PLAFST PROGRAMMABLE LOGIC ARRAY FROM STATE TABLE(U) AIR
FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF
ENGINEERING   D C PELAN DEC 83 AFIT/GE/EE/83D-57

F/G 9/2   NL

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

PLAFST

Programmable Logic Array From State Table

THESIS

AFIT/GE/EE/83D-57   Darrell C. Pelan
                          1st Lt   USAF

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY
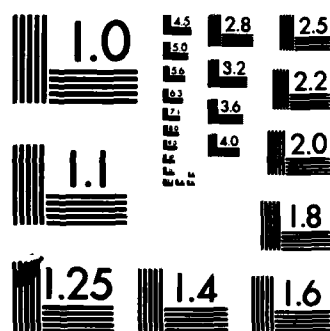
Wright-Patterson Air Force Base, Ohio

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	

By___
Distribution/
Availability Codes

Dist	Avail and/or Special
A||	

DTIC
COPY
INSPECTED
?

PLAFST

Programmable Logic Array From State Table

THESIS

AFIT/GE/EE/83D-57   Darrell C. Pelan
                          1st Lt   USAF

DTIC
ELECTE
S
FEB 29 1984
D

D

PLAFST

Programmable Logic Array From State Table

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

by

Darrell C. Pelan, B.S.

1st Lt   USAF

Graduate Electrical Engineering

December 1983

Acknowledgements

PLAFST has been a worthwhile project that at times seemed to provide endless amounts of entertainment during the past months. A strong debt of gratitude is owed to my faculty advisors, Lt Col Hal Carter and Cpt (P) Fredeirck Zapka. They provided excellent guidance at the beginning of this thesis project and were always willing to provide help when I needed it as PLAFST progressed. Their willingness to let me work independently was greatly appreciated.

Finally, I wish to express my love and gratitude to my wife, Evelyn, who provided endless moral support, acted as editor-in-chief for regulations and guidelines, and understood my all night stays at the computer center.

<div align="right">Darrell C. Pelan</div>

Contents

----

iii

List of Figures

List of Tables

AFIT/GE/EE/83D-57

Abstract

Programmable Logic Array From State Table (PLAFST) is a computer aided design (CAD) tool that takes a symbolic state table as input and produces a very large scale integrated (VLSI) circuit implementation of the symbolic state table. The state table is first reduced symbolically using equivalence partitioning. A near optimal binary state assignment is made based on the Story, Harrison, and Reinhard procedure as modified by Noe and Ryhne. Distinct state assignment variables are sorted based on cost estimates obtained by increasing the number of adjacencies in the state transition table. Once sorted, the actual costs of valid state assignments made from the state variables are calculated. Since state assignments with the lowest cost estimates are investigated first, an optimal solution is found with a small number of iterations. This binary state assignment is demonstratably less costly than either simple or gray code assignments of the state variables. The VLSI circuit consists of a programmable logic array (PLA) and clocked buffers. The state buffers are properly interconnected. The final outputs are Chip Layout Language (CLL) and Caltech Intermediate Format (CIF) descriptions of the integrated circuit. PLAFST also plots the final integrated circuit.

vii

INTRODUCTION

Problem

The primary task of this thesis is to develop a computer aided design (CAD) tool that implements a synchronous finite-state machine with a programmable logic array (PLA). This CAD tool operates on the AFIT VAX 11/780 running under the UNIX operating system. It functions in the same manner as other AFIT CAD tools, such as Chip Layout Language (CLL) and PRESTO. The user's manual has the same style as these programs.

The input file to the CAD tool developed by this thesis, PLA From State Table (PLAFST), contains inputs, symbolic states, transitions, and outputs. The input file may also include an error state for transitions from illegal states. PLAFST provides several output files showing state table reductions, state assignments, and other results of the minimization process. The final output files include a CLL file, a Caltech Intermediate Format (CIF) file, and a plot of the final circuit.

The PLA is designed with nMOS very large scale integration (VLSI) technology with a selectable lambda. The default PLA generated by PLAFST includes the combinational logic and clocked buffers for the state, input, and output signals. These buffers are standard cells from the Stanford nMOS Cell Library. The buffers that contain the next

and present states are properly interconnected. An option, -s, causes PLAFST to generate only the PLA. The designer can then use this PLA in a CLL program. PLAFST also determines, via PLAGEN, if the synchronous finite-state machine (SFSM) exceeds size constraints.

Approach

PLAFST is a shell script running under the UNIX operating system which controls the execution of programs residing on the VAX 11/780. These programs are a mixture of programs currently on the AFIT VAX 11/780 and new programs. All new programs are designed using Structured Analysis and Design Techniques (SADT) (Ref 2:63) and written in C. PLAFST causes the following tasks to be accomplished:

1. Reduces the symbolic state table

2. Assigns states

3. Reduces the combinational logic.

4. Generates the PLA CIF specification

5. Generates the SFSM CLL specification

6. Plots the SFSM

7. Generates the SFSM CIF specification.

Figure I-1 shows the relationship between PLAFST and other resident programs on the VAX 11/780. PRESTO, PLAGEN, and CLL previously existed on the VAX. CFORM translates PRESTO's output to the format required by

PLAGEN. SYM and ASSIGN comprise the bulk of the programs created during this thesis effort. MAKE_SFSM decodes SYM's output and the SFSM_PLA CIF file to generate the SFSM CLL file. The SFSM CLL file includes the PLA, buffers, and the state variable interconnections. SYM, ASSIGN, and MAKE_SFSM are discussed in the following paragraphs.
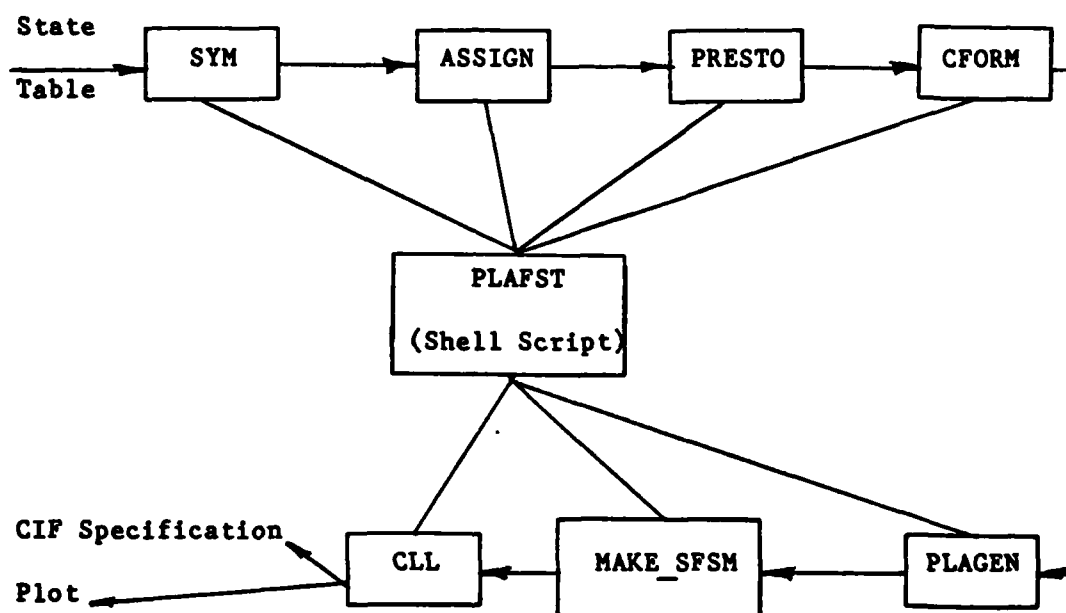
Figure I-1. PLAFST Structure Chart

Symbolic State Table Reduction (SYM). The symbolic state table is reduced by equivalence partitioning (Ref 3:22). The first step in equivalence partitioning groups all states in equivalent partitions. For this step, states are considered equivalent if and only if identical outputs are generated for each of the possible inputs (Ref 3:19). Subsequent steps modify this definition slightly. States are equivalent if and only if they identically transition between partitions

(Ref 3:20). The symbolic states are repeatedly partitioned using this definition until no further change in the partitions results. The minimum set of symbolic states is chosen by taking one state from each partition.

One state may be designated as an error recover state in the event that an undefined state is entered. The total number of states possible in any binary SFSM will always be a power of two. There is no guarantee in the symbolic state table and especially in the reduced table that the number of states will be a power of two. Consequently, there will probably be some states that are undefined. In case the SFSM erroneously transitions to one of these states, it should be designed to transition to a specific state to recover from this error.

However, the algorithm used by PLAFST to assign an optimum state assignment will be degraded by a designated error state. The algorithm uses undefined states as don't care conditions in the Quine-McCluskey state table reduction. A designated error state causes all states not defined in the input file to transition to the error state. Therefore, the algorithm can not use these states to minimize the costs of the PLA.

The possibility that the number of states will equal a power of two must also be considered. In this case, the SFSM has no undefined states to which it can transition. Therefore, the error recovery state designation is ignored.

State Assignment (ASSIGN). The states of the symbolic state table are assigned binary values based on the Story, Harrison, and Reinhard (SHR) optimum state assignment. The SHR method was originally optimized for use with J-K flip-flops (Ref 4:1365). P. S. Noe and V. T. Rhyne optimized the SHR optimal state assignment algorithm for the D flip-flop in 1976 (Ref 5:306). PLAFST uses the Noe and Rhyne version of the SHR algorithm because of the PLA's inherent D flip-flop characteristic.

The algorithm is based on the comparison of a series of cost estimates for each of the state assignment columns. The costs estimates are based on the number of first and second level gates required to implement each state assignment column. This cost estimate is well suited to a PLA which is basically a two level NAND/NOR gate array. The state assignment column is best explained by an example. A four state SFSM has two columns of binary digits, if the states are listed by row. This is true for any binary state assignment scheme that might be used. Each of these columns is a state assignment column. Once the cost estimates are generated, they are sorted in monotonically nondecreasing order. An ordered search is performed to find an optimal state assignment. There may be more than one state assignment that satisfies the criteria of optimum state assignment. This algorithm does not consider outputs.

SFSM CLL Specification (MAKE SFSM). PLAFST automatically adds clocked buffers to the PLA generated from the assigned state table. It also interconnects the state buffers. This feature may be disabled by

use of an option, -s, in the command line to PLAFST. The clocked
buffers used are PlaClockIn and PlaClockOut from the Stanford nMOS Cell
Library (Ref 6). Since a PLA has a regular structure, the distance
between adjoining buffers is a constant. These buffers are iterated
using CLL statements with dummy variables. Buffers are also added for
each of the input and output signals. In a similar manner, metal wires
can be added to interconnect the state buffers. PLAFST replaces these
dummy variables with constant values multiplied by the number of SFSM
states. The regular structure of the PLA, clocked buffers, and
interconnecting wires is shown in Figure I-2.
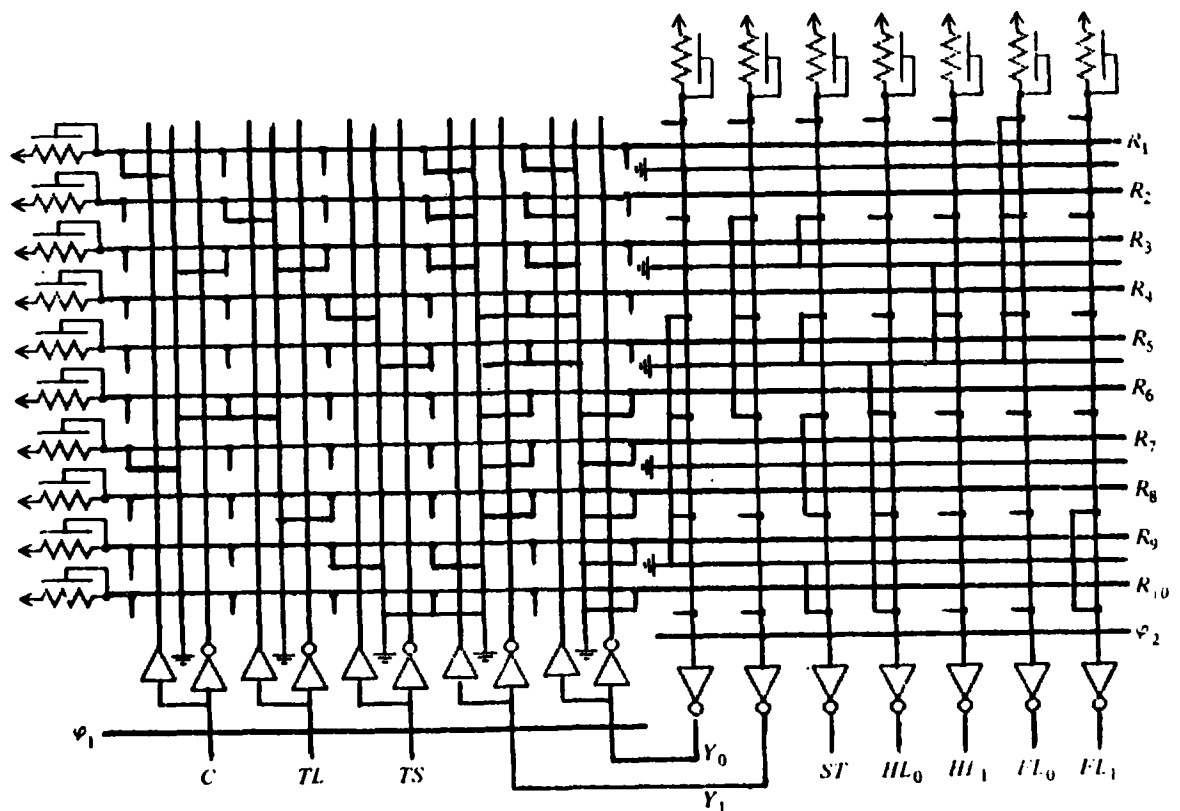
Figure I-2. PLA Finite State Machine Implementation
of a Light Controller (Ref 7:Plate 8)

I-6

Scope

This thesis is limited to completely specified synchronous finite-state machines. The number of computations required to completely minimize a state table quickly grows with the number of states in the table, much like the classic "traveling salesman" problem. This makes state minimization of completely specified SFSMs practical only for SFSMs with a small number of states. For this reason, PLAFST reduces the symbolic state table with a partitioning scheme that does not guarantee a minimum cost solution. The state assignment algorithm guarantees a minimal solution for the reduced state table. A minimal solution is one with the least number of AND and OR gates required to implement the state table with a PLA.

State minimization of incompletely specified finite-state machines is much more complex than completely specified finite-state machines due to state-splitting (Ref 1:406). An unspecified state is a state whose output is not designated as true or false. Since the unspecified state can be specified as a 1 or 0, the state must be split into two rows in the symbolic state table. One of the new rows has the state output specified as a 1. The state output of the other row is a 0. If a row in the symbolic state table had two unspecified states, four new rows would have to be added to the symbolic state table in order to account for all possible combinations of the two unspecified states. State-splitting can quickly expand the size of the symbolic state table and vastly increase the number of computations required just to reduce the table.

I-7

Background

Computer Aided Design Tools.   This thesis is concerned with   the
development   of a computer aided design (CAD) tool   that   implements
synchronous finite-state machine designs with programmable   logic arrays
(PLA).   This CAD tool operates   in the Air Force Institute of
Technology's (AFIT) CAD environment.   Currently, AFIT CAD   facilities
are capable of designing very large scale integrated (VLSI)   circuits.
These facilities operate under UNIX on the VAX 11/780 computer.   Many
of the CAD tools presently used at AFIT, and UNIX itself, are written in
the C language.

The   Caltech Intermediate Format (CIF) is used to specify VLSI
circuits at AFIT.   CIF files are very exact specifications of the
actual masks used to fabricate an integrated circuit.   Like assembly
language programming, CIF programming is very tedious.   Fortunately, a
higher order language that generates CIF files exists on the VAX 11/780.
This language, called the Chip Layout Language (CLL), is an English-like
language that allows the use of symbols, constants, and arithmetic
expressions rather than the direct coded chip layouts of CIF. The CAD
tool developed by this thesis functions as a higher order language that
creates another level of abstraction between the designer and CLL.

<u>Synchronous Finite-State Machines</u>.   The synchronous finite-state
machines implemented by this thesis are devices that transition between
stable states when triggered by a clock signal.   The synchronous
finite-state machine (SFSM) accepts inputs at the beginning of a
transition, generates outputs during the transition, and finally arrives
at a new state.   The inputs and outputs include the present and next
states respectively. All states, inputs, transitions, and outputs are
completely specified. Examples include computer control units, traffic
lights, and digital watches.

Present State	Next State/Output	
	0	1 ←——Inputs
LG	LG/1	LY/0
LY	LR/0	LR/0
LR	LR/0	LG/1

LG = Light Green   Input = 1 Time Out
LY = Light Yellow   = 0 Not Time Out
LR = Light Red   Output = 1 Walk Sign on
                                 = 0 Don't Walk
                                     Sign on

Figure I-3. Symbolic State Transition Table

A SFSM is generally designed using either a Mealy or Moore state
diagram which graphically depicts each state, state transitions, inputs,
and outputs. A state diagram can be directly translated to a symbolic
state transition table, Figure I-3.   Each state symbol represents an
actual physical mode of the SFSM.   For example, state LG represents the
traffic light mode in which the green light is on and the red and yellow
lights are off. The inputs and outputs are coded in a binary format.

I-9

The input codes designate the columns of the Next State portion of the table. The next state and output, separated by a slash, are listed in the appropriate column and row.

Figure I-3 is an example of a symbolic state transition table that describes the operation of a simple traffic light. The possible situations or states for this traffic light are listed in the Present State column. A Time Out signal is the only input. The symbolic state table shows all possible transitions from the present to next state given that the input is a 1 or 0. The table also shows when the walk sign will be turned on by the output code following the slash in the Next State column. The program developed by this thesis uses this type of information as input.

Summary of Current Knowledge

A SFSM is fabricated by digitally encoding the information shown in Figure I-3, minimizing the encoded information, and then generating a VLSI circuit that implements the SFSM. The two basic circuits of an SFSM are the combinational logic and memory elements. The combinational logic transforms the state machine inputs and present state into the outputs and next state. The memory elements retain the present state of the synchronous finite-state machine until the next transition. When the clock pulse is received, the next state furnished by the combinational logic becomes the new, present state in the memory elements.

SFSMs can be realized in several ways. One way is to create a programmable logic array (PLA). A PLA has several advantages over other methods, especially when VLSI constraints are considered. VLSI circuits are much denser if the circuit is composed of a basic cell that is repeated in a matrix array. Since only the basic cell must be designed, array structures also require less time to design than a circuit made from discrete components. Memory circuits and PLAs are examples of cells used repeatedly in a matrix.

The two phase clock that controls the PLA input and output buffers has two major advantages. The first advantage is the latch formed by the clocked input and output buffers. This latch is an inherent D flip-flop. Since this D flip-flop can be used as the required SFSM memory element, the PLA implementation of a SFSM does not require external memory. The second advantage is that the SFSM is insensitive to data propagation delays in the PLA matrix. However, this is only true if the clock cycle is longer than any propagation time through the PLA combinational logic.

Figure I-4. Two Phase Clock

The clock consists of two phases which are never "high" at the same time and are both "low" for a short time between phases. This is shown in Figure I-4. The input and output buffers are clocked by different phases. For purposes of discussion, the input buffer will be clocked by phase 1 and the output buffer will be clocked by phase 2. The clocking mechanism for the buffers are pass transistors in the data path. These pass transistors are activated only when the controlling phase is high. Data is passed to the buffer when the pass transistors are activated. Any changes in the data will also be passed to the buffer while the pass transistor is activated. Once the clock phase transitions low, further changes in the data will have no affect on the buffer and the data in the buffer is used by the next stage of the PLA. For this reason, data is considered clocked into the buffer on the falling edge of the clock phase.

The block diagram in Figure I-5 shows the feedback lines which connect the output and input buffers. The output buffers contain the next state of the SFSM. The input buffers, which contain the SFSM's present state, act as D flip-flops. They retain the present state of the SFSM during phase 2 when the next state and outputs are generated by the PLA matrix. The SFSM transitions between states during phase 1 when the next state is clocked into the input buffers via the feedback lines. The input buffers function as D flip-flops by outputting the same information that is clocked into them and remembering this data until new data is clocked in.

```
     Phase 1                                        Phase 2

Inputs  ┌─────────┐      ┌────────────────┐      ┌─────────┐  Outputs
─ ─ ─ ─►│ Input   │─ ─ ─►│ Combinational  │─ ─ ─►│ Output  │─ ─ ─ ─ ─►
        │ Buffer  │      │    Logic       │      │ Buffer  │
        └─────────┘      └────────────────┘      └─────────┘
            ▲ ▲                                        │
            │ │                                        │
            │ └────────────────────────────────────────┘
            │                                          │
            └──────────────────────────────────────────┘
```
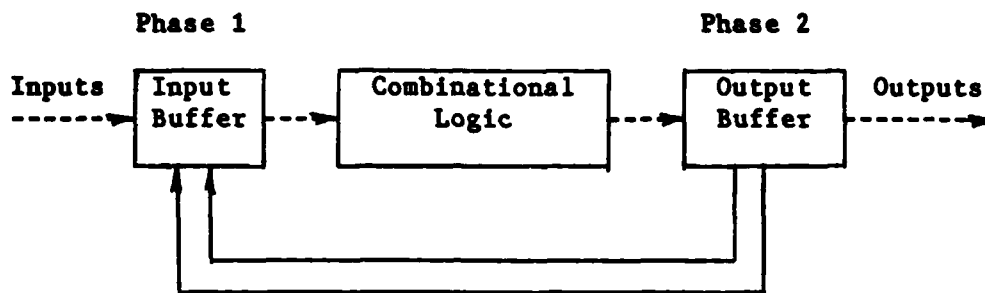
Figure I-5. Programmable Logic Array Block Diagram

A PLA's insensitivity to propagation delays is a result of the two
phase clock discussed above. Information from the input buffers begins
to propagate through the PLA matrix at the beginning of phase 1. At the
end of phase 1, the data is clocked into the input buffer and the input
data can no longer change. Data is clocked into the output buffers in
the same manner during phase 2. The data clocked into the input buffers
has until the end of phase 2 to propagate through the PLA matrix. As
long as the time between clock phases is longer than the propagation
time through the PLA, differences in data propagation times will have no
effect on the PLA output.

System

PLAFST is required to enhance AFIT's CAD capability to implement synchronous finite-state machines. Enhancement means the SFSM designer works at the symbolic level to specify state transitions, inputs, and outputs. The designer lets PLAFST handle all the details of generating the SFSM CIF specification for manufacture. These details include the reduction of the symbolic table, optimum state assignment, PLA generation, and generation of the SFSM CIF specification. Currently, only the PLA generation can be accomplished by an automated tool. PLAFST must also be user friendly and be integrated into the AFIT CAD environment. As a user friendly program, PLAFST checks the input file for errors and supplies meaningful messages about the type of error and where the error occured.

