



Scholars' Mine

Masters Theses

Student Theses and Dissertations

1968

The computer-aided generation of flow-tables for asynchronous sequential circuits

Ronald Lee Altman

Follow this and additional works at: https://scholarsmine.mst.edu/masters_theses

 Part of the [Electrical and Computer Engineering Commons](#)

Department:

Recommended Citation

Altman, Ronald Lee, "The computer-aided generation of flow-tables for asynchronous sequential circuits" (1968). *Masters Theses*. 5186.

https://scholarsmine.mst.edu/masters_theses/5186

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

THE COMPUTER-AIDED GENERATION OF FLOW-TABLES
FOR ASYNCHRONOUS SEQUENTIAL CIRCUITS

BY

RONALD LEE ALTMAN, / 945

A
THESIS

submitted to the faculty of

THE UNIVERSITY OF MISSOURI - ROLLA

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

Rolla, Missouri

1968

Approved by

John P. Kelly (advisor) John P. Kelly

John P. Kelly

TZ/TT
21
489

140570

ABSTRACT

One step in the synthesis of asynchronous sequential circuits is the construction of a flow table. This paper discusses the requirements of a computer program to allow on-line generation of flow tables for asynchronous sequential circuits. Such topics as the form of the data entered into the program, the type of terminal required, the routines necessary for the designer to enter and correct data, and the internal data structure are discussed. An algorithm for the generation of these flow tables is also presented.

ACKNOWLEDGEMENTS

The author wishes to express his sincere appreciation to Dr. James H. Tracey for his careful guidance throughout the entire project.

The author also wishes to acknowledge the typing efforts of his wife, Barbara.

TABLE OF CONTENTS

ABSTRACT	ii
ACKNOWLEDGEMENTS	iii
LIST OF FIGURES	v
I. INTRODUCTION	1
II. DATA REPRESENTATION	7
III. EXTERNAL FORMAT	16
IV. DATA CORRECTION	19
V. INTERNAL DATA FORMAT	21
VI. FLOW TABLE GENERATION ALGORITHM	29
VII. CONCLUSION	37
BIBLIOGRAPHY	38
APPENDIX A, A Description of the Support Routines Necessary for a CRT Terminal	39
VITA	43

LIST OF FIGURES

Figure		Page
1	Flow table for an asynchronous sequential circuit.	2
2	Labled flow table.	3
3	The primitive-form flow table for the DC S-R flip-flop.	5
4	A typical timing diagram for flow table of Figure 1.	9
5	The correspondence of the timing diagram with flow table location.	10
6	Timing diagram information represented as a sequence of code words.	12
7	Vertical representation of code word sequence.	13
8	Sample flow table generation program.	17
9	Typical flow table segment with two inputs and one output that could be generated by this program.	24
10	The flow table of Figure 9 represented as a subscripted array.	26
11	Output don't-care values.	29
12	Flow table generation algorithm.	32

I. INTRODUCTION

Sequential switching circuits are normally divided into two categories, synchronous and asynchronous. The synchronous circuits make use of a clock which permits the circuit to respond only during certain time intervals as defined by the clock. In an asynchronous circuit there is no clock and the circuits are permitted to function at the full speed capability of the gates. This paper will deal with asynchronous sequential circuits.

A restriction on the type of asynchronous sequential circuits to be considered here is that they operate in the fundamental mode and that only one input be allowed to change at a time. Circuits which operate in the fundamental mode are constrained to operate under the condition that the inputs are never changed unless the circuit is internally stable. Internal stability means that if the present input is maintained, there will be no further changes in the state of any signal lines.

One of the most important tools in logical design of sequential circuits is the flow table which was first formulated by Huffman¹ in 1954. The standard representation of a flow table as shown in Figure 1 is an array with the rows representing the internal states, the columns the input combinations, and the entries in the table representing the next internal state. The stable states are

shown in Figure 1 circled. Stable states are states where the next state is the same as the present state. The outputs in Figure 1 are shown only with the stable states.

		XY			
		00	01	11	10
S_1	$\textcircled{S_1}/0$	S_2	--	$\textcircled{S_1}/0$	
S_2	$\textcircled{S_2}/1$	$\textcircled{S_2}/1$	--	S_1	

Figure 1. Flow table for an asynchronous sequential circuit.

The dashes in the column under $X = 1, Y = 1$ (Figure 1) are called don't-cares. Don't-care states are states that because of constraints on the inputs will never be entered, or if they are entered the designer does not care what the outcome will be. With this being the case it is not necessary to specify what will happen under these conditions.

To illustrate the operation and meaning of the flow table in Figure 1 it has been redrawn and labels added in Figure 2.

The operation of the flow table can be demonstrated by considering the case where the circuit is in internal state 1 with inputs of 00 (location C Figure 2). The next state is the same as the present state. The circuit is stable and the next state is circled. The output is 0,

	XY			
	00 C	01 E	11	10 D
S ₁	(S ₁)/0	S ₂	-	(S ₁)/0
S ₂	(S ₂)/1	(S ₂)/1	-	S ₁ H

G
F

Figure 2. Labeled flow table.

and since the circuit is stable it will remain at 0 until there is a change in the input. Now if input X becomes a 1, the inputs are now 10 in internal state S_1 (location D Figure 2), the next state is S_1 and the output is 0. If input X becomes a 0 (location C Figure 2), the next state is S_1 . It is a stable state and the output is 0. If the inputs change to 01 (location E Figure 2) the next state is S_2 . The next state now becomes the present state (location F Figure 2) and the output becomes 1. The inputs are still 01 with the present state S_2 (location F Figure 2). The next state is S_2 , the output is 1 and the circuit is again stable. Thus, the entire operation of the circuit can be specified in flow table form and the output sequences for any given input sequence can be found in the manner just described.

The flow table can be a means of conveying large amounts of information about the circuit in a very compact manner. For example, the flow table of Figure 1 represents

a conventional DC S-R flip-flop. The word description of this flip-flop would be as follows: This circuit has two inputs X and Y and one output Z. Z is to be off (at logic 0) whenever X but not Y is on (at logic 1). Z is to be on whenever Y but not X is on. Z is to remain in its previous condition whenever X and Y are both off. X and Y are never both on at the same time.

