A GENERALIZED MAPPING DEVICE AND ITS APPLICATION IN HIGH SPEED DIGITAL COMPUTERS.

by

R.J. Potter, B.E.(Hons).

A thesis submitted for the Degree of Doctor of Philosophy, in the Department of Electrical Engineering of the University of Adelaide, August, 1967.

I hereby state that, to the best of my knowledge, this thesis contains no material previously published or written by another person, except when due reference is made in the text of the thesis.

Acknowledgement.

The author wishes to acknowledge the assistance and support of his supervisors, Prof. M.W. Allen and Prof. J.L. Woodward for suggestions and criticism during the course of the project. He also wishes to thank the staff of the Digital Techniques Laboratory for their assistance in the construction of equipment.

The author is indebted to the Australian Atomic Energy Commission and the Commonwealth Scientific and Industrial Research Organization for financial help in the early stages of his candidature.

Summary.

This thesis discusses methods for achieving flexible internal micro-structures for high speed digital computers. In addition to the well known technique of micro-programming, a flexible register interconnection system, called a mapping unit, is proposed. Logical and hardware forms of the mapping unit are described.

Methods for inclusion and applications of the mapping unit in micro-programmed structures are studied. The effect of the mapping unit upon machine operations involving standard and non-standard data formats is discussed, with appropriate examples. The field of non-standard machine instructions is discussed and extended to include problems met in providing compatible operation between two dissimiliar computers. Some problems and solutions in providing machine language compatibility via micro-programmed simulators are discussed.

The examples given utilize CIRRUS, a micro-programmed computer available for research purposes within the University of Adelaide.

TABLE OF CONTENTS.

Page No.

1. Introduction. 1
2. Structural Flexibility in Computers. 3
   - 2.1. Micro-Programming. 3
   - 2.2. Machine Instructions in Micro-Programmed Computers. 4
   - 2.3. Analysis of Micro-Operations. 5
   - 2.4. The Mapping Unit as a Re-formatting Device. 6
3. A Description of a Mapping Unit. 8
   - 3.1. Logical Form of a Mapping Unit. 8
   - 3.2. Extensions to the Logical Operation of the Mapping Unit. 15
   - 3.3. Implementation of the Mapping Unit. 18
     - 3.3.1. System Design. 18
     - 3.3.2. Hardware Details . 21
4. The Inclusion of a Mapping Unit in a Computing Structure. 33
   - 4.1. External Characteristics. 33
   - 4.2. The Inclusion of a Mapping Unit in CIRRUS. 33
     - 4.2.1. The CIRRUS Computer. 33
     - 4.2.2. Provision of a Mapping Unit in CIRRUS. 35
5. Mapping Units and Machine Instructions. 49
   - 5.1. Machine Language Instructions and Micro-Programming. 49
   - 5.2. Arithmetic Operations 49
   - 5.3. Shifting Operations. 60
   - 5.4. Branching Operations. 60
   - 5.5. Bit and Character Handling Operations. 61
   - 5.6. Masking and Shifting Instructions. 61

TABLE OF CONTENTS. (contd.)

Page No.

6. Mapping Units and the Provision of Non-Standard Data Forms and Operations. 66
   - 6.1. Standard Data Forms. 66
   - 6.2. Interpretive Programming. 66
   - 6.3. Provision of Alternate Formats. 67
   - 6.4. Provision of Non-Standard Operations. 69
   - 6.5. An example- Quarter-Word Operation in CIRRUS. 70

7. Inter-Computer Compatibility. 75
   - 7.1. Requirements and Definitions. 75
   - 7.2. Compatibility via High Level Languages. 77
   - 7.3. Machine Language Compatibility. 78
     - 7.3.1. Translation of Machine Instructions. 78
     - 7.3.2. Simulation of the Source System. 79
   - 7.4. Micro-programmed Simulators. 82
     - 7.4.1. Depth of Simulation. 82
     - 7.4.2. Internal Formats in a Micro-programmed Simulator. 83
     - 7.4.3. External Formats in a Micro-programmed Simulator. 90
   - 7.5. Compatibility - An Example. 94
     - 7.5.1. The Source Computer. 94
     - 7.5.2. Memory Simulation. 95
     - 7.5.3. Program Control. 96
     - 7.5.4. Exceptions and Interrupts. 96
     - 7.5.5. The Micro-programmed Simulator. 96
   - 7.6. Limitations of Micro-programmed Simulators. 98

8. Conclusion. 100

TABLE OF CONTENTS. (contd.)

	Page No.
Appendices	103
Appendix 1.	103
Appendix 2.	119
Appendix 3.	125
Appendix 4.	137
Appendix 5.	139
Appendix 6.	145
Appendix 7.	147
Appendix 8.	150
References	158

SECTION 1.

Introduction.

1. Introduction.

This thesis is the result of a project designed to investigate various techniques for providing flexible digital computer structures which might be simply controlled. Prior to the commencement of the project, a digital computer CIRRUS [1] had been designed and constructed within the Department of Electrical Engineering, University of Adelaide. This computer offered a number of novel solutions to problems of low cost, high speed design of computers and featured a flexible micro-programmed structure under the control of a fixed or read only store [2] holding the micro-instructions. The hardware form of the computer had been chosen to provide efficient micro-code interpretation of a wide variety of machine language data formats and instruction types. In addition, a multi-programming feature formed an integral part of the design of the computer, [3] so that a comprehensive interrupt system, together with a micro-programmed interrupt processing and program switching routine were provided.

This project is concerned with re-formatting of information within any computer's micro-structure, in order that non-standard data formats and instructions may be provided for use at the machine language level. This re-formatting technique is difficult to provide within existing structures and Sections 3 and 4 of this thesis deal with the design of a re-formatting or mapping unit and its inclusion in CIRRUS, Unfortunately, due to lack of funds, a full size unit has not been built, but simulation of its effects upon the computer has been carried out. Sections 5 and 6 detail the effect that the mapping unit has upon some computer operations, whilst Section 7 discusses at

some length the problem of providing machine language compatibility between two dissimilar computers. The application of the mapping unit to such a problem is studied. Other problems arising in the provision of compatibility via micro-programmed simulators are also explored and some solutions to these problems are presented.

All examples up to and including Section 6 are illustrated on the CIRRUS computer. Section 7 uses both a commercially available computer, the International Business Machine Corp. System/360 and the CIRRUS computer, to illustrate some of the solutions to compatibility problems.

The overall treatment of most topics is descriptive rather than analytical or programmatic in order to retain simplicity and clarity, and limit the length of the thesis.

Papers arising from this project that have been published are:-

Some Problems in the Design of Compatible Computers, Proceedings of the 3rd Australian Computer Conference, Canberra, May, 1966, pp 7/2/1.

With M.W. Allen. A Hardware Device for Generalized Mapping Functions, IEEE Transactions on Electronic Computers. Vol EC-15 No.1, Feb, 1966, p 118.

SECTION 2.

Structural Flexibility in Computers.

2.1. Micro-Programming.

The technique of micro-programming was first described by Wilkes in 1953 [20]. Essentially a micro-program consists of a time ordered series of elementary operations (micro-instructions) within the hardware structure of a computer, e.g. register to register transfers, specification of adder inputs, etc. Implicit in a micro-instruction is the name of the next micro-instruction to be executed. This "next micro-address" facility can be made dependent upon one or more pieces of information within the controlled structure to yield a conditional branching micro-instruction. A micro-structure and an associated micro-program can be regarded as a "sub-computer" and each machine language instruction of the computer is interpreted as a series of micro-instructions within this sub-computer [16].

In the past decade, advances in the design of fixed or read-only storage have allowed the micro-program concept to be directly implemented in the control unit of a computer. Micro-programs related directly to the controlled structure are held in successive locations of a store (generally a fixed store, since this form of storage offers significant advantages in cost/bit/cycle-time over other forms of store) and micro-instructions are read sequentially from this store into a control element. This control element interprets the micro-instructions and provides the specified paths and timing signals within the remainder of the computer. An integral part of the system , which may be under micro-instruction control, selects the address in fixed store of the next micro-instruction to be executed. When execution of the

current micro-instruction is complete, the fixed store is re-driven and the cyclic process of micro-instruction extraction, execution and next address determination continues.

For reasons of speed, many faster computers dispense with this form of control. However, the conventional control sequence generators consisting of flip flops, delays, gates etc., which they use, may be regarded as logically equivalent to a set of micro-programs. All conclusions drawn in respect of micro-program controlled computers can be applied directly or indirectly to computers controlled by sequence generators.

2.2. Machine Instructions in Micro-Programmed Computers.

Since any machine instruction is interpreted as a series of micro-instructions in a micro-programmed computer, the set of micro-instructions provided in any computer must be the set of micro-instructions necessary for performing all instructions in the machine language instruction repertoire. A small number of different micro-operations is generally sufficient to implement a large range of machine language instructions of the same general type, e.g. most arithmetic operations can be resolved as a series of data transfers, sign reversals and/or additions.

There are generally a number of alternate ways tomicro-program any machine instruction. It is also possible to micro-program any process which may be required for only one or two machine language instructions and has not been provided in the hardware for reasons of economy.

The number of types of data that any computer operates upon has increased as programming research

reveals new, efficient representation for data. Early computers were provided only with fixed point arithmetic at machine language level. Operations on floating point formats and other data forms had to be interpretively programmed at machine language level. Then floating point arithmetic hardware and suitable machine language instructions were provided to improve computing efficiency. Then computer languages created a need for character handling instructions, to assist in translation of these languages to the machine code. At the present time, graphical structures are being investigated in some detail. To improve the operating efficiency of any computer, the provision of large numbers of machine instructions for each of these data forms is obviously desirable. Undoubtedly, future computer designs will provide a wide range of data formats and machine operations for manipulating these formats. The use of fixed storage techniques in control units, allows economical provision of large numbers of different machine instructions.

2.3 Analysis of Micro-Operations.

Micro-operations fall into three main classes:-

(a) Inter-register transfer operations,

(b) Manipulative operations,

and (c) Conditional micro-branching operations.

Considering only classes (a) and (b), it may be noted that these generally operate upon data in a hardware format. This hardware format need not necessarily be directly related to the machine language data format that the micro-program is operating upon e.g. floating point forms generally pack fixed point integral exponents and fixed point fractional mantissas into one word, whereas the computer hardware operates

only upon separate integers and fractions. Presently, transformation of formats may occur in two ways:-

(1) Sequences of micro-operations may be set up using logical operations (AND,OR and shifting operations) to isolate or assemble specified data formats.

(2) Special purpose transfer paths may be provided in the set of allowable inter-register transfers to modify data between specified formats.

In either case,the introduction of extra data formats involves adding either more special purpose micro-programs, which will slow the operation of the computer as a whole, or more special purpose inter-register transfer paths, which will complicate the hardware and require extra facilities for their control.

In the next section, we present a device for providing flexible re-formatting operations at high speed. If such a unit is provided in a computer structure, the provision of new data formats and operations upon these formats becomes straight-forward. The reduction of new machine language data formats to existing operating formats enables existing micro-instructions to be used in providing micro-programs for the new machine operations. Existing data formats and operations will be unaffected.

2.4. The Mapping Unit as a Re-formatting Device.

The re-formatting device is derived from a proposal by Mercer [16] for a variable interconnection path between two registers in a computer, which he called a "T-matrix". In the following sections, we present a mechanization of this device, which we will call a mapping unit, and some studies of its

application within a micro-programmed computer. The availability of such a device extends the effectiveness of the micro-programming technique by removing the barrier of fixed register interconnections, and substituting a micro-program specified transfer operation.

SECTION 3.

A Description of a Mapping Unit.

3.1. Logical Form of a Mapping Unit.

The function required of a mapping unit is one of arbitrary re-arrangement and masking of binary digit positions within a parallel, fixed-length computer word. Some examples are given in Figure 3.1. The mapping unit itself can be regarded as a form of "dependent " register in the sense of Bartee, Lebow and Reed [15], pp 19-20.

By examining the number of gating elements in such a unit implemented in conventional logic circuitry, some idea of the order of logical complexity of such a unit can be obtained. Consider the following system, suitable for inclusion in present day C.P.U.'s.

Input	33 bits (32 data bits plus an invariant '1')
Output	32 bits
Number of available mappings	512 (of the 10^{50} possible)

We will assume that the required mapping function is indicated by contents of an independent register S, consisting of bits $s_1, s_2 \ldots\ldots s_9$; the inputs to the unit are designated $x_1, x_2 \ldots\ldots x_{33}$; the outputs of the unit are designated $y_1, y_2 \ldots\ldots y_{32}$.

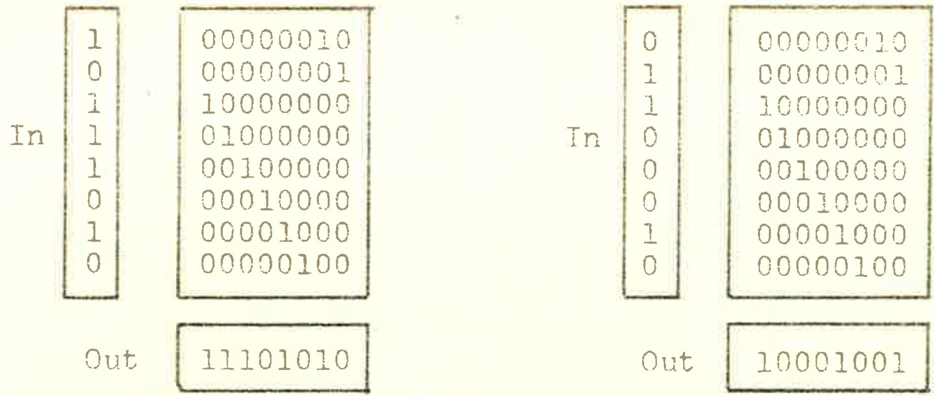
Hence:-

$$y_i = f(x_1, x_2 \ldots\ldots x_{33}, s_1, s_2 \ldots\ldots s_9) \text{ for } i=1,2\ldots\ldots 32 \qquad [3.1]$$

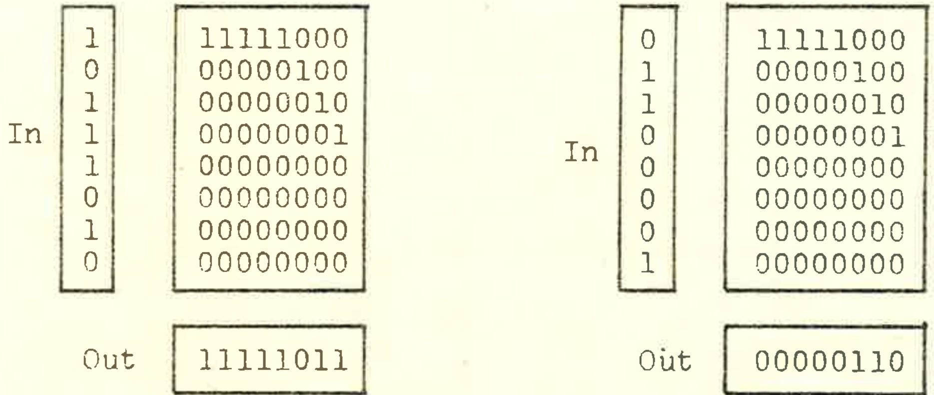
or, for facility in decoding,

$$y_i = f_2(x_1, x_2 \ldots\ldots x_{33}, s_1, s_2, \ldots\ldots s_9, \bar{s}_1, \bar{s}_2 \ldots\ldots \bar{s}_9) \text{ for } i=1,2\ldots\ldots 32. \qquad [3.2]$$

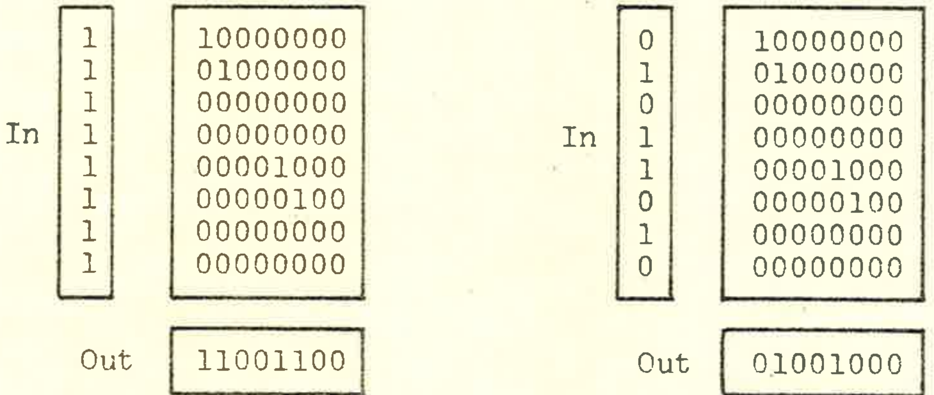
Assuming that "double"mappings are not permitted,



(a) Circular shift, 2 places left.



(b) Right shift with sign extension, 4 places.



(c) Arbitrary masking function, mask = 11001100.

Figure 3.1. Examples of Mapping Functions.

i.e. no bitwise functions of the form

$$y_i = x_j \text{ OR } x_k \qquad [3.3]$$

occur within any available mapping, it can be seen that each output bit must be driven (in the canonical form at least) by a 512-input OR gate, each of whose inputs is driven by a 10-input AND gate. Assuming that logic is performed by hardware requiring one gating element per literal appearing in the logical equations, such a mapping unit requires

$$N = 512 \times 10 \times 32$$
$$\simeq 1.6 \times 10^5$$

gating elements.

In practice, limitations on the fan-in and fan-out capabilities of the hardware elements will increase this figure by forcing the inclusion of elements used solely as buffer and amplifying devices, whilst both the effects of the sparseness of the mappings (i.e. the introduction of invariant zeros into the output) and minimization of the logical equations of the unit can be expected to reduce the figure. It is probable that the first effect will at least offset the combined effects of the last two, particularly if high speed operation of the unit is to be achieved. The mapping unit, if implemented in this form, requires of the order of 10^5 gating elements for its construction. It may also be noted that this figure is directly proportional to the number of available mappings that the unit may perform. The implementation of a mapping unit in this form is physically difficult and economically impossible.

A second method of viewing the problem yields an insight into the system such that implementation by an alternative method is feasible. As originally

proposed by Mercer [16], the mapping unit, (or T matrix as he called it), can be regarded simply as a flexible interconnection between two registers, the input register A and the output register B. (See Fig. 3.2).

Any bit b_j of the output register can be expressed as the bit-wise logical sum of intersection of the input (regarded as a boolean vector) and the jth column of the boolean transfer matrix T_s.

$$b_j = \Sigma_i (a_i \text{ AND } t_{ji}) \qquad [3.5]$$

where Σ represents the operation of bit-wise logical summation of a boolean vector.

It may also be noted that any particular mapping is defined solely on positions within the input and output vectors and is specified independently of the values of these vectors. The value of the transfer matrix T_s is therefore independent of the input A and is a function solely of the required mapping, which is indicated by the contents of the selection register S. Conversely, each mapping is identified by a particular value of S, which has associated with it a particular value of T_s.

The problem has therefore been reduced to two distinct sub-problems:-

(1) Given a mapping "address" in S, evaluate the appropriate T_s matrix, and hence t_{ji},

and (2) Using these values of t_{ji}, together with the input information, evaluate the output bit b_j by equation [3.5]

The first problem, that of "decoding" a particular input vector to yield an extended output

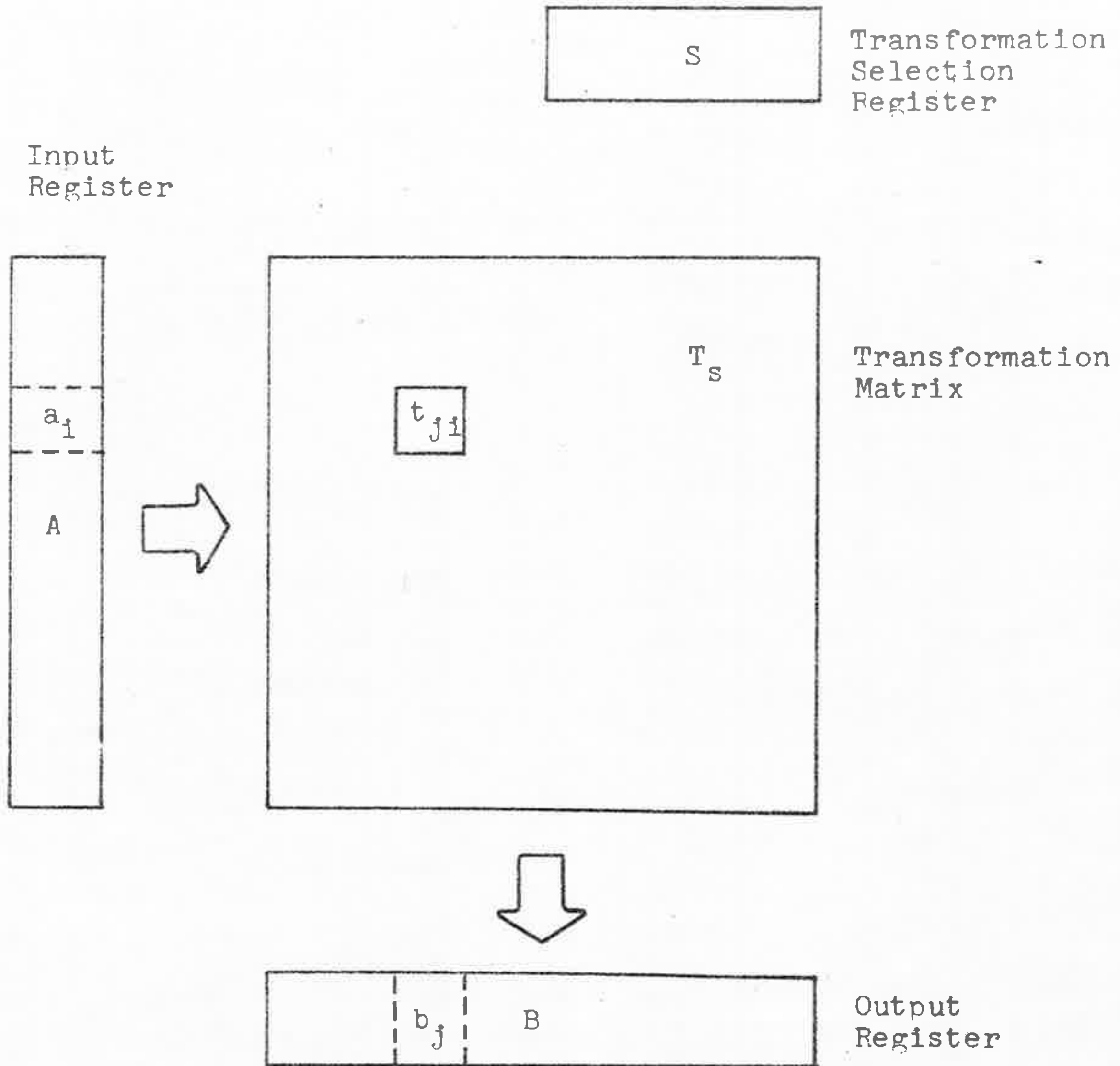


Figure 3.2. Mapping Unit Terminology.

vector (or matrix, as in this case), has been studied in a variety of logical environments. The most attractive solutions, particularly when the problem is of a relatively large scale, involve the use of some form of storage. The input vector to be decoded' is treated as the address of a location within the store. The contents of this location, when read out of store, form the output or "decoded" vector. Forms of storage commonly encountered are diode matrices, variable random access stores and fixed or read only storage [17].

The advantages of using fixed storage for this application are well known. Briefly, fixed storage offers

(a) A significant advantage in cost/bit/cycle-time figures,

(b) High operating speeds,

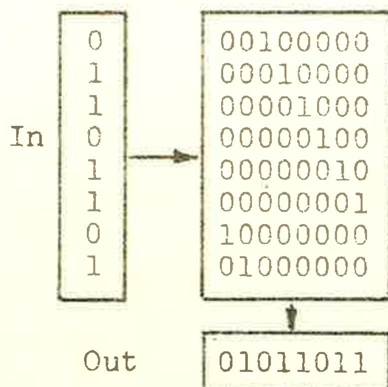
(c) Physical permanence of stored information,

(d) Relatively simple techniques for the modification of stored information,

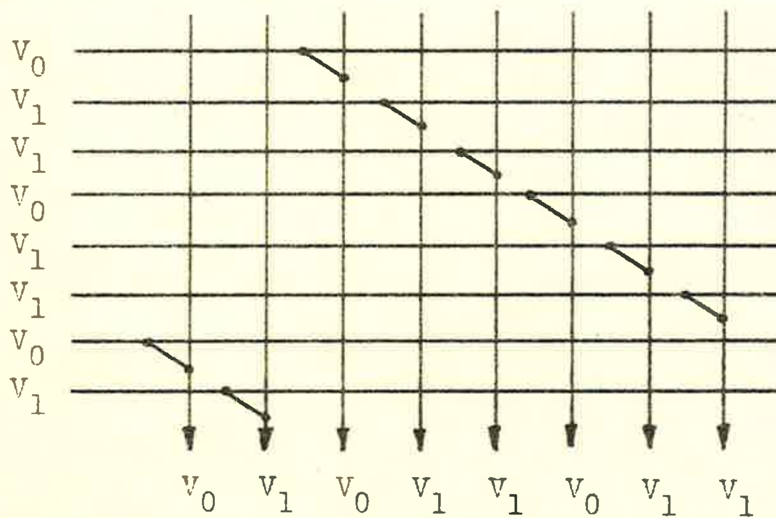
and (e) Simple implementation, suitable for automatic assembly.

The second problem, that of implementing equation [3.5], is a reduced form of the general combinatorial problem discussed at the start of this section. For the system specified above, approximately 3×10^3 gating elements would be required to implement equation [3.5]. A major difficulty is to make all t_{ji} bits simultaneously available for logical operation. These t_{ji} bits number n^2 in an n bit mapping, and unless due regard is paid to economizing the t_{ji} circuitry, the economics of the system as a whole will suffer.

In fact, the second problem of implementation can be simplified by noting that the matrix T_s specifies switching points within the grid formed by the input



(a) Symbolic transformation, using T matrix.



(b) Logical representation by switching.

Figure 3.3. Switch Representation of a Mapping Unit.

and output busses, (See Fig. 3.3). The second part of the system may therefore be implemented by providing a set of switches which may be controlled by the individual t_{ji}. (t_{ji} =1 is equivalent to a closed switch; t_{ji} =0 is equivalent to an open switch).

The logical design of the unit therefore calls for the provision of two main elements:-

(1) A decoding device which transforms the value of the selection register to the value of the matrix T_s. From a consideration of the size and speed of the decoder, it can be seen that it will probably take the form of a fixed store.

(2) A switching matrix, controlled by the transformation matrix T_s, which interconnects the input and the output lines in the required pattern.

3.2 Extensions to the Logical Operation of the Mapping Unit.

Several extensions to the basic form of the mapping unit presented above are possible:-

(a) Mappings need not be performed between registers of the same length. It is possible to extend either the input or the output register to improve the efficiency of the transformation operation. For example, Figure 3.4 shows the generation of two separate 8 bit signed integers from an 8 bit word.

(b) By providing an extra input bit, $a_{n+1} = 1$, the injection of invariant bits into the output becomes possible. An example is given in Fig. 3.5.

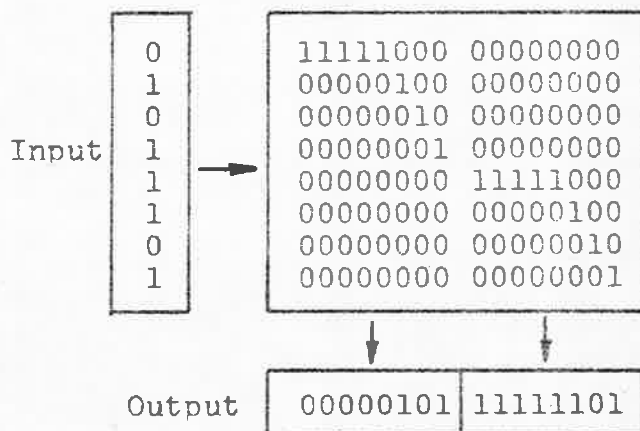


Figure 3.4. Example of Mapping Between Unequal Length Input and Output Registers.

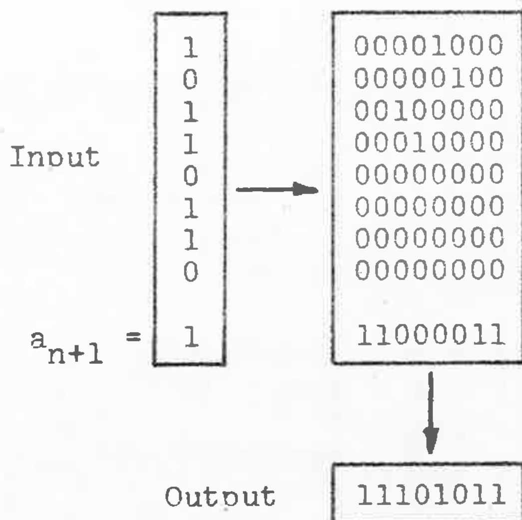


Figure 3.5. An Example of the Injection of Invariant Bits into the Output.

(c) The mapping device is useful for generating the special vectors of Iverson [18] and is useful in implementing descriptions of operations in this language (see Section 7).

3.3. Implementation of the Mapping Unit.

3.3.1. System Design.

In implementing a hardware version of the logical form of the mapping unit described in the previous section, two criteria were observed:-

(1) The design should be such that any increase in the number of available mappings should result only in an increase in the complexity of the decoding circuits associated with the selection register S. No increase in the number of switching elements in the transfer matrix should occur.

(2) The mapping unit should be designed as a dependent register suitable for inclusion in a computing structure and should operate in times comparable to other dependent registers (adders, logical operators, multipliers, shifters etc.) as implemented in up-to-date technology.

The proposed designs for the fixed store and the switching array are shown in Fig. 3.6, and an idealized timing diagram for this system in Fig. 3.7.

Briefly, the method of operation is as follows:-

(1) The input information is placed in the input register A and the address of the required transformation is placed in the selection register S.

(2) The value of S is decoded and the appropriate drive line P^S is driven by a current pulse.