Since AFIT's CAD tools for VLSI design are hosted on a VAX 11/780 running UNIX, PLAFST must also execute under this operating system. Three of the seven tasks listed in Chapter 1, Approach, can be accomplished by programs already running on the VAX 11/780. This leads to the idea that PLAFST should be a shell script that calls other programs to accomplish specific tasks. A shell script also aids development and testing of the entire program. Programs for each task can be tested individually as they are completed. The shell script, itself, can be tested by the use of "stubs" or using the ECHO command to display the desired command. In this manner, the behavior of the shell

script can be tested without actually running the called programs. SADTs are used to strengthen these modular development and testing concepts. Maintenance of PLAFST and it's related programs is also enhanced by not proliferating computer languages within the AFIT CAD environment. Since UNIX, PRESTO, and other CAD programs are written in C, all programs related to PLAFST are written in C. This includes the shell script which uses C like commands.

Size

There are three size limits that must be considered. The first is the physical size of the SFSM. PLAFST assumes that the SFSM is a contiguous device that is fabricated on single die. No provisions are made to split SFSMs across more than one die. It is the designer's responsibility to determine if the SFSM generated by PLAFST exceeds this criteria.

The second size consideration is the host computer limits in terms of time and resources. Programs that are used by PLAFST, such as PLAGEN and PRESTO, may have additional size constraints. Obvious indicators would be excessive time to execute PLAFST and messages from the UNIX operating system. Since PLAFST provides an output file from every program module, the user should be able to easily determine which module exceeded it's own limits or those of the host computer. An additional tool is the PS command. This command shows which program is currently running and the elapsed execution time.

The third consideration is the actual scaling factor for lambda used in the generation of the SFSM. The lambda size must be supplied by the user.

Options

PLAFST has four options. The first option, -s, allows the user to stop execution of the program after the PLA has been generated. The final PLAFST output is the SFSM PLA CIF specification without buffers or feedback lines. The SFSM PLA could then be used with different buffers or connections than PLAFST provides. The default is to add the Stanford nMOS Cell Library cells PlaClockIn and PlaClockOut and interconnect the state buffers using poly and diffusion wires (Ref 6).

The second option, -d, is for debugging. Print statements used to debug PLAFST are controlled by IF statements that test for the debug switch. The print statements output to the standard error file and show the effect of critical data manipulations. ASSIGN, the program which implements the optimum state assignment has an additional debug feature, mass-debug. The option is not invoked by PLAFST. It can be used by running ASSIGN with the -m and -d options. Mass-debug must be used with the debug option to provide meaningful output. This option will generate massive files, so it is best to route the output to a terminal for viewing. Even small state tables with five states and ten transitions will generate mass-debug files in excess of 150k.

The third and fourth options, -sa and -gc, control the state
assignment method. The -sa option merely assigns states in the same
order as the input file. For example, the first state would be assigned
the number zero and the fifth state would be assigned the number 4. The
-gc option assigns the state values using a gray code. The gray code
assignment method has no more than one binary digit difference between
adjoining states. For example, a four state SFSM would have state
assignments of 00, 01, 11, and 10. These values would be assigned to
the symbolic states in the same order as the input file.

Inputs

PLAFST is invoked by the command line:

  PLAFST [-s][-d][-sa, -gc] < input.file

The input file includes the number of states, inputs, outputs, a CIF
number, and the lambda size on the first line. Subsequent lines include
the state, input, and output names. The input and output names are
assumed to represent independent binary variables. The next two blocks
of information are the state array followed by the output array. The
input file is not sensitive to which line information is on, only the
order in which it appears.

The delimiter between any information in the input file is one or
more spaces. A slash and any number of spaces is used to separate
output variables which are true for the same transition. An additional

special character is the asterisk, *, which denotes an error recovery
state. The error recovery state must be preceded by the asterisk which
may not have intervening spaces. PLAFST will send the user a warning if
an error state is not included. The reasons for and against an error
state are discussed under State Assignment in Chapter 1. The general
format and a specific example (Ref 7:87) are shown in Figures II-1 and
II-2.

```
4 3 5 950 2.5    /* #_states, #_inputs, #_outputs, CIF_#, lambda    */

*HG              /* Designated error state and first symbolic state */
HY               /* Symbolic states                                 */
FG
FY

car              /* Input names                                     */
long_timeout
short_timeout

s                /* Output names                                    */
h0
h1
f0
f1

HG HG HG HG HG HG HY HY                    /* Next state array       */
HY FG HY FG HY FG HY FG
FY FY FY FY FG FG FY FY
FY HG FY HG FY HG FY HG

f0 f0 f0 f0 f0 f0 f0/s f0/s                /* Output array           */
h1/f0 h1/f0/s h1/f0 h1/f0/s h1/f0
h1/f0/s h1/f0 h1/f0/s h0/s h0/s h0/s
h0/s h0 h0 h0/s h0/s h1/f0 h1/f0/s
h1/f0 h1/f0/s h1/f0 h1/f0/s h1/f0 h1/f0/s
```

Figure II-1. Input Array Example

The number of states, inputs, and outputs must be supplied to
PLAFST so that it can determine what each symbol is supposed to

represent. The CIF number which PLAFST uses must be supplied to CLL and PLAGEN. PLAGEN additionally requires that the lambda size be specified.

Symbol names must be 25 characters or less and include the alphabet, 0-9, and the underline character, _. The characters may be in any order and case is significant. These conventions are tailored after CLL.

    #_states #_inputs #_outputs CIF_# lambda

    State Names

    Input Names

    Output Names

    State Array

    Output Array

Figure II-2. Input Array General Format

The order in which the symbols are listed is extremely important. PLAFST uses the order of the state and input names to decode the information in the array. PLAFST assumes that the input is a standard symbolic state table such that the present states are listed vertically and the inputs are listed horizontally across the top of the state table. The first state name is associated with the first row of the state table. The next state name is paired with the next row. In a similar manner, the first input name is associated with the left most column in the state table. Successive input names should head the remaining columns. Input names will arbitrarily be assigned increasing

binary values from left to right. The first input name is assigned the binary value zero. Input names are placed in the SFSM PLA in the same order as the input file. The first input name is the left most input to the PLA. In the same manner, the outputs are ordered from left to right according to their order in the input file. The first output name is the left most output from the PLA.

The first array is the state transition array. This array is listed in order by row with each row listed from left to right. The output array follows the state transition array and is organized in the same manner. However, there are some differences. The output variables are listed only if they are true for a given transition. There may also be more than one output variable true at the same time. In this case, the variables are separated by a slash, /, and by any number of spaces. A zero must be used to show that none of the outputs are true for a given state. This is required to preserve the order of the output array.

PLAFST determines if the correct number of states and outputs appear and sends an appropriate message to the user. An incorrect number causes the program to stop execution.

Overview

PLAFST was designed at the system level using SADTs. This approach starts with the basic inputs, outputs, and some means to convert the inputs into the outputs. The means, in this case, is PLAFST. This is shown in Figure III-1, Node A-0. Node A-0 is decomposed into more detailed levels. Each successive level gives more information about how an input is transformed into an output. This progression is seen in Nodes A0, A1, and A3. Node A2 is not broken down past the A0 level since the only program in A2 is PLAGEN. PLAGEN is an existing program that will not be altered during the development of PLAFST.

The remainder of this chapter is devoted to the SADT documents. These include the Node List, Data Dictionary, and the node descriptions. The Node List contains the names of all nodes. The Data Dictionary defines all information passed between nodes. The node descriptions elaborate on the tasks accomplished by each node.

Node List

Node A-0: PLAFST - Programmable Logic Array From State Table

Node A0: PLAFST

Node A1: Manipulate State Table

Node A11: Reduce Symbolic Table

Node A12: Assign States

Node A13: PRESTO

Node A14: Change Format

Node A2: PLAGEN

Node A3: Make SFSM

Node A31: Add Clocked Buffers

Node A32: Connect State Buffers

Node A33: CLL

Data Dictionary

All Nodes:

Keyboard Input: The command line entered to the UNIX operating system. The command line includes PLAFST and the input file. The command line may include the option to stop program execution after PLAGEN.

PLA CIF Specification: The CIF file that describes the PLA that implements the SFSM.

PLA Truth Table: The truth table that describes the SFSM PLA after the states have been assigned.

Reduced PLA Truth Table: The PLA Truth Table after the combinational logic has been reduced.

Reduced Symbolic State Table: The original input state array after it has been reduced through equivalence partitioning.

SFSM CIF Specification: The CIF file that describes the SFSM PLA including the clocked buffers and interconnected states.

SFSM Plot: A plot of the SFSM integrated circuit.

Symbolic State Table: The original state array from the input file.

Node A0, A1, and A3:

PLAFST Control: The sequence of control by the shell script.

Node A1:

Dot Format: This file is the Reduced PLA Truth Table in the format used by PRESTO. The format must be changed before PLAGEN can use the information in the truth table.

Node A3:

Partial SFSM CLL Description: A CLL program that includes the SFSM PLA and the clocked buffers. The state buffers are not yet connected.

PLAFST is a UNIX shell script that initiates various other programs to generate the files shown from the symbolic state table input. The files are generated in the order shown from top to bottom. The final outputs are the PLA implementation of the SFSM specified by the input file. Reference Figure III-1.

I1 Keyboard Input

I2 Symbolic State
Table

PLAFST

Reduced Symbolic
State Table   O1

PLA Truth Table   O2

Reduced PLA Truth Table   O3

PLA CIF Specification   O4

SFSM CIF Specification   O5

SFSM Plot   O6

Figure III-1. Node A-0

Node A0

This node shows the primary breakdown of PLAFST. PLAFST consists of three main modules. The first module manipulates the input file into a PLA truth table. The second module generates the PLA structure. Finally, the third module adds the necessary buffers and connections to complete the SFSM. Reference Figure III-2.

Node A1 - Manipulate State Table. This node accomplishs the first of the three tasks listed the Approach section of Chapter 1. The input symbolic state table is reduced using equivalence partitioning. The states are assigned optimum binary values and the PLA Truth Table is generated. The combinational logic is reduced to make the Reduced PLA Truth Table file.

Node A2 - PLAGEN. This node consists of the program PLAGEN. PLAGEN creates the PLA Structure from the Reduced PLA Truth Table.

Node A3 - Make SFSM. This node modifies a CLL program with dummy variables in order to add clocked buffers and state interconnections to the PLA generated by PLAGEN. This node may not be executed if the option, -s, is used in the command line. In this case, the final output is the file created by PLAGEN.

Node: A0
Title: PLAFST - Programmable Logic Array From State Table
Date: 3 Jul 83
Rev.: 1.0
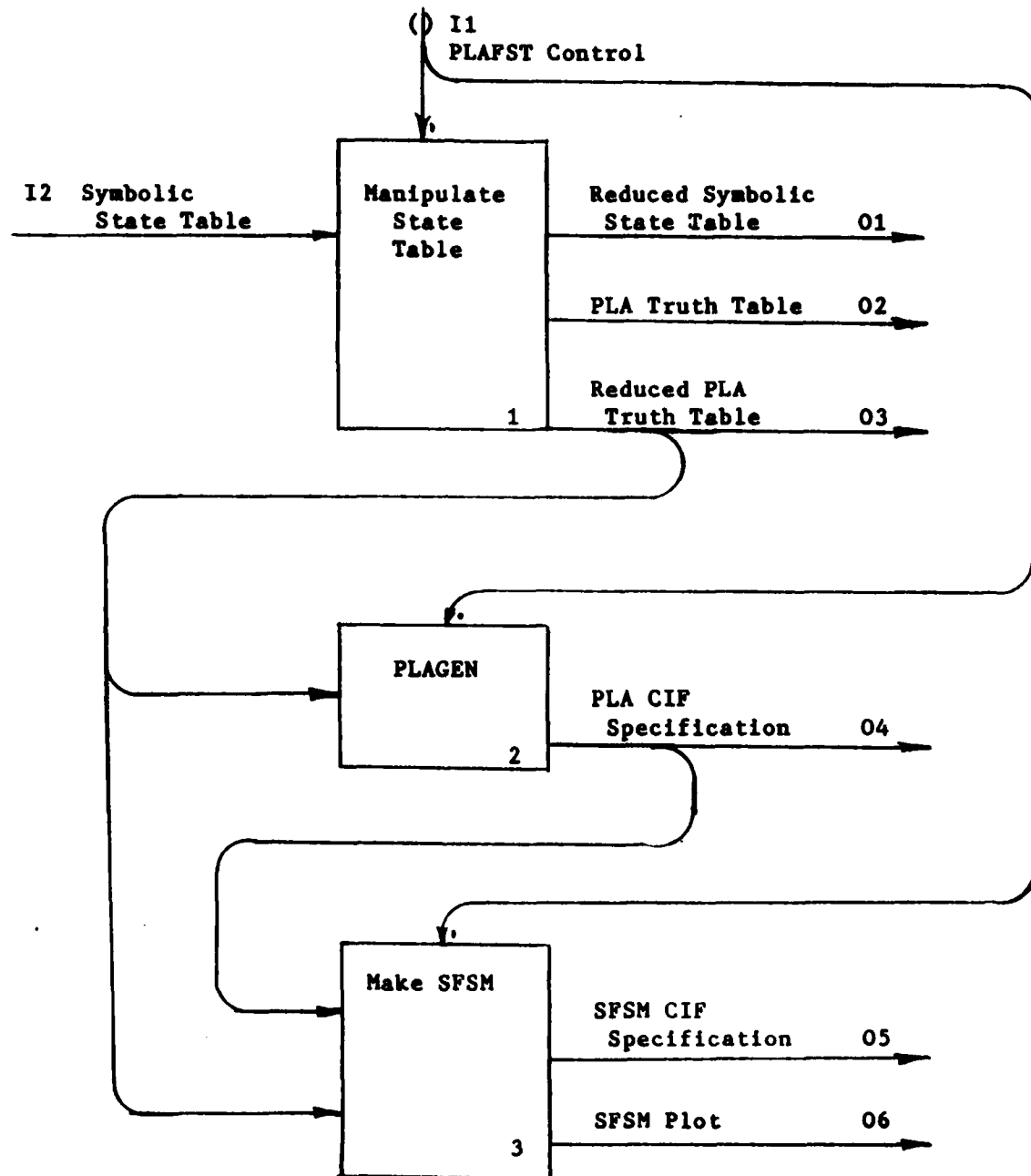
Figure III-2. Node A0

Node A1

This node shows the system level details of transforming the input file into the Reduced PLA Truth Table. The first module reduces the symbolic state table in the input file. The second module is the most complex. It contains the algorithm for assigning optimum binary values to each of the symbolic states. PRESTO, the third module, already exists in the AFIT CAD library. It reduces the combinational logic of the PLA Truth Table. The final module, A14, transforms the format used by PRESTO to the format used by PLAGEN. Reference Figure III-3

Node A11 - Reduce Symbolic Table. This module implements the equivalence partitioning algorithm discussed in the Approach section of Chapter 1. The symbolic states are divided into equivalent partitions and one state is chosen from each partition. These states are then organized into the Reduced Symbolic State Table.

Node A12 - Assign States. This module performs the Noe and Rhyne modified SHR optimal state assignment algorithm for D flip-flops. The algorithm has four basic parts. The first determines the state assignment column variables. The second part calculates the cost of each of these variables. The third step sorts the cost estimates in monotonically nondecreasing order. The fourth step performs an ordered search on the sorted cost estimates to determine an optimum state assignment. The solution may not be unique and does not consider the outputs.

Node A13 - PRESTO. This is a program currently in use at AFIT. PLAFST supplies the file generated by Assign States to PRESTO and uses the output file to generate the PLA structure.

Node A14 - Change Format. Not all CAD programs at AFIT are compatible. Before the output file from PRESTO can be used by PLAGEN to make the SFSM PLA, the format of the file must be changed. This program accomplishes this task.

Node: A1
Title: Manipulate State Table
Date: 3 Jul 83
Rev.: 1.0

I1 PLAFST Control

I2 Symbolic
State Table

Reduce
Symbolic
Table
1

Reduced Symbolic
State Table   O1

Assign States

2

PLA Truth Table   O2

PRESTO

3

Dot Format

Change Format

4

Reduced PLA
Truth Table   O3

Figure III-3. Node A1

III-10
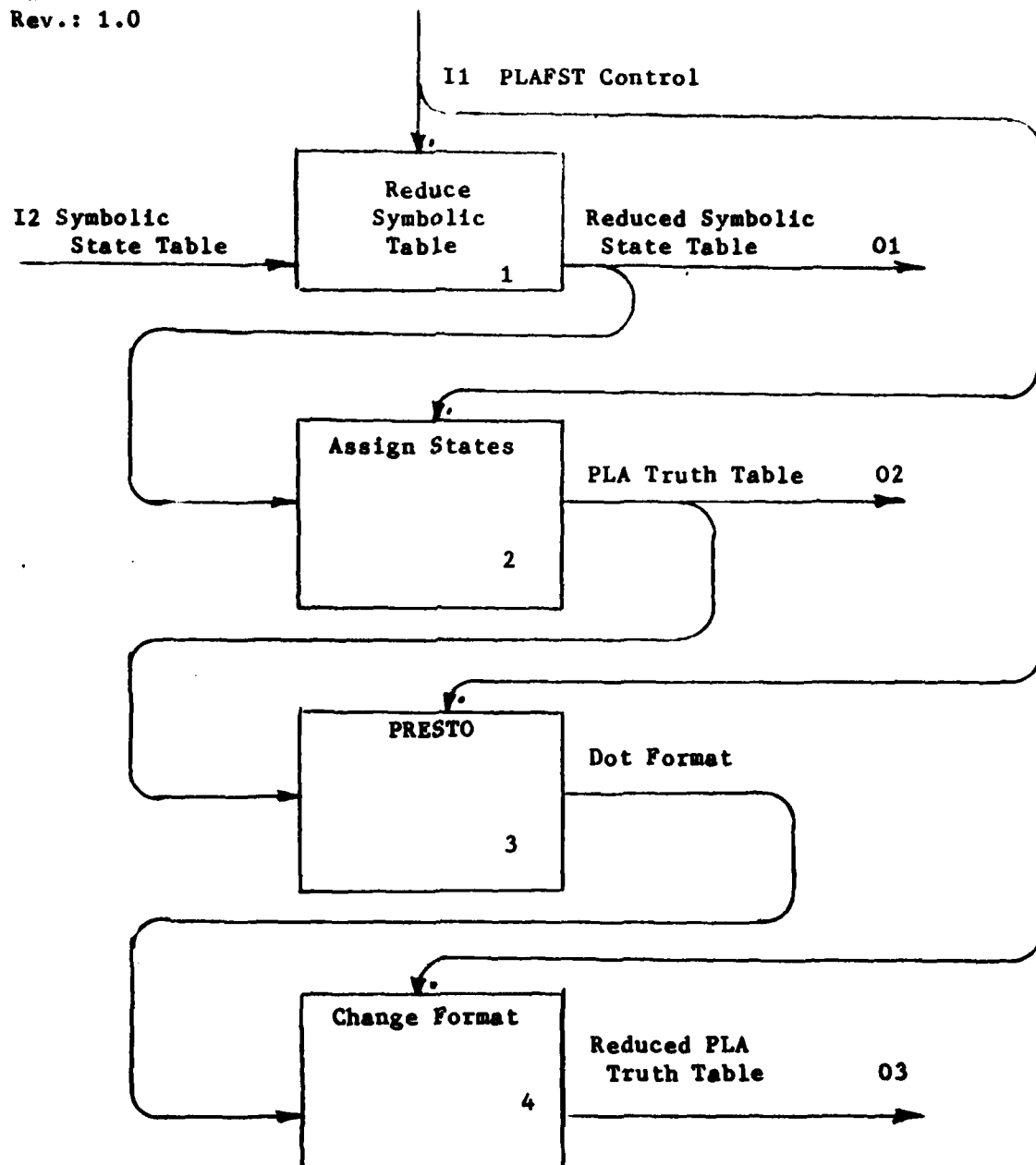
Node A3

This node modifies a CLL program with dummy variables. The dummy variables include the CIF number for calling the SFSM PLA, the number of clocked buffers, and the number of state buffers. Reference Figure III-4.

Node A31 - Add Clocked Buffers. This modules modifies the number of clocked buffers connected to the SFSM PLA in the CLL program. This number is calculated from the size of the Reduced PLA Truth Table.

Node A32 - Connect State Buffers. This module adds the correct number of wires to interconnect the state buffers. This number is also calculated from the size of the Reduced PLA Truth Table.

