AN ABSTRACT OF THE THESIS OF

KEI-LEUNG ALBERT LUN for the MASTER OF SCIENCE
(Name) (Degree)

in Electrical and Electronics Engineering presented on Dec. 14, 1972
(Major) (Date)

Title: A SYSTEM DESIGN OF A FORTRAN HARDWARE COMPILER

Abstract approved: Redacted for privacy
Professor Louis N. Stone

This paper describes the system design of a Fortran hardware compiler which converts the original code of a source program into an intermediate code. This code contains features that allow easy machine code generation. An evaluation is first made on the marketability of such a system and then a brief discussion on the features of the intermediate code generated by the hardware compiler.

The system is divided into a number of functional blocks. Each block consists of a control unit and a set of hardware logic components. The control unit is realized by Programmable Logic Arrays and all hardware components are state-of-the-art products. Estimates on the typical operating speeds of the functional blocks are made. Flow charts and state diagrams are used to describe the logic flow of the functional blocks.

System Design of a Hardware Fortran Compiler

by

Kei-Leung Albert Lun

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

June 1973

APPROVED:

Redacted for privacy

Professor of Electrical and Electronics Engineering
in charge of major

Redacted for privacy

Head of Department of Electrical and Electronics Engineering

Redacted for privacy

Dean of Graduate School

Date thesis is presented Dec. 14, 1972

Typed by Susie Kozlik for Kei-Leung Albert Lun

ACKNOWLEDGEMENTS

The author would like to thank Professor L. N. Stone for his guidance during the preparation of this thesis. He would also like to thank Dr. Harry Goheen whose advice has been most helpful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SYSTEM DESIGN OF A HARDWARE FORTRAN COMPILER

I. INTRODUCTION

Qualitative Evaluation

The throughput of a computing system depends greatly on how efficient its processing time is being used and how much of it is actually used on data processing. The fact is that a great deal of the processing time is spent not on problem solving, but on the various time-consuming, but essential house-keeping chores. These chores are usually carried out by an elaborate system program with the support of many other, equally complex subprograms. Occasionally, the systems program may call upon the services of one of its compilers to compile a certain source program. A compiler, basically, is an information handling process, consequently a large part of its complexity arises from symbol searching, statement analyzing, and code conversion. Its implementation by software is ill suited to a conventional computer, the inception of which is to deal with processing data. As a result, compilation is a slow process.

Many ideas have been suggested to improve the efficiency of existing software to speed up compilation. However, it is clear that no matter how elegant software is, its speed is inherently limited by the speed of the hardware which it uses. A new direction should be taken.

When one looks more closely at the somewhat mysterious compiling process, behind the facade of terminologies and definitions, one finds a well defined logical process that offers many interesting possibilities for hardware implementation. This paper is an exploration of these possibilities.

Suggestions to bypass software compilation are not new. Baskkow and associates (1967) have proposed a reasonably detailed logic design of a small scale computer which modifies only slightly a Fortran source program by hardware and executes directly the resulting code. Melbourne and Pugmire have also suggested the construction of a microprogram control computer, the machine code of which is the language Fortran itself (Melbourne, 1965).

This paper resembles the above mentioned ideas in its vision that software compilation is inefficient, and some of its functions, such as symbol search, statement analyze, and others, are definitely replaceable by hardware. It is different in its design philosophy which is to construct with state-of-the-art hardware a hardware compiler (HC) that can be used by a conventional computer to compile a Fortran program. The HC converts the source program into a set of intermediate code which is designed to allow easy conversion to any actual machine codes by software residing in the main computer.

It is the objective of this paper to give a detailed systems design of the HC, with the design goal that it will be able to serve many different types of computers. In other words it is not machine dependent. Only the state of the art will be used throughout the paper in the design of the HC.

Feasibility Study of Hardware Compiler

Any practical design project should begin with a feasibility study of the product to be designed. Whereas a precise market analysis of the hardware compiler is complex and out of the realm of this paper, an insight into its probable market potential can be derived by making certain reasonable estimates.

Assume that a time-sharing computer system S1 is overloaded and hence yielding poor response time. Two kinds of investments can be made to improve the system's throughput. First, a faster and more expensive computer can be used in place of the existing computer (let the new system thus formed be called S2). Second, a hardware compiler may be installed to increase the speed of Fortran compilation. The tradeoffs of these two investments may be examined.

Let \$X be the leasing cost per month of S1, \$Y be that of S2, and \$Z be that of the hardware compiler (HC). Also let

$$\frac{\text{speed of S2}}{\text{speed of S1}} = a$$

and

$$\frac{\text{Compilation speed of S1 with HC}}{\text{Compilation speed of S1 alone}} = b$$

and let P be the percentage of S1's load that belongs to compiling.

Then, from an investment point of view, the amount of money invested in compiling using S2 is

$$C1 = \$Y \times \frac{P}{a}$$

and the amount of money invested in compiling using S1 and HC is

$$C2 = \$X \frac{P}{b} + \$Z$$

The curves of these equations as a function of P are drawn in Figure 1. It shows that if the compilation load exceeds P_c the hardware compiler is definitely a better investment, owing to the fact that over this region less money has to be spent on compiling if the hardware compiler is used.

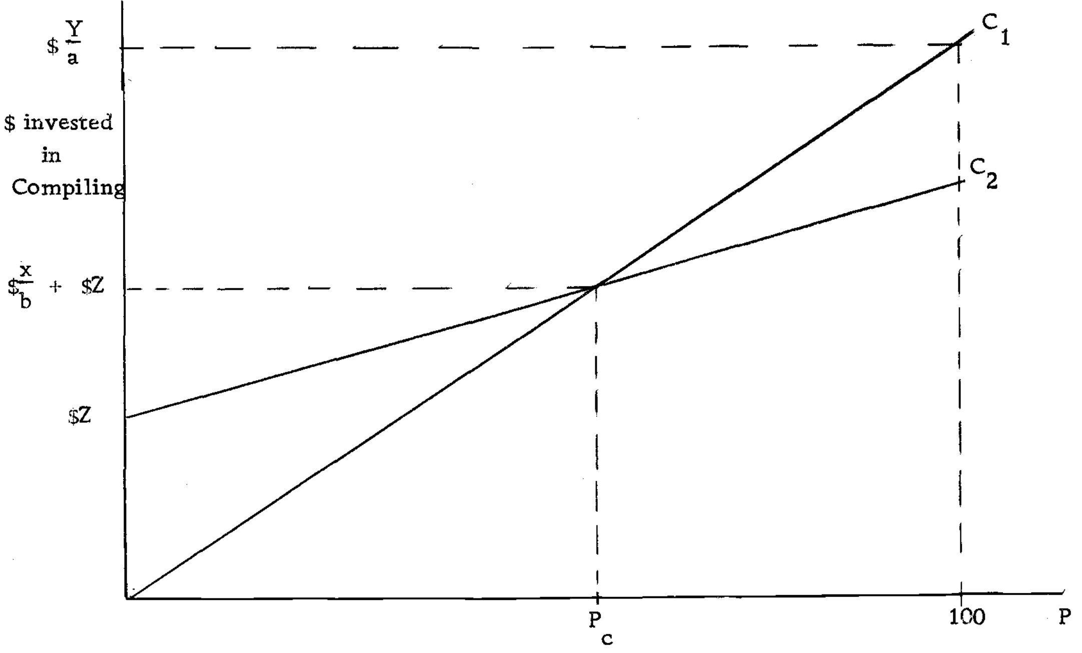


Figure 1. Plot of C_1 and C_2 versus P

As an example of a typical computing facility, the Oregon State University's OS-3 system may be examined. The leasing cost per month of the system is about \$35K, \$15K of which goes to renting the CDC-3300, which is the computer of the system. Replacing the CDC-3300 with CDC-3500 (which leases at \$20K per month) and keeping the existing peripheral facilities would lead to a new system leasing cost of \$40K per month.

It can be safely assumed that

$$\frac{\text{speed of CDC-3500}}{\text{speed of CDC-3300}} \leq 1.5 \text{ .}$$

It can also be safely predicted that

$$\frac{\text{speed of compilation of CDC 3300 with HC}}{\text{speed of compilation of CDC-3300}} \geq 5$$

Five is a conservative guess in view of the fact that the hardware compiler takes charge of symbol searching, statement analyzing, and other time-consuming tasks of compilation, leaving only rather straight-forward jobs for the main computer. Studies show that at least 30% of the central processing unit time of OS-3 is being spent on Fortran compilation. Assuming these values for X, Y, a, and b, and P_c, the leasing marketable price per month of the HC may be found:

$$\$40\text{K} \times \frac{0.3}{1.5} \leq \$35\text{K} \times \frac{0.3}{5} + \$Z$$

$$\therefore \quad \$Z \leq \$5.9\text{K/month}$$

To approximate the actual marketable price (not the leasing price) of the hardware compiler, the ratio R between the actual price and rental price per month must be known. Let this ratio be the same as that for OS-3. It is known that OS-3's actual cost is about \$1.5 million. Hence,

$$R = \frac{\$1.5 \times 10^6}{\$3.5 \times 10^3} \simeq 40$$

Using this value for R, the actual marketable price for the hardware compiler is:

$$40 \times \$5.9K \leq \$236K$$

This means that the design should shoot for a selling price of no more than \$236K and a compilation speed increase of at least 5 over a conventional software compiler.

II. SYSTEM CONCEPT OF THE HARDWARE COMPILER

The hardware compiler should be viewed as a special purpose processor whose function is to convert a source Fortran program into an intermediate code (c-code) which is easily convertible by the main computer's software to its actual machine code. In the design of the HC, the following design principles are applied:

1. The HC should be able to interface with most word oriented general purpose computers and still perform it's useful function as an intermediate code generator.
2. Since this is a preliminary design attempt, system modularity and flexibility rather than elegance are stressed to enable easy design moderations later.
3. Realizing a complete design project in its detailed glory is an impossible undertaking for a paper this size, only important design concepts will be given. Examples will be used whenever practical to illustrate these concepts and how they may be implemented.
4. State-of the-art hardware components and concepts are used as a rule to demonstrate the credibility of the systems structure.

The discussion of the system design will be in the following logical sequence:

1. main computer and HC's relation;
2. block structure of the HC and its different functional units;
3. detailed discussion of each functional unit, its control and its hardware components. (Flow chart and state diagram description will be used to indicate the logic flow of the control of each functional unit).

III. MAIN COMPUTER AND HC'S RELATION

Host Machine's Responsibility

As far as the system program of the host machine is concerned, the hardware compiler (HC) is just another peripheral device and should be treated as such. When a Fortran control statement, one which requests Fortran compilation, is read, the host machine executes the statement by first energizing the specified input device which contains the source program. It then reads the whole program into its memory. During reading, the host machine also notes the type of code with which the program is written, ASCII, BCD, Hollerith, or others, and inserts end of record (EOR) marks to separate one statement from another.. It also inserts an end of file (EOF) mark when it reaches the end of the source program. Once this process is done, the host computer stores the whole program in one of its auxilliary storage devices.

The host machine then checks with the HC to see if it is free to operate on the program. If it is free, the host computer will send it the information concerning the code type of the source program and which device has the program in store. The HC will then be energized and left alone to operate on the program. If the HC is found busy working on another program, the host machine will put the code type and device address information of the source program in its compilation waiting list. It will then resume with its other operations.

Hardware Compiler's Job (Interrupt Request)

As soon as the hardware compiler finishes processing one Fortran program, it will send an interrupt request to the main computer. The instant the main computer is free to service this request, it will first check with its compilation waiting list to see if any other program is waiting to be compiled. If there is, the information concerning its code type and device storage address will be passed to the HC which will then be energized once again. The main computer's compilation routine will then be activated to take over the final stage of the compilation. If there is no program in the waiting list, the HC will not be reenergized.

Description of the Code Generated by the HC

The C-code generated by the hardware compiler has already been claimed to be easily convertible to actual machine codes for most general-purpose, word-oriented computers. This section testifies to that claim by giving the set of codes that the HC will generate for certain types of Fortran statements and how the host machine's software can convert them into actual machine instructions.

A sentence generated by the hardware compiler is composed of words of 16 bits each. These words have two fields, the identification field, and the value field. Table 1 gives the meaning of a C-word and its two fields.

The identification field is 6 bits long and gives the classification of the word, whether it is a variable, statement number, Fortran command, constant, or an operator. For symbols like variables, statement numbers, and constants, their value parts are their addresses in the symbol table of the hardware compiler. It will be shown later that since there is an one to one correspondence between the symbol table of the hardware compiler and that of the host computer, transformation from one symbol table address to another is direct and simple. For symbols like operators (+, =, -, and so forth) or command identifiers (GO TO, IF, DO), the 10 bit value field simply uniquely defines the nature of the operator or command.

Before discussing the C-code produced by the HC, it is proper to describe the memory organization of the host computer which has a conventional compiler (this is just an example). Figure 2 is a simplified diagram of the memory organization of such a machine. The memory is divided into a compiler area (containing the routines that do the compiling), symbol table area, program area, and the I/O area. The compiler area contains the software routines of the main computer of instructions found in the I/O area of the memory. The translated

Table 1. Meaning of words produced by the hardware compiler.

Type of word	6 bits Identification field		10 bits Value field
undefined statement number	0	0	address
defined statement number	0	1	address
DO range number undefined	0	2	address
DO range number defined	0	3	address
simple variable	1	0	address
subscripted variable	1	1	address
simple formal parameter	1	2	address
subscripted formal parameter	1	3	address
subroutine name	1	4	address
constant	3	0	address
operator	4	0	internal code
relational operator	4	1	internal code
command name	5	0	internal code

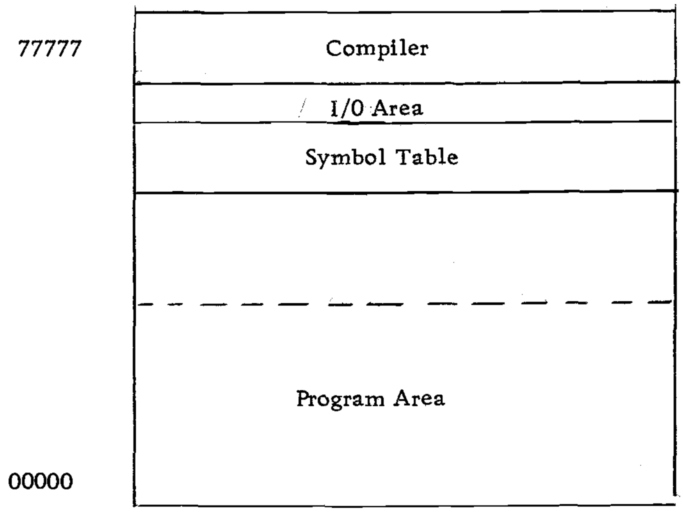


Figure 2. Simplified diagram of the memory organization of the host machine

results are stored in the program area which has a pointer, Program Counter, that addresses the next empty space of the program area. The symbol table contains the run time address of every symbol used in the program.

Having discussed briefly what the organization of memory may be like in the host machine, the following program segment is used to illustrate the usefulness of the code produced by the HC.

Let the original program segment be

```
          D = 1 ;
          C = 2 ;
          A = 1. 4 ;
12        B = A + C * D
..................
..................
          GO TO 12 ;
```

The code produced by the HC for the segment of the program looks like,

```
[=]      [D]     [const]  [1]  [:]   [;]    . . . . . . 1
[=]      [C]     [const]  [2]  [:]   [;]    . . . . . . 2
[=]      [A]     |const|  [1]  [.]   [4]   [:]  [;]  . . . 3
[12]     [X]     [ C   ]  [D]  [T1]  [+]   [T1] [A]  [T1]
[=]      [B]     [ T1  ]  [T1] [ ;]  . . . . . . . . 4
[GOTO]   [12]    [ ;   ]    . . . . . . . . 5
```

The software of the host machine should recognize the following properties of the above set of code:

1. A, B, C, D are variables.

2. 12 is a defined statement number.

3. [const] is a symbol that indicates what follow are digits of a constant and ":" delimits the constant.
4. The symbol ";" indicates the end of one instruction.
5. The symbol GOTO indicates the instruction being one of the Fortran commands.

With the above salient features in perspective, the host machine should treat the generated code of instruction 1 by the following procedure:

1. input the statement into the I/O area of the memory.
2. perform address transformation S(X) on all statement numbers, variables, and constants, where S is the address transformation and X is the symbol table address generated by the HC.
3. convert constants into binary representations and store them in symbol table .
4. generate machine code for the instruction.

Instructions 2 and 3 are handled similarly and at the end of compiling instruction 3, the memory of the main computer will look like,

Symbol Table		Program Area	
Address	Contents	Address	Contents
....	....	0000	LDA S(1)
....	....	0001	STA S(D)
S(D)		0002	LDA S(2)

Address	Contents	Address	Contents
....	....	0003	STA S(C)
S(1)	binary rep. of 1	0004	LDA S(1.4)
....	....	0005	STA S(A)
S(C)			
.....	.....		
S(2)	binary rep. of 2		
.....	.....		
S(A)	.....		
S(1.4)	binary rep. of 1.4		

Instruction 4 is handled differently because it is preceded by a statement number (12). The host machine's software should

1. find the address transformation S(12),
2. store the contents of the program counter (which equal to 0006 at this time) in S(12)
3. find transformations, S(B), S(C), S(A), S(D), S(T1),
4. generate machine codes.

The following program contents will be augmented to the one shown above,

Symbol Table		Program Area	
Address	Contents	Address	Contents
S(12)	0006	0006	LDA S(D)
S(B)		0007	MUA S(C)....Multiplication

Address	Contents	Address	Contents
. . .		0008	STA S(T1)
S(T1)		0009	LDA S(A)
		0010	ADA S(T1) - - Addition
		0011	STA S(T1)
		0012	LDA S(T1) - - - Equation
		0013	STA S(B)

When the main computer's software encounters

```
GOTO  12  ;
```

it will do the following,

1. Perform address transformation S(12),
2. Fetch 0006 from S(12),
3. Generate machine code for the instruction.

   UJP 0006 (unconditional jump to address 0006).

IV. BLOCK STRUCTURE OF THE HC

The block structure of the hardware compiler is shown in Figure 3. It consists basically of five different functional blocks:

1. Input block is responsible for
   a. recognizing any illegal characters that may be present in the source program.
   b. converting the external program code, be it ASCII, BCD, or Hollerith, into an internal code which is recognized by the rest of the functional units.
2. The Scanner is responsible for identifying the statement type of an instruction and energizing the proper one of the Lexical Analyzer Units.
3. Lexical Analyzer Units (LAU's) are responsible for converting every source program into a string of words in the S-language in which every symbol in the source program is replaced by its special identifier.
4. The Syntax Analyzer Unit (SAU) takes the string of words produced by each LAU and generates an intermediate code for it.
5. The Output Unit transfers the output of the HC, intermediate code or error messages, to a special storage device and issues an interrupt request to the host machine.

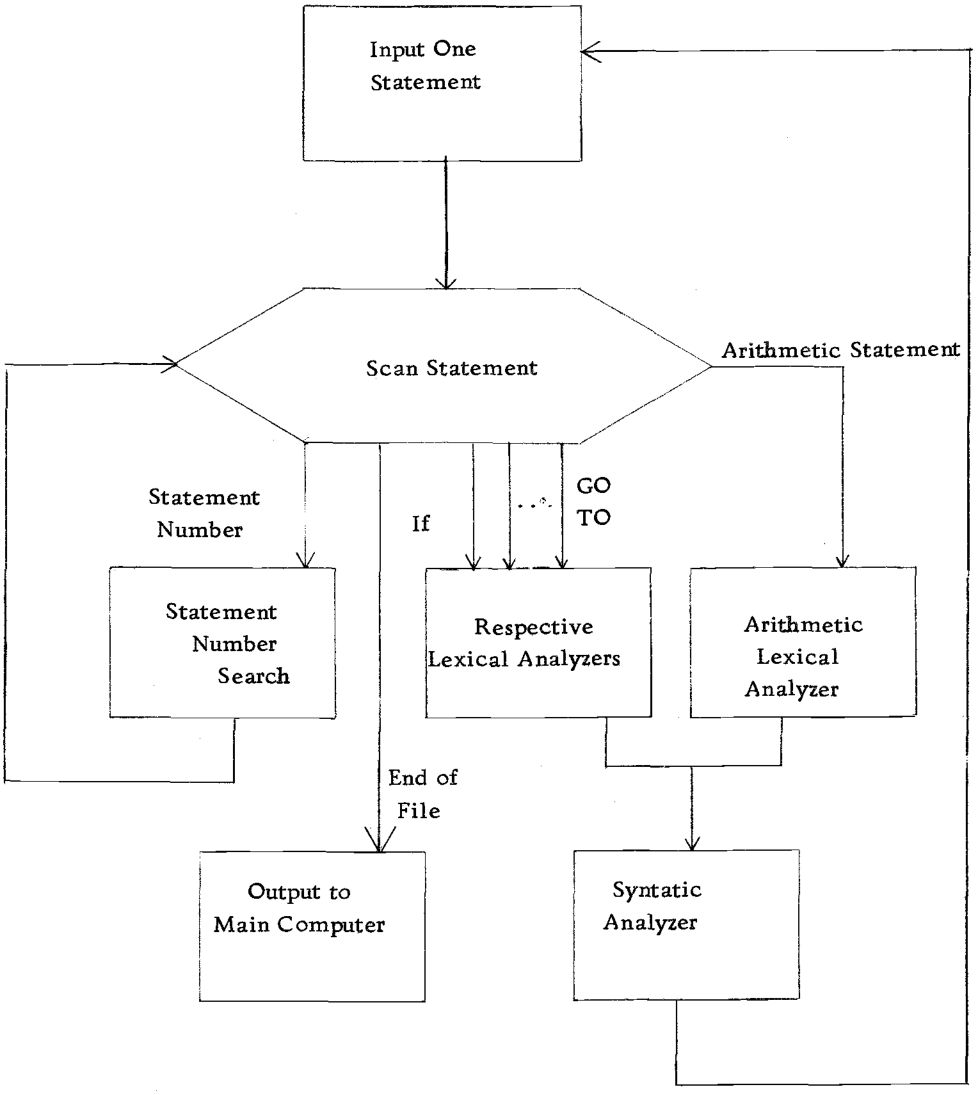


Figure 3. Block diagram of hardware compiler

Since each functional unit, except for the scanner, comprises some kind of code conversion process, it is evident that storage has to be provided for the resulting code. Design modularity implies the use of separate memory modules for each different resulting code. Table 2 summarizes some of the different memory units that are present and their respective functions.

Table 2. A cross reference between HC's memory modules and their respective functions.

Memory Module	Function
Input Buffer Memory (IBM)	Input internal code storage
Syntax Memory (SM)	S-language storage
Code Memory (CM)	Intermediate code storage
Error Message Memory (EMM)	Error message and line number of the instruction associated with it

In this design semiconductor memory modules are used instead of more common core memory units because of the following considerations:

1. The speed of semiconductor memory can be up to 5 times that of a core memory.
2. All the memory modules used in the HC are figured to be less than 10K bits in capacity. Up to that memory capacity,

cost per bit is cheaper when semiconductor memory units are used (Moore, 1971).

3. Power failure is a great concern for the use of semi-conductor memory, because of the volatile nature of the device. However, since all memory modules are in fact scratch pad memories, power failure is not a fatal problem during the operation of the HC.

In addition to these memory modules, there are other special hardware components (code converters, shift registers, push-down-list memory units, etc.) associated with each functional unit. These components are designed to carry out those operations required of the functional unit. To control the functioning of these components, there exists for each functional unit a microprogram control unit. The discussion of the characteristics of the hardware components and the control units will form the bulk of the paper.

V. HARDWARE IMPLEMENTATION OF A CONTROL UNIT

The control of each functional unit described in the above section operates in accordance with an algorithm which can also be thought of as the sequence of steps followed by a finite state machine. Since the algorithm is complicated, its hardware realization usually involves a sequential machine of numerous states, inputs and outputs. The success of producing such a machine depends greatly on the availability of inexpensive and fast hardware components. Advanced medium scale integration (MSI) and large scale integration (LSI) components have provided such an availability, thus giving more credence to the belief that most software routines can be replaced by hardware with good cost and performance results. Such ideas have been expressed by many researchers and seem to represent the current trend of thoughts in the computer industry.

The technique used in this paper to implement each control unit resembles closely microprogramming as understood in the industry and more closely still one of the hardware implementation approaches expressed by Thurber *et al.* (1971).

The building block of a control unit is the Programmable Logic Array (PLA) currently marketed by Texas Instruments (TI). The design of each control unit is based on the fact that any algorithm can be implemented by a finite state machine. It is also observed that all finite state machines are describable by logic equations in their canonical form (sums of products). These sums of products may then be realized by a PLA. Figure 4 is a block structure of a PLA.

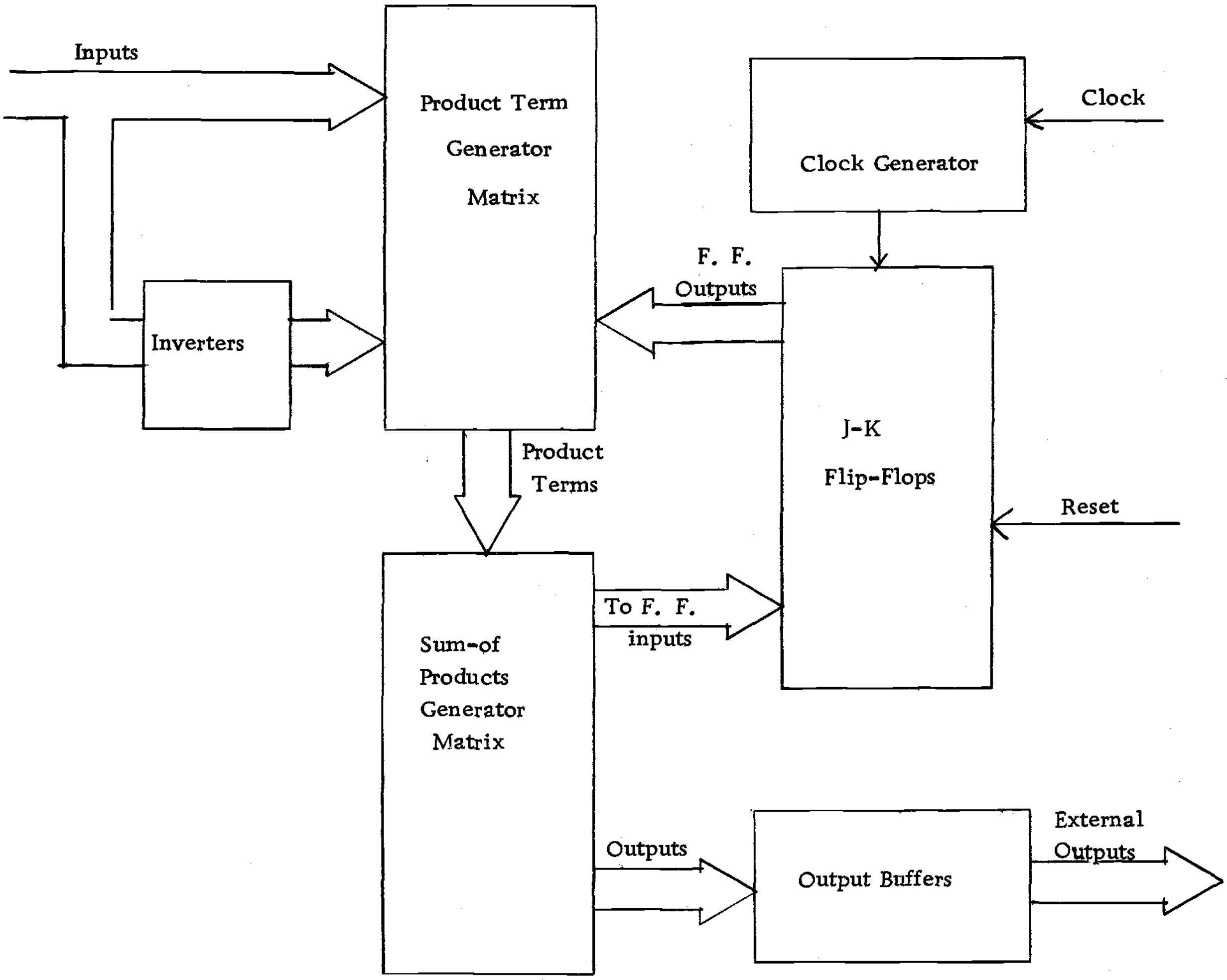


Figure 4. Block diagram of TI's programmable logic array

As can be seen in Figure 4, if a logic expression is written in its canonical form then the PLA has

1. a first programmable matrix to generate the product terms

(AND matrix); this matrix can also be thought of as an address decoder.

