Synthesis heuristics for large asynchronous sequential circuits

Robert Judson Smith

SYNTHESIS HEURISTICS FOR LARGE ASYNCHRONOUS SEQUENTIAL CIRCUITS

by

ROBERT JUDSON SMITH II, 1944-

A DISSERTATION

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI - ROLLA

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

1970

James H. Tracey
Advisor

S. H. Szygenda

Herbert R. Alcorn

C. Y. Ho

Frank J Kern

ABSTRACT

Many well-known synthesis procedures for asynchronous sequential circuits produce minimal or near-minimal results, but are practical only for very small problems. These algorithms become unwieldy when applied to "large" circuits with, for example, three or more input variables and twenty or more internal states.

New heuristic procedures are described which permit the synthesis of very large machines. Although the resulting designs are generally not minimal, the heuristics are able to produce near-minimal solutions orders of magnitude more rapidly than the minimal algorithms.

A method for specifying sequential circuit behavior is presented. Input-output sequences define submachines or modules. When properly interconnected, these modules form the required sequential circuit. It is shown that the waveform and interconnection specifications may easily be translated into flow table form.

A large flow table simplification heuristic is developed. The algorithm may be applied to tables having hundreds of rows, and handles both normal and non-normal mode circuit specifications.

Nonstandard state assignment procedures for normal, fundamental mode asynchronous sequential circuits are examined. An algorithm for rapidly generating large flow table internal state assignments is proposed.

The algorithms described have been programmed in PL/1 and incorporated into an automated design system for asynchronous circuits; the system also includes minimum and near-minimum variable state assignment generators, a code evaluation routine, a design equation generator, and two Boolean equation simplification procedures. Large sequential circuits designed using the system illustrate the utility of the heuristic procedures.

ACKNOWLEDGEMENTS

The author would like to express his appreciation to Dr. J. H. Tracey for his guidance, advice and patience during the studies which led to this dissertation. Thanks are also extended to the author's wife, Jean-Marie, and to Sandra Wilson for their diligent typing of this manuscript.

The research described in this dissertation was supported in part by National Science Foundation Grant GK 2017.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figures Page

LIST OF TABLES

I. INTRODUCTION

Sequential circuits which operate without synchronizing (or clock) signals are commonly called asynchronous sequential circuits. An important advantage of asynchronous design is that the circuit may respond to input changes at basic device speed, rather than awaiting the arrival of clock pulses.

The operation of an asynchronous sequential circuit is often described by means of a flow table (see figure 1). The flow table

	I_1	I_2	I_3	I_4
1	①/00	2	①/11	6
2	1	②/11	4	3
3	-	6	1	③/00
4	④/10	6	④/10	6
5	4	⑤/10	-	6
6	1	⑥/01	1	⑥/00

Figure 1. A Typical Flow Table -- Example A

columns represent input states, while the rows represent internal states assumed by the machine. Each flow table entry specifies the next-state resulting from a given input and internal state.

A circuit is said to be operating in the fundamental mode if no change in input state is allowed unless the circuit is stable, i.e., the next-state of the circuit is the present state. Output specifications are usually associated with stable next states. If the next-state is not the present state, the latter is termed unstable and implies a transition to another state. In normal mode circuits, transitions must be made directly to a stable state. This work is largely concerned with normal, fundamental mode asynchronous sequential circuits.

The circuit model to be used throughout is shown in figure 2. The sequential circuit is composed of a set of inputs $I_1,\ldots,I_n$, present state variables $y_1,\ldots,y_m$, outputs $O_1,\ldots,O_k$, and next-state variables $Y_1,\ldots,Y_m$, which after passing through asynchronous delays $d_1,\ldots,d_m$ become present state variables. The delays usually represent

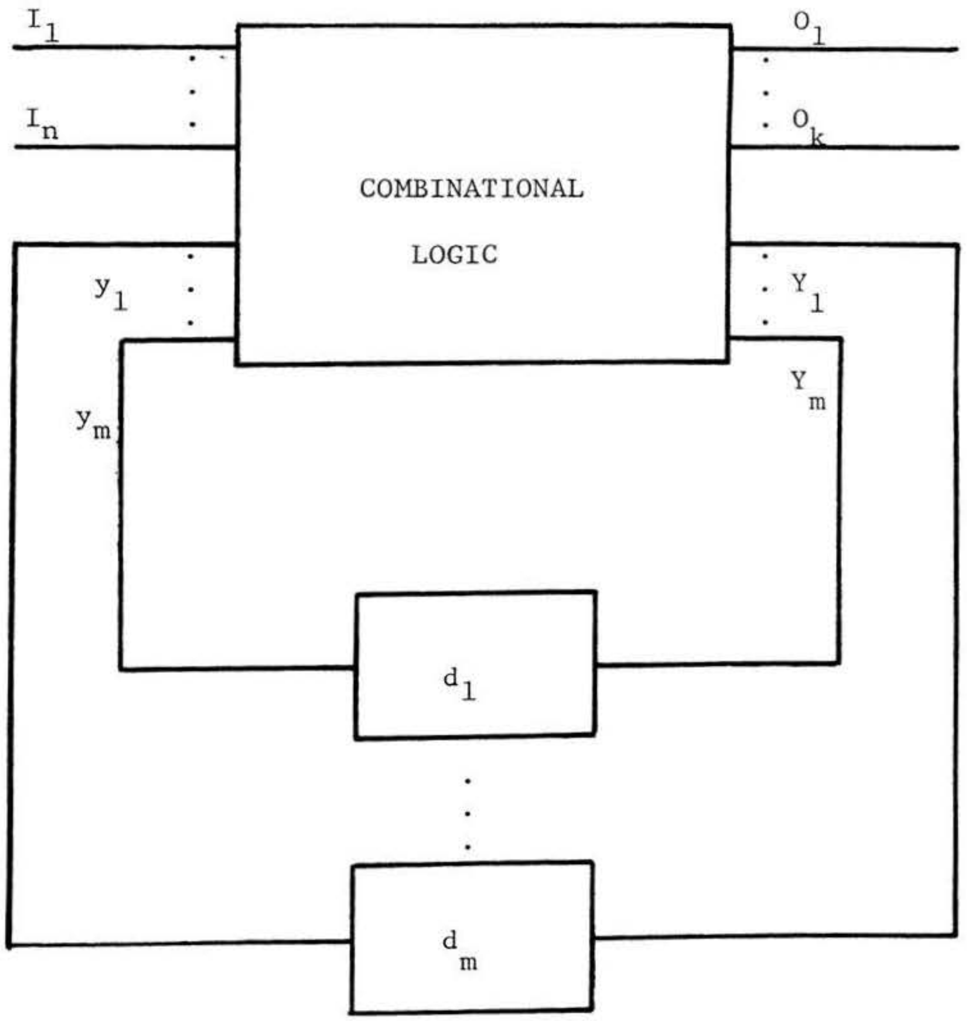


Figure 2. Asynchronous Sequential Circuit Model.

the propagation times of next-state signals through the combinational logic.

A commonly employed manual synthesis procedure[1] begins with the formulation of a verbal or diagrammatic circuit behavior description. The circuit description is translated into the form of a flow table, usually employing a non-algorithmic procedure. The flow table is then minimized or simplified using one of several available algorithms[2,3,4,5].

A satisfactory internal state assignment must then be found for the reduced flow table. The greatest difficulty in making an asynchronous sequential circuit state assignment is avoiding critical races. A critical race exists when, due to unequal signal transmission delays, there is a possibility that the stable state reached is not the intended one. Huffman[6], Liu[7], and others have described universal state assignments which depend only on flow table size. Universal, or standard assignments, are relatively easy to construct and are independent of flow table structure. Tracey[8] has shown how to construct nonstandard codes (dependent on flow table structure) which permit no critical races. Nonstandard codes generally have fewer state variables and yield simpler circuits than standard codes.

Once a critical-race-free code has been generated, the designer forms a transition table by substituting internal state codes for next-state entries in the flow table. Excitation and output Boolean equations are then derived from the transition table. Finally, these design equations are simplified and converted to hardware implementation form.

The above manual synthesis procedure is practical only when applied to quite small circuits. For larger sequential circuits, several authors have described automated design systems which perform steps of the manual procedure.

Elsey[9] in 1963 described a machine language computer program which accepted a primitive (one stable state per row) flow table. Very elementary simplification procedures were applied to the flow table and a non-normal mode standard assignment was generated. Unsimplified design equations were produced by reading directly from the transition table. Although Elsey's program produced design equations with a large amount of simplification still required, it was able to synthesize extremely large flow tables: a 117 column by 33 row flow table design was produced in 213 seconds.

Smith, et. al., have written a PL/1 program[10] which accepts a simplified flow table description of an asynchronous sequential circuit. Either minimum or near-minimum variable state assignments may be generated. An assignment evaluation algorithm predicts which of several codes generated will yield the simplest design equations. A complete set of design equations is then produced without constructing transition or excitation tables. Each design equation is simplified to an irredundant sum of prime implicants, and static hazards are removed. This automated design system functions well for flow tables of up to about 15 rows by four columns, but becomes prohibitively slow for larger flow tables.

Burton and Noaks, in a recent paper,[11] have briefly mentioned any asynchronous design automation program under development. Given a simplified flow table, a redundant state assignment is generated which allows the excitation equations to be readily derived. The code is then simplified by examining the design equations resulting from the redundant assignment. The program does not presently generate output equations. Since the system is still under development, no performance data have been published.

Tan[12] recently described a computer aided procedure for realization of asynchronous sequential circuits. The circuit to be synthesized is described by a simplified flow table and several state assignments are constructed. The code exhibiting the least amount of state variable dependency is selected for use. Design equations are not generated or simplified.

All of the synthesis procedures described above--both manual and programmed--have serious limitations. The manual procedure can be used only on flow tables having fifty or less next-state entries. Manually exercised minimum or near-minimum variable state assignment algorithms become unmanageable for flow tables of more than eight rows. Manual simplification (or minimization) of Boolean equations of more than seven variables is generally difficult.

None of the automated design systems described adequately deal with the highly significant problem of flow table simplification. The nonstandard state assignment techniques described in (6,7,8) all appear to be unsuitable for large flow tables because they require the manipulation of extremely large amounts of data. Elsey's nonnormal mode realizations lead to unnecessarily complex excitation equations. None of the systems cited are capable of simplifying large systems of Boolean equations.

This dissertation describes several algorithms which have been developed expressly to synthesize very large asynchronous sequential circuits. Emphasis has been placed on reducing synthesis costs without introducing large amounts of hardware redundancy. Heuristic procedures have been used to improve synthesis speed, at the cost of circuit minimality. Since minimal solutions for designs of the size considered are unknown it is not possible to evaluate heuristic solutions in terms of minimal designs. The procedures described herein will rather be justified by comparing their performance on medium and small circuits with previously known algorithms, and by demonstrating their capability to synthesize circuits far larger than the capacity of other algorithms.

A design automation system has been developed to facilitate comparison of various synthesis procedures. With this system, problem descriptions may be entered at any of six stages in the automated design procedure, and synthesis may be interrupted at any later stage. Several of the minimal or near-minimal techniques employed in the system are adoptions of programs previously developed by the author.[10]

Other routines, which will not be described in detail, include a state assignment evaluation program[13], a Boolean equation sum of products simplification routine[14], and a static hazard removal program. Although the programming system currently operates in a batch processing environment, it is intended to eventually be available in conversational mode.

The asynchronous sequential circuit design programs previously developed require that the circuit be initially described in the form of a flow table. However, complex sequential circuits are usually not perceived initially as flow tables. Often designers think first of of responses to specific sequences of input states. A specification for the required circuit is then derived by assembling the sequence specifications in some desired manner. The result--usually a very informal description--must then be translated into flow table form. For large circuits (with perhaps five or more inputs and many outputs), the task of writing a flow table description may become quite formidable--a flow table representing a circuit with five input variables has 32 columns.

Chapter Two describes a sequential circuit specification technique which closely resembles the informal "response to input sequences" approach which precedes flow table construction. It is shown that the resulting specification may be translated into either a single flow table, or into a network of interconnected, relatively simple module descriptions.

The simplification of large flow tables is not performed by any of the normal mode design automation systems cited. Since large sequential circuit flow tables are almost always generated in non-minimal form, simplification is desirable in order to reduce large flow table synthesis costs and hardware complexity. Much work has been done in the area of flow table minimization; however, it is shown that minimization is impractical for large flow tables. Little has been published concerning simplification of large flow tables.

Chapter Three describes a heuristic flow table simplification algorithm. It is based on easily detected compatibility relationships and immediate table reduction. A programmed version of the algorithm allows the user to influence the "cost" (i.e., computer time consumed) of a flow table simplification. The procedure may be applied to either normal or non-normal flow tables.

State assignment techniques incorporated in known synthesis systems have been found inadequate for large flow tables. Chapter Four examines presently available coding procedures and proposes an extension of Tracey's method two[8] for use on large flow tables.

II. A SPECIFICATION TECHNIQUE FOR LARGE ASYNCHRONOUS SEQUENTIAL CIRCUITS

Sequential circuit specifications as originally conceived by designers seldom resemble the familiar flow table form. Often, a designer originates a sequential circuit behavior description in the form of a word statement, or a series of "responses to inputs", which after evaluation and modification is manually translated into flow table or hardware circuit form.

An important facet of a design automation system ignored by previously developed systems[10,11,12] is the translation of designs in originally conceived form into the more tractable flow table form. This chapter presents a sequential circuit description technique which closely resembles the "input/output" thought process and is easily translated into flow table form.

A. Background

Altman[15] has described a method for translating a sequence of input/output (I/O) specification pairs into a flow table having one stable and one unstable state per row. Each row's stable state corresponds to the input state for an input/response specification; the output associated with this stable state is the specified circuit response. An unstable next state entry in the flow table row corresponding to the previous specification is the only transition leading to the stable state. Likewise, an unstable next state entry causes the transition to the following stable state.

A sequence, as used here, consists of a set of I/O specifications which follow one another such that each has at most one predecessor. A problem with a sequence description of a sequential circuit is the possibility of having to repeat long lists of specifications in order to express alternate behaviors at a "branch point;" each string of inputs to which the circuit is to react in a specified manner must be explicitly recorded.

Furthermore, not every sequential circuit can be specified by a finite list of I/O pairs. For example, any circuit which is to produce a repeated sequence of outputs in response to inputs until a certain series of inputs is applied cannot be specified by a single I/O sequence.

A more general formulation of the above problem is the inability of single sequence specifications to describe cyclic behaviors of indeterminate duration.

An input/response description method will next be described which overcomes the above difficulties. It will be shown that this extension of the previously described method increases only slightly the effort required to translate I/O specifications into flow table form.

B. Sequential Circuit Specification Using Input/Output Sequences

The sequence may be used as a building block to describe more complex circuit behavior. The first I/O pair of a sequence will be called the _head_ of the sequence, and the last specification the _tail_. At some point in the I/O description of a sequential circuit, it may be desirable to indicate that one of two or more alternate sequences will be followed, depending on the next circuit input. At such a _branch point_, the sequence previously under development is terminated and the heads of the alternate sequences follow its tail.