Node: A3
Title: Make SFSM
Date: 3 Jul 83
Rev.: 1.0

I1   PLAFST Control

I2   PLA CIF
        Specification

```
                    ┌──────────┐
                    │  Add     │        Partial SFSM CLL
                    │  Clocked │        Specification
                    │  Buffers │
I3   Reduced PLA    │          │
        Truth Table │        1 │
                    └──────────┘
```

Add Clocked Buffers   1

Partial SFSM CLL Specification

```
                    ┌──────────┐
                    │ Connect  │
                    │  State   │        SFSM CLL
                    │ Buffers  │        Specification
                    │        2 │
                    └──────────┘
```

Connect State Buffers   2

SFSM CLL Specification

```
                    ┌──────────┐
                    │  CLL     │
                    │          │        SFSM CIF
                    │          │          Specification    O5
                    │          │
                    │          │        SFSM Plot          O6
                    │        3 │
                    └──────────┘
```

CLL   3

SFSM CIF Specification   O5

SFSM Plot   O6
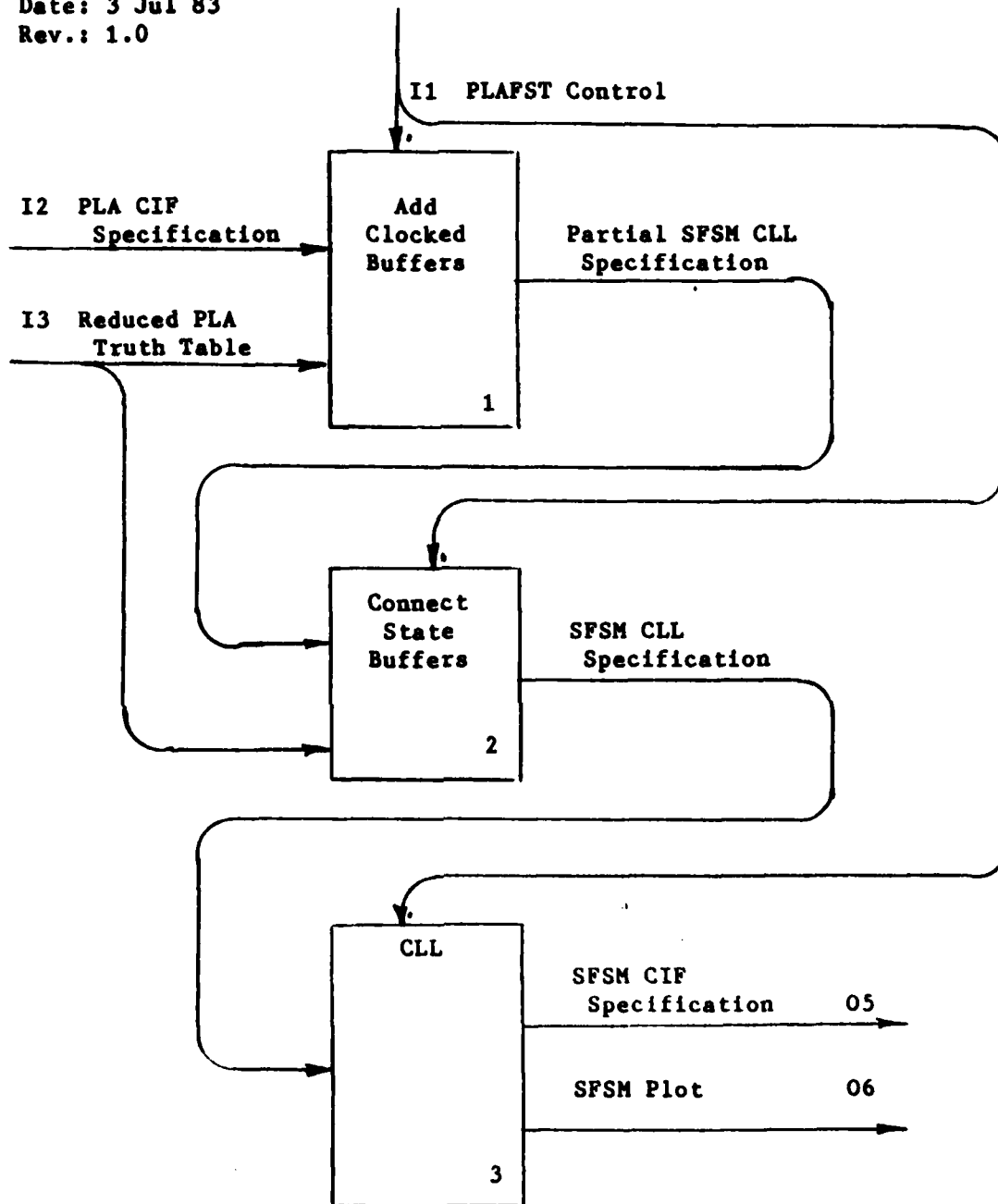
Figure III-4.   Node A3

DETAILED DESIGN

Overview

This chapter considers only two nodes from the system SADTs. Nodes A11, Reduce Symbolic Table, and A12, Assign States, provide the optimum truth table that is processed by existing programs in the AFIT CAD library. PLAFST uses three powerful CAD programs to accomplish the VLSI circuit design and specification. These programs are PRESTO, PLAGEN, and CLL. Since these programs are used as "black boxes", the system design description can adequately specify their inputs and outputs. The three remaining nodes were also adequately discussed in the system design. These nodes are A14, A31, and A32. They deal with file format changes and simple scaling of dummy variables within an established file.

Reduced Symbolic Table

This node is not broken into lower level SADTs because the algorithm is a straightfoward sorting routine. This routine, equivalence partitioning, consists of two parts (Ref 3:19-23). The first is a presort of the symbolic states. The states are partitioned into equivalent groups that generate identical outputs for each possible input condition. The states are then sorted a second time by state transitions only.

The second sort partitions the states into groups with identical transitions for each of the possible inputs. Transitions are identical if they are between the same groups as partitioned by the previous sort. The second sort is repeated until no changes are made in the groups of states. Once the sorting process is completed, one state from each equivalence partition is chosen. These states form the reduced state table.

Primary error checking is also be done by this routine. Error checking is based on the first five numbers in the input file. These numbers, in order, are the number of states, inputs, and outputs, the CIF number, and the lambda size. They control the number of symbolic names and the size of the next state and output arrays. The error checking detects if the symbolic names do not match those in appropriate array. It also detects if the array dimensions are too large or small.

Assign States

This node contains the algorithm for an optimum state assignment and is the heart of this thesis. The algorithm is a modification of the SHR optimal state assignment algorithm developed in the early 1970's by Story, Harrison, and Reinhard (Ref 4). Modifications to the SHR algorithm were presented in a series of articles through 1977 written by Noe and Rhyne (Ref 5, 8-10). These modifications included generation of the basic column variable set, cost estimation, and application of the algorithm to the D flip-flop case. The algorithm is discussed in detail in the following paragraphs. The SADT description follows this

discussion. Node descriptions refer back to the detailed discussion.

Step 1 - Basic Column Set. The algorithm is based on the concept of a state assignment column. A state assignment column is one bit wide and N bits long. N is the number of states in the state table. The state assignment columns are used in groups of n where n is an integer greater than or equal to log (R). An example of three state assignment columns is shown in Figure IV-1. These state assignment columns are from Figure IV-2 which depicts the the basic column set for a state table with five states. A five state SFSM requires three state variables, so three state assignment columns are used. The optimization Noe and Rhyne algorithm is shown in the Example section, Chapter IV.

The state assignment columns are also called y-variables and are subscripted. The subscript is the decimal value of the binary state assignment column read from top to bottom. The top bit is the most significant. Figure IV-1 also shows a valid state assignment. Reading across the columns, each row of bits is unique.

$$y_4 \quad y_9 \quad y_{14}$$

0	0	0
0	1	1
1	0	1
0	0	1
0	1	0

Figure IV-1. Three State Assignment Columns

The basic column set is the minimum number of state assignment columns that must be investigated to arrive at an optimal state assignment. The basic column set is significantly smaller than the total number of distinct state assignment columns. For example, a five state machine would have 65,535 distinct state assignment columns, but the minimum set has only 15 columns (Ref 8: Table 1). One can see that if all states were investigated, state assignment problems would quickly require impossible amounts of computer time, like the classic traveling salesman problem.

The method for determining the basic state assignment column set depends on whether the number of states, R, is equal to one more than a power of two. If $R = (2^m + 1)$ where m is any integer, then the state assignment columns can be listed directly (Ref 8:874). The y-variables are subscripted in the same manner as Figure IV-1.

$$y_1, y_2, y_3, \ldots, y_{2^{R-1}-1}$$

In all other cases the y-variables must be generated. The formula for generating the y-variables is shown below (Ref 8:874). The number, n, is an integer equal to log (R).

Number of bits in assignment

$$\binom{R}{R-2^{n-1}} \binom{R}{R-2^{n-1}+1} \cdots \binom{R}{2^{n-1}}$$

Number of 1 bits in assignment

IV-4

The complements of each of these y-variables must also be determined for the D flip-flop case. The y-variable complements can be listed directly, regardless of R, by:

$$y_{\overline{i}} \quad \text{where: } \overline{i} = 2^R - i - 1$$

The basic column set includes the y-variables generated by the appropriate method and their complements. An example for R = 5 is shown below. Note that although the y-variables can be simply listed, they can also be generated by the method used when R is even. Figure IV-2 shows the variables generated by simply listing them and the corresponding binary values.

y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Figure IV-2. Basic Column Set For R = 5

The generation method for R is shown in Figure IV-3 for comparison purposes with R = 5. The variable, r1, refers to the first row as shown in Figure IV-2. The r variables to the left of the vetical bar are 0s and the variables to the right are 1s. Each group of row variables describes a single column shown in Figure IV-2. For example, the first group specifies y_{15} .

$$\binom{5}{1} = (r1|r2, r3, r4, r5), (r1, r3, r4, r5|r2), (r1, r4, r5, r2|r3),$$
$$(r1, r5, r2, r3|r4), (r1, r2, r3, r4|r5)$$

$$\binom{5}{2} = (r1, r2|r3, r4, r5), (r1, r3|r4, r5, r2), (r1, r4|r5, r2, r3),$$
$$(r1, r5|r2, r3, r4), (r1, r2, r3|r4, r5), (r1, r3, r4|r5, r2),$$
$$(r1, r4, r5|r2, r3), (r1, r5, r2|r3, r4), (r1, r5, r3|r4, r2),$$
$$(r1, r4, r2|r3, r5)$$

Figure IV-3. Basic Set Column Generation For R = 5

<u>Step 2 - Cost Estimation.</u>   Cost estimation is used to derive the minimum cost of a particular state assignment column. Cost is defined as the number of inputs to a two level AND-OR gate array. The cost estimation method uses a modified Karnaugh map. The inputs are assumed to have a predetermined binary code.   The input codes are arranged across the top of the modified map exactly like a Karnaugh map. The state symbols, S , are listed along the vertical axis of the state table. The modification assumes that all rows that are a power of two are group adjacent (Ref 4:1368). "This assumption provides a degree of adjacency between the S minterms that is as great or greater than can exist for an actual coding of the S minterms, and consequently it provides a lower bound on the cost of an excitation expression regardless of the coding that may be subsequently assigned to the S minterms (Ref 4:1369)." Later steps that determine the actual costs of the y-variable state assignment do not follow this last assumption.

Noe and Ryhne further restrict the cost estimation procedure by requiring that unassigned states in the y-variable be assigned a 1 or 0

such that the number of 0's equal the number of 1's. They also require that any group of states that cross the boundary between y = one/zero contain an equal number of states from each side of the boundary. This idea is illustrated in the Example section.

The additional restrictions made by Noe and Ryhne result in a set of minimum numbers that is equal to or greater than the minimum numbers generated by an unmodified SHR procedure. This can reduce the number of trials made by the Story, Harrison, and Reinhard (SHR) search procedure (Ref 9:328).

The cost estimation procedure begins by assigning each state in the state table a 1 or 0 based on the particular y-variable. The unassigned state table rows are then assigned a 1 or 0 so that the total number of rows with y = 0 equals the number of rows with y = 1. Don't care symbols are denoted by a "x". The 1's and x's are grouped together like Karnaugh maps with two exceptions. The first is that groups that cross the y = 0/y = 1 boundary must have an equal number of terms on each side of the boundary. The second, discussed above, is that rows that are a power of two are considered group adjacent.

Once the states have been properly grouped together, the cost estimate algorithm is shown below. The cost estimation procedure is repeated for the complements of each of the y variables. The lesser of the two estimates is then used as the minimum possible cost for the y-variable in the ordered search.

$$C = \begin{cases} NL + NT - SLT & ; NT > 1 \\ NL & ; NT = 1, SLT = 0 \\ 0 & ; NT = 1, SLT = 1 \end{cases}$$

Where:

NT = Number of Terms
NL = Number of Literals = $m + n - q$
SLT = Number of Single Literal Terms
m = log (Number of Inputs)
n = log (Number of State Variables) = log (R)
q = log (Number of Literals in Group)

m, n, and q are integers

Figure IV-4. Cost Equations

This cost estimate applies directly to a PLA. NL, the number of literals, is the number of product terms that must be generated in the PLA. The area required by the PLA increases proportionately with NL. NT and SLT are related to the power consumption of the PLA. NT is the number of pass transistors on a given product term line. The larger NT is, the more power is required to generate the particular term. SLT refers to a product term that has only one pass transistor. Power consumption decreases with larger values of SLT for a given total number of inputs.

Step 3 - Sort. The y-variables are sorted in nondecreasing order based on the cost estimate obtained previously. A bubble sort is used which generally takes $O(n^2)$ time (Ref 11:257). In this case, the bubble sort does not have a significant affect on the execution time. The sort is used only once on a small number of integers. These integers are the

cost estimates for each of the y-variables. When the integers are moved in their array, two associated character strings are moved. The overhead is extremely low since only integers are compared and the number of sorted items is small. For example, when the number of states is equal to five, there are only 15 y-variable sets to sort.

Step 4 - Ordered Search. The last step is the ordered search of the y-variable sorted list for an optimum state assignment. The first step is to find a valid state assignment. State assignments are made with n state assignment columns or y-variables. The number, n, is equal to $\log_2(R)$. The first assignment scheme is made by pairing the first y-variable in the sorted list with the appropriate number of successive variables. A valid assignment has a unique designation for each state in the state table. Not all combinations of the state assignment columns form a valid assignments. Once a valid assignment scheme is found, the actual cost of each y-variable is determined by the same method as cost estimation except that commonality of terms is not considered (Ref 4:1369). This means that the restrictions added by Noe and Rhyne are removed and rows that are powers of two are not considered group adjacent.

The minimum and actual cost estimates for each y-variable in the scheme are then totaled and the minimum and actual cost are compared. The search stops when the actual cost is equal to the minimum cost or if the next state assignment scheme has a higher minimum cost than the current actual cost. Figure IV-5 shows a flow chart of the search procedure.
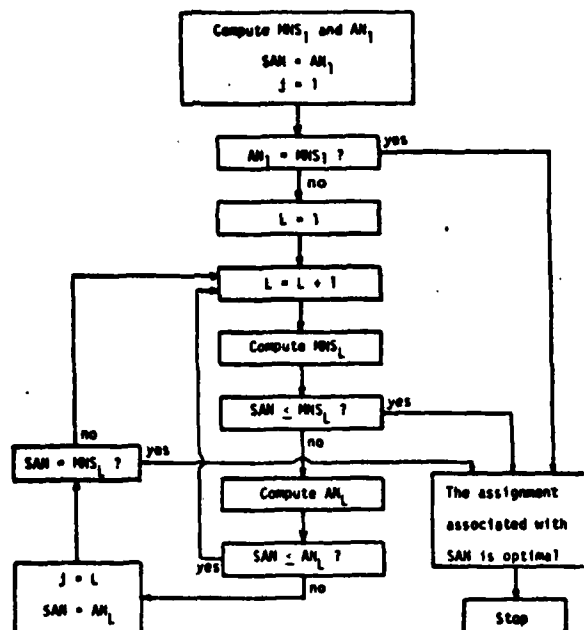
MNS = Minimum Number Sum (Cost Estimate)
AN = Actual Number (Actual Cost)
SAN = Smallest Actual Number

Figure IV-5. Flow Chart of State-Assignment Optimization Algorithm
(Ref 4:1370)

Example

A five state transition table is shown below in Figure IV-6 (Ref 9:328). Since the number of states, R, is equal to 5 which is also equal to $2^2 + 1$. Therefore, the state column variables can be simply

listed. They are shown in Figure IV-2. If a PLA were a J-K flip-flop device, only these columns would have to be investigated. Since a PLA acts like a D flip-flop, the complements of these state column variables must also be investigated. The complements are obtained by inverting the bit values in the state column variable.

PS	NS	
1	5	1
2	4	2
3	3	3
4	2	3
5	1	5

Figure IV-6. State Transition Table With Five States

Figure IV-7 shows the state transition table with the state column variable y applied against the transition table. The y variable overlays the present state column of the state transition table. States overlayed by a '1' are replaced by a '1' in the Next State portion of the transition table. Likewise, states overlayed by a '0' are replaced with zeros in the transition table. Note that the number of states in the transition table is now a power of two. The new states are labeled with a 'x' and represent don't care conditions.

PS	NS	
0	1	0
0	0	0
1	1	*1
0	0	1
1	0	1
x	1	1
x	1	1
x	1	1

Figure IV-7. State Transition Table With Y_5 Applied

Actual Cost Calculation. The actual cost for this assignment is calculated from the table in Figure IV-7. The ones in the state transition table are assigned binary numbers in an array like manner. The binary numbers are then changed to a gray code (Ref 12:338). The rows and columns in the transition table are labeled 0 to 7 and 0 to 1 respectively. For example, the one next to the asterisk in Figure IV-6 would be assigned a binary value of 0101. This binary assignment is derived from the placement of the one in row 2 and column 1. The binary value is therefore $2 * C + 1$. C is equal to the number of columns, which is two in Figure IV-7. Note that the first three bits from the left designate the state and the last bit denotes the state transition column.

This binary number is changed to a gray code through an exclusive-or operation. The exclusive-or operation is applied to the most significant bit and a zero. The exclusive-or operation is repeated for the most significant bit and the next most significant bit. This operation is "rippled" through the portion of the binary number that designates the state. Bits that denote the column are not changed. This is done so that table values in the same column remain adjacent. In Figure IV-7, only the left-most three bits are changed to a gray code. The results of the exclusive-or operations form the new gray code variable. The zero is in effect pushed through the binary variable. This is shown below:

start zero - 0 / 0101 - binary variable

0111 - result of exclusive-or

PS	NS	
0	1	0
0	0	0
1	1	1	②
0	0	1
1	0	1
x	1	1	②
x	1	1
x	1	1	③

$$2 + 2 + 3 + 3 = 10$$
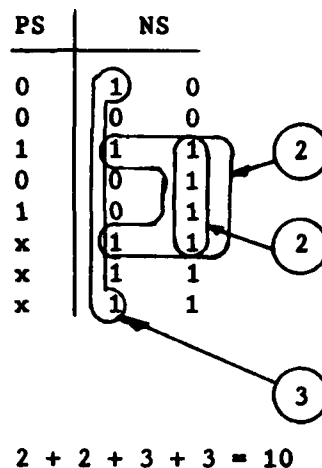
Figure IV-8. State Transition Table With Y_5 Applied

Once all the ones in the state transition table have been converted
to a gray code, a standard Quine-McClusky algorithm is used to determine
the minimum cover required to implement the table. The cost for the
table is calculated using the equations in Figure IV-4. The groupings
and cost calculations are shown in Figure IV-8. Each group of four
terms has a cost of two. The group with two terms costs three AND-OR
gates. There are three groups which makes the total cost ten. The
computer representation of one of the four term groups is -11-. The
cost of this group is simply calculated by counting the number of 1s or
0s in the in the term. Thus, this group has a cost of two AND-OR gates.

The cost calculation is repeated for the complement of y_5, y_{26} . The
state transition table for y_{26} is shown in Figure IV-9. This figure also
shows the grouping and cost calculation. The y-variable with the lower
cost is used for the cost calculation. In this case, since y_5 and y_{26} are
equal, y_5 is chosen.

```
PS │   NS
 1 │ 0   1
 1 │ 1   1 ── 2
 0 │ 0   0
 1 │ 1   0
 0 │ 1   0 ── 3
 x │ 1   1
 x │ 1   1 ── 2
 x │ 1   1
```

$$2 + 3 + 2 + 3 = 10$$

Figure IV-9. State Transition Table With Y_{26} Applied

Cost Estimation. The cost estimation procedure is accomplished in the same manner as the actual cost estimation. There are, however, two exceptions. First, the state transition table is rearranged so that each half of the table has all "one" or all "zero" rows. The "type" of the row is determined by the y-variable. For example, the first row in Figure IV-9 is a one row. Figure IV-10 shows the rearranged table of Figure IV-7 and its cost estimation. The dotted line denotes the zero/one boundary of the table. All rows in the upper half are either a zero row or a don't care row. The lower half has only one and don't care rows.

```
PS │   NS
0  │ 1    0          3
0  │ 0    0
x  │ 1    1
0  │ 0    1          2
- -│- - - - - - -
1  │ 0    1
1  │ 1    1          2
x  │ 1    1
x  │ 1    1
```

3 + 2 + 2 + 3 = 10 .

Figure IV-10. State Transition Table With Y_5 Applied

The second difference is that groupings that cross the zero/one
boundary must have an equal number of terms on each side of the
boundary. Figure IV-11 shows the rearranged table and groupings for y_{26} .
Since the complement of the y_5 has a lower cost estimate, its value is
used as the cost estimate for y_5.

```
PS │   NS
1  │ 0    1          2
1  │ 1    1
x  │ 1    1
1  │ 1    0
- -│- - - - - - -
0  │ 1    0
0  │ 0    0
x  │ 1    1
x  │ 1    1          2
```

2 + 2 + 2 = 6

Figure IV-11. State Transition Table With Y_{26} Applied

Node List

Node A121: Generate Basic State Columns

Node A122: Estimate Costs

Node A123: Bin Sort

Node A124: Ordered Search

Node A125: Format

Data Dictionary

Node A12:

Basic State Columns: The binary and decimal representations of the

   basic set of distinct state columns discussed in Step 1.

Cost Estimates: The cost estimates generated in Step 2 are

   associated with the Basic State Columns.

Sorted Cost Estimates: A list of the Basic State Columns and

   associated Cost Estimates in nondecreasing order based on Cost

   Estimate. This list is generated by Step 3.

Optimum State Assignment: A subset of the Basic State Columns that

   form.a valid, optimum state assignment and its actual cost.

   The subset contains n state columns and is selected in Step 4.

Node A12

Nodes A121 - A124 correlate to Steps 1 - 4 discussed above. The last node, A125 Format, creates a file that is suitable for PRESTO. Format basically substitutes the optimum state assignment into the Reduced Symbolic State Table. This creates the binary PLA Truth Table.

