number of bits, may create unnecessary problems. It is quite clear that present MSI modules are unsatisfactory in many ways. This is due to the fact that it is the I.C. manufacturers and not the equipment designers who take the initiative in choosing the components to be produced. This, in turn, is due to existing design philosophies. In most cases the starting point to a digital system design is taken as a description of the required operations. Because of the wide variety of techniques used both for making the descriptions and turning them into hardware, every design requires

components of different nature. On the other hand, most design algorithms aim at finding solutions which use low-level switching components (i.e. gates or flip-flops). Therefore, enough demand cannot be accumulated to convince the manufacturers that the production of a particular high level component is justified. Hence the designers have to adapt their designs according to the state of the component market. Though some attempts have been made to design a set of universal modules which can be used in a wide variety of applications (e.g. PDP-16), on closer examination most of the proposals are found to be inadequate. A new component modelling philosophy is required to solve this problem. A set of highly generalised modules can be defined and placed in the library. Then, the designer can delete the unnecessary features to obtain a somewhat simpler version of the component. For example, in the case of a register, options available can be the number of bits, required operations (count, load, shift,... etc), external control facilities (e.g. a 'count enable' input), special outputs (e.g. to indicate that the counter is full),...etc. The actual IC's can be built in a modular fashion by 'plugging in' some special components to a basic module and (if required) using different interconnection masks during the production. The basic module should contain very little control logic and should be expandable. The required features can be added to it by connecting special components which provide the necessary logic to perform different operations. If necessary some of the connections inside the basic component can be changed by using a different interconnection mask.

8.2. Improvements to component specification.

The method of specifying a truth-table or a state-table can be improved by allowing the use of a teletype to input Boolean equations or state-transfer relations (expressed in a suitable language). Programs to accept this form of input need not be very complicated. In the case of combinational components all that is required is a syntax analyser which can recognise literals, logical operators and parentheses and generate the 'canonical' terms from the input expressions. However, a special language is necessary to describe the behaviour of sequential components. The language used can be a very simple one since there are no data operations to describe. More general languages of this nature are being used (64) to specify state-tables (or truth-tables) of much larger size than those used in the CAD system presented in Chapter 4. These programs also check for inconsistencies created when making the specifications. This is not required here since the tables are small enough to allow the user to detect mistakes by visual inspection. All forms of input (graphical or linguistic) should be made on-line and the display should be immediately updated, as is the case at the present.

Another useful facility would be to be able to describe a component as above and later repeat it a number of times. This can be achieved by defining a Macro component as required. However, simple repetition can be obtained more easily and without adding new blocks to the Instance table. Also, don't care values for outputs or internal states should be allowed and the simulator program should be modified to warn the user when a component branches into an unidentified next-state or if an input terminal receives a don't care value (three value simulation is another possibility).

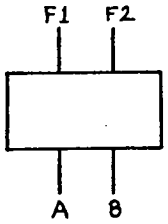
The use of the light-pen should be continued since it is often much easier to point at an entry to change its value, rather than to type an equation .

8.3. Additional display facilities.

In general, the generation and display of logic diagrams are quite satisfactory. However, certain modifications can be made to improve their quality and to make the programs easier to use. At the present, if a component is displayed on more than one page, a connection made to one of its terminals can be seen only if that connection is on the current page. Hence, if the user attempts to connect an input terminal to an output and if that terminal was already connectdd to an output terminal, the program rejects the attempted connection but it is not known what and where the other connection is. The user has to turn to another page to find that. This is a handicap and should be removed. It is difficult to indicate precisely to which other terminals a given one is connected. The problem is not only in finding a suitable form to show this (i.e. displaying the name and terminal number of the component to which a given terminal is connected) but also in updating this information as connections on other pages are modified. Therefore a simpler but less specific remedy has to be used. For example, an asterisk ('*') can be displayed on a terminal which is already connected. Note that if the connection is on the current page, the asterisk should not be displayed.

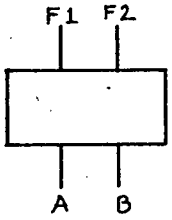
Another problem is to identify those terminals to which only 'multi-line' connections can be made (i.e. data-ports). Such terminals can be displayed with a slightly different representation

to allow easy recognition. For example, such terminals can be displayed longer or thicker than the rest. Also, it should be possible to interchange terminals according to the connections required. The user should be able to select a light-button and identify the terminals which he wishes to interchange. Nothing happens on the display but the truth-table or the state-table becomes modified as required. This is illustrated in Figure 8.1.



A	B	F1	F2
0	0	0	1
0	1	1	1
1	1	1	0
1	0	1	0

(a) Original



A	B	F1	F2
0	0	0	1
0	1	1	0
1	1	1	0
1	0	1	1

(b) After interchanging the inputs

Figure 8.1 'Interchange' Facility.

Note that the use of this facility should be allowed only when the terminals involved are not connected to any other terminals.

When a Macro component is defined its internal components are removed from the display but can be regenerated and modified. However, the order of the external terminals are not identified on

the display. For example, if the Macro has two inputs, it is not clear which internal terminal is the first and which is the second. This problem can be solved by displaying a number on the internal terminal to identify its position on the Macro.

It should be possible to give a different name to each individual copy of components which are used more than once. This can be achieved by storing the generic name (name first given when defining terminal behaviour, or by default for library components) in the Master block and storing another name in the Instance block. It would be a very simple task indeed to switch from one name to the other since this is only a display procedure.

8.4. Improvements to the data structure.

Some improvements can also be made to the present organisation of internal (computer) representation. Facilities should exist to link similar data structures together so that designs generated by different people can be brought together. Also, the values of output terminals should be stored in a different place and their position in the Instance block should be replaced by pointers to the storage area. This arrangement eliminates the need for transferring values to and from the master table when simulating a macro component. Instead, the pointers will be changed to get the same effect. To use the data storage more efficiently, a ring data structure can be formed. Separate rings should be formed for the Master and Instance blocks. However, on its own, this type of storage would not be satisfactory for simulation. Each Instance block should be able to locate the Master immediately (without any search) so that simulation can be carried out at full speed. Therefore, 'map' tables should be formed

to show the position of each block in the data structure. Then, all references to the Master or Instance blocks can be made via the map table. This arrangement would also make the blocks relocatable.

8.5. Increasing the speed of simulation

To increase the speed of simulation, a different simulator program could be written. The first change is needed in the type of the 'input pointers'. As shown in Chapter 6, with the present arrangement of pointers, some input terminals may have to point to other inputs thus forming a chain. During simulation, these pointers are followed until an output terminal is encountered. Timewise, this is a costly operation. Also, having the pointers at the input terminals rather than at the outputs is not a good arrangement if fast simulation is required. Instead, outputs should point to the inputs which they drive. In this arrangement an output may have more than one pointer since it may be connected to a number of inputs. When a component is simulated, the names of all components which receive an input from it can be entered into a circular buffer. The position in the buffer where the name of a component is entered depends upon the delay value of the output terminal. After each time step, the buffer is rotated so that the positions of the names of components which have been simulated can be used again. Hence only those components which become 'active' are simulated and the periods of time when all outputs remain stable are skipped over.*

(*) Note that autonomous components should be treated separately as there are no outputs to drive them.

These changes cannot be achieved as 'modifications' to the existing simulator and the program would have to be rewritten accordingly. One disadvantage that the new simulator would have is the necessity to 'pre-process' the data structure prepared by the drawing program before simulation can begin. The choice was made to have pointers at the input terminals rather than at the outputs since such is the requirement of the drawing program. Otherwise, only pin-to-pin connections between an input and output terminal can be made.

Chapter 9: Conclusions.

Existing CAD methods for digital systems design can be divided into two main categories. The most popular approach is to use a formal design language to describe the behaviour of the target system. These descriptions can be realised in hardware by using a variety of techniques some of which were described in Section 2.1. Although methods of this nature are very useful, most of them are unsatisfactory in expressing timing relations and system structure. Methods in the second group use flow-charts (or similar diagrams) to separately represent the flow of data and control signals in a digital system. Conversion into hardware is achieved by using special pre-designed components which realise the operations indicated by the flow-chart symbols. This approach is relatively new and a large scale design problem has not yet been solved in this manner. In general, such methods seem promising especially in designing control circuits. However, implementing the data operations presents some problems since system structure is not defined clearly.

Chapter 3 presents a different design philosophy which does not use abstract forms to make the descriptions. Operations to be performed by the target machine are divided into two types (data and control) and components are defined to realise them. Some of the components are provided by the design system whereas others are defined by the user. Generally accepted tabular methods are used to describe the terminal behaviour of user-defined components. Connections are made by drawing logic diagrams so that system structure is defined in a natural way. CAD programs have been written to apply these ideas and a design problem was solved to illustrate their use. The general conclusions drawn from in-

vestigating the existing methods of design and applying the author's approach to a practical problem are given below.

(i)- Separation of data and control operations from each other simplifies the problem of designing a digital system.

(ii)- Most data operations can be realised by using high-level standard components. Therefore, the CAD system should provide a library of suitable data components.

(iii)- Control operations can be realised by using sequential or combinational components. State-tables and/or truth-tables can be used for specifying the terminal behaviour of these components without committing oneself to a particular way to implement them.

