Investigation of Sequential Machine Design Techniques for Implementation of a TRAC Scanning Algorithm

Raymond F. Cotton
University of Central Florida

INVESTIGATION OF SEQUENTIAL MACHINE DESIGN TECHNIQUES FOR IMPLEMENTATION OF A TRAC SCANNING ALGORITHM

BY

RAYMOND F. COTTON
B.S.E., University of South Florida, 1968

RESEARCH REPORT

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in the Graduate Studies Program of
Florida Technological University, 1973

Orlando, Florida

ABSTRACT

INVESTIGATION OF SEQUENTIAL MACHINE DESIGN TECHNIQUES FOR IMPLEMENTATION OF A TRAC SCANNING ALGORITHM

BY

RAYMOND F. COTTON

This report will demonstrate the design techniques to translate a given scanning algorithm into a hardwired pre-processor. The language to be "pre-processed" is TRAC (Text Reckoning and Compiling) devised by Mooers and Deutsch.

The major drawback in the current implementation of TRAC is speed. The software overhead required for string manipulations and execution of the input scanning algorithm is the major degrading factor. A TRAC machine consisting of a hardwired pre-processor to scan the input and produce formatted data for a stack oriented evaluator is proposed.

The control machine for the input scanning algorithm for the pre-processor is designed using various sequential machine design techniques.

The one-hot code and the minimum state variable design represent the two extremes which are presented.

TABLE OF CONTENTS

INTRODUCTION

A class of hardware/software trades which is of particular interest is the specialized system. One such specialized machine is the machine which is optimized to execute programs written in a higher level language. Such a machine is described in this paper. The higher language is TRAC.[1] This machine will offer a hardwired pre-processor plus an architecture which is tailored to provide specialized run time support for the functions provided by the language.

The intent of this report is to give insight into the pre-processor design. This will be given by demonstrating the techniques and steps required in the design of the state control machine. This control machine is the segment of sequential logic that controls the state sequence of the machine and the various support registers, counters, etc., as well as controlling the data flow. The techniques presented are not intended as the complete design, but only to take that one step further from concept to implementation.

The constraints of this report are that portions of the pre-processor, such as the interrupt and data transfer sequences will be generalized and presented as plausible concepts for eventual implementation in a future system.

Furthermore, certain aspects of the TRAC language such as diagnostics, error recovery, invalid statements, etc., will not be covered.

[1]The name TRAC is a trademark for a specific text-handling language that was developed and is being maintained by the Rockford Research Institute Inc., Cambridge, Mass.

CHAPTER I

SYSTEM OVERVIEW: TRAC LANGUAGE

In the TRAC language, one can write procedures for accepting, naming and storing any character string from the source; for modifying any string in any way; for treating any string at any time as an executable procedure, or as a name, or as a text; and for printing out any string. The TRAC language is based upon an extension and generalization to character strings of the programming concept of the "macro." Through the ability of TRAC to accept and store definitions of procedures, the capabilities of the language can be indefinitely extended. TRAC can handle iterative and recursive procedures, and can deal with character strings, integers and Boolean vector variables. (1)

The advantage of the TRAC language is that it provides (i) high capability in dealing with back-and-forth communications between an operator at a terminal and the machine, so as to allow him to make insertions and interventions during the running of his work; (ii) maximum versatility in the definition and performance of any well-defined procedure on text; (iii) ability to define, store and subsequently use such procedures to extend the capabilities of the language; and finally (iv) maximum clarity in the language chosen. (1)

A TRAC string may contain a substring enclosed by a matching pair of parentheses, such as (···) where the dots indicate a string. The matching parentheses indicate the scope of some particular action.

There are three cases, represented by #(···), ##(···) and (···). The first two formats indicate the presence of a TRAC "primitive function." The format #(···) denotes an "active function," while the format ##(···) denotes a "neutral function." This distinction is clarified below. The string interior to either kind of function is generally divided into substrings by commas as in #(-,-,-) where these substrings constitute the arguments of the function. Parentheses in the format (···) have roughly the same role as paired quotation marks, and, in particular, whatever string is inside the paired parentheses is protected from functional evaluation.

TRAC strings are dealt with by the pre-processor according to a scanning algorithm which works from left to right and obtains the evaluation of nested expressions from inside outward. In the expression

```
#( ,#( , , #( ) , ##( ) ))
4   3      1      2
```

the functions are evaluated in the order indicated. As each function is evaluated by the system, it is replaced in the TRAC string by the string which is its value. The evaluation of an active function is followed directly by the evaluation of any function in its value string not protected by matched parentheses. The value string of a neutral function is not further evaluated.

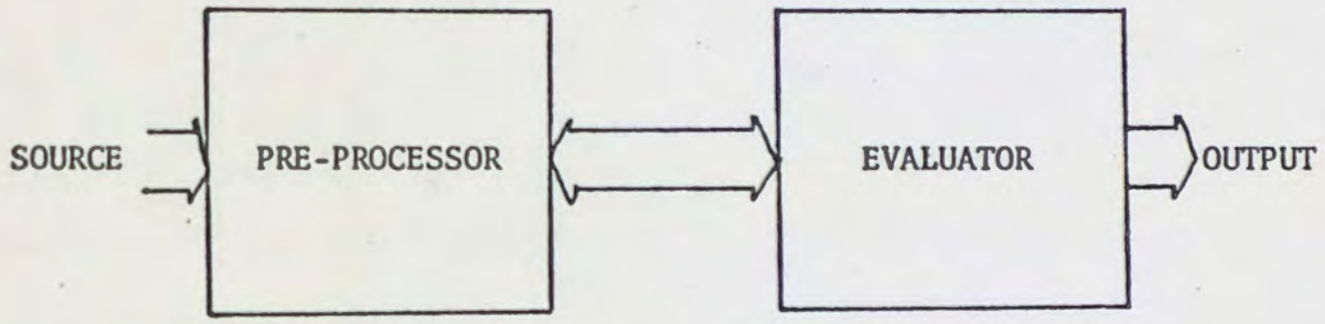
Currently TRAC expressions are scanned and evaluated by a software algorithm. At the begining, the unevaluated strings are in the "active string" and the "scanning pointer" points to the leftmost character in this string. As characters have been treated by the scanning algorithm, they may be added to the right hand end of a "neutral string," which is so called because its characters have been fully treated by the algorithm and are thus neutral like alphebetic characters. Thus, in software, execution of a TRAC instruction is performed by scanning successive characters in the active string and performing certain actions depending on the character being scanned. Unless one of the control symbols is encountered, characters are normally copied from the active string to the neutral string.

Unfortunately, the software overhead required for string manipulations and execution of the input scanning algorithm is quite prohibitive in obtaining an acceptable performance.

CHAPTER II

TRAC IMPLEMENTATION

A stack oriented TRAC processor is proposed. The processor consists of a pre-processor and an evaluator. This division is, as has been stated, in the interest of upgrading performance.



By keeping these two parts of the processor, the pre-processor and the evaluator, conceptually and physically separate, either the language or the evaluator primitives may be redesigned without extensive design changes. (2) In other words, if another language is chosen, only the pre-processor need to be redesigned to scan the new language and provide formatted data to the existing stack oriented evaluator. Conversely, if the evaluator is substituted it must have a stack oriented replacement. Hence, only half of the TRAC processor is affected.

The stack has some rather unique properties that aid in the compilation and evaluation of nested expressions. Stacks turn out to be a natural structure in a number of different programming applications. In particular, stacks crop up during the evaluation of expressions which are nested in other expressions. Therefore stacks whose entries are created and deleted in a last in, first out order but which permit access to information below the top of the stack are useful in the implementation of the TRAC language. (3)

The internal organization of single-address computers forces the wasting of both programming and running time for the storage and recall of the intermediate results in the sequence of computation. Before an operation can be executed the data must be placed into the proper registers and memory cells, and their contents must often be completely rearranged before the next operation can be performed. Multi-address computers are constructed to make the execution of a few selected operations more efficient, but at the expense of building inefficiencies into all the rest. To overcome the limitations of their internal organization most conventional computers require the wasteful expenditure of programming effort, memory capacity, and running time.

This problem may be attacked directly by the use of "pushdown/popup" stacks, which eliminate the need for instructions to store or recall immediate results. The Burroughs class of machines apply the stack concept. (4) An addition operation could expect to find its two arguments in the top two registers of the operand stack and the add operator in the top of the operator stack. After execution the result could be placed into the top register of the operand stack.

The source strings in the Burroughs machines are composed of strings of syllables. There are four types of syllables. The first of these, the operator syllable, causes operations to be performed. A second syllable, the literal syllable, is used for placing constants in the stack to be used as operands.