Node: A12
Title: Assign States
Date: 11 Jul 83
Rev.: 1.0

I1 PLAFST Control

I2 Reduced Symbolic
State Table

Generate
Basic State
Columns

1

Basic State
Columns

Estimate
Costs

2

Cost Estimates

Bin Sort

3

Sorted Cost
Estimates

Ordered
Search

4

Optimum State
Assignment

Format

5

PLA Truth Table   O1
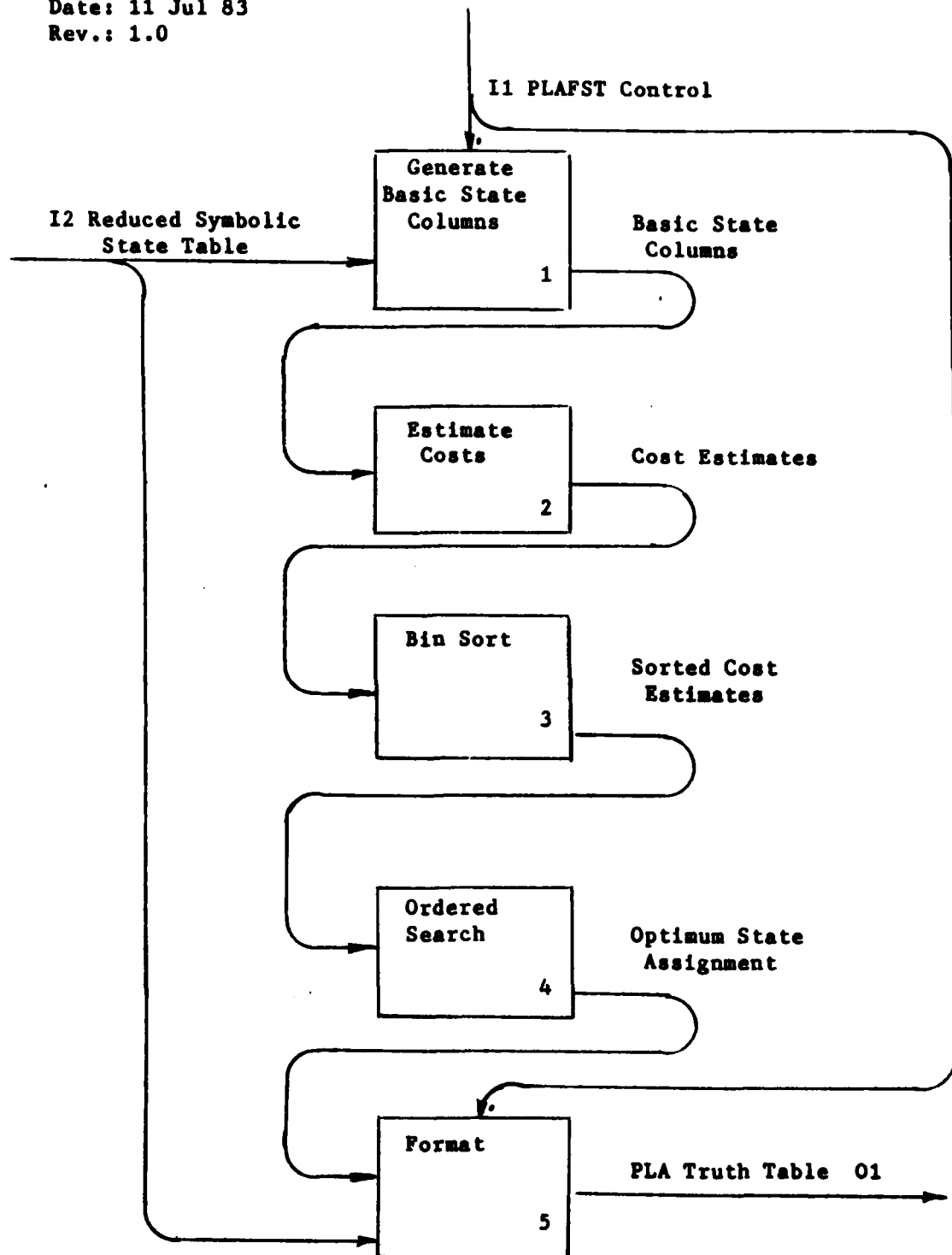
Figure IV-12.   Node A12

ANALYSIS

Overview

The analysis of PLAFST is divided into three parts. The first part looks at the purpose and outputs of the various programs within PLAFST. It analyzes what each program accomplishes. The second part of the analysis compares PLAFST against benchmark state transition tables used by others to compare results. Two individual programs, ASSIGN and SYM, are investigated in detail. ASSIGN, which computes the optimum state assignments, is also compared against its own options: Simple Assignment and Gray Code Assignment.

The third part is the sensitivity analysis. ASSIGN is the only program analyzed in this manner since it requires at least an order of magnitude more execution time than the other four programs written for this thesis. The sensitivity analysis uses state transition tables that vary only one parameter. The execution time for each state table is plotted and a Big-Oh analysis is done on the results.

Process

Figure V-1 shows the relationship between the PLAFST shell script and other resident programs on the VAX 11/780. SYM, ASSIGN, CFORM, and

MAKE_SFSM were written for PLAFST. SYM and ASSIGN comprise the bulk of the programs created during this thesis effort. CFORM translates PRESTO's output to the format required by PLAGEN. MAKE_SFSM decodes SYM's output and the SFSM_PLA CIF file to generate the SFSM CLL file. The SFSM CLL file includes the PLA, buffers, and the state variable interconnections.
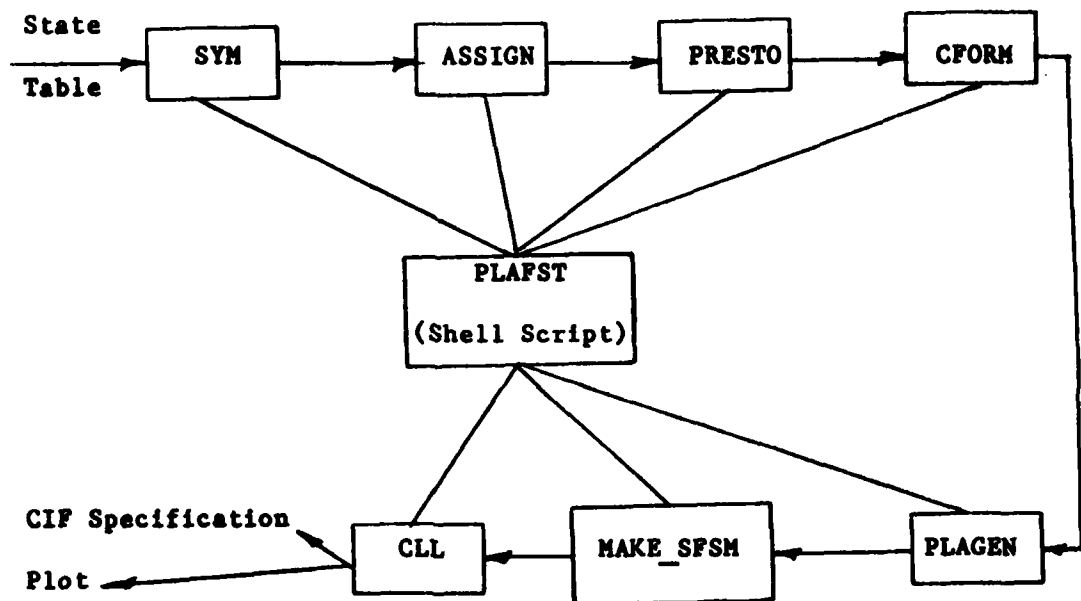
Figure V-1. PLAFST Structure Chart

SYM performs the first data reduction of the symbolic state table. The symbolic state names are translated to numbers starting with one. The symbolic input names are ignored. They are required merely for the programmers convenience. The output symbolic names are converted to strings of ones and zeros. One string is created for each state in the symbolic state table. Once the initial data processing is completed,

SYM tries to reduce the state table through equivalence partitioning. The initial symbolic state table and the output from SYM are shown in Figure V-2. In this case no symbolic reduction is possible. The fact that an error state was not designated in the original symbolic state table is shown by a zero at the end of the first line of SYMs output.

INITIAL SYMBOLIC STATE TABLE	OUTPUT FROM SYM
8 1 1 963 2.5	8 1 1 963 2.5 0
A	
B	1 2
C	3 2
D	3 4
E	5 4
F	5 6
G	7 6
H	7 8
input_1	1 8
output_1	
A B C B C D E D E F G F G H A H	00
0 0 0 0 0 0 0 0 0 0 0 0	00
output_1 0 output_1 0	00
	00
	00
	00
	01
	10

Figure V-2. Initial Symbolic State Table and PLAFST Reduced Table

ASSIGN is executed next. It uses the state transition table from SYM. The output strings are placed in an array until they are appended to the output. ASSIGN does not use them during the state assignment process. ASSIGN generates the distinct column set for the number of states in the state transition table. It then calculates a cost

estimate for each column set using the transition table and performs an
ordered search to determine an optimum assignment.   This process is
discussed in detail in Chapter IV, Example. Two other choices for state
assignment exist.   The first is simple assignment. The states are
assigned successive binary numbers starting with 0.   In Figure IV-2,
State 1 from SYMs output would be assigned a value of 0000.   State 8
would be assigned a value of 0111. The other option is a gray code
assignment. State assignments are made by the Simple Assignment portion
of the code and then translated to a gray code.   Once the state
assignment is determined, an output file like that in Figure V-3 is
generated.   Figure V-3 shows the result of the optimum state assignment
process. This data is in the format required by PRESTO.

```
.i4
.o4
.p16
0000 0000
1000 1000
0100 0010
1100 1000
0001 0010
1001 1010
0101 0110
1101 1010
0011 0110
1011 1110
0111 0100
1111 1110
0010 0101
1010 1100
0110 0001
1110 1100
.e
```

Figure V-3. Output from ASSIGN

ASSIGN does not consider commonality between PLA product terms. For this reason, PRESTO is used to reduce the size of the PLA. PRESTO was able to reduce the product terms from 16 to 11 in this example. The output from PRESTO is shown in Figure V-4.

```
.i4
.o4
.p11
010- 0010
1--- 1000
--01 0010
1--1 1010
0101 0110
-011 0110
1-11 1110
--11 0100
0010 0101
1-1- 1100
0-10 0001
.e
```

Figure V-4. Output from PRESTO

The output from PRESTO is piped through CFORM to PLAGEN. CFORM acts as a translator. It changes PRESTOs output to a format acceptable to PLAGEN and adds the CIF number and lambda size to the file. PLAGEN generates the CIF specification for the PLA that implements the original symbolic state table. The output including data sent to stderr from PLAGEN is sent to a file called foo. The data sent to stderr is the bounds of the PLA. It can appear anywhere in the output file and will usually cause an error if the file is used directly by CLL. For this reason, MAKE_SFSM reads foo and generates a duplicate file with the bounds data deleted. This includes the newline character that follows the bounds data. This insures the integrity of the PLA CIF file.

MAKE_SFSM uses this information and the state transition table size, obtained from SYMs output, to generate a CLL file that includes the PLA CIF file.

Cll is the final program executed. It uses both of the files generated by MAKE_SFSM to generate the final plot of the synchronous finite-state machine (SFSM) created from the original symbolic state table in Figure V-1. The SFSM CLL file is shown in Figure V-5. The plot of the SFSM is shown in Figure V-6.

```
#include "/usr/lib/local/s_ext.cll"
external pla (cif 963 bounds --15,0 140,111)

SFSM
§

pla(0,0);
iterate 4, 1 16, 0
    PlaClockIn ( 15, --58 );
iterate 2, 1 16, 0
    PlaClockOut ( 92, --53 );
wire poly 95, --53 w 2 d 12 1 22 diff u 7 ;
wire poly 103, --53 w 2 d 22 1 46 diff u 17 ;
wire poly 111, --53 w 2 d 32 1 70 diff u 27 ;

§
```

Figure V-5. SFSM CLL File
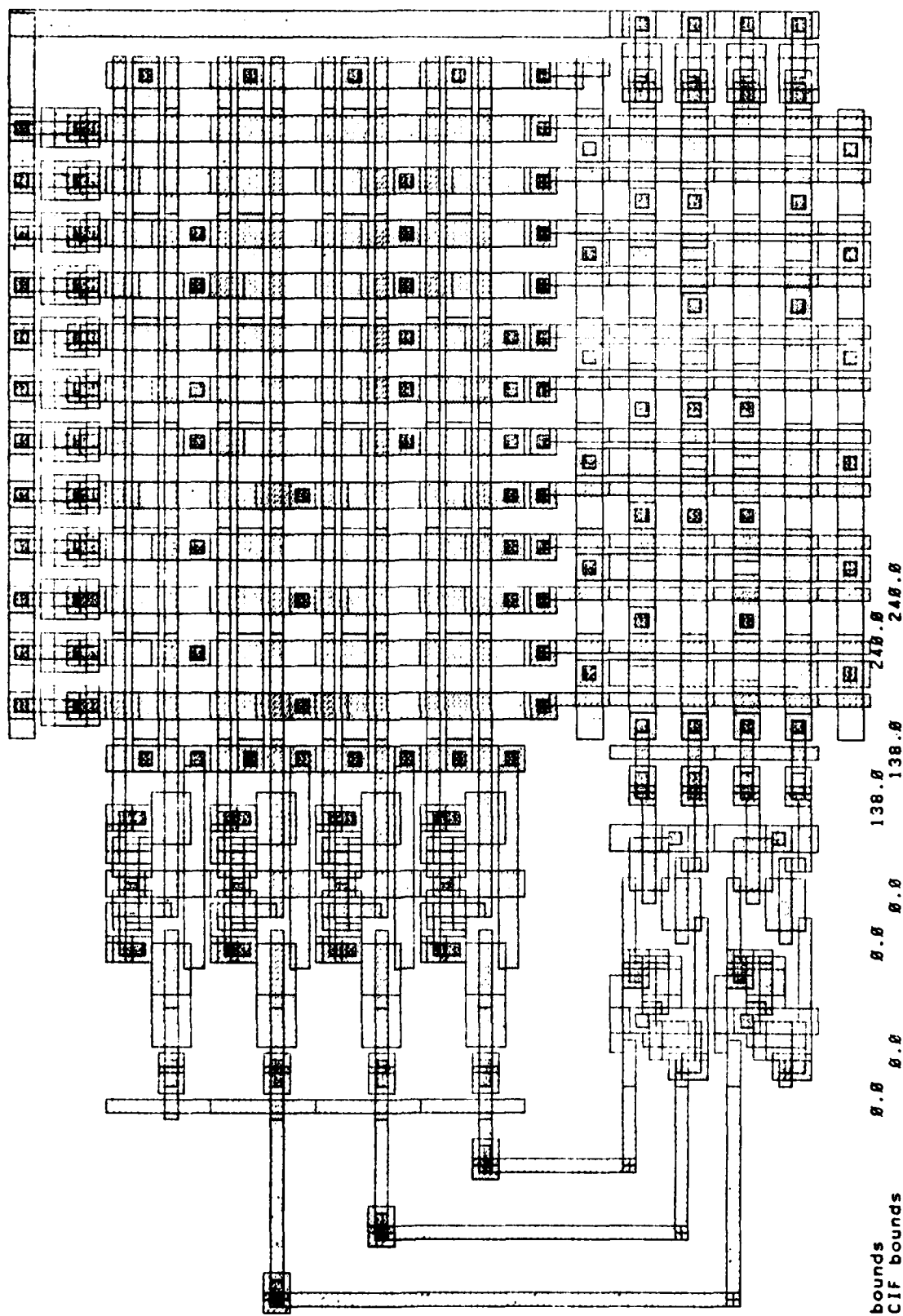
Figure V-6. Plot of SFSM Generated from Figure V-1.

Benchmark

In this section PLAFSTs performance is compared against the benchmark state tables and assignment schemes created by other authors. ASSIGNs options are also compared against each other. The execution times for PLAFST, ASSIGN, and SYM are provided. The table numbers below refer to Appendix A. The cost is the number of AND-OR gates required to implement the state transition table with the given state assignment.

PLAFST only generates a near optimum solution to the state assignment problem. This is due to the cost estimation algorithm. The debug option was used to show the internal calculations. PLAFST matched the cost estimates of Noe and Rhyne for Table A-15 on 7 of the 15 cost estimates. One of the estimates that did not match was lower than Noe and Rhyne. The other estimates indicated two to three additional gates were required. This difference in cost estimates significantly affects the the ordered search for the optimum state assignment.

However, PLAFST actually calculated a better optimum state assignment for Table 14. It also matched the optimum state assignment for Table 12. PLAFST assignments are usually within a few gates of the "optimum" solutions for the benchmarks state transition tables.

Table V-1. Benchmark Comparison

Table Number	Author	State Assignment	Cost
6	Hartmanis	15, 51, 170	22
	PLAFST	30, 51, 180	23
7	Torng	6, 8, 18	39
	PLAFST	9, 15, 26	42
8	Dolotta & McCluskey	2, 7, 9	40
	Torng	2, 5, 17	39
	PLAFST	5, 17, 29	42
9	Dolotta	3, 6, 19	19
	PLAFST	14, 24, 29	25
10	Curtis	15, 19, 21	43
	PLAFST	6, 11, 33	63
11	Curtis	67, 101, 106	75
	PLAFST	26, 67, 106	114
12	Dolotta	15, 60, 85	20
	PLAFST	15, 60, 85	20
13	Dolotta	3, 21, 36	36
	PLAFST	21, 26, 54	39
14	Dolotta	27, 46, 105	81
	PLAFST	170, 180, 198	73
15	Noe & Rhyne	4, 7, 14	12
	PLAFST	2, 14, 24	14

Table V-2 shows the execution times for PLAFST and ASSIGN for each of the tables in Appendix A. Execution times are also given for the Simple assignment and Gray Code options of ASSIGN. PLAFST and its associated programs were executed on the AFIT VAX 11/780 located in building 641, WPAFB, OH. The times in Table V-2 are in seconds and were provided by the UNIX operating system command "time". The smallest

division is tenths of a second which makes differences of one-tenth insignificant.

Table V-2. Execution Times (Seconds) for PLAFST and ASSIGN

Table Number	# States /# Inputs	PLAFST	ASSIGN Optimum	Simple	Gray
A-1	3/1	16.5	0.5	0.2	0.3
Figure II-1	4/3	56.2	1.6	0.5	0.7
A-2	4/1	17.2	0.5	0.3	0.4
A-3	5/1	21.8	2.1	0.3	0.6
A-4	6/1	26.2	2.9	0.3	0.6
A-5	7/1	26.9	2.6	0.3	0.6
A-6	8/1	28.0	2.3	0.4	0.5
A-7	5/2	43.0	9.5	0.3	0.7
A-8	5/2	42.3	8.9	0.4	0.6
A-9	5/1	24.0	2.3	0.3	0.6
A-10	6/2	50.3	14.7	0.3	0.5
A-11	7/3	197:5	73.2	0.7	1.3
A-12	8/1	25.8	2.1	0.4	0.7
A-13	6/2	52.4	14.0	0.5	0.7
A-14	8/2	55.4	7.3	0.5	1.0
A-15	5/1	26.0	2.6	0.4	0.4

Several general observations can be made from this table. In all cases, Simple and Gray took less time than Optimum. This is expected since Simple and Gray both execute in $O(n)$ time. The next section shows that ASSIGN is greater than $O(n^2)$. Specific comparisons are difficult to make since several parameters vary between the state transition tables.

The two significant parameters for ASSIGN are the number of states and the number of inputs. As the number of states increases, so does the number of distinct state column sets that must be generated. Table

V-3 shows the number of distinct state column sets for up to sixteen states. The number of distinct state column sets rises exponentially after eight states.

Table V-3. Number of Distinct State Assignment Column Sets
(Ref 4:1367)

Number of States	Distinct State Column Sets
2	1
3	3
4	3
5	15
6	25
7	35
8	35
9	255
16	6435

An increase of one in the number of inputs will double the size of the state transition table. Table V-2 shows that state transition tables with close to eight states or more than one input, require significant increases in execution time. State transition tables with seven states and three inputs like Table A-11 have greatly increased execution times.

The optimum state assignment costs for ASSIGN are compared with the simple and gray code assignment options in Table V-4. The table shows that in all cases the optimum state assignment costs less than the simple or gray assignments. The simple assignment is cheaper than the gray assignment for some state tables and more costly for others. There

is not any pattern as to which of these schemes is better for any given state table.

Table V-4. Optimum, Simple, and Gray State Assignment Comparison

Table Number	ASSIGN	SIMPLE	GRAY
A-1	9	20	16
Fig. II-1	17	38	20
A-2	14	17	16
A-3	17	25	33
A-4	21	39	42
A-5	26	51	47
A-6	23	26	41
A-7	42	62	76
A-8	42	65	62
A-9	25	42	29
A-10	63	107	103
A-11	114	160	168
A-12	20	21	31
A-13	39	102	132
A-14	73	99	97
A-15	14	27	29

Table V-5 points out some interesting facts about where ASSIGN spends most of its time. The execution times and number of iterations can not be directly compared since different state transition tables can cause great differences in the Quine-McCluskey algorithm. Some general observations and assumptions can still be made though.

The majority of iterations are small numbers. The worst case for state tables with more than four states is that three sets of two state assignment columns must be calculated for each iteration. In practice,

the number of calculations is greatly reduced since ASSIGN remembers the cost associated with each state column set and does not recalculate it. Table V-3 shows the number of distinct state assignment column sets that must be calculated. For example, Table A-14 has eight states which means that 35 state column sets must be calculated. This translates into 70 state tables which must be reduced with the Quine-McCluskey algorithm.

Table V-5. ASSIGN: Execution Time and Number of Iterations

Table Number	# States /# Inputs	ASSIGN Optimum	Iterations
A-1	3/1	0.5	1
Figure II-1	4/3	1.6	0
A-2	4/1	0.5	2
A-3	5/1	2.1	2
A-4	6/1	2.9	5
A-5	7/1	2.6	2
A-6	8/1	2.3	5
A-7	5/2	9.5	2
A-8	5/2	8.9	1
A-9	5/1	2.3	9
A-10	6/2	14.7	29
A-11	7/3	73.2	2
A-12	8/1	2.1	1
A-13	6/2	14.0	14
A-14	8/2	7.3	4
A-15	5/1	2.6	10

However, Table A-14 takes only four iterations to arrive at a solution. The worst case is that 24 state tables would have to be solved. As discussed, above this number is probably considerably less. Thus, it is reasonable to assume that ASSIGN spends most of its execution time calculating cost estimates. This seems to be borne out

by state tables with zero or one iterations which still have a significant amount of execution time. This becomes important for future efforts to reduce PLAFSTs execution time. Faster methods of estimating the cost of a state column set may yield great decreases in the execution time.

Table V-6. SYM Execution Times

Table Number	# Original States	# Reduced States	Time (s)
A-1	7	3	0.2
A-2	6	4	0.2
A-3	6	5	0.1

Table V-6 shows the execution times for SYM with those tables that were reduced symbolically. Times for other state tables range from 0.1 to 0.3 with no state reduction.

Sensitivity

This analysis tested ASSIGNs sensitivity to the two parameters discussed earlier, the number of states and the number of inputs. The number of cost estimations greatly increases with the number of states. This is shown in Figure V-3. Each additional input will double the size of the state transition table since PLAFST requires that the SFSM be completely specified.

State transition tables were constructed that varied only one of these parameters at a time and tried to maintain the same type of state transitions throughout the test data.   Figure V-7 shows the first, middle, and last state transition table used for the state sensitivity analysis.

First State Table	Middle State Table	Last State Table
2 1 1 950 2.5 0	5 1 1 950 2.5 0	9 1 1 950 2.5 0
2 2	2 3	2 3
1 1	3 4	3 4
	4 5	4 5
00	5 5	5 6
10	1 1	6 7
		7 8
	00	8 9
	00	9 9
	00	1 1
	00	
	10	00
		00
		00
		00
		00
		00
		00
		00
		10

Figure V-7. State Transition Tables Used for State Sensitivity

The progression used to extend the state transition tables can be easily seen.   Each state transitions to the next state on a zero input and to the next state plus one on a one input. This process continues

until the highest state is reached. All states after this point transition to the highest state except the last state. The last state transitions to the first state in all cases. Since ASSIGN does not consider the outputs, only one output is high in order to keep the solution from being trivial. These state transition tables can transition to any state and are strongly connected. The execution times for the eight test state transition tables are in Table V-7.