(iv)- Drawing logic diagrams (using high level components) is a very convenient and natural way to describe the structure of a digital system. Computer graphics techniques make this form of input possible.

(v)- All types of descriptions should be input to the design computer as early as possible so that automatic processing of data can begin from the early stages on.

(vi)- Information in the computer should be kept as simple as possible and the component descriptions should be stored separate from the diagrams so that either of them can be modified without affecting the other.

(vii)- As yet, there is no direct substitute for high-level functional simulation to test the design of a digital system. Analytical techniques which appear to be more efficient have been proposed to replace simulation. However, the acceptance of such techniques depends not only on their accuracy but also on the design methods that go with them. Until the abstract forms of description used in these methods become generally accepted, simulation remains

the only other alternative.

Some of the ideas put forward by the author have already been expressed by other workers in this field. However, to the best of the author's knowledge the CAD programs described in Chapter 4 are the only existing programs which implement (or make it possible to implement) these ideas as a whole, forming a design suite with practical applications. Especially the implementation of the 'Multi-Line', 'Page' (including the display of components on more than one page) and 'Trap' facilities (see Chapter 4) appear to be novel. Also no reference could be found which describes techniques similar to the ones used in the internal representation of connections (Chapter 6) and the state assignment method presented in Section 7.2.2.

It is hoped that this research will motivate others in the same field so that viable CAD methods will emerge. It is difficult to visualise a single method being used generally, because of the wide variety of disciplines adopted by the computer manufacturers. Nevertheless, for the continued development of digital systems, more powerful CAD systems are absolutely essential.

Acknowledgements

The author wishes to express his gratitude to his two supervisors Dr. J.V. Oldfield and Prof. F.G. Heath (Heriot-Watt University) without whose generous help, advice and direction this research would not be possible. Thanks are due to my wife for typing the draft and to Miss H. Ritchie for typing the original form of this thesis. A special word of thanks to Mr. A.C. Kilgour who made quite a number of improvements in the SPINDLE package. His responsiveness to the problems encountered was much appreciated.

This research work was carried out in the Computer Aided Design Project of the Department of Computer Science, Edinburgh University. The author was financed by a NATO doctoral grant provided by the Scientific and Technical Research Council of Turkey (TUBITAK). The computer facilities were provided as a part of a Science Research Council grant (ref. B/SR/8874).

Appendix A : Modified LOGOS approach.

This section presents a proposal for changing the LOGOS (77) representation system. Figure A.1 shows a different drawing of LOGOS diagrams.

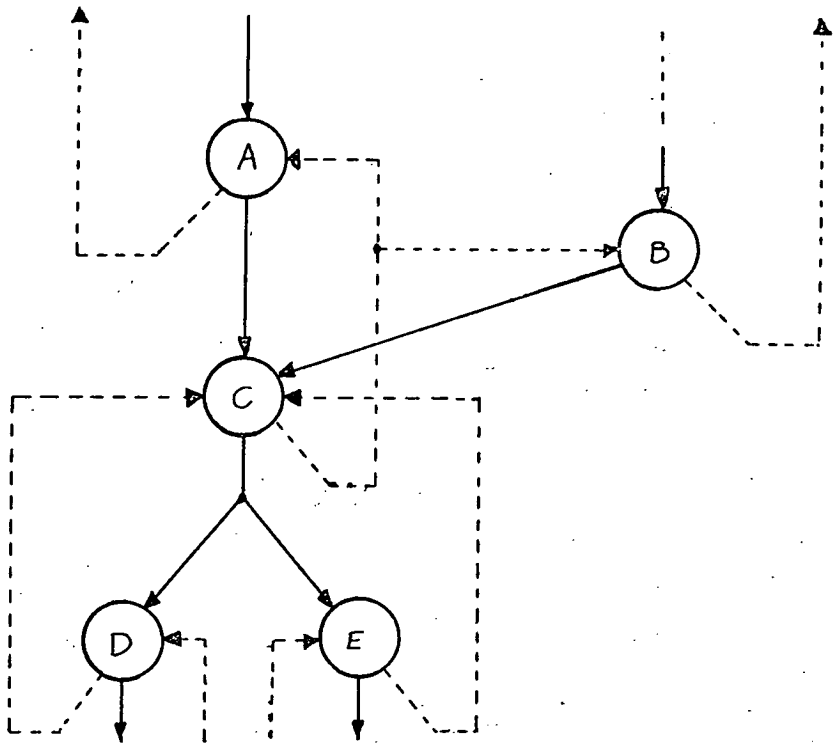


Figure A.1 LOGOS Representation with Visible Upstream Flow.

An operator fires only when all of its inputs are activated and all other operators receiving signal from it, are free. Upon firing, the operator becomes busy until it gets confirmation that it can fire again. Figure A.2 gives the flow-chart of an operator working in that fashion. Table T.1 gives the state-table of a sequential machine which realises the flow-chart and the logic design of the module is given in Figure A.3. The design is race-free and incompatible gates (i.e. gates with different propagation delays) will not cause malfunction of the circuit.

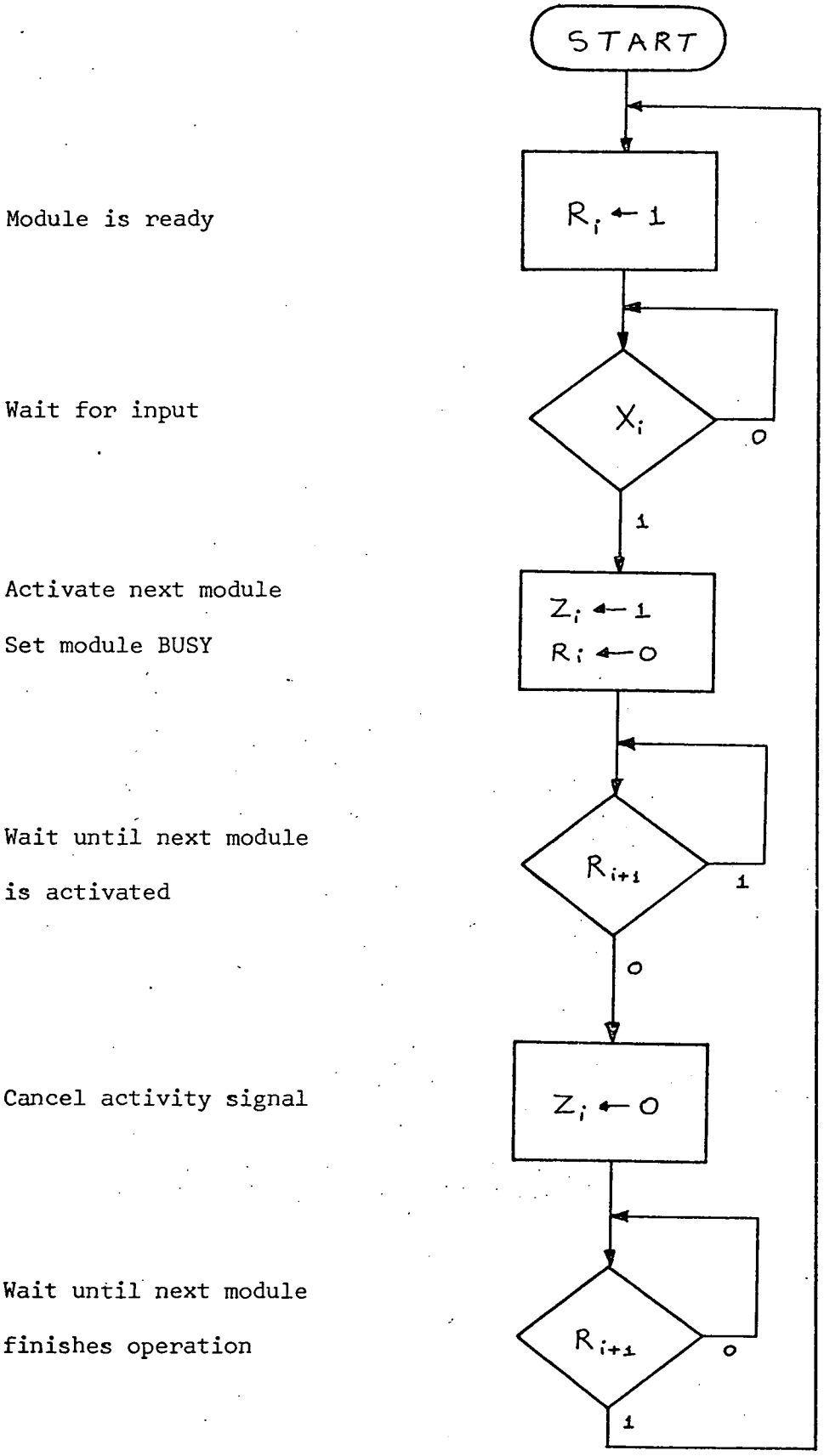


Figure A.2 Flow-Chart of Primitive Operator.