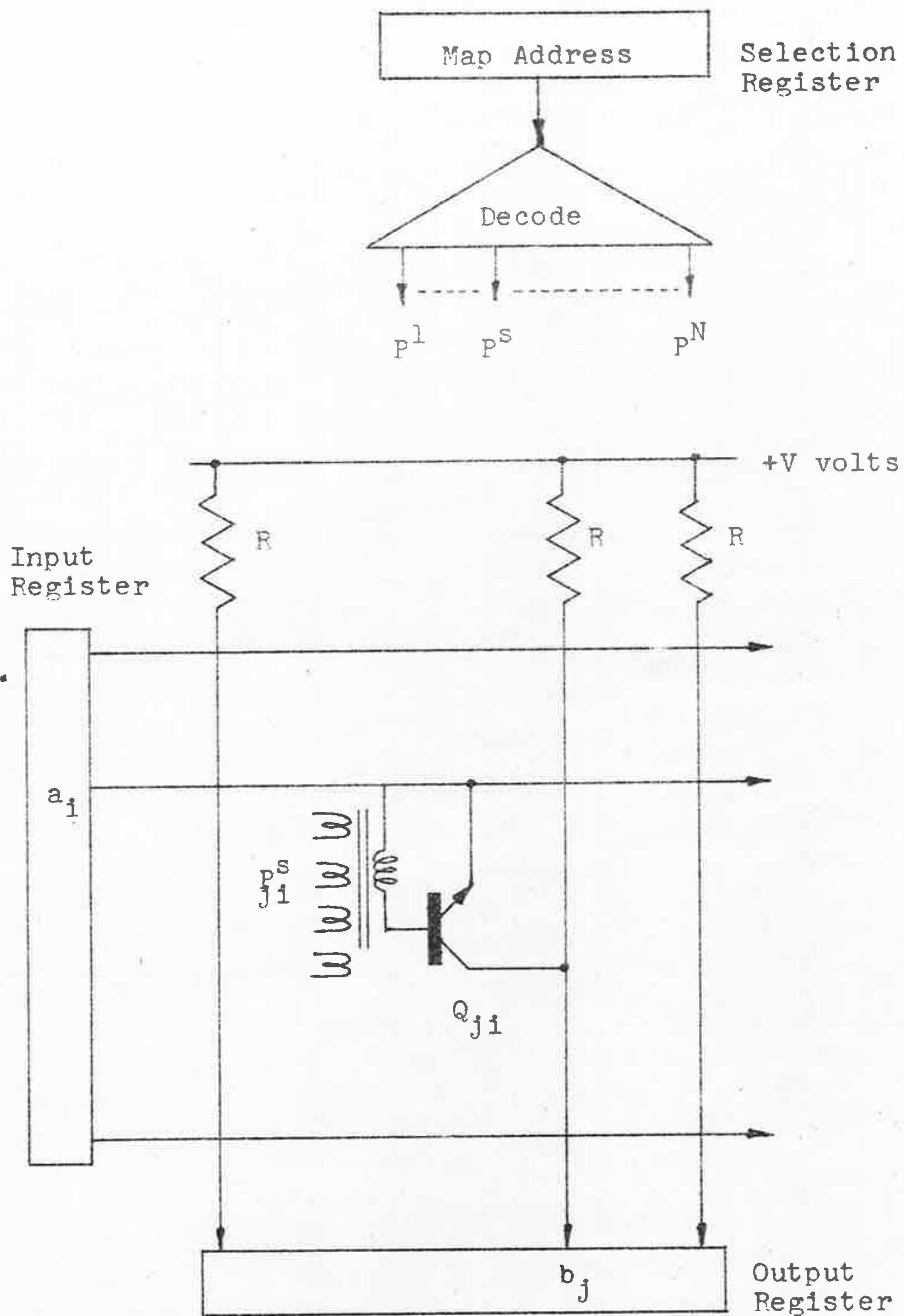


Figure 3.6. Proposed Implementation of the Mapping Unit.

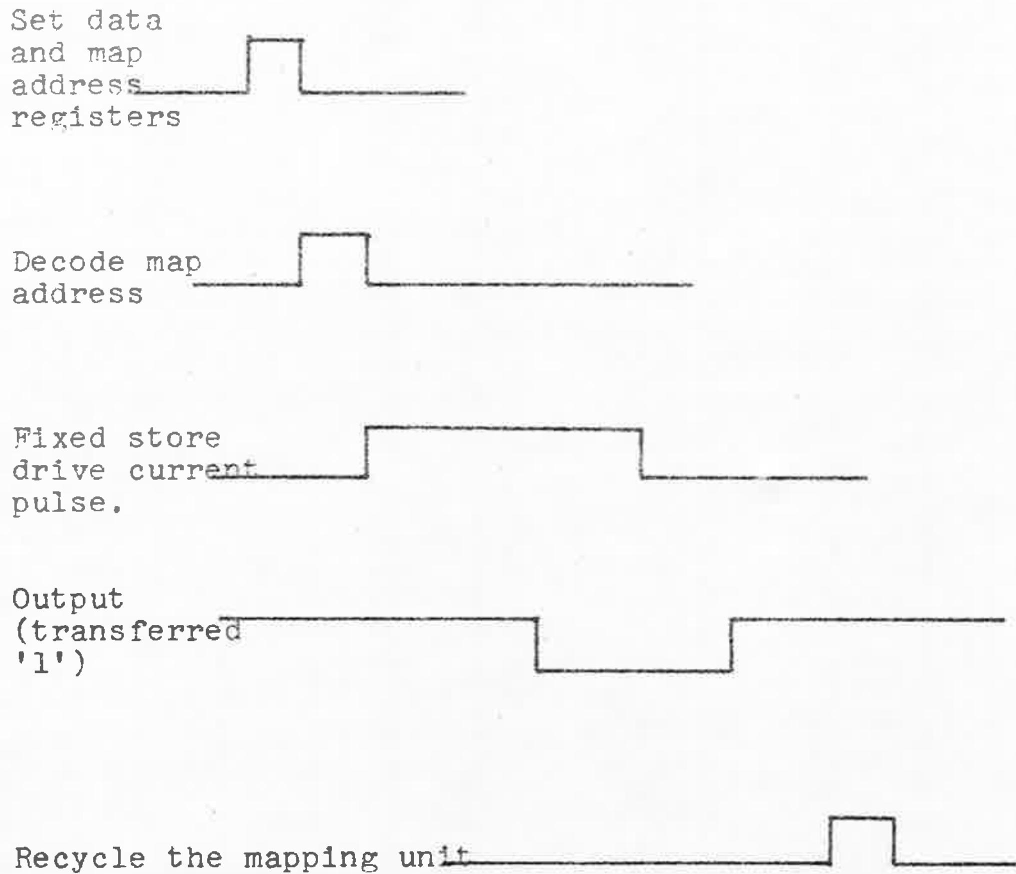


Figure 3.7. Idealized Timing Diagram of the Proposed Implementation of the Mapping Unit.

The path of this drive line P^s is determined by connecting in series all the transformer primary windings P^s_{ji} at those positions in the connection matrix where $t_{ji}=1$ in the transformation matrix.

(3) Should a transfer be specified at any matrix intersection (j,i), current will pass in the appropriate primary winding P^s_{ji}. A current therefore appears in the transformer secondary winding which, together with the threshold network R_2,R_3,C_1, drives the transistor Q_{ji} into saturation. The voltage of the emitter of this transistor,which is determined by the contents of the input bit a_j, is transferred to the output line b'_j. This output signal is used to set the output flip-flop b_j.

(4) Should no transistors switch in a column, b'_j is held to a zero representation via R_1.

3.3.2 Hardware Details.

Fixed Storage.

The fixed storage section of the proposed mapping unit is implemented as an array of multi-primary transformers, whose secondary windings (or sense windings) provide signals corresponding to the t_{ji} bit associated with the logical position of the transformer. The transformers consist of a long ferrite rod (21/2" long x 1/16" diameter, linear B-H material) along which is wound a sense winding of 15 turs., (see Fig. 3.8). The primary windings, each one associated with a particular transformation selection address, are provided in the form of printed circuit loops which either include ($t_{ji}=1$) or exclude ($t_{ji}=0$)

the ferrite rod (see Fig 3.9). By providing a current pulse of known direction in a primary loop P^S, corresponding to a value of the selection address register S, a signal is received (of a definite polarity) from those transformers which the drive line encircles.

Any attempts to find theoretical values for outputs from such a system are frustated by the non-ideal nature of the physical layout. A number of physical layouts were investigated empirically, and little or no differences in output could be detected.

However, since open flux paths are used, cross-talk between magnetic elements was investigated. Since the transistor switches are essentially uni-polar devices, i.e. signals in one direction only will tend to switch them on, it can be seen (Fig 3.10) that only second order effects can produce spurious outputs on elements distance 2 from a driven element. Such effects were generally undetectable within system ground noise etc., at an array spacing of 3/8".

Outputs vary depending upon the position of the drive wire along the length of the ferrite rod. Various non-linear patterns for the sense winding were investigated and the result is shown in Fig. 3.11.

Switches.

The form adopted for a switch element is shown in Fig. 3.12. R_2,R_3, and C_1 provide a threshold setting facility, which may be used to ease drive requirements. If sufficient drive currents are available they may be dispensed with and the base return lead tied to the emitter of the transistor.Collector clamping via R_1 and D_1, is used to reduce collector voltages in the off state and hence enhance switching speeds.

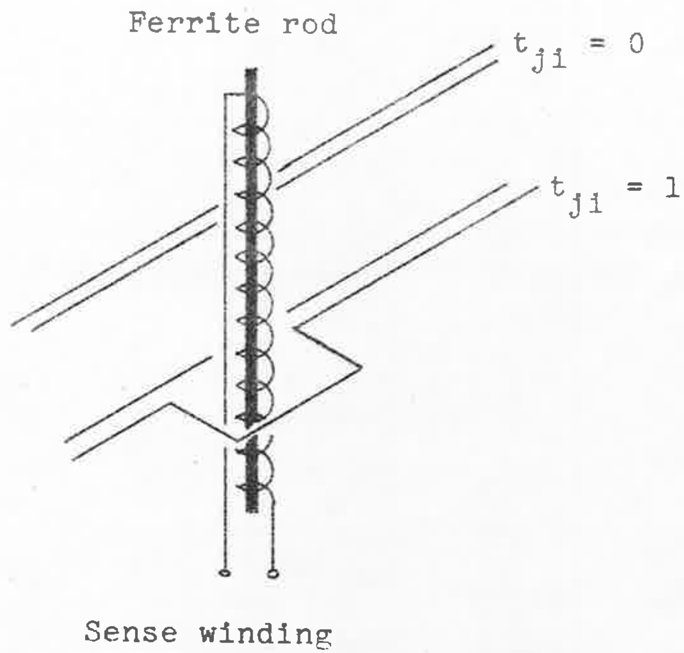
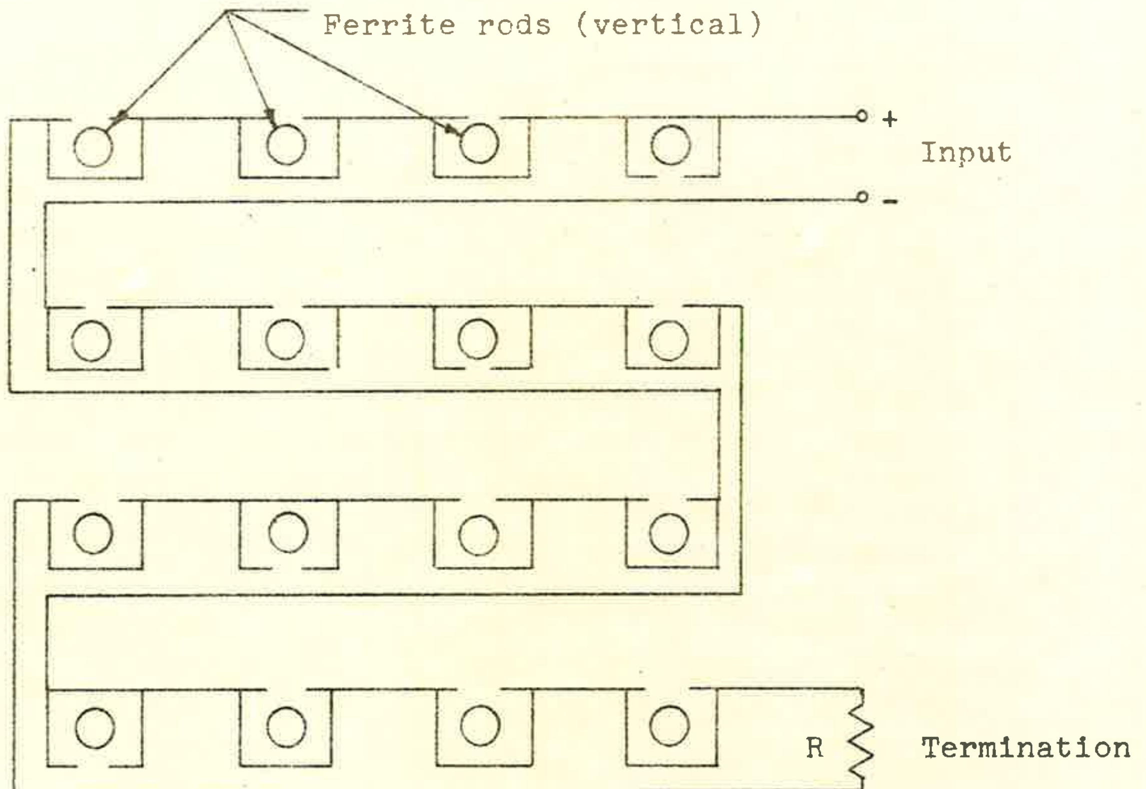


Figure 3.9a. Winding Details of the Transformar.



Transformation Matrix

0001
0010
0100
1000

Figure 3.9b. Example of a Winding Pattern for the Mapping Unit.

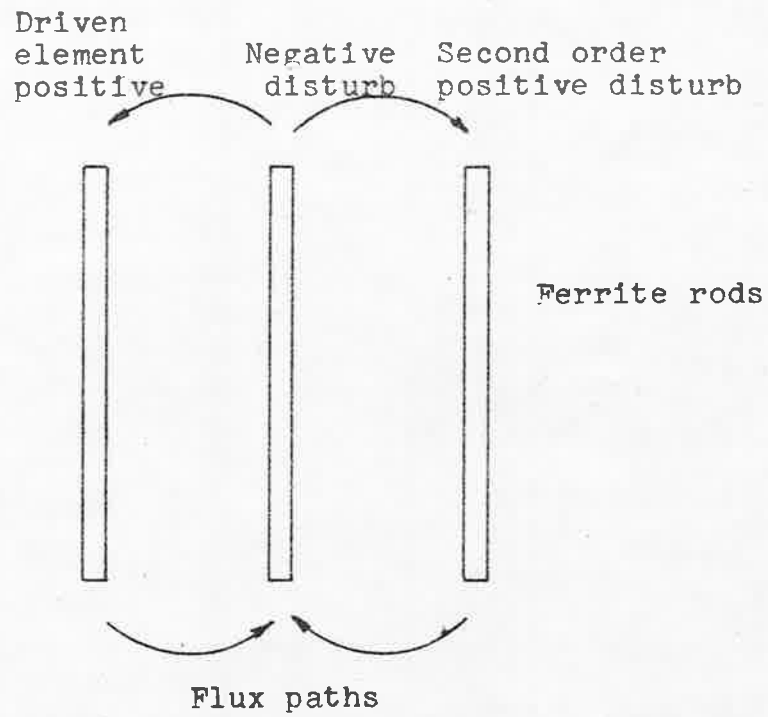


Figure 3.10. Cross-Talk Between Elements of the Proposed Mapping Unit.

L

Drive line

k.L

Ferrite rod

100%

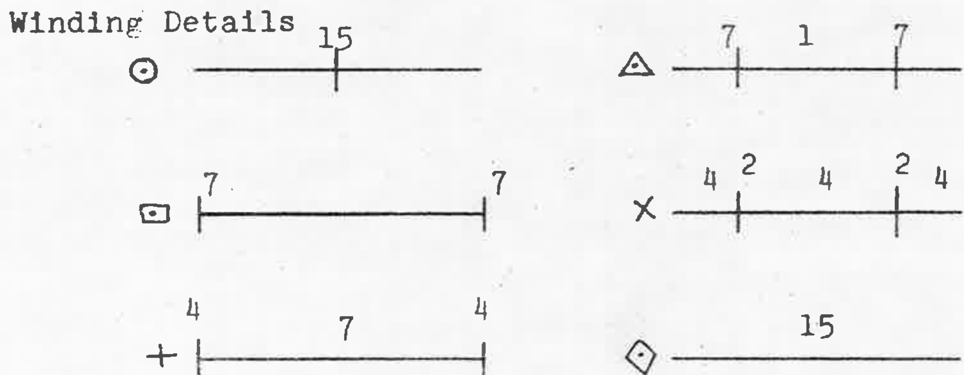
Output

50%

0.5

1.0

k



The figures refer to the number of turns in the designated area of the ferrite rod.

Figure 3.11. Sense Winding Patterns and Output.

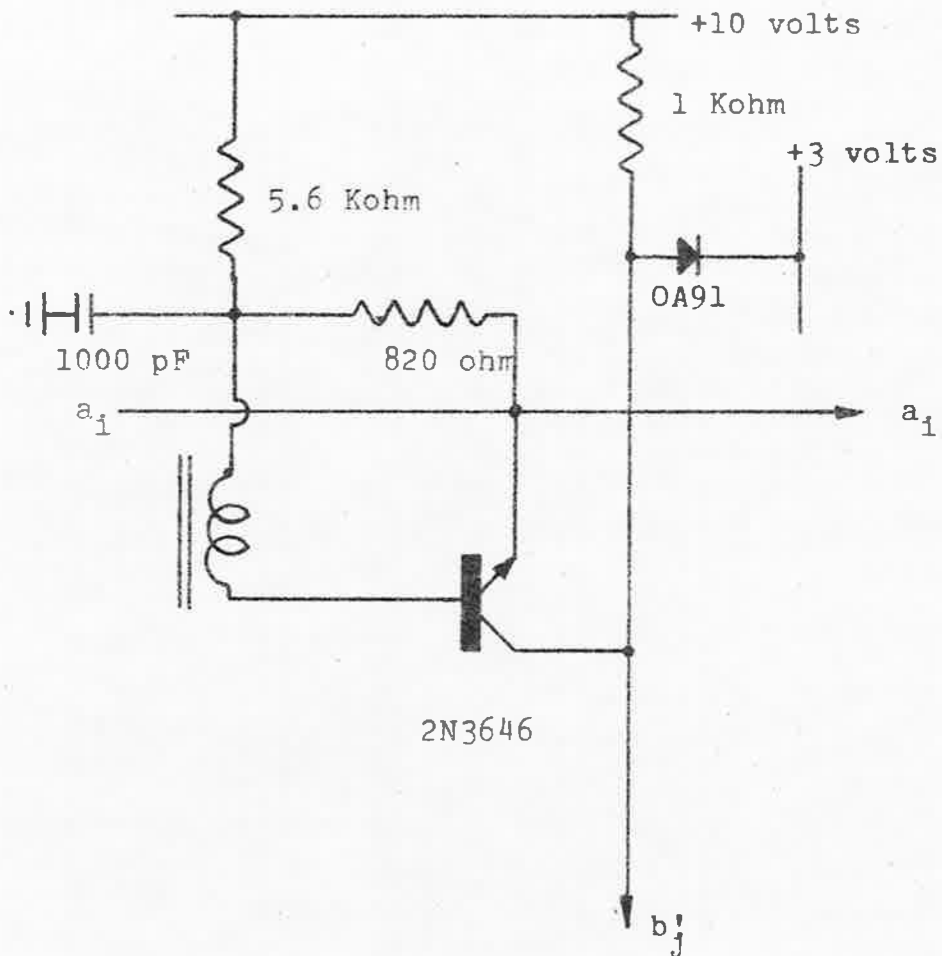


Figure 3.12. Final Form of the Matrix Switch.

Feasibility Model.

The responses of a 4x4 bit feasibility model are summarized in Fig. 3.13. An operation time of 35 nano-seconds and a cycle time of approximately 75 nano-seconds are indicated under the drive conditions used, viz, 100 mA current pulses rising (10%-90%) in approximately 10 nanoseconds.

Address decoding and register setting using moderately high speed logic will increase these times to 50 and 90 nanoseconds respectively.

The extensions of the 4x4 bit feasibility model to a practical unit of 18x36 (as proposed for CIRRUS, see Section 4), will cause some degeneration in speed. The principal factors involved will be:-

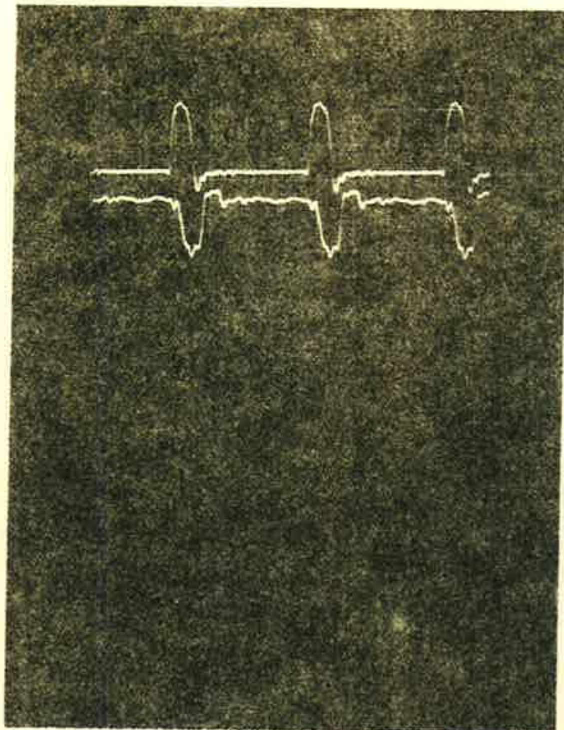
(a) The increase in the drive line lengths and number of elements driven per line will increase the inductance of the drive lines, limiting the current rise time attainable by constant voltage driving of the lines. By treating the drive lines as transmission lines and terminating them in their characteristic impedance (which will not be well defined, owing to the complex layout used, and the presence of the ferrite rods, matrix diodes, switch characteristics) these effects may be minimized.

(b) The added stray capacitance imposed upon the current generators will limit their switching speeds.

(c) Switching elements will be more heavily loaded by stray capacitances, resulting in lower switching speeds. The switch circuit

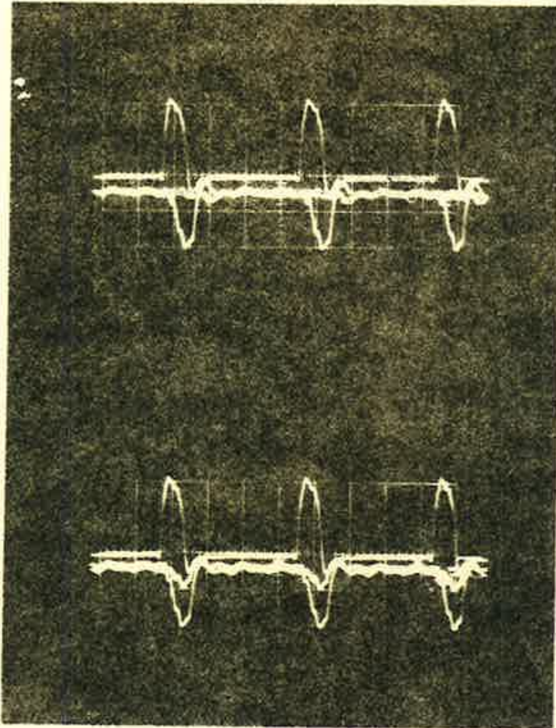
impedance levels should be kept as low as possible consistent with high switching efficiency to offset this capacitive loading.

Unfortunately, it has not proved possible to implement a full scale mapping unit, so that no full scale speed data is available. However, it is interesting to note that read-only storage units of reasonable capacity can attain cycle times of 0.1.microseconds [17]. The mapping unit can be considered as a modified fixed store, whose outputs are logically combined according to some input information. There is no factor apparent in the feasibility model which would prevent a full scale mapping unit from achieving a similar speed.



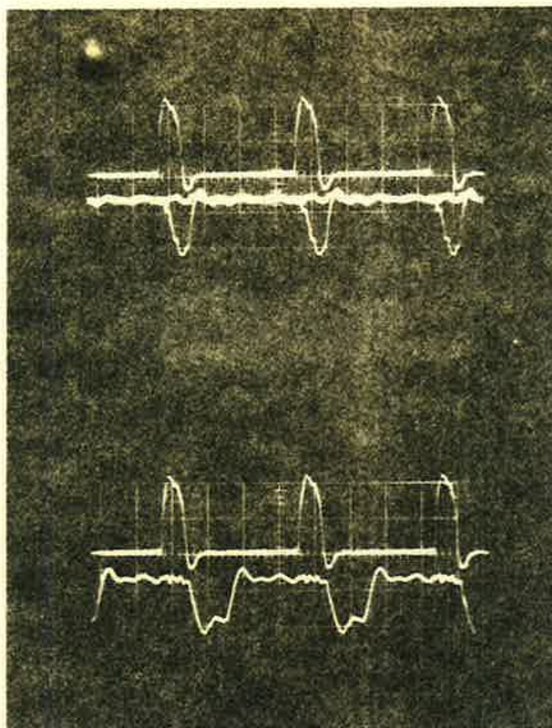
Upper	Drive Current	50 mA/div.
Lower	Selected '1' Output	2 v/ div.
Horizontal		50 nsec/div.

Fig. 3.13a. Output from the Feasibility Model.



Upper.	Upper	Drive Current	50 mA/div.
	Lower	Selected '1' O/P +	
		Non-selected '1' O/P	2 v /div.
	Horizontal		50 nsec/div.
Lower.	Upper	Drive Current	50 mA/div.
	Lower	Selected '1' O/P +	
		Selected '0' O/P	2 v /div.
	Horizontal		50 nsec/div.

Fig. 3.13b. Output from the Feasibility Model.



Upper.	Upper	Drive Current	50 mA/div.
	Lower	Selected '1' O/P +	
		Non-selected '0' O/P	2 v /div.

Lower. This trace shows the effect of an incorrectly wired sense winding. The transistor switches on at the negative transition of the drive current and sense winding time constants and the carrier storage time of the transistor determines the on period.

Upper	Drive Current	50 mA/div.
Lower	Selected '1' O/P	
	Anti-phase winding	2 v /div.
Horizontal		50 nsec/div.

Fig. 3.13c. Output from the Feasibility Model.

SECTION 4.

The Inclusion of a Mapping Unit in a Computing Structure.

4.1. External Characteristics.

When a mapping unit is treated as an entity for the purposes of logical design, the interfaces that the mapping unit presents to the remainder of the system will be largely determined by the logical form and the design philosophies of the overall system. The provision of buffers (both input and output), the format of activation and control signals and the physical form of the interfaces will be determined by system design considerations, and the mapping unit itself will be responsible for converting these 'standard' signals to forms suitable for its own internal use. One example, that of a mapping unit modified for use with an asynchronous control unit, is shown in figure 4.1.

The mapping unit can be modified by techniques of this type to forms suitable for integration in any parallel computing structure.

4.2. The Inclusion of a Mapping Unit in CIRRUS.

4.2.1. The CIRRUS Computer.

CIRRUS is an eighteen bit parallel micro-programmed computer available as a research machine within the University of Adelaide. It was originally constructed to evaluate several new ideas in computer architecture and features an extremely flexible micro-structure controlled by fixed storage. This enables special purpose structures to be implemented within the hardware by suitable sequences of micro-instructions. The following sub-section presents methods for augmenting the CIRRUS hardware to include a mapping unit.

A brief description of CIRRUS is given in

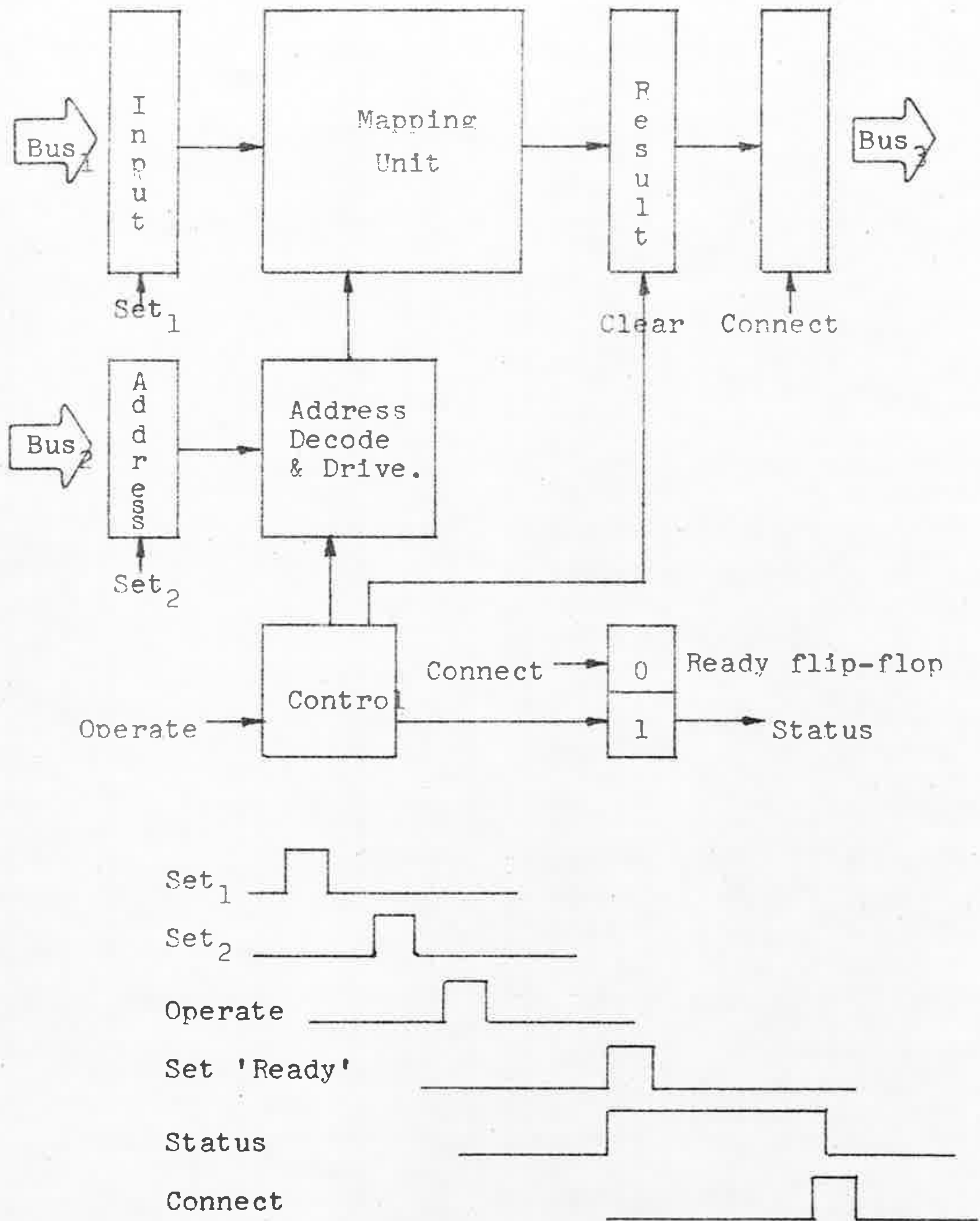


Figure 4.1. Adaption of a Mapping Unit for an Asynchronous Environment.

Appendix 1, and more detailed descriptions are available in [1],[2],[3] and [4].

4.2.2. Provision of a Mapping Unit in CIRRUS.

In providing a mapping unit within the already existing CIRRUS structure, several additional facilities are required for communication and control of the augmented structure:

(1) Information paths between the mapping unit and the existing structure must be constructed for input, output and transformation selection quantities associated with the mapping unit.

(2) Timing and control signals for the mapping unit must be generated in the main control unit of the computer. Information must be provided in the micro-instruction format for specification of these functions by micro-programmers.

(3) The specific form of the mapping unit (size, speed etc.) and the nature of any ancillary equipment attached to it will affect the method of interconnection. The method of interconnection should leave the existing structure substantially unchanged in order that presently developed software can continue to be used.

If the mapping unit were to be provided on the computer, it appears reasonable that its main areas of utilization would be in the fields of :

(1) masking and shifting operations under the control of machine language instructions.

(2) manipulation of operands for numerical processing.

(3) provision of facilities for experimental studies of machine structures.

It may be expected that in all these applications, word lengths other than the standard 18 bit (hardware) word length will be encountered, e.g. floating point operations are at present implemented in 28/8 floating point, compatibility studies will involve word lengths incommensurate with 18 bit words, many CIRRUS non-numerical operations treat 6 bit character strings packed 6 to a 36 bit word, etc. It is therefore desirable that the mapping unit prove relatively efficient when employed on mapping operations of words of length greater than 18 bits.

In practice several systems were investigated:

(S1) An 18x18 mapping unit using explicitly assigned source (input), sink (output) and address (transformation selection) registers, called into operation by a specially constructed micro-instruction.