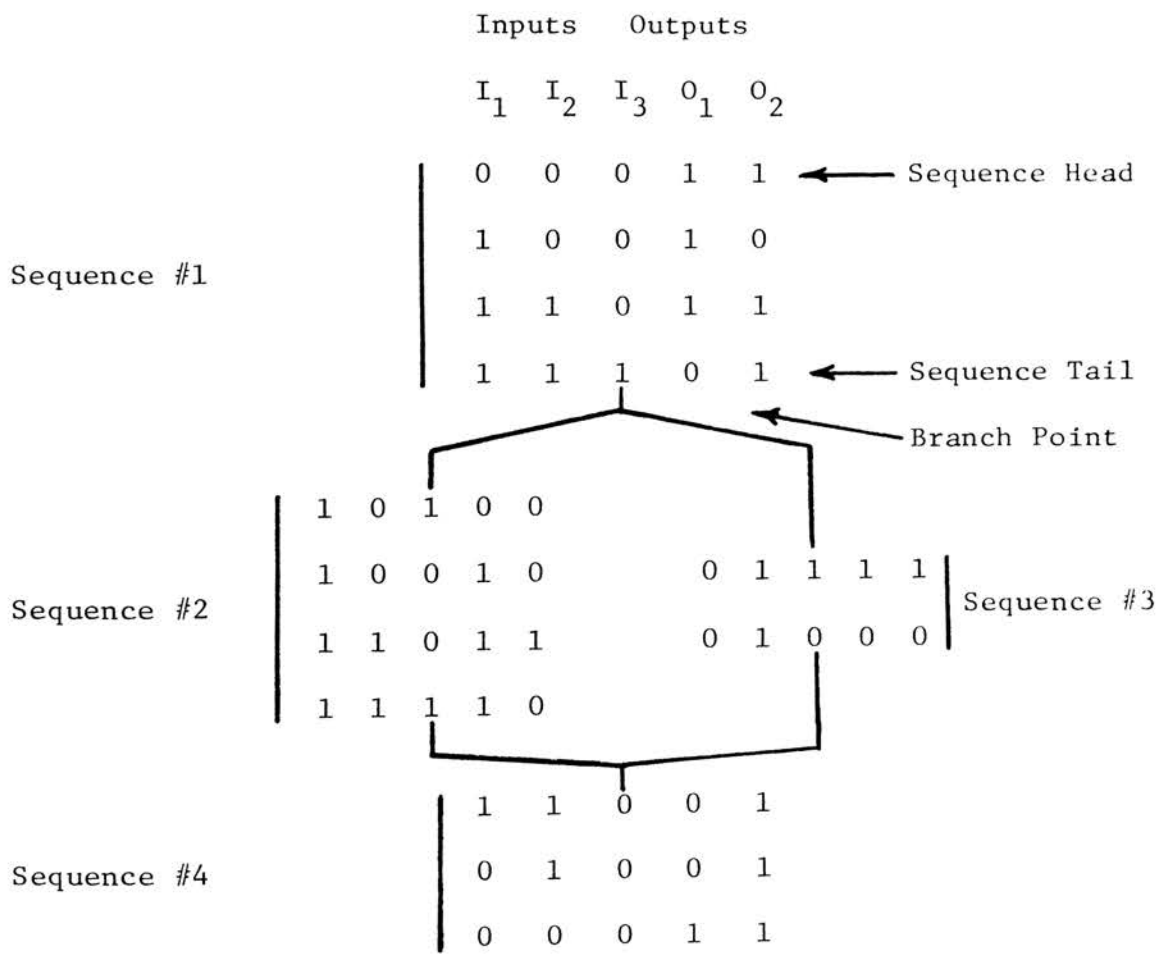
Since the sequences are often developed and recorded serially, it is convenient to introduce the "FOLLOWS" note. This device is used to record, for appropriate sequence heads, the labels associated with preceding sequence tails. Note that a sequence may FOLLOW more than one tail. Conversely, more than one sequence may FOLLOW a tail specification (or a group of them), so long as each head input state is not the head of another of the following sequences.

Figure 3 illustrates the terminology introduced above by showing the sequence representation of a circuit described with a series of I/O pairs.

It is sometimes inconvenient to use "FOLLOWS" notation to describe sequence relationships. For example, the tail of sequence four of figure 3 may be FOLLOWed by HEAD1, but at the time sequence one is recorded the preceding tail name may not be known. A "GO TO" note is provided to simplify such cases. The GO TO instruction, applied to sequence tail specifications, merely lists the labels of the following sequence heads.

Certain other notation conveniences for recording sequences are also adopted. A "*" or "-" in any position indicates that an input or output line is unspecified for an I/O pair. A blank in any position indicates that the value of the corresponding line has not changed; the last specified value for the variable thus replaces the blank. The latter feature eliminates the needless reproduction of long strings of unchanging variables.

The possibility of leaving some variables unspecified complicates the problem of detecting improper sequences: no specification may require or imply a change in output without a change in input. Two tests have been devised to detect improper sequences.



a. Desired response to inputs

Label	Specification	Notes
HEAD1:	0 0 0 1 1	
	1 0 0 1 0	
	1 1 0 1 1	
TAIL1:	1 1 1 0 1	
HEAD2:	1 0 1 0 0	FOLLOWS TAIL1
	1 0 0 1 0	
	1 1 0 1 1	
TAIL2:	1 1 1 1 0	
HEAD3:	0 1 1 1 1	FOLLOWS TAIL1
TAIL3:	0 1 0 0 0	
HEAD4:	1 1 0 0 1	FOLLOWS TAIL2, TAIL3
	0 1 0 0 1	
TAIL4:	0 0 0 1 1	

b. Sequence specification

Figure 3. A Typical Sequence Specification.

Let the first I/O specification W_1 be composed of input state N_1 and circuit response (output) O_1; likewise the second specification is denoted W_2, composed of N_2 and O_2. W_2 may properly follow W_1 if:

1. At least one input variable specified in <u>both</u> N_1 and N_2 is 1 in one case and 0 in the other.
2. If 1) is not satisfied and W_1 is not a sequence tail, then all of the following must be satisfied: a) Each variable specified in N_1 must correspond to either an identical specified value or a don't care in N_2; b) Each variable not specified in N_1 must also be unspecified in N_2; c) All output variables specified in O_2 must be specified in O_1 and both must have a common value; d) Output values not specified in O_2 may correspond to either specified or unspecified variables in O_1.

Rule 2 applies only to input state transitions within a sequence, and reduces to the requirement that W_1 and W_2 be indistinguishable.

Consider the possibly improper sequence (4 inputs, two outputs)

W_1	0*01	11
W_2	0101	00,

which fails Test 1. Assuming W_1 is not a sequence tail, application of Test 2a indicates this is an improper sequence--for input state 0101, the specification requires outputs of first 11 then 00. For the pair

W_1	0101	00
W_2	0*01	11,

Test 1 fails. Tests 2a and 2b are satisfied, but Test 2c indicates that this is also an improper specification. For input 0101, the output is required to be 00 then 11. All parts of Test 2 are, however, satisfied by

W_1	0101	00
	0*01	*0

Another potential source of difficulty under the proposed description method is the need for unique labels on sequence heads and tails. In order to simplify modification of previously recorded sequences (as design progresses), each I/O specification should have a unique name.

The beginning of a new sequence is implied by a "FOLLOWS" notation. Likewise, a "GO TO" note indicates the end of a sequence. It is not, however, necessary to use a "FOLLOWS" at the start, and a "GO TO" at the end of each sequence. The note "BEGIN" has been adopted to indicate the start of a new sequence; end of a sequence is indicated simply by "END". Sequence heads and tails noted in this manner are assumed to have predecessors and successors which are specified elsewhere.

These notation conventions do not cover the case of a sequence which is implicitly begun or terminated by an <u>explicit</u> reference to, respectively, end of the previous sequence, or beginning of a following sequence. In situations where no notation explicitly indicates a sequence begins or ends, a "FOLLOWS" or "GO TO" instruction referring to the preceding or next sequence is assumed by default. Figure 4 shows a thirteen pair sequence which illustrates the descriptive method presented here.

	LABEL	I_1	I_2	I_3	O_1	NOTES
Sequence 1	ONE	0	*	*	1	BEGIN
	TWO	1	*	*	1	
2	THREE	0	1	1	1	(implied sequence end)
3	FOUR	0	0	1	0	FOLLOWS TWO
	FIVE		1	1	0	FOLLOWS TWO, THREE
	SIX		0	1	1	
	SEVEN		1	1	1	GO TO ONE
4	EIGHT	0	1	0	1	FOLLOWS TWO
5	NINE	0	0	0	0	FOLLOWS TWO, EIGHT
	TEN		1	0	0	
	ELEVEN		0	0	0	
	TWELVE		1	0	0	
	THIRTEEN	0	0	0	1	GO TO ONE

Figure 4. Sequence Description Example B.

It will next be shown that this type of sequential circuit specification may easily be converted into conventional flow table representation.

C. Conversion to Flow Table Form

The procedure previously described for converting a sequence to flow table form requires little modification for use under the present scheme. Each input/output pair corresponds to a row of the flow table. Stable state entries (with the specified outputs) appear in all columns corresponding to the input state specification. Thus n unspecified input line values result in 2^n stable state entries in the appropriate row.

Unstable next-state entries are placed in the preceding row for each stable state. Since the preceding flow table row represents the last specification in the sequence, the sequence head stable states do not have any unstable next-state entries leading to them.

Consecutive I/O specifications to which proper sequence rule #2 applies are a special case. If there is no stable state in a column of the row W_1, the next-state entry for the lower I/O pair W_2 must be copied into the preceding row position.

Figure 5 shows the flow table segment which exhibits behavior specified by sequence 5 of figure 4.

I_1	I_2	I_3	O_1	STATE	000	001	010	011	100	101	110	111
0	0	0	0	A	$\textcircled{A}$/0		B					
0	1	0	0	B	C		$\textcircled{B}$/0					
0	0	0	0	C	$\textcircled{C}$/0		D					
0	1	0	0	D	E		$\textcircled{D}$/0					
0	0	0	1	E	$\textcircled{E}$/1							

Figure 5. Translation of a Sequence into a Flow Table Segment.

Sequence relationship data provided by "FOLLOWS" and "GO TO" instructions are conveniently recorded in a Module Flow Table (MFT). Each sequence corresponds to a single row of the MFT. Stable states in the MFT record sequence entry input states (obtained from the head I/O pair specification). Unstable next-state entries in exit (tail) input state columns indicate the next sequence to be followed for various input values. A stable and unstable state entry both in the same row and column indicates that an entry input state is also an exit state. In this case, the unstable entry is simply tagged with a minus sign.

Each sequence is translated to flow table segment form and the MFT is completed. A single flow table description of the sequential circuit is then obtained by concatenating all sequence flow table segments. Unstable next-states corresponding to FOLLOWing sequence entry rows are added to the last (tail) row of each segment and flow table translation is completed. Note that the unstable states of a row of the MFT correspond to the unstable states added to the last row of each segment.

Figure 6 shows the flow table segments which are obtained from the circuit description of figure 4.

	I_1	I_2	I_3	I_4		INPUT STATE							
						000	001	010	011	100	101	110	111
1	0	*	*	1	A	Ⓐ/1	Ⓐ/1	Ⓐ/1	Ⓐ/1	B	B	B	B
	1	*	*	1	B	J	D	H	C	Ⓑ/1	Ⓑ/1	Ⓑ/1	Ⓑ/1
2	0	1	1	1	C		D		Ⓒ/1				
	0	0	1	0	D		Ⓓ/0		E				
3	0	1	1	0	E		F		Ⓔ/0				
	0	0	1	1	F		Ⓕ/1		G				
	0	1	1	1	G	A	A	A	Ⓖ/1				
4	0	1	0	1	H	J		Ⓗ/1					
	0	0	0	0	J	Ⓙ/1		K					
	0	1	0	0	K	L		Ⓚ/0					
5	0	0	0	0	L	Ⓛ/0		M					
	0	1	0	0	M	N		Ⓜ/0					
	0	0	0	1	N	Ⓝ/1	A	A	A				

Figure 6. Sequence Flow Table Segments for Example B.

The Module Flow Table for figure 4 is shown in figure 7; the segments and the MFT have been combined as described into a single flow table shown in figure 8.

I_1	I_2	I_3	O_4		INPUT STATE 000	001	010	011	100	101	110	111
0	*	*	1	1	-,5	-,3	-,4	-,2				
0	1	1	1	2		3		②				
0	0	1	0	3	1	-,1	1	1				
0	1	0	1	4	5		④					
0	0	0	0	5	-,1	1	1	1				

Figure 7. The Module Flow Table for Example B.

INTERNAL STATE	INPUT STATE 000	001	010	011	100	101	110	111
1	①/1	①/1	①/1	①/1	2	2	2	2
2	9	4	8	3	②/1	②/1	②/1	②/1
3		4		③/1				
4		④/0		5				
5		6		⑤/0				
6		⑥/1		7				
7	1	1	1	⑦/1				
8	9		⑧/1					
9	⑨/0		10					
10	11		⑩/0					
11	⑪/0		12					
12	13		⑫/0					
13	⑬/1	1	1	1				

Figure 8. Flow Table Representation of Example B.

D. A sequence Translation Computer Program

A PL/1 program, WAVEFM, has been written which accepts I/O sequential circuit specifications of the type described. The program also incorporates several useful error detection and editing features.

The improper sequence tests have been incorporated; they produce error messages and terminate translation if an improper list of specifications is presented.

Each specification is required to have a unique 4-character label. If the I/O pair name has been previously used, an error message is produced and processing ends.

The program accepts "FOLL XXXX,YYYY,..." as FOLLOWS instructions, where XXXX and YYYY are 4-character labels previously used in the description. The GO TO instruction is identical, except for the mnemonic "GOTO". "BEGN" and "END" mark the beginning and end of sequences, while "STOP" in the label field indicates end of the circuit description.

Although now operating in a batch mode environment, the description and translation methods used by WAVEFM should prove most useful in interactive use by circuit designers. Limited text editing features were included in order to provide some "psuedo-interactive" processing by the present version of the program. Thus a "$<$" causes deletion of a single character immediately to the left of the character deletion symbol and "/" causes the deletion of an entire input record (line).

Another feature incorporated into the program is the capability to provide, on request, an error-free copy of the partial list of circuit specifications and/or the module flow table.

Finally, the program may optionally be requested to prepare a list of all branch points for which action in response to some input is not specified.

E. Extensions and Results

An alternate representation of the desired sequential circuit may be obtained by considering each sequence (or a collection of sequences) as a submachine or module. Each module has one or more entry internal states, and a single exit state. Each module realizes a portion of the sequential behavior required of the desired circuit.

Only one module at a time responds to input stimulii. The active module is selected by a control module which responds to inputs as well as "module exit state" signals. It is interesting to note that the previously developed MFT closely resembles a flow table description of the control module. Figure 9 shows a block diagram of one modular realization of example B of this chapter.

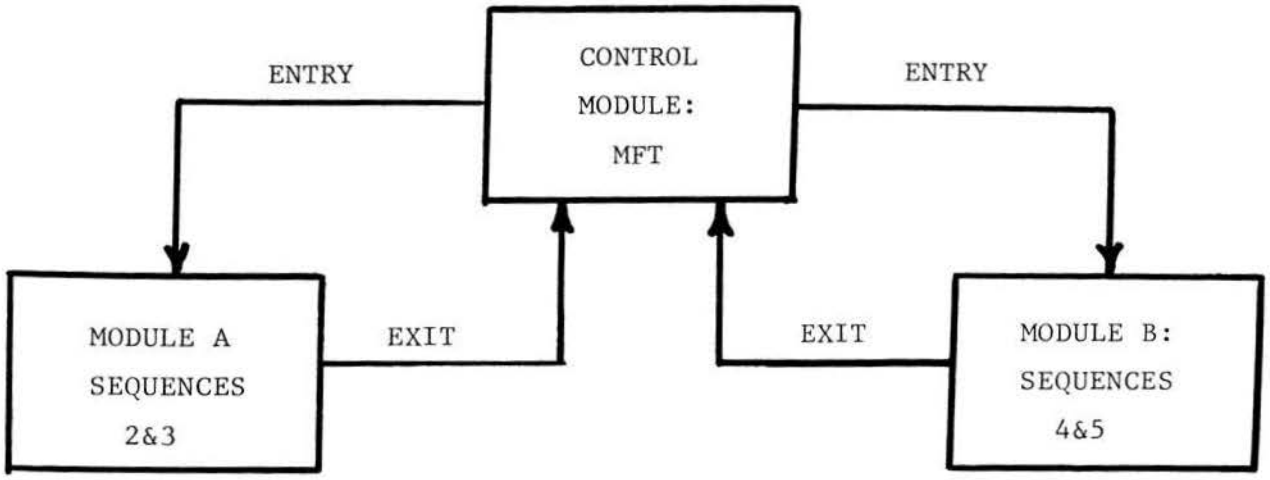


Figure 9. Modular Organization of Example B.