Table V-7. Execution Times for State Sensitivity Analysis

Number of States	Time (s)
2	0.2
3	0.4
4	0.4
5	2.0
6	2.7
7	3.0
8	2.2
9	1000+

The results are plotted in Figure V-8. The series of curved plots is grossly approximated by the upper straight line. This gives an upper bound to the time vs. state response of ASSIGN. The upper bound can in turn be approximated by:

$$O(n) = n^{(\log_2 n)-1}$$
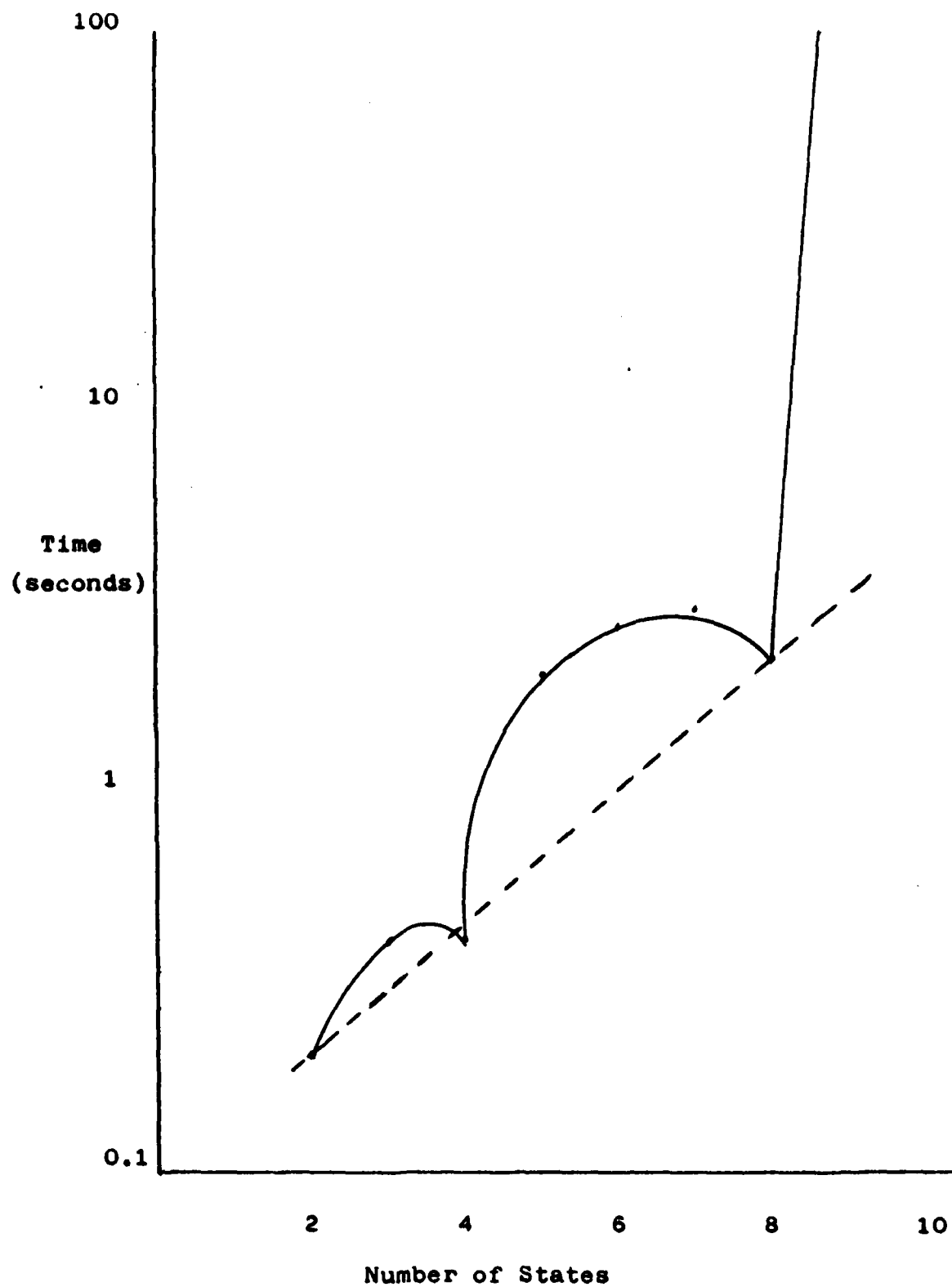
Figure V-8. Execution Times for State Sensitivity Analysis

The data shows that ASSIGN works well for state transition tables with less than nine states. When state transition tables with nine or more states are solved, the execution time grows exponentially to very large values. It is interesting to note that when the number of states is exactly equal to a power of two, the execution time decreases significantly.

The state transition tables for the input sensitivity analysis start with the five state transition table used for the state sensitivity analysis. The next three state transition tables are shown in Figure V-9. Each time the number of inputs is increased, the rows in the state transition table are repeated.

```
5 2 1 950 2.5 0      5 3 1 950 2.5 0       5 4 1 950 2.5 0

2 3 4 5              2 3 4 5 2 3 4 5       2 3 4 5 2 3 4 5 2 3 4 5 2 3 4 5
3 4 5 5              3 4 5 5 3 4 5 5       3 4 5 5 3 4 5 5 3 4 5 5 3 4 5 5
4 5 5 5              4 5 5 5 4 5 5 5       4 5 5 5 4 5 5 5 4 5 5 5 4 5 5 5
5 5 5 5              5 5 5 5 5 5 5 5       5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
1 1 1 1              1 1 1 1 1 1 1 1       1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

00000               00000000              0000000000000000
00000               00000000              0000000000000000
00000               00000000              0000000000000000
00000               00000000              0000000000000000
10000               10000000              1000000000000000
```

Figure V-9.   State Transition Tables for Input Sensitivity Analysis
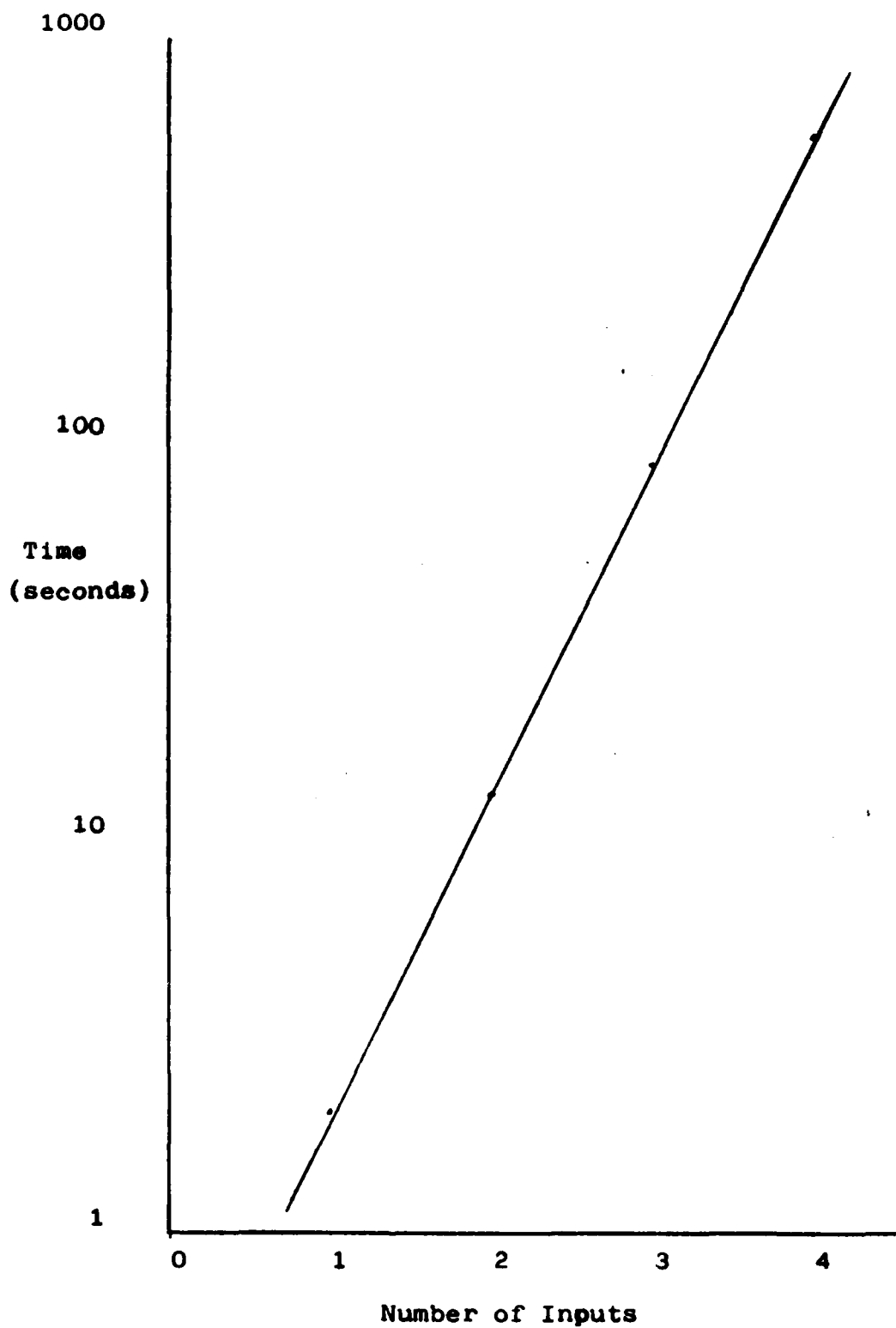
Figure V-10. Execution Times for Input Sensitivity Analysis

Table V-8. Execution Times for Input Sensitivity Analysis

Number of Inputs	Time (s)
1	2.0
2	11.4
3	82.3
4	589.7

The data from the input sensitivity analysis is shown in Table V-8 and plotted in Figure V-10. The results are a straight line on a logarithmic plot. They can be closely approximated by :

$$O(n) = n^n \log_2 n$$

The test data shows that ASSIGN is restricted to state transition tables with less than nine states and four inputs for reasonable execution times. ASSIGN also begins to require very large amounts of memory to run. The array sizes had to be increased from those in Appendix E in order to run the five state transition table with four inputs.

Conclusions

PLAFST meets its goals. It converts a symbolic state table into an integrated circuit. The integrated circuit includes a programmable logic array, clocked buffers, and interconnected state variables. PLAFST works well for symbolic state tables with less than nine states and four inputs. Symbolic state tables larger than these constraints cause inordinate increases in execution times.

PLAFST produces a near optimum state assignment that in some cases meets or exceeds published benchmarks. In other cases, its state assignment cost is relatively close to that of the published benchmarks. PLAFST is sensitive to both the number of states and number of inputs in the transition table. An exponential growth curve for execution time is exhibited for increases in the number of states or the number of inputs.

Overall, PLAFST provides a good computer aided design (CAD) tool for limited scale projects. Within its limitations, PLAFST will allow the synchronous finite-state machine (SFSM) designer to concentrate on the details of the SFSM design rather than those of the integrated circuit.

Recommendations

There are three recommendations for follow on thesis work on PLAFST. PLAFST should be integrated into the AFIT CAD environment. This would include a user manual for each of the programs developed for PLAFST and additional options for partial outputs.

The state transition tables used to test PLAFST represented benchmarks and state transition tables designed to test sensitivities. A Monte Carlo method could be used to develop a uniform set of state transition tables to throughly test and analyize PLAFSTs performance. This approach could detect the portions of PLAFST that could most benefit from further attention and increase the capabilities of PLAFST. The analysis should also investigate memory usage versus state table size and the merits of allowing only 2^n state tables. Figure V-8 shows a significant execution time decrease for state tables where the number of states is a power of two.

The third recommendation is to use the assumption that PLAFST spends the majority of its time calculating cost estimates. If this assumption is true, then a faster cost estimation algorithm would greatly increase PLAFSTs capabilities. A code profiler would help to identify the portions of PLAFST most often executed. The cost estimation algorithm should be changed to use an intuitive process or artificial intelligence techniques rather than the exhaustive Quine-McCluskey solution to calculate the cost of a particular state assignment. Another approach would be to use the n-cube algorithm like PRESTO.

BIBLIOGRAPHY

Cited References
----

1. Muroga, Saburo. Logic Design and Switching Theory. New York: John Wiley & Sons, Inc., 1979.

2. Peters, Lawrence J. Software Design: Methods & Techniques. New York: Yourdin Press., 1981.

3. Hennie, Frederick C. Finite-State Models for Logical Machines. New York: John Wiley & Sons, Inc., 1968.

4. Story, J. R., H. J. Harrison, and E. A. Reinhard. "Optimum State Assignment for Synchronous Sequential Circuits," IEEE Transactions on Computers, Vol. C-21, NO. 12:1365-1373, December 1972.

5. Noe, P.S. and V. T. Rhyne. "Optimum State Assignment for the D Flip-Flop," IEEE Transactions on Computers, 306-311, March 1976.

6. Newkirk, J., Mathews, R., Redford, J., and Burns, C. Stanford nMOS Cell Library. Stanford University, 1981.

7. Mead, C. and L. Conway. Introduction to VLSI Systems. Philippines: Addison-Wesley Publishing Company, Inc., 1980.

8. Noe, Philip S. "Remarks on the SHR-Optimal State Assignment Procedure." IEEE Transactions on Computers, 873-875, September 1973.

9. Noe, P.S. and V. T. Rhyne. "A Modification to the SHR-Optimal State Assignment Procedure," IEEE Transactions on Computers, 327-329, March 1974.

10. Rhyne, V. T. and P.S. Noe. "On the Number of Distinct State Assignments for a Sequential Machine," IEEE Transactions on Computers, 73-75, January 1977.

11. Aho, A. V., J. E. Hopcroft, and J. D. Ullman. Data Structures And Algorithms. Massachusetts: Addison-Wesley Publishing Company, 1983.

12. Chirlian, Paul M., Analysis and Design of Digital Circuits and Computer Systems. Illinois: Matrix Publishers, Inc., 1976.

Additional References

Curtis, Allen H. "Systematic Procedures for Realizing Synchronous Sequential Machines Using Flip-Flop Memory: Part I," _IEEE Transactions on Computers_, Vol. C-18, NO. 12:1121-1127, December 1969.

Curtis, Allen H. "Systematic Procedures for Realizing Synchronous Sequential Machines Using Flip-Flop Memory: Part II," _IEEE Transactions on Computers_, Vol. C-19, NO. 1:66-73, January 1970.

Kanbayashi, Yahiko. "Logic Design of Programmable Logic Arrays," _IEEE Transactions on Computers_, Vol. C-28, NO. 9:609-616, September 1979.

Parchmann, Rainer. "The Number of State Assignments for Sequential Machines," _IEEE Transactions on Computers_, 613-614, June 1972.

Sherwood, Will. "PLATO - PLA Translator / Optimizer," _Proceedings of the Symposium on Design Automation and Microprocessors_, 28-35, February 1977.

Weiner, Peter and Edward J. Smith. "On the Number of Distinct State Assignments for Synchronous Sequential Machines," _IEEE Transactions on Electronic Computers_, 220-221, April 1967.

Tang, Chao-Chih and Marilyn A. Tarpy. "An Algorithm for Deriving All Pairs of Compatible States by Closure Cases," _IEEE Transactions on Computers_, 202-207, February 1976.

APPENDIX A: TESTING

Overview

PLAFST was tested in two phases. The first phase ensured that the PLAFST programs were logically correct. The next state tables in Tables A-1 to A-6 (Ref 3:22, 48) were used to test SYM.C which performs the symbolic table reduction. Functional testing of ASSIGN.C which makes the optimum state assignment was done primarily with Table A-15. This table was used by Noe and Ryhne in their published papers on modifications to the Story, Harrison, and Reinhard procedure. Consequently, they provided detailed intermediate results of their procedure for this state table. This information was very helpful in determining ASSIGN.C's accuracy.

The second phase analyzes the performance of PLAFST's optimum state assignment algorithm. Analysis of PLAFST includes final state assignment, cost, and execution time. Tables A-6 through A-14 (Ref 4:1372) are benchmark next state tables used by Story, Harrison, and Reinhard (SHR) to compare their method with other state assignment algorithms. PLAFST options for straight binary and grey code assignments are also compared against the optimum and benchmark results. A sensitivity analysis is done with additional state transition tables which vary only one parameter. The analysis and results are in Chapter 5.

Logical Testing

PLAFST was executed with the debug option to show the input file conversion into the final SFSM PLA. The symbolic state tables used during logical testing are relatively short and PLAFST's manipulations were easily checked by hand. These state tables also checked the symbolic state reduction capabilities of PLAFST. The benchmark next state tables used in the analysis phase are only intended to compare optimum state assignments. They can not be symbolically reduced.

Analysis

Tables A-6 through A-14 are modified from the original SHR analysis by the addition of outputs. None of the state assignment schemes consider the outputs, but PLAFST requires outputs to correctly process the state tables. These state tables are benchmarks used by SHR and others. PLAFST's results are compared to the results of the other state assignment schemes for the D flip-flop. The authors and results of each of these methods are listed in Table A-16. As mentioned above, the analysis is based on the final state assignment, cost, and execution time. The execution times for each state table are only used as a guide to PLAFST's efficiency, since these times are not available for the other methods. Cost A, Table 16, is determined when commonality between product terms is not considered. Cost B considers product term commonality which in some cases reduces the cost significantly. PLAFST costs will be compared to Cost A since PLAFST's state assignment cost is

calculated before PRESTO is used to reduce the PLA through product term commonality. The entire analysis, including the sensitivity analysis is contained in Chapter 5.

Tables

Table A-1			Table A-2			Table A-3			Table A-4		
PS	x1	x2	PS	x1	x2	PS	x1	x2	PS	x1	x2
A	B/0	C/0	A	B/0	C/0	A	B/0	C/0	A	B/0	C/0
B	C/0	E/0	B	A/0	D/0	B	D/0	A/0	B	D/1	E/0
C	A/1	F/0	C	E/0	B/1	C	C/0	F/0	C	A/0	B/1
D	G/0	F/0	D	A/0	B/0	D	A/0	C/0	D	E/0	F/1
E	F/0	G/0	E	C/0	A/1	E	C/0	F/0	E	F/0	F/0
F	D/1	C/0	F	E/0	D/1	F	A/1	D/0	F	C/1	E/0
G	C/0	B/0									

Table A-5			Table A-6		
PS	x1	x2	PS	x1	x2
A	B/0	C/0	1	4/0	7/0
B	D/0	E/0	2	3/0	8/0
C	E/0	A/0	3	5/0	6/0
D	B/0	A/0	4	6/0	6/0
E	F/0	G/0	5	8/0	3/0
F	C/0	G/0	6	7/0	4/0
G	B/0	F/1	7	1/0	2/1
			8	2/1	2/0

Table A-7					Table A-8				
PS	x1	x2	x3	x4	PS	x1	x2	x3	x4
1	3/0	1/0	2/0	4/0	1	1/0	1/0	2/0	2/0
2	1/0	5/0	2/0	4/1	2	2/0	3/0	1/0	2/1
3	3/0	4/0	5/1	3/0	3	3/0	5/0	5/1	3/0
4	5/0	1/1	2/0	4/0	4	4/0	2/1	3/0	3/0
5	5/1	4/0	5/0	3/0	5	5/1	5/0	4/0	1/0

Table A-9

PS	x1	x2
1	1/0	2/0
2	2/0	3/0
3	3/0	4/0
4	4/0	5/1
5	5/1	1/0

Table A-10

PS	x1	x2	x3	x4
1	4/0	5/0	5/0	2/0
2	1/0	4/0	3/0	5/0
3	6/0	3/0	5/0	2/1
4	3/0	5/0	3/1	1/0
5	2/0	3/1	5/0	5/0
6	1/1	6/0	3/0	3/0

Table A-11

PS	x1	x2	x3	x4	x5	x6	x7	x8
1	1/0	1/0	1/0	1/0	1/0	1/0	1/1	1/0
2	2/0	2/0	2/0	2/0	2/0	2/1	2/0	2/1
3	3/0	7/0	6/0	3/0	2/1	5/0	4/0	2/0
4	4/0	2/0	6/0	4/1	2/0	2/0	4/0	2/0
5	5/0	7/0	2/1	5/0	2/0	5/0	2/0	2/0
6	1/0	6/1	1/0	1/0	6/0	4/0	1/0	6/0
7	1/1	1/0	7/0	1/0	7/0	1/0	5/0	7/0

Table A-12

PS	x1	x2
1	1/0	2/0
2	3/0	2/0
3	3/0	4/0
4	5/0	4/0
5	5/0	6/0
6	7/0	6/0
7	7/0	8/1
8	1/1	8/0

Table A-13

PS	x1	x2	x3	x4
1	1/0	4/0	5/0	2/0
2	4/0	1/0	2/0	5/0
3	1/0	4/0	3/0	2/1
4	4/0	1/0	2/1	3/0
5	1/0	6/1	5/0	2/0
6	6/1	1/0	2/0	5/0

Table A-14

PS	x1	x2	x3	x4
1	1/0	2/0	6/0	1/0
2	2/0	3/0	2/0	4/0
3	5/0	3/0	3/0	3/0
4	2/0	6/0	5/0	3/0
5	5/0	1/0	8/0	5/1
6	7/0	1/0	6/1	6/0
7	7/0	7/1	7/0	1/0
8	8/1	7/0	8/0	5/0

Table A-15

PS	x1	x2
1	5/0	1/0
2	4/0	2/0
3	3/0	3/0
4	2/0	3/1
5	1/1	5/0

Table A-16. State Assignment Method Comparison

Table Number	Author	State Assignment	Cost A	B
6	Hartmanis	15, 51, 170	22	22
7	Torng	6, 8, 18	39	39
8	Dolotta & McCluskey	2, 7, 9	40	40
8	Torng	2, 5, 17	39	39
9	Dolotta	3, 6, 19	19	17
10	Curtis	15, 19, 21	43	43
11	Curtis	67, 101, 106	75	75
12	Dolotta	15, 60, 85	20	20
13	Dolotta	3, 21, 36	36	33
14	Dolotta	27, 46, 105	81	68
15	Noe & Rhyne	4, 7, 14	12	

NAME

plafst - PLA implementation of a synchronous finite-state machine (SFSM)

SYNOPSIS

plafst [-s] [-d] [-sa, -gc] < symbolic_state.table

DESCRIPTION

PLAFST is a program that generates a PLA with clocked input and output buffers from a symbolic state table. The symbolic states are reduced using equivalence partitioning. An optimum binary state assignment is made. The state variable assignment and cost is sent to the standard error file. The cost is defined as the number of AND-OR gates within the PLA. The state outputs are properly interconnected. The buffers are PlaClockIn and PlaClockOut from the Stanford nMOS Cell Library. PLAFST generates both CLL and CIF descriptions of the SFSM and plots the integrated circuit.

The options for PLAFST are:

-s   Generate the PLA only. Do not include the buffers or state variable interconnections.

-d   Debug. Generates detailed information during the SFSM generation   process.

-sa   Simple Assignment. The states are assigned binary numbers in the   same order that they appear in the symbolic state table file.

-gc   Gray Code. Same as Simple Assignment except that the binary values   are converted to a gray code.

One state can be designated as the error state. Any undefined states will transition to this state. However, use of an error state will degrade the optimum solution. An error state has an asterisk (*) as its first character.