When a designer begins to formulate a description of a circuit, the type of flow table normally used is the primitive-form flow table. A primitive-form flow table is one which contains only one stable state per row. In a primitive flow table the output is a function only of the internal states. Figure 3 is the primitive-form flow table for the example associated with Figure 1. The primary reason for generating a primitive-form flow table is its simplicity. Other types of flow tables are generally more compact, that is they have a fewer number of internal states, but they are seldom as easy to formulate as the primitive-form. A second reason for formulating this type of flow table is that it is easy to recognize physically unrealizable machines when generating their primitive-form flow tables. This is not always the case with the other forms of flow tables.

In a primitive-form flow table the mapping of internal states into output states is a simple one-to-one relation. The mapping of inputs into internal states is not a simple

		XY			
		00	01	11	10
S	S ₁	(S ₁)/0	S ₂	—	S ₄
	S ₂	S ₃	(S ₂)/1	—	—
	S ₃	(S ₃)/1	S ₂	—	S ₄
	S ₄	S ₁	—	—	(S ₄)/0

Figure 3. The primitive-form flow table for the DC S-R flip-flop.

relation. The primary concern in the generation of primitive-form flow tables is therefore to formulate the mapping of the inputs into the internal states.

Once the flow table has been constructed there exist computer programs which will accept it as input and yield as output the complete minimal and/or near minimal, hazard-free design equations for fundamental mode asynchronous sequential circuits². Programs also exist which will accept these design equations and convert them all the way to the layout of the printed circuit boards for the hardware³. To make use of these programs the logic designer must first convert the formal description of the circuit to be designed into a series of input-output sequences. These sequences are then used to prepare the flow table that is entered into the design programs. The area which has been greatly neglected in the computer-aided design of switching circuits is that of computer generation of flow tables.

The purpose of this paper is to study the requirements of a program to link the logic designer with the design equation programs. The function of this program is to obtain design information from the logic designer and to arrange this information into a flow table or other suitable form to be input to the design program. In order for this program to become a useful tool for the designer it must satisfy the following basic conditions:

- 1). The program should be able to accept data in a form that is familiar and meaningful to the logic designer.
- 2). The program should be on-line and conversational.
- 3). The program should allow maximum flexibility in the ability of the designer to change or modify the data that he is entering into the program.
- 4). A minimum amount of time should be required for the designer to learn to use the program.
- 5). Very little knowledge of the internal operation of the program should be required of the user.

This paper will present the requirements of the flow table generation program and show how they may be met in order to satisfy these five basic conditions.

II. DATA REPRESENTATION

As pointed out in the first basic condition, the method of communication of the necessary input data from the logic designer to the program must be natural and familiar to the logic designer. The first step in determining the characteristics of a language to be used to link the designer with the flow table generation program is to consider the form in which the designer first formulates a logic design.

When a logic designer first forms in his mind the description of a logic circuit it is normally either a set of input-output sequences or a word statement describing the circuit operation. When the logic description takes the form in input-output sequences they are usually visualized in terms of a timing diagram. When the design is originally in terms of word statements, it may be converted into some other more precise form. This may be a timing diagram or some other formal representation. A timing diagram is a sketch of the logical values of the input and output variables as a function of time.

Ideally the logic designer would like to enter into the program a description of the circuit in the exact form that it was originally conceived. At the present state of the art, if the design is in the form of a word statement this is not possible because of the complex

nature of natural languages.

A data representation which will satisfy both types of data is required. There exists a semiformal language used to formulate the logical description of sequential circuits called the language of regular expressions.⁴ For small machines regular expressions have the advantage of being compact and easy to generate. For machines having more than two or three inputs and/or outputs the regular expressions become very long and complex. The length and complexity would present only minor problems if it were not for the fact that there are no good algorithms available for the reduction of these expressions. The inability to reduce regular expressions paired with the difficulty involved in formulating long complex expressions makes it desirable to find another type of representation.

The next most favorable representation would be the timing diagram discussed previously. The timing diagram has three important factors in its favor. The first is that the logic design is sometimes first formulated in terms of these diagrams thus requiring no data conversion at all. The second is that the designer is familiar with preparing and using these diagrams. A third point favoring timing diagrams is that there are at this time no other formal representations developed even to the level of the regular expressions. Because of these facts it appears that the timing diagram is the best means of expressing

the logic design description.

A timing diagram is a visual representation of input and output states (logic 1 or 0) as a function of time. To illustrate this consider the example associated with Figure 1. The timing diagram for a small set of the possible input sequences is shown in Figure 4.

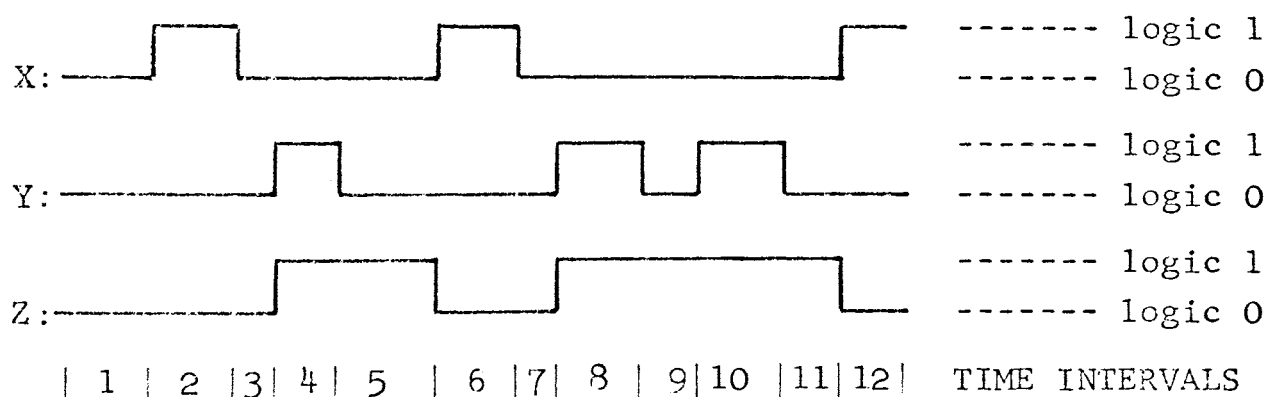


Figure 4. A typical timing diagram for flow table of Figure 1.