Present State	State Variable		Next-state Inputs (R_{i+1},X_i)				Outputs R_i,Z_i	
	Y_1	Y_2	00	01	11	10		
1	1	0	1	1	4	1	1	0
2	0	1	3	3	2	2	0	1
3	1	1	3	3	1	1	0	0
4	1	0	2	2	2	2	1	1

Table T.1 State-Output Table of Primitive Operator

The excitation and output equations are;

$R_1=X_i \cdot R_{i+1} \cdot Y_2$

$S_1=R_{i+1} \cdot Y_2$

$R_2=R_{i+1} \cdot Y_1$

$S_2=Y_1$

$R_i=\overline{Y_2}$, $\overline{R_i}=Y_2$

$Z_i=\overline{Y_1}$, $\overline{Z_i}=Y_1$

The "X_i" input can be expanded by adding more input terminals to the NAND gate to which it is connected. The "R_{i+1}" input can be expanded in the same way, using separate inverters for each, before connecting to NAND gate 2. The module produces both true and complemented outputs. However an inverter is used to obtain the $\overline{R_{i+1}}$ signal since this avoids running an extra line to each module.

A more complicated operator to activate first a data component and then the next control operator, can be built using the primitive operator described above. Its logic design is given in Figure A.4.

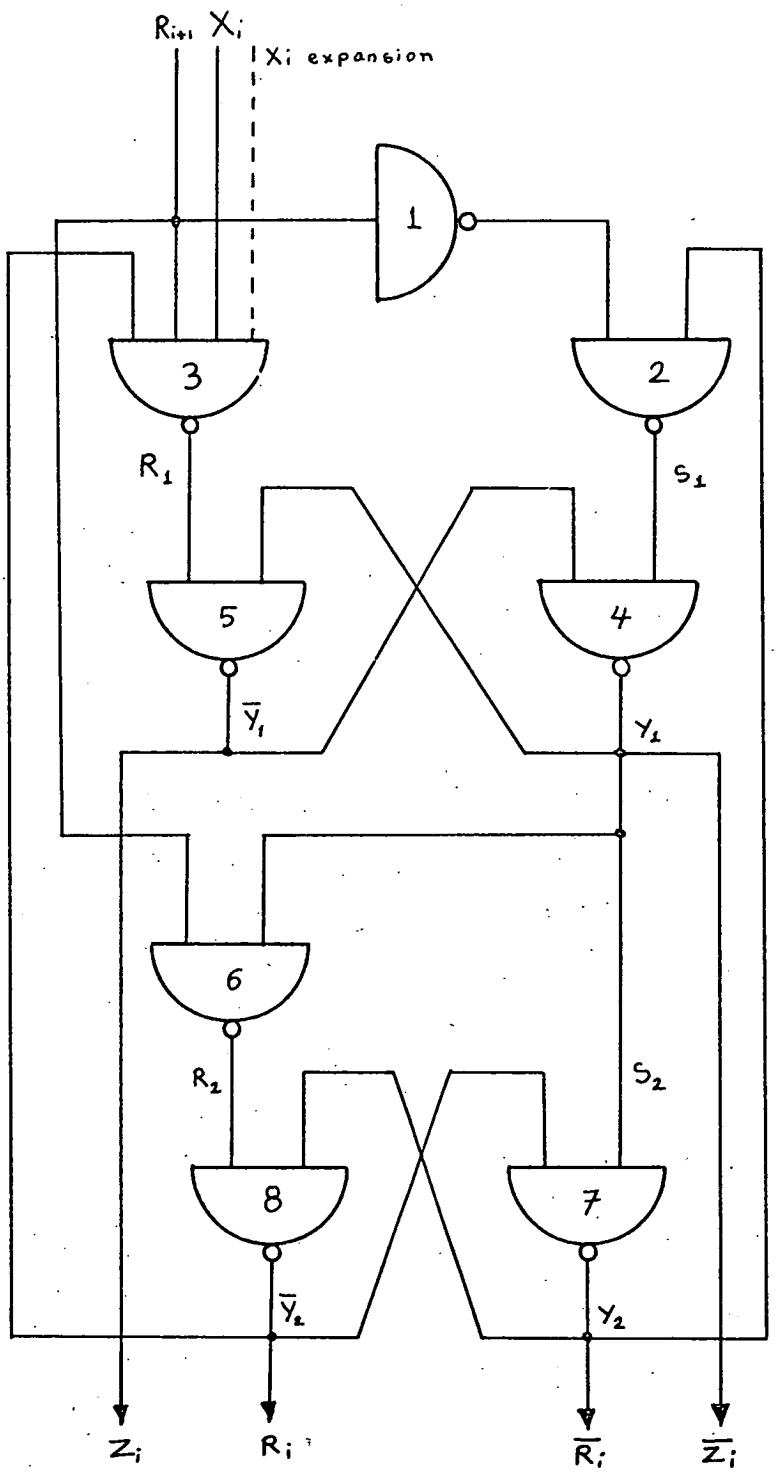
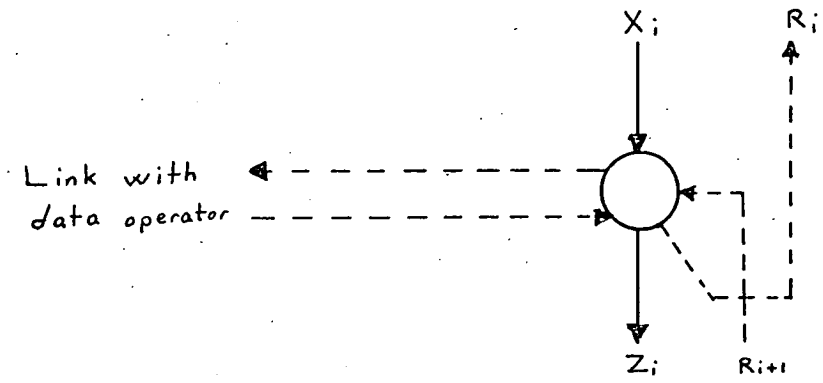
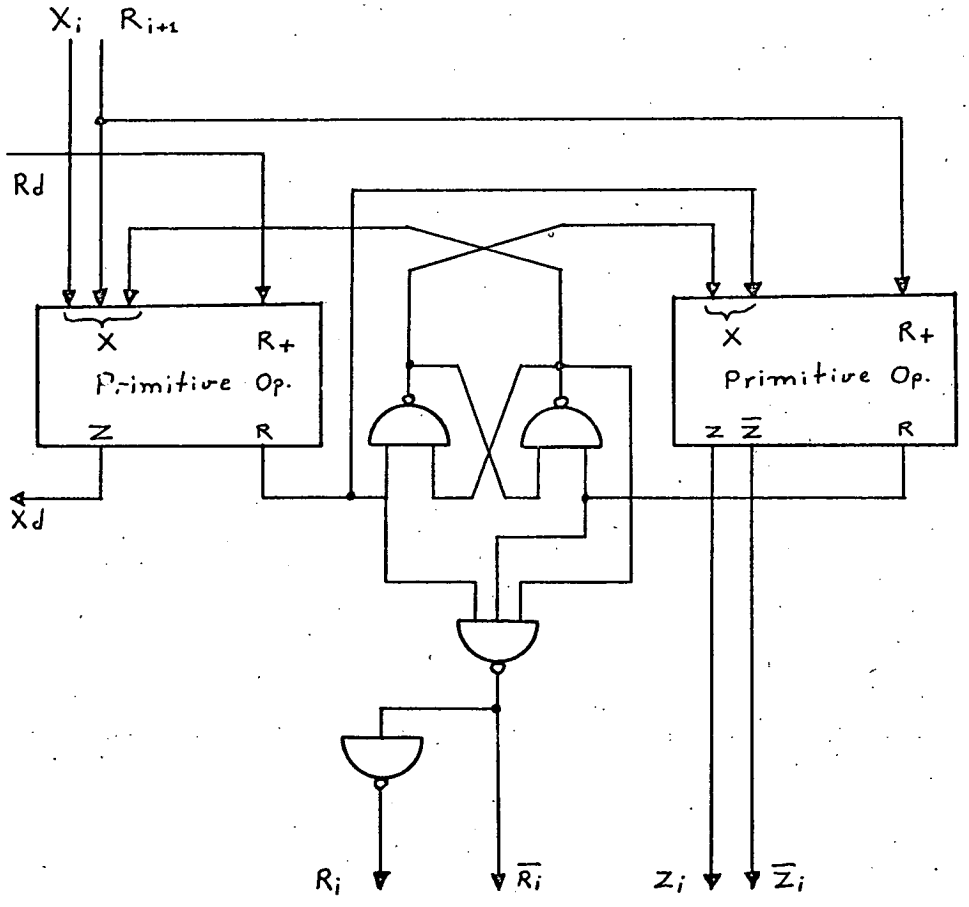


Figure A.3 Logic Design of Primitive Operator



(a) Representation



(b) Logic design

Figure A.4 Design of Macro Operator.

Figure A.5 shows a sequencer which activates more than one data operators, in sequence.