To use PLAFST you must create a symbolic state table in the format shown below:

#_of_states #_of_inputs #_of_outputs CIF_# lambda_#

Symbolic_state_name_#1
.
.

Symbolic_state_name_#n

Input_name_#1
.
.
Input_name_#n

Output_name_#1
.
.
Output_name_#n

Next State Array

Output Array

Where #_states is the number of symbolic states in the table
#_inputs is the number of inputs to the SFSM
#_outputs is the number of SFSM outputs
CIF_# is the number that the CIF symbol will have
lambda_# is the lambda scale factor for the CIF file

Symbol names are 25 characters or less, including the alphabet, 0 -
9, and the underline character, _ . The characters can be in any
order and case is significant. The order of the symbolic names is
critical. PLAFST uses their order in the file to decode the Next
State and Output arrays.

The Next State Array contains #_states times 2 raised to the
#_inputs Symbolic_state_names. The array contains #_states rows
and 2 raised to the #_inputs columns. The first row corresponds to
the first Symbolic_state_name in the file.

The Output Array has the same number of elements as the Next State
Array. If a state has no outputs for a given input, then a 0 must
appear in the appropriate place in the file. If several outputs
occur during a particular state, their names must be separated by
slashes, /, and any number of spaces.

An example is shown below:

```
/* #_states #_inputs #outputs CIF_# lambda_#          */
4 3 5 950 2.5

/* Designated error state and first symbolic state     */
*HG
HY
FG                     /* Symbolic_state_names          */
FY


car
long_timeout      /* Input_names                        */
```

```
short_timeout

s
h0
h1              /* Output_names              */
f0
f1

HG HG HG HG HG HG HY HY
HY FG HY FG HY FG HY FG
FY FY FY FY FG FG FY FY      /* Next State Array      */
FY HG FY HG FY HG FY HG

f0 f0 f0 f0 f0 f0 0  f0/s·
h1/f0 h1/f0 /s h1/f0 h1/f0/s h1/f0
h1/f0/s h1/f0  h1/f0/s h0/s h0/s h0/s  /* Output Array */
h0/s h0 h0 h0/s h0/s h1/f0 h1/f0/s
h1/f0 h1/f0/s h1/f0 h1/f0/s h1/f0 h1/f0/s
```

NOTE: The comments shown in this example can not be included in the input file!

SEE ALSO

presto, plagen, cll

BUGS

The current state assignment algorithm produces a near optimal solution only. The solution is still less costly than either simple or gray code assignment schemes.

```
#
# PLAFST is a shell script which intiates the various
# programs that operate on the symbolic state table given
# in the input file.
set noglob
if ($#argv > 5)then
    echo Too many arguments in the command line
    exit(1)
    endif
foreach i ($argv)
    if( "$i" == "-d" ) then
        set debug = -d
    else if ( "$i" == "-s" ) then
            set stop = -s
    else if ( "$i" == "-sa" ) then
            set simple = -sa
    else if ( "$i" == "-gc" ) then
            set grey = -gc
    endif
end
if ( $?simple && $?grey ) then
    echo Two assignment methods were chosen  !
    exit( 1 )
    endif
if ( $?simple ) then
    set code = -sa
else if ( $?grey ) then
    set code = -gc
endif

if ( $?debug  ) then
    sym -d sym.out
    if ( $?code ) then
        assign -d $code assign.out < sym.out
    else assign -d assign.out < sym.out
    endif
  else sym sym.out
      if ( $?code ) then
          assign $code assign.out < sym.out
      else assign assign.out < sym.out
      endif
endif

presto < assign.out | cform sym.out| plagen >& foo
make_sfsm foo sym.out > sfsm.cll
rm foo

if ( $?stop ) then
    exit(0)
    endif
cll -ls -g5.5 sfsm.cll sfsm_pla.clf
echo cll -ls -C sfsm.cll sfsm_pla.clf
```

```
/*  SYM.C   -  Fixed arrays                        */
/*  SYMREDUC - symbolic state table reduction checks the input file for
        errors and reduces the symbolic state table.  This program
        functions as a preprocessor to ASSIGN states.  SYMREDUC
        uses I/O redirection for input and output        */


#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define MAXSYMBOLS 100  /* Maximum number of symbol names for states,
                            inputs, or outputs                      */
#define MAXCOMBIN  500  /* Maximum for numinp * noutputs            */
#define MAXLEN 25 + 1   /* Maximum length for symbol names  plus 1 for
                            string termination       */
#define EOR -2          /* End of Record      */
#define EOP -1          /* End of Partition   */
#define NUMCOL 80       /* Number of columns on standard CRT   */
                /*   Used for visual output only - does not affect files */

/*     GLOBAL VARIABLES                          */

static int debug = FALSE, errorstate = NULL;
static int nstates, ninputs, noutputs, symbol, numinp = 1;
FILE *fout, *fopen();
float lambda;

/*     MAIN PROGRAM            */

main (argc, argv)
int argc;
char *argv[];
{
int flag;
char states[MAXSYMBOLS][MAXLEN], inputs[MAXSYMBOLS][MAXLEN],
     outputs[MAXSYMBOLS][MAXLEN], otable [MAXSYMBOLS][MAXCOMBIN]  ;
int stable[MAXSYMBOLS][MAXCOMBIN]  ;
register int j, k;

if (argc == 3)
    {
    debug = TRUE;
    if ( (argv[1][0] != '-') || (argv[1][1] != 'd') ) error (1);
    fout = fopen ( argv[2], "w");
    }
  else if (argc == 2)
            fout = fopen ( argv[1],"w");
        else error(1);
if ( fout == NULL ) error(9);

if ( debug )
```

```c
        fprintf(stderr," Hello!  You are now In the PLAFST DEBUG ZONE!\n");

    flag = scanf("%d %d %d %d %f",&nstates, &ninputs, &noutputs, &symbol, &lambda);
    if((flag == 0)||(flag == EOF )) error(2);

    for ( j = 0; j < ninputs; j++)
        numinp = numinp * 2 ;            /* Numinp  equals 2 raised to the ninputs */

    if (debug)              /* Print symbolic state table values from above */
        {
        fprintf(stderr,"The symbolic state table values were read in ");
        fprintf(stderr,"correctly. Their values are : \n");
        fprintf(stderr,"nstates %d,  ninputs %d\n",nstates, ninputs);
        fprintf(stderr,"noutputs %d,  CIF symbol %d\n",noutputs, symbol);
        fprintf(stderr,"lambda %1.1f  numinp  %d\n\n",lambda, numinp);
        };


    /* Load the character arrays from the input file.               */

    if ( debug )
        fprintf(stderr,"The symbolic names are now being loaded. n n");
    load ( states, nstates );
    load ( inputs, ninputs );  /* inputs are never used - included in the input
                                  file for clarity only     */
    load ( outputs, noutputs );

    if ( debug )
        {
        for ( j = 0; j < nstates; j++)
            fprintf(stderr,"MAIN: %s \n", states[j]);
        for ( j =0 ; j < ninputs; j++)
            fprintf(stderr,"MAIN: %s\n",inputs[j]);
        for ( j = 0; j < noutputs; j ++)
            fprintf(stderr,"MAIN: %s\n",outputs[j]);
        }


    if ( debug )
        fprintf(stderr,"The error state number is: %d.\n",errorstate);


    /* Load the state truth table integer representations for the state names.
       the first symbolic state in the input file becomes state #1   */

    loadstable( stable, states );

    /* Load the output truth table with the characters '1' and '0' suitable
       for PRESTO                          */

    loadotable( otable, outputs );
    if ( debug )
        fprintf(stderr,"MAIN: We are about to enter Reduce\n");
```

```
            reduce( stable, otable );  /* Reduce the given symbolic state table    */

            format( stable,otable );  /* Format the state table information for ASSIGN */
            fclose(fout);


        $



            /*      LOAD - loads character strings of maximum length MAXLEN     */

        load( p, num )
        char p[MAXSYMBOLS][MAXLEN];
        int num;
        $
        int j, k, strlen();

        if ( debug )
            fprintf(stderr,"LOAD: the number of variables to load is %d\n",num);
        for (j = 0; j < num; j++)
            $
            scanf("%s",p[j] );
            if ( p[j][0] == '*')
                $
                for ( k = 0; k < MAXLEN - 1 ; k++)
                    p[j][k] = p[j][k + 1];
                errorstate = j + 1;
                $
            if ( strlen(p[j]) > MAXLEN - 1 )
                p[j][MAXLEN] = 0;                 /* Terminate the string if too long */
            if ( debug )
                fprintf(stderr,"LOAD: %s\n",p[j]);
            $
        $


        /* LOADSTABLE - loads STABLE with a copy of the input state array.
                Symbolic states names are replaced with integers to denote states. */

        loadstable(stable, states)
        int stable[MAXSYMBOLS][MAXCOMBIN];
        char states[MAXSYMBOLS][MAXLEN];
        $
        int j, k, l, flag, match, *p, MAX = MAXLEN;
        char string[MAXLEN], c;

        if (debug)
            $
            fprintf(stderr,"Numerical Next State Table.\n\n");
            fprintf(stderr,"LOADSTABLE: Input Term #    Term     State #\n");
            $

        for (j = 0; j < nstates; j++)
```

D - 3

```
                    $
        p = stable[j];
        if ( debug )
            fprintf(stderr,"LOADSTABLE:  %d\n",numinp);
        for (k = 0; k < numinp ; k++)
            $
            flag = scanf("%1s",string);   /* Get the first character   */
            l = 0;
            while ( (( c = getchar() ) != ' ')&&( c != '\n') )
                if ( l < MAXLEN )
                                        /* Tests if variable is longer than */
                    string[ ++l ] = c;    /* MAXLEN characters.  Ignores     */
                                        /* additional characters          */
            string[ ++l ] = 0;  /* Terminate the string            */
            if ( debug )            .
                fprintf(stderr,"LOADSTABLE: %d.  %s ", k, string);
            match = FALSE;
            for (l = 0; l < nstates ; l++)  /* Search for a match with the
                                              symbolic state names     */
                $
                if ( debug )
                    fprintf(stderr,"  %s %d ",states[l], l);
                if ( strcmp( string, states[l]) == 0)
                    $
                    if ( match ) error(8); /* Two indentical symbolic names */
                    if ( debug )
                        fprintf(stderr,"  %d \n", ( l + 1));
                    *p++ = l + 1;
                    match = TRUE;
                    break;
                    $
                $
            if ( match == FALSE ) error(6); /* No match with symbolic state names */
            if ( (flag == 0)||(flag == EOF) ) error(5);
            if ( debug )
                fprintf(stderr,"\n");
            $
        $
    $


/* LOADOTABLE - Loads OTABLE with character strings suitable for PRESTO
        Example: f0 is the fourth of five output variables in the input file.
                 It is the only output variable listed for a particular
                 state/input combination (NODES) in the output array.
                 LOADOTABLE generates the character string "00010".
                                                              */



loadotable( otable, outputs )
char otable[MAXSYMBOLS][MAXCOMBIN], outputs[MAXSYMBOLS][MAXLEN];
$
int j, k, l, cnt;
char c, string[MAXLEN], *p;
```

MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

```
        If ( debug )
            fprintf(stderr,"Output table character strings \n\n");
        for (J = 0; J < nstates; J++)
            S
            for ( k = 0; k < numinp * noutputs ; k++)
                otable[J][k] = '0';

            for (k = 0; k < numinp * noutputs ; k += noutputs)
                S
                for (l = 0; l < noutputs; l++) /* Max number of outputs for a given
                                                        state/Input combination    */
                    S
                    cnt = 0;
                    scanf( "%1s", &c);
                    If ( debug )
                        fprintf(stderr,"LDOTABLE:  %c\n",c);

                                    /* Get the first non-white character    */
                    If (c == '/')                       /* Get next character */
                        S
                        scanf( "%1s", &c);
                        S
                      else if ( l != 0)
                        S
                        ungetc(c, stdin);
                        break;   /* Got all of the outputs for this NODE   */
                        S
                    string[0] = c;

                    while ( c = getc(stdin))
                        S
                        If ( c == ' ') break;
                          else if ( c == '/') S ungetc(c,stdin); break; S
                          else if ( c ==  '\n') break;
                          else if ( c ==  EOF) break;
                        If ( debug )
                            fprintf(stderr," LDOT: c %c",c);

                        If ( cnt < MAXLEN - 1 ) string[++cnt] = c;
                        S
                    string[++cnt] = 0;  /* Terminate the string    */
                    If ( debug )
                        fprintf(stderr,"LOADOTABLE: string %s J%d  k%d ",string,J,k);
                                                        /* Print the string. */
                    for ( cnt = 0; cnt < noutputs; cnt++)
                        If (strcmp(string,outputs[cnt]) == 0) otable[J][k + cnt] = '1';
                    S
                If ( debug )
                    fprintf(stderr,"\n");
                S
            otable[J][k + noutputs - 1 ] = 0; /* Terminate the string for this NODE  */
            If ( debug )
                fprintf(stderr,"LOADOTABLE:  %s\n",otable[J]);
            S

        S
```

```c
/* REDUCE - reduces the symbolic state table          */

reduce( stable, otable )
int stable[MAXSYMBOLS][MAXCOMBIN];
char otable[MAXSYMBOLS][MAXCOMBIN];
{
int j, k, l, cnt, cnt1, count, partition[MAXSYMBOLS * 4],
    *pl1, *pl2, *pl3, change = TRUE, pass = 0, done, test = FALSE;
int  state[MAXSYMBOLS], ptable[MAXSYMBOLS][MAXCOMBIN] ;

        /* Partition requires at most nstates * 2.  Nstates * 4 is used
            to allow for repartitioning during the symbolic state reduction. */
        /*       Partition based on outputs              */

if ( debug )
    fprintf(stderr,"REDUCE: \n");
pl1 = partition;
cnt = 0;
for (j = 0; j < nstates ; j++)
    {
    pass = FALSE;
    for ( pl2 = partition;  pl2 < pl1; pl2++ )
        {
        if ( debug )
            fprintf(stderr,"REDUCE: pl2 %d, *pl2 %d\n",pl2-partition,*pl2);
        if ( *pl2 == j + 1 ) pass = TRUE;  .
        }
    if ( pass ) continue;

                /* Skip states already in Partition.
                    J + 1 is used becuase the Stable and Otable reference
                    is always one less than the state number the
                    reference points to.  Applys to k + 1 below also. */
    for (k = j ; k < nstates ; k++)
        if ( strcmp ( otable[j], otable[k]) == 0 )
            {
            *pl1++ = k + 1;
            if ( debug )
              {
              fprintf(stderr,"REUDCE: pl1 %d, *pl1 %d",pl1-partition-1,*(pl1-1));
               fprintf(stderr,", j %d, k %d\n", j, k);
              }
            }
    *pl1++ = EOP;        /* Set End of Partition    */
    if ( debug )
      fprintf(stderr,"REDUCE: pl1 %d, *pl1 %d\n",pl1-partition-1,*(pl1-1));
    cnt++;
    }
*pl1 = EOR;
if ( debug )
    {
    fprintf(stderr,"REDUCE: Initial number of partitions is: %d  \n", cnt);
    prnt( partition );
    }
```

```
if ( cnt == nstates)
    {
    fprintf(stderr,"No State Reduction possible. \n");
    return;
    }

        /* Symbolic State reduction              */


while ( change )
    {
    change = FALSE;
    if ( debug )
        fprintf(stderr,"REDUCE: Symbolic state reduction pass # %d.\n", pass++);
    cnt1 = 0;                  /* Partition Counter     */
    for (pi1 = partition; *pi1 != EOR; pi1++)
        switch(*pi1)
            {    /* Each state is listed in Partition only once.  Generate
                    State table of each symbolic state's partition.   */
            case NULL: break;
            case EOP: cnt1++; break;
            default: state[*pi1 - 1] = cnt1;
            }
    pi3 = pi1;         /* Set pi3 equal to EOR          */
    if ( debug )
        {
        fprintf(stderr,"Symbolic State Parition n");
        for ( j = 0; j < nstates; j++)
            fprintf(stderr,"State: %d  Partition %d\n", j+1, state[j]);
        prnt( partition );
        fprintf(stderr,"\nState Transitions \n");
        }
    for ( j = 0; j < nstates ; j++ )   /* Convert state numbers to partition */
        {
        if ( debug )
            fprintf(stderr,"%d. ",j + 1);

        for ( k = 0; k < numinp; k++)      /* numbers                      */
            {
            ptable[j][k] = state [ stable[j][k] - 1 ];
            if ( debug )
                fprintf(stderr," %d", ptable[j][k]);
            }
        if ( debug )
            fprintf(stderr,"\n");
        }

    if ( debug )
        fprintf(stderr,"REDUCE: partition based on transitions. \n");
    pi1 = partition;         /* Reset pi1 to the beginning of Partition  */
    while (*pi1 != EOR)      /* Repartition states based on transitions   */
        {                    /*  between the last set of partitions       */
        done = FALSE;
        for ( pi2 = pi1 + 1; *pi2 != EOP; pi2++)
            {
```

```c
            if ( debug )
                {
                fprintf(stderr,"The states under consideration are");
                fprintf(stderr," %d and %d\n", *pl1, *pl2);
                for (j = 0; j < numinp; j++)
                    fprintf(stderr," %d",ptable[*pl1 - 1][j]);
                fprintf(stderr,"\n");
                for (j = 0; j < numinp; j++)
                    fprintf(stderr," %d", ptable[*pl2 - 1][j]);
                fprintf(stderr,"\n");
                if ( !(done) )
                    prnt( partition );
                }
            for (j = 0; j < numinp; j++)
                if( ptable[*pl1 - 1][j] != ptable[*pl2 - 1][j] )
                    {                          /* State transitions not equal   */
                    change = TRUE;            /* So move to new partition      */
                    done = TRUE;
                    *pl3++ = *pl2;             /* Move state to end of list   */
                    *pl2 = NULL;          /* Erase state from present partition */
                    break;                /* Jump out of comparison for loop   */
                    }
                }

        if ( done )
            {
            *pl3 = EOP;
            *++pl3 = EOR;                    /* Set new EOR   */
            }
        if ( debug )
            prnt( partition );
        pl1 = ++pl2;            /* Set pl1 equal to start of next partition   */
        }

    pl1 = pl2 = partition;  /* Remove NULLs from Partition list      */
    while ( *pl2++ != EOR )
        if ( *pl2 != NULL )
            *++pl1 = *pl2;
    if ( debug )
        {
        fprintf(stderr,"Remove the NULL states.....\n");
        prnt( partition );
        }


    }

        /* Final partitioning is in Partition[]            */
        /*  Take one state from each partition             */

cnt1 = 0;
if ( debug )
    fprintf(stderr,"The final partitions are listed below:\n");
pl1 = state;
*pl1 = partition[0];
```