(S2) An 18x18 mapping unit using fixed source and address registers, feeding its output to one of the existing shifting inputs of the lower registers (which will not be required in the augmented structure, since shifting operations will be performed by the mapping unit). The selection of source and sink registers may be controlled via existing micro-instruction formats.

(S3) An 18x18 mapping unit, by-passing the add-logic unit, having its own address register, and being able to select one of two sources and feeding its output to the lower registers. Bits used to control the add-logic unit may be used to control such

a mapping unit. Selection between the outputs of the add-logic unit and the mapping unit may be accomplished by input selection to the lower registers.

(S4) A 36x36 mapping unit, using explicitly assigned source, sink and address registers.

(S5) A 36x18 mapping unit, having an explicitly assigned source register, its own address register and feeding its output to one of the lower register inputs. A 36x18 mapping unit is proposed in order to circumvent two logical -OR operations encountered in the mapping of a 36 bit word by an 18x18 mapping unit. (see below.)

A diagrammatic summary of the resulting structures and their modes of operation are given in Figs. 4.2-4.6. A summary of estimated costs and operation times on various problems are given in Tables 4.1 & 4.2.

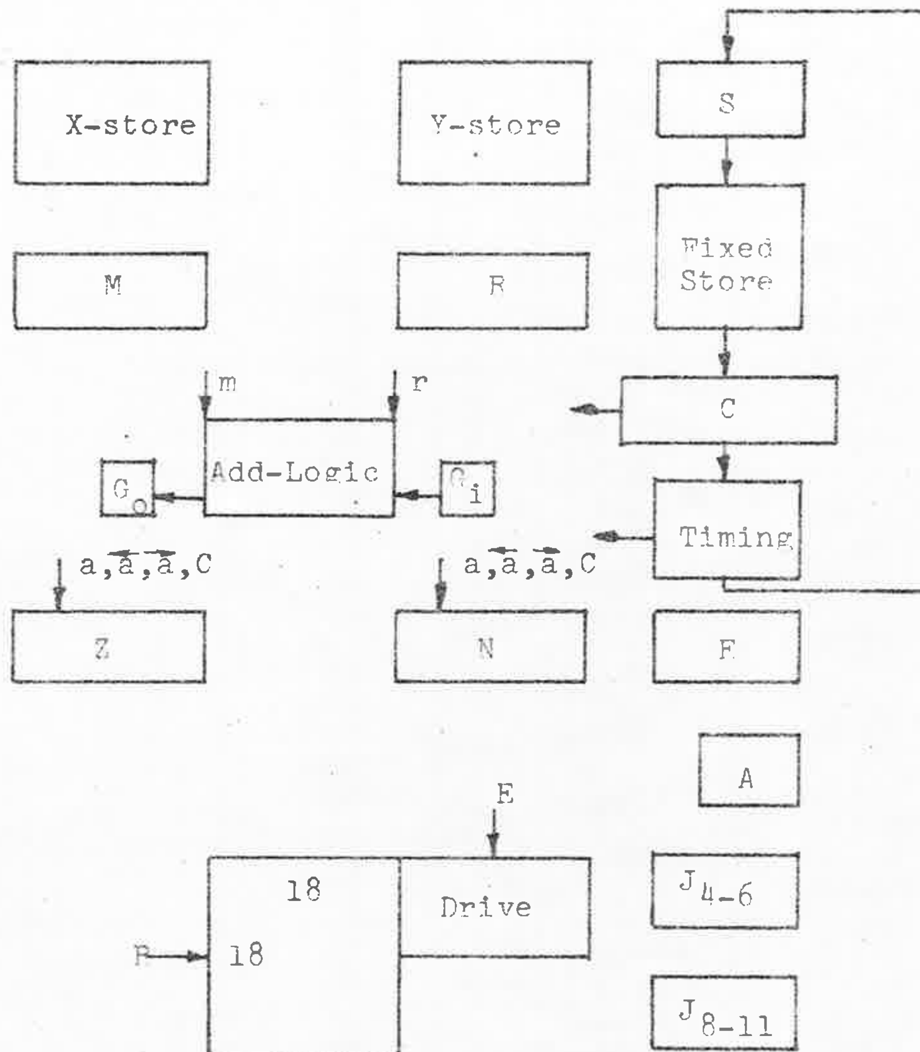
The selection of one of these systems for inclusion in CIRRUS involves the consideration of the following points.

(1) In assessing speed/cost ratios for various forms of the mapping unit, it must be appreciated that

(a) the cost of the mapping unit is only a small fraction of the cost of the computer as a whole (approximately $A50,000 component cost)

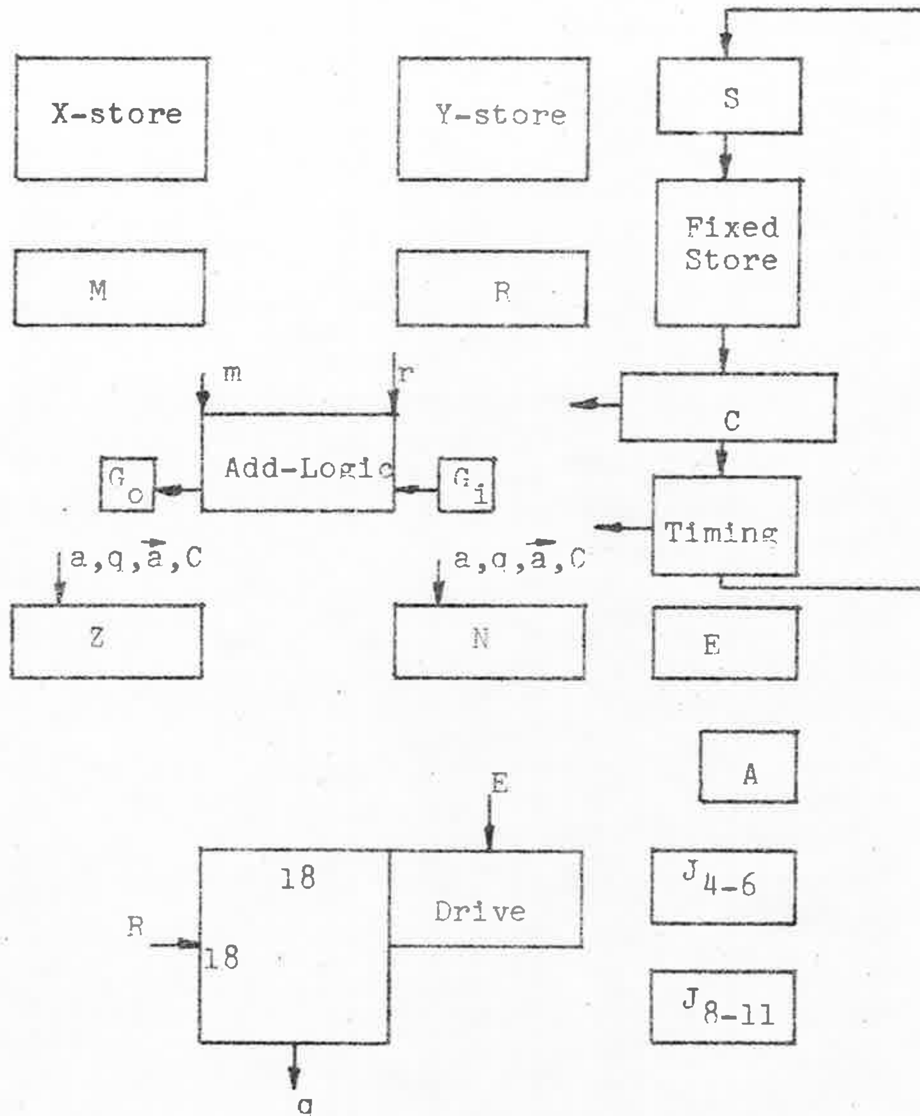
and (b) the speed of the computer in conventional operations will be only slightly improved by the addition of a mapping unit.

(2) Certain forms of the mapping unit tend to be rather more flexible in application than others. In particular, those systems



N.B. The output from the mapping unit appears directly in hhe Z register. The mapping unit is activated by an $\overleftarrow{a}$[left shifted] transfer to Z, in any micro-instruction of types FA, AXY, AX,AY.

Figure 4.2. Mapping Unit in CIRRUS, System S1.



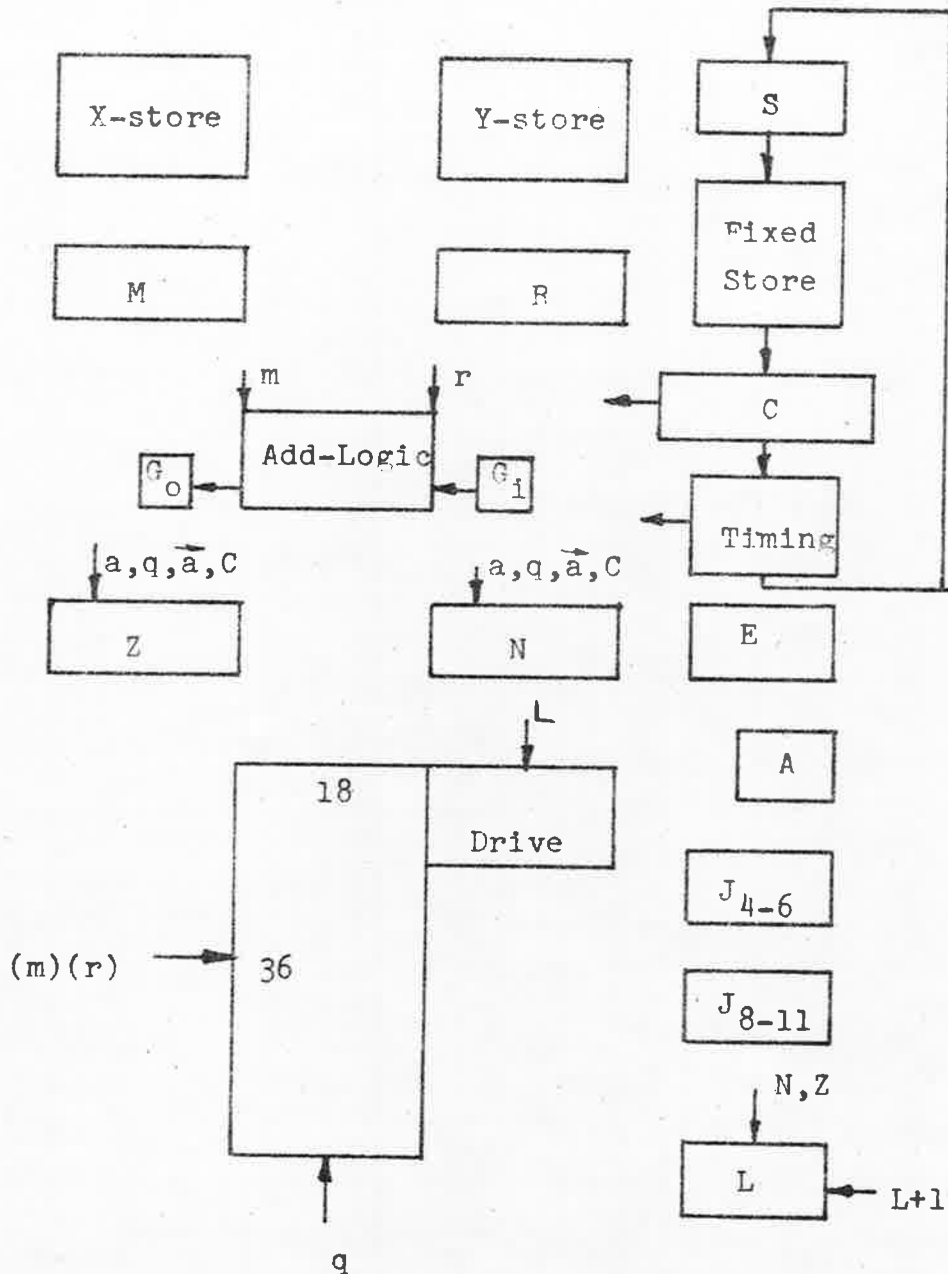
N.B. Mapping operations are always initiated by micro-instruction types FA, AXY,AX,AY.

Figure 4.3. Mapping Unit in CIRRUS, System S2.

X-store
Y-store
S
Fixed Store
M
P
m
r
C
G_o
Add-Logic
G_i
Timing
a, q, $\vec{a}$, C
a, q, $\vec{a}$, C
Z
N
E
L
A
18
Drive
m, r
18
J_{4-6}
J_{8-11}
q
N, Z
L
L+1

N.B. The mapping Unit is activated on all FA, AXY, AX, and AY micro-instructions.

Figure 4.3. Mapping Unit in CIRRUS, System S3.



N.B. The mapping unit is activated by micro-instruction types FA, AXY, AX, AY.

Figure 4.5. Mapping Unit in CIRRUS, System S5.

N.B. The output of the mapping unit appears automatically in the Z and N registers. The mapping unit is activated by a transfer of the form a Z in micro-instruction types FA, AXY, AX, AY.

Figure 4.6. The Mapping Unit in CIRRUS, System S4.

having their own address register (S3 & S5) offer the following advantages:

(a) Transformation selection addresses may be simply incremented by a hardware counter technique, rather than using the add-logic network. This improves the speed of the mapping unit (S3) over the speed of a similar unit (S2) at little extra cost (≅8%).

(b) Both the S3 and S5 systems do not use registers within the general pool to hold augmented address quantities. Hence the structure tends to retain the flexibility provided in the original design, which was rigorously minimized.

(c) Should the number of available mappings have to be increased, extension of the selection address register and associated decoding is relatively simple.

The disadvantage of the presence of an extra selection address register lies in the fact that this register is only infrequently used by the system. It therefore has a low utilization factor and it correspondingly tends to lower the utilization factor of the machine as a whole.

(3) The provision of a mapping unit having at least one dimension equal to time the word length (S4 or S5) has the advantage that shifting of multi-length quantities is somewhat simplified. As an example, consider the mapping of a 36 bit quantity with an 18x18 mapping unit (S3). The processes required are summarized graphically in

Table 4.1. Alternative Mapping Unit System Costs.

System	Control	Input Selection and Buffering	Selection Register	Decoding Logic & Line Drivers	Fixed Store & Switching Elements	Output Buffers	TOTAL
S1	20	18	0	676	324	18	1056
S2	20	18	0	676	324	54	1092
S3	40	54	63	676	324	18	1175
S4	20	36	0	676	1296	36	2064
S5	40	36	63	676	648	72	1535

N.B. Costs are expressed in units equivalent to the cost of a 2-input transistor (or integrated circuit) NOR gate.

Table 4.2. Alternative Mapping Unit System Speeds.

System	Problem Number					
	1	2	3	4	5	6
S1	4.5	7.5	13.5	40.5	37.5	43.5
S2	4.5	7.5	13.5	25.5	37.5	37.5
S3	3.0	7.5	7.5	22.5	33.0	33.0
S4	3.0	7.5	7.5	3.0	19.5	19.5
S5	3.0	7.5	7.5	4.5	19.5	19.5

N.B. 1. All times are given in micro-seconds

2. Problem descriptions

1. 18 bit mapping, information in registers
2. " " " " " Y-store
3. " " " " " X-store
4. 36 bit mapping, information in registers
5. " " " " " Y-store
6. " " " " " X-store

Fig. 4.7 and may be described by the equations

$$U' = Map_1(U) \text{ OR } Map_2(L)$$
$$L' = Map_3(U) \text{ OR } Map_4(L)$$

where U,L are the two 18-bit halves of the original quantity UL, and U',L' are the two 18-bit halves of the mapped quantity U'L'. The process requires four mapping operations,two logical OR operations and (in CIRRUS, where the number of hardware registers in the general pool is not large enough to store all intermediate quantities) two storage cycles to a buffer position in core store. These operations require a total of 22.5 micro-seconds in CIRRUS. Alternatively using a 36x36 mapping unit (S4) the result may be obtained in one mapping operation, 3.0 micro-seconds in CIRRUS. The system S5 using a 36 input, 18 output mapping unit avoids the production of intermediate quantities, but needs to two mapping cycles to produce the result. This single cycle increase in mapping time (1.5 micro-seconds in CIRRUS) is accompanied by a 50% decrease in cost. Mappings upon 18 bit quantities can be easily provided by the inclusion of a null or zero map over the remaining half of the input.

In the light of the foregoing arguments, it has been decided tp propose that system S5 be included within the CIRRUS system. A detailed design for this inclusion is given in Appendix 2.

Briefly the significant features of the design are:

(1) The mapping unit provided is a 36 input,

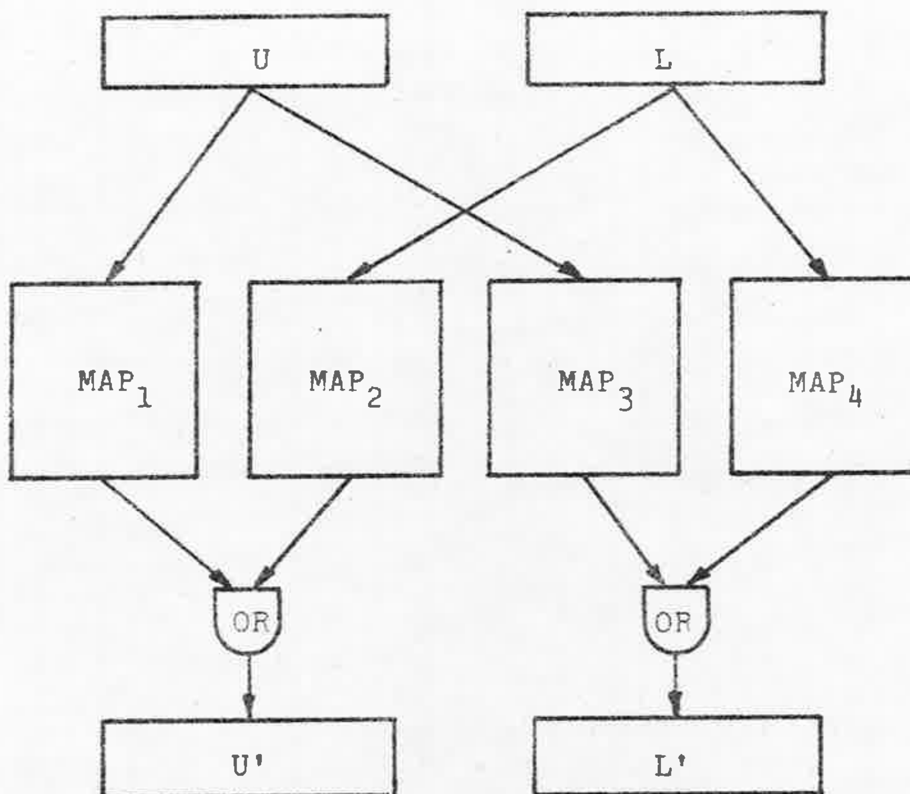


Figure 4.7. A Double Length Mapping Operation.

18 output form.

(2) The input to the unit consists of a concatenation of the inputs to the add-logic unit. By this feature, together with the selection facilities for these inputs already included in the system, a wide range of inputs to the mapping unit (including 18 bit forms with appended zeros)is possible.

(3) Outputs from the unit are fed into the lower register setting selection circuitry so that mapped outputs may enter either of the lower registers, N or Z.

(4) A selection register L is provided. This selection register may be set from one of the machine registers N or Z, prior to the mapping operation and may be incremented by 1 after completion of the mapping operation.

(5) All selection of alternative information is controlled via micro-code. (see Appendix 2 for formats).

(6) Timing of the unit is such that a mapping operation can be performed in the same time as an add-logic operation. In fact, the mapping unit can be envisaged as operating in parallel with the add-logic unit, selection of outputs determining which function is actually being performed.

Programs for CIRRUS have been written to simulate both the existing structure and the structure including the mapping unit. Sample listings of these programs are given in Appendix 3. All applications of the augmented structure described hereafter were investigated with these simulators.

SECTION 5.

Mapping Units and Machine Instructions.

5.1. Machine Language Instructions and Micro-Programming.

It is the purpose of this section to examine the effect that the inclusion of a mapping unit within a computing structure has upon the speed and efficiency of the execution of "conventional" machine instructions. "Conventional" machine instructions are those instructions commonly found within most present day computer instruction repertoires, viz, arithmetic and logical operations, branching operations and certain non-numerical operations (principally shifting processes).

All computers have a repertoire of machine instructions or orders which can be divided into three main classes (or sequential combinations of these classes), viz, numerical operations, non-numerical operations or branching operations. Each instruction may be logically divided into a series of elementary operations specifying transfers and modifications of information within the computing structures. These "sub-operations" or micro-instructions may be implemented either in some form of control sequence generator, or within a fixed storage structure. A sequence of micro-instructions is called a micro-program and may be expressed symbolically as a time ordered list of transfers between independent and dependent machine registers.

We will be concerned here with examining the effect that a mapping unit has upon micro-programs for specific machine operations.

5.2 Arithmetic Operations.

It is common practice to provide both fixed point and floating point operations in present day

computer of reasonable size. In analyzing micro-programs necessary to implement the common arithmetic operations in fixed and floating point formats, the following points may be noted:

(1) The common operations provided within a computing structure at machine instruction level are addition, subtraction, multiplication, division and certain unary operations, e.g. load, negate, unload etc. In general, more complex operations, e.g. square roots, sines and cosines etc., are handled by machine language instructions.

(2) The only arithmetic operations generally provided at the hardware level are that of fixed point addition, and fixed point complementation (sign reversal). All other operations can be implemented as a series of additions and/or complementations. For example, a multiplication may be performed as a series of multiplier bit inspections, conditional additions and shifts of the so formed partial product and multiplier.

(3) Floating point representations employ a notation of the form

$$N = f.2^e$$

where N, the number represented, is carried in the computer as two fixed point quantities f, a signed fixed point fraction $1/2 \leq |f| < 1$, and e, a signed fixed point integer. The two quantities are commonly packed into one computer word, sacrificing some precision for an increase in range.

(4) Floating point operations can be reduced to a series of fixed point operations upon the

fractional and exponential parts of the numbers involved. Typically this process requires (a) s separation of the various fractions and exponents, (b) pre-conditioning of some or all of these fixed point quantities, (c) the actual fixed point operations upon the quantities, (d) post conditioning of the results and (e) re-construction of the resultant fixed point fraction and exponent into the normal floating point form.

The inclusion of a mapping unit within the computing structure will facilitate the execution of a number of micro-operations encountered in arithmetic processes, viz

(1) In the building up of complex operations from a series of simpler basic operations, the mapping finds application as a masking device and as a variable length shifting element. The mapping unit as a shifting element is treated at some length in Section 5.3. We present two examples of its application here in arithmetic operations.

Example 1. In the addition of two signed numbers N_1, N_2, represented in their floating point forms $f_1.2^{e_1}$, $f_2.2^{e_2}$, the result is given by

$$R = f_1.2^{e_1} + f_2.2^{e_2} \qquad [5.1]$$

In the interests of accuracy, both floating point numbers are assumed to be normalized, i.e.,

$$1/2 \leq |f_1, f_2| < 1 \qquad [5.2]$$

Before addition of the fractional parts occurs, the exponents must be stripped from the representations and the binary

points of the resulting fixed point fractions must be aligned. In order that overflow (the loss of bits from the most significant end of a word) does not occur, alignment of binary points is accomplished by pre-shifting the fractional part of the number whose exponent is the lesser, a distance to the right in places equal to the difference of the exponents i.e., if $e_1 \geq e_2$,

$$R = (f_1 + f_2 . 2^{e_2 - e_1}) . 2^{e_1} \qquad [5.3]$$

In the addition of two arbitary numbers, the length of pre-shift may vary from zero to a number equal to the length of the computer word. This shift may be performed by one operation of a mapping unit, addressed relative to some base by the modulus of the difference of the two exponents.
After the addition has been performed, the result should be normalized, i.e., the resultant fraction f_r reduced to

$$1/2 \leq |f_r = f_1 + f_2| < 1 \qquad [5.4]$$

and the exponent e_r suitably adjusted.
From 5.2 it follows that the range of f_r is

$$0 \leq |f_r| < 2 \qquad [5.5]$$

which may be divided into four sub-ranges

(1) $1 \leq |f_r| < 2$ [5.6]
(2) $1/2 \leq |f_r| < 1$ [5.7]
(3) $0 < |f_r| < 1/2$ [5.8]
(4) $|f_r| = 0$ [5.9]

Case (1) may be detected by the well known overflow techniques. Normalization of such a result may be performed by a single right

shift of the fraction, together with an increase of the exponent by 1.
Case(2) is already normalized and requires no further processing.
Case (4) denotes a zero result which obviuosly cannot be represented in normalized floating point form.
Case (3) requires that the resultant fraction f_r be left shifted an amount to satisfy equation [5.4], with appropriate modification to the exponent e_r. This amount will depend upon the actual values of the addend and augend, and must be determined from an examination of f_r. It is well known (Appendix 4) that, for a signed binary fixed point f, represented by a binary vector $b_1, b_2, b_3 \ldots\ldots b_n$, using diminished radix complements (one's complements) for negative number representations, if

$$1/2 \leq |f| < 1 \qquad [5.10]$$

then

$$b_1 \neq b_2 \qquad [5.11]$$

Correspondingly if

$$0 \leq |f| < 1/2 \qquad [5.12]$$

then

$$b_1 = b_2$$

A similar result holds for radix complements (two's complements) excepting f = -1/2, when f is represented by the binary vector 1100.. ...0.
This dissimilarity of the sign and most significant bits reveals a method for the normalization of case (3). The resultant

fraction f_r is left shifted until the sign bit of the representation is unlike the most significant data bit. This normalization operation may be performed by repetitive applications of a single shift, each followed by a difference test of the sign and data bits. The process terminates at the first time such a difference is detected. A mapping unit (or any multiposition shift element) may be used to perform normalization in a single operation, provided that the required shift distance is known. A "shift detector" may be used to generate such a shift length indication from an examination of the register containing the fraction f_r.
The logical design for such a shift detector is presented in Fig. 5.1. The AND gates A_1 and A_2 and the OR gate O_1, produce a signal equivalent to the logical difference of two bits B_1 and B_{i+1}. If no higher order differences have been found, this signal is applied via gate A_3 to a diode encoding matrix, which produces signals on the lines L_{1-6} to indicate the appropriate length shift. The logical difference signal is also applied to the lower order circuitry via O_2 to suppress the injection of less significant difference signals.
If this logical operation is applied to a fraction which yields -1/2 on normalization when radix complements are used for negative

number representation, an incorrect result is obtained. Fractions which yield the anomalous result are of the form

111....1100....0,

a configuration which is simply detected by hardware. Fig 5.2 shows a logical design for the radix complement shift detector.

These circuits are in logical form only, and implementation of them may require modifications of these designs to adapt them to hardware characteristics.

Example 2.

Much attention has been given to improving the efficiency of the multiplication operation [5],[6],[7].

The classical method of binary multiplication within a digital computer operates on a principle of examining of a multiplier bit, adding of the multiplicand to a partial product if this multiplier bit is a one, a shifting of both multiplier and partial product one place to the right. This process cycles until all multiplier bits have been examined. In an n bit computer therefore, n inspections, n shifts and an average of n/2 additions are performed in each multiplication. To improve the efficiency of the process, two approaches are available. The first consists of speeding the operation of addition by the provision of carry save adders,[8], stacked adders [5] etc.,in an effort to decrease total addition time, which is a major fraction of the total multiply time.

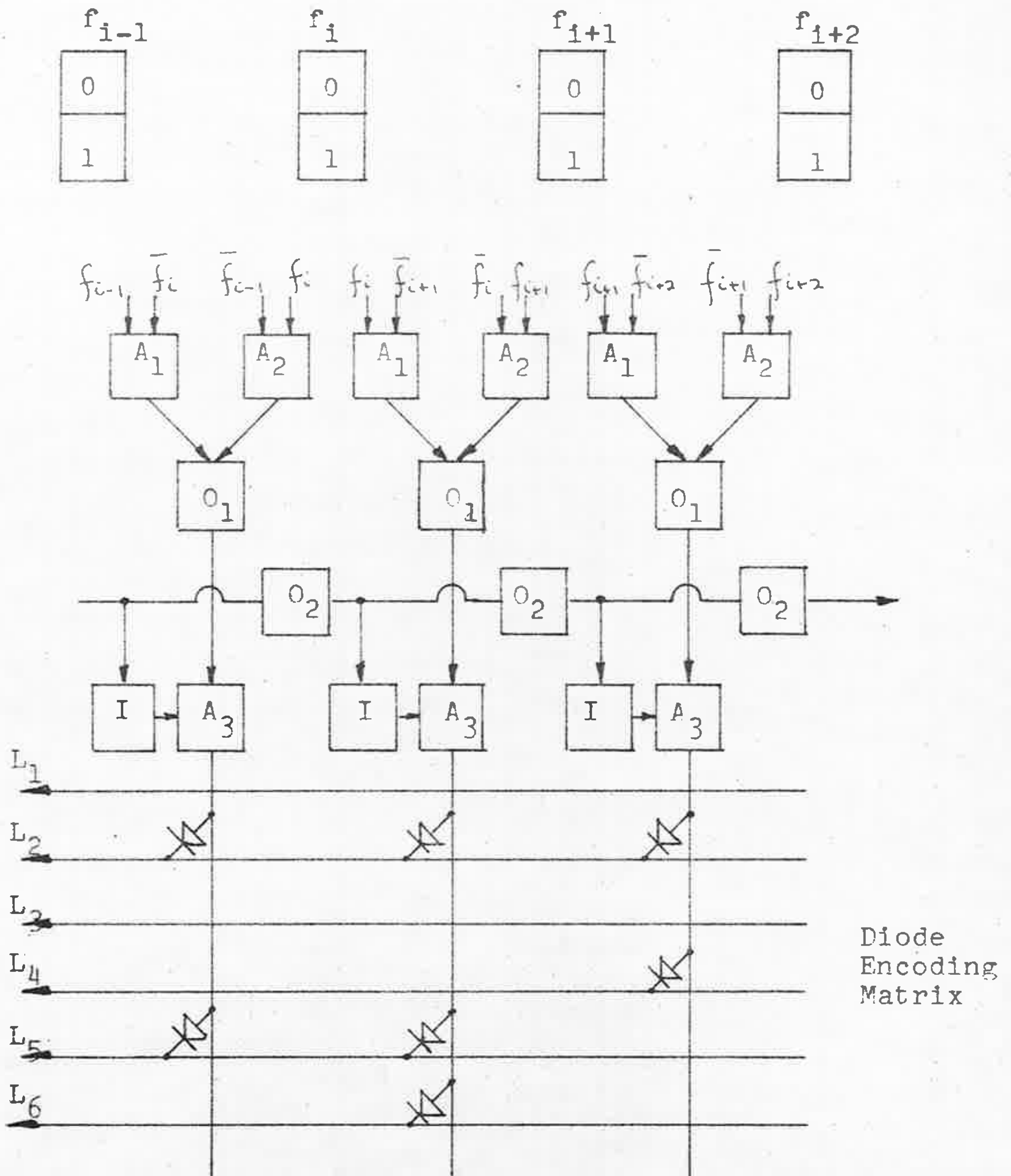
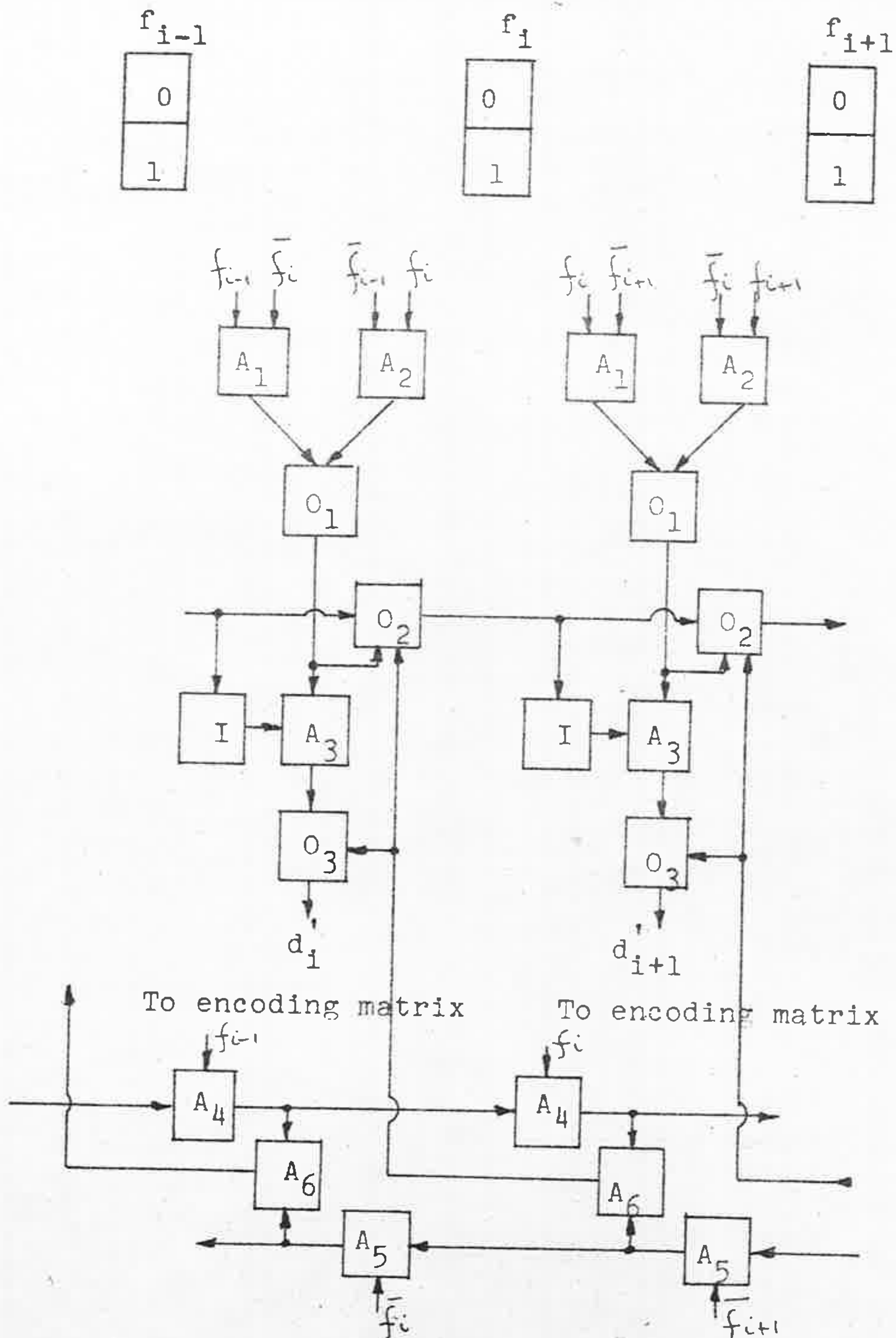


Figure 5.1. A Shift Detector for Use in Normalization.