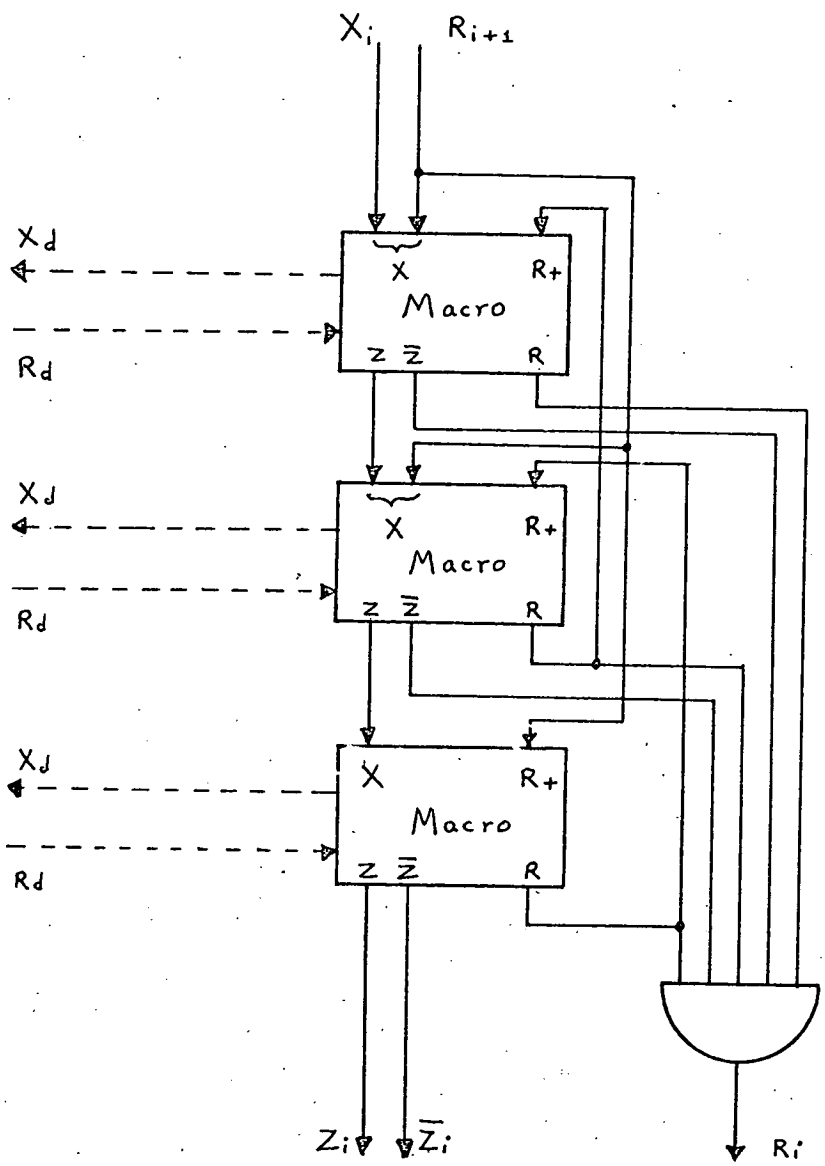


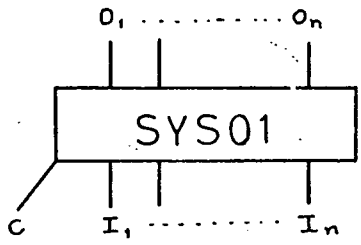
Figure A.5 Design of Sequencer

A very useful property of the modules described above is that they possess identical terminal characteristics.

APPENDIX B: List of library components available from the design programs.

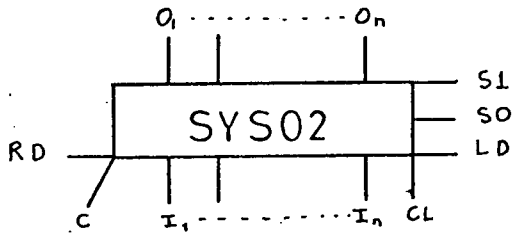
This section lists all the library components available from the design system described in Chapter 4 and describes their terminal behaviour. Library components are given default names from SYS01 to SYS11 but they can be renamed.

1. SYS01 (AND gate)



Realises the function $O_i = C \cdot I_i$

2. SYS02 (Register)



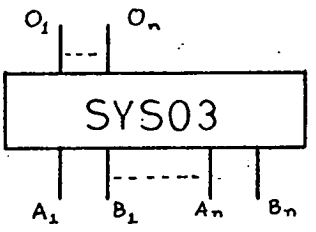
A general purpose register (similar to Texas Inst. SN54194) which is controlled by the mode inputs SO, S1 as shown below (operations are carried out on the falling edge of the clock pulse).

SO	S1	Action
0	0	Ignore clock pulse.
0	1	Shift right (Data on RD replaces data on O_1).
1	1	Accept parallel data.
1	0	Shift left (Data on LD replaces data on O_n).

Table T.2 Operations of the Register

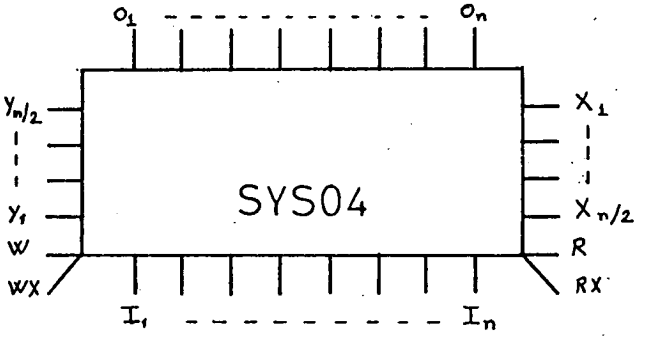
Register contents can be cleared irrespective of the position of the clock pulse by placing a logic value on the CL input.

3. SYS03 (OR gate)



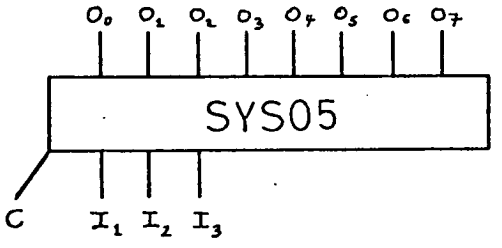
Realises the function $O_i = A_i + B_i$

4. SYS04 (Memory



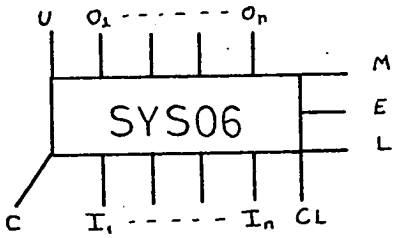
Represents a memory component. I_1 to I_n are the inputs to the memory buffer register and O_1 to O_n are its outputs. X_1 to $X_{n/2}$ and Y_1 to $Y_{n/2}$ are the X and Y address inputs (inputs to the memory address register). To write into the memory, W is set to logic one and a positive pulse (0-1-0) is applied to WX. Memory access time can be specified by the user and W must be kept at logic one until the access is completed. Read operations are enabled in an identical manner, using the R and RX inputs.

5. SYS05 (Decoder)



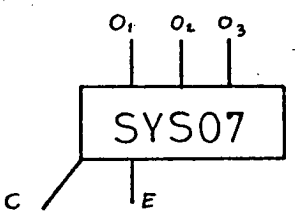
Decodes n inputs into 2^n outputs (in binary order) when C input is at logic one. Outputs become zero when C is returned to zero.

c. SYS06 (Counter)

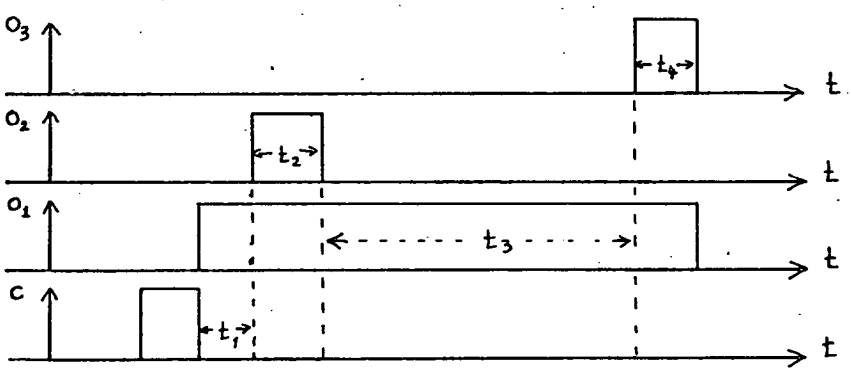


A controllable counter (similar to Texas Inst. SN54193) which can be loaded from the parallel inputs by setting L to zero (asynchronous loading). Counting is enabled when E becomes one and the M input allows counting downwards (M = 0) or upwards (M = 1). U is an overflow bit which is set when the counter becomes zero (if M = 0 and E = 1) or when all bits are set (if M = 1, E = 1).

7. SYS07 (Memory controller)

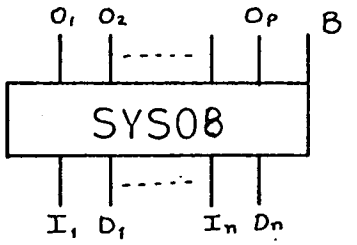


A component which is used to activate the memory operations. It is enabled by setting E to logic zero. Then, a positive pulse applied to the C input produces the following sequence of outputs.