The other two syllables, the operand call and descriptor call syllables, address locations in a program reference table. The purpose of the operand call syllable is to place an operand in the stack. The purpose of the descriptor call syllable is to place the address of an operand in the stack.

In a Burroughs machine, such as the B-5000, the stack is composed of a pair of registers, the A and B registers, and a memory area. As operands are picked up by the programs by use of above mentioned syllables, they are placed in the A register. If the A register already contains a word of information, that word is transferred to the B register prior to loading the operand into the A register. If the B register is also occupied by information, then the word in B is stored in a memory area defined by an address register S. Then the word in A can be transferred to B and the operand brought into the A register. The new word coming into the stack has pushed down the information previously held in the registers. As each pushdown occurs, the address in the S register is automatically increased by one. The information contained in the registers is the last information entered into the stack. The stack operates on a "last in-first out" principle. As information is operated on in the stack, operands are eliminated from the stack and results of operations are returned to the stack. As information in the stack is used up by operations being performed, it is possible to bring a word from the memory area addressed by the S register, and the S register is decreased by one. In this manner, processing of data is accomplished without the need for instructions to store or recall intermediate results.

The pre-processor presented in this paper generates similar type strings to the stack oriented evaluator. The major difference will be that the evaluator will contain both an operator stack and a operand stack.

CHAPTER III

PRE-PROCESSOR DESCRIPTION

The functional block diagram of the pre-processor shown in Figure 1 operates according to the State Transition Diagram of Figure 2. The pre-processor includes a control machine, an input buffer, a character hold register, a decoder, an encoder, two counters, an operator hold register, control tag lines to the system, and an unspecified control machine for interrupt handling and data transfer sequences to the evaluator. For simplicity, the complexity of the block diagram is kept to a minimum.

The purpose of the pre-processor is to receive and scan a given source string and provide stack formatted data, as well as execute signals, to the evaluator for analysis. For nested functions the results are placed back into the input stream to be scanned again. The following is an attempt to describe the necessary control signals and data flow of the pre-processor.

The signal "STEP" is generated to introduce the next character of the source input to the character hold register. This signal also transfers the information in the character register to the buffer. When the transfer is complete, both the buffer and the character register are decoded. The character register decode presents either a "control character" or an "any other character" signal to the control machine.

Depending upon which state the control machine is in and the character decoded, various output control signals will be generated to allow the pre-processor to function according to the State Diagram.

As the operator from the input TRAC statement is two characters, it is sequentially loaded into a holding register as it is scanned. The signals OPLOAD and SHR are for this purpose. The operator is then encoded and subsequently presented to the evaluator for insertion into an operator stack.

A flip-flop is set if the sequence of control character dictate that the function is an active function. This signal BAF is part of the encoding of the operator. Another flip-flop provides a similar function for a neutral function.

If an argument string is to be presented to the evaluator's operand stack the output of the character register is transmitted to the system by the signal STORE. As each character is transmitted to the system an argument counter is being incremented for future notification to the evaluator via an interrupt giving the argument length.

To provide the quote mode, in which input strings may be protected from evaluation, a parentheses counter is employed. This counter is generally incremented for the left parenthesis control character and decremented for the right parenthesis control character. The counter is decoded to provide an input to the control machine for a count or no count status.

Prior to an interrupt sequence, a flip-flop E_{DONE} is reset by the control machine. The signal $\overline{E}_D$ will keep the control machine in the interrupt state until this flip-flop is set by the evaluator.

The signal STORE# will gate to the system the encoded # control character. This is required for the third # control character. An example would be the sequence

```
#(PS,PART###567)
```

A similar signal STORE CO, exists for the concatenate operator. This is required for the sequence

```
#(PS,567#(AD,5,4))
```

The buffer register is decoded for the control character comma. The concatenate store signal is generated if the signal # in the character register is not preceded by a comma. This allows the evaluator stack processing to concatenate arguments.

The interrupt signals are possibly control tag signals to the evaluator. Again, the interrupt handling and data transfer sequence is not described in this paper but are assumed to be a plausible method for presenting data and execute signals to the evaluator.

With the functional block diagram and the desired input scanning algorithm a state sequencing diagram may be devised.