This circuit uses an encoding matrix identical to the Shift Detector of figure 5.1. This matrix is not shown here.

Figure 5.2. Shift Detector for Radix Complements.

The second approach [10] is to reduce the number of additions taking place by the application of the identity

$$\sum_{i=j}^{k} 2^{i} = 2^{k+1} - 2^{j} \qquad [5.13]$$

This process, although it necessitates the provision of addition and subtraction facilities for forming the partial products, enables the multiplier to neglect "strings" of consecutive digits of the same value in a multiplier. For example, the multiplier

01011110001001110000

may be recoded,

10 (-1)000(-1)001(-1)0100(-1)000

or more efficiently by ignoring isolated 1's,

10(-1)000(-1)00010100(-1)000

where a 1 signifies an addition of the multiplicand to the partial product, (-1) signifies a subtract of the multiplicand from the partial product and 0 signifies no change to the partial product.

A theoretical study of the frequency of operations in such a system gives an average value of n/3 [9] additions and subtractions per multiplication operation. The multiplier is of course not physically recoded,but a suitable control effects the required sequence of additions and subtractions. An integral part of this control may be a shifting element driven by a circuit interrogating the multiplier for bit strings. Upon encountering a string, the control performs the appropriate function, interrogates a circuit similar to the shift detector of the previous example for

the required length of shift, updates a bit counter, performs the shift on the partial product and multiplier, and cycles. The mapping unit may play an effective part in providing a low cost multi-position shifting element.

(2) The reduction of floating point formats to fixed point quantities, and the re-combination of fixed point quantities into floating point format is generally accomplished by the provision of fixed, special purpose register interconnections. These interconnections in effect provide a small number of mapping functions specifically tailored to the machine structure and the floating point format in use. The fixed interconnection system suffers from inflexibility in that the use of non-standard formats at machine language level is effectively prohibited. (Interpretive programming can be used with non-standard formats, resulting in the degradation of machine performance by at least an order of magnitude, See below). The provision of a mapping unit ensures flexibility in the choice of floating point formats that can be employed at any time within the computer. Although this feature may not alone be sufficient to justify the inclusion of a mapping unit in a computer, it does re-inforce the case for inclusion and provides excellent flexibility in the choice of operating formats at any time. (The question of format selection and flexibility is discussed below in Section 6).

These two examples show the general fields of application of the mapping unit within arithmetic micro-programs. It provides a degree of flexibility in shifting functions and in format resolution, at low cost.

5.3. Shifting Operations.

As we have already noted in the previous section, the mapping unit may serve as an extremely flexible shifting unit. The provision of sufficient maps within the mapping unit allows shift operations of all lengths to be performed in the same time, in contrast to a large class of computers wherein longer shift operations are executed as a series of shorter shifts. As an example, Figure 5.3 presents a summary of CIRRUS shifting operations, both with and without the inclusion of a 36x18 bit mapping unit.

In the existing structure, i.e. without the mapping unit, shift times are proportional to the distance shifted, which is a consequence of the fact that the micro-structure of the existing CIRRUS contains provision only for single place shifts. The augmented structure, i.e. including the mapping unit, performs the shifts as a mapping operation regardless of shift length. Micro-programs and sample mapping data for both the existing and augmented CIRRUS structures are given in Appendix 5.

5.4. Branching Operations.

The mapping unit does not assist greatly in the execution of branching operations, although "Branch on Bit" facilities may be provided within the bit handling system. (See Section 5.5 below).

5.5 Bit and Character Handling Operations.

Provided that the computer addressing system has sufficient flexibility to reach below the word level, a flexible sub-word (either bit or character) programming system can be implemented. The more significant part of an address is used as a store address to retrieve one word of data. The less significant part of a word can then be used as a map address to mask and shift the desired bit (or character) out of the word into a standard format. Figure 5.4 shows a possible character operation within the CIRRUS augmented structure,

The CIRRUS system is not well adapted for this mode of operation since the binary addressing scheme does not fit well with packing six 6-bit characters to a memory word. Similarly, the bit addressing system using binary addressing and 36-bit words is wasteful of address space and address arithmetic becomes rather complex. Rather more efficient in programming terms, (but wasteful of main store space) is to allow only 32-bits (2^5) to be addressed in any word, and pack four 8-bit characters to a word. (See Figure 5.5).

5.6. Masking and Shifting Instructions.

Machine instructions for the purpose of masking (logical AND function) and shifting data within computing structures are standard facilities on almost all computers at the present time. The provision of a mapping unit makes available a wide range of combined masking and shifting operations within the micro-structure of the computer. By appropriately linking control of the mapping unit to the programmer via a special machine instruction, the flexibility of the mapping unit is available to the machine language

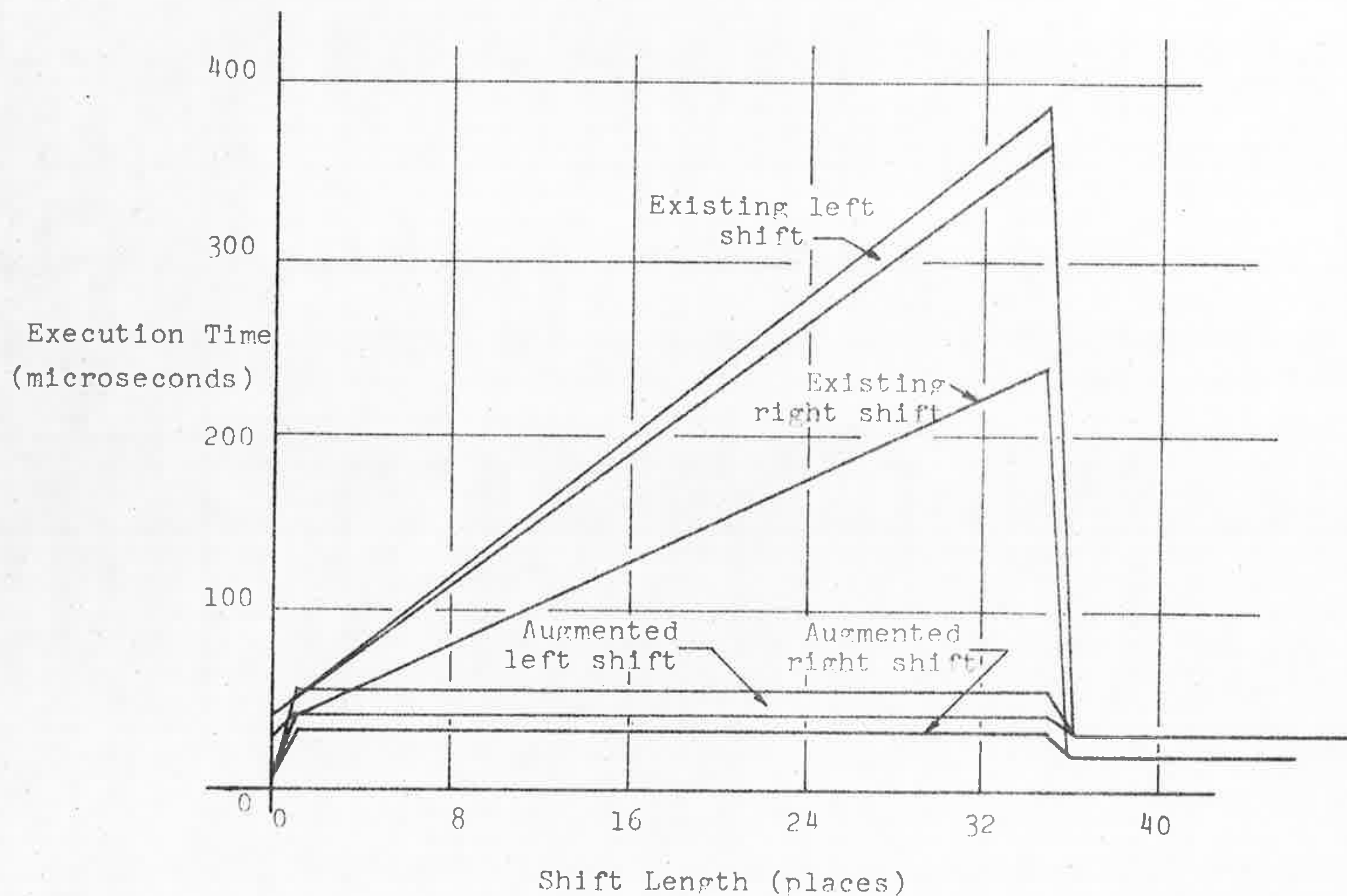


Figure 5.3. The Timing of Shift Operations in CIRRUS.

programmer and to the general user. Such an instruction for the augmented CIRRUS structure is detailed in Appendix 6.

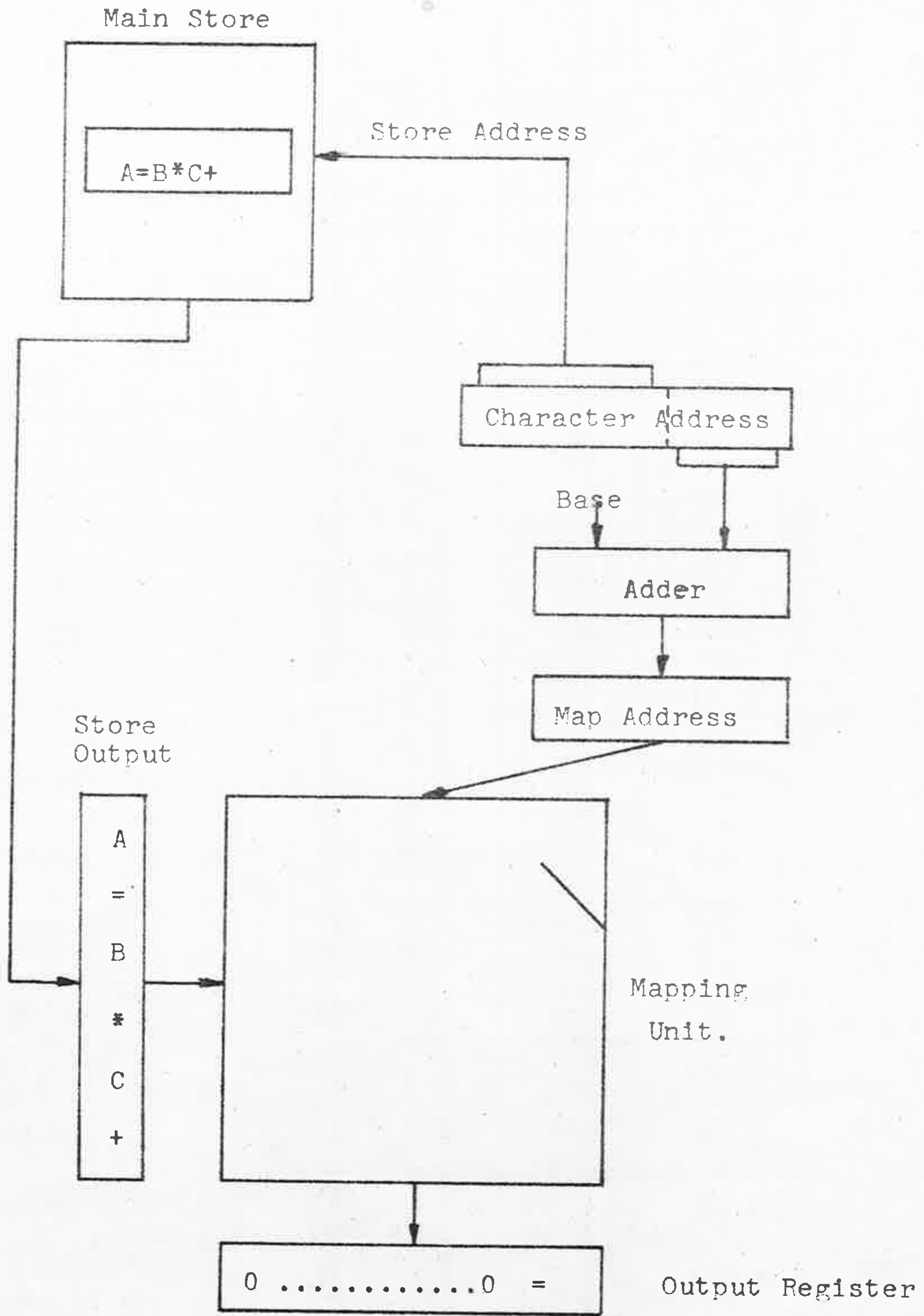


Figure 5.4. Character Handling with the Mapping Unit, Six 6-bit Characters per Word.

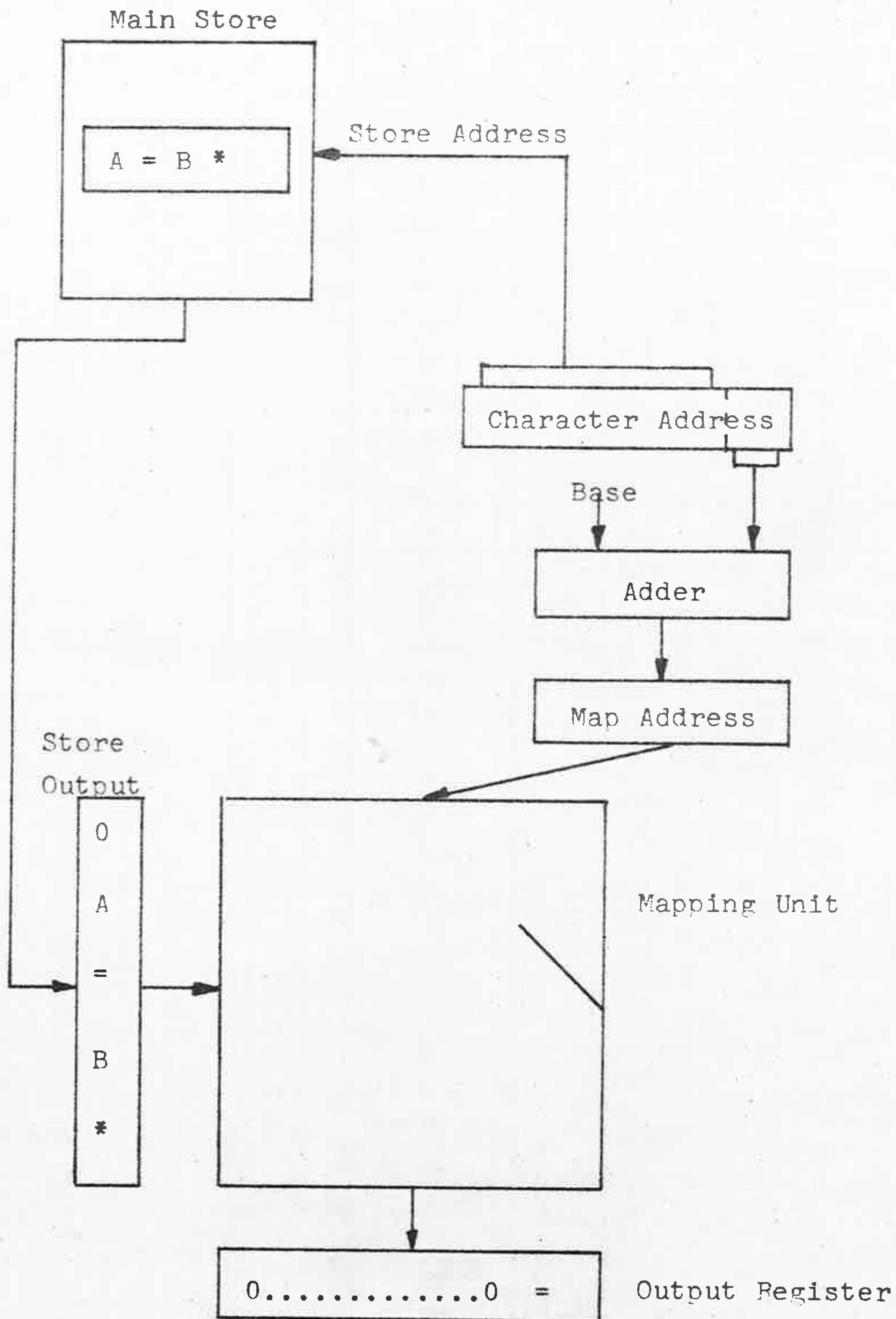


Figure 5.5. Character Handling with the Mapping Unit, Four 8-bit characters per Word.

SECTION 6.

Mapping Units and the Provision of Non-Standard Data Forms and Operations.

6.1. Standard Data Forms.

Most present day computers offer a set of internal data representation which fall into four classes, viz, fixed point arithmetic representations, floating point arithmetic respresentations, alpha numeric character representations and boolean valued variable respresentations. Other special purpose representations are sometimes included. In general, the parallel binary computer will use only one definite format for each class of representation and the provision of further distinct formats is limited by the inability of instructions applying to each class of representations to accommodate themselves to alternate formats.

6.2. Interpretive Programming.

A technique of introducing non-standard formats and operations on the information represented in these formats, into the computer is well known as interpretive programming. Basically information in the non-standard representation is broken down by standard machine operations into a number of pieces of information, each represented in a standard format. Standard machine operations then perform the required information upon the components and the result is re-assembled into the non-standard format. By calling each operation on non-standard information via a sub-routine, a relatively simple programming technique is obtained. However, the system suffers from inefficiency since each non-standard operation is performed by a number of standard machine instructions.

This inefficiency may be tolerated if it leads to a decrease in hardware cost. For example, on smaller computers, floating point hardware may be omitted

and floating point operations performed as a series of fixed point operations in order to lower hardware costs. Infrequently used formats, e.g. double precision, complex arithmetic etc. are normally provided as interpretively programmed software features on nearly all "scientific" computers.

6.3. Provision of Alternate Formats.

At the micro-level of operations within a computer, i.e. generally at the hardware level, operations are performed on essentially fixed point quantities during arithmetic functions and boolean vectors during logical and character operations. These "micro-forms" do not necessarily bear a simple correspondence to the formats of data representations used at the machine language level. For example, a floating point number in CIRRUS consists of fraction and interger packed together within one machine language word of 36 bits. However, during micro-operations upon this floating point representation,the exponent and fraction are separated to enable fixed point operations to be performed upon both parts. (See Fig 6.1.).

The provision of an extended set of machine language data formats reduces to a problem of providing an appropriate set of transformations within the hardware, which will reduce the information contained in the machine language level format to an equivalent set of quantities in micro-format representations. This limited range of micro-formats may then be manipulated by existing micro-operations sequences to yield a desired result and this result may then be re-formed into the non-standard format being used.

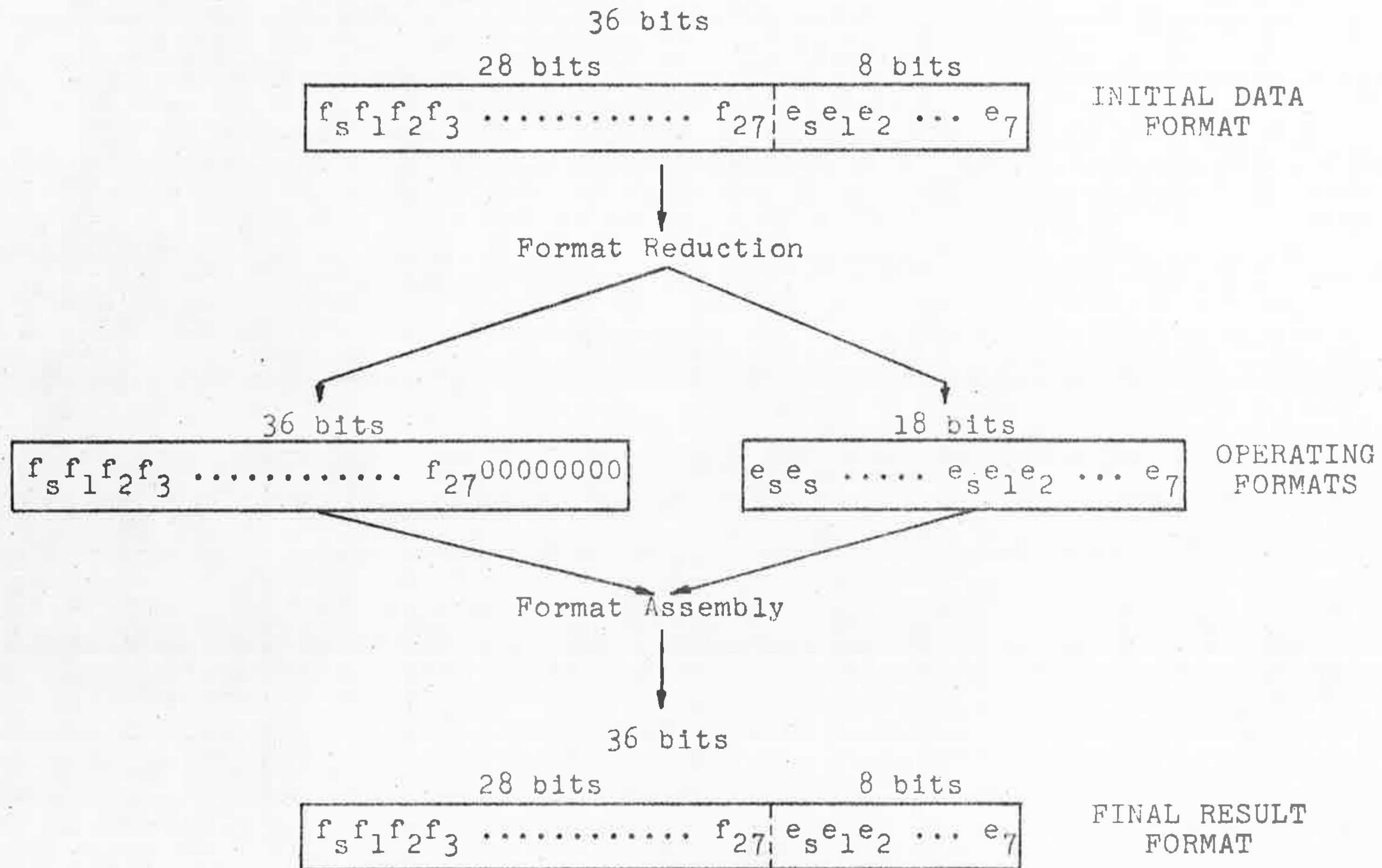


Figure 6.1. Reduction and Assembly of CIRRUS Floating Point Format.

The transformation of machine language formats to micro-formats and vice versa is at present normally performed by a set of special purpose register interconnections performing masking and' shifting functions to isolate or prepare various micro-formatted quantities from a machine language format. This technique is very efficient for any given format, but the provision of a range of machine language formats requires a range of transformation interconnections, one or more for each format, which leads to unwieldy hardware implementations.

If a set of alternative machine language formats are provided within the computer, two methods are available for transformation generation at the micro-level:

(a) A set of special purpose register interconnections can be implemented to effect micro-format separation and conjunction.

(b) Transformations between formats may be accomplished by a combination of logical AND, OR and shifting micro-operations.

Method (b) suffers from lack of speed particularly when operating upon complex formats. Method (a) can be implemented by direct connections which increases the complexity of both the hardware and the control of the computer.

By using a mapping unit as the format converter, high speed, excellent flexibility and simple control of the computer are obtained.

6.4. Provision of Non-standard Operations.

The provision of non-standard operations at machine language level on both standard and non-standard formats may be accomplished by interpretive

programming as described above. However, the low computing rate of this form of program may render desirable the provision of non-standard operations at the machine language level in an effort to improve the computing rate.

An analysis of the required operation, together with the data formats on which it performs, will establish a micro-program which will effect the required data modifications. The efficiency of this micro-program will depend upon a number of factors, chief of which revolves about the suitability of the set of micro-operations available to implement the required operation.

6.5. An example-Quarter-Word Operation in CIRRUS.

Some time after CIRRUS was commissioned, two direct-view storage oscilloscopes were added to the operating consoles for the display of operating messages and graphical data. The active area of each of these displays consists of a square array of 512x512 elements, any of which can be intensified by transmitting its co-ordinates from the C.P.U. to the 'scope. In the interests of storage efficiency within the computer, it has been proposed that co-ordinate pairs be packed into a CIRRUS half-word of 18 bits. Each co-ordinate consists of 9 bits and occupies a CIRRUS machine language "quarter" word. Since CIRRUS operates on 18 bit fixed point quantities at the hardware (and micro-code) level, arithmetic operations on individual co-ordinates are complicated by

(a) The position of the co-ordinate within the CIRRUS word,

and(b) The presence of one other 9 bit co-ordinate within the half-word format, creating dangers of its distortion and/or obliteration.

A total of eleven new operation types have been designed to facilitate 9 bit integer arithmetic. Table 6.1 presents a summary of these operations, which have been written for the CIRRUS structure augmented by a 36x18 mapping unit (see Appendix 2). For comparison, a set of interpretive routines were written to perform the same functions in the existing CIRRUS structure. Table 6.2 presents a comparison of the execution times of each type of operation. A very considerable saving in time results from the use of the mapping unit to provide format transformations. Appendix 7 gives some listings of typical forms of both the micro-programs and interpretive sub-routines.

Notes.

(1) The CIRRUS machine language format is:
(variant) (function) (y-register)(x-address)

v	fn	y	x

(2) The data format for the quarter-word representation is

y	a	b	c	d
x	A	B	C	D

(3) Negative quantities are represented in two's complements.

Function	Variant	Operation
PYQ	U	A →a
	X	B →b
	Y	C →c
	L	D→ d
NYQ	N	-A →a
	X	-B →b
	Y	-C →c
	L	-D →d
PXQ	U	a →A
	X	b →B
	Y	c →C
	L	d →D
RPQ	U	a →A,B,C,D
	X	b →A,B,C,D
	Y	c →A,B,C,D,
	L	d →A,B,C,D
SYQ	U	A+a→a, B+b→b
	X	C+a→a, D+b→b
	Y	A+c→c, B+d→d
	L	C+c→c, D+d→d

Figure 6.1. Quarter Length Arithmetic Instructions.

Function	Variant	Operation
DYQ	U	A-a→a, B-b→b
	X	C-a→a, D-b→b
	Y	A-c→c, B-d→d
	L	C-c→c, D-d→d
RYQ	U	a-A→a, b-B→b
	X	a-C→a, b-D→b
	Y	c-A→c, d-B→d
	L	c-C→c, d-D→d
MPQ	U	axA→a, bxB→b
	X	axC→a, bxD→b
	Y	cxA→c, dxB→d
	L	cxC→c, dxD→d
PTQ	U	Branch to x if a positive
	X	Branch to x if b positive
	Y	Branch to x if c positive
	L	Branch to x if d positive
NTQ	U	Branch to x if a negative
	X	Branch to x if b negative
	Y	Branch to x if c negative
	L	Branch to x if d negative
ZTQ	U	Branch to x if a zero
	X	Branch to x if b zero
	Y	Branch to x if c zero
	L	Branch to x if d zero

Table 6.1 cont. Quarter Length Arithmetic Instructions.

Function	Execution times (micro-seconds)	
	Micro-operations with mapping unit.*	Interpretively programmed.
PYQ	49.5	580
NYQ	52.0	700
PXQ	49.5	610
RPQ	46.5	1780
SYQ	57.5	3160
DYQ	84.0	3160
RYQ	84.0	3240
MPQ	129.0	3510
PTQ	52.5	670
NTQ	51.0	520
ZTQ	51.0	560

*N.B. The operations represented by these figures include one instruction extraction, modification and interpretation cycle.
The execution times given are averages over all legal variants.

Table 6.2. Comparison of Execution Times of Quarter Length Instructions.

SECTION 7.

Inter-Computer Compatibility.

7.1. Requirements and Definitions.

During the last decade, the electronic digital computer has been accepted as a useful tool in a large number of areas. This acceptance has been achieved principally by the continuing improvement of both hardware technology and software support facilities, resulting in reliability and cost/computation figures two of three orders of magnitude better than those obtainable ten years ago. This increased acceptance of computers has resulted in a large number of new installations being commissioned in both the scientific and commercial fields. In addition, a large number of existing installations have been re-equipped because of obsolescence or increased work load.

This situation has created two factors which are of some importance:-

(a) A large amount of programming effort is being expended by individual uses in duplicating already existing programs. In addition, many programs already existing for one type of computer have to be translated, either mechanically or manually, in order that they may run on other machine types.

(b) When an installation is re-equipped, it is commonly desirable, in the first instance at least, to maintain some of the software facilities and operating procedures used by the computer being replaced. This means that all such software and production programs must be re-written, an expensive and time consuming operation, or alternatively the new computer is restricted to those

machines which are compatible with the replaced computer.

A third factor is likely to be important in the near future, when multi-user, multi-computer networks, connected via digital transmission links will become common. These systems envisage a "dynamic" execution scheme, wherein individual jobs are routed to idle computers. Any user cannot therefore predict which processor is actually performing his computation, and hence his program must be executable on all computers in the system, i.e. make no reference, implicit or explicit, to machine structure, unless either he is able to call for execution on a particular computer, whereupon several of the advantages of a multi-computer system are lost, or all computers in the system are compatible with one another.