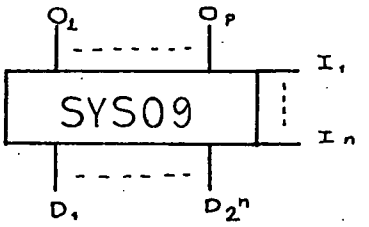
Values t_1, t_2, t_3 and t_4 are specified by the user.

8. SYS08 (Highway)



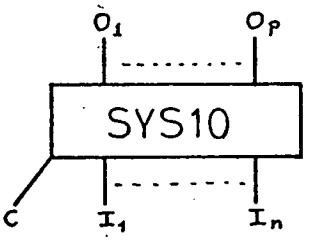
Only multi-line connections (carrying 'p' bits can be made to the D inputs. When I_i becomes one, data on D_i replaces the outputs (O_1 to O_p) and B becomes one. Further requests from other inputs are ignored until I_i is set to zero again. If simultaneous requests are made to get the highway, the leftmost one is answered.

9. SYS09 (Multiplexor)



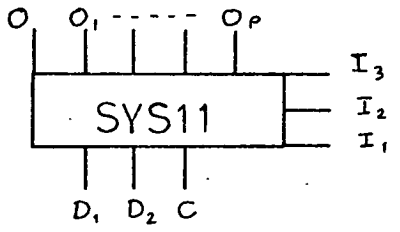
Only multi-line connections (carrying 'p' bits) can be made to the D inputs. Data on the i'th data port (i is the binary number represented by the values on inputs I_1 to I_n) replaces the outputs (O_1 to O_p).

10. SYS10 (Read-only memory)



When C is set to one, the contents of the p-bit word at address i (i is the number represented on the I inputs) replaces the outputs (O_1 to O_p).

11. SYS11 (Arithmetic and logical unit)



Only multi-line connections (carrying 'p' bits) can be made to D_1 or D_2. SYS11 performs arithmetic or logical operations on the input data (D_1 and D_2) according to the table given below.

$I_1I_2I_3$	Operation
0 0 0	$D_2 + D_1$
0 0 1	D_2 minus D_1
0 1 1	minus D_2
0 1 0	D_2 plus D_1
1 1 0	$D_2.D_1$
1 1 1	'NOT' D_2
1 0 1	$D_2 \oplus D_1$
1 0 0	'NOT' $(D_2 + D_1)$

Table T.3 Operations of the SYS11 Component.

C input can be used as the carry (or borrow) input from a previous bit and O generates a carry (or borrow) value for the higher order bits.

APPENDIX C: Features of the supporting computer system

Figure C.1 gives a simplified block diagram of the computer system where only those parts essential to the running of the author's design programs, are shown.

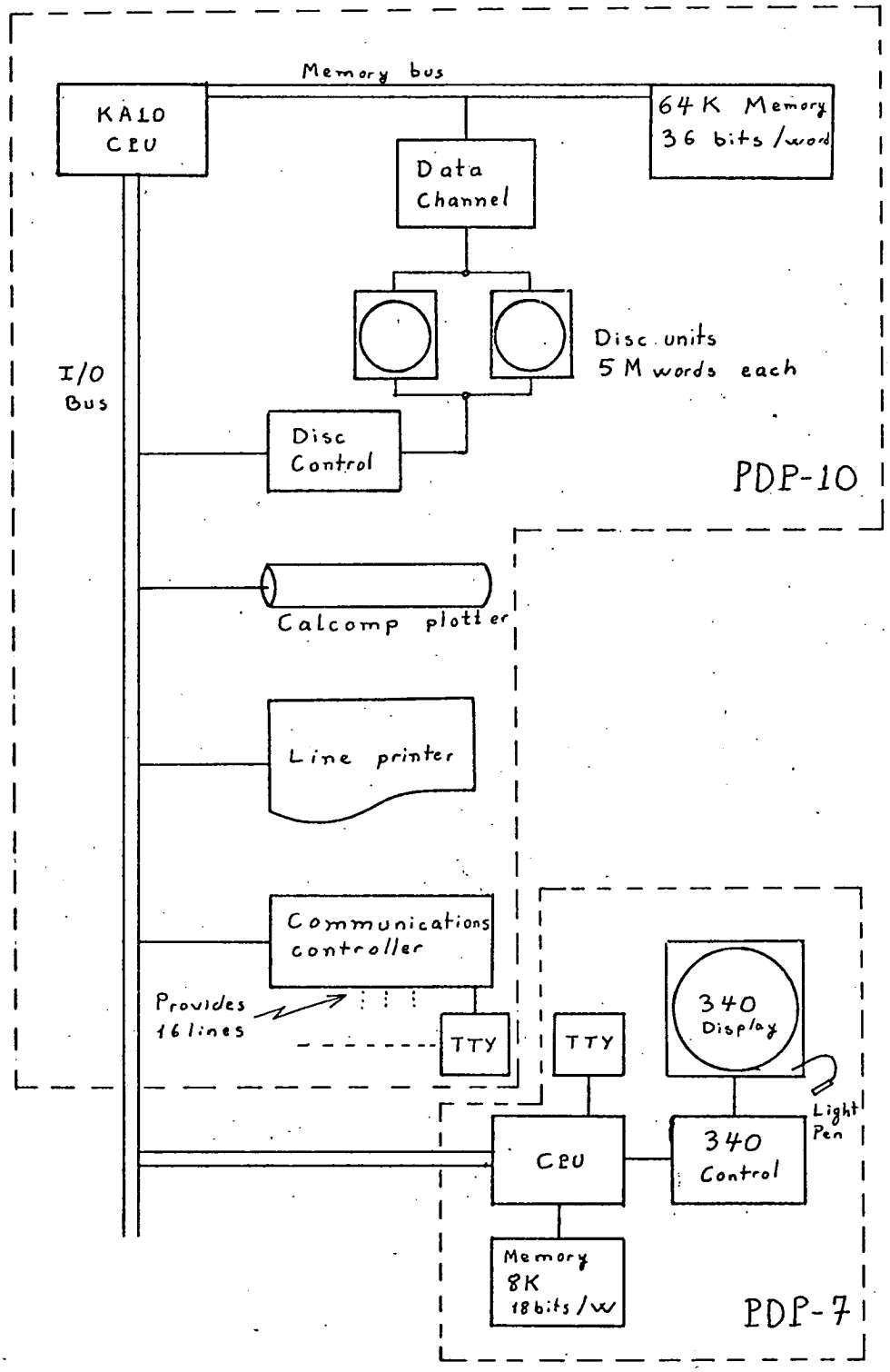


Figure C.1 Computer System.

The central part of the system is PDP-10 computer which is time-shared among several users. A monitor program which resides in core at all times, allocates resources to other programs and dynamically schedules their execution such that each one is given a fixed time slice on the central processor. When a job uses up its time slice or is awaiting the completion of an I/O operation, its execution is stopped and another user's program is started. If all jobs running in the system cannot be kept in core together, as is usually the case, some of them are 'swapped' out onto disc until they get another chance to continue. Communication with the monitor is achieved by using remote terminals (Teletype 33 or Tektronix 4010). Directly linked to the I/O bus is a satellite computer (PDP-7) which forms the interface between the user program and 340 display unit. The user submits his design programs by using one of the terminals which is positioned next to the display screen. The PDP-10 filing system allows each user to have a private disc directory and user programs are normally stored on disc at all times. Therefore submitting a job requires only instructing the monitor accordingly.

Drawings are generated by using special subroutines obtained from the SPINDLE (57) graphics package. These routines enable the user to prepare, modify or examine the contents of a 'display file' which contains instructions for the 340. For example, there are routines which generate instructions to position the beam, draw vectors or characters at a variety of scale and intensity (brightness) settings, enable or disable the light pen, display a tracking cross which can follow the movements of the pen,... etc. The display file is divided into segments and each segment is given a name for identification. An identical copy of this file is sent to the PDP-7

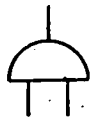
(initially in toto and subsequently incrementally)where it is interpreted and the drawings are displayed. Light-pen interrupts are first analysed by the PDP-7 executive, which sends the name of the segment hit and some other information to the user program. Logic drawings are prepared such that each component, connection or light button is in a separate segment. Hence the user program can identify the particular item selected by examining the name sent from the PDP-7. The tracking cross is used in drawing 'rubber-band' lines for making the connections. The cross is slaved to the light-pen and follows its movements on the screen. When tracking is stopped (by releasing the shutter of the pen), a 'loss of tracking' interrupt is generated and the last recorded position of the cross is sent to the user program. This information is used to calculate the X and Y increments of the line segment to be drawn. The display file is modified accordingly and changes (or additions) are sent to PDP-7.

Hard copies of drawings are obtained by dumping the display file on disc and using another program to generate plotter instructions from it. Outputs from the simulator program are first stored on disc and then listed on the line printer by using standard PDP-10 software.

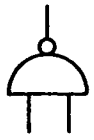
Appendix D: Key to logic diagrams.

The following representations have been used in the logic diagrams.