There are several important points to be mentioned concerning timing diagrams. First, the intervals of time numbered 1 through 12 are not all of the same length. This points out the fact that the circuit is asynchronous and is not restricted to function only at specified times. It can also be seen that the timing diagram shows the circuit reacting instantaneously to changes in the input. To show how the timing diagram is related to the flow table, Figure 5 lists the time intervals and the corresponding location in the flow table of Figure 2.

<u>TIME INTERVAL</u>	<u>TABLE LOCATION</u>
1	C
2	D
3	C
4	F
5	G
6	D
7	C
8	F
9	G
10	F
11	G
12	D

Figure 5. The correspondence of the timing diagram with flow table location.

By observing the table of Figure 5 it can be seen that the timing diagram provides information about the stable states of the flow table. The edges of the wave forms represent transitions through unstable states. With a timing diagram as input information the flow table generation program has only to fill in the necessary unstable states to complete the flow table.

In order to simplify notation, the set of logical values for all of the input variables at any given time interval will be defined to be the input code word

associated with that time interval. To illustrate this, consider the time interval number 1 of Figure 4. The input code word for this time interval is 00. For time interval 2 the input code word is 10. The convention used to order the bits of the code word is arbitrary and in this case the first bit refers to the first input, the second bit to the second input, etc. For example, the code word for time interval 2 is 10 where the 1 refers to the logic value of input X, and the 0 refers to the logic value of input Y. It is also sometimes convenient to refer to the code words by their decimal value. This is found by considering the code words as binary numbers and converting these to decimal. For example the code word 11 would be called by its decimal name as 3, 10 by the name 2, etc.

All of the conventions discussed for input code words apply equally well for the output code words.

Applying the concept of code words to the timing diagram it can be seen that timing diagrams can also be represented as a string of input-output code words. The timing diagram of Figure 4 translated into code words is shown in Figure 6.

With the data represented in timing diagram format, the terminal through which the data is entered into the program must be capable of entering either sketches or code words. There are basically three types of terminals

X:	0	1	0	0	0	1	0	0	0	0	0	1
Y:	0	0	0	1	0	0	0	1	0	1	0	0
	└─ Input code word 1											
Z:	0	0	0	1	1	0	0	1	1	1	1	0
	└─ Output code word 1											

Figure 6. Timing diagram information represented as a sequence of code words.

that would be suited for this problem: a CRT display, a sketch-pad type terminal, and the teletype terminal. Since other information than just the timing diagram would be required, a CRT or a sketch pad would also require a keyboard.

If a CRT or sketch-pad terminal is used to enter the data, the timing diagram sketches will be converted to a sequence of code words arranged as in Figure 6. With the teletype terminal the code words may be entered as in Figure 6 but it would require an unnecessary amount of programming to rearrange the code words into the proper form. With the code words in the form of Figure 6 it would require the designer to enter all of the bits of X before he entered Y, and all of X and Y before Z, etc. This forces the designer to perform some preliminary pencil and paper work before entering the sequences. A better way to enter the code words is to arrange them vertically as in Figure 7. This permits the designer to eliminate preliminary paper work since he enters all of the values

	X	Y	Z	
Input code word 1	0	0	0	Output code word 1
	1	0	0	
	0	0	0	
	0	1	1	
	0	0	1	
	1	0	0	
	0	0	0	
	0	1	1	
	0	0	1	
	0	1	1	
	0	0	1	
	1	0	0	

Figure 7. Vertical representation of code word sequence.

of the inputs and outputs on the same line. This method adds a restriction to the total number of inputs and outputs; they may not exceed the number of spaces per line of the teletype. This restriction is not as limiting as it might appear since there are normally about 50 characters per line on a standard teletype unit. The programs available for synthesizing logic equations from these flow tables will not in general accept any more than 25 inputs and outputs.

There are several major advantages and disadvantages for each of the three types of terminal. The primary advantage of the CRT and the sketch-pad is that they would allow the designer to enter the data in a graphical

format. The sketch-pad has a major disadvantage in that changes in the data sketches would result in a very cluttered display. The CRT terminal does not have this disadvantage; in this case diagrams can be quickly and easily removed from the screen. With the teletype terminal the data must be entered in code word form, and there is a small problem with correcting errors. Errors on the teletype cannot be easily removed, as in the case of the CRT, but they can be neatly worked around. This can be done by typing to the right of or directly below the incorrect entry. Another disadvantage of the CRT and the sketch-pad would be that a complex routine would be needed to take care of proper vertical alignment of the sketches. This is not a problem with the teletype since each code word is itself a time interval. It is therefore only necessary to enter the bits of the code words as shown in Figure 7.

Another important consideration is the cost of these different types of terminals. An exact cost comparison of the CRT and the teletype is not possible because of the wide range of features and options available. To illustrate the large cost difference the following are the approximate prices for IBM units suitable for this application. The IBM CRT display unit with light pen cost approximately \$90,000. The IBM Printer Keyboard unit costs \$3000. Considering this large price difference,

and the simpler data conversion routines required, the remainder of this paper will be oriented around the teletype terminal.

III. EXTERNAL FORMAT

Since the program is to be on-line and conversational, an important requirement to be considered is the format of the dialogue between the designer and the program. With this type of system it is important to keep the format flexible. A discussion of some of the most important requirements for flexibility is included here and a typical program is shown in Figure 8.

The primary purpose of the program is to request data from the designer and then assemble this data into a flow table. Since the program requests the data from the designer these requests should be clear and the form of the answer should be apparent by the manner of the request. It is not always possible to word a request such that it is evident how to respond. In this case the user of the program should be supplied with a print-out of a typical design problem such as that shown in Figure 8. This may be supplied in an operator's manual along with a detailed description of the routines available to the user.

One of the most important features of the program is to allow the designer the freedom to arrange his data in a manner such that it conveys the maximum information to both the program and the designer. In order to realize this feature the program must allow the user to insert

HOW MANY INPUT VARIABLES WILL THERE BE?

2.

WHAT ARE THEIR NAMES?

IN_1, IN_2

HOW MANY OUTPUT VARIABLES ARE THERE?

1.

WHAT IS ITS NAME?