2. a second programmable matrix to generate the sum of products (OR matrix); this matrix can also be thought of as containing a set of micro-instructions.
3. J-K flip-flops used in feedback loops to permit implementation of sequential logic.

A detailed description of the concepts and construction of a control unit is given in Section VII.

VI. INPUT FUNCTIONAL UNIT

Introduction

In the design of the input functional unit, it is assumed that the host machine has the necessary interface facilities to provide inter-peripheral communication between the HC and any existing auxilliary storage device that may have a source program in store. All the HC has to do to initiate inputing a Fortran program from the storage device is to send the interface the address code of the input device and the code with which the program is written. The former information enables the interface to energize the correct storage device and the latter enables the interface to store serial input from the storage device in its buffer register until enough bits are accumulated. These bits will then be transferred in parallel to the input latch of the hardware compiler. In addition, the interface may be inhibited temporarily from reading further from the storage device when it receives a READ INHIBIT signal from the HC.

A transferral of new information from the interface is accomplished by a TRANSMIT READY (TRX) signal which serves to tell the HC that a new character, is at the output lines of the input latch, and is ready to be processed. The block flow chart of the input control unit and the block diagram of the hardware associated with the input unit are shown in Figure 5 and Figure 6 respectively.

Block Explanation of the Input Control Unit

The block diagram of the logic flow of the Input Control Unit is shown in Figure 5. Since more than one input code of a source program is acceptable, the input control unit (ICU) has to convert each kind of code into one internal character code. Table 3 shows this character code and its meaning. To do this code conversion, the ICU energizes one of its code converters.

Looking at Figure 5, it can be seen that the ICU carries out its job by first sending the interface the information concerning the code type and storage device of the source Fortran program. It then reads one character at a time from the interface, does code conversion on the character, and then checks to see if the character is legal. If it is legal, the ICU will store its internal character code in the input buffer memory (IBM). If the character is illegal, the ICU will simply skip the instruction.

When the ICU reads an end of record (EOR) mark, it will store the internal code ; in the IBM and considers its job done for the time being. A more detailed description of the ICU and its way of handling an input program in Hollerith will be shown in a later section.

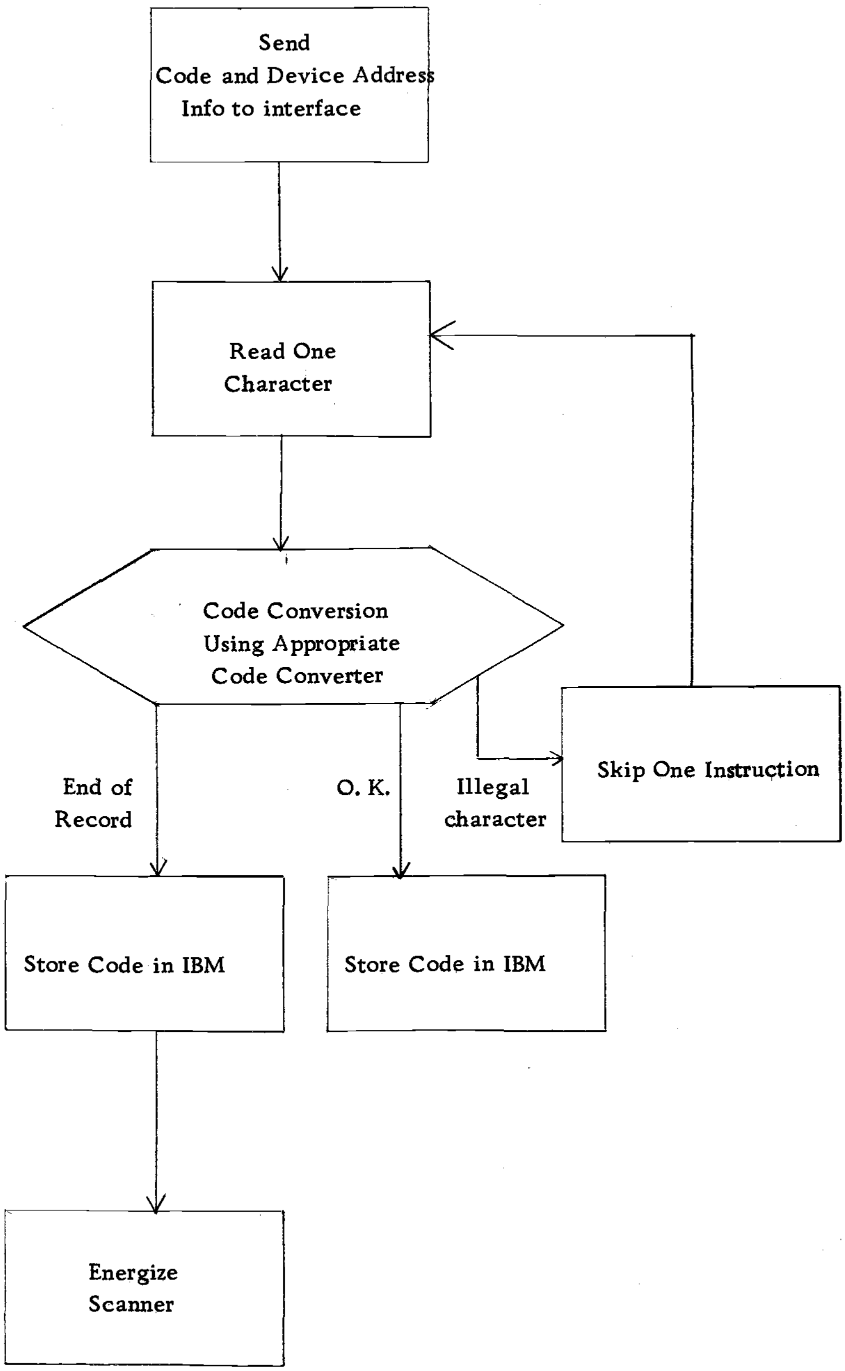


Figure 5. Block flow chart of input control unit

Table 3. Internal code vs. character.

code	character
00	0
01	1
02	2
...	...
11	9
12	A
13	B
...	...
43	Z
44	.
45	,
46	=
47	+
50	-
51	*
52	/
53	(
54	)
55	;
56	
57	
...	...
...	...
77	illegal

Hardware Components of the Input Unit

The block diagram of the hardware components used by the ICU is shown in Figure 6. The following is a brief description of the function of each component:

1. Code converters: There are three code converters, the ASCII-internal, BCD-internal, and Hollerith-internal code converters. Each code converter is so designed to convert

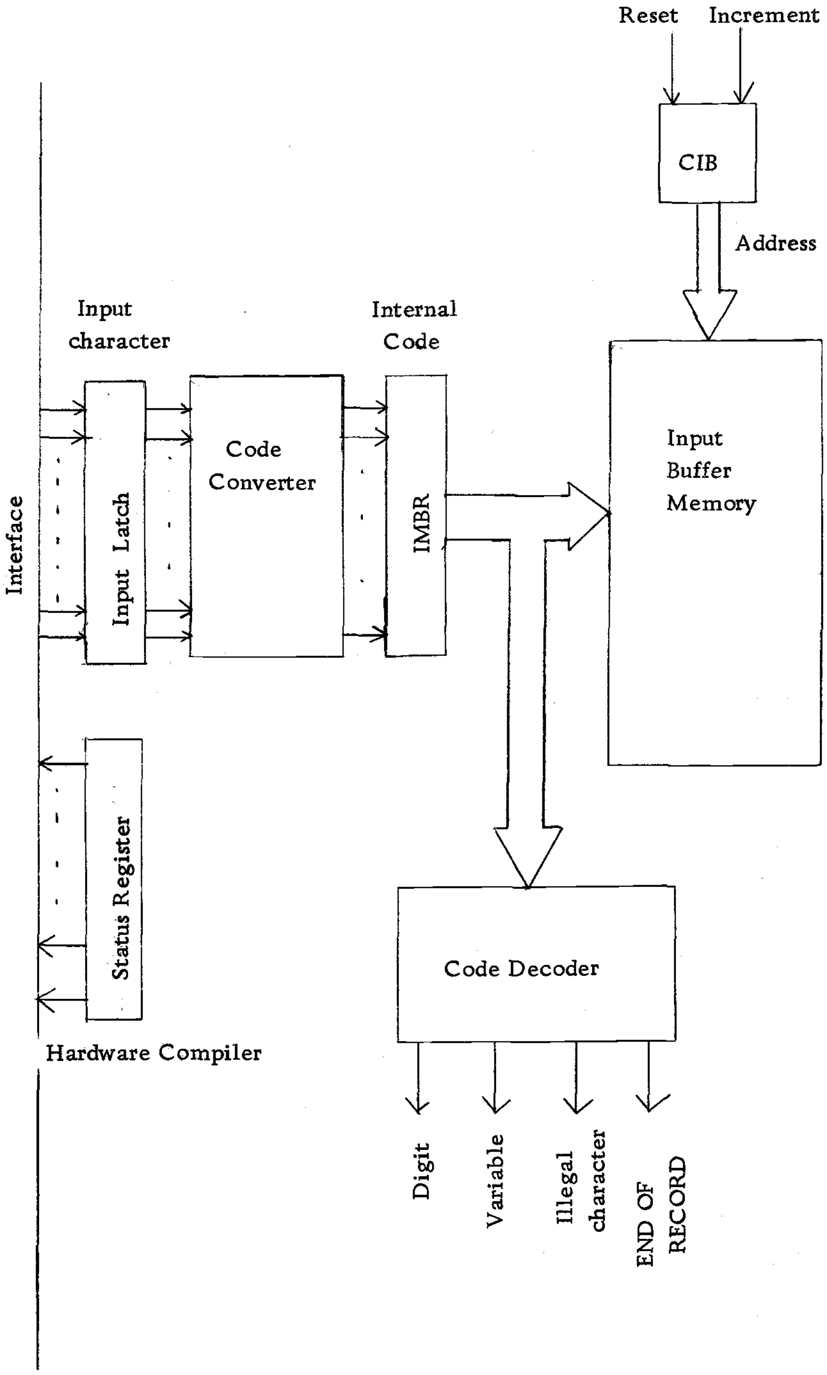


Figure 6. Block diagram of input functional block

the set of all illegal characters to 77_8 (see Table 3). These code converters are similar to TI's TMS 2603, so their construction is well within the state of the art.

2. Input buffer memory (IBM): The IBM is a 256 word, 6-bit-per word, random access, semiconductor memory. The word size of the IBM is quite arbitrary but the word length is determined by the number of bits in the internal character code, which is six. The word size, set at 256 words, is chosen because of the observation that this memory size is common in metal oxide semiconductor (MOS) memories, and the assumption that no Fortran instruction will have more than that number of characters. Typical Read/Write cycle time of such memory units is 600 ns.
3. Input buffer memory register (IBMR): It is a 6 bit register which contains the word to be written in the IBM preceding a WRITE operation. It also contains the word read from the IBM after a READ operation.
4. Counter input buffer (CIB) is an 8 bit binary counter. It can be incremented and decremented and reset. Its contents is the binary address of a character in the IBM. Its address decoder determines which memory cell is to be addressed during either a READ or a WRITE operation.

5. Decoder (DC): The 6 bit to 64 output line decoder is used extensively by the HC's control units. Its job is to take a 6 bit internal character and to decode it into the output line which corresponds to the internal code. Figure 7 is a block diagram of the DC. Some of the DC's output lines are or'ed together, so that the or'ed output will be known collectively as digit, letter, or special characters.

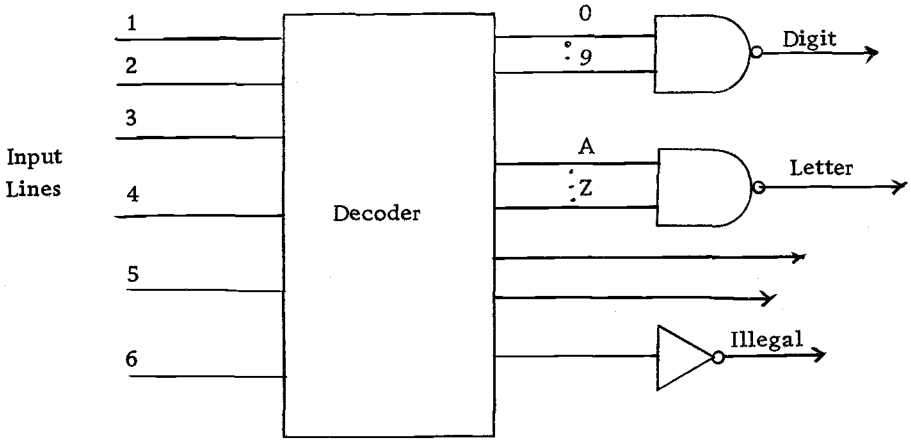


Figure 7. Block diagram of decoder DC

MSI decoders using TTL logic currently sold in the market give inverted outputs. In other words, the output line which corresponds to the input code will be the only line low while the rest of the output lines remain high. To illustrate, Figure 8 is an example. Figure 8 shows a 2 line to 4 line decoder with some of its output lines connected together to perform the required or'ed function.

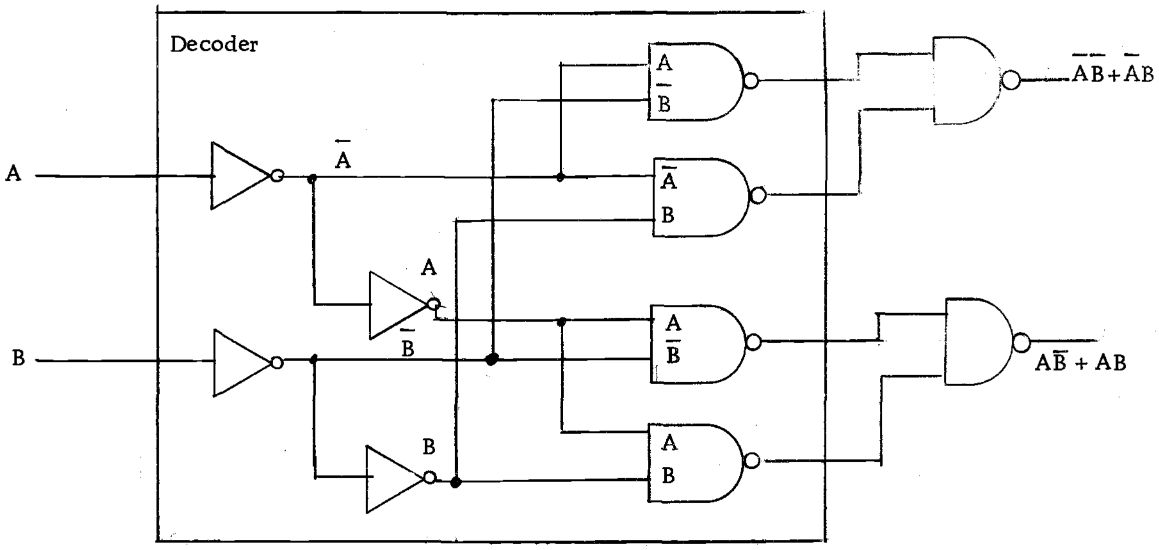


Figure 8. Example of a simple decoder

6. Line Counter (LNC) is a binary counter which stores the number of lines that have already been compiled.

7. Counter N (CN) keeps track of the number of characters that have been input so far.

8. HC Status word register (STWR) contains the information of the input code and address of storage device sent by the host computer. There is also a busy bit which indicates that the HC is busy when it is flagged.

9. Error Message Memory (EMM): The EMM contains error messages issued by the various control units. Each error message contained in the EMM is accompanied by its line number which indicates with which instruction the error message is associated. The address counter of EMM

(CEMM) contains the binary address of the next available a memory cell of the EMM. The EMM buffer register (EMMBR) provides the necessary buffer for the EMM during either a read or write operation.

10. The input latch (INLH) provides temporary storage for one input character from the interface. Its output will follow the input data only when the latch enable clock is high.

Detailed Discussion of the Input Control Unit Particularly Its Handling of Hollerith Input

Figure 9 is the detailed flow chart of the ICU and its way of handling Hollerith input code (other input code is handled similarly). Figure 9 shows that the ICU first sends those contents of its status word register (STWR) concerning the code type and the storage device address of the source program to the interface of the host computer. It then waits for the interface to send one character to it (this corresponds to the read operation shown in the flow chart). That happens when the interface gives out a TRANSMIT READY signal. The ICU then energizes the Hollerith-internal code converter to convert the character to its internal code representation (code convert). This code is stored in the IBMR. To find out what kind of character it is, the ICU energizes the decoder (DC).

To be acceptable, Hollerith or card input must have the following features:

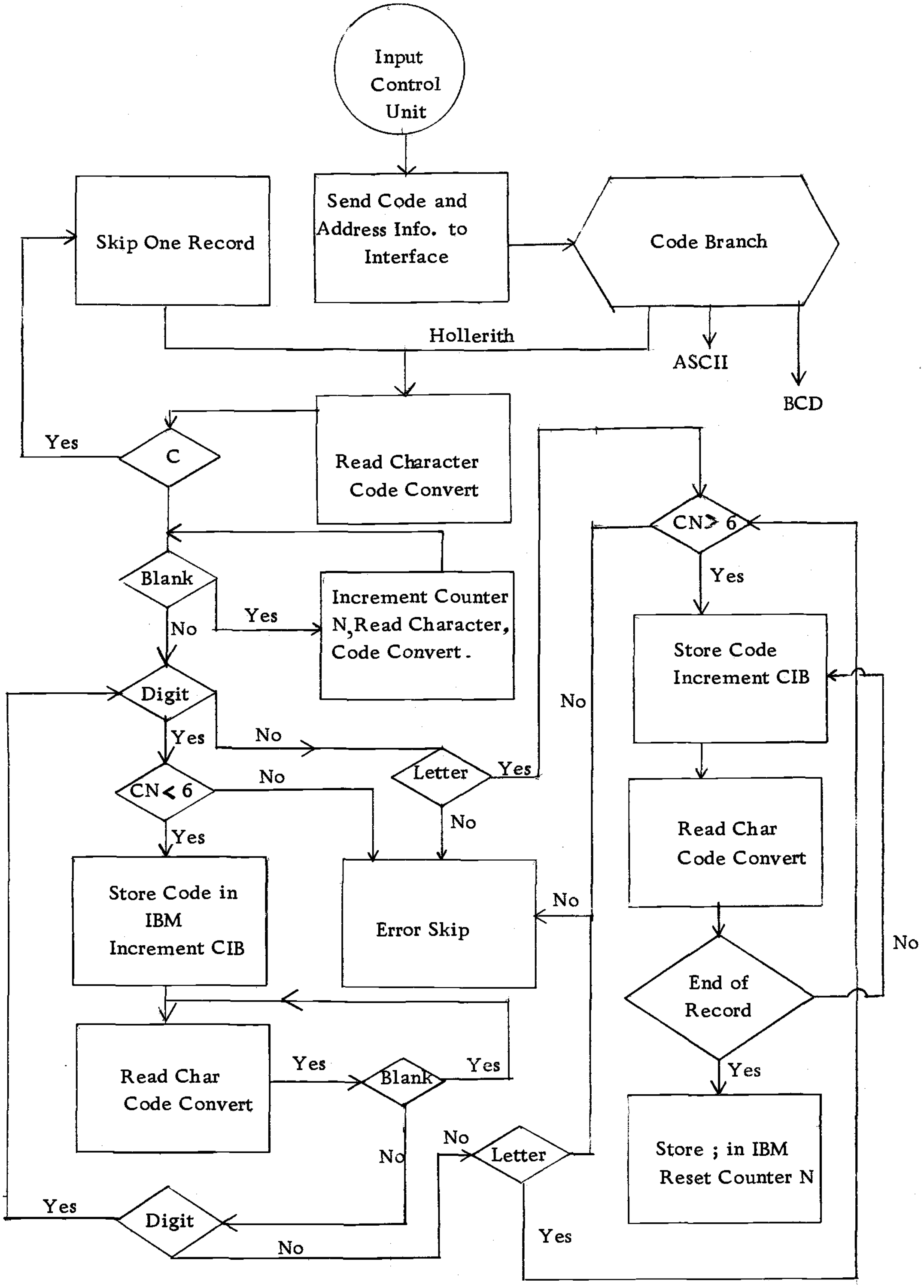


Figure 9. Flow chart of input control unit

1. On an input card, the first column must be either a blank or contains a number or the letter C. The presence of a number means that this is the first character of a statement number. The presence of "C" means that the card is meant to be a comment card and hence should not be compiled.
2. Only blanks or numbers are allowed to be present in columns 2 to 6.

The flow chart of the ICU shown in Figure 9 is quite self-explanatory. Some examples may help to clarify how that algorithm works.

Assuming card inputs are of the forms:

Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Card 1	1	3							A	=		B			EOR
Card 2	C			T	H	I	S		I	S	C				
Card 3									1	2		A	....		
Card 4		A	=	....											

The ICU will recognize that card 2 is a comment card and will skip over it. It will also recognize that card 3 and 4 are not properly constructed and will store error messages in the EMM and skip the rest of the instruction. Only card 1 will be recognized as legal and will be stored in the input buffer memory in its internal form.

VII. HARDWARE IMPLEMENTATION OF THE INPUT CONTROL UNIT

Procedure of Implementation

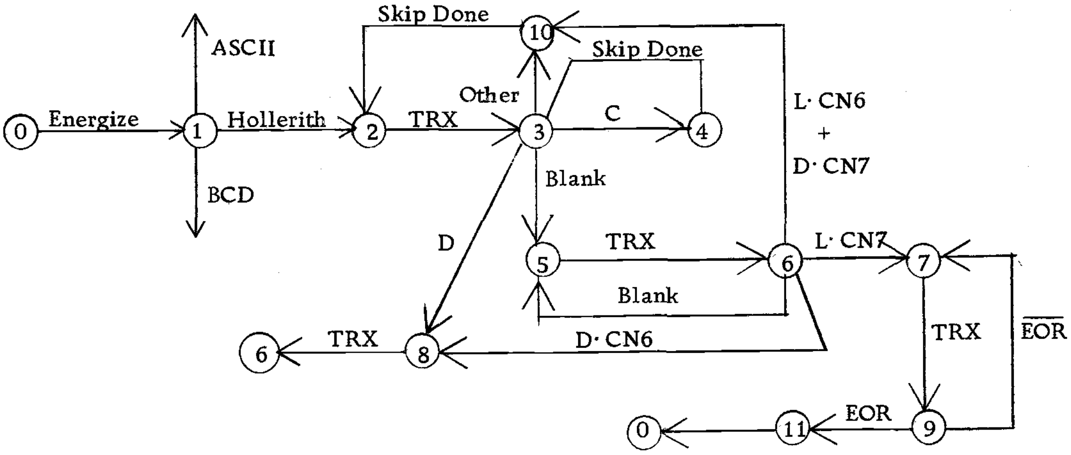
The steps which must be taken to implement the ICU or any other control units by Programmable Logic Array (PLA) are as follows:

1. convert the flow chart description of the control unit into its state diagram equivalent of a sequential machine;
2. construct from the state diagram the transition table of the machine;
3. derive logic equations which are sums of products;
4. program the PLA to generate these sums of products. (This is a customs fabrication process done by the manufacturer.)

Each step will be examined in the following sections.

Conversion of the Flow Chart of the ICU to State Diagrams

The state diagram of the ICU is shown in Figure 10. The conversion of the flow chart description of the ICU to the diagram in Figure 10 is quite straightforward. Each state of the diagram corresponds to a processing block in which an ordered sequence of actions will take place. These actions are actually control pulses that enable the hardware components to perform their designed functions. The



Legends: TRX = TRANSMIT READY signal from the interface
CN6 = (Counter N) less than 6
CN7 = (Counter N) greater than 6
D = digit
L = letter
EOR = end of record mark
Skip done = done signal from the skip record unit

Figure 10. State diagram of ICU

Table 4. Actions associated with each ICU state.

State	Actions
0	Idle
1	Transfer storage device address and code info to interface, Read enable, Set busy bit of (STWR)
2	No operation
3	Enable code converter (ENCC), enable decoder (ENDC)
4	Energize Skip Record Unit (ENSKR)
5	Increment Counter N
6	ENCC, ENDC
7	Store code in IBM, increment CIB, increment Counter N
8	Store code in IBM, increment CIB, increment Counter N
9	ENCC, ENDC
10	Error type to EMBR, (LNC) to EMBR, write EMM Increment CEMM, energize SKR
11	Increment LNC, energize scanner, inhibit read of interface

actions associated with each state is given in Table 4. As can be seen from Table 4, in state 2 no operation is carried out. State 2 is then an example of a waiting state in which the machine will idle until certain input conditions occur (in this case, it is the presence of the TRANSMIT READY signal issued by the interface). Transition from one state to the other can be caused by the presence of some input conditions or simply as the next operation of the sequential machine after a sequence of prescribed actions have taken place.

In state 4, the skip record unit is energized to skip one record. This control unit functions as a subroutine in conventional programming. Its detailed design description is given in the next section.

Skip Record Control Unit

The skip-record-control unit (SKR) is used extensively by other control units when the occasions arise that one instruction should be skipped. When the SKR finishes its job, it issues a DONE signal to the control unit that has energized it, so that the latter may resume its next course of actions.

There are two things that the SKR must do. First, it has to read from the interface until an EOR mark is reached. All the information thus read will be discarded. Second, it has to erase from the IBM all the information that has previously been stored. The way it accomplishes this is by writing O's in the IBM, starting from the current address in counter-input-buffer (CIB), until the counter reaches zero. Then its job is done.

To demonstrate how a control unit can be implemented by PLA, the step by step design of the SKR unit is given in the following sections.

The flow chart of the SKR control unit is shown in Figure 11. From it, a state diagram of the control unit is constructed which is shown in Figure 12 and Table 5. Using conventional logic design method, from the state diagram in Figure 12, the state transition table of the SKR control unit may be constructed, which is shown in Table 6.