It is the purpose of this section of the thesis to study the problems arising in the provision of compatibility between two computers. In particular, some of the problems arising in a machine code compatibility scheme will be investigated.

Some terms used within this section are now defined

If all valid programs, whose operations are independent of their execution time and speed, yield identical results when run on two basically different computers, then those computers are said to be compatible.

The "source" computer system, consisting of the source hardware and the source software, is that system whose cahracteristics are to be implemented within some alien computer system. This alien system is called the "host" system. The "host" hardware may be run under two software systems:-

(a) the normal host software, which is that

software designed for the host hardware without any compatibility feature as a design criterion. This system is ideally the most efficient method of operating the host hardware.

(b) the object software, which is designed to render the host and source computer systems compatible. The host hardware plus the object software is known as the object system.

Obviously there are a large number of object software systems of varying efficiencies for any given host hardware and source computer.

In all situations considered hereafter, it is assumed that the host computer contains at least sufficient storage capacity and at least the required number and types of input/output units to render compatibility with the source system possible.

7.2. Compatibility via High Level Languages.

Procedure oriented languages (P.O.L.) now in common use, e.g. FØRTRAN, ALGØL, CØBØL etc., may be used to achieve compatibility between computer systems. Any program written in the P.O.L. may be compiled (i.e. translated to machine language) by two computers, and within the limitations of the word length, input/output formats and types and compiler equivalence, identical results should be obtained. This technique is suitable for programs where no reference is made to machine structures, i.e. no machine language sun-routines or functions are included in the program. References to any machine language will invalidate this procedure.

Higher level languages, e.g. problem oriented

languages such as STRESS, COGO etc., and data structured languages such as LISP, FØRMAC etc., can be used in a similar manner.

7.3. Machine Language Compatibility.

Situations may arise where P.O.L. compatibility is insufficient. Often, when a program is written in assembly code for reasons of speed and/or efficiency, the investment of time and money in such a program is of a higher order than in a P.O.L. program, and it is desirable that such programs be transferrable to other machines.

This transferral may be effected by two basic techniques.

7.3.1. Translation of Machine Instructions.

The original program may be re-programmed using the philosophies and structures of the original program but using host machine instructions. This procedure may be relatively straight forward if data structures and operational techniques can be preserved, but will become increasingly complex as basic differences (e.g word lengths, machine instruction sets, operating modes etc.) develop between computers.

Some work has been devoted to mechanizing the translation of machine languages [21].

If the host and source systems are of similar structure, it may be possible to translate the majority of machine instructions between the source and object programs on a 1:1 basis. For those source machine instructions which have no host equivalents, a set of host machine instructions having an equivalent function may be defined, i.e. 1:n instruction equivalences are

used. The translator technique can also be made to evaluate n:1 instruction equivalence situations in the interests of improved object execution efficiency. As the number of 1:n translations increases, obviously the computing rate of the object program is decreased, and if alternate methods exist for performing a similar function in fewer instructions, the efficiency of the object program also decreases.

For maximum efficiency it is therefore desirable that the source and host machine language sets be closely related.

If data formats are markedly different in the source and host computers (e.g. parallel computers with different word lengths, character and parallel machines, etc.) care must be taken to rationalize data operations involving part words. For example, in the construction of symbol tables by compilers, many different parameters may be packed into a single word to improve storage efficiency. The rationalization of such generic differences may be beyond the scope of a mechanical translator.

7.3.2 Simulation of the Source System.

Compatibility between the source and object systems may be achieved by simulating the source computer upon the host hardware. The host computer will then accept source machine language instructions and data forms and execute the source program under the control of the simulator. The effectiveness of such a technique will depend upon the method of implementation of the simulator within the host hardware.

If any form of interpretive machine language programming is used, a heavy penalty is paid in terms of the object system computing speed. As an example,

consider the well adjusted quarter length interpretive operations outlined in Section 6. We may consider another CIRRUS provided with these quarter length operations as the hypothetical source computer, and the existing CIRRUS computer as the host hardware. The source data formats used are well suited to the simulating computer, and little or no interpretation of instruction as such is needed. The execution times quoted represent an average figure of 34 machine language operations in the host hardware for each source instruction executed in the simulator. This figure will increase if instruction extraction, modification and interpretation sequences are included in the simulator.

Even for better matching host and source computers, a lower limit of the order of some tens of host instructions executed per source instruction executed , is encountered. For example, the implementation of a "trace" routine (a device for following program execution paths and detecting intermediate results) is commonly provided on most computers and takes the form of a very simple simulator of a computer upon itself, and utilizes the equivalence of the host and source instruction sets. A trace routine written for CIRRUS performs essentially 25-30 instructions in the trace loop for each program instruction executed. Printing options may significantly increase this figure.

An alternative form of simulator is available on the machines which are controlled via modifiable micro-program stores. Generally, the micro-program store takes the form of fixed or read only storage which is read once per machine cycle to provided operational timing patterns and functional unit

interconnections for that cycle. Fixed storage is electronically unmodifiable, but in many hardware forms of the unit, physical modification of the stored information is possible.

By modifying the set of micro-programs for a computer, different operational sequences may be formed which correspond to new machine instructions. Therefore, by re-designing the micro-programs within the fixed store to conform to the source computer specifications, it is possible to provide machine language compatibility with the cource computer in the host hardware.

This compatibility is essentially a one-way compatibility from the source system to the host system. It is reasonable to expect that the ratio of the computations rates of the host hardware under the micro-programmed simulator and the normal host software will be greater than unity, but at least an order of magnitude better than the interpretive programming rates discussed above. (Note that if the ratio of the rates is less than unity, the object system is a "better" system than the originally designed system, and there is no advantage in writing further programs in the host system).

In the following sub-sections, we will discuss some of the problems associated with the implementation of a micro-programmed simulator, and examples will be drawn from a study of the implementation of a commercial system, the IBM System/360, within the CIRRUS computer.

7.4. Micro-programmed Simulators.

7.4.1. Depth of Simulation.

The range of simulation techniques for achieving compatibility is extensive. One extreme is implicit in the execution of the same procedure oriented language program by two different computers. The computers have been arranged for the purposes of this computation as a set of named data locations and a time ordered set of transfers and modifications have been specified on both computers to achieve a desired result. The two computers are functionally equivalent or compatible whilst operating on this P.O.L. program. At the other end of the range, we can consider two computers operating upon identical programs with identical hardware. This is a trivial situation; the computers are either identical (an original and a copy), or one computer structure contains the other as a sub-set.

Micro-programmed simulators lie somewhere between these two extremes, and it is important to determine the depth to which the simulator must be "aware" of the detailed operations of the source computer. It is of course essential to provide adequate storage and peripheral units, but the actual degree to which the detailed operation of these units and the detailed operation of source arithmetic processes must be simulated will have a large effect upon the efficiency and effectiveness of the compatible computer.

The realization that the logical structure of a computer as seen by a programmer, and the engineer's hardware structure implementing this logical structure, are two separate entities [14] allows a definition of

the level of simulation to be provided in a compatible computer. Namely, all logical features visible at the desired compatibility level must be provided by the simulator if the simulation is to be functionally complete. A feature is visible if its value is possibly necessary in the execution of two or more distinct operations. The value associated with such a feature must be retained between two distinct (and therefore time separated) operations, and it may be used by a third operation interposed between the original operations. At the machine language level, visible features are storage locations, index registers, sequence counters, general purpose program registers etc. Hardware registers holding partial results used only within a single machine instruction are invisible, and need not be provided in the simulator.

7.4.2. Internal Formats in a Micro-programmed Simulator.

When implementing a micro-programmed simulator to effect compatibility between the source and the host computers, several problems occur which are connected with formats. Firstly, the two computers may be of basically differing types e.g. the source computer may operate as a character (serial) machine and the host computer may be a parallel binary machine, etc. Secondly, even if both computers are parallel, differing words between the computers will involve some difficulties. Thirdly, formats and codes used by each machine may differ widely, particularly in data representation.

These problems may be resolved into three general areas.

Storage.

The efficiency and ease of storage of source

format words within the host memory depends entirely upon the relative word lengths of the source and host machines. For example, if 100% memory utilization is to be achieved with dissimilar word lengths, an involved source address to equivalent host address transformation is required since source word boundaries will almost certainly overlap the host word boundaries (see Fig. 7.1a). If overlap does occur, then multiple accesses to host storage must occur and the words so obtained must be masked, shifted and merged to obtain the source memory word requested.

If A_s is the required source address,
s is the source system word length,
h is the host system word length,
and $[x]$ represents the integral part of x, then the required host address A_h containing at least the first bit of A_s is given by

$$A_h = s.\ [A_s/h] + [(A_s - h.[A_s/h])\ .(s/h)] \qquad [7.1]$$

and a further k memory access must be made to ensure that the last bit of A_s is obtained, where k satisfies

$$(A_s+1)\ .\ s < (A_h+1+k)\ .\ h \qquad [7.2]$$

If the values of h and s are such that these equations do not reduce to simple operations, such as shifting and masking, it is doubtful that this addressing problem can be effectively solved as it stands. For example, if $s = 36$, $h=48$, equation 7.1 becomes

$$A_h = 36 \times [A_s/48] + [(A_s - 48\ [A_s/48]) \times 3/4]$$

and division by 48 is difficult since it is incommensurate with any power of 2.

An alternative method which sacrifices storage efficiency for addressing simplicity, is to use an extended source word within the host hardware. By

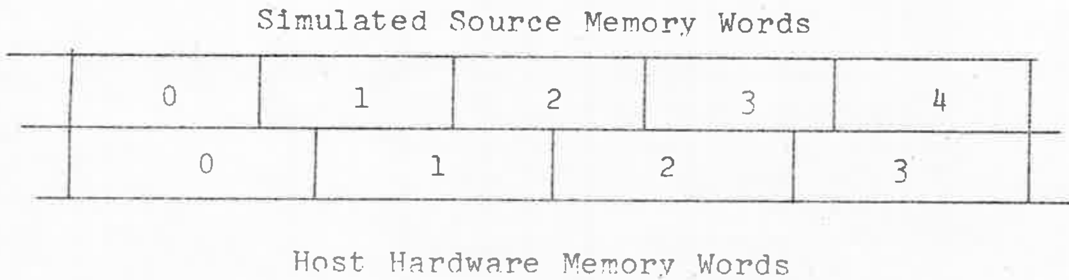


Figure 7.1a. Storage Word Overlap for $h \neq 2^n.s$

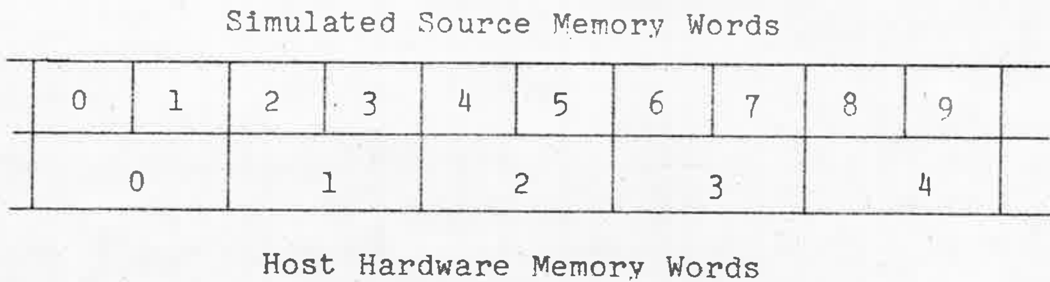


Figure 7.1b. Storage overlap for $h = 2^n.s$

selecting an extended source length which is related by a power of 2 to the host word length, simple address decoding becomes possible (see Fig.7.1b). In the above example, the simulated source word length would be increased to 48 bits so that

$$A_h = A_s$$

but the storage efficiency falls to 75%.

In cases where $s < h/2$, the storage of two or more source computer words per host word becomes feasible. The extraction and insertion of individual words from host words then involves masking and shifting functions, and the inclusion of a mapping unit substantially simplifies these operations.

In the micro-simulation of a character-serial machine upon a parallel computer, addressing and extraction problems are substantially similar to the case of $s < h/2$. For optimum addressing efficiency, the number of source characters per host word should be a power of 2. If this is not so, the injection of "waste" bits as described above, will decrease the storage efficiency, but retains a simple addressing scheme.

Arithmetic.

The efficient simulation of source arithmetic operations within the host hardware depends to a large extent upon the ratio of word lengths of the source and host computers. If $s < h$, arithmetic operations are comparatively straight forward. The source words can be extended to the length of host words by the injection of appropriate bits at the higher (integral) of lower (fractional) ends of words, whereupon normal host arithmetic can be performed, if due regard is paid to overflow positions etc. Source formats may be re-generated prior to the return of results to store.

However, if s>h, i.e. a source word occupies more than one host word, difficulties may arise. In particular, the host hardware must perform operations upon part length source words. The provision of part length arithmetic (with respect to the source system) introduces problems of partial result generation and storage (see Fig 7.2). Both partial result and storage will tend to slow the operation of the micro-programmed simulator. The necessity of generating a number of partial results means increased computing time is required for any operations and the formation of partial results implies that they must be held available to the computer for sunsequent processing. As most computers are supplied with only a minimum amount of fast hardware storage (static registers), these partial results may have to be stored within some form of local memory for a time. The operation times of storing and retrieving partial results will add to the overhead of the simulator, reducing its computing rate.

The data formats used within the source x system will have to be resolved into entities that the host hardware can meaningfully operate upon. Format resolution has already been discussed in Section 6. The provision of a mapping unit results in a flexible data transformation ability within the host hardware, enabling rapid re-formatting of simulated source data formats.

However, one inescapable problem may arise in the arithmetic section of a compatibility study. It occurs in the representations of negative numbers where two forms are in common use, viz, radix complements (2's complements) and diminished radix complements (1's complements). Differentiation between these systems

n n

a b $N_1 = a.2^n + b$

×

c d $N_2 = c.2^n + d$

b.d

+

a.d

+

b.c

+

a.c

$N_1.N_2$

$$N_1.N_2 = a.c.2^{2n} + (a.d + b.c).2^n + b.d$$

Figure 7.2. Execution of a Multiplication by Partial Length Arithmetic.

normally exists within hardware entities below the level of micro-program control, principally in the logical designs of the adder(s) and complementing circuitry. As the form of negative number representation is visible at the machine language level, storage must occur in the source representation if compatibility is to be provided, but arithmetic must be performed in the host representation if the arithmetic capability of the host is to be utilized. Therefore, a conversion from one form to the other must occur at each extraction and insertion of data from the memory. Fortunately, one's and two's complements are closely related and can be formed by a single addition or subtraction in either type of machine. However, the extra operations at each simulated arithmetic operation will result in a general slowing of the micro-programmed simulator.

All other arithmetic operations and facilities (overflow, zero detection etc.) are relatively easily handled. Some operations may require rather lengthy simulations in the host of functions handled by hardware in the source computer. Such simulations are generally straight forward, but time consuming. (see below).

Instructions and Control.

Instruction formats will normally differ between source and host computers. Besides differences in word length, there will generally be differences in functions associated with groups of bits, and it is likely that there will be need for some form of re-arrangement of bits within an instruction word. The mapping unit will provide a flexible, high speed re-arrangement facility to match the source specifications

to the host functions.

The simulation of one-address, two-address, three-address etc. source computers in a computer of a different type presents few problems, since the concept of the number of addresses in an instruction is introduced via micro-program to the set of machine instruction specifications. There is no specific provision of a number of addresses per instruction at the hardware level.

Some computers of course may have two (or more) stores (main store and local store, main store and a set of accumulators etc.) when they can be referred to 1+1 address computers. Should simulation of a 1+1 address machine be attempted on a 1 address machine, some logical division of the main store is necessary to simulate the second store at the machine language level. Relative addressing techniques for the store may be used to solve such a problem, although a time penalty must be accepted at each simulated store operation.

7.4.3. External Formats in a Micro-Programmed Simulator.

Here we will discuss some problems arising in the use and control of input-output units. Magnetic tape will be the medium commonly discussed since both source and host systems are likely to be provided with magnetic tape units, and this medium poses most of the problems encountered in considering of compatible peripheral operations.

We will discuss three main topics: coding, external formats and channel supervision and control.

Codes.

Codes to represent alpha-numeric characters, despite attempts at standardization, still vary to some

extent. Should the source and host codes for a particular peripheral unit differ, some form of code conversion must be provided within the host system. For example, if the codes used by the source and host line printers are different, the micro-programmed simulator, which is using source software and formats, will produce characters for the line printer in source code. However, these characters are destined for the host's line printer which uses the original host system coding. The printed characters will be invalid.

The timing of the code conversion is difficult. Typical general purpose channels have no variable code conversion capability. Code conversion should therefore occur as the information for output is being stored, since this is the last time the computer "sees" the information. However, this cannot be done at machine language level if compatibility constraints are observed, nor can it be done at micro-program level during the storage instruction, as not all storage instructions are associated with information for subsequent output. There are three alternative methods available to solve the problem. Firstly, code modifications may be made to host peripherals. This may be difficult as codes can be deeply embedded into the peripheral unit and its controller. It also deprives the host system of its normal codes. Secondly, code changes may be made to the source software systems used on the simulator. This will satisfy the majority of programs using the standard peripheral drivers, but programs using peripherals but by-passing the standard drivers will still not operate on the nominally compatible computer. Thirdly, for output operations, the peripheral activation instruction can be used to trigger the action of a code conversion routine. The

peripheral will not be activated until this routine is completed. On input the operation termination interrupt will produce an equivalent result.

The precise form of code conversion, which is triggered by this method, is dependent to some extent upon the type of peripheral associated with the data. For example, on output, line printers convert electronically signalled data (source code) to a visual impression (host code). The impression produced corresponds to the code used. Magnetic tape on the other hand, merely records the data in the signalled form (source code), for re-insertion to the computer at a later time (source code). Only peripherals of the first type require code conversion from the source to the host codes.

External Formats.

Consider the situation arising when the object computer is reading a source computer prepared tape written in binary mode. Information is recorder on the tape in multiples of the number of characters per source language word. As this information is being read, words are being assembled using multiples of the number of characters in the host hardware word. Information passing into the object system is therefore in the form of tightly packed source language words in the host storage. If the word lengths in the two computers differ, some post-read conversion of this format to the standard simulated source format is necessary. This unpacking on input (and corresponding packing on output) is a complex operation, analgous to the memory packing operations discussed in Section 7.4.2. above.

Unfortunately the problem does not yield readily

to any techniques so far discussed, the main reason being that the character by character assembly (or dismantling) is occurring within the magnetic tape controller and is under pure hardware control. This process is "outside" the influence of the computer's micro-program control. The provision of alternate assembly methods in magnetic tape controllers is uneconomic considering the infrequency of the use of such an operation, so that a penalty of packing and unpacking store words during peripheral operations must be accepted. The packing (or unpacking) operation occurs at the commencement (or end) of a peripheral unit operation, and hence, although it occupies a finite period of time it does not affect the peak data transfer rate attainable by the peripheral system.

Channel Supervision and Control.

Most computers are provided with one or more general purpose channels for improving the efficiency of peripheral transfers. These channels generally consist of a single word buffer plus control interrupt and memory access circuitry. An operating channel can transfer data between the memory and a connected peripheral unit without affecting the status of the central processing unit. This process allows computation to occur concurrently with peripheral transfers.

If the source computer contains channels, it will be necessary to provide some form of channel interpreter at the micro-program level, so that references to source channels may be transformed to the appropriate format for operation of the host channels (if any). Such channel parameter transformation is also used to communicate peripheral status and

channel interrupt signals back to the object program.

The simulated source channel will take the form of a set of simulated status and interrupt buffers provided in local store and updated as programs and host channels refer to the simulated channel. Unit destination, associated coding tables and packing information will also be provided in a form suitable for use by the channel simulator.

7.5. Compatibility - An Example.

7.5.1. The Source Computer.

As a specific example of the problems arising in providing compatibility between two computers, we will consider the procedures necessary to provide a micro-programmed simulator for the I.B.M. System/360 [14] within the CIRRUS computer. System/360 was chosen as the source computer because of its wide acceptance, and because detailed logical and structural descriptions [19] are available.

System/360 is provided with a main store, organised as 8 bit bytes, each individually addressable. Sixteen general purpose registers of 32 bits and four floating point registers of 64 bits are provided. A comprehensive order code provides operations on a wide range of data types held in registers and/or storage. A number of general purpose channels are provided, one being of a multiplexer type for slow (character) peripherals. A real time clock, facilities for connection to other computers and various control facilities are provided. System/360 is implemented in a number of hardware configurations ranging in performance over several orders of magnitude. All models in the family are compatible with those members of lesser performance than themselves.

The operation of the System/360 can be logically described by a set of micro-programs running concurrently within the logical structure, under a series of interlocks and interrupts. We will discuss in detail the particular micro-program associated with instruction extraction, modification and interpretation. This process, called "routine flow" has its counterpart in all computers,and since it is executed once per machine instruction executed, its efficiency very largely determines the efficiency of the computer as a whole. This process is termed "system program" and is referred to as "C.P.U." by reference [19].

7.5.2. Memory Simulation.

System/360's main memory is divided into bytes of 8 bits, each byte being individually addressable. In order to provide reasonable storage efficiency together with relative simple address decoding procedures, we have chosen to pack two System/360 bytes into the lower 16 bits of an 18 bit CIRRUS hardware word. The X-store format is shown in Fig.7.3, which results in a storage efficiency of 88.9%.

The CIRRUS Y-store will be used to provide local storage for the simulated forms of the various compatibility logical entities defined within System/360 which are necessary for compatibility. In addition to the general purpose and floating point registers (whose simulated forms occupy 48 words) the Y-store will hold all machine indicators, instruction registers, decoded addresses etc. used in the specification of the System/360's operation.

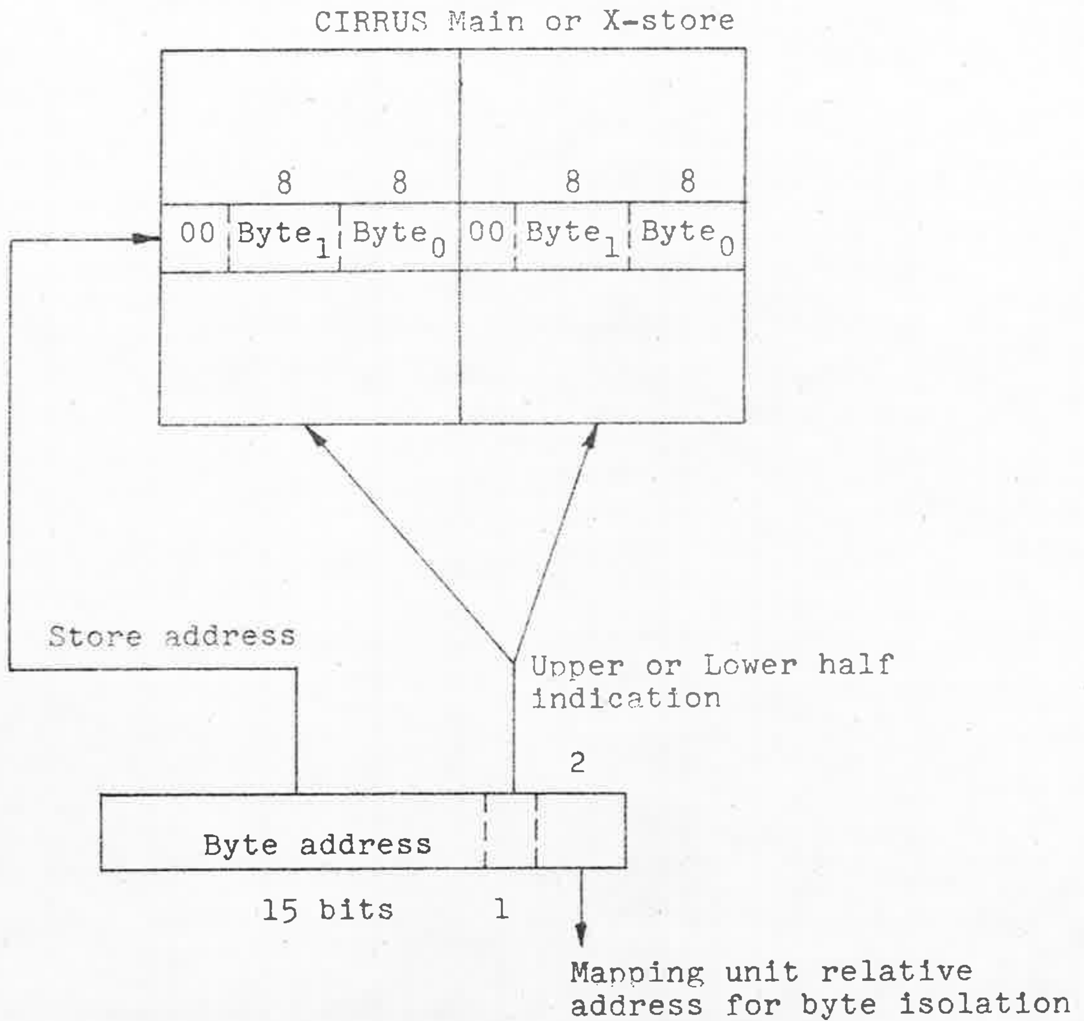


Figure 7.3. Simulated Source Memory in CIRRUS X-store.

7.5.3. Program Control.

The System/360 operates under the control of a "program status" word, PSW, of 64 bits. These 64 bits include a sequence counter for instruction addressing, an instruction length code (indicating 2 to 6 bytes), various masks for interrupt and status checks, status bits etc. It will be convenient to separate functionally different parts of the program status word to save mapping operations in each routine flow execution.

7.5.4. Exceptions and Interrupts.

System/360 is provided with an exception register of 16 bits. Various bits are set upon invalid conditions arising within the micro-structure, e.g. attempts to execute operations not provided in the computer,memory addressing faults etc. This exception register is cleared prior to the commencement of the routine flow operation. At certain times during the routine flow and after the execution of the machine instruction, the exception register is interrogated for any indication of malfunction. Should a malfunction be indicated, the appropriate interrupt is generated.

Interrupts may also be generated by the channels, machine faults, external signals and privileged operations. The interrupt set is interrogated after any machine language instruction execution.

The detection of an interrupt causes an appropriate interrupt processing routine to be activated.

7.5.5. The Micro-Programmed Simulator.

A flow diagram of the basic operations performed within the System/360 routine flow is given in Fig.7.4. A micro-programmed simulator has been written for CIRRUS to simulate this routine flow operation. Each

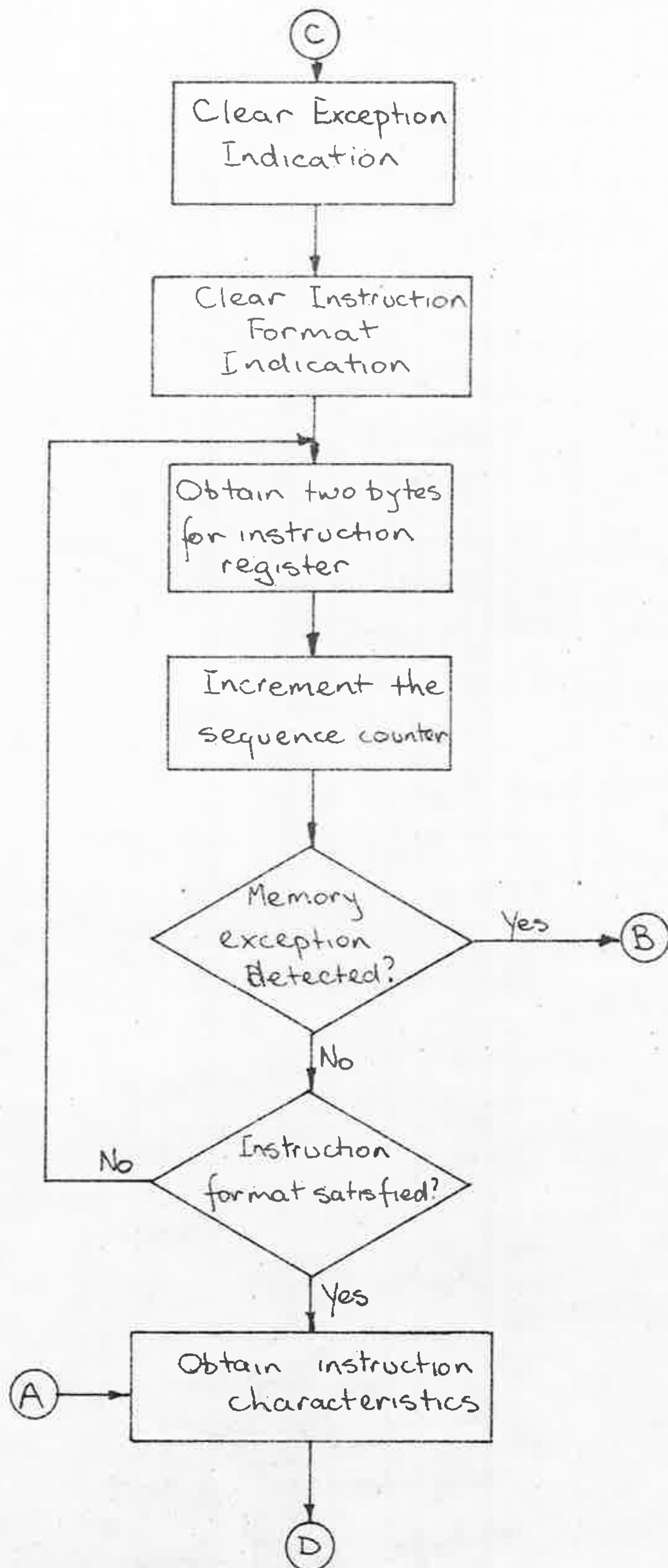


Fig. 7.4. The Source Routine Flow to be Simulated,

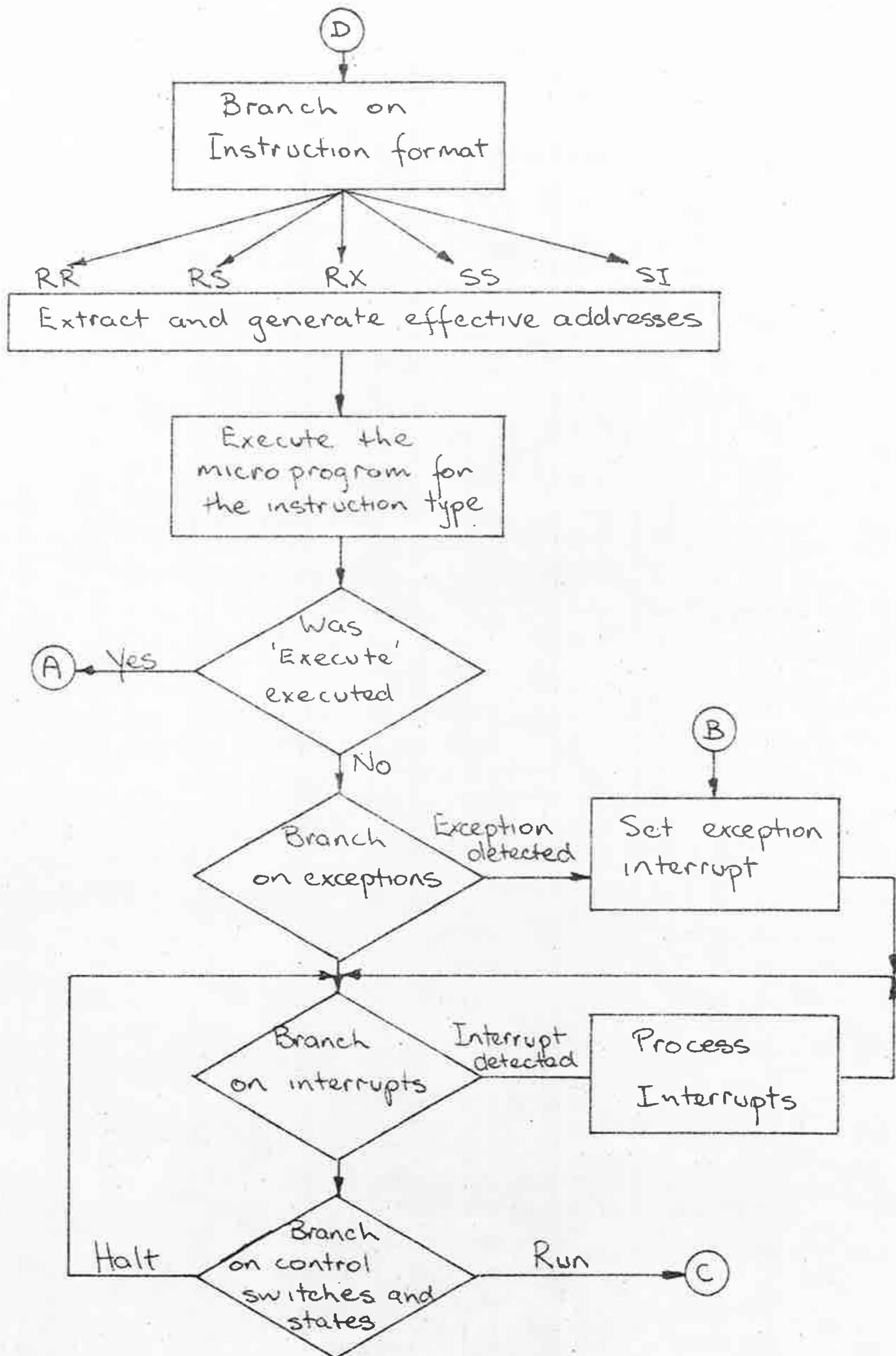


Fig. 7.4.(Contd) The Source Routine Flow to be Simulated.

logical entity which is necessary for the simulation resides in CIRRUS and is retrieved and operated upon as required. Many references within the System/360 set of micro-programs may be replaced or discarded since CIRRUS has alternate methods for dealing with them. For example, the System/360 queueing system for access to memory is logically a very complicated process involving priorities of channels over CPU, privileged areas of store which may only be accessed by the computer under a special mode, various address checks etc.