OUT

ENTER THE INPUT SEQUENCES FOR THE FOLLOWING:

W	I	I	
O	N	N	O
R	—	—	U
D	1	2	T
#			
..
1	0	0	0
2	1	0	0
3	1	1	1
4	0	0	0
5	0	1	0
6	1	1	0
7	1	0	0
8	0	0	0
9	0	1	0
10	0	1	0
11	0	0	0
12	1	0	0
13	1	1	1
14	1	0	1
14	1	0	0
15	0	1	0
16	0	0	0

END.

Figure 8. Sample flow table generation program.

Note: The data entered by the designer has been shaded.

blanks and commas wherever he chooses. Examples of this can be seen in the list of input names in Figure 8.

The general method of data request is as follows. The computer types out a request which ends generally with a question mark and a carriage return and line feed. The user is then free to enter the requested data. As mentioned previously, the use of spaces and commas is left up to the designer. In the example dialogue of Figure 8, the use of the question mark is illustrated with the request "What are their names?". The end of communication between the designer and the program is denoted by a period or the word END.

It is often necessary for the designer to determine the numerical position in the input stream of a particular code word. The number of the code word of the current line appears at the beginning of the line in Figure 8. These numbers are printed automatically by the program.

IV. DATA CORRECTION

After the designer has entered the data into the program he will often wish to change that data. For example the name given by the designer to the output variable in the program shown in Figure 8 was ended with a right paren when it should have been ended with a period. To correct this error the user typed the vertical arrow (the character used to call the routine to delete the last character) and then typed the correct symbol, in this case the period.

Another useful routine to correct errors is the "delete the current line" routine. This routine is represented by the horizontal arrow. The routine functions as its name implies. It causes the values that have been entered into the program on the current line to be deleted from memory. The program then allows the designer to start again at the beginning of a new line.

The two correction routines mentioned above will handle most of the corrections to a typical design problem. It may sometimes be the case that the designer would like to modify only a single code word in the sequence. The routine to handle this is the MODIFY X ($I_1, I_2, \dots, I_n, O_1, O_2, \dots, O_m$). This command may be given at any time in the input stream by giving a line feed and carriage return and then entering the instruction in the form

shown above. The I's are the input values in the order that they were first entered in the program. The O's are the output values. The X is the position in the input stream that is to be changed. For example, if the designer wanted to change the input stream in position 3, which now contains the value 111, to the value such that IN_2 and OUT are both 0 and IN_1 is 1, the MODIFY 3(1,0,0) command would be given.

After several corrections have been made to the code word data the printed page may become cluttered with corrections and requests for corrections. A routine to redisplay the data up to the last entry could be executed. This command could be executed any time up until the program is told to continue. The command could be initiated by giving a carriage return and line feed followed by the command REPRINT. This routine will skip three or four lines and reprint all of the corrected code word data that has been entered up to the point that the command was given. If the command was given before the entire sequence of code words was entered, the program will terminate the REPRINT routine in a position such that new code words may be entered.

V. INTERNAL DATA FORMAT

The structure of the data in the computer is a problem which arises in every programming problem. In this case the primary data coming into the machine will be in the form of a string of 0's and 1's. It should be noted that the data referred to here are the strings of code words. Other data are also entered, for example the names of the input and output variables, but their form is not as important to the primary purpose of the design program.

Since the data coming into the computer will be binary in nature, it is logical and most beneficial in terms of computer storage, to allow each entered binary digit to require only one bit of computer core. At this point it becomes necessary to consider the relationship between the length of a code word and the length of the hardware computer word. There are two approaches to consider. The first is to recognize the hardware word as a boundary and work only with the fixed hardware word length. The second is to disregard the hardware word length using software techniques to define the relationship between the hardware word and the code word. The choice between the two alternatives depends on the computer system available, the software, and the experience and skill of the programmer. If the first system is used

it would require that certain sections of the program be written in assembler language. If the second method is chosen either assembler language or a very high level language such as PL/1 would be required to allow individual bit manipulation. The choice made here is to assume use of the higher level language of PL/1 and assume that a word will be defined by the program to be of the correct length. For example an input code word for a program which has five inputs will be five bits long in the computer. In PL/1 this is a simple matter of defining the code word to be a bit string of length five.

There are two major reasons for assuming the use of PL/1. The first is that it is available on the IBM 360 system and is much simpler to use than the assembler language. The ease of use is by no means a trivial matter since it may require as much as six to twelve months to become sufficiently familiar with the assembler language to program relatively simple algorithms. In PL/1 it is not necessary to be familiar with the entire language to program rather complex algorithms, and PL/1 is structured in a manner which is easy to learn even for the person who has never programmed before. The second reason for assuming the use of PL/1 is that the program which makes use of the flow table that is generated in this program is written in PL/1. With the programs in the same language the transferring of data from one to the other can be

accomplished without a major restructuring of data.

With the freedom to define the length of input and output code words it is logical to define them as follows. The input code words will be defined as a bit string vector one word wide and M words long. M , the length of the array, is not exactly known until all of the code words are entered. Since this length must be defined, a nominal value should be given to this length which is dependent on the storage space available. When the array is filled the entire array would then be stored on tape or disk and the core area could again be used for new data. This process would be repeated until all of the code words are entered into the program and stored on disk or tape. At this point the first set of code words would then be entered into the array and used to generate the flow table. The output code words would be handled in the same manner.

The flow table to be generated is in primitive form and a stable state is never used more than once. A typical flow table of this type appears in Figure 9. It can be seen that this flow table is very simple in nature since each row has only two specified entries and has only one output, that being associated with the single stable state. The internal representation of the flow table could then be considered as an array. Each internal state would be a row of the array, for example

	AB 00	01	10	11
1	①/1	2	—	—
2	—	②/1	3	—
3	4	—	③/0	—
4	④/0	—	—	5
5	6	—	—	⑤/1
.

Figure 9. Typical flow table segment with two inputs and one output that could be generated by this program.

internal state 1 would be the first row of the array. Each column would represent an input combination; thus the entry in the column for the input combination 101 could be found by taking its integer value and adding two. Two is added since the input combination 000...0 will be considered to be the second column, and the output of the stable state for that row will be given in the first column. Figure 10 illustrates the flow table layout as an array. The exact number of internal states is not known at the beginning of the program so the maximum number of rows should be fixed to some arbitrary value, A, and the disk storage should be used in the same manner as was done with the input code words.