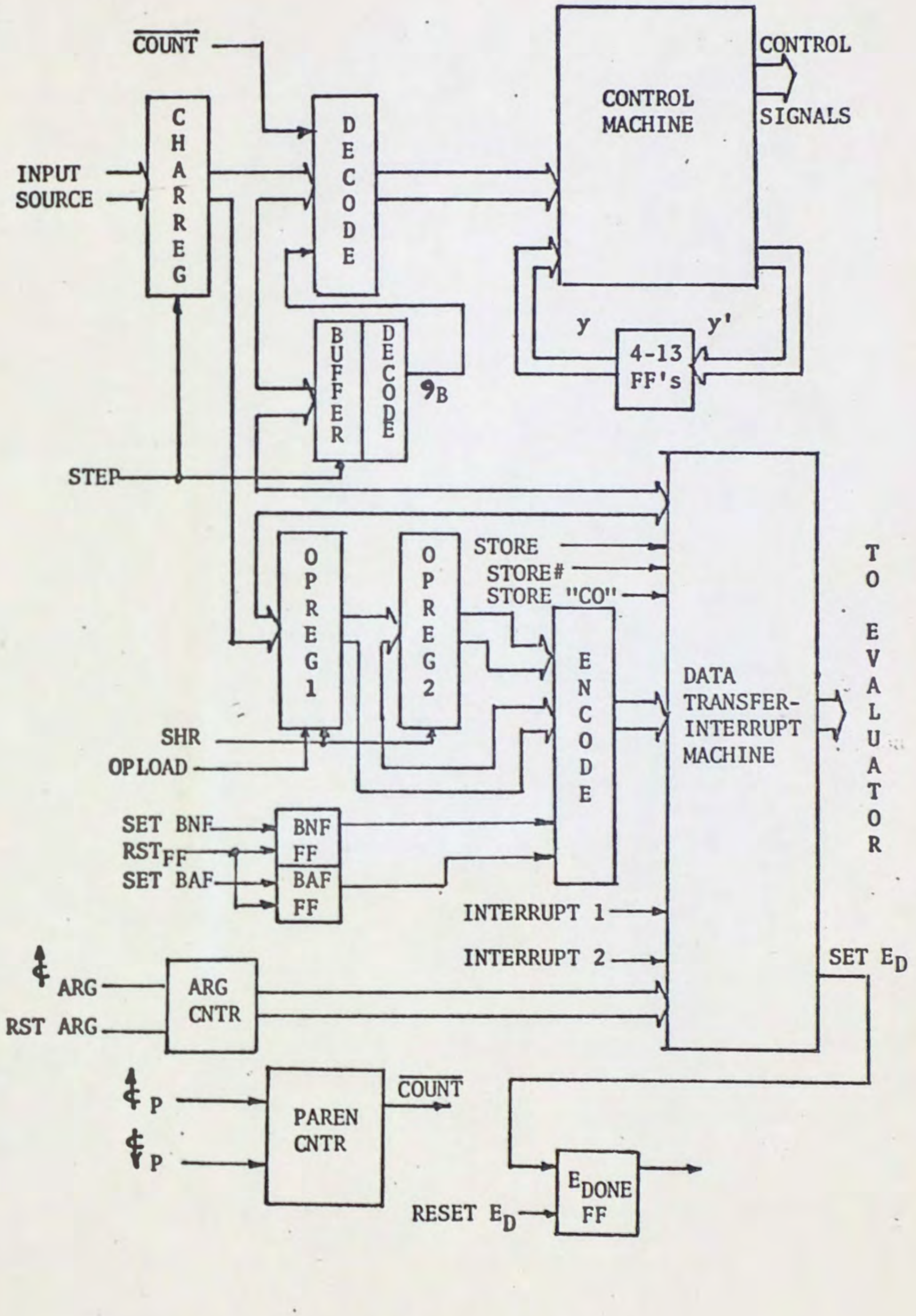


Figure 1 - Functional Block Diagram

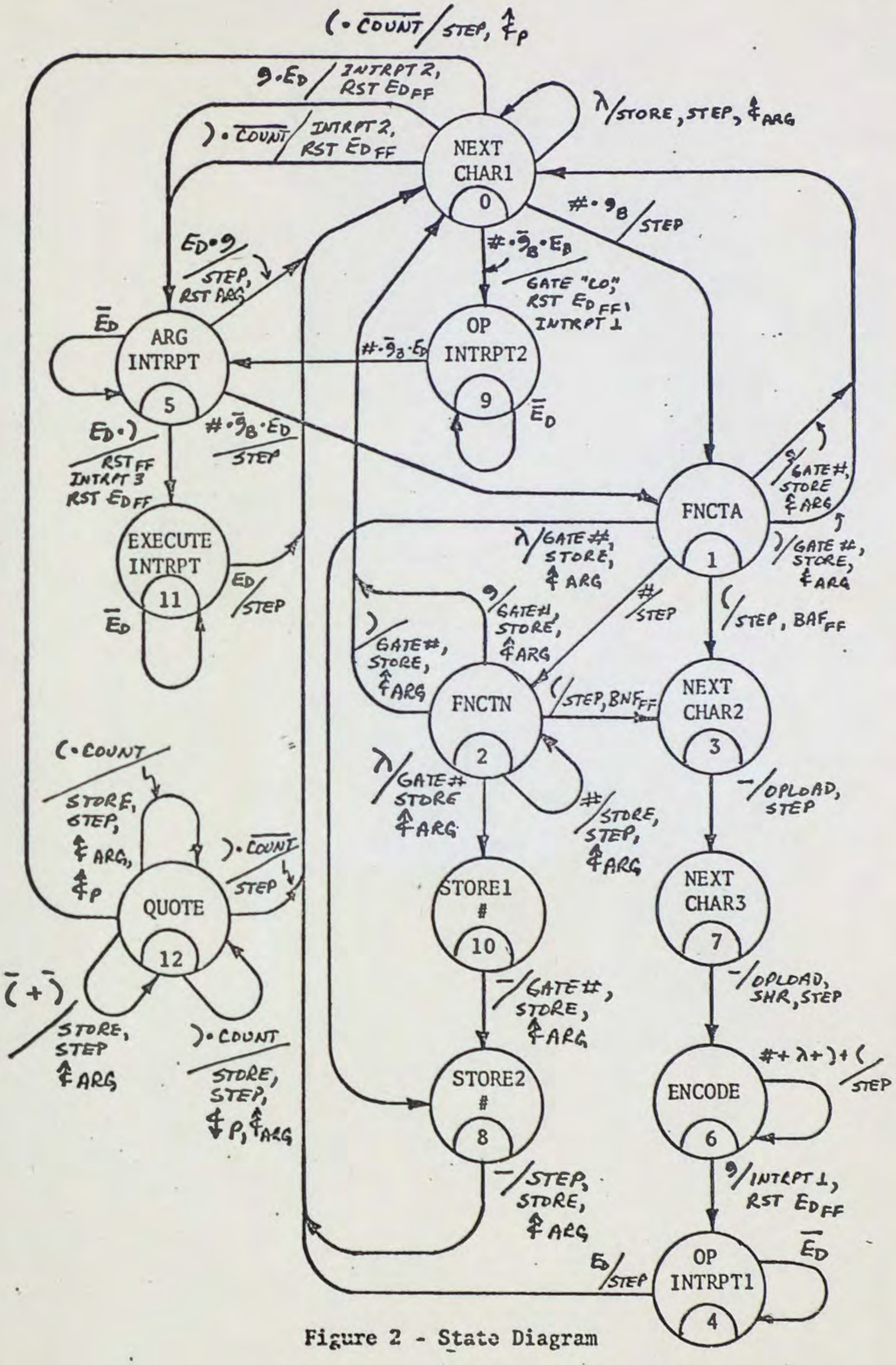


Figure 2 - State Diagram

CHAPTER IV

STATE DIAGRAM DESCRIPTION

The first requirement in the preliminary design of the pre-processor is to model the given TRAC scanning algorithm.

In general, the scanner-recognizer algorithm is as follows:

1. If the string of characters begins with "#(" these two characters are deleted, and the two characters that follow are stored in an operator register. A signal BAF (begin active function) is generated. An operator interrupt is generated to present the encoded operator to the evaluator. When the evaluator relenquishes control the pre-processor returns to "NEXT CHAR1" state.
2. If the string of characters begins with "##(" these three characters are deleted, and the two characters that follow are stored in an operator register. A signal BNF (begin neutral function) is generated. An operator interrupt is generated to present the encoded operator to the evaluator. After the evaluator has completed its task the pre-processor is notified and control returns to "NEXT CHAR1" state.
3. If the character "," is detected in the "NEXT CHAR1" state the machine generates an argument interrupt to the evaluator to indicate the end of an argument. The data transfer will give the length of the argument. When the interrupt is completed control returns to "NEXT CHAR1" state.

4. If the character ")" is detected in the "NEXT CHAR1" state and the parentheses counter is zero, two interrupts are given. First an argument interrupt to indicate the closing argument length and second, an execute interrupt to allow the evaluator to evaluate the specified function.
5. If the character "(" is detected in the "NEXT CHAR1" state and again the parentheses counter is zero, the control machine moves to the "QUOTE" state. While in the "QUOTE" state and the parentheses count is not zero, all other characters enclosed by parentheses are transferred to the evaluator as an argument string. Only when the parentheses counter is decrimented to zero does control return to the "TEST CHAR1" state.
6. All noncontrol characters "λ" are transferred to the evaluator by a command "STORE." As each character is transferred, an argument counter is incremented.
7. If the character "#" is detected and the preceding character is not a "," a concatenate signal and operator interrupt is generated. This is required when active or neutral functions are nested within an argument.

The object of the pre-processor is to delete all control characters to present the operators to the evaluator for storage in an OP stack, and to present all operands to the evaluator for insertion into an operand stack. The signals BAF and BNF are used to identify the type of function. Upon detection of a closing parenthesis the function is evaluated by the evaluator. The results may or may not be inserted back into the input string.

The State Diagram is shown in Figure 2. The State Diagram represents the state transitions and output signals required to implement the given scanning algorithm. Careful examination of the state transitions will show subtleties not easily described above. For further insight an example of the language is the string

#(EQ,#(CL,C),##(CL,#(CL,B)),(WOW),(##(CL,C)))

NAME OF FORM	VALUE
A	#(CL,B)
B	#(CL,C)
C	#(CL,AB)
AB	A

The pre-processor will cause EQ^A and CL^A to be stored in the operator stack, and C in the operand stack. The first closing parenthesis will result in the system evaluating the active function CL,C. As a result, #(CL,AB) will be placed in the input string. Again the pre-processor will cause CL^A and AB to be placed in the operator and operand stacks respectively. The closing parenthesis will cause the system to evaluate CL,AB resulting in A being placed in the operand stack. At this point EQ^A is in the operator stack and A in the operand stack.

The pre-processor will move onto the next argument. EQ^A will be pushed down in the operator stack and replaced by a neutral CL^N. The second CL^A will push EQ^A and CL^N down in the operator stack. In the operand stack, B will push A down. The first closing parenthesis will cause the system to evaluate CL^A,B resulting in #(CL,C) being placed into the input stream. Again CL^A and C are subsequently placed into the operator and operand stacks (with the previous CL^A,B being deleted).

This sequence is followed until the active function results in the value A being placed in the operand stack. At this time CL^N EQ^A

is in the operator stack and AA resides in the operand stack. The second closed parenthesis causes CL^N,A to be evaluated and placed into operand stack as #(CL,B). At this point EQ is in the operator stack and A followed by #(CL,B) is in the operand stack.

The pre-processor will move onto the next argument. As the argument is protected by parentheses the pre-processor will simply cause WOW to be placed in the operand stack, pushing down A and #(CL,B).

The pre-processor moves to the last argument and places ##(CL,C) into the operand stack as it is protected from evaluation by parentheses. At this point in the processing of the language statement, the operator stack contains EQ^A and the operand stack has A followed by #(CL,B) followed by WOW followed by ##(CL,C). When the pre-processor recognizes the last closing parenthesis the system evaluates the contents of the two stacks. A and #(CL,B) are examined in the operand stack and compared. As they are not equal, ##(CL,C) is selected and evaluated. The result #(CL,AB) is placed in the operand stack. Control is then passed back to the pre-processor and scanning of the source string is continued.

CHAPTER V

STATE MINIMIZATION

It is often desirable, from economic and other viewpoints, to eliminate the duplication of equivalent states.

Two states of a machine are said to be equivalent if it is impossible to distinguish between them by submitting input sequences and observing the output sequences generated by the machine.