(i)- AND gate

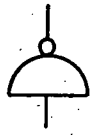


(ii)- NAND gate

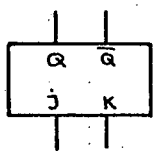


(iii)- OR gate

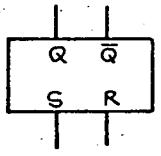
(iv)- INVERTER



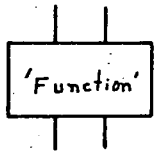
(v) J-K FLIP-FLOP



(vi)- R-S FLIP-FLOP



(vii)- Others



References & Bibliography

(1) H.G. Adshead, et.al.: "New Dimensions in Automatic Logic Testing and Diagnosis." Proc. IEE Conf. on CAD. April 1972. pp. 112-118.

(2) Anon.-: "PDP-16 Computer Designers Handbook". DEC - 1971.

(3) Anon.-: "DISC 8/16 Microminiature Computer". Inertial Systems Dept. Ferranti Ltd. 1972.

(4) Anon.-: "APL/360 Primer". Student text. IBM Corp. 1969.

(5) Anon.-: "SIMBOL. A Primer For The Simulation Language". ICL Ltd. 1968.

(6) Anon.-: "DA70 User Manual". Plessey Company Ltd. 1971.

(7) Anon.-: "REDAP 22 Logic Simulator". REDAC Ltd. 1971.

(8) Anon.-: "Semiconductor Components Data Book Two". TEXAS Instruments. 1971.

(9) Anon.-: "FLOG. Logic Simulation Program". TSL Ltd.

(10) D.B. Armstrong.: "On the Efficient Assignment of Internal Codes to Sequential Machines". IRE Trans. on Elect. Comp. October 1962. pp. 611-622.

(11) M.B. Baray & Y.H. Su.: "A Digital System Modelling Philosophy and Design Language". Proc. of the 8th annual design automation workshop. Atlanta City. 1971.

(12) Bartee, Lebow & Reed.: "Theory and Design of Digital Systems". McGraw-Hill, 1962.

(13) T.C. Bartee.: "Digital Computer Fundamentals". McGraw-Hill, 1960.

(14) L.A. Belady, et.al.: "A Computer Graphics System for Block Diagram Problems". IBM Sys. Journ . Vol.10, No. 2. 1971 pp. 143-163.

(15) C.G. Bell & A. Newell.: "The Description and Use of Register Transfer Modules." IEEE Trans on Comp. Vol C-21, No. 5, May 1972, pp. 495-500.

(16) C.G. Bell, J. Grason, A. Newell.: "Designing Computers and Digital Systems." Digital Press, 1972.

(17) C.G. Bell & A. Newell.: "A Panel Session - Computer Structure - Past, Present and Future". Proc. Fall Joint Comp. Conf. 1971 pp. 387-396 .

(18) C.G. Bell & A. Newell.: "Computer Structures: Readings and Examples." McGraw-Hill, 1971.

(19) Biorci (Ed).: "Network and Switching Theory." Academic Press, 1968.

(20) G. Birtwistle.: "Notes On The SIMULA Language." Norwegian Computing Centre. Publ. No. 5-7, 1969.

(21) A.H. Boyce.: "A Study of Diagnostic Testing of Sequential Circuits." IEE Proc. Conf. on CAD, April 1972. pp. 309-314

(22) A.H. Boyce.: "CLOIS - Computerised Logic Information System." IEE Colloquium, digest No. 1971/19. November 1971. Contribution No. 6.

(23) F.T. Bradshaw.: "Some Structural Ideas for Computer Systems." IEEE Proc. Compcon. September 1972.

(24) J.A. Brozozowski.: "On Single loop realisations of Sequential Machines." Inf. and. Control, Vol. 10, No. 3, 1967, pp. 262-314.

(25) P.W. Case, et.al.: "Solid Logic Design Automation." IBM Journ. of Res. & Dev, Vol. 8 April 1964, pp. 127-140

(26) S.G. Chappel & S.S. Yau.: "Simulation of Large Asynchronous Logic Circuits Using an Ambiguous Gate Model." Proc. Fall Joint Conf., 1971, pp. 651-661.

(27) Y. Chu.: "An ALGOL-Like Computer Design Language." Comm. of the ACM, Vol.8, No. 10, October 1965, pp. 607-615.

(28) M.Cohn & S. Even.: "Identification and Minimisation of Linear Machines." IEEE Trans. on Elect. Comp., Vol EC-14, No. 3, June 1965, pp. 367-376.

(29) E.D. Crocket, et.al.: "Computer Aided System Design", Proc. Fall Joint Conf., 1970, pp. 287-296.

(30) W.A. Davis & J.H. Brzozowski.: "On the Linearity of Sequential Machines." IEEE Trans. on Elect. Comp., Vol. EC-15, No. 1, February 1966, pp. 21-29.

(31) W.A. Davis.: "Sequential Machines Realisable With Delay Elements Only.", IEEE Trans. on Comp. Vol. C-19, No. 4, April, 1970, pp. 353-355.

(32) I.B. Dervisoglu.: "A New Method in State Assignment of Synchronous Machines.", Departmental Memo CAD-I-137, Dept. of Comp. Sci., Edinburgh University, November 1971.

(33) I.B. Dervisoglu.: "Computer Aided Logic Design.", IEE Colloquium, digest No. 1973/5, February 1973, contribution No. 2.

(34) T.A. Dolotta & E.T. McCluskey.: "The Coding of Internal States of Sequential Circuits.", IEEE Trans. on Elect. Comp., Vol EC-13, No. 5, October 1964, pp. 549-562.

(35) J.R. Duley & D.L. Dietmeyer.: "A Digital System Design Language (DDL).", IEEE Trans. on Comp., Vol C-17, No. 9, September 1968, pp. 850-861.

(36) J.R. Duley & D.L. Dietmeyer.: "Translation of a DDL Digital System Specification to Boolean Equations. "IEEE Trans. on Comp., Vol C-18, No. 4, April 1969, pp. 305-313.

(37) A.D. Falkoff, K.E. Iverson, E.H. Sussenguth.: "A Formal Description of System /360.", IBM Sys. Journal, Vol 3, No. 3, 1964, pp. 198-263.

(38) C.C. Foster.: "Computer Architecture.", Van Nostrand Reinhold, 1970.

(39) T.D. Friedman.: "Methods Used in an Automatic Logic Design Generator (ALERT).", IEEE Trans. on Comp., Vol. C-18, No. 7, July 1969, pp. 593-614.

(40) G.B. Gerace.: "Digital System Design Automation - A Method for Designing a Digital System As a Sequential Network System.", IEEE Trans. on Comp., Vol. C-17, No. 11, November 1968, pp. 1044-1067.

(41) E.L. Glaser.: "Introduction and Overview of the LOGOS Project", IEEE Proc. Compcon. September 1972.

(42) D.F. Gorman & J.P. Anderson.: "A Logic Design Translator.", Proc. Fall Joint Comp. Conf., 1962, pp. 251-261.

(43) M.A. Harrison.: "Lectures on Linear Sequential Machines.", Academic Press, 1969.

(44) J. Hartmanis.: "Two Tests for The Linearity of Sequential Machines.", IEEE Trans. on Elect. Comp., Vol. EC-14, No. 6, December 1965, pp. 781-786.

(45) G.G. Hays.: "Computer Aided Design: Simulation of Digital System Logic.", IEEE Trans. on Comp., Vol. C-18, No. 1, June 1969, pp. 1-10.

(46) F.G. Heath.: "Project LOGOS - A Computer Aided Design System For Integrated Software And Hardware.", Proc. IEE Conf. on CAD, April 1972, pp. 225-230.

(47) F.G. Heath & C.W. Rose.: "The Case For Integrated Hardware/ Software With CAD Implications.", IEEE Proc. Compcon. September 1972.

(48) F.G. Heath.: "Digital Computer Design." Oliver & Boyd, 1969.

(49) G.C. Jain & G.F. Adshead.: "Automatic Exhaustive Testing & Diagnosis of Sequential Logic Networks.", ICL Company. Internal Publ.

(50) J.S. Jephson, et.al.: "A Three-Value Computer Design Verification System.", IBM Sys. Journal, Vol, 5, No. 3, 1969, pp. 178-188.

(51) D.L. Johnson & D.H. O'Keefe.: "The Application of Shift Registers To Secondary State Assignment.", IEEE Trans. on Comp., Vol C-17, No. 10, October 1968, pp. 954-977.

(52) H.J. Kahn & J.W.R. May.: "The Use of Logic Simulation in the Design of a Large Computer System.", Dept. of Com. Sci. Manchester University. Internal publ.

(53) A.A. Kaposi & D.R. Holmes.: "Logic Network Analysis.", Computer Aided Design, Vol. 3, No. 1, Autumn 1970, pp. 9-18.

(54) A.A. Kaposi.: "On the Testability of Digital Hardware.", Computer Aided Design, Vol 4, No. 4, 1972, pp. 169-171.

(55) R.M. Karp.:" Some Techniques of State Assignment for Synchronous Sequential Machines.", IEEE Trans. On Elect. Comp. Vol. EC-13, No. 5, October 1964, pp. 507-518.

(56) R.M. Karp & R.E. Miller.: "Parallel Program Schemata", Journ. of Comp. and Syst. Sci. No. 3, 1969, pp. 147-195.

(57) A.C. Kilgour.: "SPINDLE Users Manual", CAD. Proj., Dept. of Comp. Sci, Edinburgh Univ., 1973.