Initially the entries in all rows of the flow table should be don't-care, the assumption being that the designer is only concerned with the circuit operation as specified by the input-output sequences. All sequences not specified by the designer will be considered don't-care. The name of each internal state will be the number of the row in the array containing its entries. For example internal state 2 is row 2 of the array. In order to represent don't-care internal states there will be no internal state 0. A 0 as a next state will therefore be considered a don't-care entry. The problem of don't-care outputs is not as easily solved as that of internal states.

Figure 10. The flow table of Figure 9 represented as a subscripted array.

		FLOW(i,j)				
		1	2	3	4	5
j=						
i=	1	3	1	2	0	0
	2	3	0	2	3	0
	3	0	4	0	0	5
	.					
	.					
	.					

NOTES: The length of a word stored in memory has been assumed to be 12 bits. This is an arbitrary assumption for the purpose of this example only. All entries are shown as decimal integers. The entry $i=1, j=1$ is the value of the output and has the value 3. This is the decimal integer value of the binary number 000000000011. This binary number is specifying that the first output has the value 1 since the first pair of bits have the same value 1.

Since it is possible to have outputs of 0 or 1 neither of these may be used to represent a don't-care output. A solution to this problem could be to use a character such as a dash to represent a don't-care output. This is not desirable since the internal representation of characters may be six or more bits long. Since the 0 and the 1 can both be stored in a single bit of memory it could be unreasonable to require a don't-care to take up six bits. The solution to this problem, which is compatible with the programs that will eventually use the flow table², is to use two bits of memory to store the output value. If both bits are the same then the output is that value. If the two differ then the output is to be a don't-care.

With the representation of the don't-care determined it remains to initialize the memory data area, with the memory laid out as an array as shown in Figure 10. Each entry will have a specific meaning. For example, the first entry in the first row of the array will be the output of the stable state in the first row of the flow table. The next 2^n (n = number of inputs) entries contain the next state entries of the first row. The first entry in the second row is the output associated with the second row and the entries of the second row are in the next 2^n words. This pattern repeats itself throughout the data area.

It should be noted that the length of a word in the array will limit the number of possible internal states and outputs. For example, if L is the length of a word in the array in bits, then the number of internal states possible would be $2^L - 1$. The number of outputs would be limited to $L/2$, since each output variable requires two bits of memory. These two equations will allow the programmer to specify the exact word length for the flow table array. The program can calculate the two lengths discussed above and define the array to have words whose length is equal to the larger of the two values. Since the output code word requires two bits for each data bit the length chosen should be an even number.

VI. FLOW TABLE GENERATION ALGORITHM

An algorithm is presented here which may be used to generate the primitive-form flow tables previously discussed. The algorithm is presented in a form which makes use of the subscripting routines of any higher level language. As has been previously mentioned, PL/1 will serve as the language model. This has been done primarily to show how specific operations can be carried out, for example the initialization of data arrays. The discussion of these specific points in terms of a specific language will allow the programmer to understand their purpose and provide a pattern from which the program can be constructed.

The first step in the construction of the flow table is the initialization of the data area. To initialize the data area to the don't-care condition it is necessary to initialize both the output variables and the next state entries. Since the output variables are always in the first column of the array, it will be necessary to initialize this column to the output don't-care value shown in Figure 11. The remainder of the array should be set to

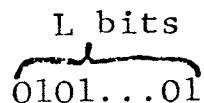
L bits

 0101...01

Figure 11. Output don't-care values.

all 0's. The initialization of the array in PL/1 is a very simple single statement instruction and to initialize any column also requires only one statement.

After the data area designated by the array FLOW has been initialized to don't-care the flow table may be constructed. The flow table generation algorithm and a list of symbols used are given on the following pages.

A LIST OF SYMBOLS USED IN FLOW TABLE ALGORITHM

- FLOW(X,X) - A two dimensional subscripted array containing the generated flow table.
- L - Length of a word in the array FLOW in bits.
- n - Number of inputs.
- M - Total number of input-output code word pairs.
- m - Number of outputs.
- A - Maximum number of rows of flow table that can be stored in a single overlay.
- B - Total number of overlays of FLOW initially 0.
- ICW(i) - A subscripted vector containing $i=1,2,\dots,M$ input code words each of length n.
- OCW(i,j) - A double subscripted vector containing M output code words each of length m, with the subscript $i=1,2,\dots,M$ denoting the code word and $j=1,2,\dots,m$ denoting a specific bit of the code word.
- OC(j) - A single vector of length $2m$ containing the output code with each bit duplicated.
- PS - Has as its value the present internal state value.
- NS - Has as its value the next internal state value.
- FLOW(*,*) - PL/1 notation designating all rows and all columns of array FLOW.
- FLOW(*,1) - Designates the first column of the array FLOW.