The transition table is divided into five columns. Each entry in the input column indicates the kind of input condition being applied to the present state of the sequential machine or control unit. Each entry in the timing level column determines which control action has to take place during the current phase of the present state. This is required because during any given state, a sequence of actions must take place in an orderly manner, and some means has to be developed to insure the right action at the right time. For example, in state 3 the following actions must take place in the order given below:

0 to IMBR, Write (IBMR) in IBM, Decrement CIB.

Therefore, there are three phases to state 3.

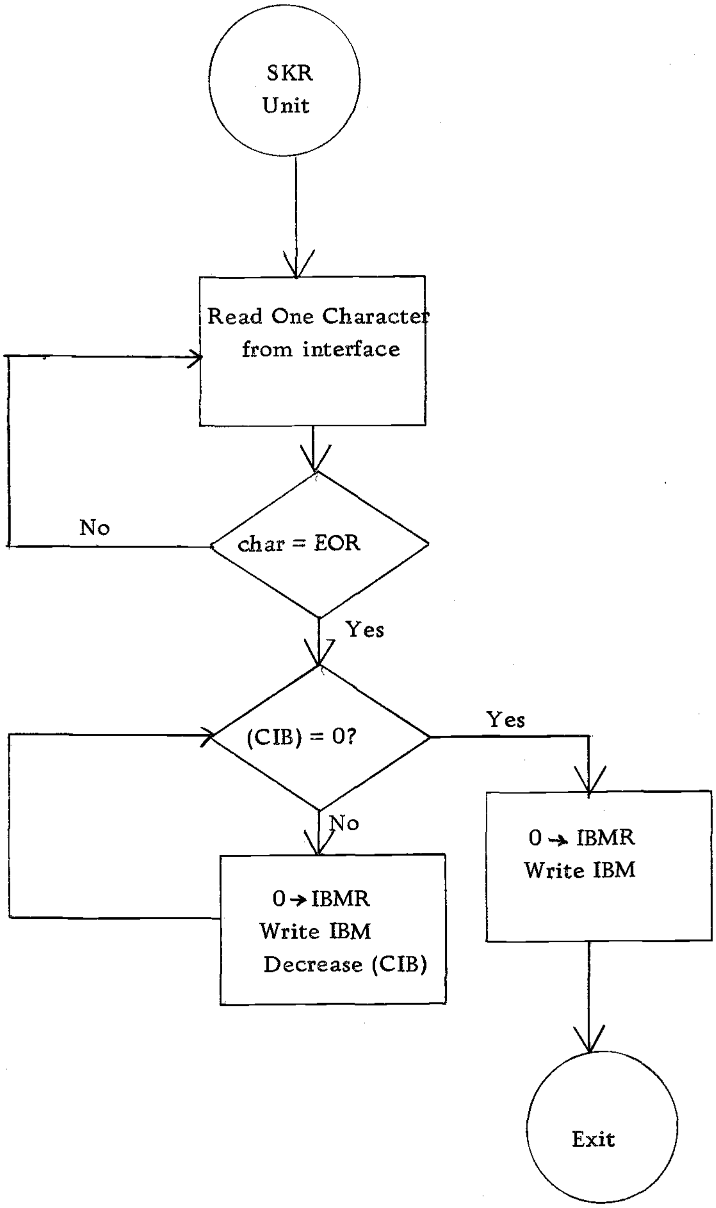


Figure 11. Flow chart of skip record unit

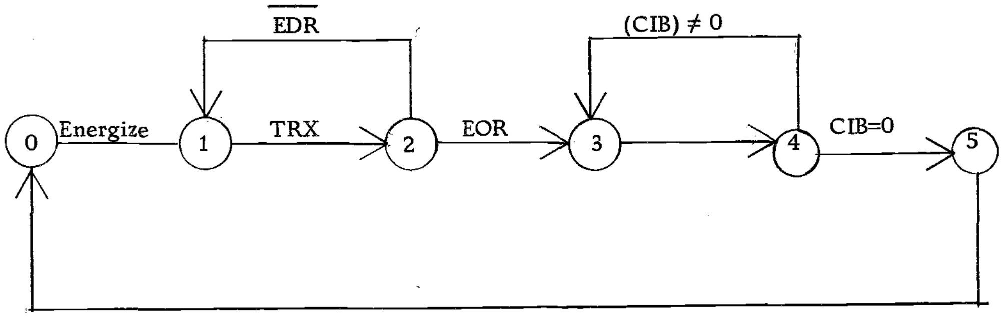


Figure 12. State diagram of SKR unit

Table 5. Actions associated with each SKR state.

State	Operations
0	Idle
1	No operation
2	ENCC, ENDC
3	0 to IMBR, write IBM, decrement CIB (DCIB)
4	No operation,
5	0 to IMBR, write IBM, DCIB increment LNC, skip done

Table 6. Transition table of SKR. Legend - REBC: reset Binary Counter x: Don't care

Timing Level	Code	Input	Code	Present State	Code	Next State	Code	Output	Code
x		$\overline{\text{energize}}$	0 0 0	0	0 0 0	0	0 0 0	No op	0 0 0 0
x		energize	0 0 1	0	0 0 0	1	0 0 1	No op	0 0 0 0
x		$\overline{\text{TRX}}$	0 1 0	1	0 0 1	1	0 0 1	No op	0 0 0 0
0		TRX	0 1 1	1	0 0 1	2	0 1 0	ENCC	0 0 0 1
1	0 0 1	x		2	0 1 0	2	0 1 0	ENDC	0 0 1 0
2	0 1 0	$\overline{\text{EOR}}$	1 0 0	2	0 1 0	3	0 1 1	REBC	0 0 0 0
2	0 1 0	EOR	1 0 1	2	0 1 0	1	0 0 1	REBC	0 0 0 0
0	0 0 0	x		3	0 1 1	3	0 1 1	0 → IMBR	0 0 1 1
1	0 0 1	x		3	0 1 1	3	0 1 1	write IBM	0 1 0 0
2	0 1 0			3	0 1 1	3	0 1 1	DCIB	0 1 0 1
3	0 1 1	x		3	0 1 1	4	1 0 0	REBC	0 0 0 0
0	0 0 0	(CIB) ≠ 0	1 1 0	4	1 0 0	3	0 1 1	No op	0 0 0 0
0	0 0 0	(CIB) = 0	1 1 1	4	1 0 0	5	1 0 1		0 0 1 1
1	0 0 1	x		5	1 0 1	5	1 0 1		0 1 0 0
2	0 1 0	x		5	1 0 1	5	1 0 1	ILNC	0 1 1 0
3	0 0 1	x		5	1 0 1	5	1 0 1	skip done	0 1 1 1
4	1 0 0			5	1 0 1	0	0 0 0	REBC	0 0 0 0
	$N_1N_2N_3$		$Q_1Q_2Q_3$		$X_1X_2X_3$		$P_1P_2P_3$		$Y_1Y_2Y_3Y_4$

Each entry in the present state column indicates what state the control unit is in. This information together with the input and timing level information determines what output action has to take place (shown in output column) and to what state the control unit must go next (shown under the next state column).

Each entry in the table also contains the binary code for the information. Three variables N_1, N_2, N_3 are needed to encode each timing level, three (Q_1, Q_3, Q_3) for each input condition, three (X_1, X_2, X_3) each present state, three (P_1, P_2, P_3) for each next state, and four variables (Y_1, Y_2, Y_3) for each output.

From the state transition table logic equations can be written in their canonical forms and they are shown below:

$$K_1 = \overline{N}_1 N_2 N_3 \;\; \overline{X}_1 X_2 X_3$$

$$J_2 = \overline{N}_1 \overline{N}_2 \overline{N}_3 \;\; Q_1 Q_2 \overline{Q}_3 \;\; X_1 \overline{X}_2 \overline{X}_3 \; + \; N_1 \overline{N}_2 \overline{N}_3 \;\; X_1 \overline{X}_2 X_3$$

$$K_2 = \overline{N}_1 \overline{N}_2 \overline{N}_3 \;\; \overline{Q}_1 Q_2 Q_3 \;\; \overline{X}_1 \overline{X}_2 X_3 \; + \; \overline{N}_1 \overline{N}_2 \overline{N}_3 \;\; Q_1 Q_2 \overline{Q}_3 \;\; X_1 \overline{X}_2 \overline{X}_3$$

$$J_2 = \overline{N}_1 N_2 \overline{N}_3 \;\; Q_1 \overline{Q}_2 Q_3 \;\; \overline{X}_1 X_2 \overline{X}_3 \; + \; \overline{N}_1 N_2 N_3 \;\; \overline{X}_1 X_2 X_3$$

$$\begin{aligned} K_3 = {} & \overline{Q}_1 \overline{Q}_2 Q_3 \;\; \overline{X}_1 \overline{X}_2 \overline{X}_3 \; + \; \overline{N}_1 \overline{N}_2 \overline{N}_3 \;\; Q_1 \overline{Q}_2 \overline{Q}_3 \;\; \overline{X}_1 X_2 \overline{X}_3 \\ & + \; \overline{N}_1 N_2 \overline{N}_3 \;\; Q_1 \overline{Q}_2 Q_3 \;\; \overline{X}_1 X_2 \overline{X}_3 \; + \; \overline{N}_1 \overline{N}_2 \overline{N}_3 \;\; Q_1 Q_2 \overline{Q}_3 \;\; X_1 \overline{X}_2 \overline{X}_3 \\ & + \; \overline{N}_1 \overline{N}_2 \overline{N}_3 \;\; Q_1 Q_2 Q_3 \;\; X_1 \overline{X}_2 \overline{X}_3 \end{aligned}$$

$$J_3 = \overline{N}_1 \overline{N}_2 \overline{N}_3 \;\; \overline{X}_1 \overline{X}_2 X_3 + \overline{N}_1 N_2 N_3 \;\; \overline{X}_1 X_2 X_3 \; + \; N_1 \overline{N}_2 \overline{N}_3 \;\; X_1 \overline{X}_2 X_3$$

$$\begin{aligned} Y_1 = {} & \overline{N}_1\overline{N}_2N_3 \; X_1\overline{X}_2X_3 + \overline{N}_1N_2\overline{N}_3 \; X_1\overline{X}_2X_3 + \overline{N}_1\overline{N}_2N_3 \; X_1\overline{X}_2X_3 \\ & + \overline{N}_1N_2\overline{N}_3 \; \overline{X}_1X_2X_3 + \overline{N}_1\overline{N}_2N_3 \; \overline{X}_1X_2X_3 \end{aligned}$$

$$\begin{aligned} Y_2 = {} & \overline{N}_1\overline{N}_2N_3 \; \overline{X}_1X_2\overline{X}_3 + \overline{N}_1\overline{N}_2\overline{N}_3 \; X_1\overline{X}_2\overline{X}_3 \; Q_1Q_2Q_3 \\ & + \overline{N}_1N_2\overline{N}_3 \; X_1\overline{X}_2X_3 + N_1\overline{N}_2\overline{N}_3 \; X_1\overline{X}_2X_3 \end{aligned}$$

$$\begin{aligned} Y_3 = {} & N_1\overline{N}_2\overline{N}_3 \; X_1\overline{X}_2X_3 + \overline{N}_1\overline{N}_2\overline{N}_3 \; X_1\overline{X}_2\overline{X}_3 + \overline{N}_1N_2\overline{N}_3 \; \overline{X}_1X_2X_3 \\ & + \overline{N}_1\overline{N}_2\overline{N}_3 \; \overline{X}_1X_2X_3 + \overline{N}_1\overline{N}_2\overline{N}_3 \; \overline{X}_1X_2X_3 \; \overline{Q}_1Q_2Q_3 \end{aligned}$$

In the above set of equations that describe the SKR, there are 32 product terms.[1] Since TI's PLA (TMS 2000JC) can generate up to 60 product terms, the implementation of these equations by PLA is within the state of the art. The equations themselves are sufficient information for the manufacturer to fabricate custom PLA. However, extra hardware components are needed to provide the required input conditions and timing levels to the PLA.

To generate the timing level for each machine state, a 4 bit binary counter[2] is used. Figure 13 shows the block diagram of such a counter (TI's SN 5493). For the counter, if R_1 and R_2 are high, the binary counter will be reset and inhibited from further counting.

---

[1] Simplification of logic equations will lead to less product terms.

[2] 4 bit counter is standard, even though only 3 bits are actually needed.

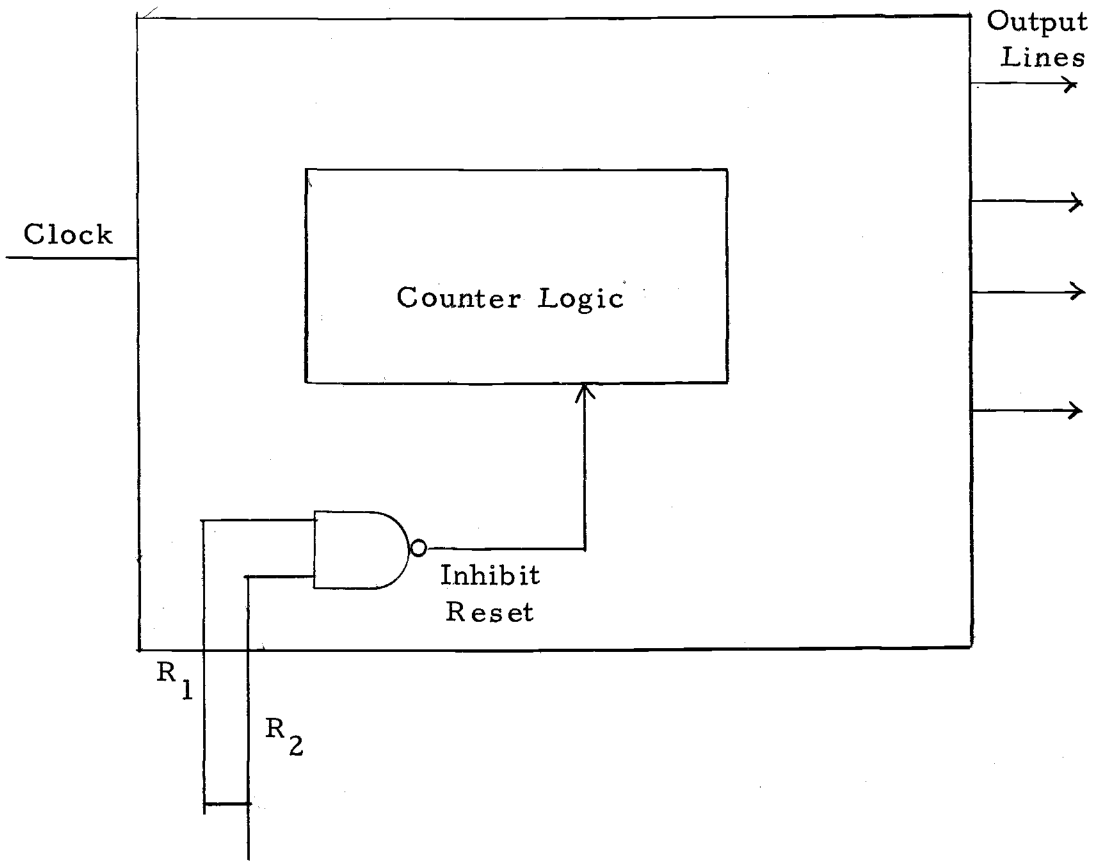


Figure 13. Block diagram of binary counter

Since each input condition is actually a clock pulse, it has to be encoded into its three bit binary representation. A Read Only Memory encoder may be used for that purpose. The output lines of the encoder will give the binary representation of the input condition with which it is fed (see Figure 14 for the block diagram of the input encoder).

Operation of a Control Unit

So far only the sequential machine aspect of the SKR control unit has been mentioned. This section shows that the PLA control unit can also be viewed from a microprogram control standpoint.

The block diagram of a PLA control unit is shown in Figure 14. There it can be seen that the AND matrix of the PLA can be viewed as an address decoder, the output lines of which address the OR matrix which can be viewed as a ROM that contains a set of microinstructions.

Initially, the control unit is in its idle state, which means that the outputs of the J-K flip-flops are all zero's (see Table 6). This condition is decoded by ROM1 into a high signal condition for its output line A. Line A addresses the NO OPERATION AND NO STATE CHANGE command in ROM2. The function decoder decodes this command into a BINARY COUNTER RESET/INHIBIT control pulse. Consequently, the Binary Counter is inhibited from upcounting and its contents will remain at zero. At the same time, the J-K flip-flops are unaffected.

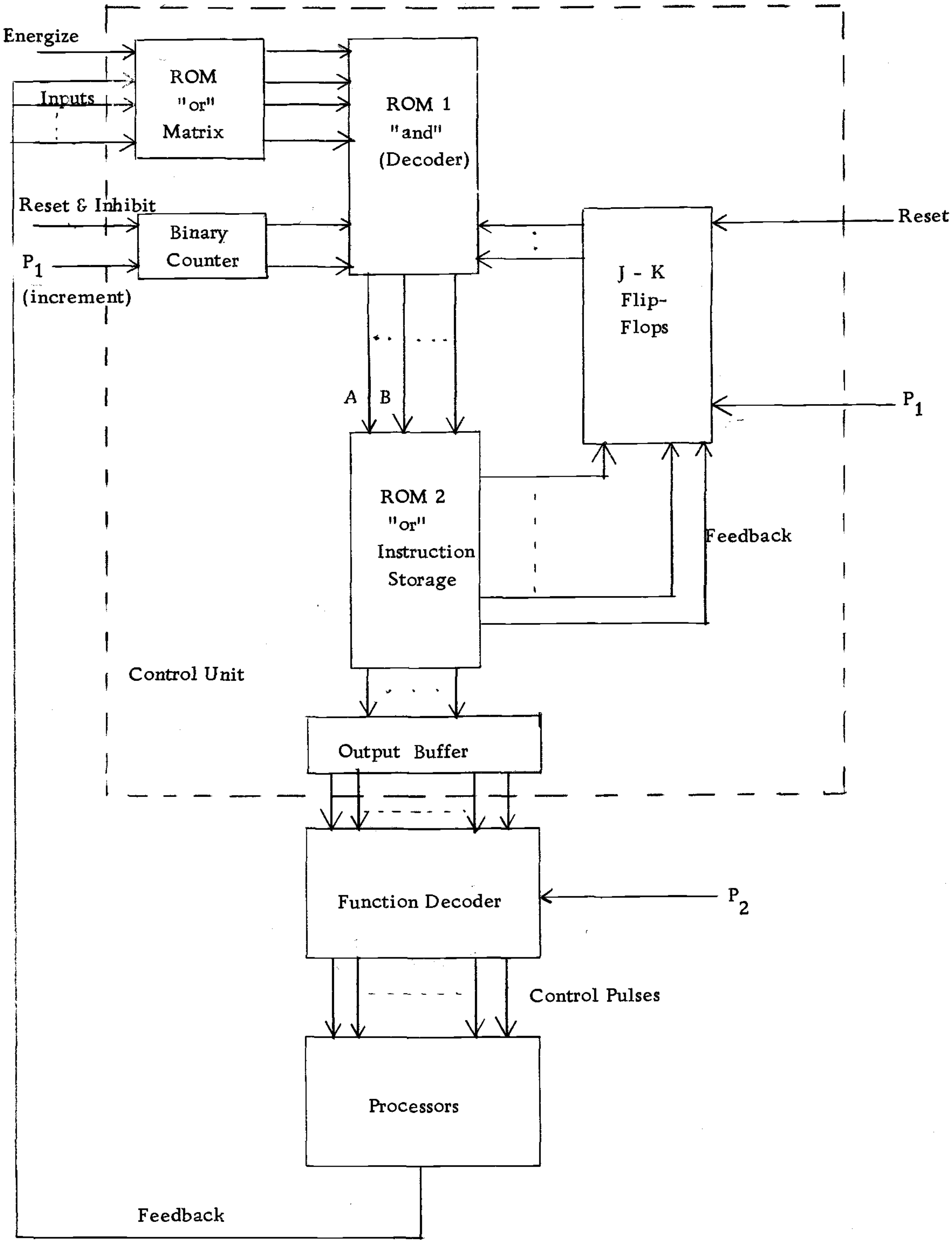


Figure 14. Block diagram of a control unit

As a result, the control unit could remain in this idle condition indefinitely.

When an ENERGIZE signal is received (for the SKR this signal could have come from the ICU) by the ROM input encoder, output line B of the ROM1 will be pulled high. Line B addresses a command in the ROM2 of the form:

	J_3	K_3	J_2	K_2	J_1	K_1
0 0 0 0	0	0	0	0	0	1
microinstruction	feedback					

The feedback input portion of this command will change the J-K flip-flops to state 1, which is the first active state of the control unit. The microinstruction portion of the command still indicates an inhibit condition for the binary counter. When the ROM input encoder receives the TRX signal, line C of the ROM2 will become high this time. Line C addresses a command in the ROM2 of the form:

	J_3	K_3	J_2	K_2	J_1	K_1
ENCC	0	0	0	1	1	0
microinstruction	feedback					

The feedback portion of this command will cause the J-K flip-flops to go to state 2. The microinstruction portion of the command will be decoded by the function decoder into some gating pulse other than the BINARY COUNTER INHIBIT/RESET pulse. Hence, the next clock pulse to the BC will be counted as the first timing level, and its binary information will be fed to ROM1. This information together with the encoded input condition and the state information will be decoded by

ROM1 into another address line, which will then determine the next command to be carried out.

Timing and Operating Speed

From the discussion of the previous section, it can be seen that a control unit in fact operates in two cycles, the fetch and the execution cycle. The fetch cycle marks the change of the contents of the binary counter and of the internal J-K flip-flops. At the end of the fetch cycle a new microinstruction should appear in the output buffer register of the PLA. The execution cycle marks the decoding of the microinstruction and the generation of the required control pulse. Allowing inherent delays in flip-flop operations and the access time of the PLA, the fetch cycle should last about 100 nanoseconds. The same length of time is also given to the execution cycle of the control unit. All microoperations can take place within one complete set of fetch and execution cycle of the control unit except for those regarding memory referencing. Current semiconductor memory has typical cycle time of 600 nanoseconds, which means that a time period equal to 3 sets of fetch and execution cycle time are required to accomplish a memory reference operation.

To provide for this timing scheme, a two phase clock must be used. The pulse width of such a clock is typically 25 nanoseconds, allowing 75 nanoseconds separation time between the two phase pulses.

Figure 15 shows the timing diagram of the two phase clock. Thus, P_1 enables the J-K flip-flops and increments the binary counter of the control unit and P_2 enables the function decoder.

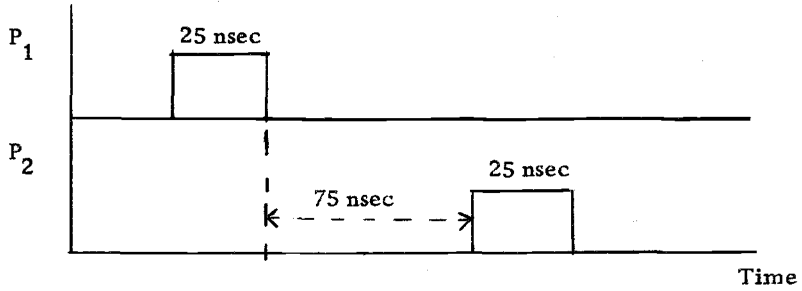


Figure 15. Timing diagram of two phase clock

Using this information on the fetch and execute cycle time of the control unit, a determination of the typical operating time required to input a statement of N characters may be made. From Figure 10, the state diagram of the input control unit, it can be seen that data transmission from the interface and the cycle operation of the control unit proceed in parallel. Since data rate is at best one transmission per 3 μsec and the time required by the control unit to input one character is at worst 1.8 μsec as shown in Table 7, the operating time required to input the statement depends only on the transmission rate. In Table 7, time is calculated by assuming that each microinstruction takes 0.2 μsec, each memory reference 0.6 μsec, and an additional 0.2 μsec is required for state transition. Hence, time required to input the statement can be found by

$$N \times T$$

where N is the number of characters in the statement and T is the period of transmission.

Table 7. Worst case estimate of time needed to input one character by input control unit.

State	Action	Time (μsec)
6	ENCC, ENDC	0.6
7	Write (IBM) In. CIB, In. CN	1.2
Total Time		1.8

Legend:
In: increment

VIII. SCANNER FUNCTIONAL UNIT

General Description

The purpose of the scanner is to identify to what statement type the instruction under examination belongs and hence the correct lexical analyzer be energized. The set of statements allowed by the HC is a subset of FORTRAN IV.

The algorithm used to identify statement types is suggested by John A. Lee (1967). He observes that every Fortran instruction can be classified as either arithmetic or non-arithmetic. Once it is determined that an instruction is non-arithmetic, an identification by the scanner of certain key words will serve uniquely to identify the statement type of an instruction. He also points out that an arithmetic statement has certain features that completely set it apart from non-arithmetic statements.

An arithmetic statement must have an "=" sign as its first special character or else the "=" sign must be the first special character following the right parenthesis. Also, it must not have a comma appearing before a left parenthesis. Therefore, during the first analysis phase the scanner will take the following examples as arithmetic statements.

A = B*C ; (the first special character is an =)

A(1,2) = B*C (the first special character after RP is =)

The following examples violate those rules of an arithmetic statement and hence will not be taken as such.

DO 123 I=1,2,3; (comma precedes left parenthesis)

DIMENSION A(1,2); (first special character after right parenthesis is not =).

Once it is determined that an instruction is non-arithmetic, the first non-statement-number symbol (DO or DIMENSION, in the above examples) will be packed and stored in the shift register (SHR). Symbol packing means that the first five characters of a symbol are extracted and stored in the SHR. For example, in the case of the symbol DIMENSION, the characters D, I, M, E, and N will be extracted and stored in the shift register. Once symbol packing is accomplished, the scanner can identify the specific statement type of the instruction by comparing the contents of the SHR with the set of Fortran command names.