An intuitive approach may locate the equivalent state by examination of random pairs of states. However, an algorithm exists that is more efficient. It is as follows: (5)

1. Partition the set of all states into sets such that all members of a set have identical output rows in the STT.

2. Under each state, for each input symbol record the number of the set of which the following state is a member.

3. Divide existing states sets so that all members of a new set possess the same subscripts. When no sets are formed the algorithm terminates. The states within the set groupings may be considered as equivalent.

In the control machine for the pre-processor and its associated State Transition Table (STT) of Figure 3a and 3b, the first and second step in the above algorithm will result in no two states being equivalent. By inspection, it is seen that no two states have the same output sequences and therefore there are no equivalent states represented in the State Diagram.

CHAPTER VI

IMPLEMENTATION OF THE CONTROL MACHINE

The State Diagram of Figure 2 and the State Transition Table of Figure 3a and 3b list the conditions under which a transition from one state to another is required. It is seen that a 13-state machine must be synthesized. Many input signals are involved and many output signals must be generated. A formal design procedure for finite-state machines using Karnaugh maps in their general sense is not very effective for large problems with a large number of input variables.

Implementation is required of a 13-state machine with 9 input variables and 17 output signals. At a minimum, 4 flip-flops are required; however, at the other extreme, 13 flip-flops could be used, one for each state. The latter state assignment greatly simplifies design effort.

Two common approaches will be discussed in the following sections.

One-hot Flip-flop per State

For simplicity of design, the one-hot flip-flop per state control machine is presented.

The information contained in the State Diagram allows us immediately to draw the logic of the 13 flip-flop machine. This set of flip-flops is in essence a "state sequencing" register in as much as it resembles a serial shift register. With this approach one and only one flip-flop is set at all times. Each flip-flop thus acts as the source

PRESENT STATE	NEXT STATE								
	$\#\cdot 9_B$	$\#\cdot\overline{9_B}\cdot E_D$	$9\cdot E_D$	λ	$(\cdot\overline{COUNT}$	$(\cdot COUNT$	$)\cdot\overline{COUNT}\cdot E_D$	$)\cdot COUNT$	$\overline{E_D}$
	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9
0	1	9	5	0	12	d	5	d	d
1	2	2	0	8	3	3	0	0	d
2	2	2	0	10	3	3	0	0	d
3	7	7	7	7	7	7	7	7	d
4	d	d	0	d	d	d	d	d	4
5	d	1	0	d	d	d	11	d	5
6	6	6	4	6	6	6	6	6	d
7	6	6	6	6	6	6	6	6	d
8	0	0	0	0	0	0	0	0	d
9	d	5	d	d	d	d	d	d	9
10	8	8	8	8	8	8	8	8	d
11	d	d	d	d	d	d	0	d	11
12	12	12	12	12	12	12	0	12	d

d = don't care - invalid input

Figure 3a - State Transition Table

PRESENT STATE	OUTPUT*								
	$\#\cdot 9_B$	$\#\cdot\bar{9}_B\cdot E_D$	$9\cdot E_D$	$\wedge$	$(\cdot\overline{COUNT}$	$(\cdot COUNT$	$)\cdot\overline{COUNT}\cdot E_D$	$)\cdot COUNT$	$\bar{E}_D$
	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9
0	A	LO Q	MQ	AB C	AE	d	MQ	d	d
1	A	A	BC P	BC P	AG	AG	BC P	BC P	d
2	AB C	AB C	BC P	BC P	AH	AH	BC P	BC P	d
3	AJ	AJ	AJ	AJ	AJ	AJ	AJ	AJ	d
4	d	d	A	d	d	d	d	d	∅
5	d	A	AD	d	d	d	IN Q	d	∅
6	A	A	LQ	A	A	A	A	A	d
7	AJ K	AJ K	AJ K	AJ K	AJ K	AJ K	AJ K	AJ K	d
8	AB C	AB C	AB C	AB C	AB C	AB C	AB C	AB C	d
9	d	∅	d	d	d	d	d	d	∅
10	AB P	AB P	AB P	AB P	AB P	AB P	AB P	AB P	d
11	d	d	d	d	d	d	A	d	∅
12	AB C	AB C	AB C	AB C	AB C	AB CE	A	AB CF	d

*See Table 1 for output designations

Figure 3b - State Transition Output Table

TABLE 1 - OUTPUT DESIGNATIONS

A = STEP	Step Buffer and Charreg
B = STORE	Store Tag
C = $\overset{\uparrow}{\text{¢}}$ ARG	Upcount argument counter
D = RST ARG	Reset argument counter
E = $\overset{\uparrow}{\text{¢}}$ P	Upcount parentheses counter
F = $\underset{\downarrow}{\text{¢}}$ P	Downcount parentheses counter
G = BAF_{FF}	Set Begin Active Function FF
H = BNF_{FF}	Set Begin Neutral Function FF
I = RST_{FF}	Reset BAF_{FF} and BNF_{FF}
J = OPLOAD	Load Opreg 1
K = SHR	Shift Opreg 1 into Opreg 2
L = INTRPT 1	Interrupt 1 Tag
M = INTRPT 2	Interrupt 2 Tag
N = INTRPT 3	Interrupt 3 Tag
O = GATE CO	Gate concatenate operator
P = GATE #	Gate # character
Q = RST $E_{D_{FF}}$	Reset $E_{DONE_{FF}}$

for the state terminals NEXT CHAR1, FNCTA, etc., which we have assumed to exist.

With a one-hot code the ith flip-flop is set when the machine is in the ith state, all other flip-flops being reset. Thus the states of the control machine might be encoded

STATE	Y_1	Y_2	Y_3	Y_4	Y_5	.	.	:	Y_{13}
0	1	0	0	0	0	.	.	.	0
1	0	1	0	0	0	.	.	.	0
2	0	0	1	0	0	.	.	.	0
3	0	0	0	1	0	.	.	.	0
•	•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•	•
12	0	0	0	0	0	.	.	.	1

This assignment is desirable if the design utilizes small scale integration (20 circuits per chip) and the emphasis is on maintenance of the machine. We can identify the state of the machine by determining which flip-flop is set. Also, another advantage exists. Suppose the following network were available at the module level.

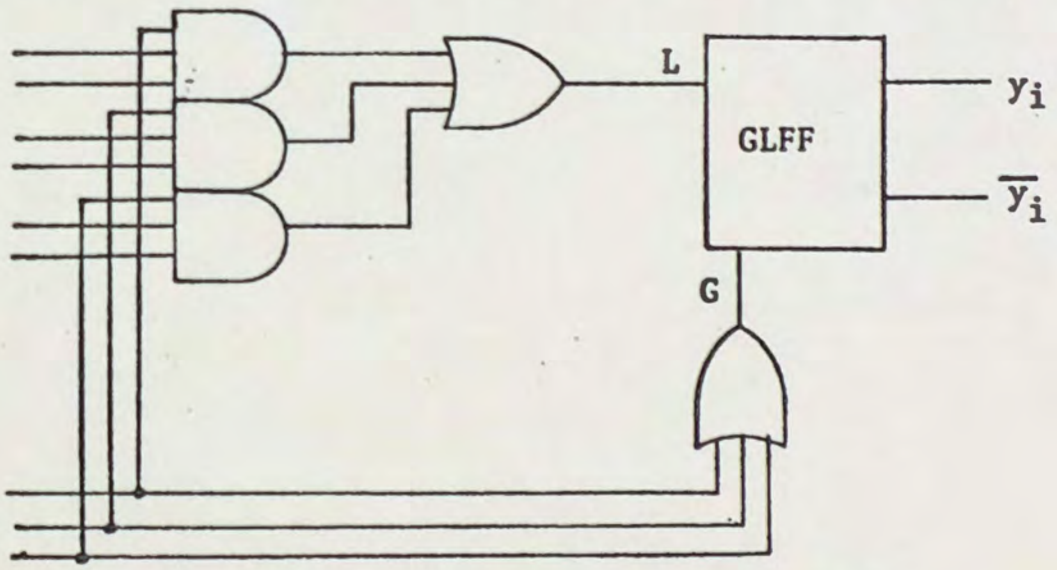


Figure 4 - One-hot per State Module

This module utilizes the Gated Latch flip-flop (GLFF) for the memory element. The GLFF assumes a state equal to the value of the L input signal when the gate input G is present. As long as G = 0, the flip-flop does not change state.

This flip-flop is useful when information must be transferred from one memory element to another as in a state sequencing register. This transfer of information can be accomplished by the JK flip-flop with additional logic. Such a Gated Latch flip-flop could be the following circuit.