A flow table translation program for modularly organized circuits has not been written, because the design automation system related to the programming effort presently synthesizes only single flow tables. Little difficulty should be encountered in adopting the translation algorithm to the modular case. Investigation will, however, be required to develop heuristics for determining modular partitioning. The decomposition of a very large sequential circuit description into several smaller ones is a powerful synthesis aid.

The single flow table translation program has been applied to only a few long specification lists. A typical description involved 22 sequences containing a total of 158 four-input, three-output specifications. The 158 row by sixteen column flow table was produced in only 65 seconds.

The flow tables produced by the methods described in this chapter generally can be greatly simplified. Indeed, if these tables are to be used to actually synthesize circuits, it is important to reduce (if possible) the number of internal states (rows) in the flow table. Chapter III is devoted to the problem of simplifying very large flow tables.

III. REDUCTION OF LARGE INCOMPLETELY SPECIFIED FLOW TABLES

Flow tables often contain more internal states than are required to specify the desired circuit behavior. In such cases it is advantageous to reduce the flow table to more compact form, for synthesis costs increase with flow table size, and circuit complexity is roughly proportional to flow table size. The simplification of completely specified flow tables is much less difficult than that for incompletely specified tables.[16] Since practical asynchronous sequential circuit descriptions are seldom formulated as completely specified tables, the more general, incompletely specified case is treated here.

A. Background

The following defintiions are useful in this chapter. If a sequence of inputs is applied to flow table P when it is initially in internal state r, then this sequence is said to be applicable to r if the state of the flow table is specified after each input, except possibly the last. Thus, when an applicable sequence of inputs is applied, no unspecified next-state entries are encountered, except possibly after the final input. Unspecified flow table entries are taken to imply that behavior of the machine ceases to be of interest once the unspecified state is entered. Stable states which have no output specified imply that circuit outputs will be ignored so long as the output remains unspecified.

Two output states are comparable if they are identical whenever both are specified. Two internal states s_a and s_b are compatible if they yield comparable output sequences for all possible input sequences. It is clear that s_a and s_b are compatable only if for each input state their outputs are identical whenever both are specified, and their next-state entries are compatable whenever both next-states are specified.

A compatibility class C is a set of internal states which are all pairwise compatible. A set of states Q is implied by a set of states R if, for all inputs, Q is the set of all specified next-state entries for R. As used herein, this definition will be slightly modified when applied to compatibility class candidates. C_a implies C_{bi} if for each input state I_i either all next-state entries are in C_a or all next-states are in an implied class C_{bi}. Using the latter concept of implication, it may be seen that a single class C_a may imply one or more classes C_{bi}. Since each of the C_{bi} may in turn imply other classes, an implication chain may be formed. All compatibility class candidates C_a which imply others are termed conditionally compatible, since the implied classes of the chain must be subsets of known compatibility classes before compatibility can be established for C_a.

A maximal compatible (or maximum compatibility class) is one which is not contained in any other compatibility class.

A set of compatibility classes covers a flow table if every state of the flow table is contained in one or more classes of the set.

A set of compatibility classes is closed if for every input state the set of next-states implied by each class C_i in the set is contained in at least one of the classes of the set.

It can easily be shown that a reduced flow table which covers the original one may be formed from a closed set of compatibility classes which contains each state of the original table. Each row of the reduced table corresponds to a compatibility class.

Paull and Unger, in a classic paper[2], presented an algorithm for obtaining maximum compatibility classes for incompletely specified flow tables. An implication table is formed, recording pairwise compatibility. Figure 10 illustrates the implication table for flow table A of figure 1. Dash entries indicate state pairs which are

5	X				
4	1,4	5,6			
3	-	5,6 3,6	1,4 3,6		
2	X	X	1,4 3,6 2,6	1,4 2,6	
1	2,6	1,4 2,5	X	2,6 3,6	3,6 1,4
	6	5	4	3	2

Figure 10. Implication Table for Flow Table A.

compatible, while X's indicate pairs which are incompatible. State pairs which are conditionally compatible have the implied pairs entered in the appropriate cell. Conditional compatibility chains are systematically examined and the final implication table contains no implied pair entries which are not conditionally compatible.

A systematic method is described for obtaining the set of all maximal compatibles from the implication table. The Paull-Unger maximal compatible algorithm is well known and will not be presented here.

Paull and Unger were unable to present a systematic procedure (other than complete enumeration) for obtaining a minimum closed collection of compatibility classes. They pointed out that an upper bound on the number of states in the minimized flow table is the number of maximal compatibles--but this number is usually greater than the number of rows in the original flow table. Several suggestions are offered for manual minimization of flow tables of up to fifteen rows.

Grasselli and Luccio have published an algorithm[3] which solves the cover and closure problem without resorting to complete enumeration. The incompatibility table is formed and used to produce the set of all maximal compatibles.

A procedure is described for obtaining the collection of all compatibility classes which can be used to construct a minimal flow table. This set of compatibility classes is much smaller than the set of all maximal compatibility classes and their included subclasses. A significant reduction in intermediate data is achieved, but only through increased computation.

The final step in Grasselli's flow table minimization procedure is the construction and reduction of a cover and closure (CC) table, used to select closed sets of compatibility classes which cover the original flow table. The CC table is very similar to a prime implicant table, but is somewhat more difficult to reduce.

Kella[5] recently developed a procedure for finding all minimal covers for an incompletely specified flow table. The generation of all prime compatibility classes is avoided by generating reduced machines by recursively adding new states, rather than starting with a set of compatibility classes which cover the original flow table. Only reduced machines with the minimum number of states are considered as new states are added, thus avoiding non-minimal reductions.

A sequential machine M_a is a partial machine of machine M_b if the state table of M_a is included in but less than M_b. Every transition in M_b not covered by M_a is considered unspecified in M_b. A reduced machine $\overline{M}_b$ is based on a reduced machine $\overline{M}_a$ if $\overline{M}_a$ is a partial machine of $\overline{M}_b$.

Kella's procedure begins by finding all state pairs of the original flow table which are pairwise incompatible. An algorithm is then presented for finding all reduced machines M_a for the first (i+1) rows of the original table M, which are based on the M_a of M(i). Thus the consideration of each row of M in turn leads to the production of all minimal flow tables. The procedure involves finding all maximum compatibility classes for the partial table M(i+1) which include state s_{i+1}; using the list of incompatible states this process is much less difficult than finding the set of all maximum compatibility classes for the original machine.

The three algorithms outlined above have been examined in some detail in order to emphasize the amount of effort required to minimize very large flow tables. For an N-row table, the amount of data and effort required to produce pairwise compatibility or incompatibility information is in general proportional to N^2 for large tables. The amount of computation involved in generating maximum compatibility classes is rather problem dependent, but is roughly proportional to N^6.* Effort expended in developing prime compatibility classes and reducing a CC table also increases approximately exponentially.

The Kella algorithm,while in general requiring less effort than the Grasselli-Luccio procedure, is still far too cumbersome to economically reduce extremely large flow tables.

None of the methods outlined are well suited to automated flow table reduction. All require that an extremely large amount of intermediate data be preserved. Another disadvantage of these techniques is that they produce <u>all</u> minimal flow tables; for large tables it becomes impractical to produce more than a single reduced flow table. Furthermore, experience with other switching theory minimization problems--Boolean functions and asynchronous state assignments, to name just two--has shown that minimization becomes prohibitively costly for very large problems. Although they do not in general produce minimal results, it is clear that economical flow table reduction procedures must simplify rather than minimize large tables.

---

*There are $P = (N^2-N)/2$ row-pair comparisons to be made in forming a compatibility or incompatibility table. Suppose that $1-1/r$ of the row pairs are incompatible. Consider only attempting to form three member compatibility (or incompatibility) classes: three two-sets must be examined for each three-set. There are $R = P/r$ possible two-sets. The number of pair comparison look-ups required is $W = \binom{R}{3} = \binom{P/r}{3} = \binom{(N^2-N)/2r}{3}$, which is proportional to N^6. For example, with $N = 10$ and $r = 4$, $W = 155$; however, for $N = 100$ and $r = 4$, $W = 3\times10^8$. This very rapid increase in effort required to produce maximal compatible generator routines for minimum variable state assignments described in (10 and 17). Also see Chapter IV.

One of the least complicated simplification procedures is merging. Two flow table states may be merged[18] if their next-state entries are the same state whenever both are specified. The state resulting from the merger has a stable state or output specification wherever either of the original states had a stable state or an output specification. Merging thus does not remove redundant stable states; however if there are no redundant stable states, merging produces a minimal flow table. Although merging usually prevents a reduction of large flow tables to minimal form, it is based on a simple relationship between rows which is easily detected.

Two rows of a flow table are equivalent unless in some flow table column a) their outputs are specified to be different, b) the output or next-state of one row is specified and the other is not, or c) the next-state entries of the two rows are not equivalent[19]. Only one of the two or more equivalent rows need be included in a simplified flow table.

B. A Flow Table Simplification Heuristic

The operating speed or amount of effort required by a large flow table simplification procedure is related to the simplicity of the state relationships detected. It is also affected by the volume of intermediate data which is required to be generated and evaluated. Conversely, the amount of simplification achieved (compared to minimal reduction) is in general improved by detecting complex compatibility relationships and using large amounts of intermediate data.

The algorithm presented below is intended to rapidly produce a simplified--but in general non-minimal--flow table. The table to be simplified is assumed to be incompletely specified, with many next-state entries unspecified. The simplification method is independent of flow table source; the method described in Chapter II might, for example, be employed to produce such tables. The procedure is designed to be most economical when applied to extremely large (up to several hundred state) flow tables, and is intended primarily for automated design applications.

Two important considerations affect the design of a flow table simplification heuristic. First, the procedure must not require exhaustive computations or comparisons. The effort expended in economically simplifying large flow tables must be literally orders of magnitude less than that characteristic of known minimization procedures.

Digital computer main memory size limitations restrict the volume of data immediately accessible to a simplification program. (Secondary storage is uneconomical for frequently accessed data). The generation of massive blocks of intermediate data is also expensive. Thus a modest amount of data should be utilized by the successful flow table simplification heuristic.

These two constraints have led to the adoption of a simple strategy: only a single set of compatibility classes, representing the reduced table, is generated. Cover is insured by insisting that each state of the original machine be a member of one and only one compatibility class. Closure is preserved by continuously updating next-state and output specifications for the compatibility classes; current closure requirements thus reduce to satisfying compatibility requirements for the partially reduced machine.

The partial machine next-state entries are stored in a twodimensional array wherein each row is reserved for a compatibility class, and columns correspond to input states. A Boolean matrix is utilized to store output states associated with stable states. A tag number is associated with each flow table state; if zero, the state is either a single element compatibility class or has not yet been added to the reduced machine. If the tag is negative, the state represents a compatibility class containing two or more elements (states). A positive tag points to the state number (in the reduced machine) which corresponds to the compatibility class containing the row in question; positive tags thus map original machine states into compatibility classes, and eventually into states of the reduced machine.

	I_1	I_2	I_3	I_4
1	①/000	①/111	2	*
2	1	②/010	②/011	3
3	8	9	*	③/101
4	*	2	④/011	3
5	1	⑤/111	4	*

1	1	1	2	0
2	1	2	2	3
3	8	9	0	3
4	0	2	4	3
5	1	5	4	0

Next States

000	111			-1
	010	011		-1
			101	0
		011		+2
	111			+1

Output States   Tags

Figure 11. Flow Table and Corresponding Representation.

Next-state zero indicates that the flow table entry is not specified. Row 4 has been combined with row 2, and the resulting compatibility class has been stored in row 2; likewise, rows 1 and 5 have been combined and the resulting class is in row 1.

Each state of the original flow table is considered in turn. To reduce the number of row-pair comparisons performed, row i is compared only with compatibility classes--or rows--in the limited range $(i-p) \leq i \leq (i+q)$. Because of this 'look-ahead' provision in the range of comparison for row i, the current status of state (i+q) is used. Thus prior to examining state (i+q) from the original table, the tag numbers must be used to map next-state entries for original row (i+q) into compatibility class references if appropriate.*

The limited flow table examination range employed here may also be visualized as a "window" which moves down the flow table. Only rows currently exposed in the window are used in flow table simplification.

To minimize the amount of intermediate data, only four simple types of row-pair compatibility test are utilized; this arrangement also improves the operating speed of the simplification procedure drastically.

Consider row i from the unsimplified flow table as it is being added to the reduced table. An attempt is first made to add the row

*This action is actually performed for rows (q+1) and on--the first q rows of the original table 'prime' the reduction procedure and require no updating.

to a compatibility class within the examination range having a negative tag (implying two or more original states in the class). Since the compatibility class is represented by its resulting flow table row, i can be added to class j if it is compatible with (compatibility class) flow table row j.

Two flow table rows are compatible, written $x \sim y$, if for each input state having specified next-state in both rows, 1) both next-state entries are identical or 2) both next-state entries are stable states and the output states agree whenever both are specified.*

Row i is immediately added to the first compatibility class j with which it is compatible. The resulting compatibility class has a stable state and output specification wherever either of the previous rows was stable. If both were stable for some input, only the outputs are combined. For convenience, the new class is placed in the same location as the old compatibility class; this practice generally reduces the number of next-state entries which must be changed, since next-state i is likely to appear less frequently than j. The tag for row i is set equal to j, and known (i.e., within the range "window") next-state entries corresponding to stable states i are changed to j.

Next, an attempt is made to add each lower ($k > j$) compatibility class to the new class containing state i. Any classes which can be

---

*This definition is much more restrictive than that usually encountered in the literature. A third condition, that the next-states themselves be compatible, has been discarded in order to develop an economical simplification heuristic. All compatibility classes developed under the restricted definition also satisfy the more general case.

added to the new class j are included immediately by updating the appropriate tag, next-state and output entries.

Finally, the new class j is checked for compatibility with any single member classes--rows which have previously been found incompatible with all others in the known segment of the reduced table.

The new compatibility class expansion procedure causes compatibility classes which may contain many original table rows to grow quite rapidly; this is advantageous because it quickly decreases the size of the partially reduced table and thus reduces the number of row pair comparisons performed in each step.

If a newly considered flow table row is incompatible with all known compatibility classes (i.e., those in range) with two or more elements, an attempt is made to combine that row with each known single element compatibility class. These classes correspond to rows of the original flow table which have been found incompatible with all known rows. If a single row j is discovered to be compatible with i, a new compatibility class is formed and recorded in the old compatibility class position j as outlined above; the remaining single element classes are also checked for compatibility with new class j, as in the previous new class case.

Figure 12 illustrates the formation of a new compatibility class. Row 6 of the original table is added to the partially reduced table composed of classes 1,2, and 3.

Row 6 is incompatible with classes 1 and 2 which represent two or more rows of the original table. Row 6 is conditionally compatible with row 3. With a look-ahead factor of 3, rows 7,8, and 9 are then considered. Rows 6 and 7 are conditionally compatible; then row 8 is discovered to be compatible with row 6. A new two element class is then formed in row 8. The tag and next-state entry modifications which occur are shown in Figure 13.

	I_1	I_2	I_3	I_4	Tag	
1	$\textcircled{1}$/000	$\textcircled{1}$/111	2	*	-1	
2	1	$\textcircled{2}$/010	$\textcircled{2}$/011	3	-1	
3	8	9	*	$\textcircled{3}$/101	0	
4	*	2	$\textcircled{4}$/011	3	+2	Partially Reduced
5	1	$\textcircled{5}$/111	4	*	+1	$\uparrow$
i=6	$\textcircled{6}$/111	$\textcircled{6}$/000	7	*	0	
7	*	1	$\textcircled{7}$/000	10	0	
j=8	$\textcircled{8}$/111	*	7	3	0	
9	6	$\textcircled{9}$/000	*	3	0	

Figure 12. Discovery of a New Compatibility Class.