Description of the Operation of the Scanner Control Unit

Figure 16 is the flow chart of the scanner control unit. The first decision the scanner makes is whether or not the first non-blank symbol in statement is a statement number. If it is, the statement

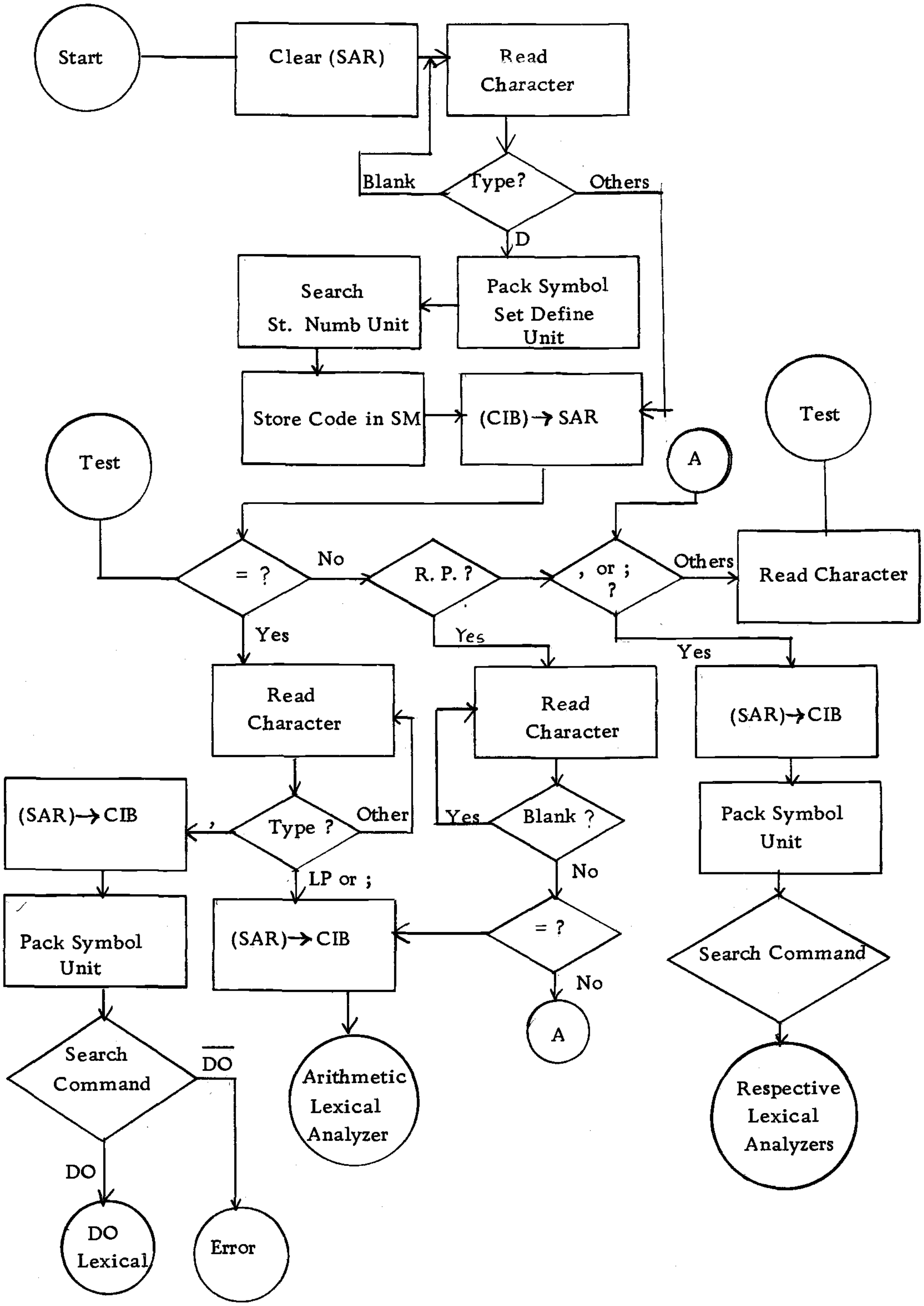


Figure 16. Flow chart of scanner

number is packed into the SHR, the define flag is set to indicate to the statement number search unit of this nature of the statement number, and the search unit is energized. Having taken care of the possible presence of a statement number, the scanner has to determine whether the statement is arithmetic or non-arithmetic.

If it sees a comma or a ";" as the first special character in the instruction, it can already be certain that the statement can neither be a DO nor an arithmetic statement. It can identify the exact nature of the statement by energizing the command-decoder-encoder (see section on hardware of scanner for explanation of command-decoder-encoder).

If the scanner sees that the first special character following the right parenthesis (RP) is an "=", it can be sure that the instruction is meant to be an arithmetic statement. If that character is not an "=", it can again be certain that the instruction is non-arithmetic.

If it sees an "=" sign as its first special character, the possibility is narrowed down to either a DO or an arithmetic statement. It then reads more of the statement. If the first special character it encounters this time is a comma, then it can assume that the statement must be meant to be a DO. The scanner packs the first non-statement-number symbol into the SHR and energizes the command-decoder-encoder to test to that effect. If it is not a DO, an error has occurred in the structuring of that statement.

However, if the first special character it sees after the "=" sign is either a left parenthesis or a ";", it will assume that the statement is in fact an arithmetic assignment statement. It will then energize the arithmetic lexical analyzer. The state diagram of the control unit and the actions associated with each state is shown in Appendix A.

Hardware Associated with the Scanner

The block diagram of the hardware components used by the scanner is shown in Figure 17. Some of these components are used by the ICU and have been previously described. This section describes the other components.

1. Command-decoder-encoder: The identification of command names is by means of a 30 bit to M bit ROM code converter. In actual fact, the code converter is made up of two ROM units. The first is a 30-line-to-N-line decoder (AND matrix). N is the number of Fortran commands allowed by the HC and is left undefined in this paper. The second unit is an N-line-to-16-line encoder. The encoder is necessary because the code that stands for each command name in the S-language (see Table 10) differs from the original internal character code. The choice of the 30-bit-input to the command-decoder-encoder is due to the suggestion of Lee. He observes that in order to save memory space the identification of

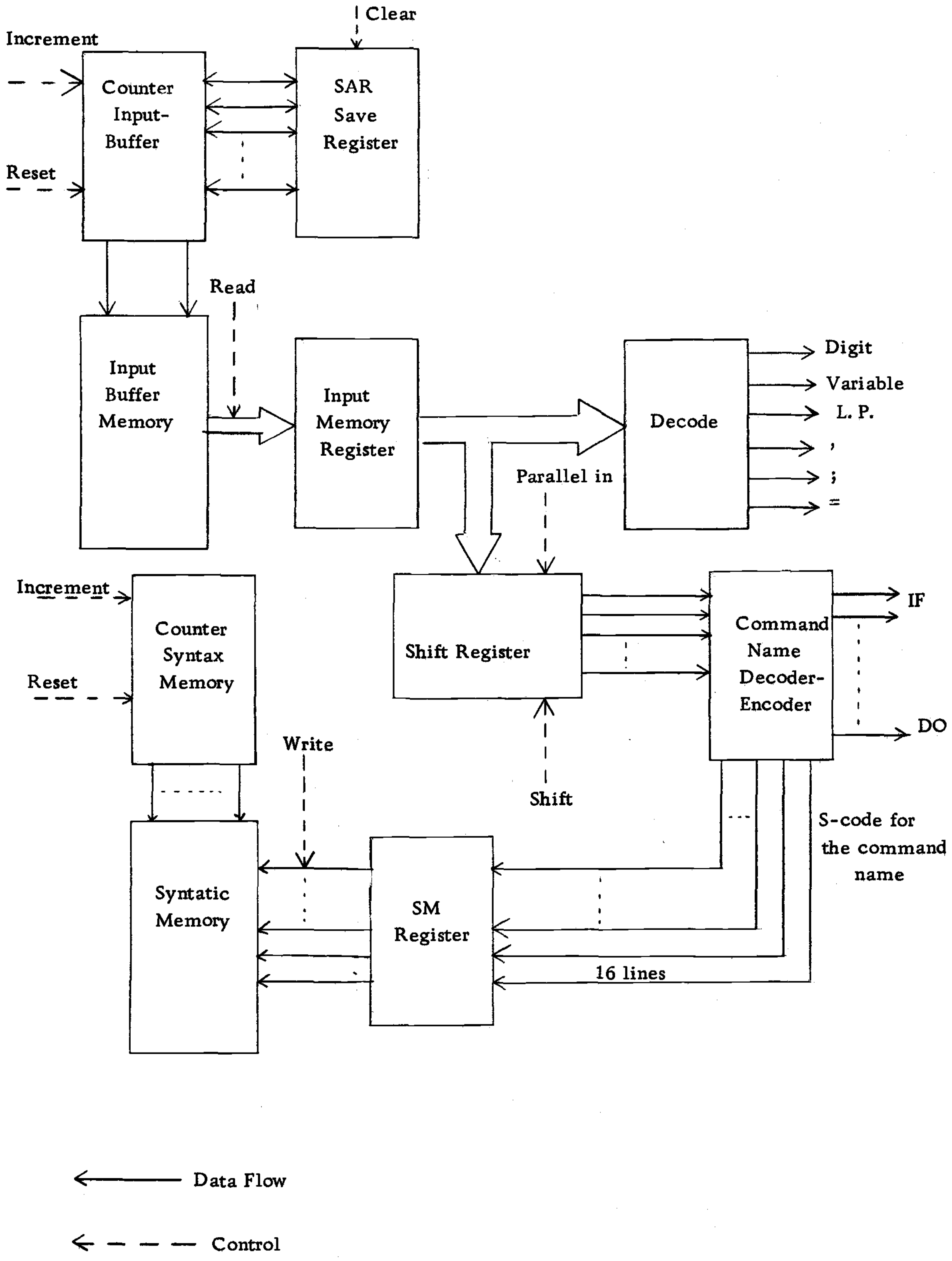


Figure 17. Block diagram hardware of scanner

certain key characters in a Fortran command name is sufficient to determine its exact type. In this paper it is observed that the first 5 characters of a command name is sufficient to differentiate all the command names, hence the choice of 30 bits. In order to find out if a symbol is one of the command names, the symbol (the first five characters of the symbol) is packed into the shift register (SHR), and the command-decoder-encoder is energized. Figure 18 is the block diagram of the command-decoder-encoder.

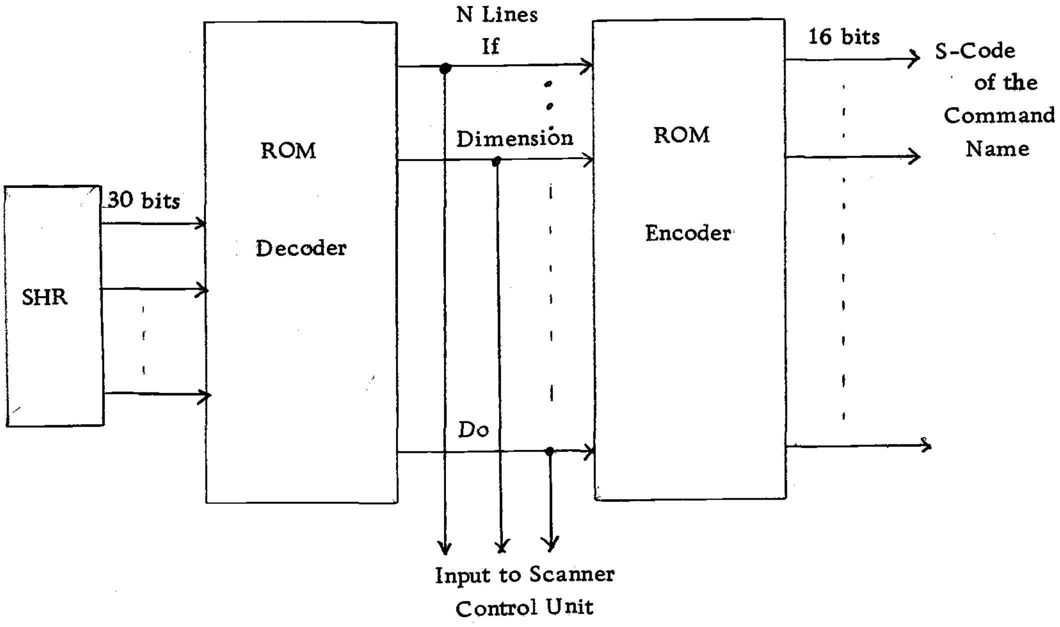


Figure 18. Block diagram of command name decoder-encoder

2. Shift register (SHR): The SHR is a 30 bit parallel in, parallel out, right shift register. It is used to store the characters of a symbol that have been packed. It has its own control logic that allows a 6 bit right shift of its contents when it receives a shift command from one of the control units. The choice of 30 bits for the SHR is due to the fact that the internal code of the HC is 6 bits long and a maximum of 5 characters may be used to form one symbol. Only its first 6 bits are connected externally to other registers for parallel input. All output lines, however, are used to transfer data to other data to other registers. Figure 19 is the block diagram of the SHR.

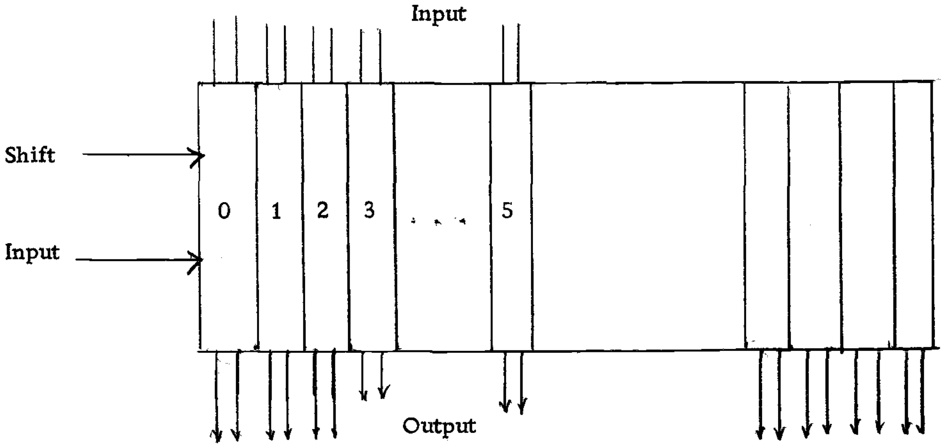


Figure 19. Block diagram of SHR

3. Save register (SAR): The SAR is an 8 bit register, used mainly to save addresses.

Operating Speed of the Scanner

The exact time required for the scanner to determine whether a statement is arithmetic or not depends on the nature of the statement. But using the statement

$$A = B+C*D;$$

as a test example, one can roughly estimate the typical operating time required to scan one statement. Table 8 summarizes, according to the state diagram of the scanner in Appendix A, the actions the scanner must take to scan the test example. Occasionally, the same state is visited several times and the first column of the table gives how many times that state has been visited during the operation of the scanner.

Table 8. Estimate of time required to scan a trial statement.

No. of Visits	State	Operation	(μsec) Time/visit	(μsec) Total Time
1	0	idle	0.2	0.2
1	1	Clear (SAR) RIBM, ICIB ENDC	1.4	1.4
1	5	No Op	0.2	0.2
5	6	RIBM ICIB, ENDC	1.4	7.0
1	10	(SAR)→ CIB ENALAU	0.6	0.6
Operating Time				9.4

Sub-Units Used by the Scanner

Looking at Figure 16, the flow chart of the scanner control unit, it can be seen that there is a number of control subunits which the scanner energizes in order to complete its job. The pack symbol (PSYM) control unit is designed to extract the first five characters of a symbol and store them in the shift register (SHR). Therefore, a symbol can be longer than five characters, however only its first five characters will be used internally to identify the symbol. The rest of the characters are discarded. The PSYM also returns with the first character of the next symbol in the IBMR. Counter N which was mentioned earlier as a hardware component used by the ICU is now used to keep track of the number of characters that have been packed. After the unit is finished, a DONE signal will be issued to the calling control unit. The flow chart of the PSYM control unit is shown in Figure 20. Its state diagram and the actions associated with each state of the control unit are shown in Appendix B.

The other sub-unit used by the scanner is the statement number search unit. Since the concepts of symbol searching and the construction of symbol table are very important, a whole section is devoted to describing it. Appendix B shows that the time required to pack a symbol of n characters is

$$T = n \times 3\ \mu\text{sec}$$

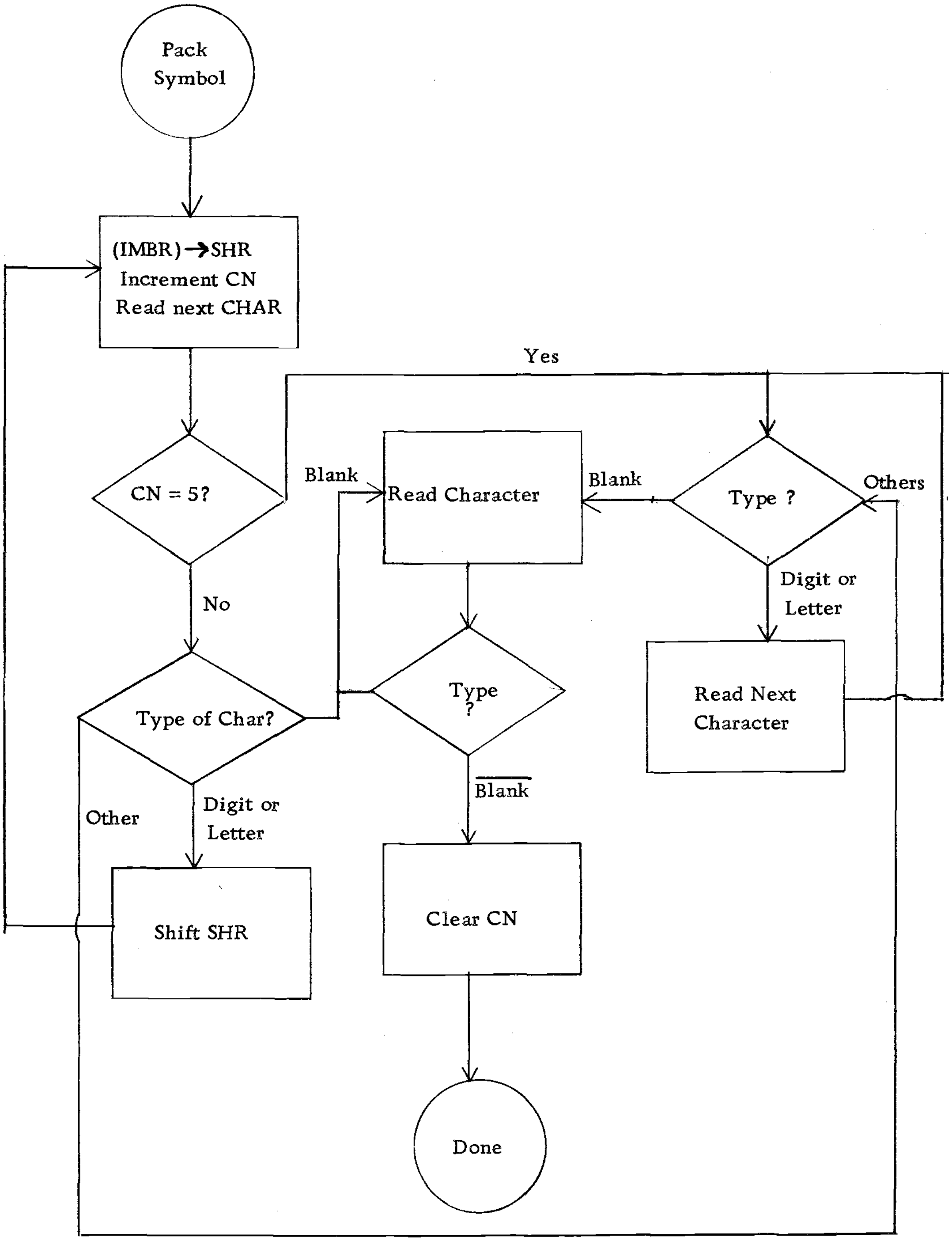


Figure 20. Flow diagram of pack symbol

IX. SYMBOL TABLE

Meaning of Symbol Table

The symbol table of a conventional compiler contains information that is essential to the compilation of each source statement. An item (or word) in the symbol table (SYMTAB) is divided into a key and a value part. The key of an item is the name which the source program must use to address the item itself. The value part contains information on the type of item that is stored in that particular slot of the SYMTAB. This information will classify the item according to the following table (Table 9).

Table 9. Classification of items in the SYMTAB.

1. statement number	defined undefined DO range number
2. variable	simple variable subscripted variable simple formal parameter subscripted formal parameter
3. function name	name of subroutine
4. constant	

With the exception of the constant, the value part of an item in the SYMTAB also contains the actual run time address of the symbol. For example, in the case of a statement number its value part will also contain the actual address of the instruction to which the statement number refers. The following example may clarify these concepts.

In a segment of a program,

```
GO TO  12
..........
.......

12 A = 1.1
```

after a conventional compiler is done with the last program instruction, the memory of a hypothetical computer may look like the following:

	Memory Address	Instruction		
Program Area	00000	UJP 0021		
	00001	....		
	.....	....		
	.....	....		
	00021	LDA 10002		
	00022	STA 10001		
	.....	....		
	07777	....		
		Key	Value	
SYMTAB area	10000	12	D. SN	00021
	10001	A	S. V	10001
	10002	floating point representation of 1. 1		
	.....	....		

Legend : D. SN: defined statement number
S. V. : simple variable

The execution of this program will lead to an unconditional jump to memory address 0021. Then the accumulator will be loaded with the contents of memory address 10002, which is the floating point representation of the number 1.1. And finally the contents of the accumulator will be stored in memory address 10001. From this example, it is clear that the construction of the SYMTAB is an essential part of the compilation process. To create the SYMTAB and to make use of it, conventional compilers use elaborate search routines the execution of which is time-consuming.

SYMTAB of the Hardware Compiler

Since the HC assumes no knowledge of the structure of the host machine, it cannot assign the actual address to a symbol (for example 10001 for A). This has to be left to the software of the host machine. What it can and does do is to set up its own SYMTAB which is a true image of the SYMTAB of the host machine, and replaces each symbol in the program with its identifier and address in this SYMTAB. This will eliminate any symbol search by the host machine software. For example, in the following program segment:

```
      A = B-C
12    D = 1.1
```

The HC's SYMTAB hypothetically may look like Figure 21.

SYMTAB ADDRESS	Key	Value
......	....	....
005	A	S. V. 005
......	....	....
010	B	S. V. 010
......	....	....
015	C	S. V. 015
......	....	....
021	D	S. V. 021
022	1. 1	Cons. 022
023	12	D. St. 023
......	....	....
......	....	....

Figure 21. Hypothetical SYMTAB of HC

As can be seen in Figure 21, the value part of an item stored in HC's symbol table is exactly the same as that of the host machine's SYMTAB. The only exception is that in HC's SYMTAB, constants are not converted into their binary representation before they are stored in it. The reason for this is given in a later section.

The specific code for the value part of an item stored in the SYMTAB is shown in Table 11 (which is the word code table of the S-language). Basically, the S-code is a 16 bit code, the first six bits identify what type of statement symbol it is, and the last ten bits contain the address of the symbol in the symbol table.

Hardware Implementation of SYMTAB

The symbol table (SYMTAB) of the hardware compiler is a search memory. With the improvement of circuit technology, there is a growing interest in the development of hardware search memories. Presently there are three popular ways by which search memory may be built:

1. content addressable memory using LSI technology.
2. recirculating memory built by dynamic shift registers.
3. content addressable memory built with conventional random access memory (RAM) with read only store (ROS) control and a hash coding network.

In method 1, the whole memory is fabricated on a single chip (currently a 12 word, 8 bit-per-word, chip can be found in the market). The chip contains built-in circuitry that performs any of the functions of read, write, or interrogate in one cycle time (2 μsec). However, at this stage of the technology, such memory chip is still expensive, running close to $1.5 per storage bit. It is not practical to build a large search memory with these chips.

In method 2, the search scheme is depicted in Figure 22. It is composed of M N-bit dynamic recirculating shift registers operating in parallel to store information. N defines the number of words in the memory and M the number of bits per word. An item to be searched

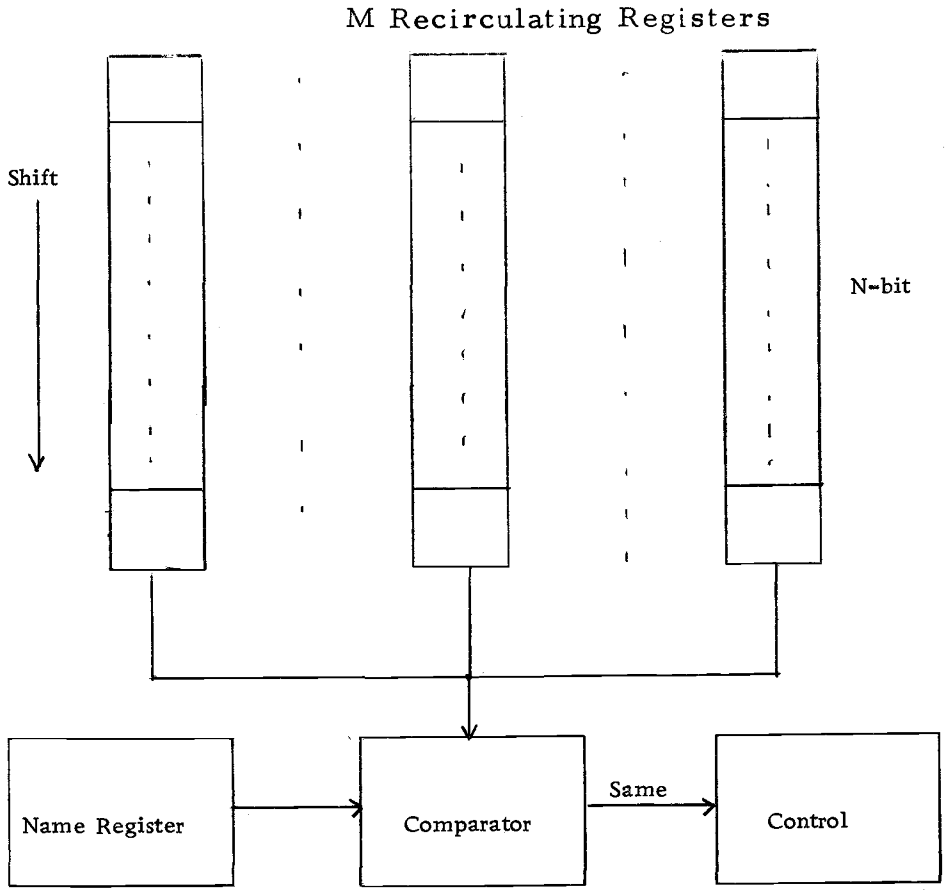


Figure 22. Block diagram of a search memory using dynamic recirculating registers

in the memory is first stored in the name register (NR) and is then compared with the base of the recirculating memory. If they are the same the item has been located. If not, the shift registers will be shifted end round (recirculated) and again comparisons are made between the base of the memory and the contents of the NR. This is continued until either a match occurs or the whole set of memory contents have been searched. Dynamic shift registers can operate at a very high shifting frequency (4 Mhz). Such a search memory can be quite fast. However, a large search memory is still hard to be implemented by this method.

Within the state of the art, method 3 proves to be a very economical and efficient way of constructing a search memory. It has merits, that the above two methods lack, that of providing a large search memory (40 K bits or more) with reasonable cost. In the design of the HC, it is felt that cost is the over-riding design constraint, owing to the fact that even if HC is slow, it still serves to free the central processing unit (CPU) of the main computer and is still a good product.