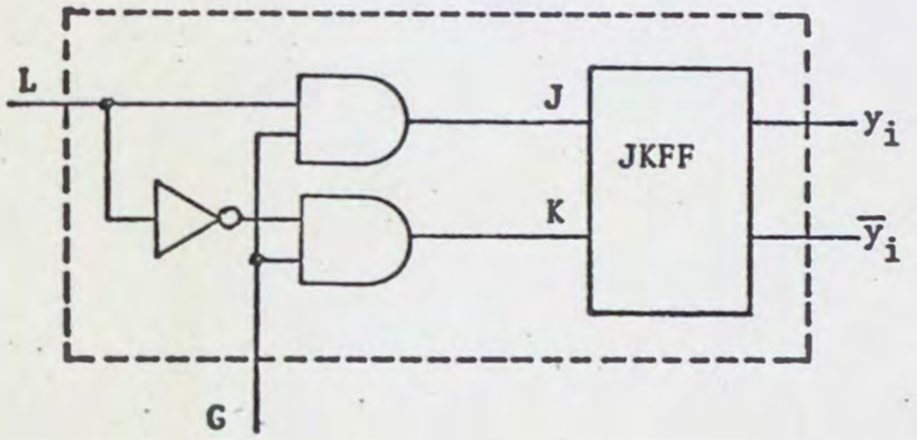


Figure 5 - GLFF With JKFF Implementation

With this module of Figure 4 we can synthesize directly from the State Diagram. One module would be required for each state. For each transition into a state, we connect the input signals to terminals of an AND gate to form the latch signal. The present state signal is then connected to one of the OR gate terminals to form the gate signal. The clock is, of course, assumed.

For example, choosing the transition from FNCTA to NEXT CHAR1, Figure 6 represents the logic required.

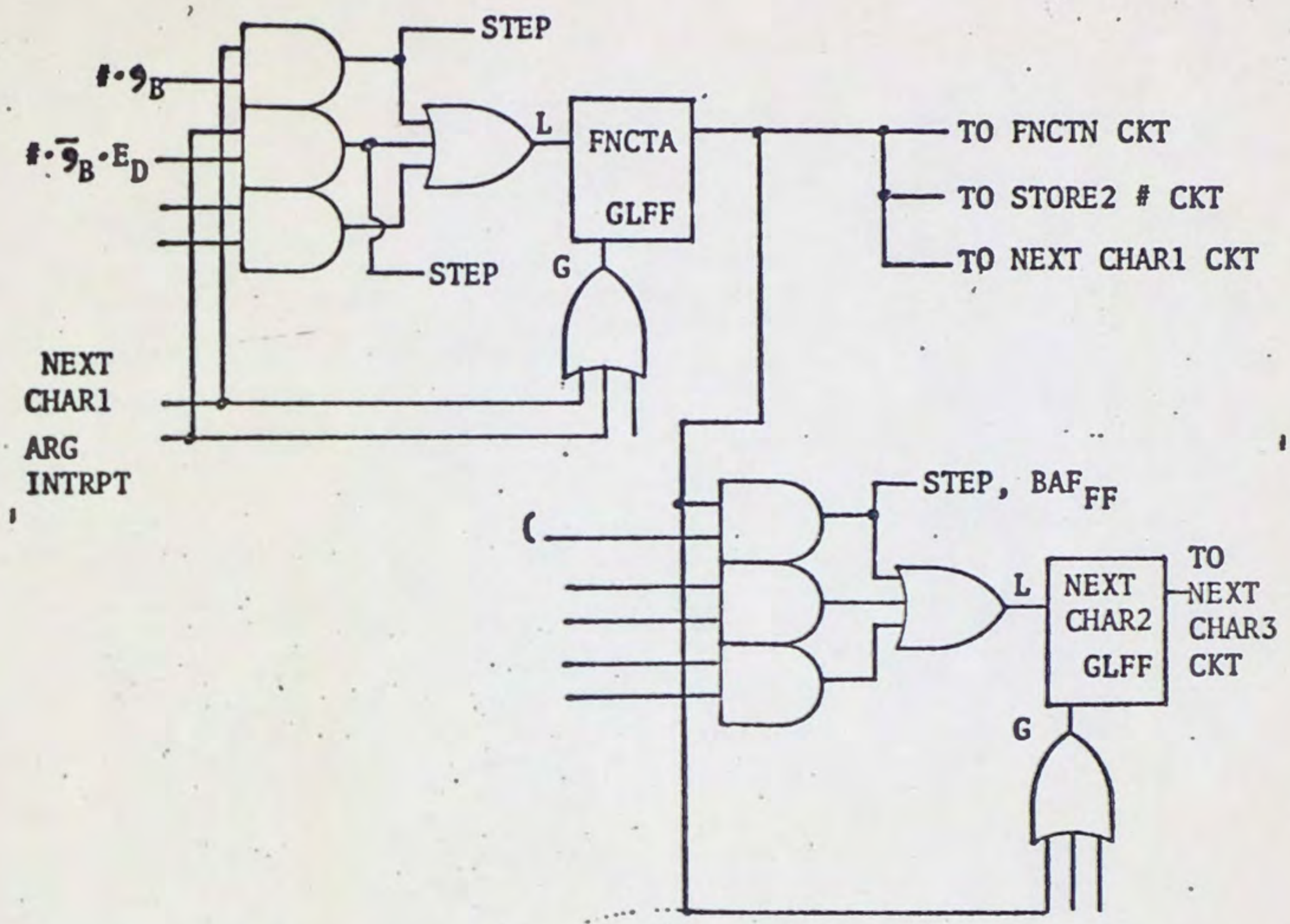


Figure 6 - One-hot State Sequence Example

The above module logic represents the transition shown in Figure 7.

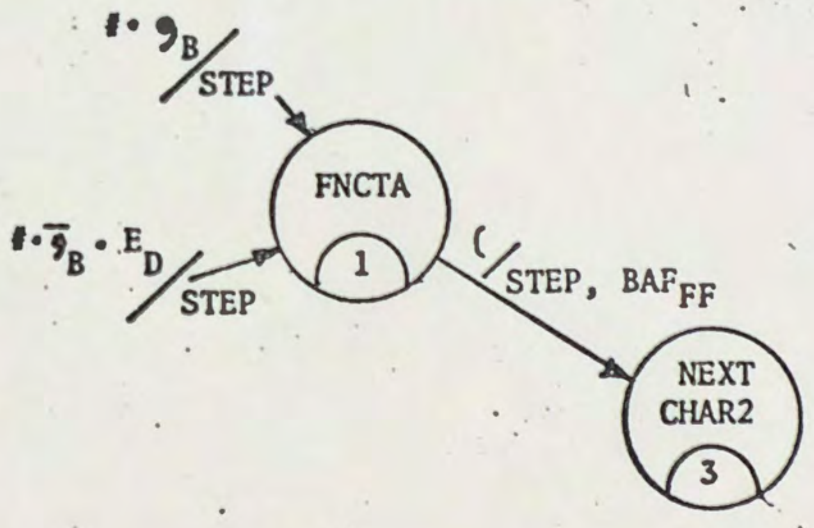


Figure 7 - Transition from FNCTA to NEXT CHAR2

For the complete control machine the above process must be repeated for each of the remaining twelve state transitions. The outputs generated by the control machine will also have to be generated in a similar fashion.

Minimum Flip-flop Control Machine

The other extreme in the design of the control machine for the pre-processor is the minimum state variable machine. Here, only the minimum number of flip-flops required will be used.

A machine with α states can only be realized with β or more binary flip-flops where α and β satisfy.

$$\alpha \leq 2^{\beta}$$

For the minimum flip-flop machine the equality of the above expression is desired. Thus for α of 16 states, β must be 4. However, in the pre-processor control machine only 13 states are required, but since β must be an integer, 4 flip-flops must be used.

Therefore, a second approach is to use 4 flip-flops and formally design the 13 state control machine. This approach will lead to a most desirable control unit. However, the size of the State Transition Table and State Diagram prohibits the use of standard Karnaugh map techniques.

One alternative, would be to use the Quine-McCluskey algorithm. However, the State Transition Table shows that this would include manipulation of a thirteen variable function[2] such as:

[2]A redesign of the decode block of Figure 1 could reduce the size of the function to 8 variables, four state variables and 4 input variables.

$$F = Y_1\ Y_2\ Y_3\ Y_4\ X_1\ X_2\ X_3\ X_4\ X_5\ X_6\ X_7\ X_8\ X_9$$

where the input $X_1\ X_2\ X_3\ \ldots\ X_9$ is a one-hot code and $Y_1\ Y_2\ Y_3\ Y_4$ represents the four state variables.