	I_1	I_2	I_3	I_4	Tag
1	$\textcircled{1}$/000	$\textcircled{1}$/111	2	*	-1
2	1	$\textcircled{2}$/010	$\textcircled{2}$/011	3	-1
3	8	9	*	$\textcircled{3}$/101	0
4	*	2	$\textcircled{4}$/011	3	+2
5	1	$\textcircled{5}$/111	4	*	+1
i=6	$\textcircled{6}$/111	$\textcircled{6}$/000	7	*	+8
7	*	1	$\textcircled{7}$/100	10	0
j=8	$\textcircled{8}$/111	$\textcircled{8}$/000	7	3	-1
k=9	8	$\textcircled{9}$/000	*	3	0

Figure 13. Formation of the New Class

An attempt is then made to add remaining classes or rows to the new class formed in row 8. Row 9 is found to be compatible with class 8 and is thus included in the new class, as shown in figure 14.

	I_1	I_2	I_3	I_4	Tag
1	$\textcircled{1}$/000	$\textcircled{1}$/111	2	*	-1
2	1	$\textcircled{2}$/010	$\textcircled{2}$/011	3	-1
3	8	8	*	$\textcircled{3}$/101	0
4	*	2	$\textcircled{4}$/011	3	+2
5	1	$\textcircled{5}$/111	4	*	+1
6	$\textcircled{6}$/111	$\textcircled{6}$/000	7	*	+8
7	*	1	$\textcircled{7}$/100	10	0
j=8	$\textcircled{8}$/111	$\textcircled{8}$/000	7	3	-1
9	8	$\textcircled{9}$/000	*	3	+8

Figure 14. Addition of a Row to the New Class.

It has been found that for partially reduced incompletely specified flow tables, row pair (i,j) is sometimes the only implicant for class pair (m,n). In this case, the compatibility of pair (i,j) implies that of pair (m,n). However, this situation does not occur frequently enough to justify rechecking the compatibility of each row pair after formation of each new compatibility class. To do so would increase manyfold the amount of effort expended in flow table reduction.

It can be shown, however, that in general only a small fraction of row pairs need be rechecked. Furthermore, these pairs can be easily located during the process of next-state entry updating after formation of the new compatibility class (i,j).

Theorem: If row pair (k,j) is an implicant of pair (m,n) then both i and j must have stable states under some input state(s), and for at least one of these inputs, both i and j must appear as explicit next-state entries in rows m and n.

Proof: If (m,n) implies (i,j) then i and j must be next-state entries under at least one input state of pair (m,n). Normal mode operation requires that transitions lead directly to stable states, so both i and j must be stable for the given input.

It is clear that the above theorem dramatically reduces the amount of rechecking which needs to be done after compatibility class formation. Rechecking does, however represent a significant increase in computational effort, and should be further justified.

First consider two relatively small compatibility classes i and j. In a large table, it is quite likely that the number of unstable next-state entries leading to them will be small. Rechecking in this case is inexpensive, especially since the number of stable states per row may be small, further reducing the likelihood of both being stable in the same column.

If on the other hand i and j are large compatibility classes having many stable state columns, rechecking may involve a large number of row pairs and thus become less desirable.

In the formation of new compatibility classes described above, at least one of the constituents of the new class is always a single row from the original flow table, i. Rechecking is performed after formation of new classes resulting from the construction of class (i,j). If rechecking discovers compatible state pairs (m,n), they are immediately combined, but further rechecking based on these "secondary" new classes is not performed.*

Figure 15 shows the reduced flow table resulting from the simplification illustrated in figure 14. Notice that as new class 8

	I_1	I_2	I_3	I_4	Tag	Recheck
1	$\textcircled{1}$/000	$\textcircled{1}$/111	2	*	-1	0
2	1	$\textcircled{2}$/010	$\textcircled{2}$/011	3	-1	0
3	8	8	*	$\textcircled{3}$/101	0	1
7	*	1	$\textcircled{7}$/100	10	0	0
8	$\textcircled{8}$/111	$\textcircled{8}$/000	7	3	-1	1

Figure 15. A Recheck Opportunity.

was constructed, the recheck flag for row 3 was set due to rows 8 and 9 having stable states under I_2, and row 3 having a next-state entry leading to the new stable state.

Rechecking pair (3,8) results in the formation of a new class (3,8) which is placed in row 3. Figure 16 shows the flow table segment after rechecking is completed.

---

*Experimental simplification of large randomly redundant tables has shown that locating implicants of secondary new classes, although costly, results in little if any increase in overall flow table simplification.

	I_1	I_2	I_3	I_4	Tag
1	①/000	①/111	2	*	-1
2	1	②/010	②/011	3	-1
3	8	8	*	③/101	+8
7	*	1	⑦/100	10	0
8	⑧/111	⑧/000	7	⑧/101	-1

Figure 16. Flow Table Segment After Rechecking.

A row i from the original table is not considered further unless, after the processes described, it is found to be incompatible with all known (in range) classes. Since the amount of reduction achieved may be significantly decreased by such rows, another attempt is made to find compatibility classes containing i.

A single implication chain consists of a collection of state pairs such that each pair of states (excluding perhaps the last) is conditionally compatible and implies only the next pair in the chain.

Figure 17 shows a partial flow table containing a single implication chain.

Since the generation and use of implication data is expensive in terms of both storage and computation, only single implication chains are used in the flow table simplification heuristic. Additional constraints restrict the consideration of implication relations to those situations most likely to produce economical simplification.

	I_1	I_2	I_3
1	①/1	6	4
2	②/1	6	5
3	1	③/1	③/0
4	1	3	④/1
5	1	6	⑤/1
6	2	⑥/1	3

$$\begin{tikzcd}
(1,2) \arrow[dr] & & \\
& (4,5) \arrow[dr] & \\
& & (3,6) \arrow[uull, bend left]
\end{tikzcd}$$

Figure 17. Partial Flow Table and Single Implication Chain.

As has already been implied, a row pair (i,j) is used as the first element in an implication chain only if 1) one of the two states is incompatible with all known states, and 2) only a single pair of states (p,q) is implied by (i,j). These pairs are detected as the pair compatibility process previously described is executed: as state j is considered for compatibility with state i, if i and j are conditionally compatible and imply only a single pair of states (implicants) p and q, j is marked. If i is not found compatible with any known state, then an attempt is made to build a single implication chain based on (i,j) implies (p,q).

Implication chains which may be used to find valid compatibility classes terminate in several ways. If the final implicant pair p and q are unconditionally compatible, then all pairs in the chain are compatible. If a pair already in the chain is the only implicant of the last pair (i.e., the chain closes on itself) then all pairs in the chain are compatible.

A chain building attempt fails if some chain implicant pair (p,q) has two or more implicants, if a pair of implied states are incompatible, or if the chain length exceeds some threshold. The latter has experimentally been shown to be unimportant;the restricted implications considered cause almost all chains to be very short. A fourth type of chain failure closely resembles a closed chain: if one state but <u>not both</u> of an implicant pair has previously appeared in a chain, the chain fails.

If a chain is successfully completed, compatibility classes are calculated in reverse order, beginning with the last class added to the chain. Rechecking may be performed after this operation is completed for <u>all</u> implicant pairs. The advisability of rechecking here is highly problem dependent but usually yields little additional simplification--at a relatively high cost.

Figure 18 illustrates the simplification obtained by reducing the single implication chain shown in figure 17.

After the process described above has been completed for each original flow table row, the flow table must be reorganized to eliminate the rows with positive tags and to complete the updating of next state entries.

Each row is considered in turn, until all rows have been processed. If row i has a positive tag (indicating inclusion of i in a compatibility class stored elsewhere) the flow table portion consisting of zero or negatively tagged rows <u>above</u> the "known" part of the table (with the window of the known rows based on row i) is

	I_1	I_2	I_3	Tag
1	①/1	3	4	-1
2	②/1	6	5	+1
3	1	③/1	③/0	-1
4	1	3	④/1	-1
5	1	6	⑤/1	+4
6	2	⑥/1	3	+3

Figure 18. Flow Table Simplification Using Single Implication Chains.

examined for unstable entries valued i. These next-state entries are changed to the appropriate state number of the class containing row i. A search is then made to find a row $j > i$ with a zero or negative tag to "fill" the space occupied by the eliminated row i. If such a row is found, next-state entries are changed to reflect the relocation of row j to position i. If no rows are available to fill state i, the flow table reorganization process is complete.

Figure 19 illustrates the flow table reorganization for the reduced table shown in figure 18. Notice that the second row,

	I_1	I_2	I_3	Tag
1	$\textcircled{1}/1$	3	$\overset{2}{\cancel{4}}$	-1
2	$\textcircled{2}/1$	6	5	+1
3	1	$\textcircled{3}/1$	$\textcircled{3}/0$	-1
4	1	3	$\textcircled{4}/1$	-1
5	1	6	$\textcircled{5}/1$	+4
6	2	$\textcircled{6}/1$	3	+3

Before Reorganization.

	I_1	I_2	I_3
1	$\textcircled{1}/1$	3	2
2	1	3	$\textcircled{2}/1$
3	1	$\textcircled{3}/1$	$\textcircled{3}/0$

Final Form

Figure 19. Reorganization of a Reduced Flow Table.

having a positive tag, is replaced by the lowest row 4 having a negative tag.

The procedure outlined produces excellent simplification if a large portion of the flow table is in the range of consideration. However, the effort implied by such a large range is considerable. Thus, it is recommended that the procedure presented here be applied iteratively using a more economical range. Simplification processing ceases when a simplification yield requirement is not met.

Figure 20 is a brief flow diagram of the flow table simplification heuristic presented here.

C. Programmed Implementation and Results

The flow table simplification heuristic described has been programmed in PL/1. Although the program will not be described in detail, the performance of the programmed procedure illustrates the utility of the simplification heuristic itself. It should be noted that the program was written in a high level language and emphasized algorithm clarity rather than execution efficiency.

Experience gained in several previously developed flow table simplification algorithms led to a program implementation of the procedure containing several minor modifications of the simplification heuristic described here. These changes permitted the evaluation of constraint placed on various phases of the simplification process.

A rather trivial assumption was also made to allow an experimental simplification routine to be developed more rapidly. It was assumed that, as flow table simplification proceeds, enough memory is available to store all of the partially reduced machine. Thus as row i of the original flow table is added to the reduced machine, it

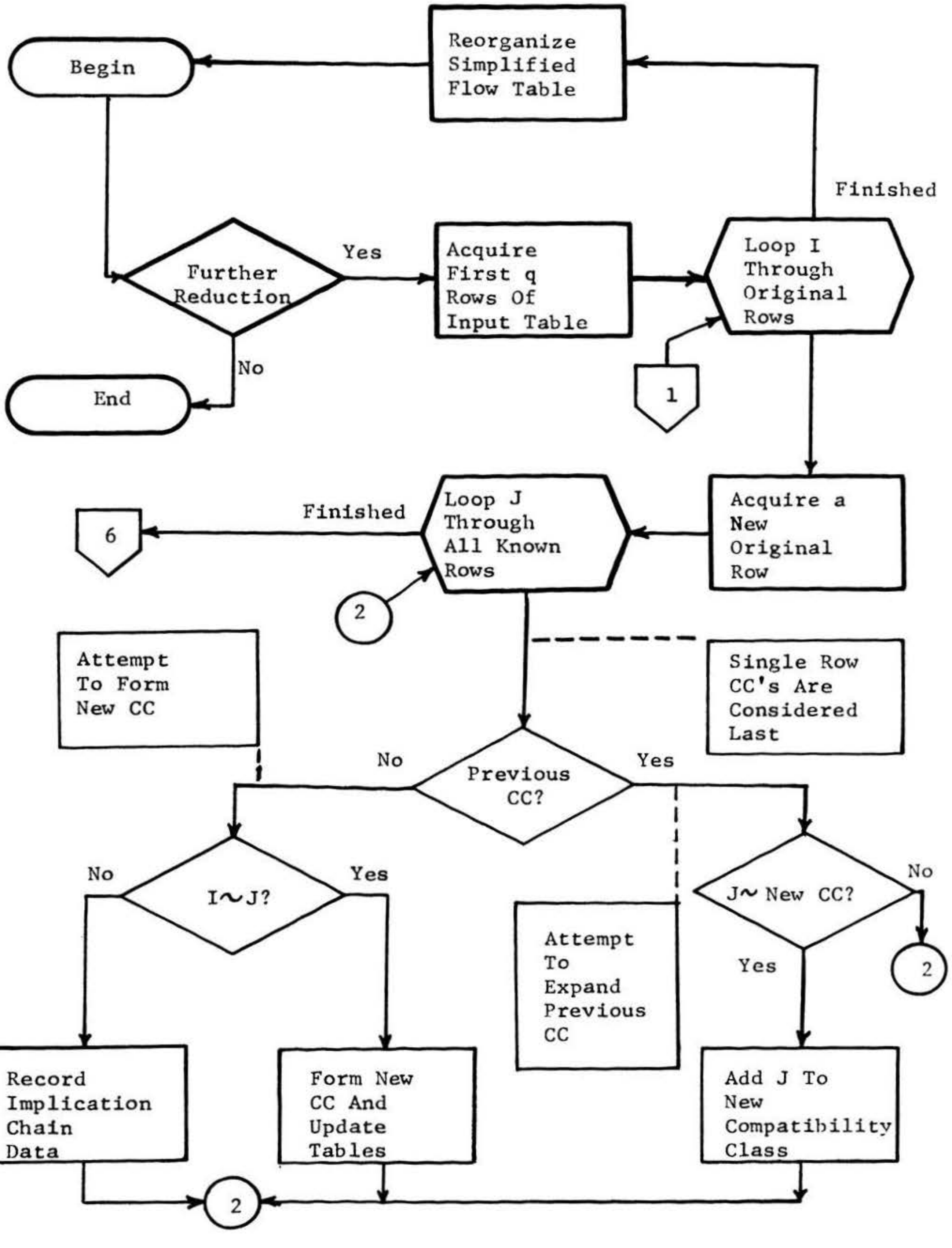

Reorganize Simplified Flow Table
Begin
Finished
Further Reduction
Yes
Acquire First q Rows Of Input Table
Loop I Through Original Rows
No
End
1
Finished
Loop J Through All Known Rows
Acquire a New Original Row
6
2
Attempt To Form New CC
Single Row CC's Are Considered Last
No
Previous CC?
Yes
No
I∼J?
Yes
J∼ New CC?
No
Attempt To Expand Previous CC
Yes
2
Record Implication Chain Data
Form New CC And Update Tables
Add J To New Compatibility Class
2

Figure 20a. The Flow Table Simplification Heuristic.