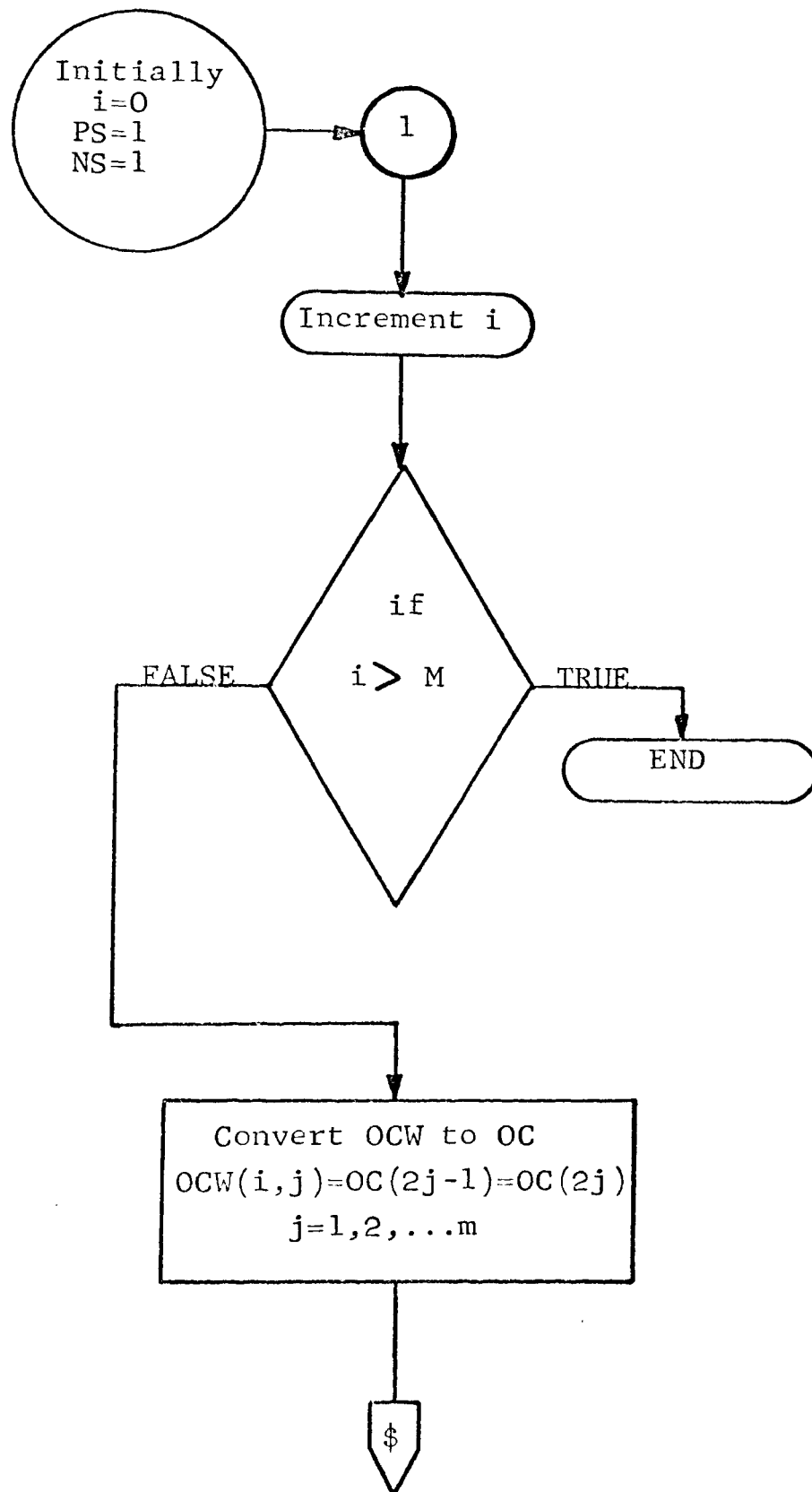


Figure 12. Flow table generation algorithm.

Figure 12. (continued)

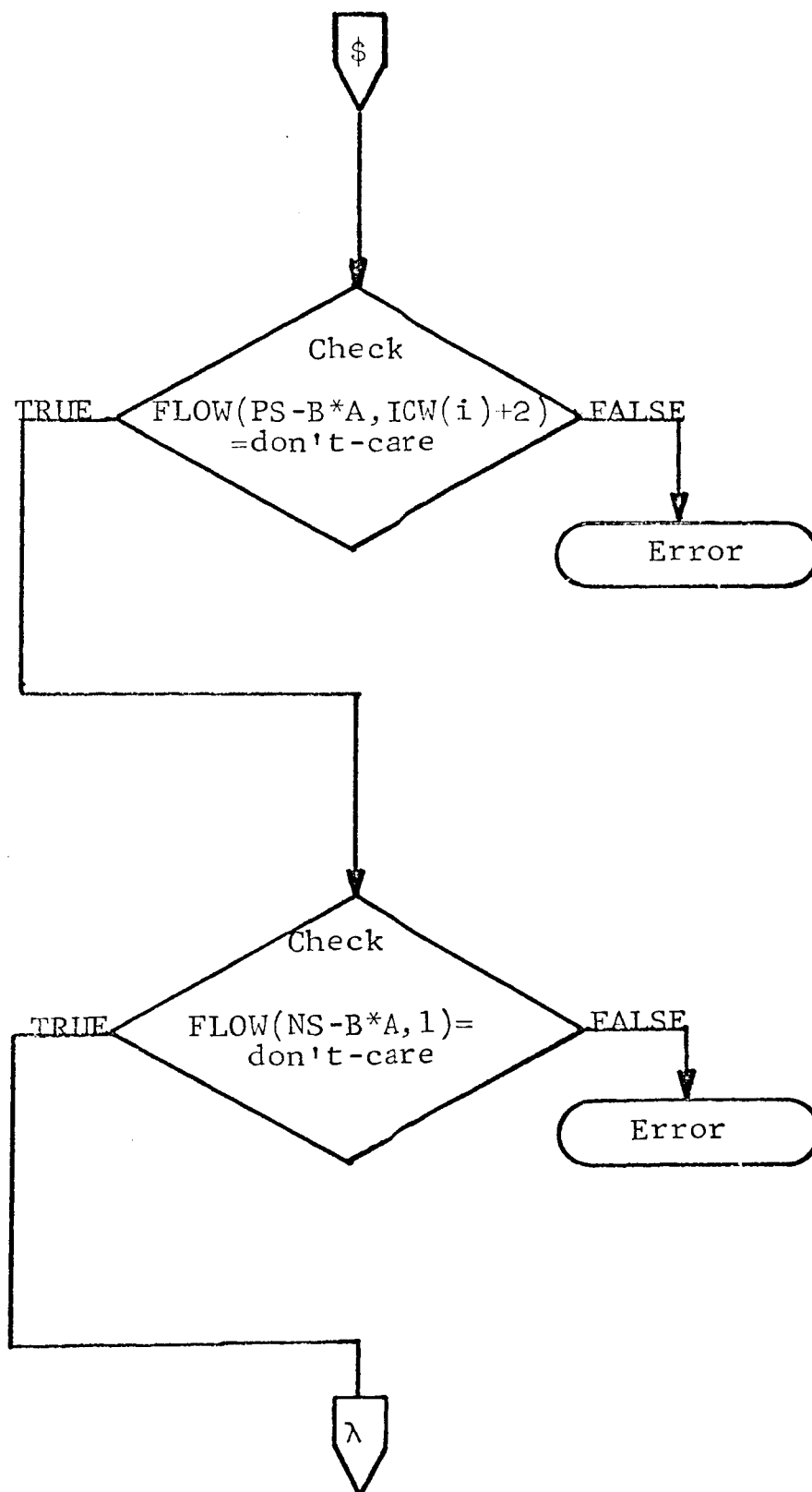


Figure 12. (continued)