D - 8

```
        for ( pl2 = partition; *pl2 != EOR; pl2++ )
            $
            if ( debug && ( *pl2 > 0 ))
                fprintf(stderr,"%d.   %d n", cnt1, *pl2);
            if ( *pl2 == errorstate) *pl1 = *pl2;
            if ( *pl2 == EOP)
                $
                cnt1++;
                *++pl1 = *( pl2 + 1 );   /* Make state a list of the reduced
                                             states without EOR and EOP   */
                $
            $
                /* Update STABLE    */
*++pl1 = EOR;
if ( debug )
    prnt(state);


pl1 = state;
for ( pl2 = partition; *pl2 != EOR; pl2++)
        $                       /* Change states in STABLE to reduced table set */
    if ( (*pl2 > 0)&&(*pl2 != *pl1) )
        for ( j = 0; j < nstates; j++)
            for ( k = 0; k < numinp; k++)
                if ( stable[j][k] == *pl2 )
                    stable[j][k] = *pl1;
    if (*pl2 == *pl1 ) test = TRUE;    /* Test if STATE (*pl1) equaled    */
    if (*pl2 == EOP )                  /* value within the partition    */
        if (test)
            $
            pl1++;
            test = FALSE;
            $
          else error(10);
    $


if ( debug )
    $
    fprintf(stderr,"\nThe original number of states was %d.  ", nstates);
    if ( nstates == cnt1)
        fprintf(stderr," No reduction possible\n");
      else
        $
        fprintf(stderr,"The reduced number of states is %d.\n", cnt1);
        fprintf(stderr,"The states in the reduced set are listed below:\n");
        for (j = 0; j < cnt1; j++)
            fprintf(stderr,"%d \n", state[j]);
        $
    $
if ( nstates == cnt1 ) return;    /* Do NOT rearrange global variables
                                     before returning          */

for (j = 0; j < cnt1; j++)          /* Sort Partition from low to high order  */
    for ( k = j + 1; k < cnt1; k++)  /* Insure next state rows are not changed
                                     before required in the next FOR loop */
```

```
               If (state[j] > state [k])
                   $
                   i = state[j];                 /* SWAP   */
                   state[j] = state[k];
                   state[k] = i;
                   $
       If ( debug )
           $
           fprintf(stderr,"REDUCE: Change stable and otable to reflect the new");
           fprintf(stderr," partitions.   \n");
           $

       for (j = 0; j < cnt1; j++)       /* Change next state & output tables to  */
           $                            /* reflect reduced number of states      */
           for ( k = 0; k < numinp; k++)
               stable[j][k] = stable[ state[j] - 1 ][k];
           for ( k = 0; k < (numinp * noutputs + 1); k++)
               otable[j][k] = otable[ state[j] - 1 ][k];
           $
       nstates = cnt1;    /* Change global variable to reflect new number of states */

           /* Make sure states are numerically sequential.  For example:
              1 2 3 4 instead of 1 2 5 6.  Nonsequential stae numbers are
              highly likely when any reduction is done.       */
       for ( j = 0; j < nstates; j++)
           If ( state[j] != j + 1 )
               for ( k = 0; k < nstates; k ++)
                   for ( i = 0; i < numinp; i++)
                       If (stable[k][i] == state[j])
                           stable[k][i] = j + 1;

       $



       /* PRNT() - print a single dimension array until value EOR or -2 is reached */

       prnt(p)
       Int *p;
       $
       Int *p1;
       p1 = p;
       fprintf(stderr,"Partitions: \n");


           for ( p1 = p; *p1 != EOR; p1++)
               fprintf(stderr," %d", *p1);
           fprintf(stderr,"\n");
       $



       /* FORMAT - prints an text file of global variables Nstates, Ninputs,
                   Noutputs, Symbol, Lambda, and the two arrays: Stable and Otable
                                                                   */

       format(stable,otable)
```

```
                int stable[MAXSYMBOLS][MAXCOMBIN];
                char otable[MAXSYMBOLS][MAXCOMBIN];
                $
                int j, k, i;

                if ( debug )
                    fprintf(stderr,"Hello from FORMAT n");
                fprintf(fout,"%d %d %d %d %1.1f",nstates,ninputs,noutputs,symbol,lambda );
                fprintf(fout," %d\n\n",errorstate);
                for ( j = 0; j < nstates; j++)
                    $
                    for ( k = 0; k < numinp; k++)
                        fprintf(fout,"%d ",stable[j][k]);
                    fprintf(fout,"\n");
                    $
                fprintf(fout,"\n\n");
                if ( ( numinp * noutputs ) > NUMCOL )
                    $
                    fprintf(stderr,"Warning - the output array will print funny since ");
                    fprintf(stderr,"numinp * noutputs is longer than %d.", NUMCOL);
                    $
                for ( j = 0; j < nstates; j++)
                    fprintf(fout, "%s\n",otable[j]);
                $




                /* ERROR - Prints error messages to stderr     */

                error(n)
                int n;
                $
                char *p1 = "ERROR - Usage: symreduc [-d] outfile.ext < infile ",
                *p2 = "ERROR - Illegal symbolic state table parameter in first line of file.",
                *p3 = "ERROR - One of the symbolic state table parameters is zero.",
                *p4 = "ERROR - Variable name in the state or output arrays did not match",
                *p4p1 = " any of the given symbolic names.",
                *p5 = "ERROR - Incorrect number of state or output table variables. ",
                *p6 = "ERROR - Invalid string in the state table. ",
                *p7 = "ERROR - Partition list error.",
                *p8 = "ERROR - At least two symbolic state names are identical!!    ",
                *p9 = "ERROR - Can't open the output file:     ",
                *p10 = "ERROR - State transitions changed to state NOT in Partition";

                switch(n)
                    $
                    case 1: fprintf(stderr,"%s\n\n", p1); break;
                    case 2: fprintf(stderr,"%s\n\n", p2); break;
                    case 3: fprintf(stderr,"%s\n\n", p3); break;
                    case 4: fprintf(stderr,"%s%s\n\n", p4, p4p1); break;
                    case 5: fprintf(stderr,"%s\n\n", p5); break;
                    case 6: fprintf(stderr,"%s\n\n", p6); break;
                    case 7: fprintf(stderr,"%s\n\n", p7); break;
                    case 8: fprintf(stderr,"%s\n\n", p8); break;
```

```
        case 9: fprintf(stderr,"%s\n\n", p9); break;
        case 10: fprintf(stderr, "%s\n\n", p10); break;
        default: fprintf(stderr,"ERRONEOUS call to ERROR!!!\n\n");
        };
    exit(1);
    }
```

```
/*****************************************************************/

/* ASSIGN.C  -  Fixed arrays                                  */
/* ASSIGN - Makes an optimum state assignment for the state table
         in the input file.  The output file is suitable for PRESTO.
         Options include simple assignment and grey code schemes.
         Debug is also an option.                            */

#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define MAXCOL  100  /* Maximum number of distinct state assignment columns */
#define MAXNUMBER  50   /* Maximum number of states, inputs, or outputs   */
#define MAXCOMBIN  500  /* Maximum for numinp * noutputs             */
#define MAXLEN 25 + 1   /* Maximum length for symbol names  plus 1 for
                           string termination           */
#define EOR -2          /* End of Record      */
#define EOP -1          /* End of Partition   */
#define NUMCOL 80       /* Number of columns on standard CRT   */
                  /*   Used for visual output only - does not affect files */


/*****************************************************************/

/*      GLOBAL VARIABLES                                  */

static int mass_debug = FALSE, debug = FALSE,
          simple = FALSE, grey = FALSE;
static int nstates, ninputs, noutputs, symbol, errorstate, numinp ;
static int dsanum, maxstate, num2n, num2n_1, num2R_1, num2R;
FILE *fout, *fopen();
float lambda;



/*****************************************************************/

/*      MAIN PROGRAM              */

main (argc, argv)
int argc;
char *argv[];
{
int flag;
char otable [MAXNUMBER][MAXCOMBIN], svar[ MAXCOL ][ MAXNUMBER ]  ;
int strans[MAXNUMBER][MAXCOMBIN];
register int j, k;

fout = fopen (argv[argc-1], "w");
if (fout == NULL) error(2);
if (argc > 4)error (1);
```

```
for ( j = 1; j < (argc-1); j++)
    if ( argv[j][0] == '-' )
        switch(argv[j][1] )
            {
            case 'd': debug = TRUE;
                        break;
            case 's': if ( argv[j][2] == 'a' ) simple = TRUE;
                        else error(1);
                        break;
            case 'g': if ( argv[j][2] == 'c' ) grey = TRUE;
                        else error(1);
                        break;
            case 'm': mass_debug = TRUE;
                        break;
            default: error(1);
            }


if ( debug )
    fprintf(stderr," Hello!  You are now in the PLAFST DEBUG ZONE!\n");

flag = scanf("%d %d %d %d %f",&nstates, &ninputs, &noutputs, &symbol, &lambda);
if (flag > 0 ) flag = scanf("%d", &errorstate);
if((flag == NULL)||(flag == EOF )) error(3);

numinp = power(ninputs);          /* Numinp  equals 2 raised to the nstates */

if (debug)                /* Print symbolic state table values from above */
    {
    fprintf(stderr,"The symbolic state table values were read in ");
    fprintf(stderr,"correctly. Their values are : \n");
    fprintf(stderr,"nstates %d,   ninputs %d n",nstates, ninputs);
    fprintf(stderr,"noutputs %d,   CIF symbol %d n",noutputs, symbol);
    fprintf(stderr,"lambda %1.1f   numinp  %d\n",lambda, numinp);
    fprintf(stderr,"errorstate  %d\n\n", errorstate);
    }



/* Load the character arrays from the input file.              */

if ( debug )
    fprintf(stderr,"The state table values are now being loaded.\n\n");
loadint ( strans, nstates, numinp );
loadchar ( otable, nstates );
if ( debug )
    format(strans,otable);

setnum();   /* Set the values of global variables dsanum, maxstate, num2n
                and num2n_1         */
if ( errorstate != NULL )
    {
    expand(strans  >table )
                        * I_ .e undefined states transition to the errorstate */
    setnum();   /* R :et the values of global variables dsanum,
                    maxstate, num2n, and num2n_1         */
```

```
            $
    if (simple) code_simple( strans, svar);
      else if ( grey ) code_grey( strans, svar);
      else optimum(strans, svar );

    merge ( strans, otable, svar );
    fclose(fout);
    $




    /*****************************************************************/

    /* EXPAND - makes all undefined states transition to the errorstate
                and designates the output as all zeroes   */

    expand(strans, otable )
    int strans[MAXNUMBER][MAXCOMBIN];
    char otable[MAXNUMBER][MAXCOMBIN];
    $
    int j, k;

    for ( j = nstates; j < num2n; j++)
        $
        for (k = 0; k < numinp; k++)
            strans[j][k] = errorstate;
        for ( k = 0; k < (numinp * noutputs); k++)
            otable[j][k] = '0';
        otable[j][k] = 0;    /* Terminate the string    */
        $
    nstates = num2n;           /* All states are now defined - therefore the number
                                  of states is equal to the maximum number.   */
    $




    /*****************************************************************/

    /* OPTIMUM - assigns an optimum state variable code to the state table. */

    optimum( strans , svar )
    int strans[MAXNUMBER][MAXCOMBIN];
    char    svar[MAXCOL][MAXNUMBER];

    $
    char c, yvar[MAXCOL][MAXNUMBER], YVAR[MAXCOL][MAXNUMBER];
    int j, k, l, n;
    int cost[MAXCOL], est, EST;

    if ( (num2n_1 + 1) == nstates )     /*            m              */
                                    /* Nstates equals 2 + 1 - so list y variables   */
        for ( j = 0; j < num2R_1; j++)
            decbin( yvar[j], (j + 1), nstates );
```

E - 3

```c
        else              /* Nstates is even so generate the y variables  */
           {
        for ( j = 0; j < nstates; j++)
            yvar[0][j] = '-';
        yvar[0][k] = 0;               /* Initialize the first string     */

        n = nstates - num2n_1;
        j = 0;
        while ( n < nstates / 2 )
            {
            j = gen0( yvar, n, 0, 0 );
            yvar[j][0] = '-';   /* Change the first character in the "seed"
                                   for GEN1              */

            j = gen1( yvar, n, j, 0 );
            n++;
            }
        if ( nstates % 2 )   /* Nstates is odd so call both gen0 and gen1 */
            {
            j = gen0( yvar, n, j, 0);
            yvar[j][0] = '-';
            j = gen1( yvar, n ,j, 0);
            }
          else j = gen0( yvar, n, j, 0);

        for ( k = 0; k < dsanum; k++)    /* Fill the '-'s with a '0' or '1' */
            {
            if ( yvar[k][0] == '0' ) c = '1';
              else c = '0';
            for ( i = 0; i < nstates; i++)
                if ( yvar[k][i] == '-' ) yvar[k][i] = c ;
            }
           }

    if ( debug )
        fprintf(stderr,"Distinct State Assignment Columns\n\n");
    for ( k = 0; k < dsanum; k++)
        {
        for ( i = 0; i < nstates; i++)
            if ( yvar[k][i] == '1') YVAR[k][i] = '0';
              else YVAR[k][i] = '1';
        YVAR[k][i] = 0;    /* Terminate the string.  YVAR is the complement
                              of yvar - required by D flip flop implementation  */

        if ( debug )
            fprintf(stderr,"%s    %s\n\n", yvar[k], YVAR[ k ] );
        }

    if ( j != dsanum )
        {
        fprintf(stderr,"j %d,  dsanum %d\n", j, dsanum);
        error(7);
        }
    if ( debug )
        {
```

```c
            fprintf(stderr,"State Transition Table\n");
            form(strans);
            $

      for ( J = 0;  J < dsanum;  j++)
            $
          est = cost_est( strans, yvar[j]);
          EST = cost_est( strans, YVAR[j]);
          if ( est < EST ) cost[j] = est;
            else cost[j] = EST;
          if ( debug )
              $
              fprintf(stderr,"OPT: J %d,est %d, EST %d ", J, est, EST);
              fprintf(stderr,"cost %d    yvar %s,  ",cost[j],yvar[j]);
              fprintf(stderr,"YVAR  %s\n",YVAR[ J ] );
              $
          $
      sort ( cost, yvar, YVAR );

      if ( debug )
          for ( J = 0;  J < dsanum;  j++ )
              fprintf(stderr,"cost %d, yvar %s, YVAR %s\n",cost[j],yvar[j],YVAR[j]);
      opt_assign ( strans, cost, yvar, YVAR, svar );

      $


/*********************************************************************************/


/* COST_EST - copies the state transition table into a truth table using
              the given yvar of YVAR string.  Splits the table into halves
              based on whether the yvar string is a '0' or '1'.  It
              divides the undefined states so that there are an equal
              number of '0' and '1' rows.
              This is done decrease the cost obtained when
              the table is evaluated by QMcClus.  COST_EST is used only
              to obtain the minimum cost estimates.          */

cost_est( strans, yvar)
int strans[MAXNUMBER][MAXCOMBIN];
char *yvar;
$
char s[MAXNUMBER][MAXCOMBIN], c, yvarp[MAXCOL];
int J, k, i, m, half, estimate;

strcpy( yvarp, yvar );

truth_table( s, strans, yvarp );   /* Make truth table for yvarp */

if(debug)
    $
    fprintf(stderr,"COST_EST: TRUTH TABLE for %s\n", yvarp);
    formc( s, nstates);
    $
```

E - 5

```
            /* Separate '0' and '1' rows */
            /* Test yvarp starting at yvarp[half] for values equal to yvarp[0].
               Stop when yvarp[half] is equal to the first yvarp value - This
               row must be moved to the other ('0' or '1') half of the table */

        half = num2n_1 ;
        while( (yvarp[half] != *yvarp) && ( half < nstates )) half++;
        for( j = 1; j < num2n_1 ; j++)
            if( *(yvarp + j) != *yvarp )
                {
                if ( mass_debug )
                    fprintf(stderr,"half before %d\n",half);

                if ( half < nstates )
                    /* SWAP       */
                    for( k = 0; k <= numinp; k++)
                        {
                        c = s[j][k];
                        s[j][k] = s[half][k];
                        s[half][k] = c;

                        c = yvarp[ half ];
                        yvarp[ half ] = yvarp[ j ];
                        yvarp[ j ] = c;

                        }
                else        /*   MOVE */
                    {
                    for ( k = 0; k < numinp; k++)
                        s[half][k] = s[j][k];
                    yvarp[half] = yvarp[j];
                    if ( mass_debug )
                        {
                        fprintf(stderr,"yvarp.half %c , ", yvarp[half]);
                        fprintf( stderr," yvarp.j %c\n", yvarp[j]);
                        }
                    yvarp[j] = '*';
                    half++ ;
                    }

                if ( half < nstates )
                    while( (yvarp[half] != *yvarp) && (half < nstates) ) half++;
                else
                    yvarp[half++] = '*';
                if ( mass_debug )
                    fprintf(stderr,"half after %d\n",half);
                }

        while ( half < num2n )
            if ( yvarp[half++] < '0' )
                yvarp[ half - 1 ] = '*';
        yvarp[ half ] = 0;
        if ( debug )
```

```
        fprintf(stderr,"Rearranged column variable: %s\n", yvarp);

    for( j = 0; j < num2n; j++)   /* Fill in undefined states   */
        if ( ( yvarp[j] != '0' ) && ( yvarp[j] != '1' ) )
            {
            for ( k = 0; k < numinp; k++)
                s[j][k] = '1';
            s[j][k] = 0;        /* Terminate the string  */
            }

    if(debug)
        {
        fprintf(stderr,"COST_EST: Rearranged truth table for cost evalutation \n");
        formc( s, num2n );
        }

    estimate = TRUE ;
    return QMcClus( s, num2n, yvarp, estimate);
    }




/************************************************************/


/* TRUTH_TABLE - generates a character array truth table from the
                 state transistions ( strans ) and the column variable
                 ( cvar ).
                                                            */
truth_table( sctrans, strans, cvar )
char sctrans[MAXNUMBER][MAXCOMBIN], *cvar;
int strans[MAXNUMBER][MAXCOMBIN];
{
int j, k;

for( j = 0; j < nstates; j++)   /* Make first cut at Truth Table  */
    {
    for ( k = 0; k < numinp; k++)
        sctrans[j][k] = *( cvar + strans[j][k] - 1 );
    sctrans[j][k] = 0;
    }
}




/************************************************************/


/* FORMC - prints an array of num strings to stderr      */

formc( array, num )
char array[MAXCOL][MAXCOMBIN];
int num;
{
int j;
```

```c
        for ( J = 0; j < num; j++)
            fprintf(stderr,"%s \n", array[j]);

    }




/*******************************************************************/

/* QMcClus - Quine McClusky state evaluation            */

QMcClus( s, numrow, svar, estimate )
char s[MAXNUMBER][MAXCOMBIN], *svar;
int numrow, estimate;
{
int J, k, l, m, hold, lastpi, lastl, stackl, lastadj, numbits;
char prime_imp[MAXCOMBIN][MAXNUMBER],
     impl [MAXCOMBIN][MAXNUMBER];
int numones[MAXCOMBIN], differ, pass, change = TRUE, nl, nt, slt, cost;

hold = 0;                    /* Used to count the number of implicants in the
                                zero half of the table.           */
lastpi =  0;
numbits = maxstate + ninputs;
m = 0 ;

for (l = j = 0; j < numrow; j++)               /* Get initial implicants   */
   {
   for (k = 0; k < numinp; k++)
      if ( s[j][k] == '1' )
          {
          numones[ m++ ] = j ;
          if ( j < num2n_1 ) hold++;  /* The implicant is in the zero half */
          decbin( prime_imp[ lastpi++ ], l + k, numbits);
          if ( ( *(svar + j) == '1' )||( *(svar + j) == '0' ) )
              prime_imp [lastpi - 1][numbits + 1] = 1;
            else                          /* Mark the required implicants  */
              prime_imp [lastpi - 1][numbits + 1] = 0;
          }
   l += k;
   }


lastadj = lastpi;

if ( estimate )
    {
                        /*  Check for 2k rows. Gets the the implicants with
                            a 0 most significant bit first.  Only gets one
                            adjanceny per implicant from the implicants
                            with a 1 ms bit. This satisfies the criteria
                            that groupings in the transition table have the
                            same number of implicants from each half of the
```

E - 8

```c
                        table.
                        Hold equals the number of 0 half prime
                        implicants.
                                                            */
        last1 = 0;
        for ( j = 0; j < hold ; j++ )
            {
            if ( svar[ numones[ j ] ] == '*' )    /* Skip don't care rows */
                continue ;
            m = 0;
            for ( k = maxstate - 1 ; k > 0; k-- )
                if (prime_impl[j][k] != prime_impl[j][maxstate - 1] )
                    m++;                /* Checks for number of changes between
                                             1 and 0       */
            if ( (prime_impl[j][maxstate - 1] == '0')&&(m == 0) ||
                (prime_impl[j][maxstate - 1] == '1')&&(m <= 1) )
                {
                if(debug)
                    fprintf(stderr,"QM: 0 half %s  m %d\n", prime_impl[j], m );
                strcpy( impl[ last1++ ], prime_impl[ j ] );
                }
            }


                        /* Look for 2k rows in the 1 half of the table  */
    if ( mass_debug )
        for ( i = hold ; i < lastadj ; i++ )
            {
            fprintf( stderr,"QM prime implicants: %s ", prime_impl[ i ] ) ;
            fprintf( stderr," row %d, yvar %c\n", numones[i], svar[numones[i]]);
            }
        for ( i = 0; i < last1 ; i++ )
            for ( pass = 0; pass < 2 ; pass++ )  /* Look for prime implicants
                                                    first, then any adjacenies */
                {
                if ( pass == 1 )
                    j = hold ;
                for ( ; j < lastpi ; j++ )

                    {
                    switch ( pass )
                        {
                        case 0:  if ( svar[ numones[ j ] ] == '*' )
                                    continue;
                                        /* Skip the don't cares on the first pass */
                        case 1:  if ( svar[ numones[ j ] ] != '*' )
                                    continue ;
                                        /* Skip prime implicants on the second
                                            pass       */
                        }
                    m = 0;
                    for ( k = numbits - 1 ; k > 0; k-- )
                        if ( k > maxstate - 1 )
                            {
                            if ( prime_impl[ j ][ k ] != impl[ i ][ k ] )
                                break;
```