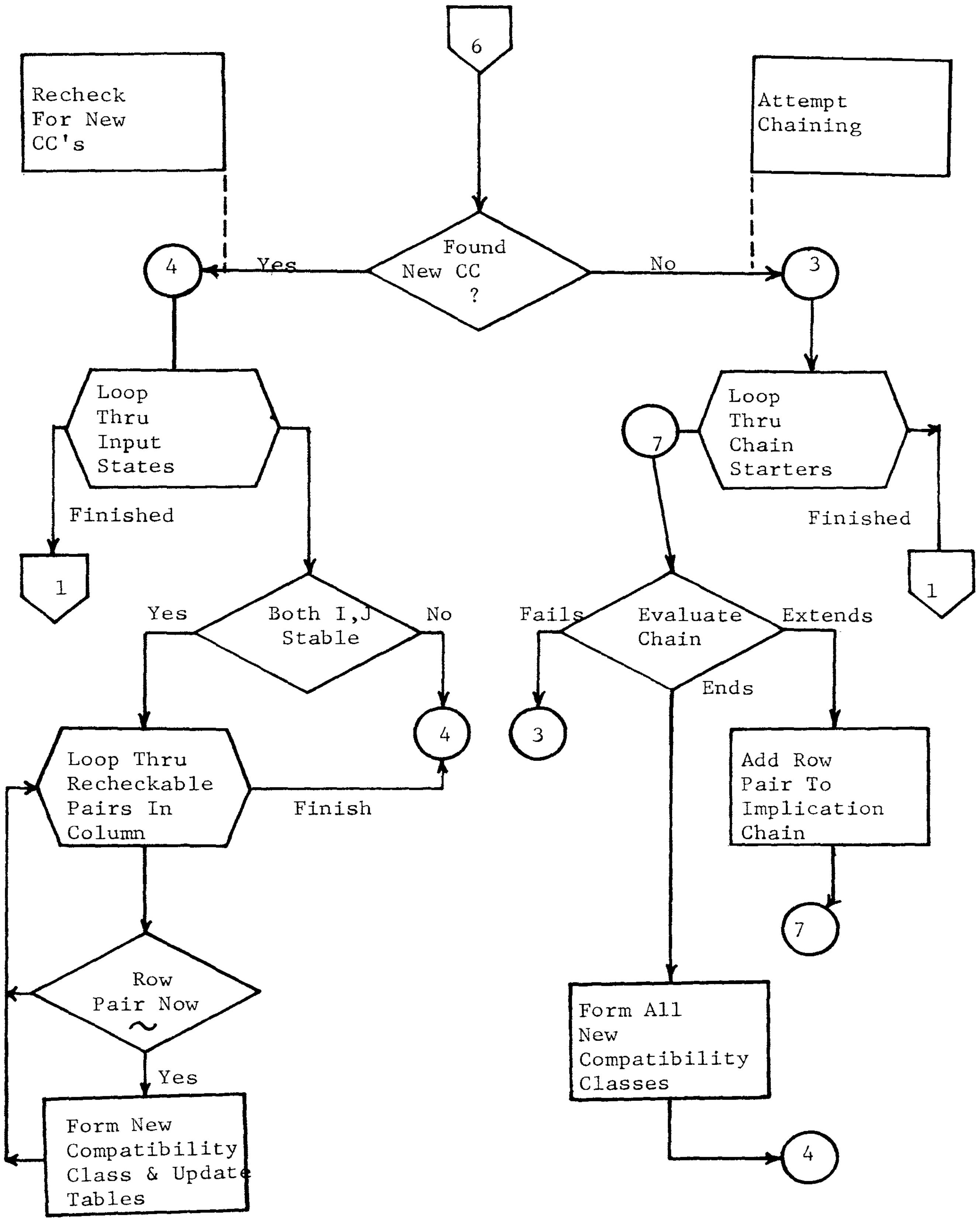


Figure 20b. The Flow Table Simplification Heuristic

(Recheck, Chaining and Reorganization).

is assumed that all compatibility classes containing rows 1 through i-1 are stored in main memory. This programming convenience eliminated the need for a partially reduced flow table segment paging and bookkeeping scheme--which although involving very significant extra programming effort is not technically important.

An interesting experimental modification of the program was a provision for varying the degree of "look ahead" used in the procedure. Although computer time costs have restricted experimentation with this parameter, some preliminary results can be reported. Figure 21 shows a plot of look ahead versus simplification time (using a S/360-50) for a single 193 rows by four column flow table. Also shown on the same graph is the degree of simplification achieved in each case. Although the effects of various degrees of look ahead are highly problem dependent, processing times generally increase as look ahead increases beyond about 10%. The degree of reduction achieved may be less dependent on look ahead, especially for values greater than 10%

The examination of an extremely large number of single implication chains may be undersirable. The programmed simplification procedure thus contained a provision for halting the chain building process after a variable number of chain failures. A variable maximum chain length test was also incorporated (i.e., fail all chains longer than the length limit). Both of these provisions were found to have almost no effect on either the degree of reduction obtained or execution time required. This result is due to the extremely low incidence of long single implication chains in the examples used, and the surprisingly small number of single implication chains discovered.

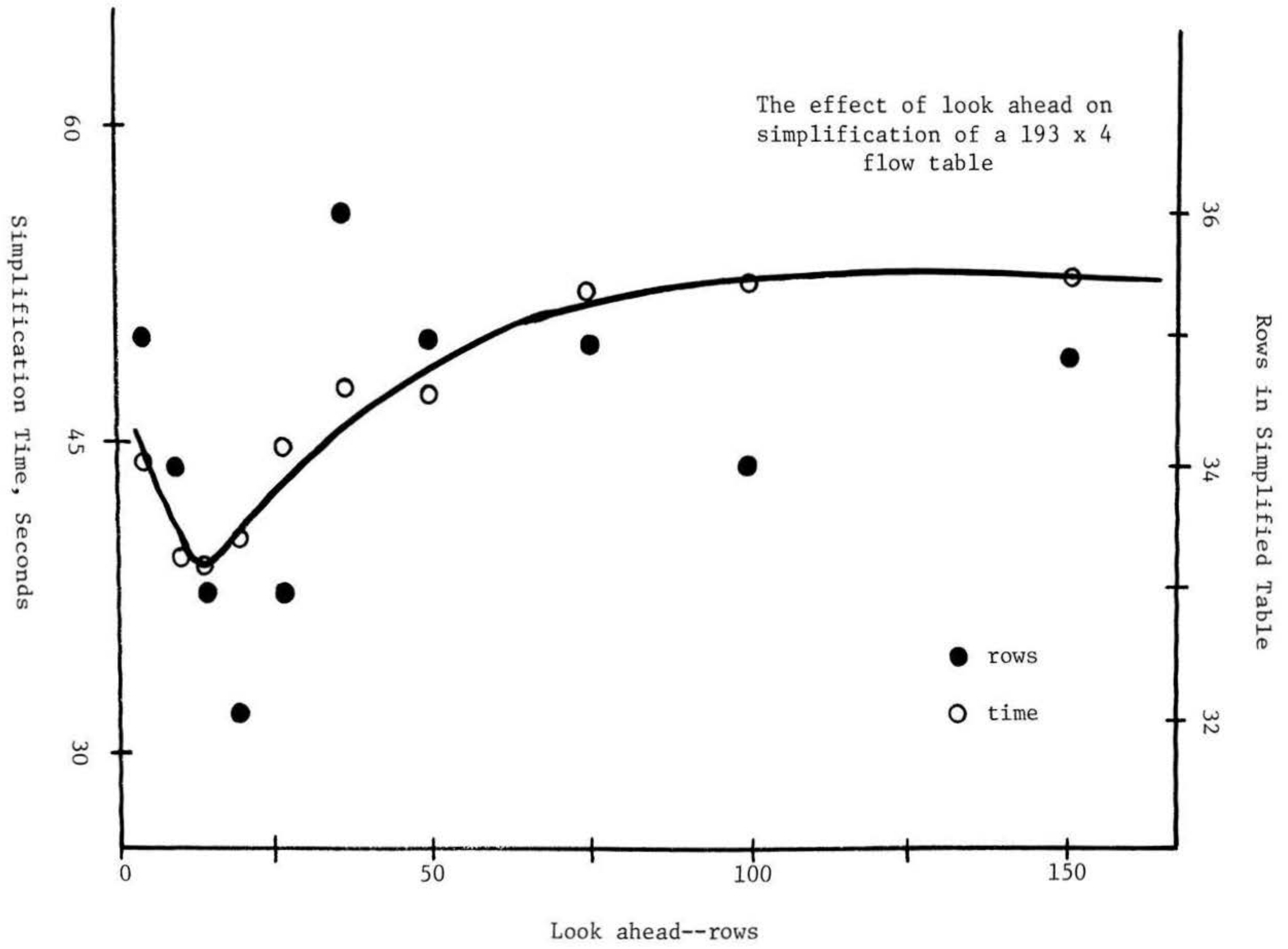


Figure 21. An Example of the Effect of Look Ahead on Flow Table Simplification

The programmed routine also contained an option for suppression of the iterated simplification feature. It was found that the increased simplification obtained was highly problem dependent. In many instances, virtually no simplification was achieved after the first pass. For other tables, significant reduction was obtained for up to three passes. In all cases, the amount of time required to complete a simplification cycle decreased markedly for successive passes.

The recheck performed after formation of new compatibility classes could also be bypassed in the programmed flow table simplification routine. It was found that rechecking contributes significantly to table reduction, especially on the first iteration.

As has been pointed out previously, minimization procedures for the large flow tables considered here are impractical. Thus no attempt has been made to program a flow table minimization algorithm. Still, some means of evaluating the performance of the heuristic is essential.

The evaluation method selected consisted of constructing a completely simplified flow table by insuring that no two rows were compatible. A random number generator was then used to introduce redundant rows having at least one stable state. Other next-state entries in the redundant rows were randomly determined to be specified or unspecified. Finally, the rows of the redundant flow table were randomly reordered, and the table was prepared for simplification.

Several very large flow tables prepared in this manner have been reduced by the simplification heuristic program. Table I summarizes results obtained using these and other examples. Appendix I contains more detailed information on trial flow table reductions, including input and output flow table characteristics, simplification mechanism, etc.

Table I. Simplification of Several Large Flow Tables

Example Number	Number Input	of Simp.	Rows Min.	Number Columns	Simp. Time, Sec.
1	193	32	23	4	40.8
2	75	28	23	4	18
3	115	25	UNK	8	49
4	217	38	23	8	82
5	158	37	UNK	16	93
6	96	26	UNK	16	31.5

The performance of the programmed version of the flow table simplification heuristic illustrates the utility of the method. Although the reduced tables are not minimal, considerable reduction is achieved at very low cost.

The availability of a practical flow table reduction method represents a further step toward the economical synthesis of large asynchronous sequential circuits. Chapter IV examines the problem of constructing state assignments for reduced tables which specify large sequential circuits.

IV. STATE ASSIGNMENTS FOR LARGE ASYNCHRONOUS SEQUENTIAL CIRCUITS

The selection of a satisfactory state assignment for an asynchronous sequential circuit is one of the most difficult tasks in the synthesis procedure. This chapter considers the state assignment problem for circuits operating in the normal fundamental mode and describes a method especially suited to the automated design of very large circuits. Operation of the machine is assumed to be described by a previously simplified flow table.

A. Background

The complexity of the state assignment problem for asynchronous sequential circuits stems from the necessity of avoiding critical races which may cause the machine to malfunction. A critical race exists when, because of the asynchronous nature of state transitions, internal state variables may change values in an order which causes the circuit to reach a final (stable) state other than the desired one. Figure 22 shows a portion of a flow table and a state assignment which contains a critical race. If the circuit is stable in state 1 under input I_2, assume the input then changes to I_1. The desired final state is 3, but if y_2 changes state before y_1, stable state 2 is reached, causing a malfunction.

One method which has been used to avoid critical races is the application of a standard state assignment, i.e. one which does not depend on flow table structure. The number of state variables required for standard state assignments has been the topic of several papers[7,20]. The least number of standard assignment variables for normal, fundamental mode circuits reported to date[20] is $S_0 + \binom{S_0}{2} + \binom{S_0}{3}$, where for an n row flow table, $S_0 = [\mathrm{Log}_2\, n]$.

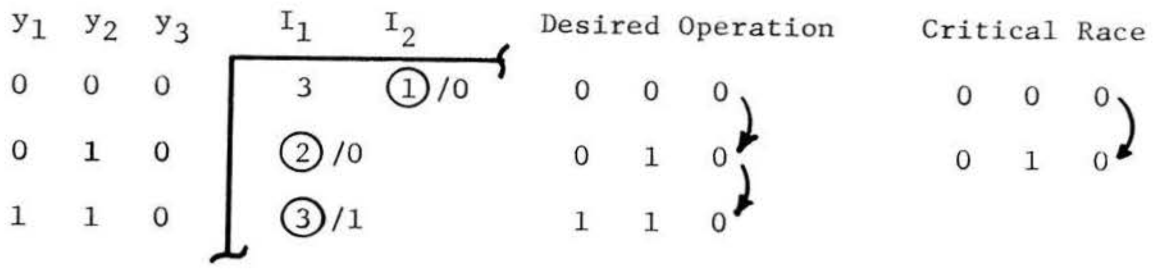


Figure 22. State Assignment Containing a Critical Race.

The complexity (and, to an extent, reliability and cost) of sequential circuit hardware is roughly proportional to the number of internal state variables. Standard assignments frequently contain an unnecessarily large number of variables. Quite often, it can be shown that a given flow table can be satisfactorily coded using a nonstandard assignment with significantly fewer variables than the corresponding standard assignment. Conversely, standard assignments can be generated orders of magnitude more rapidly than critical-race-free nonstandard codes, and thus lower synthesis costs.

Nonstandard state assignments for asynchronous sequential circuits have received much attention in recent years. In an early paper, C.N. Liu showed that a critical-race-free assignment could be formed by combining individual column codes which are themselves critical race free.[7] The column codes are obtained by considering transitions from unstable to stable states within a column.

A k-set is composed of a stable state in a flow table column, together with all states in the column that have unstable next state entries which lead to the stable state. Liu shows that for N stable states in a row, $[\log_2 N]$*variables may be used to form a critical-

*In this paper [x] is used to denote the nearest integer which is greater than or equal to X.

race-free column code. However, for tables with many columns, the number of variables in the state assignment may approach or even exceed that required for a standard assignment.

Tracey later described a method for finding minimum variable assignments for normal, fundamental mode asynchronous sequential circuits.[8] A list of constraints (two block partitions) is constructed according to the following theorem: A row assignment with one state per row can be used for the realization of normal mode flow tables without critical races if and only if for every transition (S_i, S_j): a) if (S_m, S_n) is another transition in the same column, then at least one internal state variable partitions the pair (S_i, S_j) and (S_m, S_n) into separate blocks, b) if S_k is a stable state in the same column, at least one variable partitions (S_i, S_j) and (S_m, S_n) into separater blocks, and c) for $i \neq j$, S_i and S_j are in separate blocks of a state variable partition.

Tracey's state assignment algorithm requires that the partition list be translated into a Boolean matrix; each row represents a constraint and each column corresponds to a flow table row.

Two rows of a Boolean matrix are intersectable if and only if they agree whenever both are specified. A Boolean matrix row may be added to an intersectable R only if that row is intersectable with every element in R. An intersectable which cannot be enlarged is called a maximal compatible. Each maximal compatible may be thought of as a largest possible collection of non-conflicting constraints. A minimum variable code which satisfies each of the constraints thus corresponds to a minimum number of maximal compatibles which cover the constraint list.

Algorithms for finding the maximal compatibles and then selecting a minimum cover of maximal compatibles closely resemble those employed in the flow table and Boolean equation minimization cases.

Tracey noted that for large constraint lists, the effort involved in finding minimum variable codes becomes prohibitive. He therefore presented an algorithm for the near-minimization of large constraint lists.

Both of Tracey's matrix reduction algorithms were incorporated in the asynchronous sequential circuit synthesis system described in (10), (17), and (22). The minimum variable method generally produces satisfactory results for flow tables of up to eight rows by four columns, but consumes excessive computer time for larger tables.

The second, near minimum variable matrix reduction algorithm proposed by Tracey has proven to be quite practical for tables of from 8 rows by 4 columns to about 25 rows by 4 columns. Unfortunately, the constraint matrix for larger flow tables becomes too lengthy for economical reduction even using this procedure.

For such large tables, Tracey described a third method for finding critical-race-free assignments. Again, a constraint list is formed, but pairs of k-sets, rather than transitions are partitioned in each column (each row must also be partitioned from every other row). The resulting Boolean matrix is reduced by one of the two methods mentioned above. Larger flow tables can be coded using this procedure because there are, in general, considerably fewer k-sets than transitions to stable states in a flow table.