Doubtless in actual System/360 machines, many of these operations are automatically provided by hardware features. The simulator, on the other hand, operates in a different hardware environment and must set up essential logical variables to provide an interface between the simulated source memory operation and the actual host memory operation.

Certain checks and information generation procedures of the routine flow in the System/360 have been discarded. These include certain checks for included options (e.g. floating point, decimal arithmetic) on the assumption that these will all be provided in the simulator, and certain parts of the memory access phase connected with the priority system. On the other hand, an interpretive routine is necessary to transform the CIRRUS keyboard instructions to System/360 form, the simulated System/360 forms being treated as logical variables in CIRRUS, where they are available to the simulator.

Fig. 7.5. shows a flow diagram of the routine flow simulator, with the addition of certain interpretive routines necessary to adapt CIRRUS hardware to its System/360 logical counterpart. Appendix 8 gives a

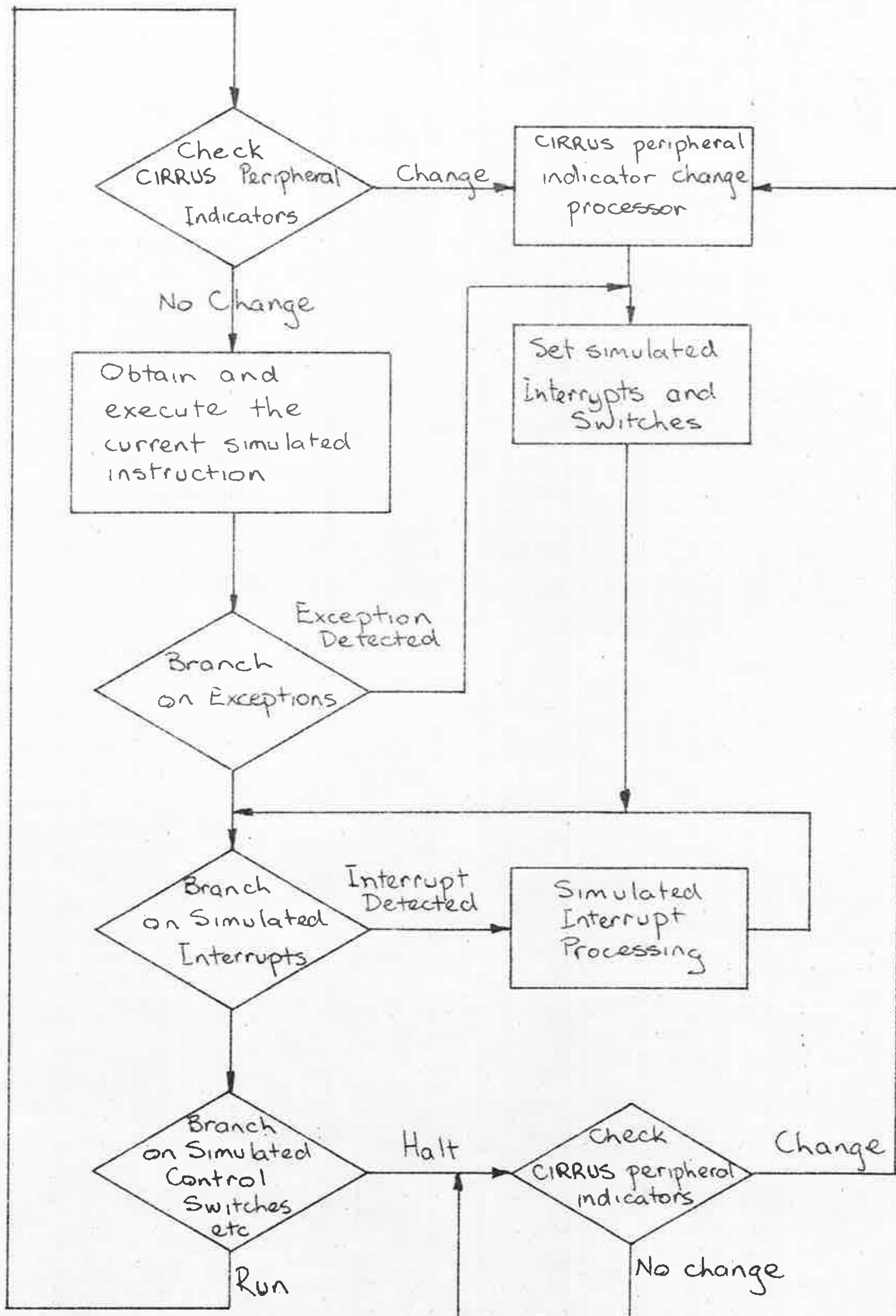


Fig. 7.5. Simulator Flow Diagram.

listing of this simulator, together with a storage map of the logical System/360 variables in CIRRUS.

Simulation of the System/360 routine flow occupies from 198 to 358 micro-seconds per cycle, depending upon the instruction format used. This compares with times of 27-39 micro-seconds for the equivalent operations in CIRRUS operating under the original host software system.

7.6. Limitations of Micro-Programmed Simulators.

The figures obtained for simulating the System/360 routine flow in CIRRUS are somewhat disappointing, but an examination of the micro-program reveals several points where heavy penalties are paid in execution time.

Firstly, a large amount of time is wasted in operating source variables which are held in local store in the simulator. The need to obtain or store approximately 20 logical variables in each routine flow, places a lower limit upon the execution time of the routine flow. In CIRRUS, this is particularly damaging since the local or Y-store operates on a 6 micro-second cycle, andwas originally designed to operate synchronously with the main or X-store. The original CIRRUS software is specifically designed with this point in mind and generally only 3 or 4 storage cycles are required in the normal routine flow.

To overcome this storage overhead, two "solutions" are available:-

(a) The local cycle time may be reduced.

(b) The number of registers within the host structure may be increased to provide static register storage space for most logical variables throughout the period of the routine flow.

Both these solutions would force extensive modifications to be made to the host computer, thus defeating one of the main objects of the compatibility exercise.

A second major factor in lengthening the simulated routine flow of the System/360 in CIRRUS, is the need to perform what would be fairly trivial hardware operations by a series of micro-instructions. If the host computer is not supplied with appropriate hardware, or if its hardware is of a different form, the logical variables in the source computer must be simulated in the host and interpreted for use by the host hardware. An example is the prefix attachment facility of the System/360 memory. If an address of less than 2^{12} is sent to memory, either a main or an alternate prefix is attached to it, depending upon the setting of a logical variable called the prefix trigger. In a hardware configuration, this function is readily mechanized by attaching suitable logical units to the address lines of the memory under the control of a prefix trigger flip flop. In the simulator however, since no prefix facility exists at the CIRRUS hardware level, such an operation must be simulated by 7 micro-instructions, including one store sycle to obtain a simulated prefix trigger and a mapping operation to incorporate the prefix. This simulation occupies 15 micro-seconds at each memory operation.

SECTION 8.

Conclusion.

8. Conclusion.

In this thesis, we have demonstrated the principle of the mapping unit, a device for implementing a set of variable interconnections between binary registers within the central processing units of conventional general purpose digital computers. The mapping unit, based upon the principles of fixed or read-only storage and associated logical functions, operates at speeds comparable to other functional units (adders etc.) likely to be found in such CPU's, and may be treated for design purposes as simply another functional unit to be incorporated in the hardware structure.

The control of such a mapping unit will be divided into two distinct sections, local and supervisory. The supervisory control, which forms part of the main control unit of the computer, is responsible for defining input information to the unit (selection of the appropriate mapping function, input data), activating the unit, monitoring the status of the output, and transmitting the output information to the appropriate destination. The local control, once activated by the supervisory control, is responsible for all internal timing operations, viz. address decoding, drive waveforms, output buffer initialization, etc.

The supervisory control may take the form of a micro-programmed control unit, in which case the mapping unit is easily integrated into the hardware and micro-instruction structures. An example of the inclusion of a mapping unit in such a structure is given, the computer involved being CIRRUS. The inclusion of a mapping unit in a structure controlled by conventional control sequence generators is

equally straight forward, although an extra amount of hardware will be required for the supervisory control.

The flexibility attained by a computing structure containing a mapping unit as an integral part has been demonstrated in a number of fields. The application of the mapping unit to resolving quantities represented in software data formats into forms suitable for hardware manipulation has been demonstrated by consideration of floating point operations, shifting operations, and quarter-length arithmetic operations in the CIRRUS structure augmented by a suitable mapping unit.

In discussing the provision of machine language compatibility between two dissimiliar computers, several techniques for executing source system programs within an alien computer were mentioned, viz. interpretive programming, machine language translation and micro-programmed simulators. The third method, that of micro-programmed simulation, offers advantages in speed and operating simplicity over the other two. Assuming that the host hardware contains some facility for extending the set of micro-programs controlling the execution of machine instructions, it becomes feasible to provide a new set of micro-programs to execute source system instructions in the host computer, using source system instruction and data formats.

The technique of micro-program simulation of alien machine functions has been examined. Several problems which are encountered were discussed in detail, and attempts were made to provide suitable solutions. These problems included definition of adequate functional simulation, instruction and data format transformation from source to host, the provision of large numbers of logical variables in

a simulator, and the provision of source system hardware functions by interpretive micro-programs in the host computer. Peripheral compatibility was considered briefly, principally on the point of peripheral operations and codings, rather than on physical characteristics.

Finally an example of machine language compatibility was offered, using the I.B.M. System/360 as the source computer and CIRRUS (augmented by a mapping unit), as the host hardware. Although the speed attained by the micro-programmed simulator is somewhat disappointing, the study serves to emphasize certain problems involved in providing compatibility features. Certain of these problems, typically logical variable storage, were accentuated by the lack of a high speed local store in CIRRUS, but others, e.g. simulation of source hardware operations, are fundamental to the nature of the problem.

APPENDICES.

Appendix 1.

A brief description of CIRRUS is given here because a large number of references to its internal structure are made in parts of this thesis. For more detailed information see [1],[2],[3],[4].

Structure of CIRRUS.

The CIRRUS hardware structure is shown in Fig. A1.1. It includes:-

(1) A set of general purpose registers M,R,N,Z, A and E. A and E are short registers, principally used for store addressing and floating point exponent buffers respectively. M,R,N,Z are nominally 18 bit registers, although overflow bits, shift bits and multiplier buffers are appended, increasing the length of R,N and Z to 19 bits.

(2) An 18 bit parallel add-logic network capable of performing

  (a) two's complements addition,
  (b) the bit-wise logical AND function,
  (c) the bit-wise logical inclusive-OR function,
  and (d) the bit-wise logical exclusive-OR function

on any two inputs presented to it. Associated with the add-logic network are two single bit buffers G_1 and G_0 used to buffer arithmetic carries arising in multi-precision arithmetic operations.

(3) A set of conditional transfer indicators J_{0-15} are included in the structure. Some of these form parts of the registers already

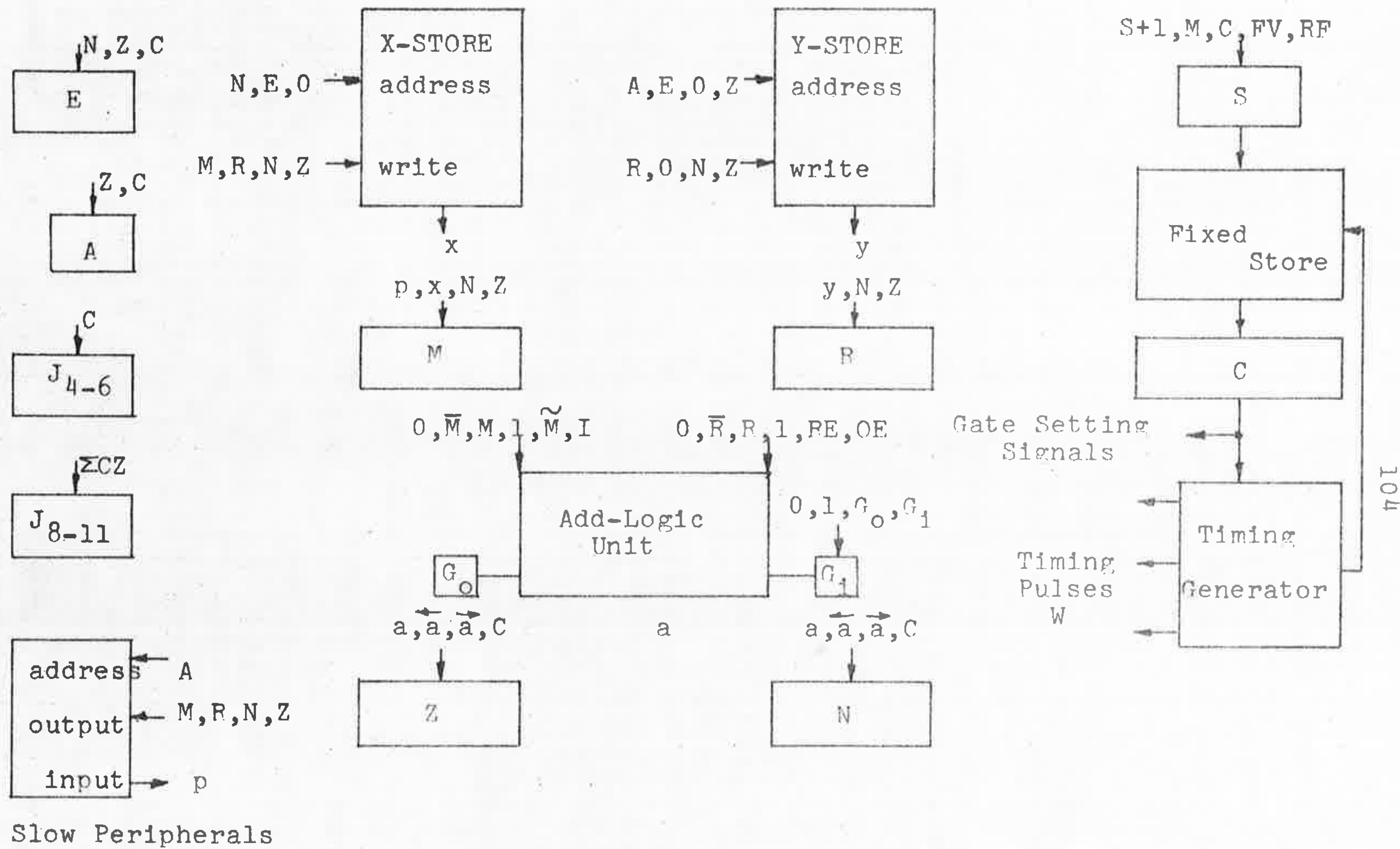


Figure A1.1. The Existing CIRRUS Hardware Structure.

described. Others, viz, J_{4-6} and J_{8-11} are provided in the form of flip-flops which may be conditionally set by the control unit. Subsequently these jump buffers may be used as indicators for conditional micro-code branch operatores (see below). A list of indicators and their equivalents are given in Table 4.1.

(4) Two random access core stores are provided. The main, or X-store, consists of 16,384 locations each of 18 bits. The register, or Y-store, consists of 1,024 locations of 19 bits. The stores read out information to the register pool and receive information for storage from the register pool. The stores derive their addressing information from the register pool, the information so derived being concatenated with a single bit from the control unit to yield on "upper" or "lower" address associated with each value of the selected address register.

(5) A set of on-line slow peripherals (single character devices) are associated with the X-store under A register addressing conditions. No reference is made to these units in the thesis and [4] should be consulted for a description of the multi-programming (input-output and computing overlap) system.

(6) The control unit consists of :-

- (a) a micro-sequence counter S of 12 bits,
- (b) a control fixed store of 4096x36 bit words,
- (c) a 36 bit control register C,

and (d) a timing generator.

J_0	= 1 always
J_1	= overflow bit of R
J_2	= bitwise OR of M_{3-12} (used in indexing)
J_3	= Sign of Z register
J_4	
J_5	= Type 7 JP Buffers
J_6	
J_7	= Sign bit of M
J_8	
J_9	Type 4 SJ Buffers
J_{10}	
J_{11}	
J_{12}	= Exponent Overflow
J_{13}	Spare
J_{14}	= Sign bit of E
J_{15}	= Bitwise OR of Z

Table A 1.1. Table of J equivalents.

The action of the control unit may be summarized as follows:-

(a) The fixed store is driven at the location specified by the contents of the S counter.

(b) The information at this address is read into the C register. This information is the current micro-instruction to be executed.

(c) The micro-instruction is executed. Various portions of the micro-instruction specify timing patterns and gate combinations. The gate combinations select the various inputs to be applied to all functional units within the computer, including the S counter. The possible setting sources for any unit are shown in the system diagram (Fig. A1.1). The appropriate codes and formats are given in Table A1.1.

(d) When the timing chain has terminated for the current micro-instruction, re-drive of the fixed store at the new S location is initiated, and the system re-cycles.

Some overlapping of micro-instructions is available, but this will be ignored in the interests of simplicity. No overlapping is used in the examples in this thesis.

CIRRUS Micro-Instructions.

There are eight different types of micro-instruction, and these are detailed in Table A1.2. These eight different types each have their own basic

Format	1–3	4–5	6	7–8	9–10	11	12–13	14	15	16	17	18–19	20–21	22–23	24–25	26	27	28–30	31–33	34–36
FM	0 0 0	Copy coding if overlapping else zeros						0	0	S+1 →S S12	Hold →A Z_y	E N Z - →E	M - N Z →M	R - N Z →R	O I G_b G_i →G_i	L	≢	O R M I M̃ I → m	O R R I RE OE → r	N Z / NS LS RS / HOLD N,Z 1 1 1
FA	0 0 1	"						"		"	"	"	"	"	"	"	"	"	"	"
AXY	0 1 0	N E - O →X_D	U L →X_H	M R N Z →x_0	A E O Z_B →Y_D	U L →Y_H	R O N Z → y	O' LAP	I O' LAPS Z	"	"	"	M x N Z →M	R y_i N Z →R	"	"	"	"	"	"
AY	0 1 1	0 0 0 0 0			"	"	"	"		"	"	"	M - N Z →M	"	"	"	"	"	"	"
AX	1 1 0	N E - O →X_D	U L →X_H	M R N Z →x_0	0 0 0 0 0			"		"	"	"	M x N Z →M	R - N Z →R	"	"	"	"	"	"

Table A1.2. CIRRUS Micro-Instruction Formats.

Table A1.2. (contd.) CIRRUS Micro-instruction Formats.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
SJ	1	0	0	0	0	0	0	0	Z_0	1	2	3	4	0	0	0	0	0	0	Z_{11}	12	13	14	15	16	17	18	19	20	21	22	23	24	0	$\Sigma C2 \rightarrow$ J8 J9 J10 J11	
SR	1	0	1	0	0	0	0	0	0 Z_0	N_1 1	2 2	3 3	4 4	0	0	0	0	0	0	N_{11} Z_{11}	12 12	13 13	14 14	15 15	16 16	17 17	18 18	19 19	20 20	21 21	22 22	23 23	24 24	N Z	1	1
				Copy coding if overlapping else zeros										0	0	C_6 C_1	$C \rightarrow A$ A	1	1	$C \rightarrow E$ E	0	0	0	0	0 E_1	0 2	0 3	A_2 4	3 5	4 6	5 7	6 8	7 9	A_p E_p	0	0
JP	1	1	1	Copy codes if overlapping else zero										S12 C $M \rightarrow S$ FV		Hold $J \rightarrow J_{4,5,6}$ C	J_4	5	6	S_1	2	3	4	5	6	7	8	9	10	11	12	S_p		Condition for branch $= J_i$		

N.B. The position of the mnemonic in the column indicates the coding for that function, e.g. for FA, N x_0 is coded as $C_7 = 1$, $C_8 = 0$.

timing sequence, and their own gate combination codes. A brief description of these micro-codes follows.

Type 1. C_{123}=001. Symbolic name FA.

This micro-instruction is principally used for high speed arithmetic manipulation of data within the register pool. No storage transfers are involved. The basic timing pattern is

(a) The upper register A,E,M and R are set from the appropriately indicated sources.

(b) The add-logic unit inputs, m and r, are developed and the output a, of the add-logic unit appears some time later (approximately 0.25 micro-seconds later in the worst case).

(c) The lower register selected to receive the output (either N or Z) is set, either by a, or by a right or left shifted one position. No multiposition shifts are available at the micro-code level.

(d) During the upper register setting period, the micro-sequence counter S may either be incremented or set to 1000_8 (the commencement of the "routine flow", or instruction extraction sequence in the normal machine language operating system. Type 1 micro-instructions are timed at 1.5 micro-seconds nominal.

Type 0. C_{123}=000. Symbolic name MP.

This instruction used only in multiplication operations, is identical in operation to Type 1 micro-instructions except that the injection of R, the multiplicand, into r is conditional on a multiplier bit, N19.

Type 6. C_{123}=110. Symbolic name AX.

This micro-instruction is similar to Type 1 micro-

instructions, except that tha main store is cycled during the execution of the micro-instructions. The operation is therefore somewhat slower. Information read from a location in memory may be set into the M register and after processing, information may be written back into the <u>same</u> location. Timing is thus keyed to a read-compute-write store cycle, and is nominally 6 micro-seconds.

<u>Type 3.</u> C_{123}=011. Symbolic name AY.

This micro-instruction is similar to Type 6 micro-instructions, except that the register,or Y-store, is the store cycled. Information from a location in store may enter the register pool via the R register,and after processing, information from the register pool is written into the same location. Type 3 micro-instructions are nominally timed at 6 micro-seconds.

<u>Type 2.</u> C_{123}=010. Symbolic name AXY.

This micro-instruction is a combination of Types 3 and 6 micro-instructions. Both stores are cycled in parallel using the read-compute-write cycle. Timing is 6 micro-seconds nominal.

<u>Type 4.</u> C_{123}=100. Symbolic name SJ.

This micro-instruction is used to set any of the jump buffers J_8,J_9,J_{10}, or J_{11}. The bit-wise OR of the intersection of the Z register and a portion of the C register is used to set the indicated J buffer. Nominal timing is 1.5 micro-seconds.

<u>Type 5.</u> C_{123}=101. Symbolic name SR.

This micro-instruction allows any of the N,Z,A or E registers to be set to a value specified within

the micro-instruction. Nominal timing is 1.5 micro-seconds.

Type 7. C_{123}=111. Symbolic name JP.

The micro-instruction performs a conditional micro-code transfer function. If the nominated J indicator or buffer is true (==1), then the S counter is set as indicated by the various C register bits. If the condition is not true (=0), the S counter is incremented by 1. In addition, the J buffers, J_4,J_5,J_6, may be set if desired. Nominal timing is 1.5 micro-seconds.

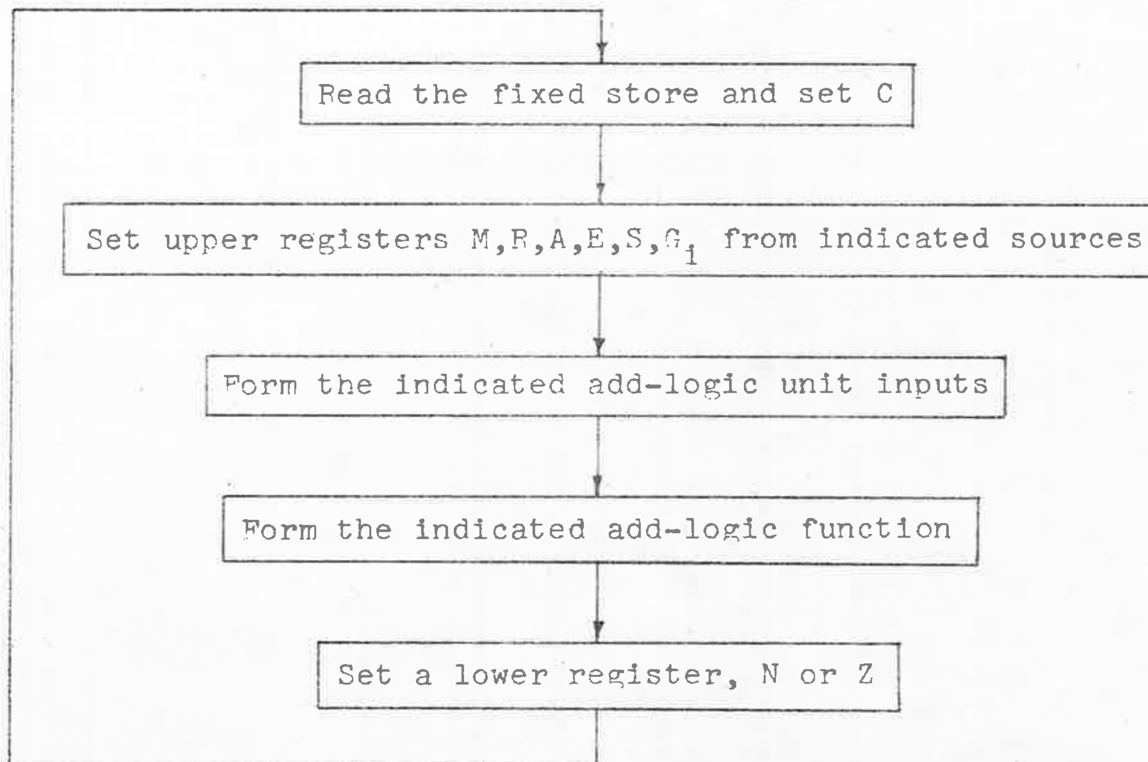
Transfer Timing.

A brief resumé of the transfers within each micro-code is given in Fig. A1.3. These transfers will only occur if specified by the micro-instruction. Generally,upper registers (M,R,A,E) are set first then information is generated via the add-logic unit and clocked to the lower registers. Stores therefore tend to feed information to the upper registers and receive information from the lower registers (unless re-generation of stored information is required).

The Symbolic Micro-Instruction Language.

In order to facilitate the investigation of various micro-programs, a symbolic micro-instruction language SYMITOR [13] and associated translators have been devised as part of the project. Input to the translator consists of micro-programs written in the symbolic language SYMITOR, and output from the program consists of assembled micro-instructions in octal form, suitable for imput to the simulators of Appendix 3. Wiring tables, used in the preparation of the

Types FA, FM.



Types SJ, SR, JP.

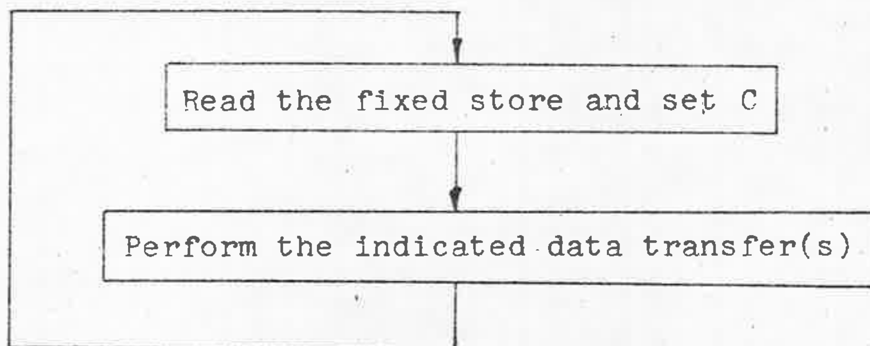


Figure A1.3. Transfer Timing in CIRRUS Micro-Instructions.

Types AXY, AX, AY.

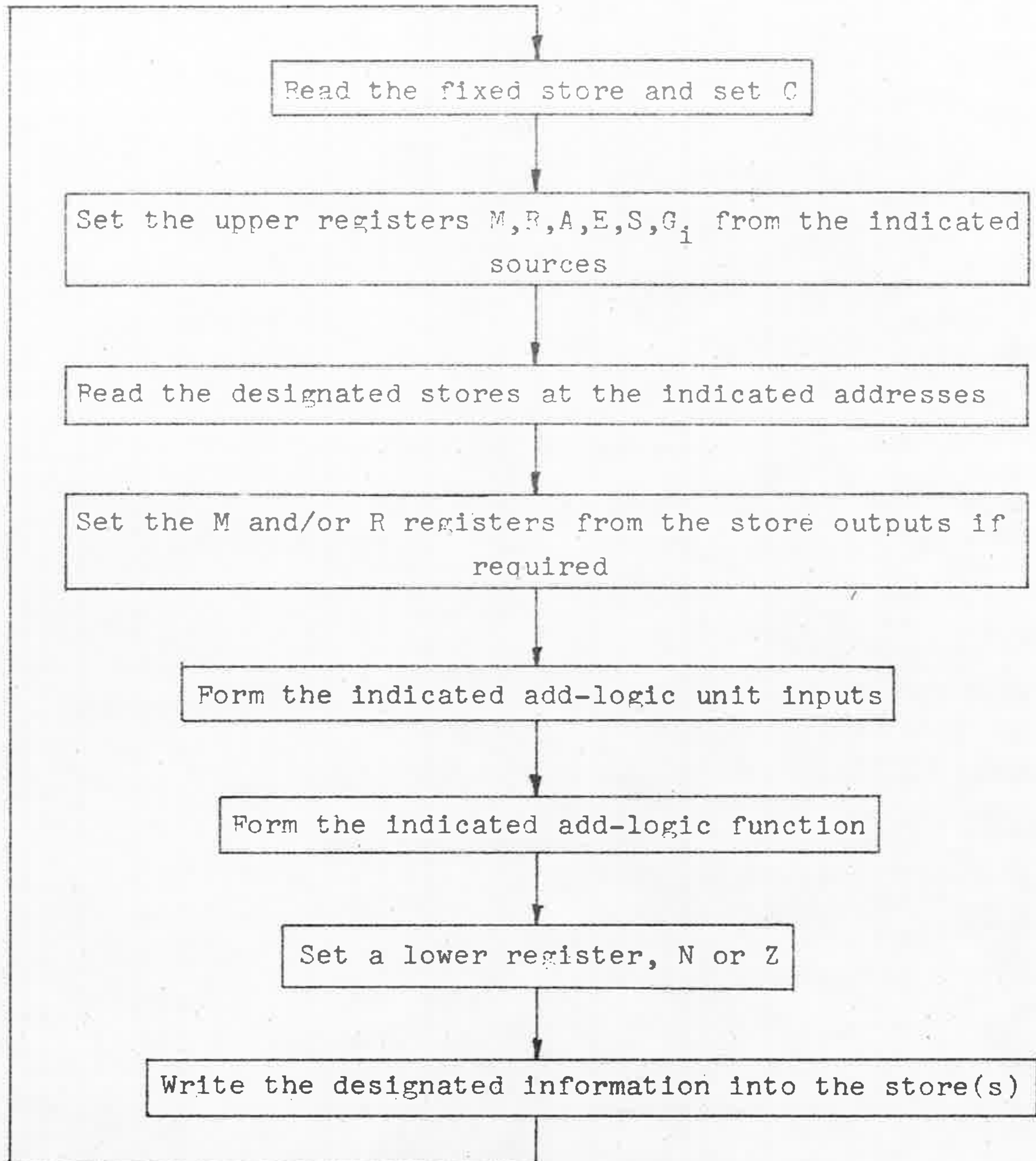


Figure A1.3. Transfer Timing in CIRRUS Micro-Instructions.

physical form of the control fixed store can also be obtained by a subsidiary routine.