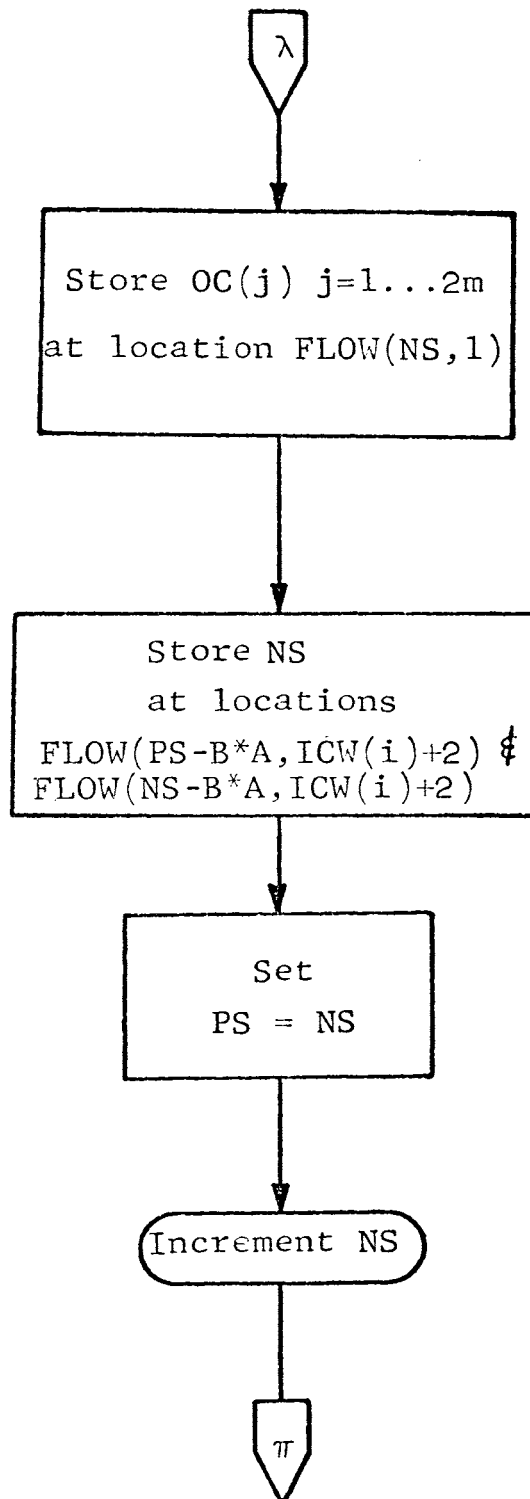


Figure 12. (continued)

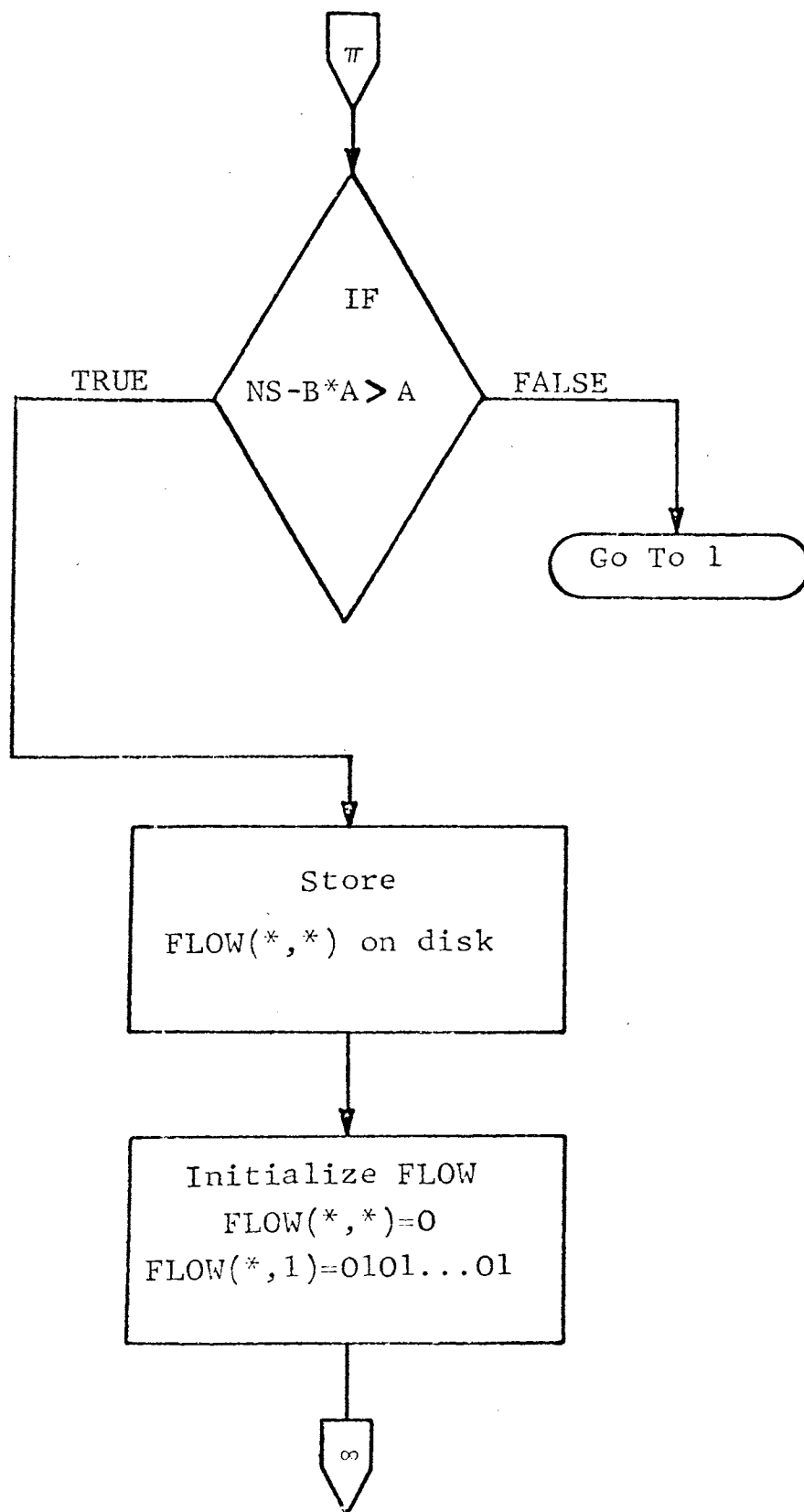
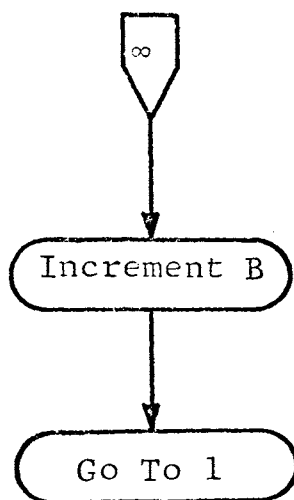