(58) D.E. Knuth & J.E. McNeley.: "SOL - A Symbolic Language For General Purpose System Simulation.", IEEE Trans. on Elect. Comp., Vol. EC-13, No. 4, August 1964, pp. 401-408.

(59) D.E. Knuth & J.E. McNeley.: "A Formal Definition of SOL.", IEEE Trans. on Elect. Comp., Vol-EC-13 No. 4 , August 1964, pp. 409-414.

(60) Z. Kohavi.: "Switching And Finite Automata Theory.", McGraw-Hill, 1970.

(61) F. Leraillez, et.al.:"CRISMASS: a Tool for Conception, Realisation, Implementation and Simulation of Sequential Synchronous Machines.", Proc. IEE Conf. on CAD, April 1969, pp. 59-67.

(62) D.W. Lewin.: "Problem Specification for Computer Assisted Logic System Design.", IEE Colloquium, digest no. 1971/19, November 1971, contribution no. 8.

(63) D.W. Lewin & M.C. Waters.: "Computer Aids to Logic System Design.", Comp. Bulletin, Vol 13, No. 11, November 1969, pp. 382-388.

(64) D.W. Lewin, E.J. Purslow, R.G. Bennets.: "Computer Assisted Logic Design - the CALD System.", Proc. IEE Conf. on CAD, April 1972, pp. 343-351.

(65) D.W. Lewin.: "Logical Design of Switching Circuits.", Nelson, 1970.

(66) D.W. Lewin.: "Theory and Design of Digital Computers.", Nelson, 1972.

(67) J. Martin.: "Design of Man-Computer Dialogues.", Prentice-Hall, 1973.

(68) W.C. McGee & H.E. Peterson.: "Micro programming Control for the Experimental Sciences.", Proc. Fall Joint Conf., 1965, pp. 77-91.

(69) R.B. Miller.: "Response Time in Man-Computer Conversational Transactions.", Proc. Fall Joint Conf., 1968, pp. 267-277.

(70) P. Naur (Ed).: "Revised Report on the Algorithmic Language ALGOL-60.", Comm. ACM, Vol. 6, No. 1, Jan 1963, pp. 1-17.

(71) J.V. Oldfield.: "Interactive Techniques.", Application Course Lectures, Proc. IEE Conf. on CAD, April 1972.

(72) C.A. Petri.: "Communication With Automata." Rome Air Development Center, Tech. Rept. RADC-TR-65-377, 1966.

(73) M.S. Pliner & C.W. Rose.: "A Primitive Data Base Management For An Integrated Computer Aided Design Facility.", IEEE Proc. Compcon. September 1972.

(74) R.M. Proctor.: "A Logic Design Translator Experiment Demonstrating Relationships of Language To Systems And Logic Design.", IEEE Trans. on Elect. Comp. Vol EC-13, No. 4, April, 1964, pp. 422-430.

(75) C.A. Rey.: "Control Point Design Using Modular Logic.", Tech. Rept. Dept. of Comp. Sci. Univ. of Illinois, 1971.

(76) J.S. Reynolds.: "A Conversational Logic Simulator For Use With a Time-Sharing Computer.", Proc. IEE Conf. on CAD., 1969, pp. 608-615.

(77) C. W. Rose & F.T. Bradshaw.: "The LOGOS Representation System.", IEEE Proc. Compcon. September 1972.

(78) J.P. Roth.: "Systematic Design of Automata.", Proc. Fall Joint Comp. Conf. 1965, pp. 1093-1100.

(79) H.P. Schlaeppi.: "A Formal Language For Describing Machine Logic, Timing And Sequencing (LOTIS). ", IEEE Trans. on Elect. Comp. Vol. EC-13, No. 4, August 1964, pp. 439-448.

(80) H. Schorr.: "Computer Aided Digital System Design And Analysis Using A Register Transfer Language.", IEEE Trans. on Elect. Comp., Vol. EC-13, No. 6, December 1964, pp. 730-737.

(81) E.P. Stabler.: "System Description Languages.", IEEE Trans. on Comp., Vol C-19, No. 12, December 1970, pp. 1160-1173.

(82) R.E. Stearns & J. Hartmanis.: "On The State Assignment Problem For Sequential Machines II. ", IRE Trans. on Elect. Comp., December 1961, pp. 593-603.

(83) F. Stevenson.: "An Introduction to LOGIC., A Computer Program For Simulation of Digital Logic Networks.", An extract from the LOGIC users manual, Norwegian Computing Centre, 1968.

(84) J.R. Story, et.al.: "Optimum State Assignment For Synchronous Sequential Circuits.", IEEE Trans. on Comp., Vol C-21, no. 12, December 1972, pp. 1365-1373.

(85) I.E. Sutherland.: "Sketchpad. A Man-Machine Graphical Communication System.", MIT Lincoln Lab., Tech. Rept. No. 296, 1963.

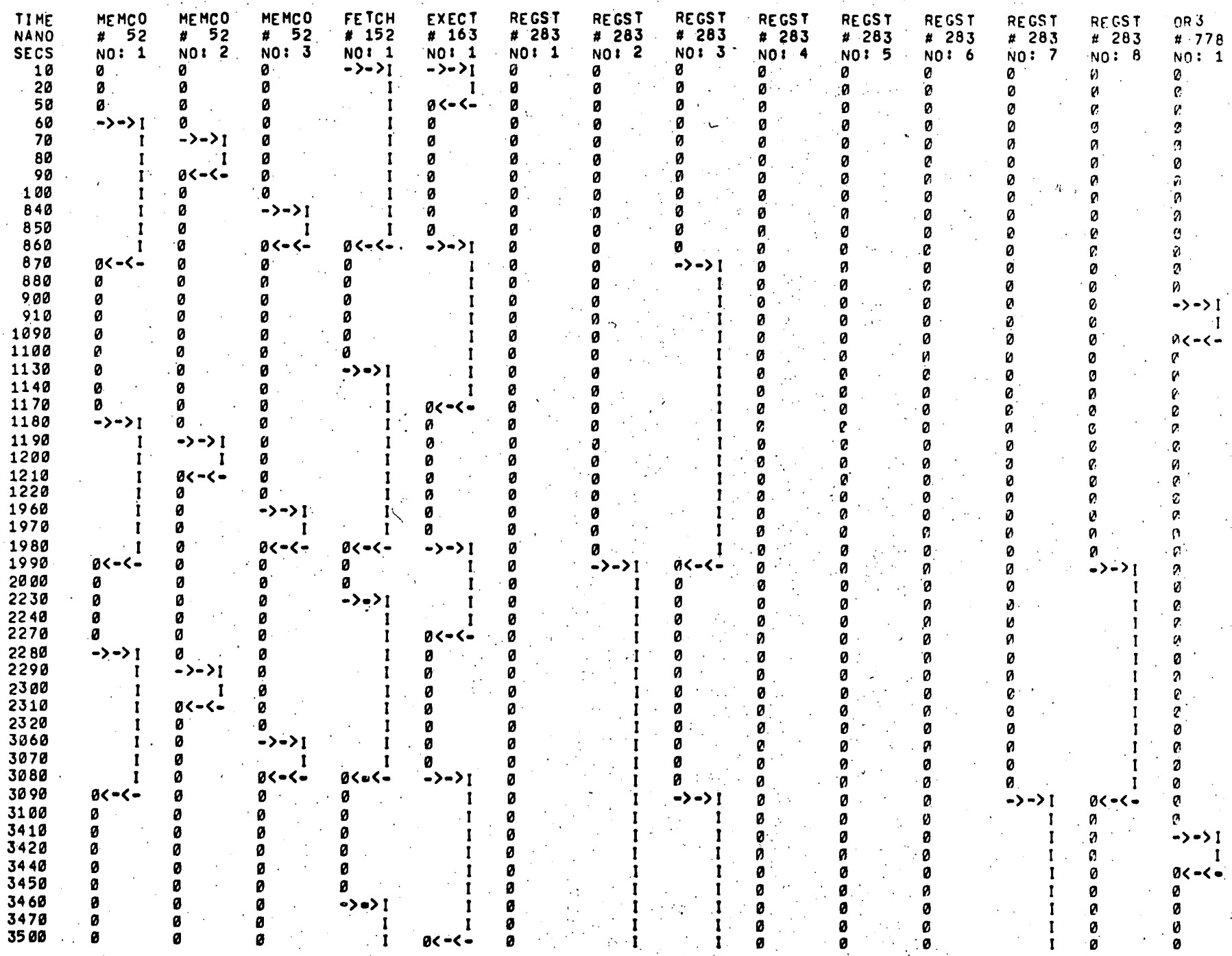
(86) H.C. Torng.: "An Algorithm For Finding Secondary Assignments of Synchronous Sequential Circuits.", IEEE Trans. on Comp. Vol C-17 No. 5, May 1968, pp. 461-469.

(87) M. Uzsoky, et.al.: "Computer Aided Design of Digital Systems in Hungary.", Proc. IEE Conf. on CAD, April, 1972, pp. 359-365.

(88) M.V. Wilkes & J.B. Stringer.: "Micro-programming And The Design of The Control Circuits In An Electronic Digital Computer.", Proc. Camb. Phil. Soc., Vol 44, Pt. 2, April 1953, pp. 230-238.