The latter Tracey assignment method may, however, become uneconomical for large constraint matrices, because the reduction pro-

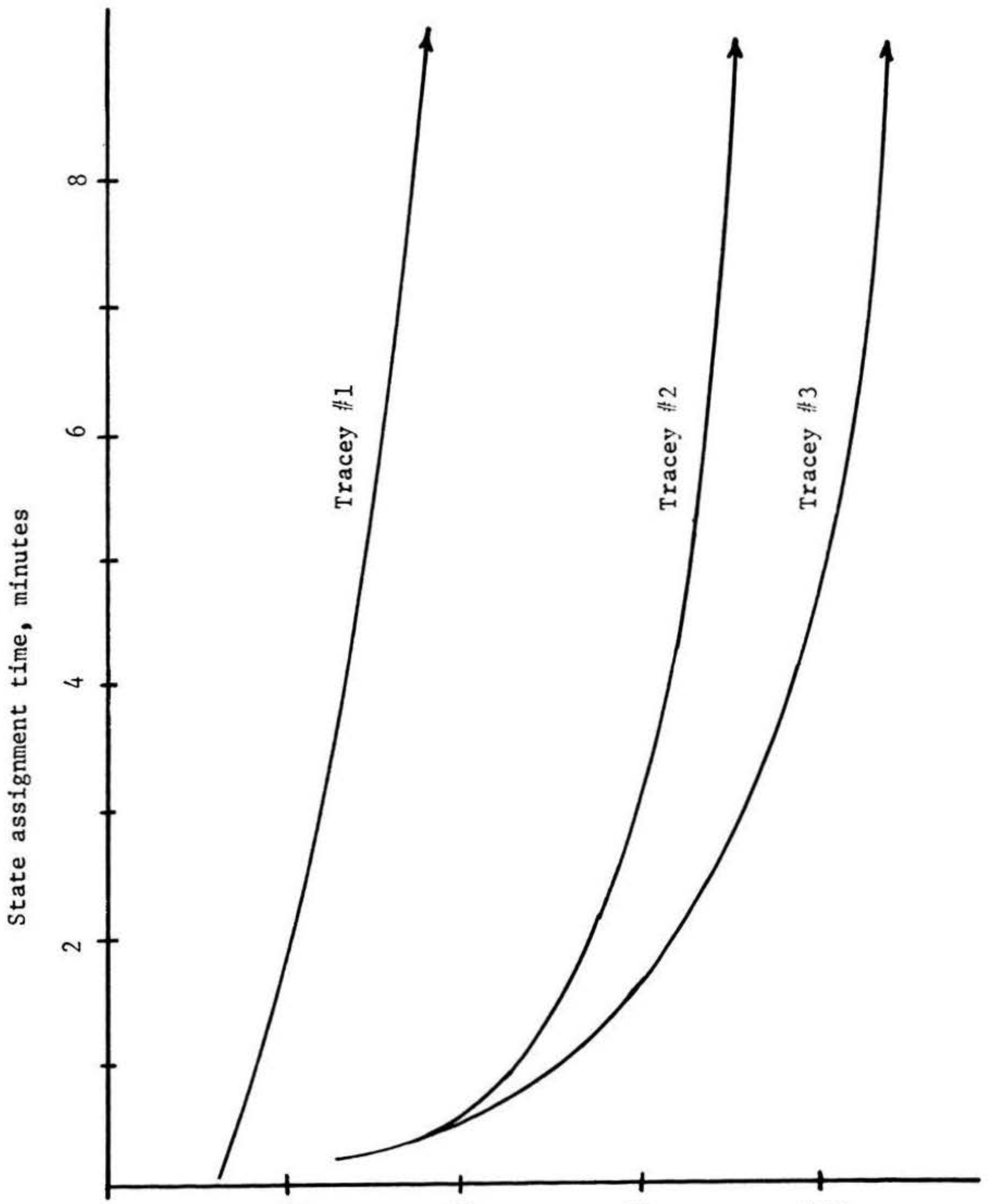


Figure 23. Flow Table Size versus Typical State Assignment Time--

Tracey Methods.

cedures become impractical for matrices exceeding certain sizes. The minimum reduction procedure matrix size limit has been found experimentally to be about 50 rows by 8 to 12 columns.[17] The near-minimum matrix reduction method size limit has been found to be about 200 rows by 30 columns.

Figure 23 shows a comparison of flow table size versus state assignment generation time for programmed versions of the three Tracey state assignment methods.

Because of the exponentially increasing computation times for larger tables, none of the Tracey assignment methods appear to be suitable for flow tables of 20 rows by 8 columns or larger. The remainder of this chapter describes a modification of Tracey's k-set partition assignment algorithm which permits the economical generation of codes for extremely large tables.

B. A Nonstandard State Assignment Procedure for Large Flow Tables

The generation of either transition or k-set partitions is not particularly difficult or time consuming even for large flow tables. As previously pointed out, the difficulty centers around the reduction of very large constraint matrices. A method is described here which avoids reducing large Boolean matrices, thus allowing nonstandard codes to be found for very large flow tables.

The strategy used is a simple one: k-set partition constraints are found. (The use of k-set partitions reduces the number of constraints which must be satisfied.) A constraint list is only allowed to grow to a predetermined size limit, then is partially reduced. This strategy will, in general, produce assignments having at least as many variables as the Tracey methods. However, the proposed method will be shown to be much faster and hence more economical for large circuits.

An example, shown in figures 25, 26, and 27, illustrates the following discussion.

The new state assignment procedure begins by finding k-set partitions for each flow table column. When the partition matrix reaches the size limit, the matrix is partially reduced, yielding state variables and a small number of constraints not satisfied by the variables generated.

K-set partition generation then resumes. However, a k-set partition is not added to the constraint list if it can be satisfied by a previously calculated state variable. The constraint matrix thus contains only those k-set partitions which remain to be satisfied. When this constraint list again reaches the size limit, the partial matrix reduction procedure is repeated.

After all k-set partitions have been found, the state variable and partition lists must be checked to insure that each flow table row is partitioned by some variable from every other flow table row. Any constraints needed to satisfy this requirement are added to the partition matrix, and it is completely reduced.

A flow diagram of the large flow table state assignment procedure is shown in figure 24. The matrix reduction scheme is not detailed since it is identical to Tracey's method two.[21]

The state assignment method outlined above produces codes more economically than the Tracey algorithms because the amount of computation required to reduce a Boolean matrix is much greater for schemes which consider the entire matrix than for methods which re-

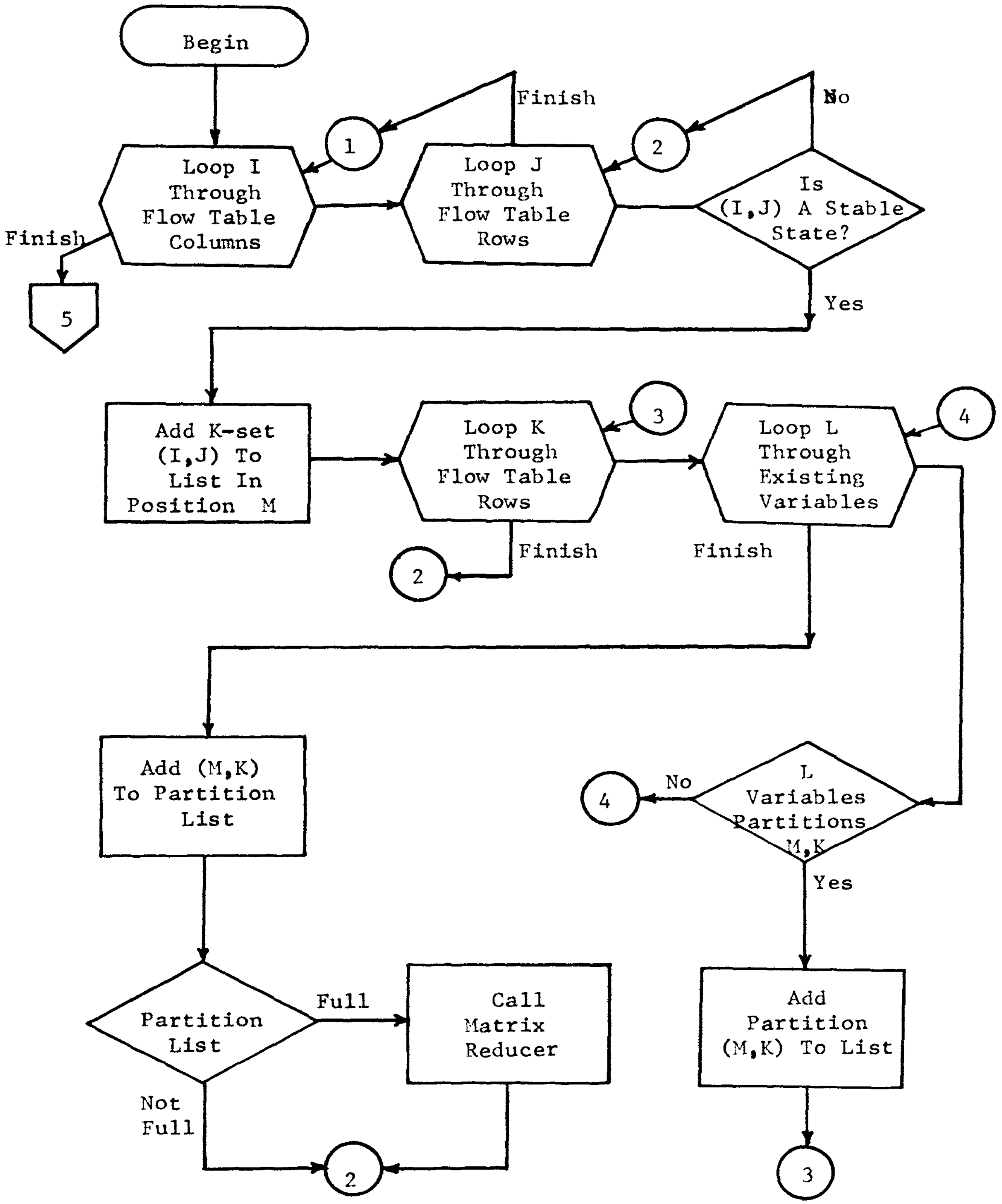

Begin
Loop I Through Flow Table Columns
Finish
5
1
Finish
Loop J Through Flow Table Rows
2
No
Is (I,J) A Stable State?
Yes
Add K-set (I,J) To List In Position M
Loop K Through Flow Table Rows
3
Finish
2
Loop L Through Existing Variables
4
Finish
Add (M,K) To Partition List
L Variables Partitions M,K
No
4
Yes
Partition List
Full
Call Matrix Reducer
Not Full
2
Add Partition (M,K) To List
3

Figure 24a. The Large Flow Table State Assignment Procedure (K-set Partitions).

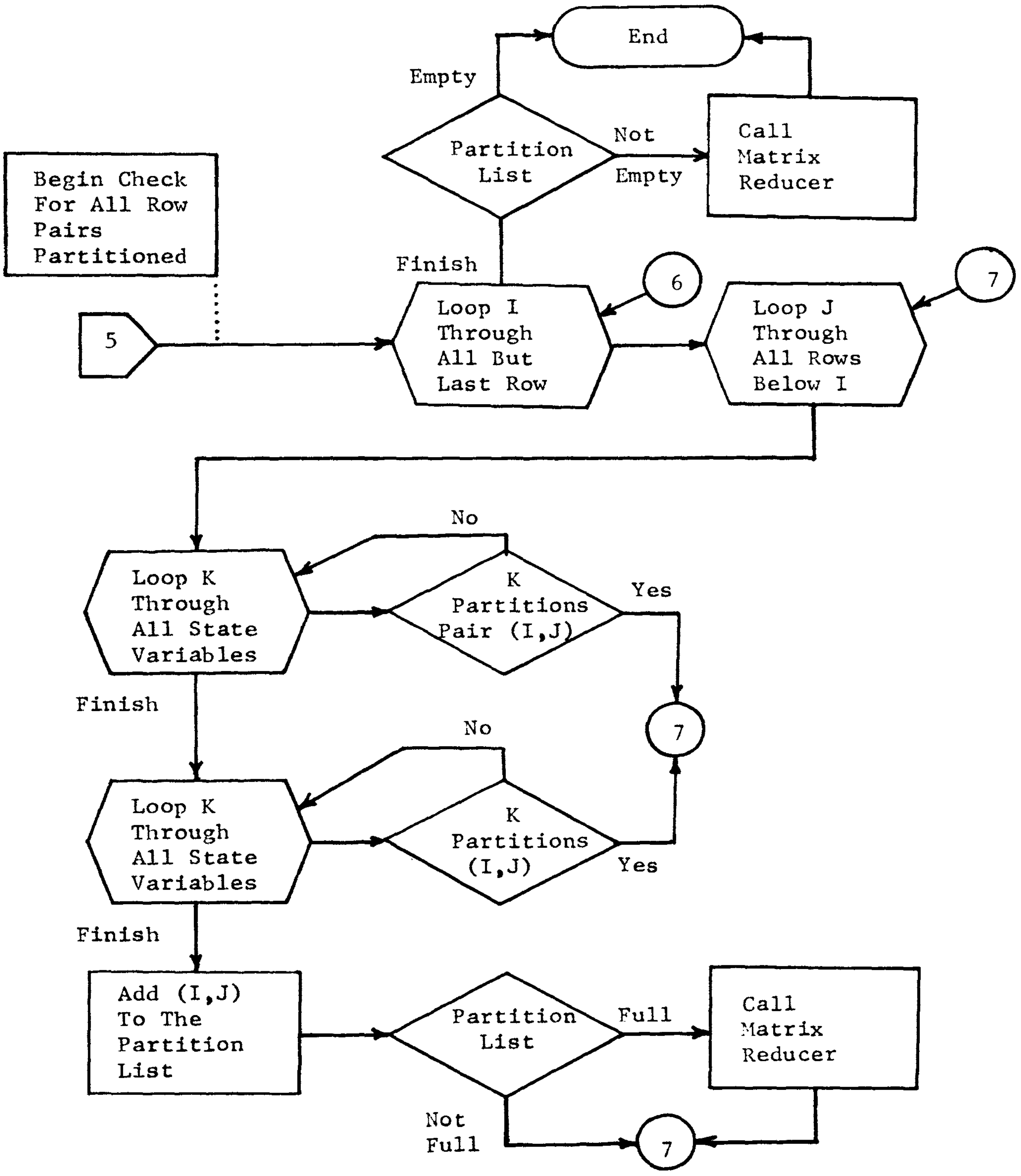

End
Empty
Partition List
Not Empty
Call Matrix Reducer
Begin Check For All Row Pairs Partitioned
Finish
6
7
5
Loop I Through All But Last Row
Loop J Through All Rows Below I
No
Loop K Through All State Variables
K Partitions Pair (I,J)
Yes
Finish
No
7
Loop K Through All State Variables
K Partitions (I,J)
Yes
Finish
Add (I,J) To The Partition List
Partition List
Full
Call Matrix Reducer
Not Full
7

Figure 24b. The Large Flow Table State Assignment Procedure (Row Partitions).

	I_1	I_2	I_3	I_4
1	①/11	①/00	9	11
2	②/01	6	10	12
3	③/00	1	--	11
4	④/10	6	--	12
5	3	⑤/01	⑤/00	12
6	1	⑥/11	9	11
7	3	⑦/10	5	--
8	4	7	⑧/11	11
9	--	5	⑨/01	12
10	--	7	⑩ /10	11
11	2	--	10	⑪ /11
12	4	--	8	⑫ /00

Figure 25. Flow Table D.

duce relatively small matrix segments. It is, for example, much simpler to reduce four matrix segments of 50 rows each than to reduce one 200 row matrix.

C. An Example

Figure 25 shows a 12 row by 4 column flow table which will be used to illustrate the state assignment procedure for large flow tables. It should be noted that the algorithm is not particularly well suited to tables as small as the one considered; this example is presented primarily to illustrate the algorithm.

Figure 26 shows the k-set partition list for flow table D. This table requires 61 transition partitions, compared to the 21 k-set partitions shown.