Figure 12. (continued)



VII. CONCLUSION

This paper has presented the requirements for a program which will generate a fundamental mode primitive-form flow table from a sequence of input-output code words. The algorithm presented for constructing these flow tables has been oriented toward the use of a high level language such as PL/1. This has been done to make use of such features as the subscripting routines and the ability to define bit string word lengths and manipulate the individual bits. This algorithm has been formulated in such a manner as to allow it to serve as a framework around which more complex algorithms which make use of the internal states more than once may be constructed.

This investigation has provided a beginning to the computer-aided construction of flow tables. It has also pointed out an important area that requires further study. This is the area of the formulation of all possible input-output sequences. It appears that some type of formal language which would allow the expression of long sequences in short descriptive phrases is required.

BIBLIOGRAPHY

1. Huffman, D. A., "The Synthesis of Sequential Switching Circuits," J. of the Franklin Institute, Vol. 257, pp. 161-190 and 257-303; March and April, 1954.
2. Smith, R. J., "A Programmed Synthesis Procedure for Asynchronous Sequential Circuits," M.S. Thesis, University of Missouri at Rolla, Rolla, Missouri November 1967.
3. Breuer, M. A., "General Survey of Design Automation of Digital Computers," Proceedings of the IEEE, Vol. 54, No. 12, pp. 1708-1721; December, 1966.
4. Brzozawski, J. A., "A Survey of Regular Expressions and Their Applications," IRE Trans. on Electronic Computers, Vol. EC-11, pp. 324-335; June, 1962.

APPENDIX A

A Description of the Support Routines
Necessary for a CRT Terminal

Presented here are the routines necessary for use with a CRT terminal. These are primarily service routines to provide the designer with a flexible format for sketching the timing diagrams on the face of the CRT screen. These routines are, in most cases, only the basic minimum required for efficiently using the CRT display. More routines may be added as experience with the system is acquired. With these routines the CRT terminal may be used in place of the teletype. No further program modifications would be required.

NAME DISPLAY: The name of each input and output variable will be printed on the screen followed by a delimiter such as a colon. This will indicate to the user that he is to begin on that line to sketch the timing diagram for that variable.

TIME INTERVALS: The switching boundaries will be indicated by vertical dotted lines. These lines will be generated and displayed on the CRT by the program. The spacing of the timing intervals will be a constant which will depend on the size and resolution of the CRT used.

DATA CONVERSION: The data entered on the CRT terminal will be sketched in a continuous line form. This form

must be converted into digital information. This requires that for each time interval a code word must be constructed. The bits of the code word may be found by comparing the vertical position of the sketch for each time interval with a center line for each variable. Segments above this center line will be considered to be logical 1, and those below will be logic 0.

CLEAN UP COMMAND: The designer will be able to issue this command at any time during the input sequence. The function of the command will be to straighten up the input sketches forming them into straight line segments to improve the clarity of the input sequences. This will be necessary since the lines formed by a light pen are not always neat and straight.

CONTINUE HORIZONTAL: This command will be given if it is necessary to extend the I/O sequences beyond the length of the display screen. This command given at the end of the line will cause the name of the next input or output variable to be printed out at the beginning of the next line and allow the designer to give the sequences for that variable until he reaches the end of the line. This printing of names will continue until the last output name and the sequence for that name has been given. At that time the computer will clear the screen and print out the first input name again, thereby allowing the designer to continue the sequence for all the variables.

This command may be given as many times as is necessary to obtain all the input sequences.

CONTINUE VERTICAL: This command will be given and executed by the program and is used to allow more input or output variables to be entered than is possible to put on the CRT face at any one time. This command clears the screen except for the timing interval lines and prints out the name of the next input or output at the top of the screen. The user continues sketching the waveforms as usual.

NOTE: Each vertical extension will be given a number, Ver 1, Ver 2, Ver 3, etc. The first set of sequences will be vertical 0. Each horizontal extension will be given a number in the same manner as the vertical. This numbering set will allow the designer to request by name any section of the input sequences. This will be useful if modifications of the input are necessary.

BLANK TIMING INTERVAL: This will allow the designer to have the specified timing interval lines removed from the CRT display but not from memory. This will reduce the visual clutter on the screen.

SHOW TIMING INTERVAL: This command will allow the user to ask for timing intervals to be redisplayed at certain points to clear up confusion.

END _____: This command will be used to terminate any commands which do not have an automatic termination by entering the name of that command in the underscored

section. This command will be entered into the computer through the keyboard.

ERASE: This will delete lines or sections from the CRT face and from memory. The area that is to be deleted will be defined by encircling the area to be deleted with the light pen.

DISPLAY HORIZONTAL M VERTICAL N: This instruction calls for the section designated horizontal M and vertical N to be displayed.

VITA

Ronald Lee Altman was born on July 4, 1945 in St. Louis, Missouri. He received a Bachelor of Science degree in Electrical Engineering from the University of Missouri at Rolla in June, 1967. He has been enrolled in the graduate school at the University of Missouri at Rolla since June, 1967. He has been on the staff of the Electrical Engineering Department since September, 1967.