This scheme of constructing a search memory was introduced by King (1971) and is shown in Figure 23. This search algorithm involves the transformation of the key of an item by some calculation (hash function) into an address. The contents of the addressed memory cell is fetched and compared with the original key. If they are equal, the item has been located. If they are not the same a collision has occurred. There are several ways for resolving collisions, random probing, direct chaining, and linear probing. Both random probing and direct chaining are more efficient than linear probing at the cost of more complicated search algorithms. Morris (1968) estimates that for a load factor of a,

$a = \frac{k}{N}$ where k is the number of items in the table and N is the size of the table, the average number E of probes for linear probing necessary to look up an item in the table is

$$E = (1 - \frac{a}{2})/(1 - a)$$

as compared with

$$E = (\frac{1}{a}) \log (1 - a)$$

for random probing, and

$$E = 1 + \frac{a}{2} \text{ for direct chaining.}$$

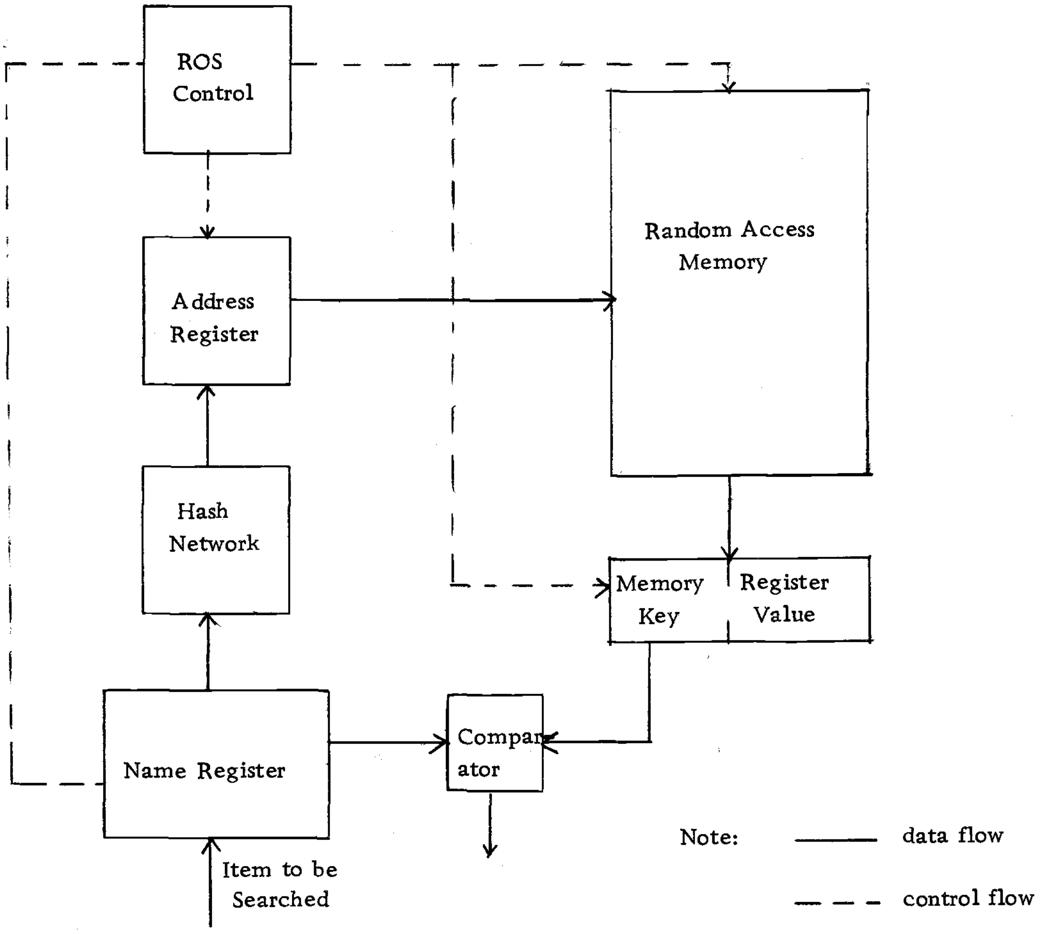


Figure 23. Block diagram of a search memory using hash network

In this paper, it is felt that linear probing, even though less efficient, offers the best features for hardware implementation because of its simplicity. And it can be seen that all three methods has comparable values for E if the load factor a is less than .5 .

The maximum number of symbols present in a program can be assumed to be no greater than 500. If a RAM of 1K word capacity is used, the load factor can be assured of no greater than 0.5. Assuming this range of operations, linear probing is no less efficient than the other methods. Presently memory modules of about 10 cents per bit can be purchased, which means that a search memory of about 40K bits may cost about

$$\$46K \times .1 = \$4.6K.$$

The use of linear probing and hash coding scheme to implement a search memory as proposed by King (1971) is shown in Figure 23. Figure 23 shows that the search memory is controlled by a control unit implementable with the same techniques already discussed. The name register (NR) is used to hold the name of the item to be read or stored just as the address register is used in a conventional memory.

King did not specify how the hash network may be constructed. However, it is observed here that since the hash network is in fact a code converter, ROM customs converters may be used to do that job. The advantage of using such a code converter is speed. Conventional ROM can operate at a speed of 60 n sec. Many well-known hashing methods can be implemented thus (Morris, 1968). No details will be given here for the exact logic design. Suffice it to say that detailed design should be taken at a later stage.

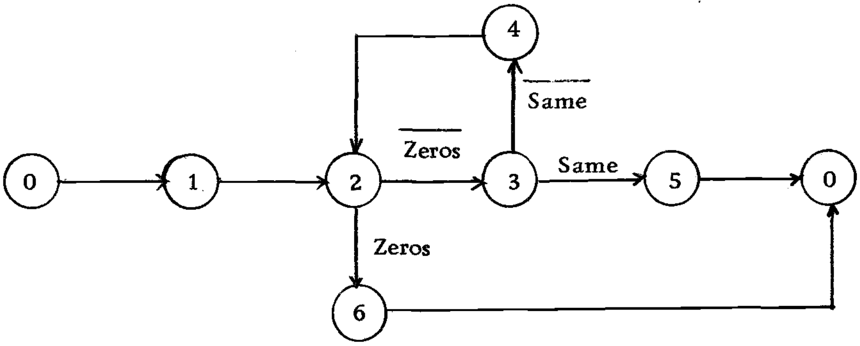
The sequence of operations to search one item is

1. Transfer the item to the name register.
2. Hash item into its address code and enter it into Address Register (AR).
3. The word addressed by AR is fetched and stored in the Memory Register (MR).
4. If MR contains all zeros, it means that an empty memory slot exists. Item is not in the memory. If the contents of NR and MR are the same, then the item has been found. The job is done.
5. If they are not the same, increment AR by one and go to step 4.

In Figure 23, the RAM which serves as the storage of the SYMTAB is organized as a 1K, 46 bit per word memory. The first 30 bits of a word form the key and the last 16 bits the value of the item. Semiconductor memory RAM is used because:

1. its access time is fast.
2. the HC SYMTAB is a scratch pad.
3. it is cheap, 10 cents per bit.

The state diagram of the search memory is shown in Figure 24.



State	Action
0	idle
1	(SHR)→NR, Energize Hash Network
2	Fetch, Decode (MR)
3	Compare (MR) with (NR)
4	↑ AR
5	Found
6	Not Found

Figure 24. State diagram of search memory

The response time of such a search memory assuming a nominal load factor of 0.5 can be estimated. The expected number of probes required to get the right result from the search memory is

$$E = (1 - \frac{.5}{2}) / (1 \quad 0.5)$$

$$= 1.5$$

For worst case consideration, let E be 2. Figure 24 shows that the number of microinstructions required to complete one probe is at worst 6, plus the fetch microinstruction which takes 0.6 μsec. Response time of the search memory is approximately

$$2 \times (1.2 + .6)\ \mu\text{sec} = 3.6\ \mu\text{sec}.$$

Statement Number Search Unit

The statement number search unit is energized whenever a statement number is encountered in an instruction, so that its S-equivalent may be found. There are four different types of statement numbers each requiring a different set of treatments by the search unit. Hence, before energizing the search unit, an appropriate flip-flop has to be set to indicate whether the statement number is being defined, not being defined, appearing as an undefined DO range, or as a defined DO range. A statement number is being defined if it is used as in

```
14   A = B ;
```

In the above instruction 14 is a defined statement number. Before energizing the statement number search unit (SNSU), the DF flip-flop has to be set.

A statement number is not being defined if it is used as in

```
GO TO 14 ;
```

14 is not being defined by the above statement, hence the DF flip-flop must be reset.

A statement number is a DO range when it appears in a DO statement, as in

```
DO 14 I = 1, 4 ;
```

In this case, the DO flip-flop should be set before the SNSU is energized. If later, statement number 14 is used in the label field in another statement, as in 14 A = C;

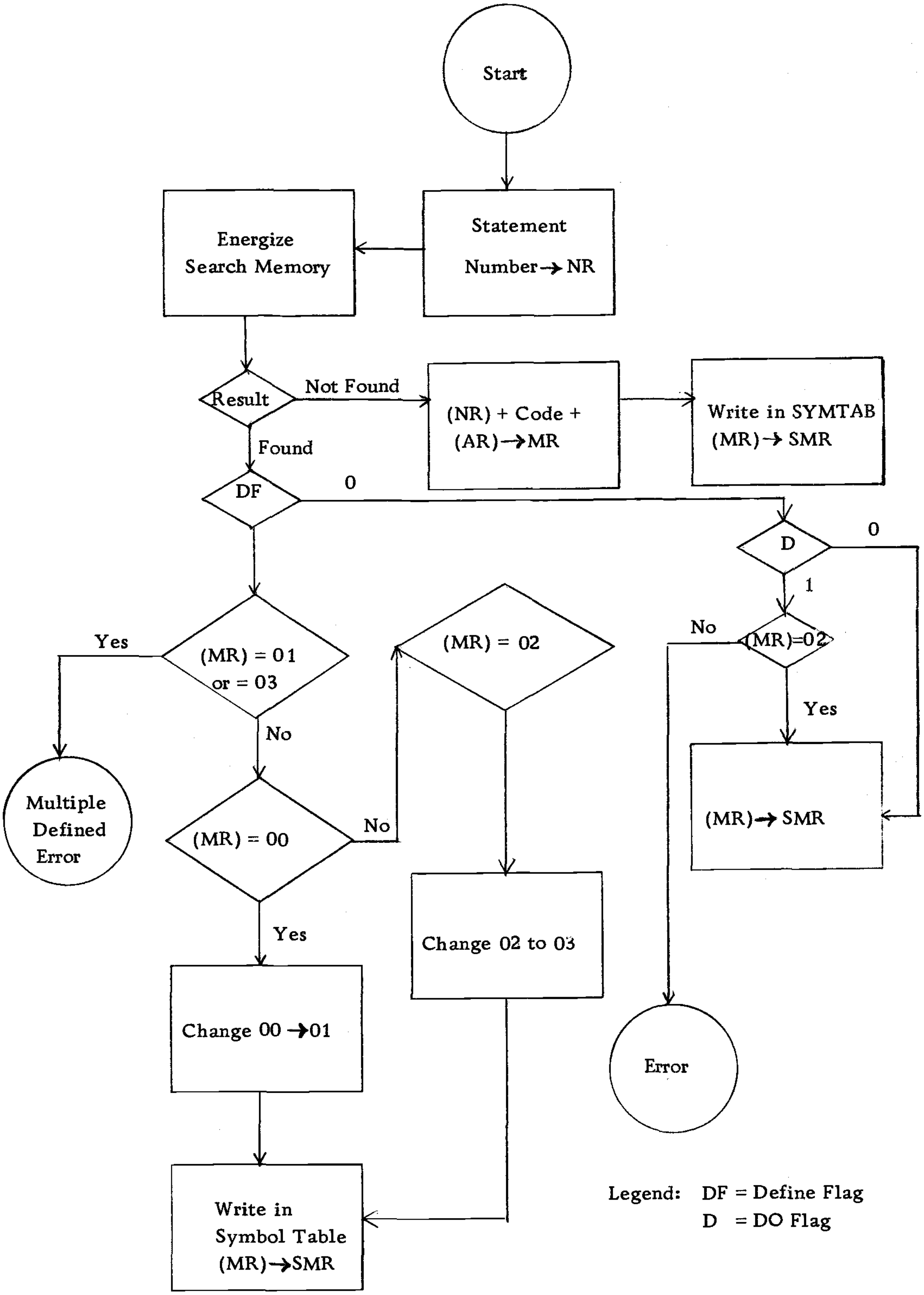
the statement number 14 becomes a defined DO range.

The statement number search unit performs its job according to the following sequence:

1. Transfer the statement number to the name register (NR) of the search memory.
2. Energize the control of the search memory.
3. If the item is found,
    a. If the defined (DF) flip-flop is on,
        i. if the value part of the word in MR, i. e. (MR), indicates that the statement number has already been defined (value is 01 XXX or 03 XXX) a multiple defined error has occurred. Go to an error routine.
        ii. if the value part of (MR) indicates an undefined statement number, the value of (MR) is changed to "defined" (from 00 to 01). The new (MR) is stored back in the symbol table and also transferred to the syntax memory register (SMR).
        iii. if the value is a DO range (02), the contents of MR is changed to a defined DO range (03). The new (MR) is stored back in the symbol table and is also transferred to the SMR.

   b. if the DO flip-flop is on,
      i. if the value part of the (MR) is 02 _ _ _ _ indicating a DO range, (MR) can be transferred to the SMR.
      ii. otherwise, an error has occurred, energize an error unit.
   c. if neither DO nor DF flip-flop is on, the contents of MR, i. e. (MR), are simply transferred to the SMR.
4. If the statement number is not found, one of the sets of code 00, 01, 02, or 03 is transferred to the first six bits of the value part of the MR. The contents of the address register AR are transferred to the last 10 bits of the AR, and (NR) are transferred to the key of memory register. The new contents of the MR are written in the symbol table, and transferred to the SMR.

The flow chart description of the search unit's operations is shown in Figure 25 and Figure 26 illustrates the data flow of the statement number search process. The state diagram of the search unit is shown in Appendix C.



Legend: DF = Define Flag
D = DO Flag

Figure 25. Flow chart of statement number search unit

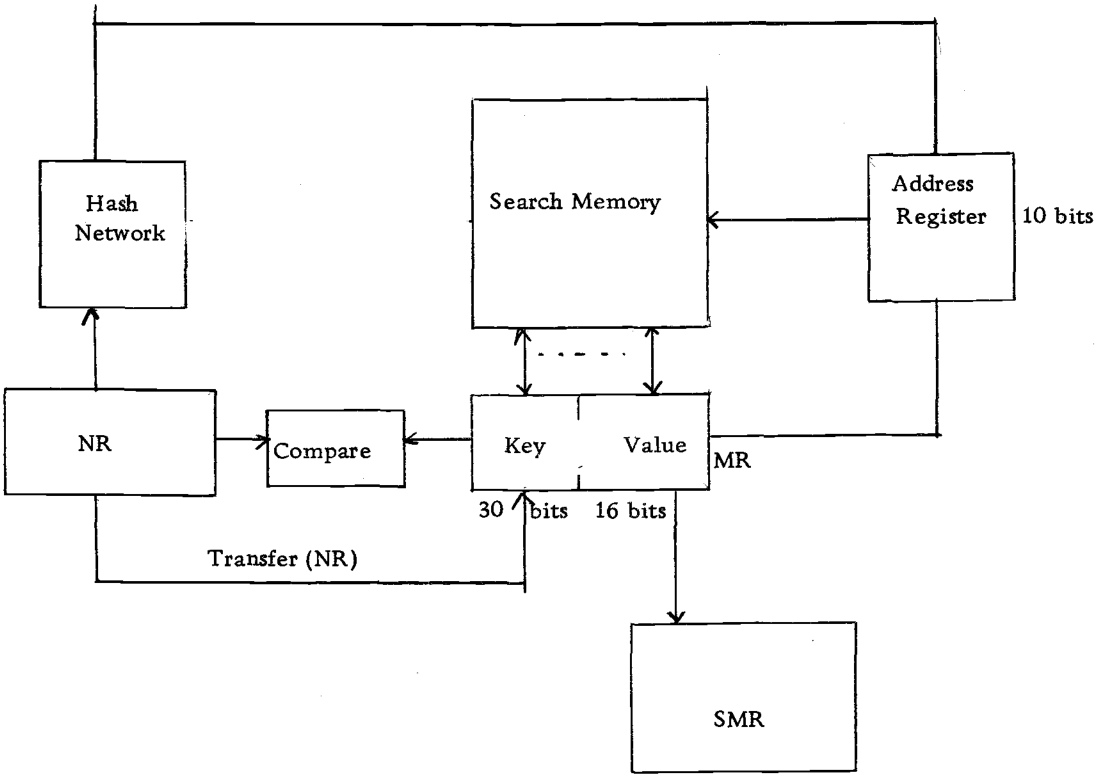


Figure 26. Data flow of statement number search

Time Needed to Search Statement Number

The time required by the statement number search unit to work on one statement number depends on how long the number is and what kind of statement number it is. Table 10 shows the typical value for this time. As can be seen, the execution of a statement number search typically takes 6 μsec.

Table 10. Estimate of time required for a typical statement number search.

State	Action	Time (μsec)
0	Idle	0.2
1	Energize Search Memory	3.8
2	Test DF	0.4
3	Test Value (MR)	0.4
5	change (MR) to 01 write SYMTAB (MR)→SMR	1.2
Total Time		6.0

X. LEXICAL ANALYZER UNITS

Background Discussion

The number of lexical analyzer units is left undefined. However, one statement that can be made is that more lexical analyzer units (LAU's) can be added modularly, depending on how many Fortran statement types are desired to be processed. In this paper only a few LAU's are described.

According to R. Hopgood (1969), whose compiling technique is adopted in this paper,

"The aim of the lexical analysis phase of the compiler is to take the input program, which is presented to the compiler in some arbitrary form, and translate this into a string of characters which will be called the S-string. This is the input to the syntax analyser."

This S-string is a grammatically correct sentence which can be analyzed by the parser provided in the syntax analysis phase. The characteristics of the S-string is summarized as follows:

1. statement numbers and variables will be replaced by their equivalents, which are their special identifiers and their addresses in the SYMTAB.
2. constants will be unchanged, but will be preceded by their addresses in the SYMTAB.

3. special statement identifiers will be used to precede any non-arithmetic statements.

Table 11 shows the code in the S-language and its individual meaning.

Table 11. S-code and its meaning.

Type of item	6 bits Code	10 bits
undefined statement number	00	Address in SYMTAB
defined statement number	01	Address in SYMTAB
DO range	02	Address in SYMTAB
simple variable	10	Address in SYMTAB
subscripted variable	11	Address in SYMTAB
simple formal parameter	12	Address in SYMTAB
subscripted formal param	13	Address in SYMTAB
subroutine name	20	Address in SYMTAB
constant	30	Address in SYMTAB
operator	4 _ _	_ _ _ _ _
identifier of command	5X	code for the command
defined DO range	03	Address in SYMTAB

The S-code generated by the LAU's are stored in the SYNTAX memory (S memory). The S-memory has a buffer register (SMR) and an address counter (CSM).

Arithmetic Lexical Analyzer (ALAU)

The arithmetic lexical analyzer unit is energized to produce the S-string equivalent of an arithmetic statement. In the S-code every variable in an arithmetic statement will be replaced by the following word,

6 bits	10 bits	
0 0 1 0 0 0	Address in SYMTAB	16 bit word

for a simple variable, or

6 bits	10 bits
0 0 1 0 1 0	Address in SYMTAB

for a subscripted variable.

Constants will remain unchanged but will be preceded by the following special code,

6 bits	10 bits
0 1 1 0 0 0	Address in SYMTAB

The ALAU is usually energized by the scanner, but may also be energized by the IF lexical analyzer. In the latter case, the ALAU considers its job done when it comes to a right parenthesis. The reason is that in an IF statement

IF (arithmetic statement) $n_1, n_2, n_3;$

the right parenthesis is the delimiter for the arithmetic expression. The IF flag should be raised if the IF lexical analyzer energizes the ALAU. Treatment of most characters that may appear in an arithmetic statement is straightforward, however extra care has to be taken with regards to treating exponentiation and relation operators.

Exponentiation in Fortran is denoted by **, so when a * is encountered by ALAU it must check the next character to see if it is also a *. If it is the exponentiation identifier 57_8 should be stored in the S-memory. If not, multiplication identifier can be stored.

Relation operators are GT, LS, EQ, and others. They must be delimited on both sides by periods. For example,

IF (A. GT. B) GO TO 2 ;

Hence, in the program when a period is encountered by the ALAU, the succeeding symbol is packed and the command-encoder-decoder has to be energized to check for the persence of relation operators. If the symbol is not one of the relation operators, it has to be a digit. Otherwise, an error has occurred.

When the ALAU uncounters the end of statement sign ";" its job is completed and it will issue an ENERGIZE signal to the syntax analyzer unit. If it encounters a right parenthesis and the IF flag is on, its job is also completed, and a DONE signal will be issued to the IF lexical energizer unit. The flow chart of the ALAU control is shown in Figure 27.

Time Required to Analyze an Arithmetic Statement

To give a rough picture on how much time is needed by the ALAU to analyze one statement, the arithmetic statement

$$A = B + C * D;$$

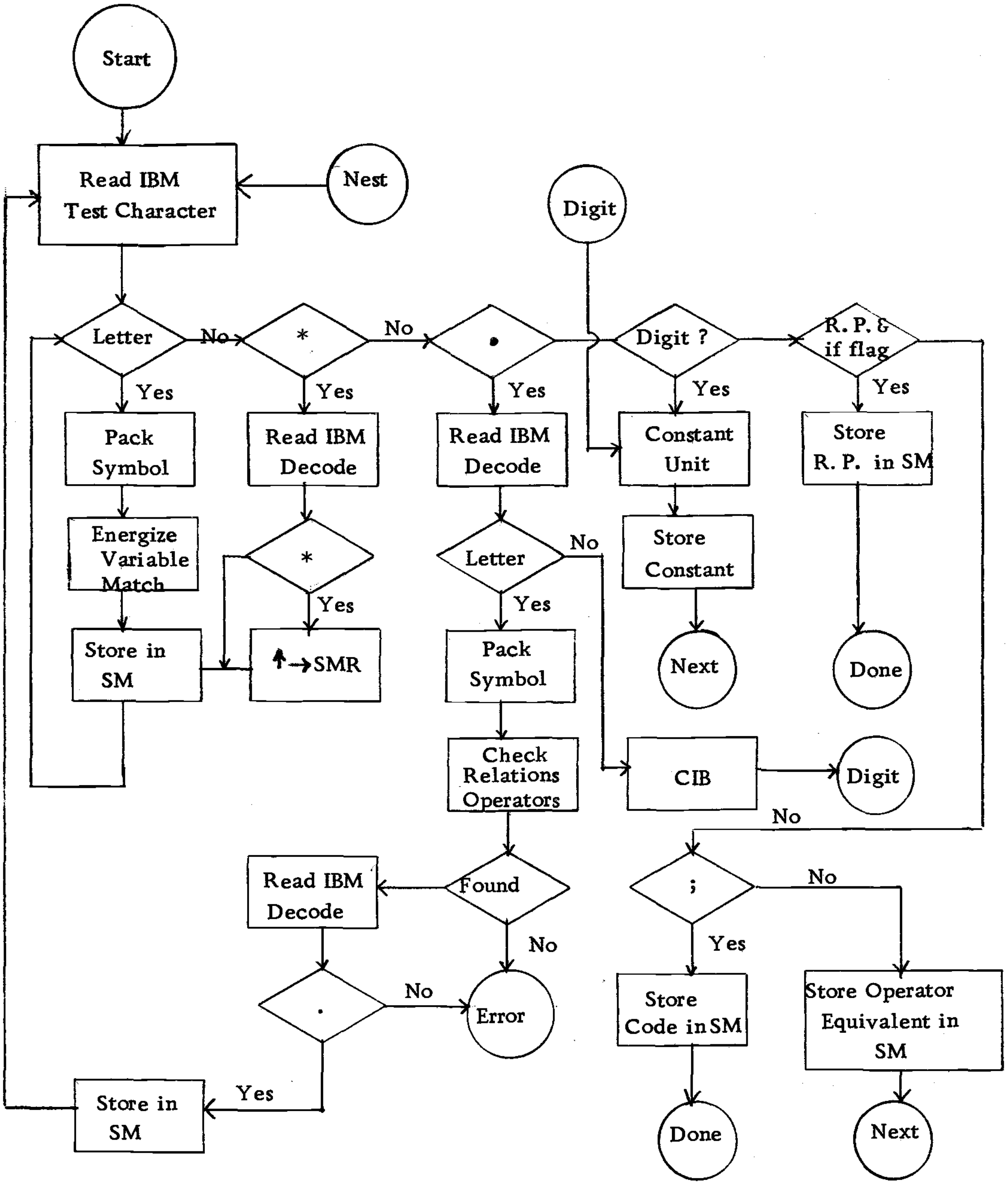


Figure 27. Flow chart of ALAU

is again used as a text example. Table 12, derived from the state diagram of the ALAU shown in Appendix D summarizes the actions of the ALAU as it analyzes the test example.

Table 12. Estimate of time required to analyze one arithmetic statement by ALAU.

No. of Visits	State	Action	Time/visit (μsec)	Total Time (μsec)
1	0	idle	0.2	0.2
8	1	RIBM, ICIB, ENDC	1.2	9.6
4	2	Pack Symbol	1	4.0
4	3	Variable Search	3.8	15.2
4	4	Write SM, ICSM,	1.0	4.0
3	11	Operator code to SM Write SM, ICSM	1.2	3.6
1	13	Code to SM, RCSM	1	1.0
Total Operating Time				37.6

According to Table 12 the execution time of the ALAU is roughly 37.6 μsec.

Variable Match Unit

The purpose of the variable match unit (VMU) is to find the S-equivalent for a given variable. There are two types of variables to be distinguished, subscripted and simple variables. If a variable is known to be subscripted, before the VMU is energized the subscripted variable flip-flop has to be set first. The VMU then performs its job by:

1. transferring the variable name from the shift register to the name register of the search memory,
2. energizing the search memory,
3. if the variable is found then transfer the value part of the MR to the SMR. The process is finished.
4. if the variable is not found ,
    a. transfer (NR) to $(MR)_{0-29}$,
        i. if it is a simple variable, transfer (10) to $(MR)_{30-35}$,
        ii. transfer (11) to $(MR)_{30-35}$, if it is a subscripted variable.
    b. transfer contents of AR to $(MR)_{36-45}$,
    c. write (MR) into symbol table and transfer (MR) to SMR.

The state diagram of the VMU can be similarly constructed as the one for the statement number search unit. It is shown in Appendix D.

Time Required by VMU to Match One Variable

Assuming the VMU has to handle a variable which has so far not been used in the program, in other words the VMU will not find it in the SYMTAB. Table 13 then summarizes the actions the VMU will take to treat a variable of this kind and the time required.

Table 13. Estimate of time required to match one variable.

State	Action	Time (μsec)
0	Idle	0.2
1	Energize Search Memory	3.8
3	$(NR) \rightarrow (MR)_{0-29}$, Variable Code $\rightarrow (MR)_{30-35}$ Write (MR), $(MR) \rightarrow SMR$	1.4
Total Time		5.4

Table 13 is derived from the state diagram shown in Appendix D and it shows that the time required by the VMU to treat one variable is typically 5.4 μsec.

Constant Unit

A constant in the S-string is preceded by the S-word

30	address of constant in SYMTAB

The constant unit performs its job by the following algorithm:

1. Pack the first six digits of the constant into the shift register.
2. transfer the contents of the SHR to the NR,
3. energize the search memory,
4. if the word is found,
    a. increment AR by one and fetch a new word from the search memory,

b. if the contents of MR are not all zeros, go back to 4. a.

c. if the contents of the MR are all zeros,

   i. transfer (NR) to $(MR)_{0-29}$,

   ii. transfer 30_8 to $(MR)_{30-35}$.

   iii. transfer (AR) to $(MR)_{36-45}$,

   iv. write (MR) into the symbol table and transfer (MR) to SMR.