In general, as the number of variables increases so does the labor. Minimizing an eleven variable function by hand is not considered a small task. The addition of don't cares into the array further complicates the already large amount of bookkeeping required.

Computer assistance is available in performing array manipulations. There are a number of programs that utilize the Quine-McCluskey tabulation approach. Such programs are available using FORTRAN V subroutines for the UNIVAC 1108 digital computer. (5)

States of the four flip-flops to be used in the control machine have been assigned as shown in the State Diagram. The code chosen is important as it reflects the required amount of combinatorial logic. However, no simple method is available for determining whether or not the assigned states will lead to the most economical combinatorial logic.

State assignment may be suggested by the problem or State Diagram, or it may be completely arbitrary. The following rules of thumb may enable the design of the combinatorial driving equations at reduced cost. (5)

1. Use the minimum number of states.
2. Assign adjacent code words to a state and the state that follows it.
3. If two present states have the same next state, assign those present states adjacent code words.

For the given State Diagram of Figure 2 these rules were impossible to apply in all cases. Rule 2 was applied in this case. However, to the extent they were satisfied, these rules tend to minimize the state-transition and flip-flop input equations of the control machine.

In the particular case that exists here, that is, the input variables to the control machine are of a one-hot code. This allows use of piecewise minimization using the State Transition Table in conjunction with Karnaugh map techniques. To do this, Marcus' (6) procedure for deriving flip-flop driving equations from a State Transition Table may be used. This procedure is not complex but does require knowledge of the logic of each type of flip-flop.

Exactly what flip-flop input signals are appropriate depends on the type of flip-flop we expect to use. The following Table 2 shows the values that flip-flop input variables must take to accomplish state transitions.

TABLE 2 - INPUT VARIABLE VALUES

MEANING	TRANSITION SYMBOL	DESIRED TRANSITION $y \rightarrow y'$	RSFF S R	TFF T	JKFF J K	GLFF G L
Retain	"0"	0 0	0 d	0	0 d	0 d d 0
Set	"S"	0 1	1 0	1	1 d	1 1
Reset	"R"	1 0	0 1	1	d 1	1 0
Retain	"1"	1 1	d 0	0	d 0	0 d d 1

The transition symbols defined as follows:

"S" means "must take setting action."
"R" means "must take resetting action."
"1" denotes FF remains set.
"0" denotes FF remains reset.

The above table is useful in implementing state transition equations. For example, if JKFF implementation were desired, for every transition of y to y', the proper value would be placed into a value map for that particular flip-flop. For the J and K lines, the values 0, 1, or d would appear in the map according to the values listed under the J and K lines in the above table. From this, map minimization could be used.

We can simplify the above technique by using a state transition map, transition symbols, and flip-flop equations in terms of the transition symbols. This state transition map may be called an "action" map as it describes in symbol form the transition required.

In order to use this action map, equations in terms of action symbols must be described. The following are such equations:

For a RSFF, $S = \Sigma_1 S + \Sigma_d 1,d$

$R = \Sigma_1 R + \Sigma_d 0,d$

For a TFF, $T = \Sigma_1 S,R + \Sigma_d d$

For a JKFF, $J = \Sigma_1 S + \Sigma_d R,1,d$

$K = \Sigma_1 R + \Sigma_d S,0,d$

For a GLFF, $G = \Sigma_1 S,R + \Sigma_d d,1^*,0^{**}$

$L = \Sigma_1 S + \Sigma_d d,1^*,0^{**}$

The summation sign indicates that all cells having the following symbols are to be given the value found at the base of the summation sign. For example, in the JKFF, for the J line, the S cells constitute the ON-array (1's); the R, the 1, and the d cells constitute the "don't care" array and all other cells are the OFF-array.

With the GLFF special consideration of the don't cares noted by the asterisks is required. If in minimizing G we assign one of these

1* or 0** don't care cells the value of 1, we must assign 1 and 0 respectively for L. If in minimizing L we assign one of these 1* or 0** don't care cells to the ON-array of L, G must be d or 0 respectively.

The advantage of writing the "action" STT is that it can be a base for implementation of any flip-flop desired. All one has to do is use the associated flip-flop input equations given earlier to insert into a value map. The action map is an interim tool to help avoid careless mistakes in determining the value map.

For implementation of the pre-processor control machine, the following steps will be taken. First, the State Transition Table of Figure 3a and 3b will be converted into action tables using the appropriate transition symbols. Second, the flip-flop input equations will be applied to generate value maps. Third, from the value maps, minimization techniques will be applied to generate the flip-flop input equations.

Implementation of the State Transition Table of Figure 3a and 3b may, at first, appear to be an impossible task. However, when broken down into its lesser component parts, piecewise analysis is possible. As the input variables are in a one-hot code format, minimization of the input variables is not required. Therefore, the piecewise approach will be the optimized design provided the same covers are used in the mapping techniques wherever possible.

The following piecewise approach is presented. The JK flip-flop will be the memory element used as this is the most popular in the industry. The J and K lines for the Y_1 flip-flop will be developed.

First the action table for this flip-flop must be constructed. Figure 8 represents this action table for Y_1. This table was con-

PRESENT STATE	SYMBOL VALUE								
	$\#\cdot 9_B$	$\#\cdot\overline{9}_B\cdot E_D$	$9\cdot E_D$	λ	$(\cdot\overline{COUNT}$	$(\cdot COUNT$	$)\cdot\overline{COUNT}\cdot E_D$	$)\cdot COUNT$	$\overline{E}_D$
	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9
0	0	S	0	0	S	d	0	d	d
1	0	0	0	S	0	0	0	0	d
2	0	0	0	S	0	0	0	0	d
3	0	0	0	0	0	0	0	0	d
4	d	d	0	d	d	d	d	d	0
5	d	d	0	d	d	d	S	d	0
6	0	0	0	0	0	0	0	0	d
7	0	0	0	0	0	0	0	0	d
8	R	R	R	R	R	R	R	R	d
9	d	1	d	d	d	d	d	d	1
10	1	1	1	1	1	1	1	1	d
11	d	d	d	d	d	d	R	d	1
12	1	1	1	1	1	1	R	1	d

Figure 8 - Y_1 Action Table

structed by determining all the transitions taking place on the State Transition Table of Figure 3a and placing the applicable symbol, from Table I, in the table for each input column.

Figure 9 represents the value table for the input J_1 to the Y_1 JK flip-flop. This table was constructed by using the J line input equation

$$J = \Sigma_1 S + \Sigma_d R,1,d$$

Since minimization will be done on a piecewise basis, the following figure represents the value map for only the $\#\cdot\overline{9}_B\cdot E_D$ input variable.

Y_1' Y_1Y_2 \ Y_3Y_4	00	01	11	10
00	1	0	0	0
01	d	d	0	0
11	d	d	d	d
10	d	d	d	d

It will be noted that states 13, 14, and 15 are represented as don't cares.. These states will never occur and therefore are don't cares and will apply for all subsequent value maps.

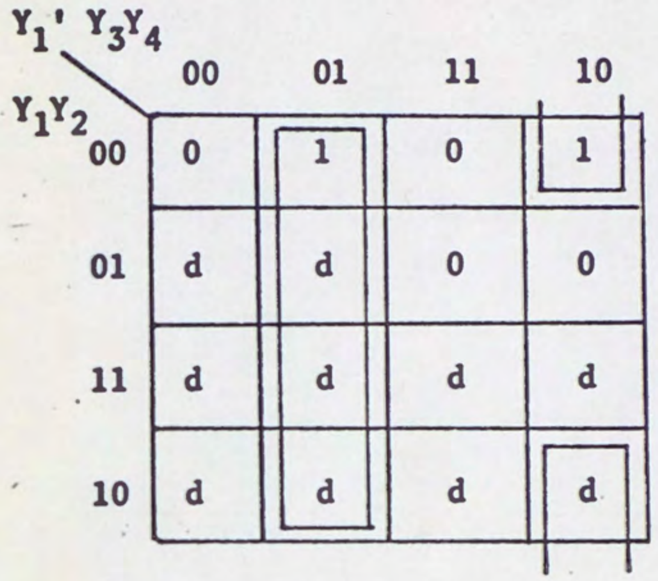
It can be seen from the above value map that the $\#\cdot\overline{9}_B\cdot E_D$ input contribution is

$$J_1 = [\#\cdot\overline{9}_B\cdot E_D](\overline{Y}_3\overline{Y}_4) + \cdots$$

The next input variable of consequence is the λ input variable. The following map represents its contribution