Column I_1	Column I_2
(1,6;2,11)	(1,3;5,9)
(1,6;3,5,7)	(1,3;2,4,6)
(2,11;3,5,7)	(5,9;2,4,6)
(1,6;4,8,12)	(1,3;7,8,10)
(2,11;4,8,12)	(5,9;7,8,10)
(3,5,7;4,8,12)	(2,4,6;7,8,10)

Column I_3	Column I_4
(5,7;8,12)	(1,3,6,8,10,11;2,4,5,9,12)
(5,7;1,6,9)	
(8,12;1,6,9)	**Rows**
(5,7;2,10,11)	
(8,12;2,10,11)	(4;12)
(1,6,9;2,10,11)	(10;11)

Figure 26. K-set Partition List for Flow Table D.

The state assignment found by reducing the above k-set partition list is shown in figure 27. Tracey's second, near-minimal, matrix reduction technique was used. (For this example the constraint matrix size limits were set at 10 rows maximum and 2 rows minimum).

The preceding example is too small to illustrate many advantages of the segment matrix reduction procedure. Experience with larger matrices has been gained through the use of a programmed version of the large flow table state assignment procedure.

Row Number	K-set Partition Code
1	1111--1
2	00011-0
3	110---1
4	0011010
5	1001010
6	0111--1
7	100001-
8	0010--1
9	1011000
10	-000101
11	000-111
12	0010--0

Figure 27. State Assignment for Flow Table D.

D. A Programmed Implementation of the Procedure

The algorithm outlined above was programmed in PL/1. The program (actually two subroutines) consists of about 325 statements. Both segment maximum and minimum sizes are input parameters.

Subprogram KPI generates k-set partitions and adds them to the constraint matrix if they cannot be satisfied by previously determined state variables. When the list reaches maximum size or all constraints have been generated, routine CODE partially reduces the resulting Boolean matrix.

CODE extracts state variables from the constraint matrix using a procedure closely resembling Tracey's reduction method two[21]. However, when extraction of a state variable causes the constraint list to become shorter than the minimum length, control is returned to KPI and constraint generation continues. The minimum length of the last constraint matrix segment is always zero.

The programmed implementation of the state assignment method was used to investigate the effect of varying the values of the maximum and minimum segment size limits. Although the results appear to be problem dependent, preliminary conclusions can be drawn. Figure 28 shows the relationship between maximum segment size and the amount of computation time required to find a single state assignment for several large flow tables. (More extensive descriptions of these and other state assignment experiments are found in Appendix 2.) Based on the (computer time cost) limited number of experiments performed, it appears that the upper matrix size limit should be about 20 rows. Results appear to be rather insensitive to minimum matrix size limits between 2 and 5 rows.

Figure 29 clearly shows the utility of the state assignment technique described in this chapter (a more extensive list of experiments appears in Appendix 2). The same flow tables have, wherever possible, been coded with Tracey's three methods and the technique developed here. The latter clearly may be used to economically code large tables for which the three Tracey methods are not practical.

The largest flow table for which a state assignment has been generated has 37 rows and 16 columns. A 27 variable state assignment was found for this table in less than 17 minutes. (A standard assignment for the same table would have 41 state variables.)

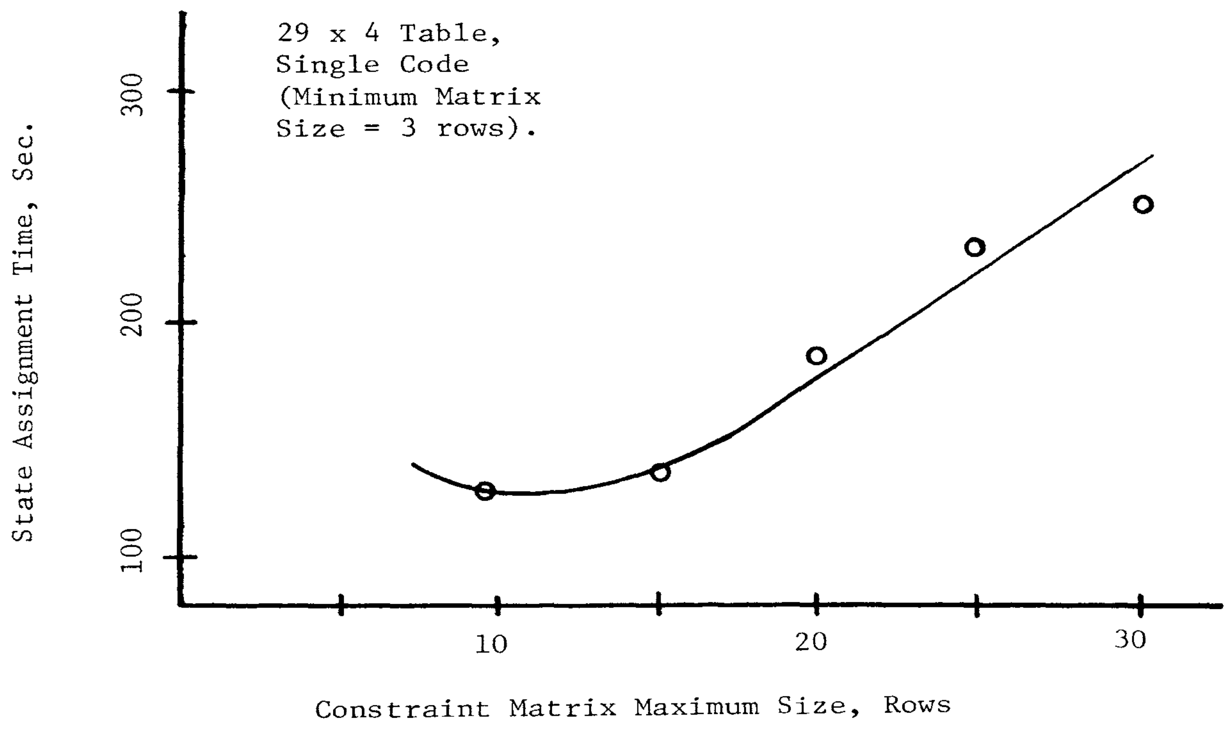


Figure 28. The Effect of Segment Size on Assignment Generation Time.

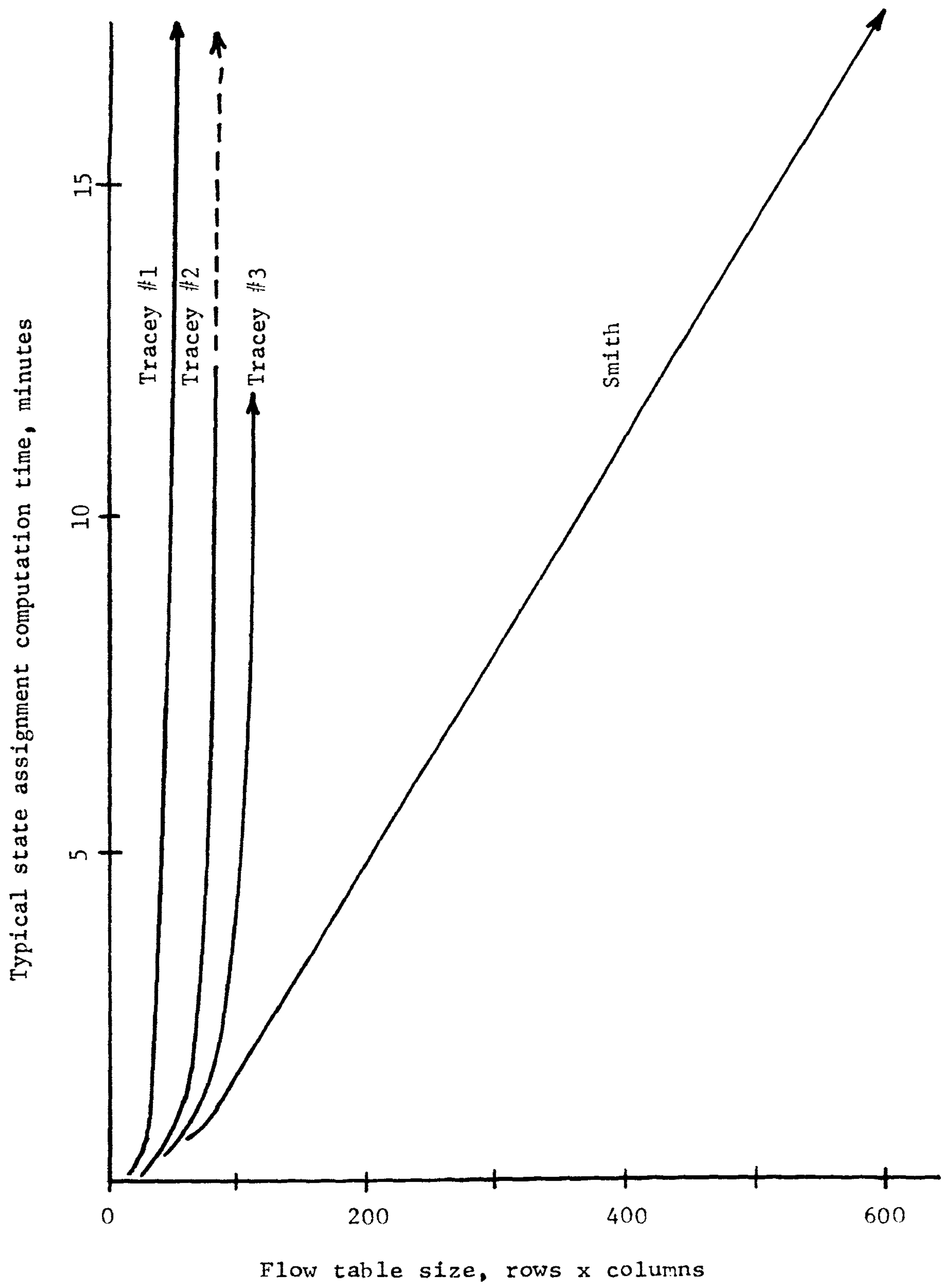


Figure 29. Comparison of Four State Assignment Techniques.

E. Summary

The large flow table state assignment procedure presented here was developed after recognizing that constraint matrix reduction techniques used in well-known algorithms are inadequate for extremely large constraint matrices. The number of constraints (or matrix rows) to be satisfied by a code is first lowered by using k-set partitions. The maximum matrix size which can be efficiently reduced is then used as a matrix segment size limit; matrices which exceed this length are partially reduced by sections.

A programmed realization of the assignment procedure has economically produced state assignments for flow tables far larger than those which could be coded by more nearly minimum variable techniques.

Chapter V briefly describes the characteristics of an experimental automated design system which incorporates the new synthesis techniques described in Chapters II, III, and IV.

V. AN AUTOMATED DESIGN SYSTEM.

The synthesis heuristics developed here have been incorporated in a programmed asynchronous sequential circuit system under development at the University of Missouri - Rolla. Although detailed discussion of the SHADE (Synthesis Heuristics for Automated DEsign) system is not appropriate to this dissertation, a summary of the system will illustrate the utility of the algorithms presented here. SHADE has been used to verify programmed versions of the heuristics and to demonstrate the economical synthesis of large asynchronous sequential circuits.

A. System Overview

Figure 30 shows a flowchart representation of the SHADE (Synthesis Heuristics for Automated DEsign) system which is useful in the discussion which follows.

The SHADE user may enter a problem at any of six steps in the synthesis procedure. Processing then continues through one or more steps following the selected entry point. The available entry points precede these synthesis steps:

1) I/O sequence translation to flow table form.
2) Flow table simplification.
3) Internal state assignment generation.
4) State assignment evaluation.
5) Design equation generation.
6) Boolean equation simplification and hazard removal.

SHADE is capable of processing multiple problems without re-initialization of the system.

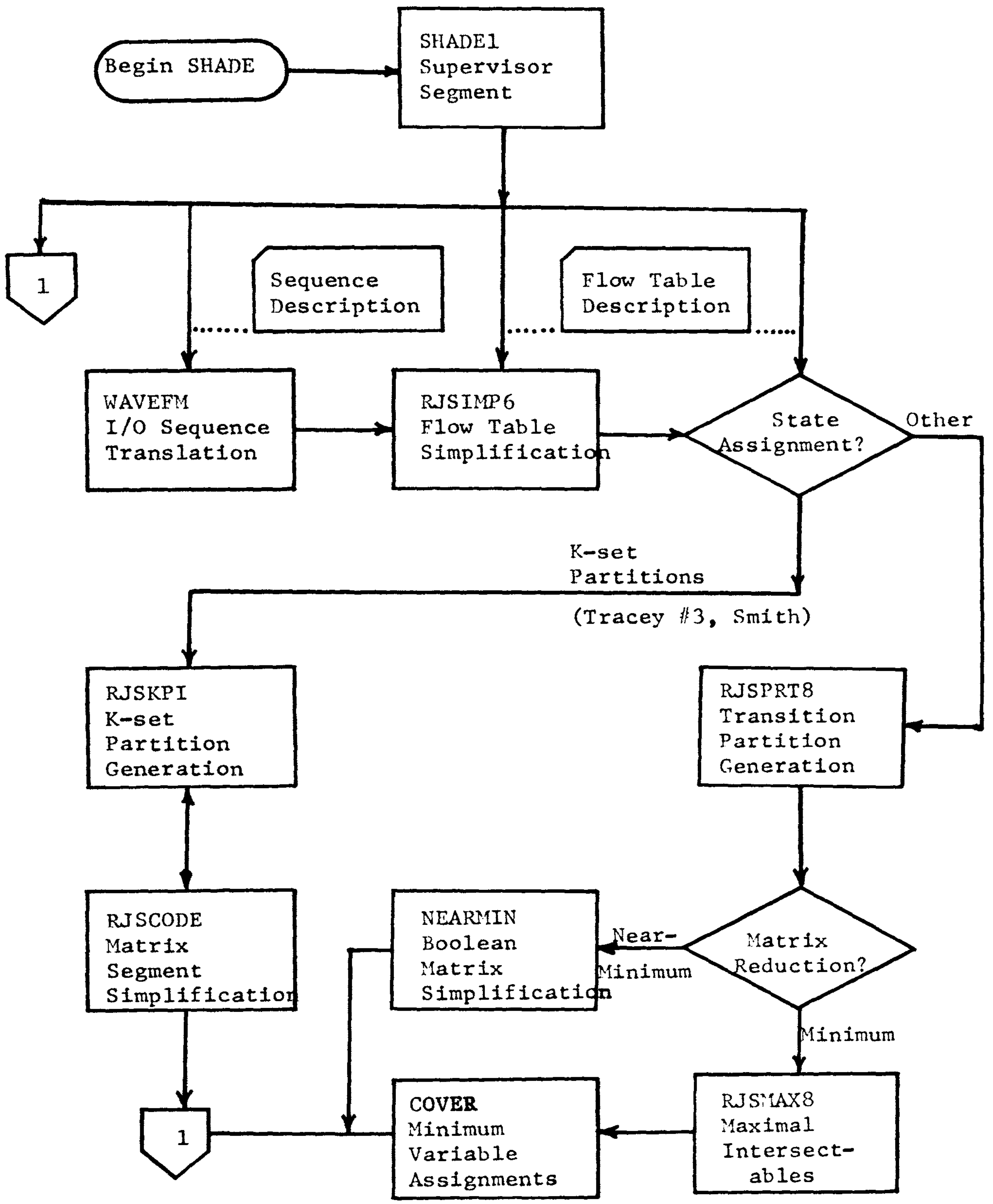

Begin SHADE
SHADE1 Supervisor Segment
1
Sequence Description
Flow Table Description
WAVEFM I/O Sequence Translation
RJSIMP6 Flow Table Simplification
State Assignment?
Other
K-set Partitions
(Tracey #3, Smith)
RJSKPI K-set Partition Generation
RJSPRT8 Transition Partition Generation
RJSCODE Matrix Segment Simplification
NEARMIN Boolean Matrix Simplification
Near-Minimum
Matrix Reduction?
Minimum
1
COVER Minimum Variable Assignments
RJSMAX8 Maximal Intersect-ables

Figure 30a. The SHADE System.