5. If the word is not found, transfer (NR) to $(MR)_{0-29}$, transfer 30_8 to $(MR)_{30-35}$, transfer (AR) to $(MR)_{36-45}$, write (MR) into the symbol table, and transfer (MR) to SMR.

The state diagram that describes these actions of the constant unit is shown in Appendix E.

The exact operating time required depends on the nature of the constants to be treated. However, it is safe to assume that the typical operating time required by the constant unit is about the same as the variable match unit which takes about 5 μsec.

DO Lexical Analyzer Unit

The DO lexical analyzer unit (DLAU) is energized to convert a DO statement of the form

DO (statement number) $I = n_1, n_2, n_3$;

into its S-string representation. The S-string of a DO statement is of the form

DO	02SYMTAB address	10SYMTAB address	=	30SYMTAB address	n_1	CNDL	,
30 SYMTAB address	n_2	CNDL	,	30SYMTAB address	n_3	CNDL	;

Preceding the DO string is the DO token which identifies that the statement is a DO. There is no error check included in the design of the DLAU. This job is relegated to the syntax analysis phase of the HC. The flow chart of the DLAU is shown in Figure 28 and its state diagram in Appendix G.

Table 14 summarizes the sequence of states that the DLAU must take to analyze a statement of the form

DO 14 I = 1, 4, 1;

It then estimates the operating time needed for the DLAU to analyze the above statement. It shows that the DLAU takes about 34.2 μsec to complete its job.

Table 14. Estimate of operating of DO lexical analyzer.

No. of visits	State	Time/visit (μsec)	Total Time (μsec)
1	0	0.2	0.2
1	1	1.4	1.4
1	2	1.0	1.0
1	3	3.8	6.0
8	4	1.2	4.8
3	5	1.0	3.0
1	6	5.4	5.4
1	7	2.0	2.0
3	8	5.4	16.2
1	9	1.4	1.4
Total Time			36.4 μsec

IF Lexical Analyzer Unit

The IF lexical analyzer unit (ILAU) is energized to produce the S-string of an IF statement. It first stores the IF token in the syntax memory. It then sets the IF flag and energizes the ALAU. After it receives the DONE signal from the ALAU, it goes on to convert the rest of the IF statement according to the following:

1. If the first non-blank is a letter, it energizes the scanner to determine what statement type is following the right parenthesis. For example, IF (A .GT. C) GO TO 3;
2. If the first non-blank symbol is a numeral the ILAU will treat it as a statement number. Henceforth, only numerals or commas will be acceptable. When the ILAU sees a ";", its job is done and it will energize the syntax analyzer. The flow chart of the ILAU is shown in Figure 27 and its state diagram is shown in Appendix H.

The flow chart of the ILAU control is shown in Figure 29.

To illustrate what the typical operating time can be expected from the ILAU, the following test example is used.

IF (A-B) 1 2, 3;

Table 15 is constructed from Appendix H. It shows all the states of the ILAU associated with analyzing the particular IF statement, and the approximate operating time as a result of cycling through these states.

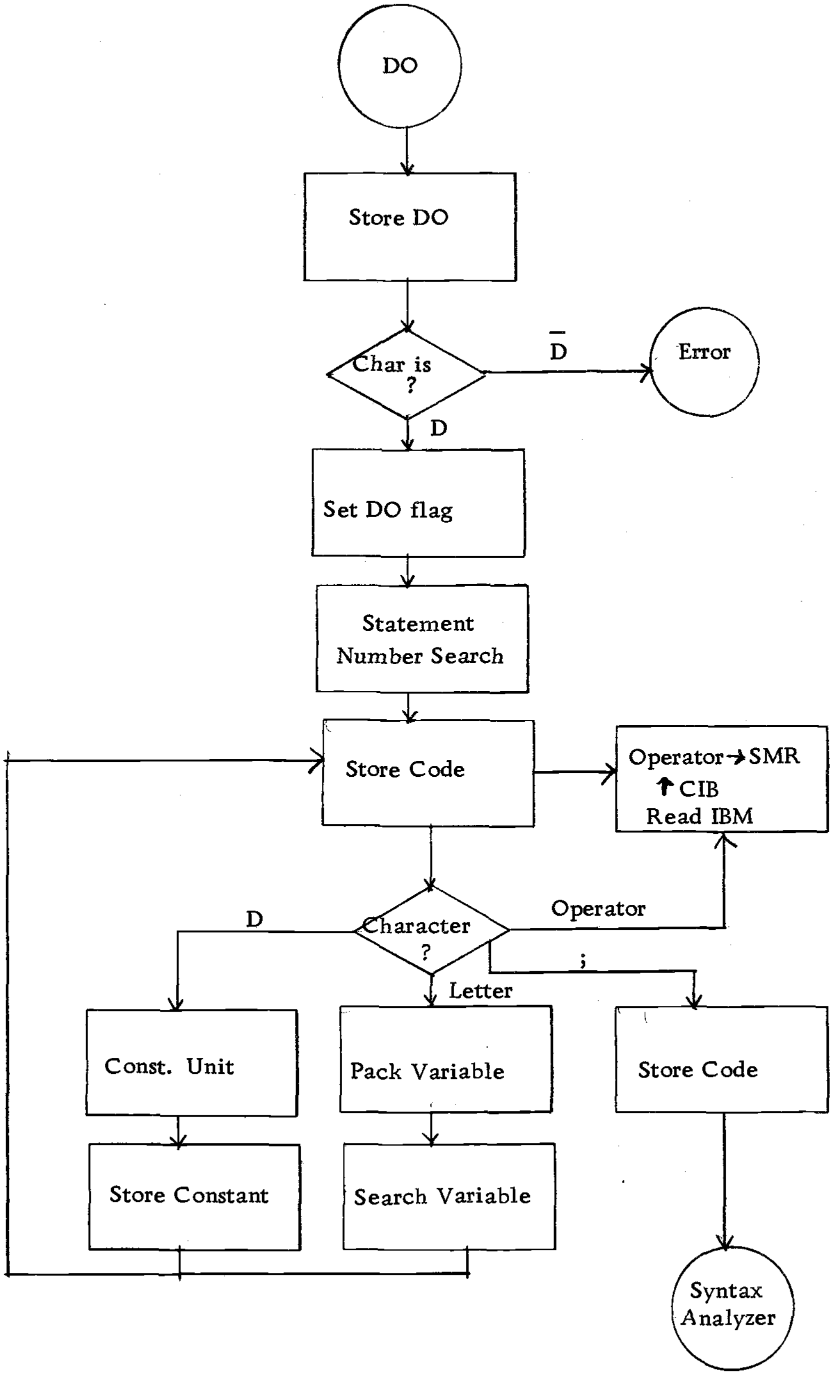


Figure 28. Flow chart of DO LAU

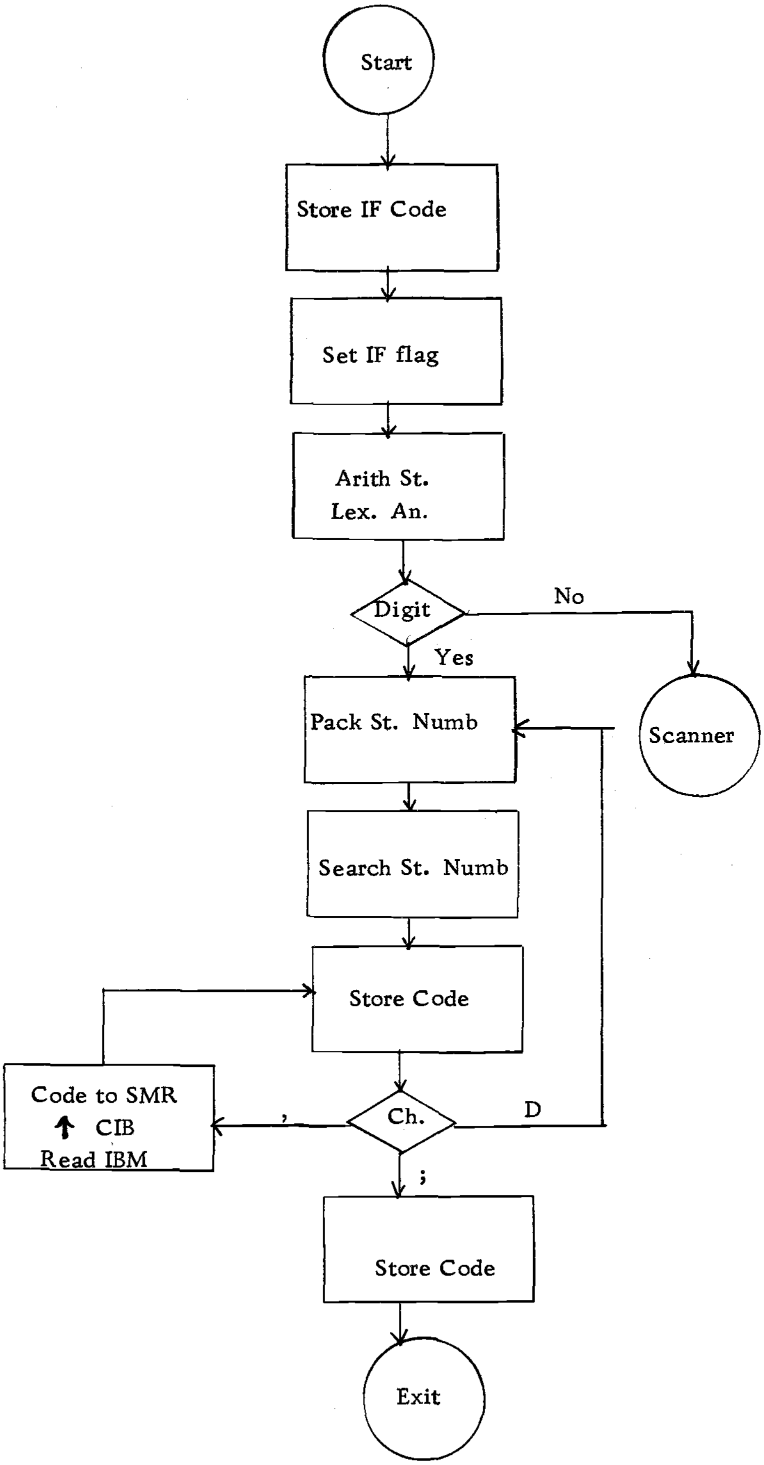


Figure 29. Flow chart of IF LAU

Table 15. Estimate of the operating time of the ILAU.

State	No. of Visits	Actions	Time/visit	Total Time μsec
0	1	idle	0.2	0.2
1	1	IF→SMR Write SM, ISMC Set IF flag Energize ALAU	16.4	16.4
2	6	ICIB, Read IBM ENDC	1.2	2.4
3	3	Pack Symbol	1.0	3.0
4	3	Energize Statement Number Search Unit	6.0	18.0
5	5	Write SM	0.8	4.0
6	2	, →SMR	0.4	0.8
8	1	; →SMR, Write SM ISMC, done	1.4	1.4
Total Time of ILAU				46.2

Table 15 shows that it takes about 46.2 μsec, for the ILAU to analyze the IF statement.

XI. SYNTAX ANALYZER UNIT

The purpose of the syntax analyzer unit (SAU) is to take the S-string produced by the various LAU's, and to use some parsing algorithm to verify that the string is in fact a legally structured string in the S-language. In addition, it will be required to output an intermediate machine code which will allow easy actual machine code generation. It is this code that will be transferred to the main computer for the final stage of compilation.

There are many well known parsing algorithms in existence and more can be expected forthcoming. To name a few, there are, the Floyd production method, the top-down analysis, the bottom-up analysis, the operator precedence analysis, and the analysis by a switching matrix. All these methods have their relative merits, however from the standpoint of LSI implementation, parsing by means of a switching matrix contains inherent logic blocks that make it subject to easy hardware implementation.

Since the method is well known and documented, no attempts will be made here to show its derivation. Only a general explanation of how the method works, including its flow chart description and its hardware implementation, will be given.

Arithmetic Statement Switching Matrix

To lead eventually to the explanation of a complete switching matrix, it is felt that a switching matrix to handle only arithmetic statements should first be explained.

The interpretation of an arithmetic statement is subject to a set of precedence rules and conventions. Hence, the need to generate a parenthesis and precedence free arithmetic string is imperative.

For an expression, such as

$$A = B + C * D ;$$

it is known by precedence rules that multiplication should be performed first, then addition, and finally equating. The code generated by the syntax analyzer unit allows just that. Figure 30 shows the resulting code that is stored in the code memory.

	CM_i	CM_{i+1}	CM_{i+2}	CM_{i+3}	Code memory address
i=0	*	C	D	T_1	
i=4	+	B	T_1	T_1	
i=8	=	A	T_1	T_1	
i=12	;				

Figure 30. Code for an arithmetic statement

The code means simply to perform the arithmetic operation found in CM_i on the next two words and to store the result in a temporary address T_m. Another example is given below, this time involving subscripted variable.

$$A(2,3) = B + C * D ;$$

Resulting code looks like:

	CM_i	CM_{i+1}	CM_{i+2}	CM_{i+3}	CM_{i+4}	CM_{i+5}	CM_{i+6}	CM_{i+7}
i=0	,	token const	2	CNDL	token const	3	CNDL	T_1
i=8	↓	A	T_1	T'_1	*	C	D	T_1
i=16	+	B	T_1	T_1	=	T'_1	T_1	T_1
i=24	;							

The special symbol ↓ is an operator that instructs the software of the host computer to do matrix address calculation in order to assign the exact address to the subscripted variable. As can be seen, constants are preceded by their tokens(which also contain their address in the SYMTAB). CNDL delimits every constant that appears in the instruction. T'_m is a temporary address that will contain the calculated address of a subscripted variable.

Note: In the main computer's memory, there will be special storage addresses that provide temporary storage for results of calculation on an arithmetic statement. T_m, T_m' stand for these temporary locations.

In order to generate this arithmetic code, the SAU uses, in addition to the switching matrix, three push-down-lists,[3] one to store operators (R), one to store operands (RN), and one to store constants. The algorithm used to analyze arithmetic statements are described in full in Appendix K. The switching matrix used to analyze an arithmetic statement is shown in Figure 31. The way to consult the switching matrix is by using the top item of the operator push-down-list (RPDL) to locate the correct row of the matrix and by the type of item just read from the S-string to locate the correct column of the switching matrix. The column-row intersection contains the information on what operations to be carried out. In other words, which control sub-units to be energized.

Hardware Components

Push-Down-List Memory

The hardware implementation of a push-down-list memory has been demonstrated by William King (1971). The following is a brief description of that design.

---

[3] These push-down-lists are known as RPDL (operator PDL), RNPDL (operand PDL), and CNPDL (constant PDL).

	RN	;	←	,	+	-	*	/	↑	(	)	r
EO	SRN				U	U				LP		
;		D	S		S	S	S	S	S			
←		C	S	C	S	S	S	S	S		C	
,			S	S	S	S	S	S	S		C	
+		C		C	C	C	S	S	S		C	C
-		C		C	C	C	S	S	S		C	C
*		C		C	C	C	C	C	S		C	C
/		C		C	C	C	C	C	S		C	C
↑		C		C	C	C	C	C	C		C	C
(			S	S	S	S	S	S	S		RP	S
f		C		C	C	C	C	C	C		C	
↓	C	C	C	C	C	C	C	C	C		C	
r				S	S	S	S	S	S		C	
ES	LA											

Legend:

RN = operand

r = relational operators (=, ≠, <, ≤, ≥)

ES = expecting statement

EO = expecting operand

ES, EO, f, and are special operators generated by the analyzer.

C = code generator

U = unary operator

S = stack operator

SRN = stack operand

LA = label or assignment statement

D = done

RP = right parenthesis

LP = left parenthesis

Figure 31. Switching matrix of SAU

A left-right shift register, with serial input and output has the same characteristics as a push-down-list (PDL). If one assumes data entering and leaving at the left end of the register (see Figure 32), a stack operation simply means shifting the register one bit right and transferring data into the register. The unstack operation is just the reverse of the above process and is corresponding to shifting the register one position left.

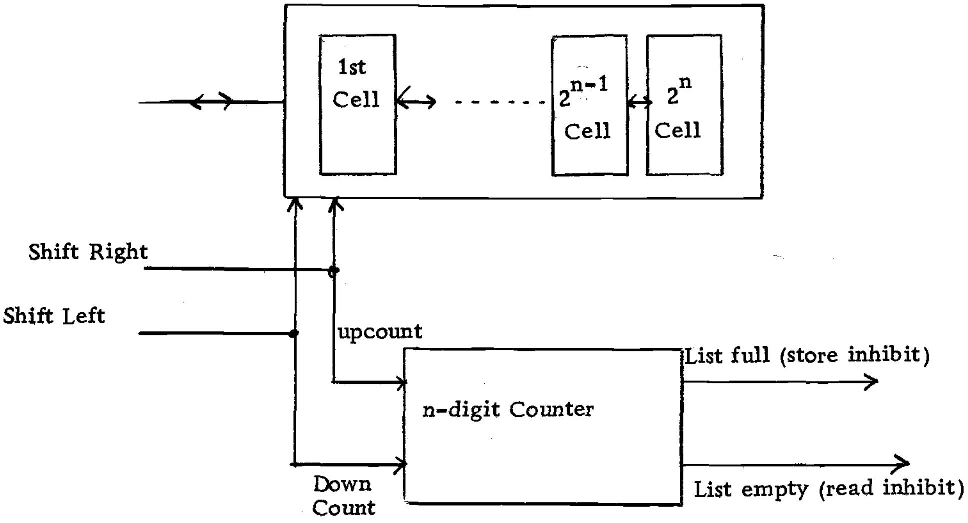


Figure 32. Block diagram of a PDL

In order to avoid stacking on an already full list, and also to give an indication if the list has become empty, an n-digit binary counter is used for a 2^n bit shift register. A shift right signal also increments the counter by one while a right shift command would decrement it by one. If after a stack operation the counter's content becomes 0, it means that the list is full and a stack inhibit signal will be issued. Conversely, if the counter shows a 0 after an unstack operation, it means that the list is now empty. In order to build a PDL of m bits per item, m shift registers may be put in parallel under the control of one common counter. Figure 31 is the block diagram of a PDL.

Special Character Generator

This code generator is in the form of an encoder which generates all the special characters that are necessary during the syntax analysis operation.

Code Memory (CM)

The CM, as has been mentioned before in Section IV, provides storage for the intermediate code generated by the SAU. It is a 2K, 16 bit-per-word, semiconductor memory. The CM has its own address counter (CMAC) and buffer register (CMBR).

Switching Matrix

The hardware implementation of the switching matrix shown in Figure 31 is by means of a ROM decoder with 32 input bits. Its block diagram is illustrated in Figure 33. The first 16 bits of the decoder are connected to the output lines of the RPDL, and the rest of the input bits are tied to the CMBR. The choice of 16 bits for each set of inputs is due to the fact that the S-code (Table 11) is a 16 bit code. The output line that becomes high as a result of decoding the input information determines which set of actions is to take place.

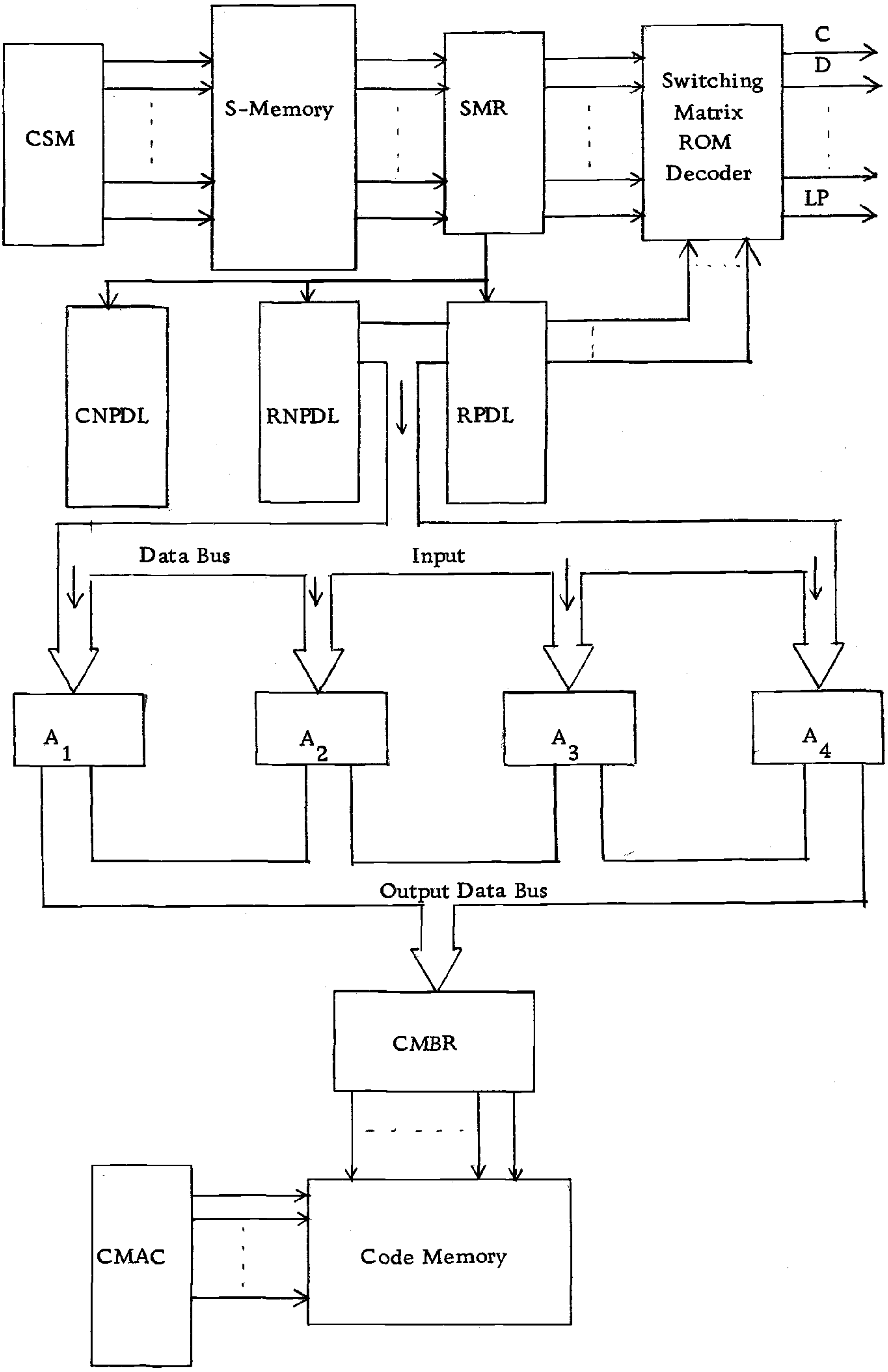


Figure 33. Block diagram of hardware components associated with SAU

Auxilliary Registers (A_1, A_2, A_3, A_4)

There are 4 such registers all of which are 16 bits long. They are used to provide temporary storages for items poped from the push-down-lists during the code generation process.

The use of these components may be more clear by the description of the SAU's algorithm in the next section.

General Description of the Syntatic Analysis of Arithmetic Statements

It should be recalled that the S-string generated by the ALAU is stored in the syntax memory (SM). To analyze this string, the SAU first stacks the symbol ";" and then the special symbol "ES" in the operator push-down-list (RPDL). It then clears the code memory address register (CMAC) and resets the counter N. After this initializing procedure, the SAU is ready to examine the S-string.

The SAU first fetches one item from the string (reads one word from the syntax memory) and transfers it to the code memory buffer register (CMBR). Using the contents of the CMBR and those of the top item of the RPDL as inputs to the switching matrix decoder (SWMDC), the SAU can energize the SWMDC to obtain the set of actions that it has to perform next. The flow charts describing the actions of the SAU are shown in Appendix J.

Running Account of the SAU's Handling of One Arithmetic Statement

Table 16 illustrates the step by step analysis of the arithmetic statement

$$A = B * C + (D + E) ;$$

The first column of the table contains the information on the nature of the top item of the RPDL, the second column that of the item just read from the S-string. The third column indicates which set of actions must take place, while the forth column describes the exact nature of these actions.

As can be seen from the Table the resulting code becomes :

*	B	C	T_1	+	D	E	T_2	+	T_1	T_2	T_1	+	A	T_1	T_1	;

After the SAU is finished with a statement, it will energize the input unit to accept a new Fortran instruction.

Operating Time Required by the SAU to Analyze Arithmetic Statement

The operating time required by the SAU to deal with the arithmetic statement given in the above section can be approximated by counting how many microoperations have to take place during the analysis process. Assuming again that each non-memory-reference operation takes 0.2 μsec and each memory reference operation 0.6 μsec, one can find from the running analysis given in the above section

Table 16. Running account of the analysis of an arithmetic statement.

RPDL	Item	Action	Detailed description of actions
		initial	SR ;, SR ES, read.
ES	A	LA	Pop R, SR EO, process.
EO	A	SRN	SRN A, Pop R, read.
;	=	SR	SR =, SR EO, read.
EO	B	SRN	SRN B, Pop R, read.
=	*	SR	SR *, SR EO, read.
EO	C	SRN	SRN C, Pop R, read.
*	+	C	generate code in C memory $\boxed{*}\boxed{B}\boxed{C}\boxed{T_1}$ SRN T_1, process.
=	+	SR	SR +, SR EO, read.
EO	(	LP	Pop R, SR (, SR EO, read.
EO	D	SRN	SRN D, Pop R, read.
(	+	SR	SR +, SR EO, read.
EO	E	SRN	SRN E, Pop R, read.
+	)	C	generate code in C memory. $\boxed{+}\boxed{D}\boxed{E}\boxed{T_2}$ SRN T_2, process.
(	)	RP	Pop R, read.
+	;	C	generate code in C memory. $\boxed{+}\boxed{T_1}\boxed{T_2}\boxed{T_3}$ SRN T_1, process.
=	;	C	generate code in C memory $\boxed{=}\boxed{A}\boxed{T_1}\boxed{T_1}$ SRN T_1, process.
;	;	D	done. Energize input unit.

Notations used in Table 16:

SR : stack an item on RPDL

SRN : stack an item on RNPDL

Pop R: remove top item from RPDL

that the time taken by the SAU is roughly

$$T_{s1} = \overset{\text{(non-memory reference)}}{\overset{\text{(microoperation)}}{(40 \times 0.2)}} + \underset{\text{(memory reference)}}{(40 \times 0.6)}$$

$$= 32.0\ \mu\text{sec}$$

The above figure is a very rough estimate, only serves to give a ballpark indication of the speed of the SAU in analyzing an arithmetic statement.

Expansion of Switching Matrix to Include IF Statements

To include non-arithmetic statements in the analysis, an expansion of the switching matrix can be carried out. This section demonstrates how such an expansion may be carried out to include one type of IF statement analysis.

There are four types of IF statements allowed in Fortran IV. Here the analysis of only one of them is examined, leaving the understanding that other forms of IF statements may be handled similarly. The IF statement type chosen represents the most commonly used in programming,

$$\text{IF (arithmetic statement) } n_1, n_2, n_3;$$

An example of this type of IF statement is

```
IF (A - B) 1, 2, 3;
```

The code generated by the HC for this IF statement type is

—	A	B	T_1	IF	T_1	1	,	T_1	2	,	T_1	3	;

The interpretation of this code is

1. subtract B from A and store results in T_1 ;
2. If T_1 is less than zero jump to address 1 ;
3. If T_1 is equal to zero jump to address 2 ;
4. If T_1 is greater than zero jump to address 3.