PRESENT STATE	VALUE								
	$\#\cdot 9_8$	$\#\cdot\bar{9}_8\cdot E_D$	$9\cdot E_D$	λ	$(\cdot\overline{COUNT}$	$(\cdot COUNT$	$)\cdot\overline{COUNT}\cdot E_D$	$)\cdot COUNT$	$\bar{E}_D$
	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9
0	0	1	0	0	1	d	0	d	d
1	0	0	0	1	0	0	0	0	d
2	0	0	0	1	0	0	0	0	d
3	0	0	0	0	0	0	0	0	d
4	d	d	0	d	d	d	d	d	0
5	d	d	0	d	d	d	1	d	0
6	0	0	0	0	0	0	0	0	d
7	0	0	0	0	0	0	0	0	d
8	d	d	d	d	d	d	d	d	d
9	d	d	d	d	d	d	d	d	d
10	d	d	d	d	d	d	d	d	d
11	d	d	d	d	d	d	d	d	d
12	d	d	d	d	d	d	d	d	d

Figure 9 - J_1 Value Table

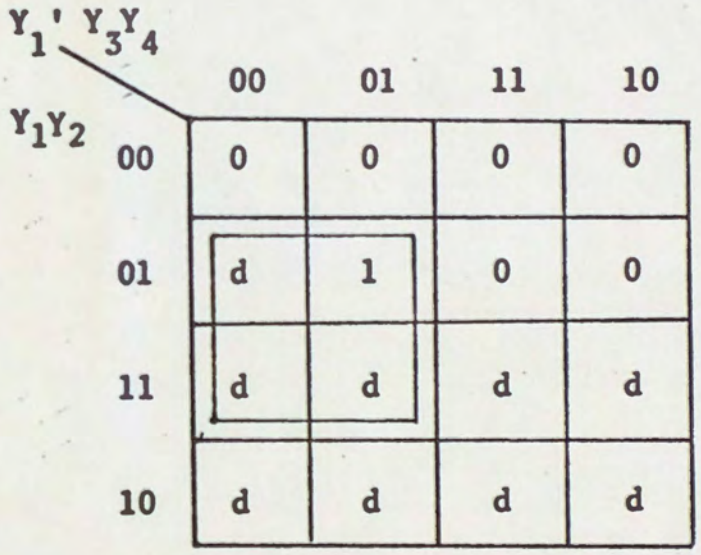


Y_1' Y_1Y_2 \ Y_3Y_4	00	01	11	10
00	0	1	0	1
01	d	d	0	0
11	d	d	d	d
10	d	d	d	d

Minimization gives

$$J_1 = [\lambda](\overline{Y}_3Y_4 + \overline{Y}_2Y_3\overline{Y}_4) + \cdots$$

Working with the $)\cdot\overline{COUNT}\cdot E_D$ variable

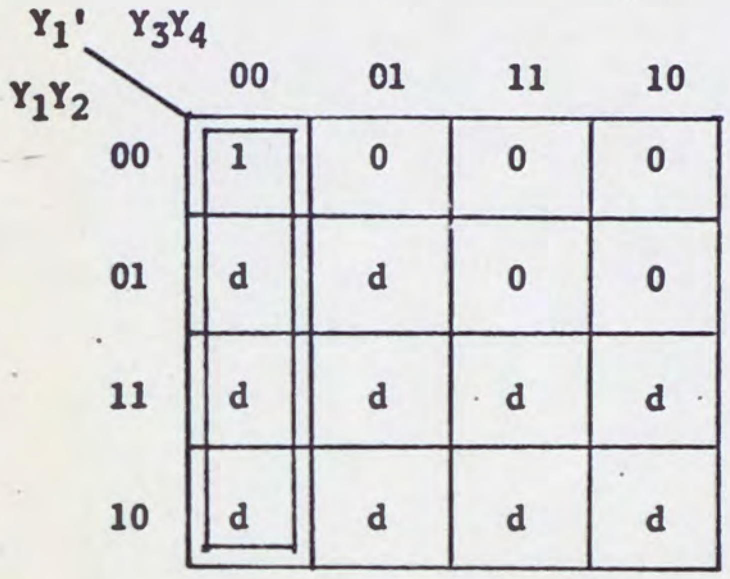


Y_1' Y_1Y_2 \ Y_3Y_4	00	01	11	10
00	0	0	0	0
01	d	1	0	0
11	d	d	d	d
10	d	d	d	d

Minimization gives

$$J_1 = [)\cdot\overline{COUNT}\cdot E_D](Y_2\overline{Y}_3) + \cdots$$

The last input for the J_1 driving equation analysis is the $(\cdot\overline{\text{COUNT}}$ input variable. Again the value map for this input is shown.



Minimization gives

$$J_1 = [(\cdot\overline{\text{COUNT}}](\overline{Y}_3\overline{Y}_4) + \cdots$$

The other input variables of Figure do not contain any "1's" in the input column. Therefore, these inputs are not represented in the total J_1 driving equations.

Summing the input contributions for J_1 gives

$$J_1 = [\#\cdot\vartheta_B\cdot E_D + (\cdot\overline{\text{COUNT}}](\overline{Y}_3\overline{Y}_4) + [\lambda](\overline{Y}_3Y_4 + \overline{Y}_2Y_3\overline{Y}_4) + [)\cdot\overline{\text{COUNT}}\ E_D](Y_2\overline{Y}_3)$$

The K_1 driving equations may be found by using the K_1 value table of Figure 10 . This table was constructed by using the K line input equation

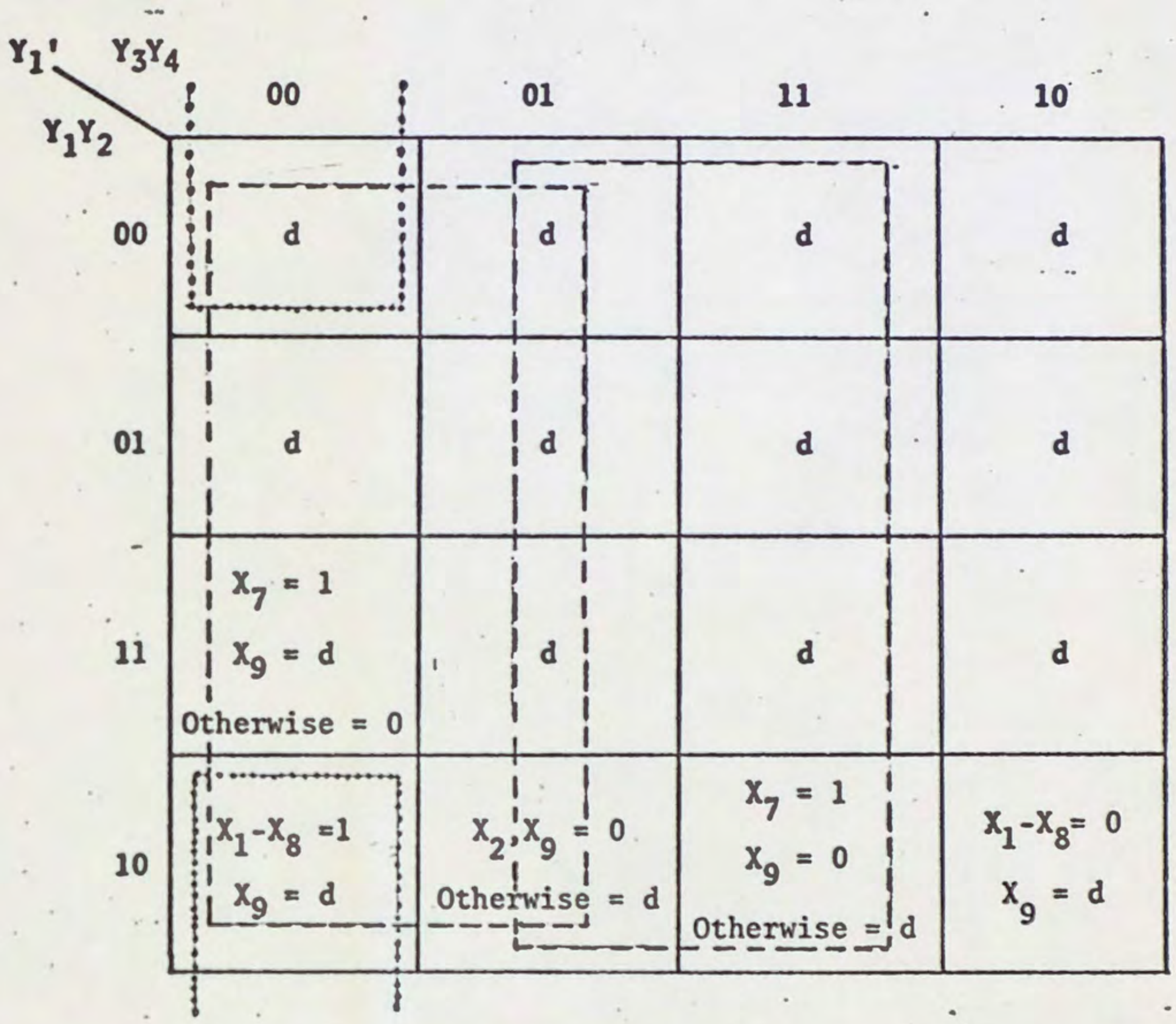
$$K = \Sigma_1 R + \Sigma_d S,0,d$$

Again minimization for the K_1 line will be done on a piecewise basis. However, instead of individual maps for each input, a composite value map will be shown. This composite map will give a better indication of common covers that are available for better minimization.