SYMITOR resembles a simple assembly language. Provision is made for labelling individual micro-instructions, defining addresses in fixed store of micro-instructions, mnemonic forms of the various elements of the machine and symbolic representation of the various functions of the machine. The elements of the language are :-

(1) Control Words. There are three control words, BEGIN (adrs), WAIT, and END. The BEGIN word is used to define the address in fixed store within which the rest micro-instruction is to be placed. It therefore must appear at the commencement of a program, and may appear at points within the program. Within the program, consecutive micro-instructions are placed in sequential locations in fixed store unless a BEGIN statement re-sets the address counter. The WAIT control causes the translation program to pause whilst further tapes (CIRRUS is a paper tape machine) are loaded. END signifies the end of a micro-program.

(2) Labels. Micro-instructions may be referred to by the pre-fixing of the instruction with a label, consisting of not more than 6 alpha-numeric characters, the first of which is a letter, followed by a colon e.g., AB4: Ø123: CAPS:. The labels are treated as symbolic addresses and will be evaluated during translation. References to them within the micro-program will be replaced by their numerical values.

(3) Micro-code types. In the interests of accuracy, the micro-programmer is required to attach the micro-instruction type number to each micro-instruction.

(4) Data Transfers. The symbol '>' is used to imply data transferral. N>M specifies that the M register will be set to the contents of the N register at the appropriate time in the execution of the micro-instruction. The system diagrams (Fig. A1.1 and Table A1.1) show all possible transfers. When stores are involved, addressing information is contained within the symbolic name e.g., XEL>M implies that the X store is to be addressed by the E register, and the lower half of the location is to be transferred to the M register. Conditional transfers to the S register have the conditional indicator attached e.g. RF12>S (J14). Transfers in the add-logic unit must specify both inputs to the unit and the function the unit is to perform (addition (+), AND (*), OR (/), logical differ (=)).
Transfers to the lower registers must also indicate any shifting which is to occur.

(5) SYMITOR Formats. Some formats are shown in Figure A1.4.

The CIRRUS Machine Language System, A-Code.

The CIRRUS machine language [12] bears little resemblance to the micro-code structure and instruction system presented above. In fact, the CIRRUS structure as seen by a programmer consists, at the present time of a 36 bit parallel computer providing a wide range

Label	Type	Data Transfer Specifications
Q1:	1	(N>M)[LS]>N
	3	(YAL>R)=I>Z
AB1:	7	RF12>S(J14), 001>J46
	2	(YAL>R)+(XNU>M)>Z>YAL, RF>S
	4	777740*Z>J9
	5	123456>Z

Figure A1.4. Examples of the SYMITOR Format.

of numerical (fixed point and floating point) and non-numerical operations. At the present time, the number of machine language instructions and variants number well over 200. Each of these machine language instructions is implemented in the micro-structure as a series of micro-instructions.

Provision is available for the addition of new machine language operations to the repertoire of CIRRUS. Suitable micro-programs can be developed and placed in the fixed store, whereupon the new instructions are available for general use, and will ultimately be available through software systems (compilers etc.) should they prove effective.

The programmer sees the computer as consisting of a main store (8,192 words of 36 bits), 16 general purpose registers (of 36 bits) and a flexible arithmetic unit providing fixed point integral and fractional operations, floating point operations, logical operations, a flexible set of conditional branching operations, etc.

This implementation of a logical structure in an apparently unrelated hardware system is a good example of the differentiation between hardware and logical forms of a computer, as observed in Section 7.3.

The CIRRUS Procedure Oriented Language, C-Code.

The CIRRUS computer is provided with a procedure oriented language, similar in scope to common P.O.L.'s. Reference [11] gives a complete description of the language. Where used in this thesis, the examples are self explanatory.

Appendix 2.

Detailed Design for a Mapping Unit in CIRRUS.

The mapping unit to be included in CIRRUS will be a 36 bit input, 18 bit output unit operating in parallel with the add-logic unit. Fig. A2.1 is a detailed system schematic diagram of those parts of CIRRUS changed by the inclusion of the mapping unit. Figure A2.1 should be read in conjunction with the remainder of the system diagrams, Figs. A1.1, A1.2.

Inputs to the unit are derived from the left and right inputs of the add-logic unit, m and r. The output from the unit, which is buffered, feeds the lower registers via one of the inputs to the lower register selector, $\bar{a}$. This lower register selector input was previously occupied with a shifted form of the result from the add-logic unit which can be discarded in the presence of the mapping unit. A selection register L is provided, which can either be set from one of the lower registers Z or N, or can be incremented by 1 under the control of the micro-instruction. The micro-instruction bits used for the control of the add-logic unit may also be used for the control of the mapping unit since outputs from the mapping unit and the add-logic unit are not used simultaneously.

Fig. A2.2 presents a schematic diagram of the proposed selection system using a source-sink technique for the generation of the necessary drive pulses. Fig. A2.3 presents an internal timing diagram of the mapping unit. Timing signals which are to be generated by the main control unit in order that the mapping unit operation may be synchronized with the micro-instruction execution are :-

(1) A clearing signal for the mapping unit output buffer must be generated at the commencement of each micro-instruction execution phase

$$T_{q\ clear} = W_{da} \qquad [A2.1]$$

(2) A signal to commence the mapping operation must occur immediately following the setting of the upper registers M and R.

$$T_{map} = W_{1(RP)} \cdot \overline{C_1} \cdot \overline{C_2} \cdot C_3 + W_3 \cdot C_{21} + W_R \cdot \overline{C_1} \cdot C_2 \cdot C_3 . \qquad [A2.2]$$

(3) A signal to transfer information into the selection register L will be generated at the same time as the upper registers are set.

$$T_L = W_{1(RP)} \cdot C_{24} \qquad [A2.3]$$

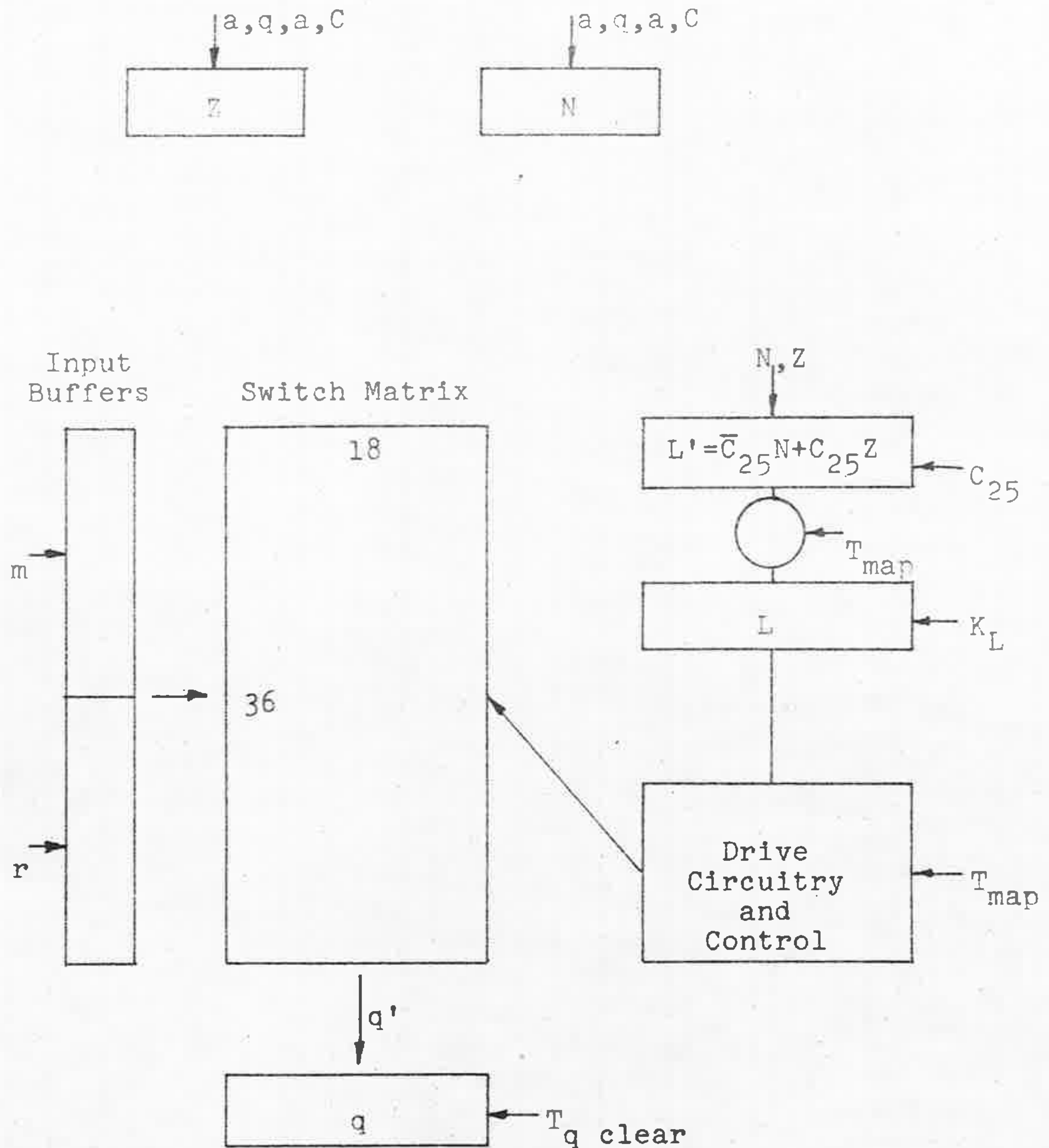
The information to be transferred into the selection register L is formed by

$$L' = \overline{C}_{25} \cdot N + C_{25} \cdot Z \qquad [A2.4]$$

(4) The selection register L must be incremented (if required) after the lower registers are set.

$$K_L = W_{2(R)} \cdot \overline{C}_1 \cdot \overline{C}_2 \cdot C_{27} + W_4 \cdot \overline{RT} \cdot C_{27} . \qquad [A2.5]$$

In all these operations, the W variables refer to various machine timing pulses derived from the timing chain within the control unit of CIRRUS.



This figure should be read in conjunction with figures 4.5 and A1.1.

Figure A2.1. Proposal for a Mapping Unit in CIRRUS.

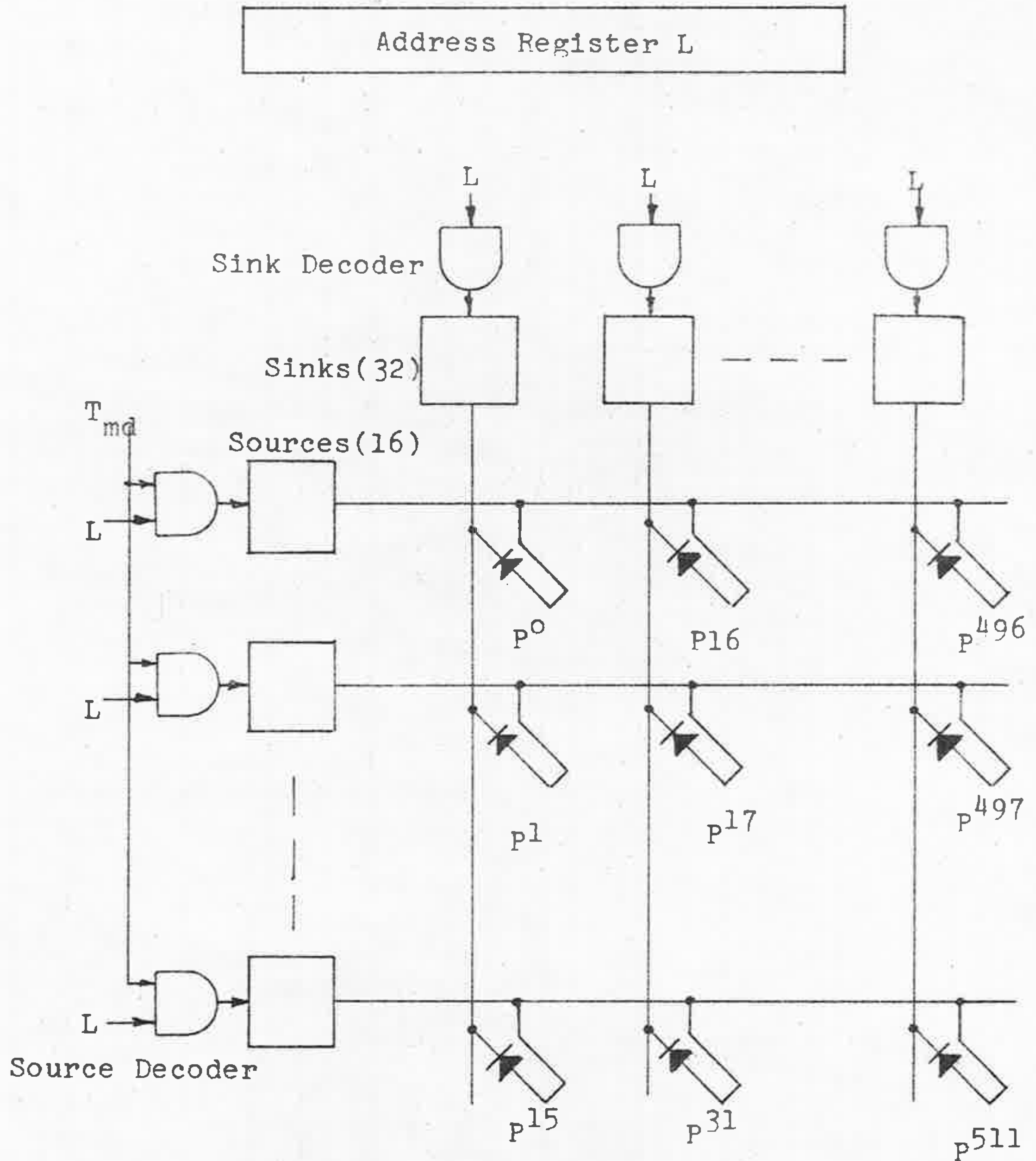


Figure A2.2. Drive System for the Mapping Unit in CIRRUS.

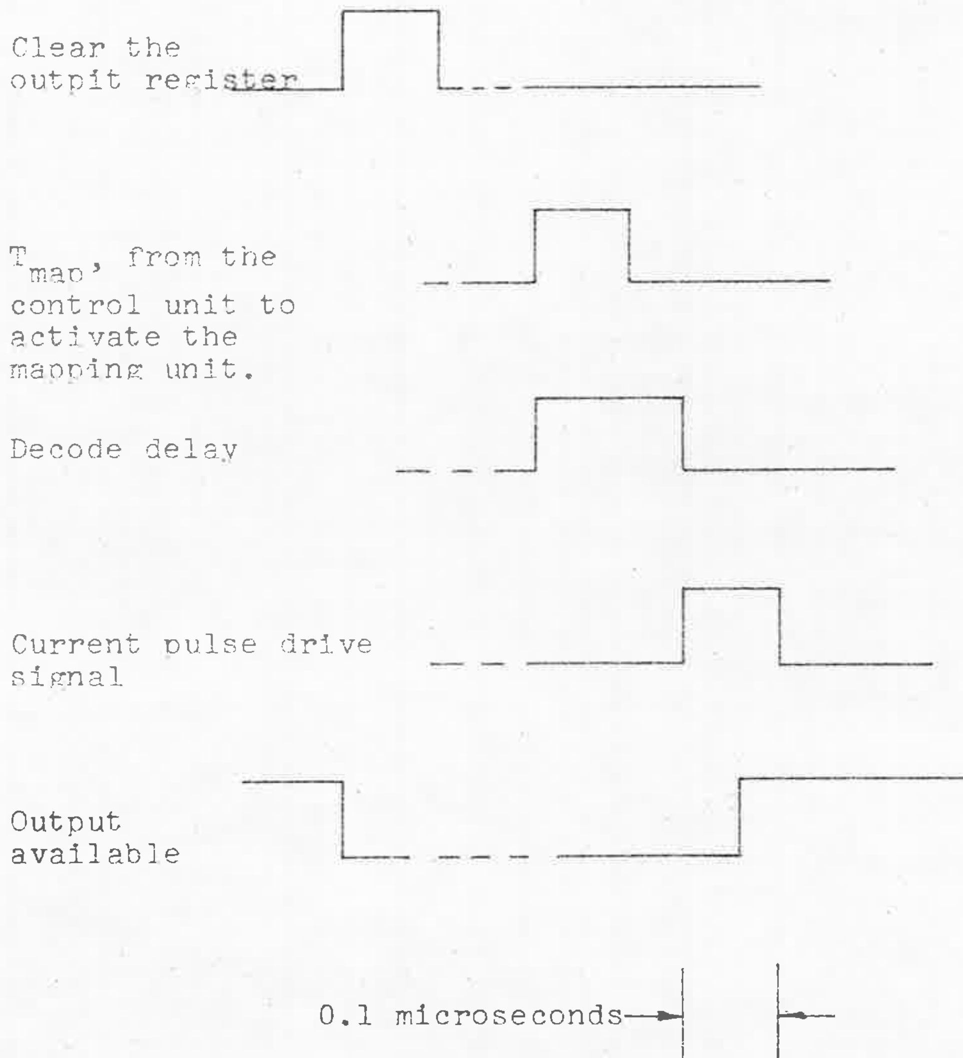


Figure A2.3. Internal Timing for the Mapping Unit In CIPRUS.

1	2	3	4–23	24–25	26	27	28–30	31–33	34	35–36
0	0	1	As for normal instructions	L L N → L Z	1	Hdd →L L+1	OMM M I → m MI	OR R I → r RE OE	N Z	0 1
0	1	0								
0	1	1								
1	1	0								

This format is to be read in conjunction with Table A1.2.

Fig. A2.4. Micro-instruction Format for Mapping Operations in CIRRUS.

Appendix 3.

Simulators for the Existing and Augmented CIRRUS Structures.

The listing of a simulator for the augmented CIRRUS structure is given overleaf.

The following service sub-routines have been omitted:-

CHOOSE, SSW1, SSW2, SSW3, SSW4, NEXT, RSHIFT, LSHIFT

The listing for the simulator of the existing structure has been omitted. It is essentially similar in form to the given augmented simulator.

```
100 2000
99 4     ..CIRRUS SIMULATOR (AUGMENTED STRUCTURE)
0-END
```

```
99
.ALLOT(.L JIND(0/15),NUM,CREG,FST(0/1025),MREG,RREG,
        NREG,ZREG,AREG,EREG,
        N25,YD,YH,YST(0/127),YO,XD,XH,XO,XST(0/1023),
        GI,GO,A,LEFT,
        RIGHT,ARING,
        .I I,J,K,SREG,SBUF,LIM,
TABLE(0/199),TYPE)
.ALLOT(.L MAPIN,MAPOUT,MAP(0/99,0/17),LREG)
        .FUNCTION(SSW1, SSW2, .L LSHIFT(.L, .I),
        RSHIFT(.L, .I),RXST,WXST,RYST,WYST,
        OPER,OPER1,CHOOSE(.I,.L,.L,.L,.L,.L,.L))
        .ALLOT(.I XADRS,YADRS,NUSE,.L NBUF)
.ALLOT(.I YADRS1, .L XI)
.ALLOT(TIME,TYPTIM(0/7)=1.5 1.5 6.0 6.0 1.5 1.5 6.0 1.5)
START:  JIND(0)=1
        TIME=0
        .FOR I=4(1)11,JIND(I)=0
        YD=YH=XD=XH=MREG=RREG=NREG=ZREG=AREG=EREG=0
        YADRS1=YADRS=XADRS=0
.FUNCTION(.L NEXT(.L))
        .FOR I=0(1)1023, FST(I)=0
        .FOR I=0(1)127,YST(I)=0
        .FOR I=0(1)1023, XST(I)=0
        .FOR I=0(1)127, .FOR J=0(1)8, MAP(I,J)=0
        .WRITE .UNIT 2,F2('LOAD FIXED STORAGE')
.FUNCTION(SSW3)
        .IF SSW3 .POS, .PAUSE
S2:     .IF NEXT(37) .NEG, .GO .TO S1
        .READ .UNIT 3(NUM)
        I=NUM
        .READ .UNIT 3(FST(I))
        .GO .TO S2
S1:     .WRITE .UNIT 2,F2('LOAD REGISTER STORE')
        .IF SSW3 .POS, .PAUSE
S4:     .IF NEXT(37) .NEG, .GO .TO S3
        .READ .UNIT 3(NUM)
        I=NUM
        I=2*I
        .READ .UNIT 3(YST(I),YST(I+1))
        .GO .TO S4
```

```
S3:     .WRITE .UNIT 2,F2('LOAD MAIN STORE')
        .IF SSW3 .POS, .PAUSE
S5:     .IF NEXT(37) .NEG, .GO .TO S6
        .READ .UNIT 3(NUM)
        I=NUM
        .READ .UNIT 3(XST(I))
        .GO .TO S5
S6:     .WRITE .UNIT 2,F2('SET INITIAL S COUNT')
        .IF SSW3 .POS, .PAUSE
        .READ .UNIT 3(NUM)
        SREG=NUM
        .WRITE .UNIT 2,F2('LOAD TRACE POINTS')
        .IF SSW3 .POS, .PAUSE
        K=0
        .ALLOT(.L NUM1,NUM2,.I INUM1,INUM2)
S8:     .IF NEXT(37) .NEG, .GO .TO S7
        .READ .UNIT 3(NUM1)
.A
        L PY 2 11,T
          PC 3 -13,2
        L PN 3 S9
.C
        .READ .UNIT 3(NUM2)
        INUM1=NUM1
        INUM2=NUM2
        .FOR I=INUM1(1)INUM2,1(
        TABLE(K)=I
        K=K+1)1
        .GO .TO S8
S9:     TABLE(K)=NUM1
        K=K+1
        .GO .TO S8
S7:     .WRITE .UNIT 2,F2('LOAD MAP PATTERNS')
        .IF SSW3 .POS, .PAUSE
S7B:    .IF .L NEXT(37) .NEG, .GO .TO S7A
        .READ .UNIT 3(NUM)
        I=NUM
        .FOR J=0(1)17, 1(
        .READ .UNIT 3(NUM)
        MAP(I,J)=LSHIFT(NUM,18))1
```

```
         .FOR J=0(1)17, 1(
         .READ .UNIT 3(NUM)
         MAP(I,J)=MAP(I,J)+NUM)1
         .GO .TO S7B
S7A:     .WRITE .UNIT 2,F2('SET SWITCHES AND CARRIAGE')
         LIM=K-1
         .IF SSW3 .POS, .PAUSE
         .WRITE .UNIT 2,FF1('S','NEXT','C','M','R','Z',
         'N25','N','E','A','GIGO','L','J4-11',
         'YD','H','(YU)','(YL)','XD','H','(X)','TIME')
FF1:     .FORMAT(////,3X,S,2X,S,6X,S,10X,S,6X,S,7X,
         S,3X,S,3X,S,4X,S,3X,S,1X,S,1X,S,4X,S,3X,S,
         1X,S,3X,S,
         4X,S,4X,S,2X,S,5X,S,7X,S)
         .GO .TO P3
LOOP:    CREG=FST(SREG)
         .IF .L CREG .ZERO,1(
         .WRITE .UNIT 2,FC('NULL MICRO-ORDER    ',SREG)
FC:      .FORMAT(///,S,L4)
         .PAUSE
         .GO .TO START)1
         SBUF=SREG
         TYPE=.L RSHIFT(CREG,33)*7
         TIME=TIME+TYPTIM(TYPE)
         .BRANCH .ON TYPE+1,(TYPE0,TYPE1,TYPE2,TYPE3, .CT
         TYPE4,TYPE5,TYPE6,TYPE7)
TYPE5:   .IF .L CREG*1 .NONZ, .GO .TO T5NZ
         .IF .L CREG*200000 .NONZ, .GO .TO T5A
         EREG=RSHIFT(CREG*7770,3)
         .GO .TO T5EX
T5A:     .IF .L CREG*4000000 .NONZ, .GO .TO T5A1
         AREG=RSHIFT(CREG*770,3)
         .GO .TO T5EX
T5A1:    AREG=AREG*76+((AREG*1)-1)
         .GO .TO T5EX
T5NZ:    .IF .L CREG*4 .NONZ, .GO .TO T5Z
         NREG=RSHIFT(CREG*377770,3)+RSHIFT(CREG* .CT
         740000000,9)
         .GO .TO T5EX
```

```
T5Z:      ZREG=RSHIFT(CREG*377770,3)+RSHIFT(CREG* .CT
          1740000000,9)
T5EX:     SREG=SREG+1
T5EX1:    JIND(12)=RSHIFT((MREG*400)-(RREG*400) .CT
          -(ZREG*400)-LSHIFT(MREG*200, )    .CT
          -LSHIFT(RREG*200,1),8)
          JIND(3)=RSHIFT(ZREG*400000,17)
          JIND(14)=RSHIFT(EREG*400,8)-1
          JIND(15)=0
          .IF .L ZREG*777777 .NONZ, JIND(15)=1
          .GO .TO PRINT
TYPE7:    I=(.L CREG*17)
          K=RSHIFT(CREG*30000000,21)
          .IF .L JIND(I) .NONZ, 1(
          .BRANCH .ON K+1,(T7RF,T7C,T7M,T7FV)
T7RF:     SREG=512
          .GO .TO T7
T7C:      SREG=RSHIFT(CREG*377740,5)
          .GO .TO T7
T7M:      SREG=.L MREG*7777
          .GO .TO T7
T7FV:     SREG=.L RSHIFT(NREG*700000,15) .CT
          +LSHIFT(ZREG*77,3)
          .GO .TO T7)1
          SREG=SREG+1
T7:       .IF .L CREG*4000000 .NONZ,1(
          JIND(4)=RSHIFT(CREG*2000000,19)
          JIND(5)=RSHIFT(CREG*1000000,18)
          JIND(6)=RSHIFT(CREG*400000,17))1
          .GO .TO PRINT
TYPE4:    I=(.L CREG*3)+8
          .IF .L ZREG*(RSHIFT(CREG*377770,3)+ .CT
          RSHIFT(CREG*1740000000,9)) .NONZ, 1(
          JIND(I)=1
          .GO .TO T4)1
          JIND(I)=0
T4:       SREG=SREG+1
          .GO .TO PRINT
TYPE0:    .IF .L N25 .ZERO,CREG=CREG*777777777757
```

```
TYPE1:   OPER
TAEX:    JIND(1)=RSHIFT(RREG*1000000,18)
         JIND(2)=0
         .IF .L MREG*170000 .NONZ, JIND(2)=1
         JIND(7)=RSHIFT(MREG*400000,17)
         .GO .TO T5EX1
TYPE3:   RYST
         OPER
         WYST
         .GO .TO TAEX
TYPE6:   RXST
         .IF .I I-2 .ZERO, .GO .TO IO
         OPER
T6A:     WXST
         .GO .TO TAEX
TYPE2:   RXST
         RYST
         OPER
         WXST
         WYST
         .GO .TO TAEX
PRINT:   .FOR I=0(1)LIM, .IF .I TABLE(I)-SBUF .CT
          .ZERO, .GO .TO P1
         .IF SSW1 .POS, .GO .TO P4
         .IF SSW2 .POS, .GO .TO P5
         .GO .TO LOOP
P1:      .WRITE .UNIT 2,F1(SBUF,SREG,CREG,MREG,RREG, .CT
         ZREG,N25,NREG,EREG,AREG,GI,GO,LREG)
F1:      .FORMAT(2(1X,L4),1X,L12,1X,L6,2(1X,L7),1X, .CT
         L1,1X,L6,1X,L3,1X,L2,2(1X,L1),1X,L3,1X,8L1,1X,L3
         ,
         1X,L1,2(1X,L7),1X,L5,1X,L1,1X,L12,F7.1)
         .FOR I=4(1)11, .WRITE(JIND(I))
         .WRITE(YD,YH,YST(YADRS1),YST(YADRS1+1), .CT
         XADRS,XH,XST(XADRS),TIME)
P3:      .IF SSW1 .POS, .GO .TO P4
         .IF SSW2 .POS, .GO .TO P5
         .GO .TO LOOP
P4:      .READ .UNIT 1(NUM)
```

```
         .FOR I=0(1)LIM, .IF .I TABLE(I)-NUM .CT
          .ZERO, .GO .TO P2
         TABLE(LIM)=NUM
         .IF .I(LIM=LIM+1)-200 .POS, .WRITE .UNIT .CT
          2,F2('LIMIT EXCEEDED')
         .GO .TO P3
P2:      .FOR K=I(1)LIM,TABLE(K)=TABLE(K+1)
         LIM=LIM-1
         .GO .TO P3
.ALLOT(.I FADRS,NUMDUM,STOR)
P5:      .READ .UNIT 3(STOR)
         .READ .UNIT 3(NUM)
         FADRS=NUM
         .READ .UNIT 3(NUM)
         NUMDUM= .I NUM+FADRS-1
         .BRANCH .ON STOR,(DF,DX,DY)
DF:      .WRITE .UNIT 2,FA('FIXED STORE DUMP')
         .FOR STOR=FADRS(1)NUMDUM, .WRITE(STOR,FST(STOR))
DF1:     .PAUSE
         .GO .TO P3
FA:      .FORMAT(S,/*L5,2X,L12)
DX:      .WRITE .UNIT 2,FA('MAIN STORE DUMP')
         .FOR STOR=FADRS(1)NUMDUM, .WRITE(STOR,XST(STOR))
         .GO .TO DF1
DY:      .WRITE .UNIT 2,FB('REGISTER STORE DUMP')
         .FOR STOR=FADRS(1)NUMDUM,1(
         K=2*STOR
         .WRITE(STOR,YST(K),YST(K+1)))1
FB:      .FORMAT(S,/*L3,2X,L7,2X,L7)
         .GO .TO DF1
F2:      .FORMAT(S)
IO:      .IF .L CREG*100000 .NONZ, .GO .TO OUT
         .WRITE .UNIT 2,F3('INPUT REQUEST',AREG)
F3:      .FORMAT(S,2X,L2,2X,L2)
         .READ .UNIT 1(NUM)
         .WRITE(NUM)
         MREG=NUM
         SREG=SREG+1
         .IF .L CREG*4000000 .NONZ, SREG=512
```

```
         OPER1
         .GO .TO TAEX
OUT:     .WRITE .UNIT 2,F3('OUTPUT RESULT', AREG)
         I=RSHIFT(CREG*600000000,28)
         XO=CHOOSE(I,MREG,RREG,NREG,ZREG,0,0)*77
         .WRITE(XO)
         SREG=SREG+1
         .IF .L CREG*4000000 .NONZ, SREG=512
         .GO .TO PRINT
.PROCEDURE RYST,
         I=RSHIFT(CREG*1400000000,26)
         YD=CHOOSE(I,AREG,EREG,0, .CT
         RSHIFT(ZREG*770000,12),0,0)
         YH=RSHIFT(CREG*200000000,25)
         YADRS1=2*YD
         YADRS=YADRS1+YH
         YO=YST(YADRS)
         .RETURN
         .END
.PROCEDURE WYST,
         I=RSHIFT(CREG*140000000,23)
         YST(YADRS)=CHOOSE(I,RREG,0, .CT
         NREG+LSHIFT(NREG*400000,1),ZREG,0,0)
         .RETURN
.END
.PROCEDURE RXST,
         NUSE=0
         NBUF=NREG
         I=RSHIFT(CREG*6000000000,31)
         .IF .I I .ZERO, .IF .L CREG*4 .ZERO, 1(
         NBUF=NREG
         NUSE=1)1
         XD=CHOOSE(I,NREG*77777,EREG,0,0,0,0)
         XH=RSHIFT(CREG*1000000000,30)
         XADRS=XD
         XO=XST(XADRS)*777777
         .IF .L XH .ZERO,XO=RSHIFT(XST(XADRS),18)*777777
         .RETURN
.END
.PROCEDURE WXST,
```

```
        .IF NUSE .ZERO, NBUF=NREG
        I=RSHIFT(CREG*6000000000,28)
        XI=CHOOSE(I,MREG,RREG*777777,NBUF,ZREG*777777,0,0)
        .IF .L XH .NONZ,1(
        XST(XADRS)=(XST(XADRS)*777777000000)+XI
        .RETURN)1
        XST(XADRS)=LSHIFT(XI,18)+(XST(XADRS)*777777)
        .RETURN
.PROCEDURE CHOOSE,
.A
        L PY 3   0,1
        L PY 0   0,3
        L PY 4 1,1,T
          PY 2   0,4
        S TC 3   7,1
.C
.END
.PROCEDURE OPER,
        .IF .L CREG*10000 .POS, 1(
        I=RSHIFT(CREG*4000,11)
        LREG=CHOOSE(I,NREG*777,ZREG*777,0,0,0,0))1
        .IF .L CREG*2000000 .NONZ,AREG= .CT
        RSHIFT(ZREG*7700,6)
        .IF .L CREG*4000000 .NONZ, SREG=511
OPO:    SREG=SREG+1
        I=RSHIFT(CREG*1400000,17)
        EREG=CHOOSE(I,EREG,NREG*777,ZREG*777,0,0,0)
        I=RSHIFT(CREG*300000,15)
OP1:    MREG=CHOOSE(I,MREG,XO,NREG*777777, .CT
        ZREG*777777,0,0)
        I=RSHIFT(CREG*60000,13)
        RREG=CHOOSE(I,RREG,YO, .CT
        NREG+LSHIFT(NREG*400000,1),ZREG,0,0)
.ENTRY OPER1,
        I=RSHIFT(CREG*14000,11)
        GI=CHOOSE(I,0,1,GO,GI,0,0)
OP2:    I=RSHIFT(CREG*700,6)
        .IF .L(CREG*7)-7 .ZERO, .RETURN
        LEFT=CHOOSE(I,0,MREG-777777,MREG, .CT
```

```
        777777,MREG*171777,0)
        LEFT=LEFT+LSHIFT(LEFT*400000,1)
        I=RSHIFT(CREG*70,3)
        RIGHT=CHOOSE(I,0,RREG-777777,RREG,777777, .CT
        (377*EREG)+(RREG*777400),EREG*377)*777777
OP3:    RIGHT=RIGHT+LSHIFT(RIGHT*400000,1)
        MAPIN=RIGHT*777777+LSHIFT(LEFT*777777,18)
        I=(.L LREG*777)
        MAPOUT=0
        .FOR J=0(1)17, 1(
        .IF .L RSHIFT(MAPIN,17-J)*1 .POS, MAPOUT= .CT
        MAPOUT+MAP(I,J)
        .IF .L RSHIFT(MAPIN,35-J)*1 .POS, MAPOUT=MAPOUT .CT
        +RSHIFT(MAP(I,J),18))1
        MAPOUT=MAPOUT*777777
        .IF .L CREG*1000 .POS, LREG=(.I LREG+1)
        .IF .L CREG*2000 .NONZ, .GO .TO LOG
        A=.X RIGHT+LEFT+GI
        GO=RSHIFT((A-RIGHT-LEFT)*1000000,18)
        .GO .TO SETLOW
LOG:    .IF .L CREG*1000 .ZERO, .GO .TO ANDOR
        A=RIGHT-LEFT
        GO=0
        .GO .TO SETLOW
ANDOR:  .IF .L CREG*4000 .NONZ, .GO .TO AND
        A=RIGHT+LEFT
        GO=0
        .GO .TO SETLOW
AND:    A=RIGHT*LEFT
        GO=0
SETLOW:            ARING=RSHIFT(A*1000000,18)
        .IF .I TYPE .ZERO,ARING=GO
        I= .L CREG*7
        .BRANCH .ON I+1,(NNS,NLS,NRS,OP2,ZNS,ZLS,ZRS,OP2)
NNS:    NREG=A*777777
NNS1:   N25=A*1
        .RETURN
ZNS:    ZREG=A*1777777
        .RETURN
NLS:    NREG=MAPOUT
        .GO .TO NNS1
NRS:    NREG=RSHIFT(A*777777,1)
        .GO .TO NNS1
ZLS:    ZREG=MAPOUT
```

```
        .RETURN
ZRS:    ZREG=RSHIFT(A*777777,1)+LSHIFT(ARING,18) .CT
        +LSHIFT)ARING,17)
        NREG=(NREG*377777)+LSHIFT(A*1,17)
        .RETURN
        .END
.END
```