Table 17 shows the extra elements that must be added to the original switching matrix to generate such a code for the IF statement.

Table 17. Additional elements for switching matrix to include IF statement.

	IF	(	R	)	SN	,	;	Item just read
ES	ELP							
ELP		SIP						
ILP			SR	ESN				
ESN					S			
IF						IFC		
,						IFC	IFC	
;						SR	DONE	
Top of RPDL								

The description of the exact actions of SIP, SR, ELP, ESN, S, IFC, SR, and DONE are given in Appendix K.

Pop RN.

Running Account of the Analysis of One IF Statement

To clarify the concepts of analyzing an IF statement presented in the previous section, Table 18 presents the step by step analysis of

IF (A - B) 1, 2, 3 ;

Table 18. Running analysis of IF statement.

RPDL	Item	Action	Detailed description of actions
		Initial	SR ;, SR ES, read.
ES	IF	ELP	Pop R, SR ELP, read.
ELP	(	SIP	Pop R, SR ILP, SR EO, read.
EO	A	SRN	Pop R, SRN A, SR EO, read.
ILP	-	SR	SR -, SR EO, read.
EO	B	SRN	SRN B, Pop R, read.
-	)	C	generate code in C memory
			- A B T_1
			SRN T_1, Process.
ILP	)	ESN	Pop R, SR ESN, read.
ESN	1	S	SRN 1, Pop ESN, read.
IF	,	IFC	generate IF code in C memory
			IF T_1 1
			SRN T_1, Process.
;	,	SR	SR, ", SR ESN, read.
ESN	2	S	SRN 2, Pop R, read
,	,	IFC	generate IF code in C memory
			, T_1 2
			SRN T_1, Process.
;	,	SR	SR ",", SR ESN, read.
ESN	3	S	SRN 3, Pop R, read.
,	;	IFC	generate IF code in C memory
			, T_1 3
			SRN T_1, Process.
;	;	Done	Pop R, ; to C memory Pop RN.

The operating time T_{IF} needed to syntax analyze this statement can be approximated by

$$T_{IF} = \text{Number of microinstructions} \times 0.2\ \mu\text{sec}$$

$$+ \text{ number of memory references X.}6\ \mu\text{sec}$$

$$\simeq 35 \times 0.2\ \mu\text{sec} + 30 \times 0.6\ \mu\text{sec}$$

$$= 7.0\ \mu\text{sec} + 18\ \mu\text{sec}$$

$$= 25\ \mu\text{sec}$$

Inclusion of DO Statements

A DO statement in its source form is

DO (statement number) $I = d_1, d_2, d_3$;

For example, DO 123 I = 1, 4, 1;

The lexical analyzer converts this DO statement into the following S form:

DO	02	SYMTAB address	01	SYMTAB address	=	30	SYMTAB address	1	CNDL	. . .

It is desired that a C-string be produced by the syntax analyzer, such that conversion from this code to any machine code by software can be done with ease. The following string is such an example

=	I	Const.	1	CNDL	+	I	Const.	1	CNDL
I	-	I	Const.	4	CNDL	T_1	DO	T_1	123

*CNDL is a special symbol that delimits constants.

The interpretation of this code briefly is

1. assign 1 to I,
2. add 1 to I ,
3. subtract 4 from I and store results in T_1,
4. jump to 123 if T_1 is positive, if not perform the next instruction in the program.

Table 19 shows the extra elements that must be added to the original switching matrix to analyze DO statements.

Table 19. Extra elements for switching matrix to include DO statement.

	DO	SN	RN	=	,	;
ES	DESN					
ESN		SEO				
EO			SRN			
DO				SR	SRD	DC1
=					DC1	
DO1					SR	DC3
,						DC2
;						DONE

The meaning of DESN, ESN, SEO, SRD, DC1, DC2, D3 is shown in Appendix L.

Running Account of the Analysis of One DO Statement

To clarify the concepts of analyzing a DO statement presented in the previous section, Table 20 presents the step by step analysis of

DO 123 I = 1, 4, 2;

As can be seen from the examples of expanding the switching matrix to analyze statements in their S-Forms, there is virtually no limit to the power of analyzing statements by switching matrix. If more statements are to be incorporated, all one needs to do is to expand the switching matrix elements. All expansions are carried out in a similar fashion as the ones that have been described. And it is felt that no additional examples are needed to prove the versatility of the switching matrix.

The hardward implementation of the SAU control unit can be carried out in the exact manner that has been previously described and will not be dealt with here.

Table 20. Example of the analysis of DO statement.

<table>
<tr><th>RPDL</th><th>Item</th><th>Action</th><th>Detailed description of actions</th></tr>
<tr><td></td><td></td><td>Initial</td><td>SR ;, SR ES, read.</td></tr>
<tr><td>ES</td><td>DO</td><td>DESN</td><td>Pop R, SR Do, SR ESN, read.</td></tr>
<tr><td>ESN</td><td>123</td><td>SEO</td><td>SRN 123, Pop R, SR EO, read.</td></tr>
<tr><td>EO</td><td>I</td><td>SRN</td><td>SRN I, Pop R, read.</td></tr>
<tr><td>DO</td><td>=</td><td>SR</td><td>SR =, SR EO, read.</td></tr>
<tr><td>EO</td><td>1</td><td>SRN</td><td>SRN 1, Pop R, read.</td></tr>
<tr><td>=</td><td>,</td><td>DC1</td><td>generate DO code, I→ (A)<table><tr><td>=</td><td>I</td><td>const.</td><td>1</td><td>:</td></tr></table>Process.</td></tr>
<tr><td>DO</td><td>,</td><td>SR</td><td>SR DO1, SR EO, read.</td></tr>
<tr><td>EO</td><td>4</td><td>SRN</td><td>SRN 4, Pop R, read.</td></tr>
<tr><td>DO1</td><td>,</td><td>SRDO</td><td>SR ",", SR EO, SRN (I), read.</td></tr>
<tr><td>EO</td><td>2</td><td>SRN</td><td>SRN 2, read.</td></tr>
<tr><td>,</td><td>;</td><td>DC2</td><td>generate DO code<table><tr><td>+</td><td>I</td><td>2</td><td>const.</td><td>I</td><td>:</td></tr></table>SRN (4) Process.</td></tr>
<tr><td>DO1</td><td>;</td><td>DC3</td><td>generate<table><tr><td>-</td><td>I</td><td>const.</td><td>4</td><td>:</td><td>T</td></tr></table>SRN T, Process.</td></tr>
<tr><td>DO</td><td>;</td><td>DC1</td><td><table><tr><td>DO</td><td>T</td><td>123</td></tr></table></td></tr>
<tr><td>;</td><td>;</td><td>DONE</td><td></td></tr>
</table>

The operating time required for syntax analyzing the given DO statement is given by

$$
\begin{aligned}
T_{DO} &= (\text{approximate no. of non-memory microoperation}) + (\text{appr. no. of memory reference microoperations}) \\
&\simeq 25 \times 0.2 + 35 \times 0.6\ \mu\text{sec} \\
&\simeq 28.0\ \mu\text{sec}.
\end{aligned}
$$

XII. OUTPUT UNIT

The function of the output unit is quite straightforward, and its implementation can be undertaken in the manner already discussed and hence will not be elaborated here. The output unit transfers the C-code produced by the HC to some auxilliary storage device and issues an interrupt request to the main computer to inform it that a half compiled program is waiting to go through the final stage of compilation.

However, if errors have been found associated with the program, the output unit will abort the C-code transferral. Instead, it will transfer all the error messages to the auxilliary storage device and gives out an error interrupt request to the main computer. This interrupt informs the main computer that it need not perform the final compilation of the program. It should, however, print out all the error messages that have been found in the original program.

XIII. SUMMARY AND CONCLUSIONS

A preliminary study on the possibility of implementing a Fortran compiler with hardware has been presented. The marketability of such a product has been demonstrated and has been found to be convincing. In the typical usage environment of a computer system, up to 30% of the CPU time can be spent on Fortran compiling.

The control of the hardware compiler is realized by programmable logic arrays (PLA). The advantage of PLA control over software stored program is a tremendous gain in speed. The PLA has a response time of 60 nsec, compared with about 1 μsec for the fastest RAM storage of software routines. Table 21 summarizes the typical operating time that can be expected from the operation of each of the control units of the hardware compiler. It should be noted the time given is only a very rough indication of the speed of the hardware compiler.

Some of the design discussions given are admittedly sketchy in certain areas. They are intended for setting ground works for further detailed design projects.

Further works to improve the design of the system are obviously needed. Speed may increase by some form of parallelism in the operations of the functional units of the hardware compiler. In other words, the operations of the functional units need not be sequential. For

Table 21. Typical operating time of control units.

Unit	Function	Operating Time (μsec)
ICU	input statements	number of characters x period of data transmission
scanner	identify arithmetic from non-arithmetic statements	9.4
SKR	skip one record	
VMU	variable match	5.4
SNS	statement number search	6.0
ALAU	arithmetic lexical analyse	37.0
CNSU	constant unit	5
DLAU	DO lexical analyze	36.4
IFLAU	IF lexical analyze	46.2
SAU	arithmetic statement	32.0
	DO statement	28.0
	IF statement	25

example, the input of another source instruction does not have to happen after the preceding instruction has been completely processed. A master control unit should probably be installed to sequence the operations of each control unit and allow independent ones to occur simultaneously.

To optimize the cost-performance ratio of the design, a more careful selection of hardware components may be taken. Other studies should be made towards a more detailed estimate of system cost, whether it meets projected marketable price, and which parts of it may be streamlined to obtain a better cost/performance ratio.

When the project was first undertaken, the author's knowledge of the mechanism of the compiler and the extent of its complexity was minimal. Having studied more the features of the compiler, he is convinced that a complete take over by hardware of the compiling process is impractical and impossible. The goal of designing a hardware compiler that can interface with more than one type of machines has generated a great deal of difficulties and headaches in the course of the design. The conclusion reached is that such a hardward compiler can be built but it simply cannot have all the elegant features a conventional compiler allows. A suggestion for further investigation is to utilize the ideas presented in this paper and aim at building a hardware compiler that will serve only one known machine.

BIBLIOGRAPHY

Barsamian, H. 1970. Firmware sort processor with LSI components. In: Proceedings of Spring Joint Computer Conference of American Federation of Information Processing Societies, Atlantic City. Vol. 36. Montvale, AFIPS. p. 183-190.

Barton, R. S. 1961. A new approach to the functional design of a digital computer. In: Proceedings of West Joint Computer Conference, Los Angeles. Vol. 19. Glendale, Griffin-Patterson, p. 393-396.

Bashkow, T. R. 1964. A sequential circuit for algebraic statement translation. IEEE Transactions on Computers C-13:102-105.

Bashkow, T. R., A. Sasson, A. Kronfeld. 1967. System Design of a Fortran Machine. IEEE Transactions on Computers C-16:485-499.

Beelitz, H. R., S. Y. Levy, R. J. Linhart. 1970. System architecture for large scale integration. In: Proceedings of Fall Joint Computer Conference of the American Federation of Information Processing Societies, Anaheim, 1967. Vol. 31. Thompson Books, Washington, D. C. p. 185-200.

Bernay R. A. 1972. An MSI concept for a binary search scanner. Computer Design 6:65-75.

Chu, Y. 1962. Digital computer design fundamentals. New York, McGraw-Hill. 481 p.

Joseph, E. C. 1967. Impact of large scale integration on aerospace computers. IEEE Transactions on Computers EC-16:558-561.

King, W. K. 1971. Design of an associative memory. IEEE Transactions on Computers C20:671-674.

Hansk, A. A., B. A. Dent. 1968. Burroughs B6500/B7500 stack mechanism. In: Proceedings of Spring Joint Computer Conference of the American Federation of Information Processing Societies, Atlantic City. Vol. 33. Thompson, Washington, D.C., p. 245-251.

Hopgood, F. R. A. 1969. Compiling techniques. London, MacDonald. 126 p.

Husson, S. S. 1970. Microprotramming principles and practices. Englewood Cliffs, Prentice-Hall. 614 p.

Lee J. A. N. 1967. The anatomy of a compiler. New York, Reinhold. 275 p.

Ledley, R. S. 1962. Programming and utilizing digital computers. New York, McGraw-Hill. 567 p.

Levy, S. T., R. J. Linhardt, H. S. Miller, R. D. Sidnam. 1967. System utilization of L. S. I. IEEE Transactions on Computers EC-16:562-566.

Melbourne, A. J., J. M. Pugmire. 1965. A small computer for the direct processing of Fortran statements. Computer Journal 1:24-27.

Moore, G. E. 1971. Semiconductor RAMS- a status report. Computer. IEEE Computer Society, Vol. 4, No. 2, Mar.-Apr. p. 6-10.

Morris, R. 1968. Scattered storage techniques. Communications of ACM. 11:38-44.

Pullin, A. 1964. A Fortran to ALGOL translator. Computer Journal 1:24-26.

Rosen, S. 1968. Hardware design reflecting software requirements. In: Proceedings of Fall Joint Computer Conference of the American Federation of Information Processing Societies San Francisco. Vol. 33. Thompson, Washington, D. C. p. 1443-1449.

Rudenberg, H. G. 1969. Large-scale integration: Promises versus accomplishments - The dilemma of our industry. Proceedings of Fall Joint Computer Conference of the American Federation of Information Processing Societies, Houston. Vol. 35. Montvale, AFIPS, p. 359-367.

Thurber, K. J., R. O. Berg. 1971. Universal logic modules implemented using LSI memory techniques. In: Proceedings of the Fall Joint Computer Conference of the American Federation of Information Processing Societies, Las Vegas. Vol. 39. Montvale, AFIPS, p. 177-194.

Tonik, A. B. 1967. Development of executive routines, both hardware and software. In: Proceedings of Fall Joint Computer Conference of the American Federation of Information Processing Societies, Anaheim. Vol. 31. Thompson, Washington, D. C. p. 395-408.

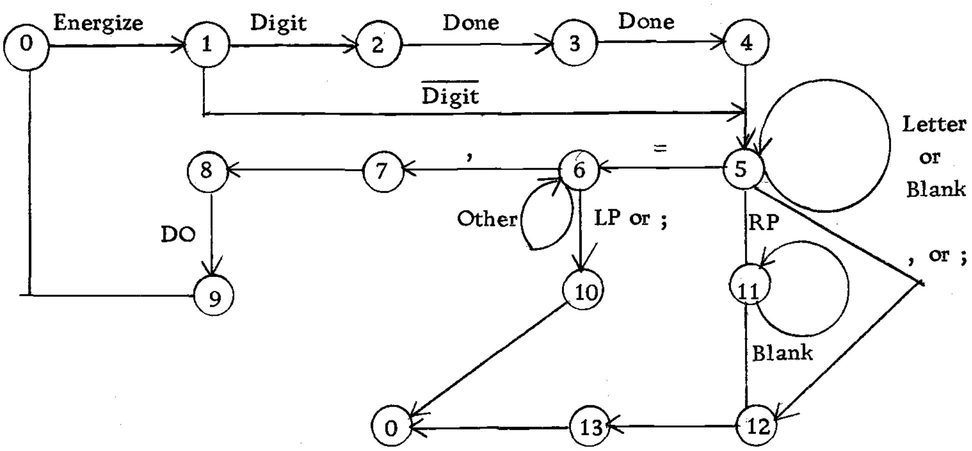
Vimari, D. C. 1970. Field-programmable read - only memories and applications. Computer Design 12:49-54.

White H. J., E. K. C. Yu. 1970. Use of read only memory in ILLIAC IV. Proceedings of Spring Joint Computer Conference of the American Federation of Information Processing Societies, Atlantic City. Vol. 36. Montvale, AFIPS, p. 197-205.

APPENDICES

APPENDIX A

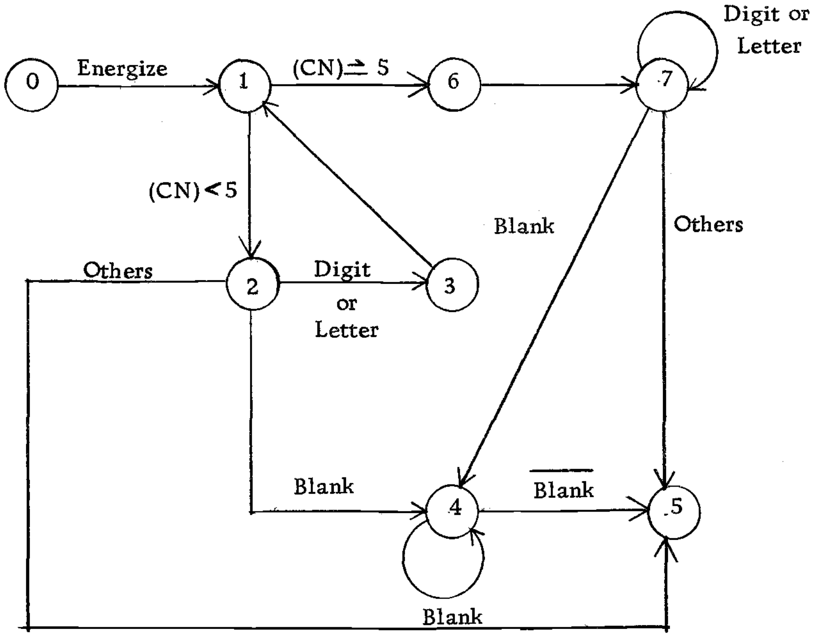
STATE DIAGRAM OF SCANNER



State	Action
0	Idle
1	Clear (SAR), Read IBM (RIBM), ↑CIB, ENDC
2	Energize Pack Symbol Unit
3	Set DF, Energize Statement Number Unit
4	Store (SMR), ↑CSM, (CIB)→SAR, RIBM, ↑CIB, ENDC
5	No op.
6	RIBM, ↑CIB, ENDC
7	(SAR)→CIB, Pack Symbol
8	Search Command
9	Energize DO Unit
10	(SAR)→CIB, Pack Symbol Unit, Search Command
11	R IBM, ENDC
12	(SAR)→CIB, Pack Symbol Unit, Search Command
13	Energize Respective LAU

APPENDIX B

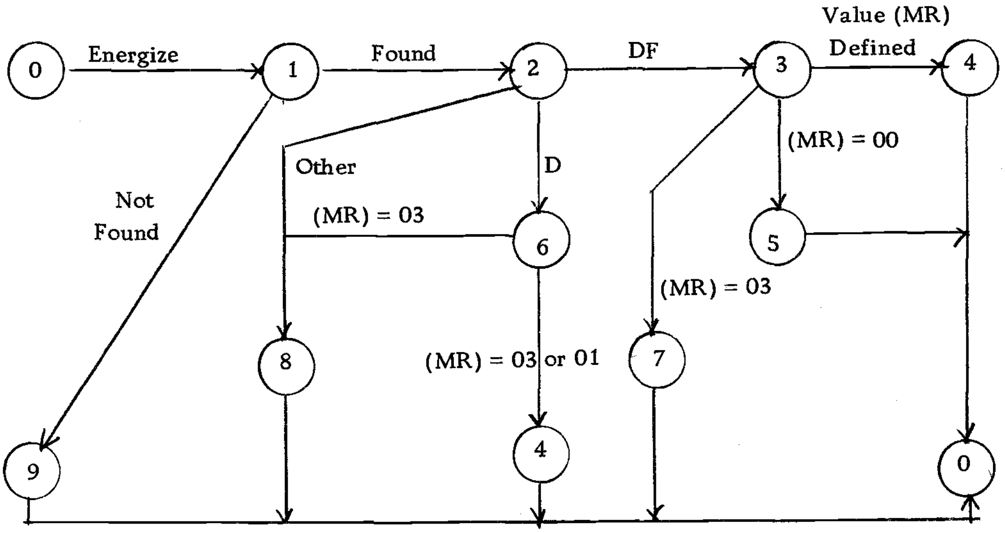
STATE DIAGRAM OF PACK SYMBOL UNIT



State	Action
0	Idle
1	(IMBR)→SHR, ↑CN, Read IBM
2	Energize DC
3	Shift SHR
4	Read IBM, ENDC
5	Clear CN, DONE
6	Energize DC
7	Read IBM, ENDC

APPENDIX C

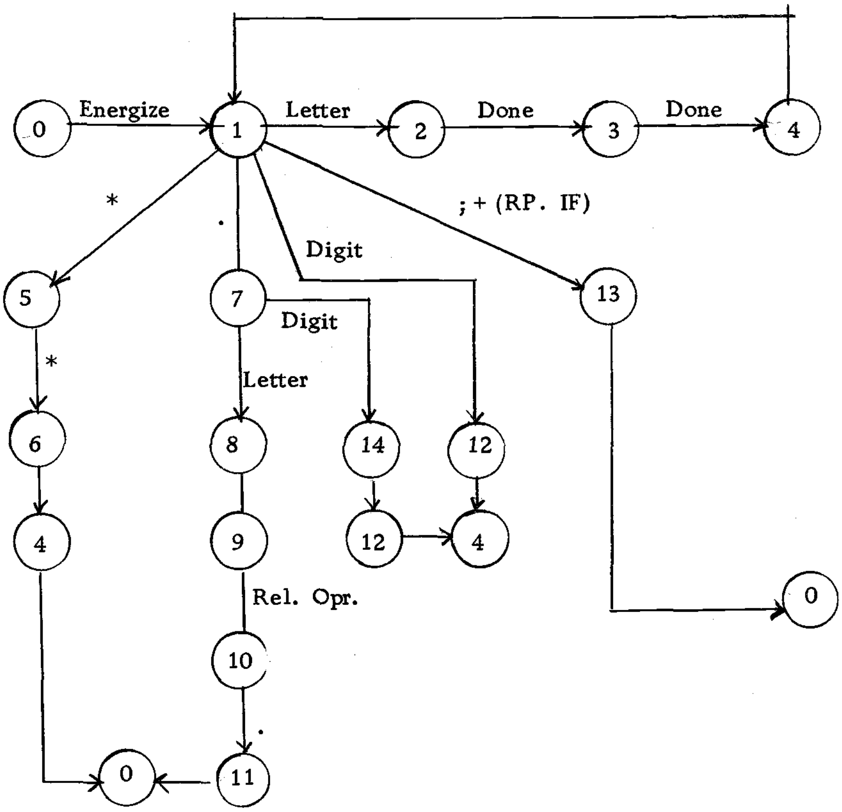
STATE DIAGRAM OF STATEMENT NUMBER SEARCH



State	Action
0	Idle
1	Energize Search Memory
2	Test DF
3	Test Value (MR)
4	Error
5	Change Value (MR) to 01, Write in SYMTAB, (MR)→SMR
6	Test Value (MR)
7	Change Value (MR) to 03, Write in SYMTAB, (MR)→SMR
8	(MR) → SMR
9	(NR) + Code + (AR)→MR Write in SYMTAB, (MR) → SMR

APPENDIX D

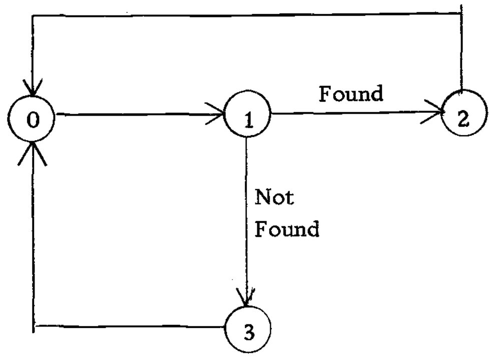
STATE DIAGRAM CF ALAU



State	Action
0	Idle
1	Read IBM, ↑CIB, ENDC
2	Pack Symbol
3	Energize Variable Match Unit (VMU)
4	Store in SM ↑CSM
5	Read IBM, CIB, ENDC
6	↑→ SMR
7	Read IBM, ↑CIB, ENDC
8	Pack Symbol
9	Energize Command-Encoder-Decoder
10	Read IBM, ↑CIB, ENDC
11	S-Code→SMR, Store SM, ↑CSM
12	Energize Constant Unit
13	S-Code→SMR, Reset CSM
14	↓ CIB

APPENDIX E

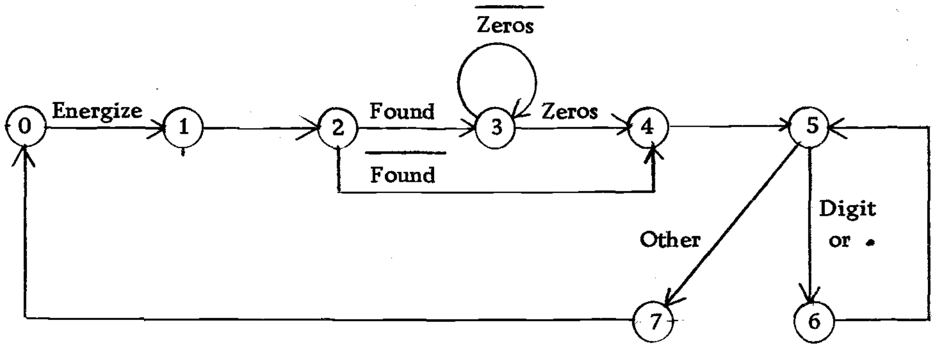
STATE DIAGRAM OF VARIABLE MATCH UNIT



State	Action
0	Idle
1	(SHR)→NR, Energize Search Memory
2	Value (MR)→SMR
3	(NR)→$(MR)_{0-29}$, Variable Code→$(MR)_{30-35}$ Write (MR), (MR)→SMR

APPENDIX F

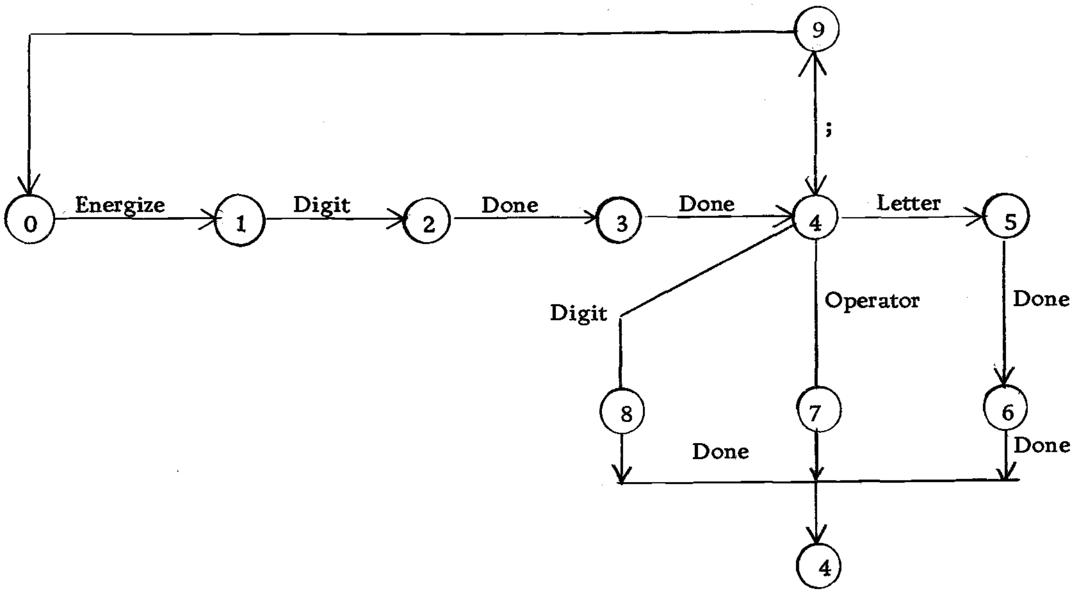
STATE DIAGRAM OF CONSTANT UNIT



State	Action
0	Idle
1	(SHR)→NR, Energize Search Memory
2	↑AR, Fetch, Test (MR)
3	$(NR) \rightarrow (MR)_{0-29}$, $30_8 \rightarrow (MR)_{30-35}$, $(AR) \rightarrow (MR)_{36-45}$
4	Write in Search Memory, (SAR)→CIB
5	Read IBM, ENDC
6	(IBMR)→SMR, Write SHR, ↑CIB, ↑CSM
7	(IBMR)→SMR, Write SHR, ↑CIB