(89) N.E. Wiseman, et.al.: " PIXIE - A New Approach To Graphical Man Machine Communications.", Proc. IEE Conf. on CAD, 1969, pp. 463-471.

(90) P.E. Wood Jr.: "Switching Theory.", McGraw-Hill, 1968 .


TIME NANO SECS
MEMCO # 52 NO: 1
MEMCO # 52 NO: 2
MEMCO # 52 NO: 3
FETCH # 152 NO: 1
EXECT # 163 NO: 1
REGST # 283 NO: 1
REGST # 283 NO: 2
REGST # 283 NO: 3
REGST # 283 NO: 4
REGST # 283 NO: 5
REGST # 283 NO: 6
REGST # 283 NO: 7
REGST # 283 NO: 8
OR3 # 778 NO: 1
10
20
50
60
70
80
90
100
840
850
860
870
880
900
910
1090
1100
1130
1140
1170
1180
1190
1200
1210
1220
1960
1970
1980
1990
2000
2230
2240
2270
2280
2290
2300
2310
2320
3060
3070
3080
3090
3100
3410
3420
3440
3450
3460
3470
3500

TIME NANO SECS	MEMCO # 52 NO: 1	MEMCO # 52 NO: 2	MEMCO # 52 NO: 3	FETCH # 152 NO: 1	EXECT # 163 NO: 1	REGST # 283 NO: 1	REGST # 283 NO: 2	REGST # 283 NO: 3	REGST # 283 NO: 4	REGST # 283 NO: 5	REGST # 283 NO: 6	REGST # 283 NO: 7	REGST # 283 NO: 8	OR3 # 778 NO: 1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	->->I	->->I	0	0	0	0	0	0	0	0	0
20	0	0	0	I	I	0	0	0	0	0	0	0	0	0
30	0	0	0	I	I	0	0	0	0	0	0	0	0	0
40	0	0	0	I	I	0	0	0	0	0	0	0	0	0
50	0	0	0	I	0<-<-	0	0	0	0	0	0	0	0	0
60	->->I	0	0	I	0	0	0	0	0	0	0	0	0	0
70	I	->->I	0	I	0	0	0	0	0	0	0	0	0	0
80	I	I	0	I	0	0	0	0	0	0	0	0	0	0
90	I	0<-<-	0	I	0	0	0	0	0	0	0	0	0	0
100	I	0	0	I	0	0	0	0	0	0	0	0	0	0
110	I	0	0	I	0	0	0	0	0	0	0	0	0	0
120	I	0	0	I	0	0	0	0	0	0	0	0	0	0
130	I	0	0	I	0	0	0	0	0	0	0	0	0	0
140	I	0	0	I	0	0	0	0	0	0	0	0	0	0
150	I	0	0	I	0	0	0	0	0	0	0	0	0	0
160	I	0	0	I	0	0	0	0	0	0	0	0	0	0
170	I	0	0	I	0	0	0	0	0	0	0	0	0	0
180	I	0	0	I	0	0	0	0	0	0	0	0	0	0
190	I	0	0	I	0	0	0	0	0	0	0	0	0	0
200	I	0	0	I	0	0	0	0	0	0	0	0	0	0
210	I	0	0	I	0	0	0	0	0	0	0	0	0	0
220	I	0	0	I	0	0	0	0	0	0	0	0	0	0
230	I	0	0	I	0	0	0	0	0	0	0	0	0	0
240	I	0	0	I	0	0	0	0	0	0	0	0	0	0
250	I	0	0	I	0	0	0	0	0	0	0	0	0	0
260	I	0	0	I	0	0	0	0	0	0	0	0	0	0
270	I	0	0	I	0	0	0	0	0	0	0	0	0	0
280	I	0	0	I	0	0	0	0	0	0	0	0	0	0
290	I	0	0	I	0	0	0	0	0	0	0	0	0	0
300	I	0	0	I	0	0	0	0	0	0	0	0	0	0
310	I	0	0	I	0	0	0	0	0	0	0	0	0	0
320	I	0	0	I	0	0	0	0	0	0	0	0	0	0
330	I	0	0	I	0	0	0	0	0	0	0	0	0	0
340	I	0	0	I	0	0	0	0	0	0	0	0	0	0
350	I	0	0	I	0	0	0	0	0	0	0	0	0	0
360	I	0	0	I	0	0	0	0	0	0	0	0	0	0
370	I	0	0	I	0	0	0	0	0	0	0	0	0	0
380	I	0	0	I	0	0	0	0	0	0	0	0	0	0
390	I	0	0	I	0	0	0	0	0	0	0	0	0	0
400	I	0	0	I	0	0	0	0	0	0	0	0	0	0
410	I	0	0	I	0	0	0	0	0	0	0	0	0	0
420	I	0	0	I	0	0	0	0	0	0	0	0	0	0
430	I	0	0	I	0	0	0	0	0	0	0	0	0	0
440	I	0	0	I	0	0	0	0	0	0	0	0	0	0
450	I	0	0	I	0	0	0	0	0	0	0	0	0	0
460	I	0	0	I	0	0	0	0	0	0	0	0	0	0
470	I	0	0	I	0	0	0	0	0	0	0	0	0	0
480	I	0	0	I	0	0	0	0	0	0	0	0	0	0
490	I	0	0	I	0	0	0	0	0	0	0	0	0	0
500	I	0	0	I	0	0	0	0	0	0	0	0	0	0
510	I	0	0	I	0	0	0	0	0	0	0	0	0	0
520	I	0	0	I	0	0	0	0	0	0	0	0	0	0
530	I	0	0	I	0	0	0	0	0	0	0	0	0	0
540	I	0	0	I	0	0	0	0	0	0	0	0	0	0
550	I	0	0	I	0	0	0	0	0	0	0	0	0	0

```
  570         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  580         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  590         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  600         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  610         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  620         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  630         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  640         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  650         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  660         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  670         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  680         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  690         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  700         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  710         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  720         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  730         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  740         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  750         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  760         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  770         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  780         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  790         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  800         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  810         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  820         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  830         I     0         0             I     0         0         0         0         0         0         0         0         0         0
  840         I     0         ->->I         I     0         0         0         0         0         0         0         0         0         0
  850         I     0             I         I     0         0         0         0         0         0         0         0         0         0
  860         I     0         0<-<-     0<-<-     ->->I     0         0         0         0         0         0         0         0         0
  870     0<-<-     0         0         0             I     0         0         ->->I     0         0         0         0         0         0
  880     0         0         0         0             I     0         0             I     0         0         0         0         0         0
  890     0         0         0         0             I     0         0             I     0         0         0         0         0         0
  900     0         0         0         0             I     0         0             I     0         0         0         0         0         ->->I
  910     0         0         0         0             I     0         0             I     0         0         0         0         0             I
  920     0         0         0         0             I     0         0             I     0         0         0         0         0             I
  930     0         0         0         0             I     0         0             I     0         0         0         0         0             I
  940     0         0         0         0             I     0         0             I     0         0         0         0         0             I
  950     0         0         0         0             I     0         0             I     0         0         0         0         0             I
  960     0         0         0         0             I     0         0             I     0         0         0         0         0             I
  970     0         0         0         0             I     0         0             I     0         0         0         0         0             I
  980     0         0         0         0             I     0         0             I     0         0         0         0         0             I
  990     0         0         0         0             I     0         0             I     0         0         0         0         0             I
 1000     0         0         0         0             I     0         0             I     0         0         0         0         0             I
 1010     0         0         0         0             I     0         0             I     0         0         0         0         0             I
 1020     0         0         0         0             I     0         0             I     0         0         0         0         0             I
 1030     0         0         0         0             I     0         0             I     0         0         0         0         0             I
 1040     0         0         0         0             I     0         0             I     0         0         0         0         0             I
 1050     0         0         0         0             I     0         0             I     0         0         0         0         0             I
 1060     0         0         0         0             I     0         0             I     0         0         0         0         0             I
 1070     0         0         0         0             I     0         0             I     0         0         0         0         0             I
 1080     0         0         0         0             I     0         0             I     0         0         0         0         0             I
 1090     0         0         0         0             I     0         0             I     0         0         0         0         0             I
 1100     0         0         0         0             I     0         0             I     0         0         0         0         0         0<-<-
 1110     0         0         0         0             I     0         0             I     0         0         0         0         0         0
 1120     0         0         0         0             I     0         0             I     0         0         0         0         0         0
 1130     0         0         0         ->->I         I     0         0             I     0         0         0         0         0         0
 1140     0         0         0             I         I     0         0             I     0         0         0         0         0         0
 1150     0         0         0             I         I     0         0             I     0         0         0         0         0         0
 1160     0         0         0             I         I     0         0             I     0         0         0         0         0         0
 1170     0         0         0             I     0<-<-     0         0             I     0         0         0         0         0         0
 1180     ->->I     0         0             I     0         0         0             I     0         0         0         0         0         0
 1190         I     ->->I     0             I     0         0         0             I     0         0         0         0         0         0
 1200         I         I     0             I     0         0         0             I     0         0         0         0         0         0
 1210         I     0<-<-     0             I     0         0         0             I     0         0         0         0         0         0
```