PRESENT STATE	VALUE								
	$\# \cdot 9_B$	$\# \cdot \overline{9_B} \cdot E_D$	$9 \cdot E_D$	$\wedge$	$(\cdot \overline{COUNT}$	$(\cdot COUNT$	$) \cdot \overline{COUNT} \cdot E_D$	$) \cdot COUNT$	$\overline{E_D}$
	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9
0	d	d	d	d	d	d	d	d	d
1	d	d	d	d	d	d	d	d	d
2	d	d	d	d	d	d	d	d	d
3	d	d	d	d	d	d	d	d	d
4	d	d	d	d	d	d	d	d	d
5	d	d	d	d	d	d	d	d	d
6	d	d	d	d	d	d	d	d	d
7	d	d	d	d	d	d	d	d	d
8	1	1	1	1	1	1	1	1	d
9	d	0	d	d	d	d	d	d	0
10	0	0	0	0	0	0	0	0	d
11	d	d	d	d	d	d	1	d	0
12	0	0	0	0	0	0	1	0	d

Figure 10 - K_1 Value Table

Using the input nomenclature of X_1-X_9 for neatness, the composite value map is then



Y_1' Y_3Y_4 / Y_1Y_2	00	01	11	10
00	d	d	d	d
01	d	d	d	d
11	$X_7 = 1$ $X_9 = d$ Otherwise = 0	d	d	d
10	X_1-X_8 =1 $X_9 = d$	$X_2, X_9 = 0$ Otherwise = d	$X_7 = 1$ $X_9 = 0$ Otherwise = d	X_1-X_8= 0 $X_9 = d$

For the inputs X_1, X_2, X_3, X_4, X_5, X_6 and X_8, the cover represented by the dotted line will give

$$K_1 = (\overline{Y}_2\overline{Y}_3\overline{Y}_4) + \cdots$$

For the X_7 input, the covers shown by the dashed lines will result in the minimization

$$K_1 = [X_7](\overline{Y}_3 + Y_4) + \cdots$$

As the X_9 input has no 1's represented in its column the K_1 value table, it has no contribution to the K_1 input equation.

With all the input variables taken into account, the total K_1 input equation may be given. For completeness the J_1 equation will be repeated.

$$\begin{aligned} J_1 = {} & [\# \cdot \overline{9}_B \cdot E_D + (\cdot \overline{\text{COUNT}}](\overline{Y}_3\overline{Y}_4) + [\lambda](\overline{Y}_3 Y_4 + \overline{Y}_2 Y_3 \overline{Y}_4) \\ & + [) \cdot \overline{\text{COUNT}}\ E_D](Y_2\overline{Y}_3) \\ K_1 = {} & (\overline{Y}_2\overline{Y}_3\overline{Y}_4) + [) \cdot \overline{\text{COUNT}}\ E_D](\overline{Y}_3 + Y_4) \end{aligned}$$

So far, only the driving equations for J_1 and K_1 of the Y_1 JK flip-flop have been developed. By a similar procedure the input driving equations may be generated for the remaining three JK flip-flops.

For the complete design of the control machine, outputs must be generated. The output signals required are represented both on the State Diagram of Figure 2 and the State Transition Table of Figure 3b. Although output generation will not be presented, similar techniques as discussed in this report may be used.

Since two extreme variations of the control machine have been presented, a comparison between the two will be presented.

CHAPTER VII

COMPARISON OF THE TWO CONTROL MACHINES

Two control machines for the pre-processor have been presented. The first machine required the maximum number of flip-flops. The second was developed using the minimum number of flip-flops.

Which control machine is the most economical depends not only upon the number of flip-flops used but also on the amount of combinatorial logic required for its implementation.

If the one-hot code control machine were implemented using the module containing the Gated Latch flip-flop of Figure 4, a comparison could be made with the minimum state variable machine. The latter machine being implemented in JK flip-flops. As the GL flip-flop contains a JK flip-flop, as shown in Figure 5, this comparison is valid.

For the one-hot code machine and its associated module implementation the number of gates required may be found using the State Diagram of Figure 2. For every input that results in a state transition an AND gate is required. If two or more AND's are required, this necessitates the use of two OR circuits. Each GLFF in the module requires two AND gates and an inverter to gate the JK flip-flop.

Investigation of the State Diagram using the above method will result in approximately 79 gates being required to sequence the one-hot code control machine.

The minimum state variable machine gating may be estimated using the J_1 and K_1 input equations previously derived. Assuming that the input equations for the Y_1 flip-flop are representative of the remaining three flip-flops, the following estimate is obtained.

For the J_1 input equation, eight gates are required. For the K_1 input equation four gates are required. The total number of gates for the Y_1 flip-flop is twelve. As four flip-flops are required for the 13 state machine the total number of gates for the machine will be approximately 48.

At this point, since both control machine implementations use JK flip-flops, it appears that the minimum state machine is more economical. However, the latter estimate assumed that the Y_1 flip-flop input logic was representative of all four flip-flops. Therefore, the apparent advantage of the minimum state variable machine may be reduced. But the fact that the one-hot machine requires an additional nine flip-flops for implementation does tend to increase the appeal of the minimum state variable machine.

CONCLUSION

The preceding chapters illustrate some of the design techniques available to translate the given scanning algorithm into a hardwired pre-processor. Two extremes were presented for implementation of the state control machine. The one-hot code machine offers not only ease of maintenance and testability, but that it has the potential of sharing a common module part number. On the other hand, the minimum state variable machine offers a more minimal design. Either approach will require an assessment of the requirements that may exist. Also, the design of the supportive hardware, such as registers, counters, etc., may be accomplished using similar techniques as those presented.

The implication of this investigation is that once support software has been specified, such as the scanning algorithm in this case, it may be translated from a flow chart to a State Diagram. In turn, this State Diagram may be translated to logic equations and their associated logic diagrams. In other words, with the use of Large Scale Integration (LSI), system support software can ultimately be converted to hardware.

In respect to "hardwired pre-processing," individual pre-processors can be used, provided the specified intermediate output required for the evaluator is satisfied.

An example is University of California's Eclectic computer which uses a FORTRAN-like source language. (2) The difficulty in pre-processing this type of language is much more complex, hence the pre-processor can be performed by a micro-coded processor or even another mini-class computer.

More in line with the concept of a hardwired pre-processor has been proposed for a FORTRAN machine by Bashkow, et al. (4) Their approach is based on a recognition that once the allowable syntax and associated sematics of language statements have been firmly specified a hardware interpreter, or machine, seems feasible.

The processor proposed in this paper has the pre-processor and evaluator both physically and conceptually separate. Therefore, with this approach and the advances being made in LSI technology, it is not unlikely that pre-processor "chips" may serve a variety of users without impacting specialized system software.

LIST OF REFERENCES

1. Calvin N. Mooers, "TRAC, A Procedure-Describing Language for the Reactive Typewriter," Communications of the ACM, Vol. 9, No. 3, March, 1966, pp. 215-19.

2. R. Cutts, et al, "An Eclectic Information Processing System," AFIPS Conference Proc., Fall Joint Computer Conference Vol. 41, Part 1, 1972, pp. 473-77.

3. P. Wegner, Programming Languages, Information Structures, and Machine Organization. New York: McGraw-Hill, 1968, pp. 52-56.

4. G. C. Bell and A. Newall, Computer Structures: Readings and Examples. New York: McGraw-Hill, 1971, pp. 267-73.

5. D. L. Dietmeyer, Logic Design of Digital Systems. Boston: Allyn and Bacon, Inc., 1971, pp. 429-550.

6. M. P. Marcus, Switching Circuits for Engineers. New Jersey: Prentice-Hall, 1967, pp. 162-76.