APPENDIX G

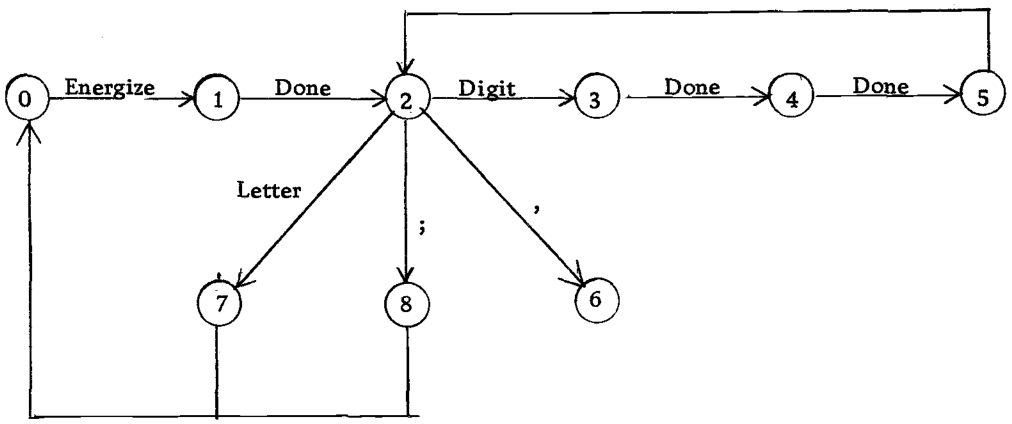
STATE DIAGRAM OF DO LEXICAL ANALYZER



State	Action
1	DO Code→SMR, Store SM, ↑CSM, ENDC
2	Pack Symbol, DO flag
3	Search Statement Number
4	Write (SMR), ↑CSM, ENDC
5	Pack Symbol
6	Search Statement Number
7	Operator Code→SMR, Write (SMR), ↑CSM, Read IBM, ↑CIB
8	Constant Unit
9	;→SMR, Write (SMR), Clear CIB, Clear CSM

APPENDIX H

STATE DIAGRAM OF IF LAU



State	Action
0	Idle
1	IF Code→SMR, Write SM, ↑CSM, Set IF flag Energize ALAU
2	ICIB, Read IBM, ENDC
3	Pack Symbol
4	Search Statement Number
5	Write SM
6	, → SMR
7	Energize Scanner
8	;→SMR, Write SM, ISMC, Done

APPENDIX I

Go To Lexical Analyzer Unit (GOTO LAU)

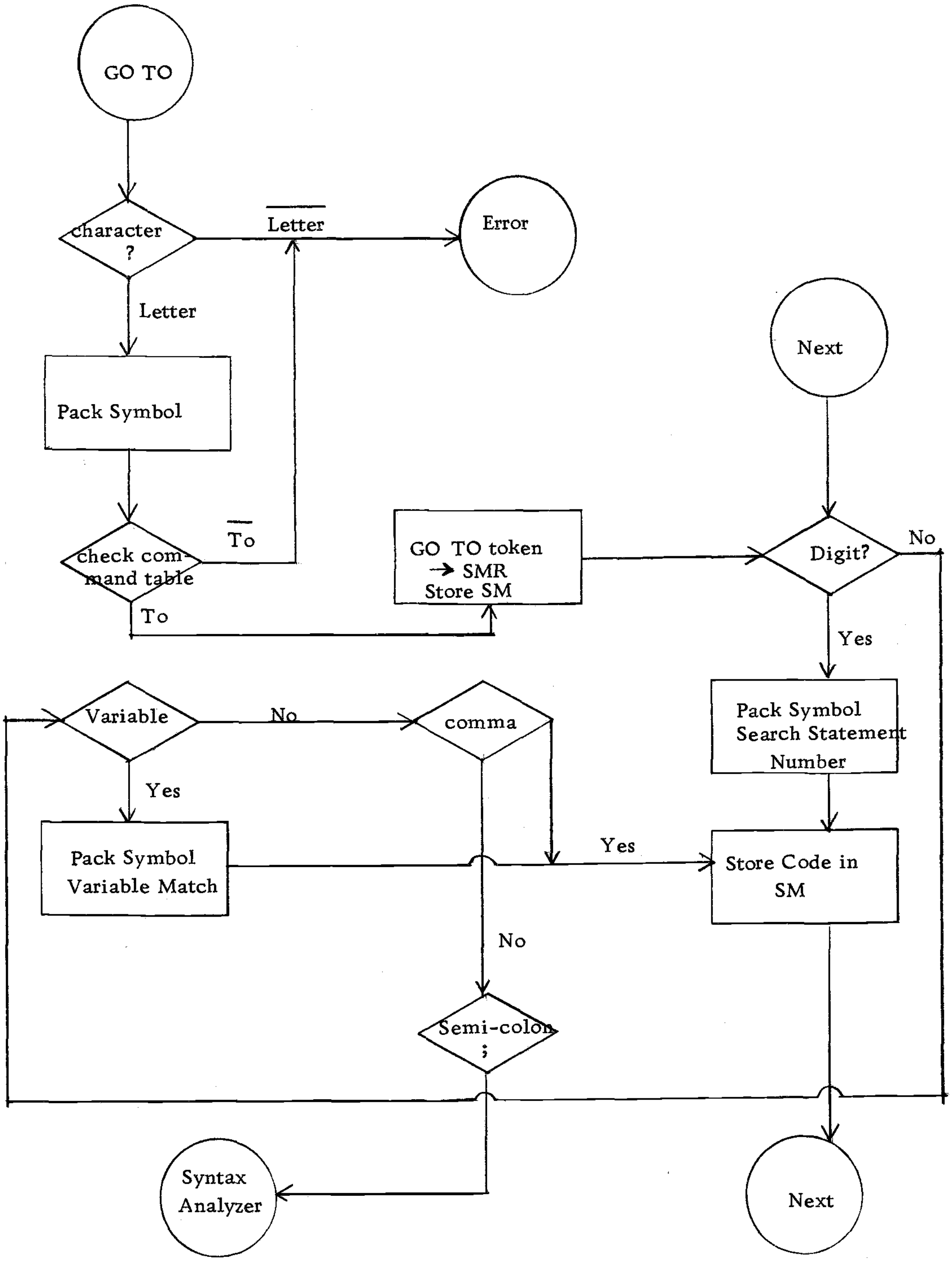
There are two kinds of GO TO statement which are allowed by the hardware compiler.

GO TO "statement number" ;

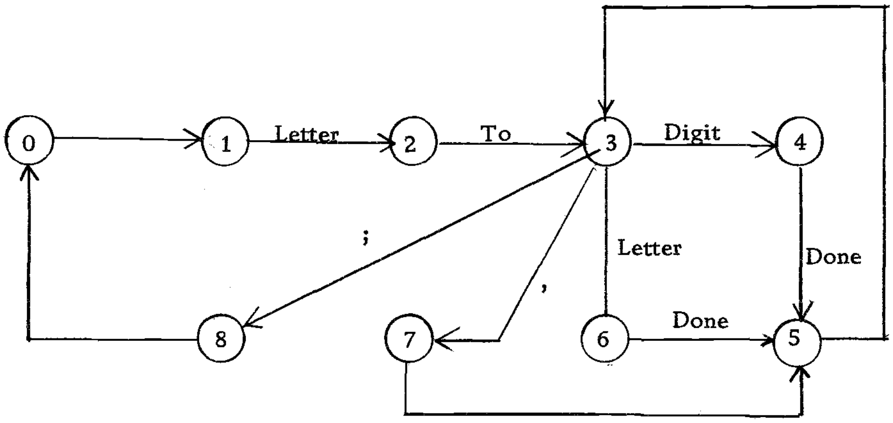
GO TO (n_1, n_2, ...) i ;

The first type of GO TO statement allows unconditional jump to the statement labeled by the statement number. The second type of GO TO, also known as COMPUTE GO TO, allows conditional control transferral to one of the statements labeled by the list of statement numbers found within the parentheses, depending on the value of the integer variable.

The GOTO LAU is energized by the scanner when it has detected that a non-arithmetic statement is a possible GO TO because it has the command GO as its first non-blank symbol. The first thing that the GTLAU does is to check the next symbol of the source statement to see if it is a TO. If it is not, an error condition has arisen. If it is indeed a TO, there is a good chance that this is meant to be a GOTO statement, unless the programmer has inadvertently made a mistake. The GTLAU will go on reading each character of the instruction, converting each digit group into statement number equivalent, each comma and parenthesis by its syntax code. Finally when it reaches the ;, it knows that it has come to the end of the statement. After storing the S-code for ;, it issues a signal to energize the SYNTAX ANALYZER. The flow chart and the state diagram of the GOTO LAU are shown in I. 1 and I. 2 respectively.



Appendix Table I. 1. Flow chart of GOTO LAU



State	Action
0	Idle
1	Decode Character
2	Pack Symbol, Check Command Table
3	Decode
4	Pack Symbol, Statement Number Search
5	Store in SM, ↑ CSM
6	Pack Symbol, Variable Match
7	Comma → SMR, ↑ CIB, Read (IBM)
8	; → SMR, Reset CSM, Energize Syntax Unit

Appendix Figure I. 2. State Diagram of GOTO LAU

APPENDIX J

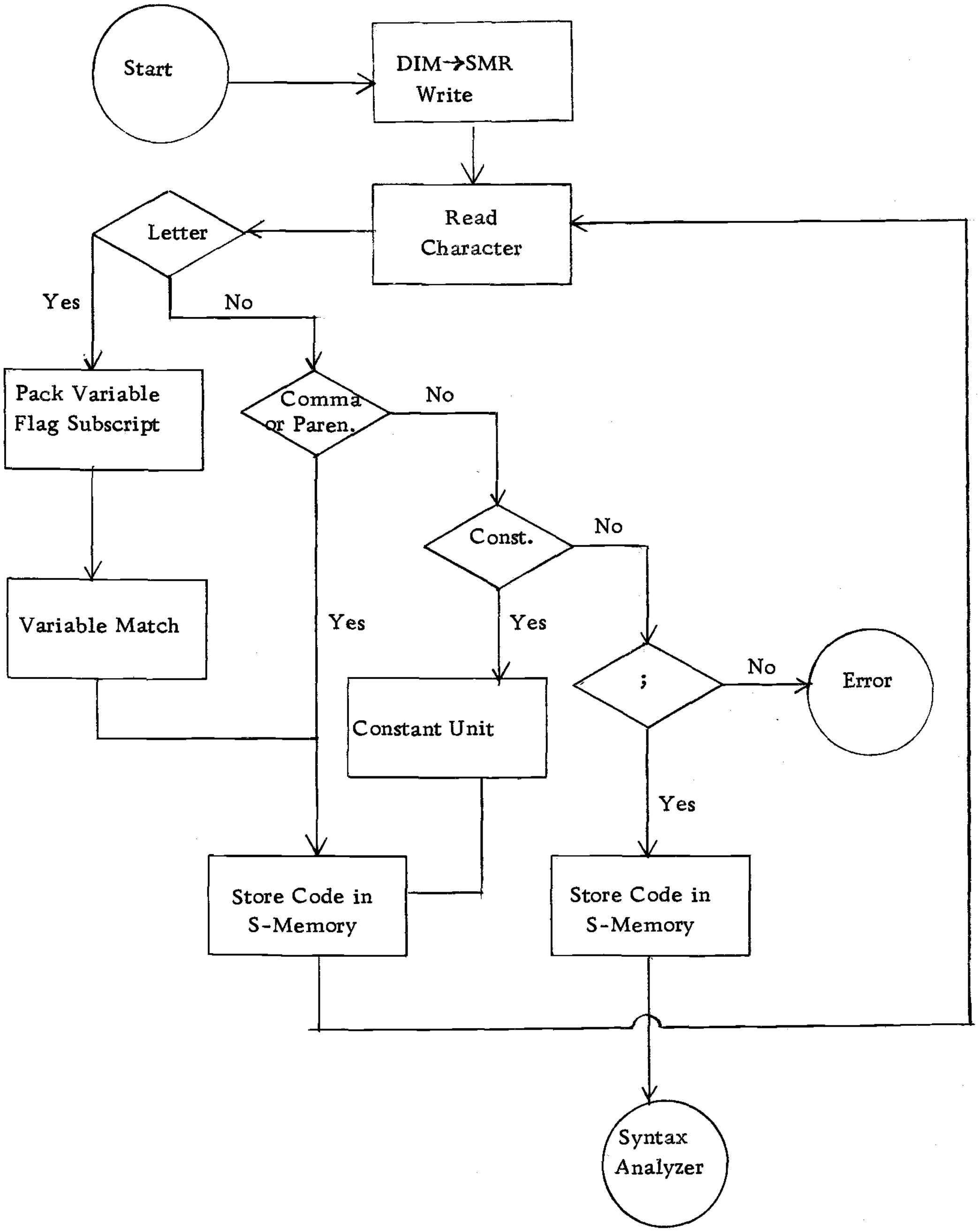
DIMENSION LAU

An example of a DIMENSION statement is

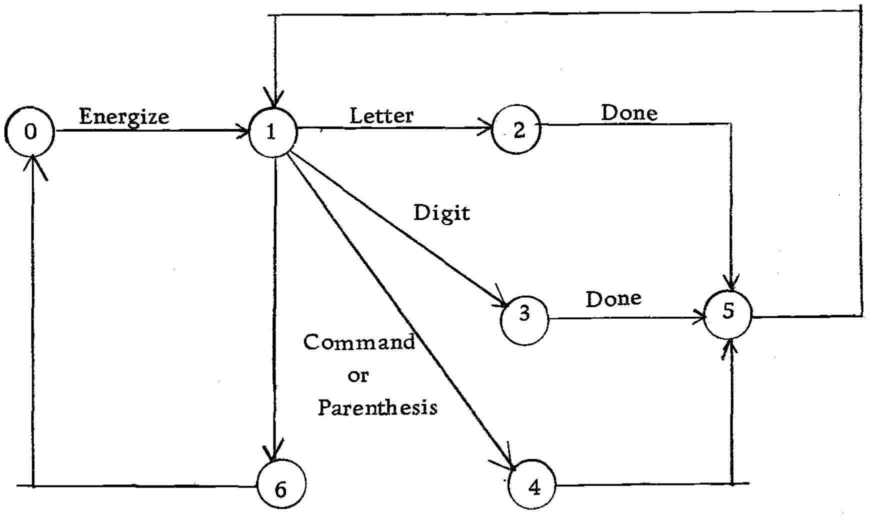
DIMENSION X_1 (d_1, d_2, ... d_n), X_2 (d_1, ... d_n) ;

The purpose of a Dimension statement is to define variables as n-dimensional arrays. DIMENSION statements must always appear before any executable statements in a program.

The DIM LAU first stores the S-code for DIMENSION in the S-memory. Then it flags the "subscript flag" when it encounters a variable to inform the variable match unit that the variable is being defined as a subscripted variable. When it encounters a constant, it energizes the constant unit. However, when it reads a parenthesis, or a comma, it just simply stores the S-code for the symbol in the S-memory. When it reads a semi-colon, it's job is done and will energize the SYNTAX Analyze Unit.



Appendix Figure J. 1. Flow chart DIMENSION LAU



State	Action
0	Idle
1	DIM→SHR, Write SM Decode.
2	Pack Variable, Flag Subscript Flip-Flop, Energize Variable Match.
3	Constant Unit.
4	(IBMR)→SMR, READ IBM, ↑CIB.
5	Store Code in SM, ↑CSM.
6	Store Code in SM, Reset SMC Energize Syntax Unit.

Appendix Figure J. 2. State Diagram of DIM LAU

APPENDIX K

MICRO-OPERATIONS OF THE SAU'S DEALING WITH ARITHMETIC STATEMENTS

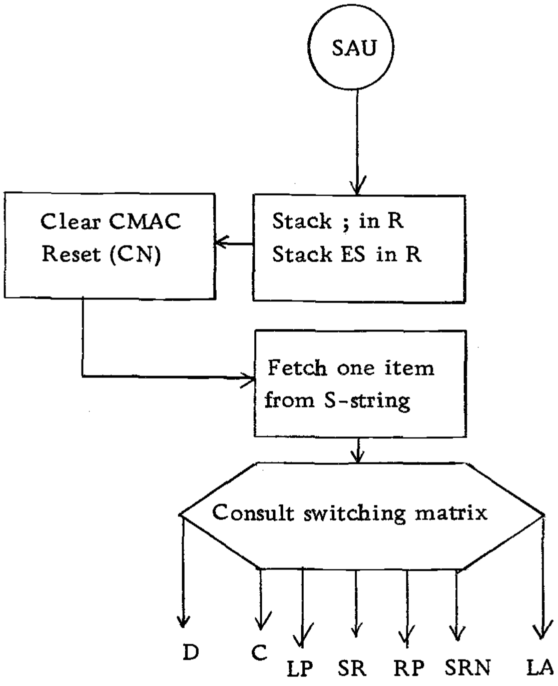


Figure K. 1. Flow chart of SAU

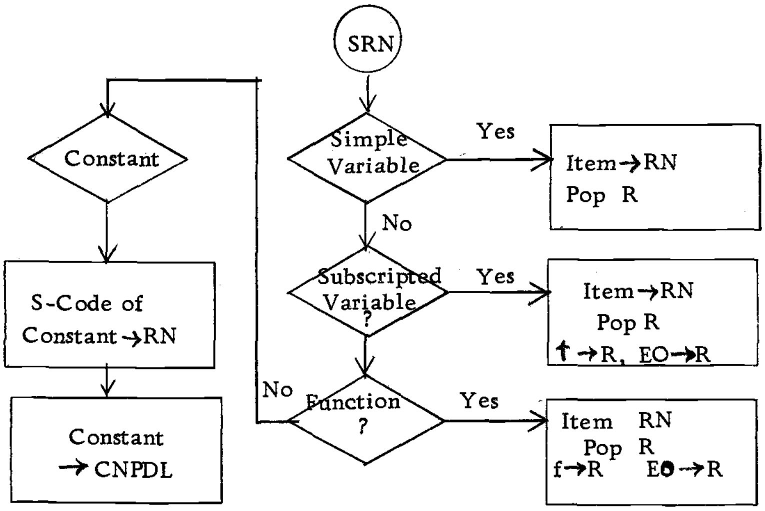
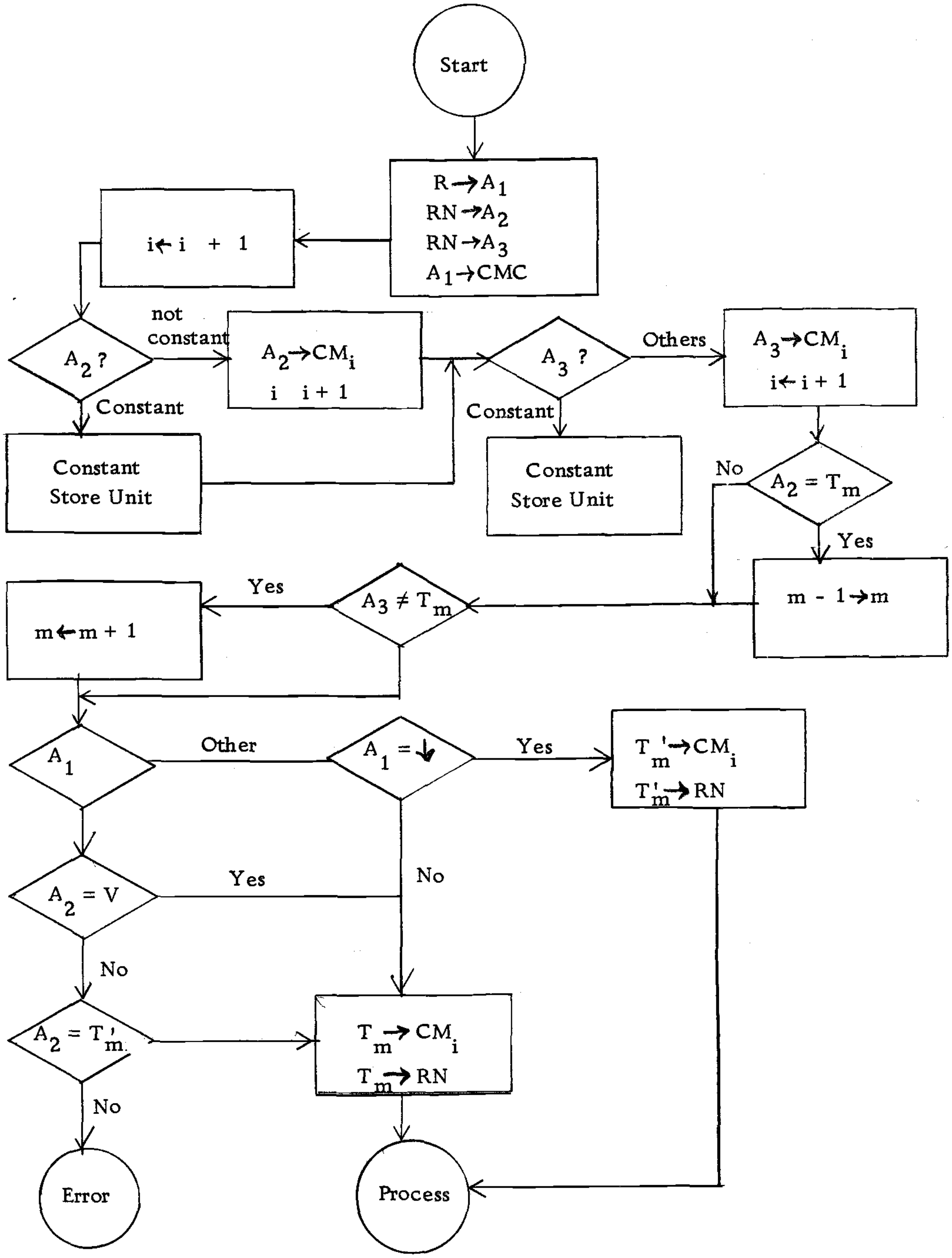
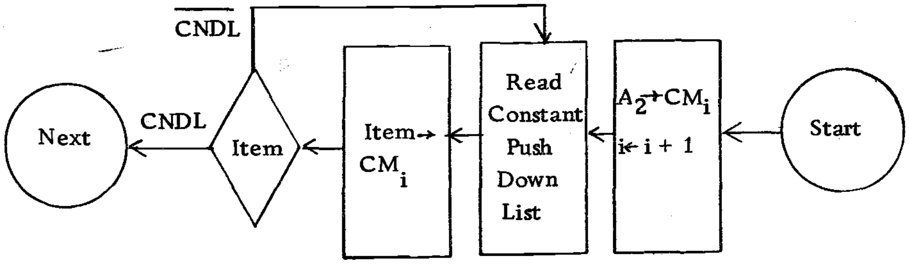


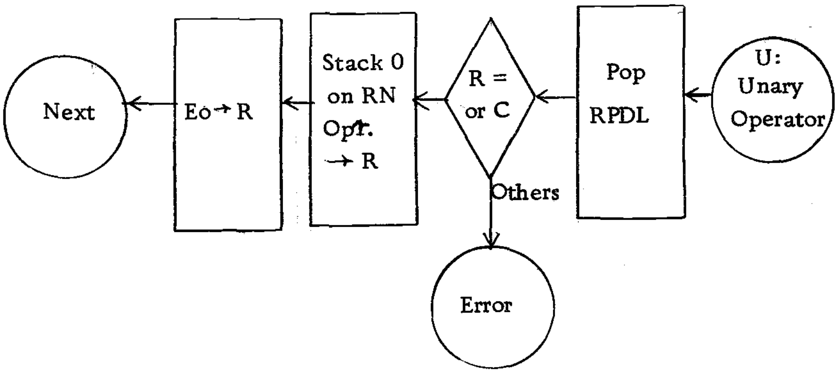
Figure K. 2. Flow chart of SRN



Appendix Figure K. 3. Flow chart of code generator

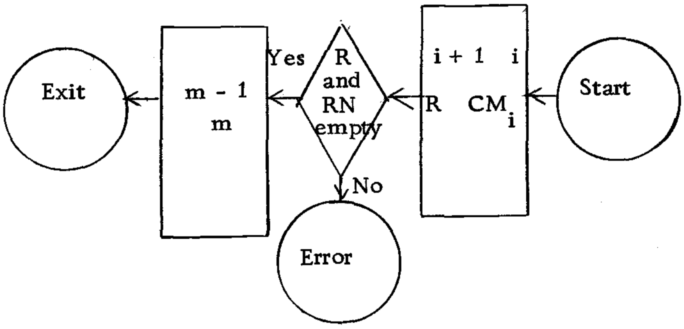


Appendix Figure K. 4. Flow chart of constant store unit



Appendix Figure K. 5

This unit is to take care of unary operators +, or -, when they are used in C = -A or D = (-A x C) + E.



Appendix Figure K. 6. Flow Chart for DONE

SR (stack operator):	Item→ R EO→ R, Go to Next
LP (left parenthesis):	Pop R; Item→ R, EO→ R; Go to Next
RP (right parenthesis):	Pop R, Go to Next
LA:	Pop R, EO→ R, Go to Next

Appendix Figure K. 7

APPENDIX L

MICRO-OPERATIONS OF IF SYNTAX ANALYSIS

ELP: Pop R push-down-list, Stack ELP in RPDL, read next word from syntax memory.

SIP: Pop RPDL, stack ILP in RPDL, SR EO, read next word.

IFC: RPDL to Code Memory, RNPDL to code memory, RNPDL to code memory, stack T in RNPDL.

APPENDIX M

THE MICRO-OPERATIONS ASSOCIATED WITH DO SYNTAX ANALYSIS

DESN : Pop item from Operator push-down-list (RPDL), stack item DO in RPDL, stack ESN in RPDL, read next word from syntax memory.

SEO: Stack an item in operand push-down-list, pop top item from RPDL, stack EO in RPDL, read next word.

DC1: RPDL to CM_i, RNPDL to A register, (A) to CM_{i+1}, RNPDL to CM_{i+2}, process.

SRD: Stack DO1 in RPDL, stack EO in RPDL, read next word.

SRDO: Stack "," in RPDL, stack EO in RPDL, Stack (A) in RNPDL, read next word.

DC2: RPDL to CM_i, RNPDL to CM_{i+1}, RNPDL to CM_{i+2}, energize constant store unit. (A) to code memory.

DC3: - to CM_i, Pop RPDL, RNPDL, to CM_{i+1}, RNPDL to CM_{i+2}, T to CM_{i+3}, stack T in RNPDL, process.