Appendix 4.

Normalized Binary Fractions.

Consider the representation of a signed binary fraction f, by an ordered set of binary coefficients b_i, such that

$$f = \sum_i b_i . 2^{-i}, \quad 1 \leq i \leq n \qquad [A4.1]$$

Let negative numbers be represented by the diminished radix complement of the magnitude of the number,

$$f = 2 - 2^{-n} - |f| \qquad [A4.2]$$

and let b_0 be the sign bit attached to the representation. Then the computer word appears as a string of binary digits,

$$b_0 b_1 b_2 b_3 \, \ldots\ldots\ldots \, b_n$$

in a computer whose word length is n+1 bits. We are interested in the range of numbers represented when $b_0 \neq b_1$.

Consider two cases :

(a) $b_0 = 0$, $b_1 = 1$

$$f = 1/2 + \sum_{i=2}^{n} b_i . 2^{-i} \qquad [A4.3]$$

The evaluation of the summation term provides upper and lower limits of its value of

$$0 \leq \sum_{i=2}^{n} b_i . 2^{-i} \leq 1/2 - 2^{-n} \qquad [A4.4]$$

$$\therefore \quad 1/2 \leq f \leq 1 - 2^{-n} \qquad [A4.5]$$

(b) $b_0 = 1,\ b_1 = 0$

$$f = -(1/2) - \sum_{i=2}^{n} (1 - b_i).2^{-i} \qquad [A4.6]$$

Substituting

$$c_i = 1 - b_i \qquad [A4.7]$$

and noting that if b_i is a binary coefficient, so is c_i, and the limits of the summation term are

$$0 \leq \sum_{i=2}^{n} c_i.2^{-i} \leq 1/2 - 2^{-n} \qquad [A4.8]$$

$$\therefore \quad -1 - 2^{-n} \leq f \leq -(1/2) \qquad [A4.9]$$

Combining cases (a) and (b), equations [A4.5] and [A4.9] yield

$$\text{if } b_0 \neq b_1, \quad 1/2 \leq |f| \leq 1 - 2^{-n} \qquad [A4.10]$$

Since the granularity of the representation, i.e. the smallest recognizeable difference between two numbers is 2^{-n}, equation [A4.10] may be written

$$\text{if } b_0 \neq b_1, \quad 1/2 \leq |f| < 1 \qquad [A4.11]$$

Appendix 5.

Overleaf are presented listings of the left shift micro-programs for the existing and augmented structures. The maps used in the augmented structure micro-program for a shift of 13 places are also presented.

The operations contain provision for the detection and notification of overflow.

```
.BEGIN
LSN0:       5    77777>Z
            1    (Z>M)*(N>R)>N  ;FORM SHIFT LENGTH
            5    44>Z
            1    (N>M)-(Z>R)[1>GI]>Z ;TEST IF LENGTH>36
            7    LSN1>S(J3)
            3    (YAU>R)>Z,    0>YAU   ;INJECT ZEROS
            4    777777*Z J8
            3    (YAL>R)>Z,    0>YAL
            4    777777*Z>J9
            7    LSN2>S(J8)
            7    LSN2>S(J9)    ;TEST FOR OVERFLOW
            1    RF>S
LSN2:       5    1000000>Z     ;SET OVERFLOW INDICATOR
            3    Z YAU,   RF S
LSN1:       3    (YAU>R)>Z
            3    (YAL>R)>N    ;OBTAIN DATA
            4    400000*Z>J8
            4    400000*Z>J9
            7    LSN3>S(J8)      ;TEST FOR NEGATIVE DATA
LSN6:       1    Z>R, M+1>Z
            7    LSN4>S(J3)
            1    Z>M,   R[LS]>Z
            1    (N>R)[LS]>N    ;SHIFT LOOP
            7    LSN5>S(J3)
            7    LSN6>S(J0)
LSN5:       4    400000*Z J9      ;TEST FOR OVERFLOW
            7    LSN6>S(J0)
LSN3:       1    Z>R,   M+1>Z
            7    LSN4>S(J3)
            1    Z>M,   R[LS]>Z
            1    (N>R)[LS]>N      ;SHIFT LOOP
            7    LSN3>S(J3)
            4    0*Z>J9
            7    LSN3>S(J0)
LSN4:       3    N>YAL,   R>Z     ;LOOP COMPLETION
            7    LSN7>S(J8)       ;TEST FOR OVERFLOW
```

Continued overleaf

Table A5.1. Existing Left Shift Microprogram.

```
        7   LSN8>S(J9)
LSN9:   3   Z>YAU,   RF>S   ;RETURN RESULT
LSN7:   7   LSN9>S(J9)
LSN8:   5   400000>N
        1   N R,    (Z>M)[RS]>Z
        1   (Z>M)+R[LS]>Z          ;INJECT OVERFLOW BIT
        3   (N R)[LS]>N,   Z>YAU,   RF>S
.END
```

Table A5.1. cont. Existing Left Shifn Microprogram.

```
.BEGIN
LSNO:      5    77777>Z
           1    (Z>M)*(N>R)>Z    ;PICK UP SHIFT LENGTH
           7    LSN5>S(J15)      ;TEST FOR ZERO SHIFT
           1    RF>S
LSN5:      5    44>N
           1    (Z>M)-(N>R)[1>GI]>Z
           7    LSN1>S(J3)       ;TEST FOR LENGTH > 36
           3    (YAL R)>Z,   O>YAL
           7    LSN6>S(J15)
           3    (YAU>R)>Z,   O>YAU     ;INJECT ZEROS
           7    LSN6>S(J15)      ;TEST FOR OVERFLOW
           1    RF>S
LSN6:      5    1000000>Z        ;SET OVERFLOW INDICATOR
           3    Z>YAU,   RF S
LSN1:      5    2>Z
           1    Z>L,M[MAP]>N
           5    14>Z
           1    (Z M)+(N R)>N    ;GENERATE SHIFT MAP ADDRESS
           3    (YAU R)>Z
           3    N>L, (YAL>R)*(Z>M)[MAP]>Z>YAL, L+1>L
           3    M+R[MAP]>Z>YAU, L+1>L  ;OBTAIN DATA
           4    400000*Z J8            ;AND SHIFT
           7    LSN10>S(J7)
LSN11:     1    M+R[MAP]>Z, L+1>L
           1    M+R[MAP] N       ;MAP OVERFLOW PORTION
           1    (Z>M)/(N>R)>Z    ;OF DATA
           7    LSN12>S(J15)
           7    LSN15>S(J8)      ;TEST FOR OVERFLOW
LSN14:     1    RF>S
LSN10:     1    -M>Z
           1    Z>M,   -R>N      ;INVERT DATA IF NEGATIVE
           1    N>R
           7    LSN11>S(J0)
LSN12:     7    LSN14>S(J8)
           5    400000>N
           3    (YAU>R)+(N>M) Z  ;SET OVERFLOW INDICATOR
           3    (Z>R)+M Z>YAU,   RF>S
LSN15:     3    (YAU R)>Z>YAU,   RF>S  ;RETURN RESULT
.END
```

Table A5.2. Augmented Left Shift Microprogram.

ADDRESS = 100_8	ADDRESS = 101_{8_0}
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
100000000000000000	000000000000000000
010000000000000000	000000000000000000
001000000000000000	000000000000000000
000100000000000000	000000000000000000
000010000000000000	000000000000000000
000001000000000000	000000000000000000
000000100000000000	000000000000000000
000000010000000000	000000000000000000
000000001000000000	000000000000000000
000000000100000000	000000000000000000
000000000010000000	000000000000000000
000000000001000000	000000000000000000
000000000000100000	000000000000000000
000000000000010000	000000000000000000
000000000000001000	000000000000000000
000000000000000100	000000000000000000
000000000000000010	000000000000000000
000000000000000001	000000000000000000
000000000000000000	100000000000000000
000000000000000000	010000000000000000
000000000000000000	001000000000000000
000000000000000000	000100000000000000
000000000000000000	000010000000000000

Table A5.3. Shifting Maps for the Augmented Left Shift Operation.

ADDRESS = 102_8	ADDRESS = 103_8
000000000000000000	000000000000000000
000001000000000000	000000000000000000
000000100000000000	000000000000000000
000000010000000000	000000000000000000
000000001000000000	000000000000000000
000000000100000000	000000000000000000
000000000010000000	000000000000000000
000000000001000000	000000000000000000
000000000000100000	000000000000000000
000000000000010000	000000000000000000
000000000000001000	000000000000000000
000000000000000100	000000000000000000
000000000000000010	000000000000000000
000000000000000001	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000
000000000000000000	000000000000000000

Table A5.4. Overflow Detection Maps for the Augmented Left Shift Operation.

Appendix 6.

Provision of a General Mapping Instruction for CIRRUS.

This instruction, provided in the CIRRUS A-Code [12] repertoire (see Appendix 1), makes the facilities of the mapping unit within the augmented structure available to the machine language programmer.

Name	MA
Variants	N,U,L
Action	v MA y x
	(1) v = U,L. The Y-register designated by y and v is transformed according to the map whose address is x.
	(2) v = N. The Y-register designated by y is transformed according to the maps whose addresses are x, x+1.
Micro-Code	See Table A6.1 overleaf
Execution Time	
	U MA, L MA 9 micro-seconds
	N MA 21 micro-seconds
	These execution times do not include instruction extraction, modification or interpretation times.

```
UMA:    3   N>L, (YAU>R)[MAP]>Z>YAU, RF>S

LMA:    3   N>L, (YAL>R)[MAP]>Z>YAL, RF>S

NMA:    3   (YAU>R) Z
        3   N>L, (YAL>R)[MAP]>Z>YAL, L+1>L
        3   M+R[MAP]>Z>YAU, RF>S
```

Table A6.1. Microcodes for a General Mapping Operation in CIRRUS.

Appendix 7.

On the following pages we present the micro-program and the interpretive program for performing the quarter length period addition, X SYQ. (see Fig. A7.1 for a schematic diagram of the operation). The interpretive version, written in CIRRUS C-Code [11] may be considerably economized in A-Code [12], the C-Code version is presented for clarity. The timings of Table 6.2. are for the economized versions.

X SYQ

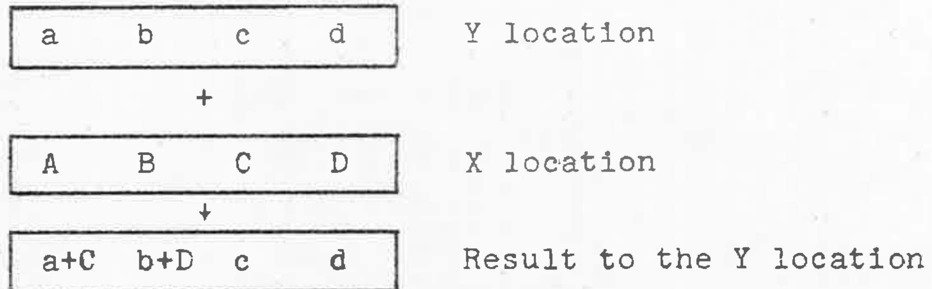


Fig. A7.1.

```
SYQX:   2   (XNL>M)+(YAU>R)>Z>XOU
        5   (MAP1)>Z
        1   Z>L, M+R[MAP]>Z, L+1>L
        1   M+R[MAP]>N
        1   (Z>M)+(N>R)>Z
        5   (MAP2)>N
        6   N>L,   (XOU>M)[MAP]>N
        3   (Z>M)/(N>R)>Z>YAU, RF>S
```

Table A7.1. Quarter Length Addition - Micro-Instruction Version.

```
.PROCEDURE SYQ(A,B,C),
..A=VARIANT, B=X LOCATION, C=Y REGISTER
.ALLOT(.I A,M,N, .L B,C,T1,T2,T3,T4)
..ESTABLISH SHIFTS REQUIRED TO ISOLATE COMPONENTS
..FOR A GIVEN VARIANT FROM THE QUARTER LENGTH FORMAT
        N=INTDIV(3-A,2)*18
        M=REM(A+1,2)*18
..ISOLATE THE INDIVIDUAL COMPONENTS, REDUCE MODULO 511
        T1=RSHIFT(B,M+9)*777
        T2=RSHIFT(B,M)*777
        T3=RSHIFT(C,N+9)*777
        T4=RSHIFT(C,N)*777
..PERFORM ADDITION, REDUCE RESULTS MODULO 511
        T1=(.I T1+T3)*777
        T2=(.I T2+T4)*777
..REFORM QUARTER LENGTH FORMAT FOR APPROPRIATE VARIANT
        C=C*LSHIFT(777777,18-N)+LSHIFT(T1,N+9)   .CT
          +LSHIFT(T2,N)
        .RETURN
        .END
```

Table A7.2. Interpretive Subroutine for Quarter Length Addition.

Appendix 8.

A Micro-Programmed Simulator.

This appendix contains a listing of the CIRRUS micro-programmed simulator of the intruction extraction, modification and interpretation of the I.B.M. System/ 360.

Interrupt and exception detectors are provided, but processing of such "abnormal" conditions do not form part of the normal "routine flow".

A storage map of the logical variables simulated within CIRRUS is also given.

ADDRESS	UPPER	LOWER	
0	Exceptions	Indicators	
1	PSW(49-64)	I^0	Instruction Registers
2	IRA	I^1	
3	PSW(32-33)	I^2	
4	Interrupts	a_1	Address Registers
5	PSW(1 -16)	a_2	
6	PSW(17-32)	a_3	
7	PSW(33-48)	l_1	Field Lengths
10		l_2	
11		l_3	
12	address compare switch	address switch	
13	operating state	rate switch	
—	//////////	//////////	
20	R^0	R^0	General Purpose Registers
21	R^1	R^1	
. . . .			
37	R^{15}	R^{15}	
40	F^0	F^0	Floating Point Registers
41	F^0	F^0	
42	F^1	F^1	
. . . .			
47	F^3	F^3	

Figure A8.1. Storage Map of CIRRUS Y-Store for The Micro-Programmed Simulator.

```
;SYSTEM/360 MICROPROGRAMMED SIMULATOR
RF:     3   (YOL>R)=1>Z
;CHECK CIRRUS INDICATORS
        7   RF1>S(J15), 000>J46
;CLEAR THE EXCEPTION REGISTER
        3   0>YOU
        5   2>A
;CLEAR THE INSTRUCTION REGISTER ADDRESS COUNTER
        3   0>YAU
RF9:    5   1>E
;OBTAIN AND INCREMENT THE SEQUENCE COUNTER, PSW(49-64)
        3   (YEL>R)+0E[1>GI]>Z>YEL
;DECOMPOSE ADDRESS
        5   (MAP1)>N
        1   N>L, R[MAP]>Z, L+1>L
        1   R[MAP]>N
;CHECK MEMORY ADDRESS FOR LEGAL FORM
        7   RF2>S(J15)
        1   (N>M) Z
        4   770000*Z>J8
;CHECK IF ADDRESS < 2**12
        7   RF3>S(J8)
        5   2>E
        5   (MAP3)>Z
        3   (YEL>R)+(Z M)>Z
;ADD THE APPROPRIATE PREFIX
        1   Z>L, (N>R)[MAP]>N
RF3:    5   12>E
        3   (YEU>R)>Z
;CHECK IF ADDRESS COMPARISON REQUIRED
        7   RF4>S(J15)
;COMPARE ADDRESS AND ADDRESS SWITCH
        3   (YEL>R)-(N>M)[1>GI]>Z
        7   RF4>S(J15)
;SET OPERATING STATE = STOP
        5   20>E
        3   0>YEU
;OBTAIN TWO BYTES FROM MAIN STORE AND PLACE
;IN APPROPRIATE INSTRUCTION REGISTER
RF4:    5   (MAP3)>Z
```

```
         1    Z>L, (N>R)[MAP]>N, L+1>L
         1    R[MAP]>Z
         7    RF5>S(J15)
         3    (YAU>R)[1>GI]>Z>YAU
         2    Z>E, (XNU>M) Z>YEL
         7    RF6>S(JO)
RF5:     3    (YAU>R)[1>GI]>Z>YAU
         2    Z>E, (XNL>M) Z>YEL
;BRANCH IF THE BYTES OBTAINED ARE PLACED IN
;INSTRUCTION REGISTER O
RF6:     7    RF7>S(J4)
         5    (MAP5)>Z
         5    1>A
;SET INSTRUCTION LENGTH CODE
         3    Z>L, (YAL>R)[MAP]>Z, L+1>L
         1    R[MAP]>N
         5    3>A
         3    (N>M)+(Z>R)>Z>YAU
;COMPARE LENGTH CODE AND NUMBER OF BYTES OBTAINED
RF7:     5    3>A
         3    (YAL>R)>N
         1    -(Z>M)+OE>Z

;BRANCH IF MORE BYTES REQUIRED
         7    RF9>S(J3), 100>J46
;OBTAIN OPERATION CODE AND BRANCH TO FIRST DIRECTORY
         5    1>A
         5    (MAP7)>Z
         3    Z>L, (YAL>R)[MAP]>Z
         1    R[MAP]>N
         1    Z>M
         7    M>S(JO), 111>J46
;ISOLATE EFFECTIVE ADDRESSES BY FORMATS
;REGISTER - REGISTER OPERATIONS
RR:      1    N>M
RR3:     5    1>A
         5    (MAP9)>Z
         3    Z>L, (YAL>R)[MAP]>Z, L+1>L
         3    Z>YEU, R[MAP]>N, L+1>L
```

```
         1    OE[1>GI]>Z
         3    Z>E, N>YEU
;BRANCH TO SECOND DIRECTORY FOR INSTRUCTION EXECUTION
         7    M>S(JO)
;STORAGE - IMMEDIATE FORMAT
SI:      1    N>M
SI3:     5    2>A
         5    (MAP11)>Z
         3    Z>L, (YAL>R)[MAP]>Z, L+1>L
         3    R[MAP]>N
;DETECT IF INDEX ADDRESS = 0
         4    17*Z>J8
         7    SI1>S(J8)
SI2:     5    4>E
         3    N>YEL
         7    M>S(JO)
;ADD CONTENTS OF THE INDEXING REGISTER
SI1:     3    Z>E, M>Z, YEL>R
         1    R+(N M)>N
         1    Z>M
         7    SI2>S(JO)
;STORAGE -STORAGE OPERATIONS
SS:      1    N>M
         5    1>A
         5    7>E
;ISOLATE AND STORE FIELD LENGTHS
         5    (MAP13)>Z
         3    Z>L, (YAL>R)[MAP]>Z, L+1>L
         3    Z>YEL, R[MAP]>N, L+1>L
         1    R[MAP]>Z
         1    Z>R, OE[1>GI]>Z
         3    Z>E, N>YEL
         1    OE[1>GI]>N
         3    N>E, R>YEL
;FORM STORE ADDRESSES, INDEXING IF NECESSARY
         5    3>A
         5    (MAP15)>Z
         3    Z>L, (YAL>R)[MAP]>Z, L+1>L
         1    R[MAP]>N
         4    17*Z>J8
```

```
         7   SS1>S(J8)
SS2:     5   5>E
         3   N>YEL
         7   SI3>S(J0)
SS1:     3   Z>E, M>Z, YEL>R
         1   R+(N>M)>N
         1   Z>M
         7   SI2>S(J0)
;REGISTER -STORAGE OPERATIONS
RS:      1   N>M
RS3:     5   1>A
;ISOLATE AND STORE EFFECTIVE ADDRESSES
         5   (MAP17)>Z
         3   Z>L, (YAL>R)[MAP]>Z
         5   6>E
         3   Z>YEL
         5   2>A
         5   (MAP18)>Z
         3   Z>L, (YAL>R)[MAP]>N, L+1>L
         1   R[MAP]>Z
;CHECK FOR INDEXING REQUIREMENT
         4   17*Z>J8
         7   RS1>S(J8)
RS2:     5   5>E
;STORE EFFECTIVE ADDRESS
         3   N>YEL
         7   RR3>S(J0)
RS1:     3   Z>E, YEL>R, M>Z
;PERFORM INDEXING
         1   (N>M)+R>N
         1   Z>M
         7   RS2>S(J0)
;REGISTER - STORAGE INDEXED OPERATIONS
RX:      1   N>M
         5   2>A
         5   (MAP18)>Z
         3   Z>L, (YAL>R)[MAP]>N, L+1>L
         1   R[MAP]>Z
         4   17*Z>J8
;TEST FOR FIRST INDEXING REQUIREMENT
```

```
        7   RX1>S(J8)
RX4:    5   1>A
        5   (MAP19)>Z
;FORM SECOND INDEX REGISTER ADDRESS
        3   Z>L, (YAL>R)[MAP]>Z
        4   17*Z>J8
;TEST FOR SECOND INDEXING REQUIREMENT
        7   RX3>S(J8)
RX6:    5   5>E
;STORE EFFECTIVE ADDRESS
        3   N>YEL
        7   RR3>S(J0)
RX3:    3   Z>E, YEL>R, M>Z
;PERFORM SECOND INDEXING
        1   R+(N>M)>N
        1   Z>M
        7   RX6>S(J0)
RX1:    3   Z>E, YEL>R, M>Z
;PERFORM FIRST INDEXING
        1   R+(N>M)>N
        1   Z>M
        7   RX4>S(J0)
;REENTRY POINT FOR ALL EXECUTION MICROPROGRAMS
;(EXCEPT 'EXECUTE')
RF20:   3   (YOU>R)>Z
;BRANCH IF EXCEPTIONS HAVE OCCURRED
        7   RF21>S(J15)
;CHECK FOR INTERRUPTS
RF26:   5   4>E
        3   (YEU>R)>Z
        7   RF22>S(J15)
;CHECK OPERATION STATE
        5   13>E
        3   (YEL>R)>Z
        7   RF23>S(J15)
;SET OPERATING STATE = STOP
        3   0>YEU
;SIMULATOR IS STOPPED, CIRRUS INDICATORS
;ARE MONITORED FOR KEYBOARD CHANGE
RF24:   3   (YOL>R)=1>Z
```

```
        7   RF1>S(J15)
        7   RF24>S(J0)
;CHECK FOR MANUAL CONDITION
RF23:   3   (YEU>R) Z
        7   RF25>S(J15)
        7   RF24>S(J0)
;CHECK FOR WAIT CONDITION
RF25:   5   5>E
        5   (MAP20)>Z
        3   Z L, (YEL>R)[MAP]>Z
        7   RF26>S(J15)
;PROCEED WITH NEXT INSTRUCTION
        7   RF>S
.END
```

Table A8.1. Listing of the CIRRUS Micro-Programmed Simulator for the System/360.

REFERENCES.

References.

(1) M.W. Allen, T.Pearcey, J.P. Penny, G.A. Rose, J.G. Sanderson : CIRRUS, An Economical Multi-program Computer with Micro-program Control, IEEE Trans. on Electronic Computers, Vol. EC-12, No. 5, Dec. 1963, pp 663-671.

(2) I.R. Butcher : A Prewired Storage Unit, IEEE Trans. on Electronic Computers, Vol. EC-11, No. 2, April 1964, pp 106-111.

(3) J. Penny and T. Pearcey : The Use of Multi-programming in the Design of a Low Cost Digital Computer, Comm A.C.M., Vol 5. No. 9, Sept. 1962.

(4) Various Internal Reports on the CIRRUS Computer, University of Adelaide.

(5) M.J. Flynn : Very High Speed Computing Systems, Proc. IEEE, Vol. 54, No. 12, Dec. 1966 pp 1901-1909.

(6) D. Jacobsohn : A suggestion for a Fast Multiplier, IEEE Trans. of Electronic Computers, Vol. EC-13, Dec.1964, p 354.

C.S. Wallace : A Suggestion for a Fast Multiplier, IEEE Trans. on Electronic Computers Vol. EC-13, Feb. 1964, pp 14-17.

(7) O.L. MacSorley : High Speed Arithmetic in Binary Computers, Proc. IRE, Vol. 48, No. 1, Jan. 1961.

(8) G. Estrene, B. Gilchrist, J.H. Pomerene : A Note on High Speed Multiplication, IRE Trans. on Electronic Computers, Vol. 5, No. 3, Sept. 1956, p 140.

(9) M. Lehman : Short Cut Multiplication and Division in Automatic Binary Digital Computers, Proc. IEE, Vol. 105B, Sept. 1958, pp 496-504.

(10) M. Lehman : High Speed Multiplication, IRE Trans. on Electronic Computers, Vol. 6 No. 3, 1957 pp 204-205.

(11) J.G. Sanderson : The CIRRUS A-Code Manual, University of Adelaide, Internal Report 1963.

(12) J.G. Sanderson : The CIRRUS C-Code Manual, University of Adelaide, Internal Report 1965.

(13) R.J. Potter : The SYMITOR Manual, University of Adelaide, Internal Report 1965.

(14) G.M. Amdahl, G.A. Blauuw, F.P. Brooks Jr. : Architecture of the IBM System/360, IBM J. of Research and Development, April 1964, pp 87-101.

(15) T.C. Bartee, I.L. Lebow, I.S. Reed : Theory and Design of Digital Machines, McGraw, Hill 1962.

(16) R.J. Mercer : Micro-programming, J. A.C.M. Vol. 4, April, 1957, pp 157-171.

(17) M.H. Lewin : A Survey of Read-Only Fixed Storage

Proc. FJCC, 1965 pp 775-788.

(18) K.E. Iverson : A Programming Language , Wiley, 1962.

(19) A.D. Falkoff, K.E. Iverson, E.H. Sussenguth : A Formal Description of System/360, IBM Systems J., Vol. 3, 1964, pp 198-261.

(20) M.V. Wilkes and J.B. Stringer : Micro-programming and the Design of the Control Circuits in an Electronic Digital Computer, Proc. Cambridge Phil. Soc. April,1953.

(21) Proceedings of the A.C.M. Re-programming Conference, Comm. A.C.M., Vol. 8, No. 12, Dec, 1965, pp 731-782.