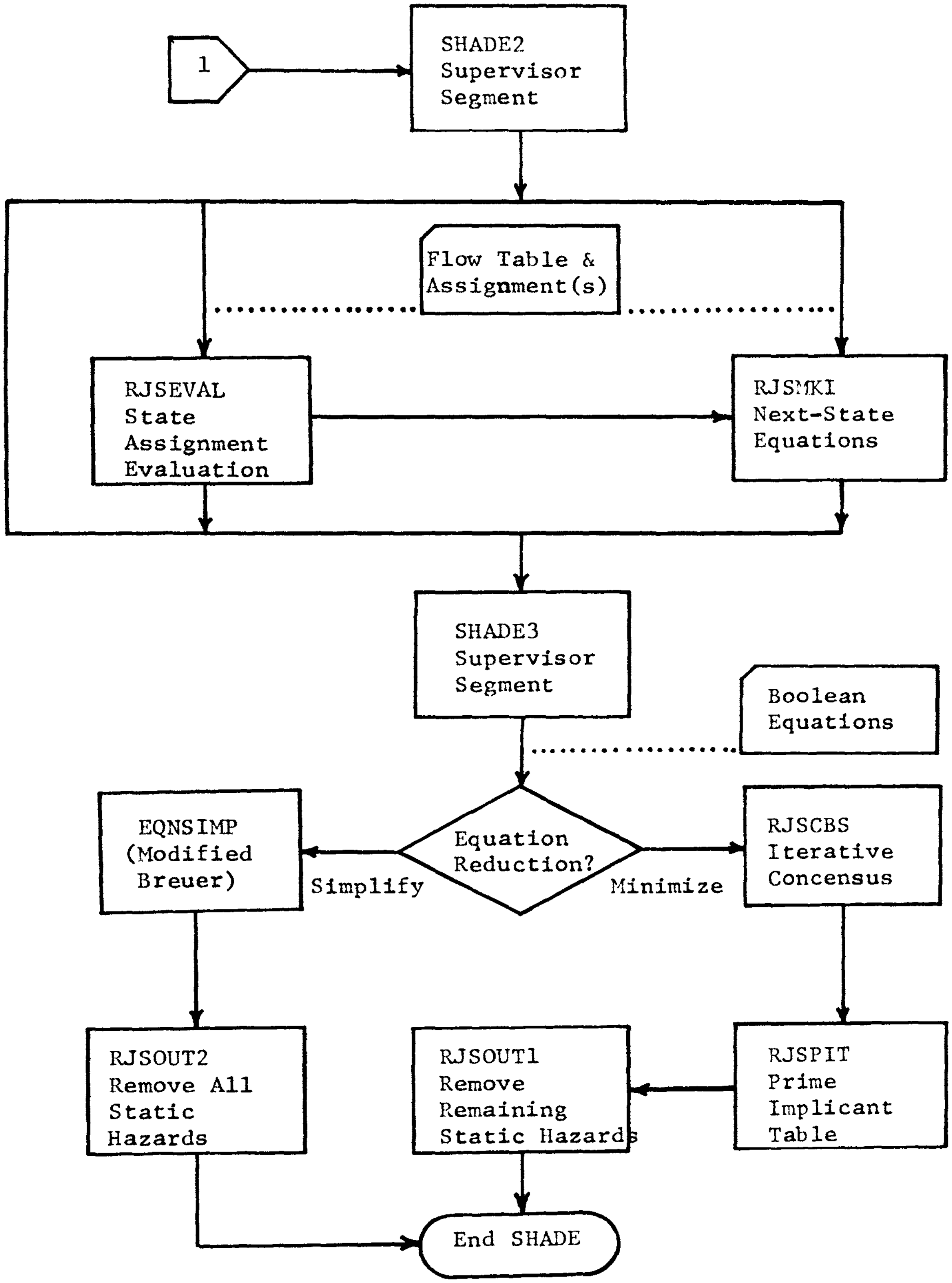

1
SHADE2 Supervisor Segment
Flow Table & Assignment(s)
RJSEVAL State Assignment Evaluation
RJSMKI Next-State Equations
SHADE3 Supervisor Segment
Boolean Equations
Equation Reduction?
Simplify
Minimize
EQNSIMP (Modified Breuer)
RJSCBS Iterative Concensus
RJSOUT2 Remove All Static Hazards
RJSOUT1 Remove Remaining Static Hazards
RJSPIT Prime Implicant Table
End SHADE

Figure 30b. The **SHADE** System (continued).

All data input operations are handled via supervisory programs SHADE1, SHADE2 and SHADE3. These supervisory routines also, at user request, print intermediate results and punch checkpoint data after each synthesis step.

As shown in figure 30, the system user may choose from several modes of operation. For example, by specifying appropriate options and parameters, state assignments may be generated by any of the three Tracey methods, or by using the procedure described in this dissertation. Designs may be generated using all state assignments discovered by a particular method, or only the predicted "best" assignment may be used. Boolean equations may be minimized using either of two methods.

The supervisory programs determine synthesis routine sequencing from a user supplied option card. The supervisor also manages all data passed between synthesis programs.

The I/O sequence translation and flow table simplification programs have been described, so will not be presented here.

B. State Assignment Generation and Evaluation Routines.

As mentioned previously, four state assignment algorithms are available in the SHADE system. The programmed implementation of the method of Chapter IV has been described and will not be reviewed here.

Programmed versions of Tracey's methods 1 and 2 are adaptations of the programs previously developed by Schoeffel and Smith[10,17,22]. These subroutines generate minimum and near-minimum variable state assignments using transition partition constraint lists. These routines have proven to be economical only for small to medium size flow tables (see Appendix 2) and are included in the SHADE system primarily for comparison purposes.

Tracey's method 3, utilizing k-set partition constraint lists and near-minimum matrix reduction, can be shown to be a special case of the procedure described in Chapter IV. In fact, the two methods are identical if the entire constraint matrix is taken as the first matrix segment. In the SHADE system, this is achieved by making the segment size limit (an input parameter) larger than the matrix size. Thus the program described in Chapter IV may be used to generate Tracey method 3 state assignments.

SHADE1, the supervisory section covering internal state assignment generation, is capable of selecting any of the four algorithms if none is specified by the user.

The state assignment evaluation subroutine is based on that described in (13). If two or more codes are generated, the routine attempts to predict which will yield simplest design equations.

C. Design Equation Generation and Reduction.

RJSMKI, the design equation generation program to be used in the SHADE system, is a modification of the one described in (22). It is interesting to note that this algorithm has proven to be economical for quite large flow tables.

Design equations, in the form of Boolean equations for next state variables, are obtained in unsimplified form. Two Boolean equation reduction procedures are available. The first, developed some time ago (10,22), produces minimal equations containing no static hazards. Unfortunately, experience has shown this group of subroutines to be extremely time consuming for large systems of Boolean equations.

In order to improve the performance of the SHADE system, a fast Boolean equation simplification routine has been added. The program, based on an algorithm described by Breuer[14], produces irredundant sums of products using only one pass through the input expression. A static hazard removal routine (under development) completes the processing of large systems of design equations.

D. Conclusions.

The synthesis heuristics developed in this dissertation successfully avoid the prohibitively expensive data storage and computation requirements associated with previously used algorithms. Their incorporation into an automated design system makes possible the synthesis of quite large circuits. The examples cited throughout this dissertation further illustrate the low cost of such an approach.

Investigations in several areas may, however, lead to further performance improvements. New methods for specifying sequential circuits should be explored. The relationship between state assignments and next state equations should be reviewed; perhaps efficient new state assignment methods could be found which directly yield design equations. Finally, the implications of modular realizations of asynchronous sequential circuits could be further developed.

APPENDIX 1: EXPERIMENTAL FLOW TABLE SIMPLIFICATION

The following tables summarize the results obtained in several flow table simplification experiments performed using the algorithm presented in Chapter III. The PL/1 programmed implementation was run on an IBM S/360-50 as part of the SHADE system discussed in Chapter V.

The large flow tables were obtained from randomly generated I/O pair specifications or by randomly introducing redundant rows into completely simplified flow tables.

Table II shows the effect of varying look ahead on simplification of a 193 row by four column table. For this particular example, it was concluded that look ahead of 10 to 20 rows (5 to 10%) produced satisfactory simplification.

Table III details several flow table simplification experiments. Reduction mechanism entries shown in Table III reflect the number of compatibility class enlargements produced by each of the listed methods.

Table II. VARYING LOOK AHEAD IN SIMPLIFICATION OF A 193x4 FLOW TABLE

Look Ahead, Rows	Simplification Time, Sec.	Output Table Size, Rows
5	45	35
10	40	34
15	40	33
20	41	32
25	45	33
35	48	36
50	47	35
75	52	35
100	53	34
150	53	35
190	53	32

Table III. EXPERIMENTAL FLOW TABLE SIMPLIFICATION RESULTS.

No.	INPUT FLOW TABLE				SIMPLIFIED FLOW TABLE			
	Rows	Cols.	Stable States	Unspec. Entries	Rows	Cols.	Stable States	Unspec. Entries
1	193	4	193	185	34	4	49	0
2	75	4	75	57	28	4	40	2
3	115	8	115	685	25	8	100	32
4	217	8	217		38	8	98	20
5	158	16	287	1968	37	16	268	137
6	96	16	194	1139	26	16	175	102

No.	PROGRAM PERFORMANCE				REDUCTION MECHANISMS			
	Time Required, Sec	Number of Itera-tions	Look Ahead, Rows	Minimum Table, Rows	Row Pair Combina-tions	Row & CC Combo.	New CC Enlarge-ment	Chain-ing
1	40	2	10	23	34	68	56	6
2	18	2	10	23	17	16	13	2
3	49	2	10	Unk.	27	14	43	6
4	82	2	10	23	36	3	79	2
5	93	1	10	Unk.				
6	32	1	35	Unk.	18	6	43	0

APPENDIX 2: SUMMARY OF STATE ASSIGNMENT EXPERIMENTS

Table IV summarizes the results of a set of state assignment experiments conducted to evaluate four non-standard state assignment techniques for normal, fundamental mode asynchronous sequential circuits. The methods (and identification numbers) used were:

1. Tracey's method one, which finds minimum variable state assignments by finding transition partitions, then minimizing the resulting constraint (Boolean) matrix.
2. Tracey's second method, which finds near-minimum variable assignments by reducing the transition partition matrix to near-minimum size.
3. Tracey's third state assignment technique, which requires the computation of K-set partitions and near-minimum reduction of the resulting constraint matrix.
4. The new state assignment algorithm proposed in this dissertation, which involves near-minimum reduction of K-set partition matrix <u>segments</u>.

The timing data was obtained by executing PL/1 implementations of the state assignment procedures on a S/360-50 computer.

Entries indicated by "*" are approximations (in most cases necessitated by premature termination of the programs).

Table IV. STATE ASSIGNMENT EXPERIMENTS.

	FLOW TABLE				STATE ASSIGNMENT		
No.	Rows	Columns	Stable States	Unspec. Entries	Method	Time Sec.	Number Variables
1	6	3	9	3	1	1.8	4
2	6	4	12	0	1	36	5
	6	4	12	0	2	21	5
3	12	4	14	6	1	10^4*	4
	12	4	14	6	2	20	5
	12	4	14	6	3	22	7
	12	4	14	6	4	22	7
4	18	4	20	12	2	171	7
	18	4	20	12	3	106	8
	18	4	20	12	4	74	9
5	23	4	34	0	4	154	14
6	26	16	175	102	4	375*	27*
7	28	4	40	3	4	1000*	12
8	29	4	34	24	4	135	10
9	34	4	49	0	4	464	16
10	38	8	98	20	4	840	28
11	37	16	268	137	4	1040	28

BIBLIOGRAPHY

1. McCluskey, E.J., Introduction to the Theory of Switching Circuits. New York: McGraw-Hill Book Company, 1965.

2. Paull, M.C. and S.H. Unger, "Minimizing the Number of States in Incompletely Specified Sequential Switching Functions," IRETEC, EC-8 (September, 1959), 356-367.

3. Grasselli, A. and F. Luccio, "A Method for Minimizing the Number of Internal States in Incompletely Specified Sequential Networks," IEEETEC, EC-14 (June, 1965), 350-359.

4. Ginsburg, S., "On the Reduction of Superfluous States in a Sequential Machine," J. ACM, Vol. 6 (September, 1959), 252-282.

5. Kella, J., "State Minimization of Incompletely Specified Sequential Machines," IEEETEC, C-19, No. 4 (April, 1970), 342-349.

6. Huffman, D.A., "The Synthesis of Sequential Switching Circuits," J. of the Franklin Institute, Vol. 257 (March and April, 1954), 161-190 and 257-303.

7. Liu, C.N., "A State Variable Assignment Method for Asynchronous Sequential Switching Circuits," J.ACM, Vol. 10 (1963), 209-216.

8. Tracey, J.H., "Internal State Assignments for Asynchronous Sequential Machines," IEEETEC, EC-15 (August, 1966), 551-560.

9. Elsey, John, "An Algorithm for the Synthesis of Large Sequential Switching Circuits," Report R-169, Coordinated Science Laboratory, University of Illinois, Urbana, 1963.

10. Smith. R.J., et al., "Automation in the Design of Asynchronous Sequential Circuits," Proc. AFIPS SJCC, 1968, 55-60.

11. Burton, D.P. and D.R. Noaks, "Complement Free S.T.T. State Assignments for Asynchronous Sequential Machines," Paper read at the Second National Symposium on Logic Design, University of Reading, Reading, Penn., March 28, 1969.

12. Tan, C.J., "Synthesis of Asynchronous Sequential Switching Circuits," Dr. Engr. Sc. Dissertation, Columbia University, June, 1969.

13. Maki, G.K., "Minimization and Generation of Next-State Expressions for Asynchronous Sequential Circuits," Masters Thesis, University of Missouri - Rolla, 1967.

14. Breuer, M.A., "Heuristic Switching Expression Simplification," Proc. 1968 ACM National Conference, 241-250.

15. Altman, R.A., "The Computer Aided Generation of Flow Tables for Asynchronous Sequential Circuits," Masters Thesis, University of Missouri - Rolla, 1968.

16. Unger, S.H., "Flow Table Simplification -- Some Useful Aids," IEEETEC, EC-14 (June, 1965), 472.

17. Schoeffel, W.L., "Programmed State Assignment Algorithms for Asynchronous Sequential Machines," Masters Thesis, University of Missouri - Rolla, 1967.

18. Huffman, D.A., "The Synthesis of Sequential Switching Circuits," J. of the Franklin Institute, Vol. 257 (March, 1954), 176.

19. Unger, S.H., "Flow Table Simplification - Some Useful Aids," IEEETEC, EC-14 (June, 1965), 473.

20. Mago, Gyula, "Universal State Assignment for Asynchronous Sequential Circuits," Unpublished Manuscript R-69-6, IEEE Computer Group Repository, 1969.

21. Tracey, J.H., "Internal State Assignments for Asynchronous Sequential Machines," IEEETEC, EC-15 (August, 1966), 552.

22. Smith, R.J., "A Programmed Synthesis Procedure for Asynchronous Sequential Circuits," Masters Thesis, University of Missouri - Rolla, 1967.

VITA

Robert Judson Smith II was born May 12, 1944, in San Francisco, California. He received his primary and secondary education in Wichita, Kansas. He has received his college education from Wichita State University and the University of Missouri - Rolla. He earned a Bachelor of Science degree in Electrical Engineering (Cum Laude) from Wichita State University in June 1966, and a Master of Science degree in Electrical Engineering from the University of Missouri - Rolla in January 1968.

Mr. Smith has been enrolled in the Graduate School of the University of Missouri - Rolla since September 1966; he held an NDEA fellowship from that time until June 1969. He was an instructor in Electrical Engineering from September 1969 until February 1970. Mr. Smith is presently involved in clinical data processing research at the Missouri Institute of Psychiatry, University of Missouri School of Medicine.