E - 9

```c
                              $
                          else
                              if (prime_impl[j][k] != prime_impl[j][maxstate - 1] )
                                      m++;            /* Checks for number of changes
                                                        between 1 and 0      */
                      if ( k > maxstate - 1 )
                          continue;       /* Implicants were in different Truth
                                                        Table columns            */
                      if ( mass_debug )
                          fprintf(stderr,"Adj. consider %s n", prime_impl[ j ] ) ;
                      if ( (prime_impl[j][maxstate - 1] == '0')&&(m == 0) ||
                          (prime_impl[j][maxstate - 1] == '1')&&(m <= 1) )
                          $
                          strcpy( prime_impl[ lastadj ], prime_impl[ j ] );
                          prime_imp [ lastadj++ ][ 0 ] = '-';
                                                          /* Mark Adjacency */
                          if ( debug )
                              $
                              fprintf(stderr,"1 half %s", prime_impl[ j ] ) ;
                              fprintf(stderr," m %d", m ) ;
                              fprintf(stderr,",  0 half %s\n", impl[ i ] );
                              $
                          break;    /* Only look for one match    */
                          $
                      $
                  if ( j < lastpi )   /* Above for loop found a match and we
                                        only need to look for one match so.. */
                      $
                      j++ ;
                      break ;
                      $
                  $
              $

      /*  Sort prime_imp into impl by the number of 1s in each term      */

  for ( j = 0; j < lastadj; j++ )
      $
      gray_code( prime_impl[ j ] );          /* Convert to Gray code      */
      for ( i = k = 0 ; k < numbits; k++ )
          if ( (prime_impl[j][k] == '1') || (prime_impl[j][k] == '-') ) i++;
      numones[j] = i;
      $
  for ( i = hold = j = 0 ; j < numbits + 1; j++)
      $
      for ( k = 0; k < lastadj; k++)
          if ( numones[ k ] == hold )   /* Move to impl */
              $
              strcpy( impl[ i ], prime_impl[ k ] );
              impl[ i ][ numbits + 1 ] = 0;          /* Set check flag */
              i++;
              $
      hold++;
      if ( i != 0 )
          $
```

```c
            impl[ i ][0] = 0;          /* Mark divisions between terms with
                                          different number of ones          */
            i++;
            }
        }
    last1 = i - 1;    /* Set pointer for impl      */
    pass = 1;
    hold = 0;


    if ( debug )
        for ( j = 0; j < last1; j++)
            fprintf(stderr,"QMcClus: %s\n", impl[ j ] );


    while ( change )          /* Quine-McClusky algorithm      */
        {
        change = FALSE;
        stack1 = MAXCOMBIN;
        for ( j = hold ; j < last1 - 1; j++)
            {
            switch ( impl[ j ][0] )
                {
                case '-': if ( pass == 1 )
                              {
                              if( mass_debug )
                                  fprintf(stderr,"Est Adjacency: %s\n",impl[j]);
                              impl[ j ][ numbits + 1 ] = 1;  /* Set check flag */
                              if ( stkchk( stack1, impl, j ) )
                                  strcpy( impl[--stack1], impl[ j ] );
                              continue;
                              }
                          break;
                case 0:   impl[--stack1 ][0] = 0;  /* Set division marker */
                          continue;
                }
            k = 0;
            i = j + 1;
            while ( ( k < 2 )&&( i < last1 ) )
                {                         /* Only check for adjancencies in this
                                             division and the next           */
                switch( impl[ i ][ 0 ] )
                    {
                    case 0: k++ ;         /* Check for division marker    */
                            i++ ;
                            continue ;
                    case '-': if ( pass == 1 )   /* Check for estimated adjacency */
                                  {
                                  i++ ;
                                  continue ;
                                  }
                    }

                differ = numbits ;

                for ( m = 0; m < numbits; m++ )
```

```
                        If ( impl[J][m] == impl[i][m] ) differ--;
                         else
                                hold = m;
                    If ( differ == 1 )
                                    /* Implicants differed at only one postion
                                        so they are adjacent.  Stkchk checks
                                        the stack for duplicate entries   */
                    $
                    If( mass_debug )
                        fprintf(stderr,"Adjacency: %s  %s\n",impl[J],impl[i]);
                    If (stack1 < J ) error (10);   /* Stack overflow    */
                    impl[ J ][ numbits + 1 ] = 1;   /* Set check flag   */
                    impl[ i ][ numbits + 1 ] = 1;   /* Set check flag   */
                    change = TRUE;          ./* Let the outer while loop
                                             know that we found an adjacency. */
                    strcpy( impl[--stack1 ], impl[J] );
                    impl[ stack1 ][ hold ] = '-'; /* Mark the adjacency    */
                    If ( ! stkchk( stack1 + 1, impl, stack1 ) )
                        stack1++ ;   /* Check if already on the stack  */
                    $
                i++;                    /* Point to the next implicant */
                $
            $
            /*  Compact impl  */
    i = 0;
    for ( J = 0; J < last1; J++)
        If ( (impl[J][numbits + 1] == 0 )&& ( impl[J][0] != 0 ))
            $
            If( mass_debug )
                fprintf(stderr,"QM compact: %s \n", impl[J]);
            strcpy( impl[ i], impl[ J ] );
            impl[ i++ ][ numbits + 1 ] = 0;     /* Zero the check flag */
            $
    hold = i;                          /* Save the start of adjacent implicants
                                         from the last search       */
    for ( J = MAXCOMBIN - 1; J >= stack1; J-- )
        $
        strcpy ( impl[ i ], impl[ J ] );
        impl[ i++ ][ numbits + 1 ] = 0;     /* Make sure that the flag */
        $                                    /*  is reset.              */

    last1 = i ;                                                   .

    If ( mass_debug )
        for ( J = 0; J < last1; J++ )
            fprintf(stderr,"QMcClus: impl  %s n", impl[J] );
    If ( debug )
        fprintf(stderr,"QMcClus pass # %d\n", pass );
    pass++ ;
    $


        /*  Done with the Quine-McClusky - so find minimum cover   */

    nl = nt = slt = 0;       /* Number of Literals, Number of terms, and
```

```c
                              · Single literal terms                    */
        stack1 = MAXCOMBIN;
        for ( j = 0; j < lastpi; j++)
            if ( prime_imp [ j ][ numbits + 1] == 1)
                {
                i = 0;
                for ( k = stack1 ; k < MAXCOMBIN ; k++ )
                    {
                    if ( mass_debug )
                        fprintf(stderr,"STACK CHECK!! %s \n",impi[k] );
                    for ( i = 0; i < numbits; i++)
                        if ((impi[k][i] != '-')&&(impi[k][ i ] != prime_imp[ j ][i]))
                            break;
                    if ( i == numbits )             /* Match!   */
                        {
                        if ( debug )
                          fprintf(stderr," Min. Cover: %s  %s \n",prime_imp[j],impi[k]);
                        break;
                        }
                    }
                if ( i == numbits )
                    continue;            /* Found a match so skip the next search */

                for ( k = lasti - 1; k >= 0 ; k--)
                    {            /*  Search the least restrictive implicants first */

                    if ( mass_debug )
                        fprintf(stderr,"CHECK!! %s \n",impi[k] );
                    if ( impi[k][0] == 0 ) continue;
                    for ( i = 0; i < numbits; i++)
                        if ((impi[k][i] != '-')&&(impi[k][ i ] != prime_imp[ j ][i]))
                            break;
                    if ( i == numbits )            /* Match!   */
                        {
                        strcpy ( impi[ --stack1 ], impi[ k ] );
                        if ( debug )
                          fprintf(stderr," Min. Cover: %s  %s\n",prime_imp[j],impi[k]);
                        break;
                        }

                    }
                }


        /* Determine the cost of state assignment          */

    j = 0;
    while ( stack1 < MAXCOMBIN )
        {
        if ( debug )
            fprintf( stderr," Stack: %s\n", impi[ stack1 ] );
        strcpy( impi[ j++ ], impi[ stack1++ ] );
        }
    lasti = j ;
```

```
            for ( j = 0; j < last1; j++)
                    {
                    for ( i = k = 0; k < numbits; k++)
                        if ( imp1[ j ][k] != '-' ) i++;
                    nt++;
                    if ( i == 1 ) sit++;
                    ni += i;
                    }
        if ( debug )
            fprintf(stderr,"QMcClus: ni %d,  nt %d,  sit %d\n",ni,nt,sit);
        if ( nt == 1 )
            if ( sit == 0 )
                cost = ni;
            else
                cost = 0;
        else
            cost = ni + nt - sit;
        return cost;



        }


        /************************************************************/


        /*      STKCHK - Checks the given stack for duplicate entries  */

        stkchk ( stack, list, p )
        int stack, p;
        char list[ MAXCOMBIN ][ MAXNUMBER ];

        {
        int j;

        for ( j = stack; j < MAXCOMBIN; j++ )
            if ( strcmp( list[ j ], list[ p ] ) == 0 )
                return FALSE;
        return TRUE;


        }




        /************************************************************/


        /* GRAY_CODE - Generates a gray code form the binary number pointed
                    to by PS.  The binary number is an one dimensional
                    array of the charcters 1 and 0.  The string is
                    nul terminated.  This algorithm was taken from
                    Digital Logic by Chirlian
                                                        */


        gray_code ( ps )
        char *ps;
        {
```

```c
        char c, *p, cstore;
        int save ;

        save = 0 ;
        c = '0';         /* Seed for the exclusive or loop   */
        p = ps;          /* Don't change the orginal pointer  */
        if ( *p == '-' )         /* Save the adjacency mark        */
            {
            save = 1 ;
            *p = '1' ;
            }

        while( (p - ps) < maxstate )
            {
                                    /* Exclusive or loop             */
            cstore = *p;        /* Save the current string value  */
            if (( ( c == '0' )&&( *p == '1' ) )||( ( c == '1' )&&( *p == '0' ) ))
                *p = '1';       /* Exclusive or is true    */
              else
                *p = '0';       /* Exor is false    */
            c = cstore ;        /* Use the present value of *p to exclusive or
                        with the next value             */
            p++;
            }
        if ( save )
            *ps = '-' ;
        }




/*****************************************************************************/

/*  SORT - sorts the arrays Cost, yvar, and YVAR using Cost as a key.  */

sort (cost, yvar, YVAR )
int cost[MAXCOL];
char yvar[MAXCOL][MAXNUMBER], YVAR[MAXCOL][MAXNUMBER];
{
int  j, k, i ;
char hold[MAXNUMBER];

for ( j = 0; j < dsanum; j++ )
    for ( k = j + 1; k < dsanum; k++)
        if ( cost[ k ] < cost[ j ] )
            {   /* SWAP  -  Everytime something is swapped, one of the
                        things is swapped to its correct location.    */
            if( mass_debug )
                fprintf(stderr,"SORT: cost %d, yvar %s\n",cost[k],yvar[k]);
            i = cost[ j ];
            cost[ j ] = cost[ k ];
            cost[ k ] = i;
```

```c
            strcpy( hold, yvar[ j ] );
            strcpy( yvar[ j ], yvar[ k ] );
            strcpy( yvar[ k ], hold);

            strcpy( hold, YVAR[ j ] );
            strcpy( YVAR[ j ], YVAR[ k ] );
            strcpy( YVAR[ k ], hold );
            $

    $




/*******************************************************************/

/* OPT_ASSIGN - performs the optimum state assignment.  It uses the cost
               estimates just completed as a guide as to when the is
               achieved.                               */

opt_assign( strans, cost, yvar, YVAR, svar )
char yvar[MAXCOL][MAXNUMBER], YVAR[MAXCOL][MAXNUMBER],
        svar[MAXCOL][MAXNUMBER];
int cost[MAXCOL], strans[MAXNUMBER][MAXCOMBIN];
$
int j, k, l, m, stnum[MAXCOL], done, san, an, mns;
int actcost[ MAXCOL ], savenum[ MAXCOL ], pass;

done = FALSE;
pass = 0 ;

for ( j = 0; j < maxstate; j++)
    stnum[j] = j;       /* initialize the state column checker */

for ( j = 0; j < dsanum; j++ )
    actcost[ j ] = -1;           /* initialize the actual cost array */

while ( ! valid( stnum, yvar)  )
    nextasn( stnum, maxstate - 1, dsanum );
for ( mns = an = j = 0; j < maxstate; j++)
    $
    an += best_cost( strans, yvar, YVAR, svar, stnum[ j ], actcost);
    mns += cost[ stnum[j] ];
    if ( debug )
        fprintf(stderr,"OPT: mns %d, an %d, svar %s\n",mns,an,yvar[stnum[j]]);
    $
if ( debug )
    for( j = 0; j < maxstate; j++ )
        fprintf(stderr,"OPT: %s \n", svar[ stnum[ j ] ] );
san = an;
for ( j = 0; j < maxstate ; j++ ) /* Save the previous   */
   savenum[ j ] = stnum[ j ] ;      /* y variable set      */

if ( san > mns )
    while ( done == FALSE )
        $
        pass++ ;
```

```c
            if ( debug )
                fprintf(stderr,"san %d, an %d, mns %d\n", san , an ,mns );
            if ( ! nextasn( stnum, maxstate - 1, dsanum ) ) break;
            while ( !valid( stnum, yvar ) )
                if ( ! nextasn( stnum, maxstate - 1, dsanum ) ) break ;
            for ( mns = j = 0; j < maxstate; j++)
                {
                mns += cost[ stnum[j] ];
                if ( debug )
                    fprintf(stderr,"OPT: %s %d\n",svar[stnum[j]],stnum[j] );
                }
            if ( san <= mns )
                {
                done = TRUE;
                if ( debug )
                    fprintf( stderr,"OPT: DONE!!!! san %d mns %d\n",san,mns);
                continue;
                }
            for ( an = j = 0; j < maxstate; j++ )
                {
                an += best_cost( strans,yvar,YVAR,svar,stnum[ j ],actcost);
                if ( debug )
                    fprintf( stderr,"OPT:  %s\n", svar[stnum[j]]);
                }
            if ( debug )
                fprintf(stderr,"OPT: mns %d,  an %d,  san %d\n", mns, an, san);
            if ( san > an )
                {
                san = an;
                for ( j = 0; j < maxstate ; j++ )   /* Save the previous   */
                    savenum[ j ] = stnum[ j ] ;     /*  y variable set       */
                }
            if ( debug )
                fprintf(stderr,"san %d, an %d, mns %d\n", san , an ,mns );
            if ( san == mns )
                {
                done = TRUE;
                if ( debug )
                    fprintf(stderr,"DONE!! san %d \n", san );
                }
            if ( debug )
                fprintf(stderr,"san %d, an %d, mns %d\n", san , an ,mns );
            }


/*   Finished - found the optimum binary assignment!      */

    for ( j = 0; j < maxstate; j++ )
        {
        stnum[ j ] = savenum[ j ] ;
        strcpy( svar[ j ], svar[ stnum[ j ] ] );
                /* An interesting note:  Because of the cost sort the
                    final combination of y variables will always be in
                    ascending order in the svar array.        */
        fprintf( stderr,"OPT: y_variable %s, ", svar[ j ] );
```

```
        k = bindec( svar[ j ] );
        fprintf( stderr,"   decimal value %d, Cost %d\n",k, actcost[ stnum[j]]);
     $
 fprintf( stderr,"OPT: mns %d, san %d\n", mns, san ) ;
 fprintf( stderr,"OPT: The solution took %d iterations\n", pass ) ;


     $



/*****************************************************************/


/* BINDEC - Converts a character string of 1's and 0's
            to a decimal number.
                                                        */


bindec ( p )
char *p;
$
int j ,k, r;

k = j = 0;
r = nstates - 1;

while ( *(p + k) != 0 )
     $
     if ( *(p + k) == '1')
         j += power( r );
     k++;
     r--;
     $
return j;
$




/*****************************************************************/


/* BEST_COST - Determines the smallest actual cost of yvar or YVAR.
               The result is stored in svar and the cost is stored in
               actcost to minimize the number of Quine McClusky calls
                                                        */
best_cost ( strans, yvar, YVAR, svar, index, actcost )
int strans[MAXNUMBER][MAXCOMBIN], actcost[MAXCOL ], index;
char yvar[MAXCOL][MAXNUMBER], YVAR[ MAXCOL ][ MAXNUMBER ],
     svar[ MAXCOL ][ MAXNUMBER ];


$
int actual, j ,k ;
char s[ MAXNUMBER ][ MAXCOMBIN ];

actual = FALSE;

if ( actcost[ index ] == -1 )   /* Cost has not yet been calculated  */
     $
```

E - 18

```
            truth_table( s, strans, yvar[ index ] );
            dont_care( s, yvar[ index ] ) ;
            j = QMcClus( s, num2n, yvar[ index ], actual );

            truth_table( s, strans, YVAR[ index ] );
            dont_care( s, YVAR[ index ] ) ;
            k = QMcClus( s, num2n, YVAR[ index ], actual );
            if ( j > k )
                $
                actcost[ index ] = k;
                strcpy( svar[ index ], YVAR[ index ] );
                $
              else
                $
                actcost[ index ] = j;
                strcpy( svar[ index ], yvar[ index ] );
                $
            $
    return ( actcost[ index ] );
    $




    /**********************************************************************/
    /*
    DONT_CARE - fills in the don't care states to the truth table
                                                                    */

    dont_care( table, string )
    char table[ MAXNUMBER ][ MAXCOMBIN ], string[ MAXNUMBER ] ;
    $
    int j, k ;

    for ( j = nstates ; j < num2n ; j++ )
        $
        for ( k = 0 ; k < numinp ; k++ )
            table[ j ][ k ] = '1' ;
        string[ j ] = 0 ;      /* Make sure that the string is completely
                                    terminated                */
        $

    $



    /**********************************************************************/

    /* VALID - determines if a state assignment scheme is valid   */

    valid ( stnum, yvar )
    char yvar[MAXCOL][MAXNUMBER];
```

```c
int stnum[MAXCOL];

$
int j, k, i, weight, wt[MAXCOL];

if ( mass_debug )
    for ( j = 0; j < maxstate; j++)
        fprintf(stderr,"Valid: %s\n", yvar[ stnum[ j ] ] );

j = 0;
while ( yvar[ 0 ][ j ] != 0 )
    $
    k = 1;
    weight = 0;
    for ( i= 0; i < maxstate; i++)
        $
        weight += (( yvar[ stnum[ i ] ][ j ] - '0') * k);
        k *= 2;
        $
    for ( i = 0; i < j; i++)
        if ( weight == wt[ i ] ) return FALSE;
    wt[ j ] = weight;
    j++;
    if ( mass_debug )
        fprintf(stderr,"Valid: weight %d\n", weight);
    $
return TRUE;
$




/**************************************************************/

/* NEXTASN - gets the next combination of state assignment columns  */

nextasn( stnum, p, limit )
int stnum[MAXCOL], p, limit ;
$
int val;

val = TRUE;
if ( stnum[ p ] < limit - 1 ) stnum[ p ] += 1;
  else
    if ( p > 0 )
        $
        val = nextasn( stnum, p - 1, limit - 1 );
        stnum[ p ] = stnum [ p - 1 ] + 1;
        $
      else
        if ( stnum [ p ] < dsanum - maxstate )
            stnum[ p ] += 1;
          else
            val = FALSE;
return val;
$
```

```c
/*******************************************************************/

/* MERGE - produces the final output file for PRESTO    */

merge ( strans, otable, svar )
char otable[MAXNUMBER][MAXCOMBIN], svar[MAXCOL][MAXNUMBER];
int  strans[MAXNUMBER][MAXCOMBIN]  ;
{
int j, k, l, m;
char store[ MAXNUMBER ];

fprintf(fout,".i%d   n.o%d", ninputs + maxstate, noutputs + maxstate );
fprintf(fout," n.p%d n", nstates * numinp ) ;
for ( j = 0; j < nstates ; j++ )
    for ( k = 0; k < numinp; k++)
        {
        decbin( store, k, ninputs );      /* Input signals   */
        fprintf(fout,"%s", store);
        for ( l = 0; l < maxstate; l++)  /* Present State    */
            putc( svar[ l ][ j ], fout );

        putc( ' ', fout );                      /* Put a space between the input and
                                    output parts of the PRESTO input file */
        m = strans[ j ][ k ]  - 1;  /* Use m as an array pointer for svar */
        for ( l = 0; l < maxstate; l++)
            putc( svar[ l ][ m ], fout );   /* Next State    */

        for ( l = 0; l < noutputs; l++ )
            putc( otable[ j ][ k + l ], fout );  /* Output signals   */
        putc( '\n', fout );
        }
fprintf( fout, ".e\n");


}



/*******************************************************************/

/* GENO - generates distinct state columns with n '0's    */

gen0( yvar, n, varnum, varpos)
char yvar[MAXCOL][MAXNUMBER];
int n, varnum, varpos;
{

if ( varpos == 0 ) yvar[varnum][varpos] = '0';
if ( n > 1 )
    for ( varpos++ ; varpos < nstates ; varpos++ )
        {
        yvar[varnum][varpos] = '0';
        varnum = gen0( yvar, n-1, varnum, varpos);
```

```c
                yvar[varnum][varpos] = '-';
                }
        else
          if ( n == 1 )
                {
                strcpy( yvar[varnum + 1], yvar[varnum] );
                yvar[varnum++][varpos] = '0';
                if ( mass_debug )
                    fprintf(stderr,"GEN0 yvar: %s\n", yvar[varnum - 1]);
                }
        else error(8);

    return varnum;
    }




/******************************************************************/


/* GEN1 - generates distinct state assignment columns with n '1's */

gen1( yvar, n, varnum, varpos)
char yvar[MAXCOL][MAXNUMBER];
int n, varnum, varpos;
{

if( mass_debug )
    fprintf(stderr,"GEN1 \n %d, varnum %d, varpos %d n", n, varnum, varpos);
if ( n > 1 )
    while ( ++varpos < nstates )
        {
        yvar[varnum][varpos] = '1';
        if ( mass_debug )
            fprintf(stderr,"GEN1 yvar: %s\n", yvar[varnum]);
        varnum = gen1( yvar, n-1, varnum, varpos);
        yvar[varnum][varpos] = '-';
        }
    else
      if ( n == 1 )
        while ( varpos < nstates - 1 )
            {
            strcpy( yvar[varnum + 1], yvar[varnum] );
            yvar[varnum][++varpos] = '1';
            if ( mass_debug )
                fprintf(stderr,"GEN1 yvar: %s\n", yvar[varnum]);
            varnum++;
            }
    else error(9);

return varnum;
}



/******************************************************************/
```

E - 22

```
/* DECBIN - converts decimal number to binary string of '0's and '1's */

decbin( p, n, numlen)
int n, numlen;
char *p;
{
int ps[MAXNUMBER], j, k;

j = 0;
while ( n != 0 )
    {
    ps[j++] = ( n % 2 ) + '0';
    n = n / 2;
    }
j--;
for ( k = 0; k < numlen; k++)
    if ( k == numlen - 1 - j )
        *(p+k) = ps[j--];
     else
        *(p + k) = '0';
*(p + k) = 0;             /*  Terminate the string          */

if ( debug )
    fprintf(stderr,"DECBIN: %s\n", p);
}




/*****************************************************************/

/* CODE_SIMPLE - Simple assignment   */

code_simple( strans, svar)
char    svar[MAXCOL][MAXNUMBER];
int strans[ MAXNUMBER ][ MAXCOMBIN ] ;
{
int j, k, m, actual, total ;
char st[ MAXNUMBER ], s[ MAXNUMBER ][ MAXCOMBIN ];

m = 0 ;
total = 0 ;
actual = FALSE ;

for ( j = 0; j < nstates; j++)
    {
    decbin( st, j, maxstate );           /* Generate binary numbers  */
    for ( k = 0; k < maxstate ; k++ )
        svar[ k ][ j ] = st[ k ];
    }
for ( j = 0 ; j < maxstate ; j++ )       /* Get them in y variable format */
    {
    for ( k = 0 ; k < nstates ; k++ )
        st[ k ] = svar[ j ][ k ] ;
```

```
            st[ nstates ] = 0 ;
            truth_table( s, strans, st ) ;
            m = QMcClus( s, num2n, st, actual ) ;
            total += m ;
            fprintf(stderr,"SIMPLE: variable %s, cost %d\n", st, m ) ;
            }
        fprintf(stderr, "SIMPLE: Cost %d\n", total ) ;
        }




/***********************************************************************/


/* CODE_GREY - Gray code         */

code_grey( strans, svar)
char    svar[MAXCOL][MAXNUMBER];
int strans[ MAXNUMBER ][ MAXCOMBIN ] ;
{
int j, k, m, actual, total ;
char st[ MAXNUMBER ], s[MAXNUMBER ][ MAXCOMBIN ];

code_simple( strans, svar );              /* Get binary assignments */
for ( j = 0; j < nstates; j++ )
    {
    for ( k = 0; k < maxstate ; k++ )   /* Convert them to gray code */
        st[ k ] = svar[ k ][ j ];
    gray_code( st );
    for ( k = 0; k < maxstate ; k++ )
        svar[ k ][ j ] = st[ k ];
    }

m = 0 ;
total = 0 ;
actual = FALSE ;

for ( j = 0 ; j < maxstate ; j++ )
    {
    for ( k = 0 ; k < nstates ; k++ )   /* Convert them t yvariable format */
        st[ k ] = svar[ j ][ k ] ;
    st[ nstates ] = 0 ;                 /* Terminate the string    */
    truth_table( s, strans, st ) ;
    m = QMcClus( s, num2n, st, actual ) ;
    total += m ;
    fprintf( stderr, "GRAY CODE: variable %s, cost %d\n", st, m ) ;
    }
fprintf( stderr, " GRAY CODE Cost: %d \n", total ) ;

}


/***********************************************************************/
```

```
/* POWER - raises the given integer to a power of two   */


power(n)
int n;
{
int j, pow = 1;

if ( n == 0 ) return pow;
  else for (j = 0; j < n; j++)
          pow = pow * 2;
return pow;
}




/*******************************************************************/


/*      LOADINT - loads integer strings of maximum length MAXLEN      */

loadint( p, numrow, numcol )
int p[MAXNUMBER][MAXCOMBIN];
int numrow, numcol;
{
int j, k, check;

if(debug)
    {
    fprintf(stderr,"LOADINT: the number of state rows to load is %d\n",numrow);
    fprintf(stderr,"LOADINT: the number of next state columns is %d\n",numcol);
    }
for (j = 0; j < numrow; j++)
    {
    for ( k = 0; k < numcol ; k++)
            {
            scanf("%d", &check);
            if ( check != NULL ) p[j][k] = check;
              else error(4);
            if ( mass_debug ) fprintf(stderr," %d", p[j][k]);
            }
    if ( mass_debug ) fprintf(stderr,"\n");
    }
}




/*******************************************************************/


/* LOADCHAR - loads character strings.       */

loadchar(otable, nstates)
char otable[MAXNUMBER][MAXCOMBIN];
int nstates;
{
int j, check;
```

```
for (j = 0; j < nstates; j++)
    {
    if((check = scanf("%s", otable[j]) ) == NULL ) error(5);
    if ( (strlen(otable[j]) ) != numinp * noutputs) error (6);
    if ( mass_debug )
        fprintf(stderr,"LOADCHAR:  %s\n",otable[j]);
    }

}




/*********************************************************************/


/* SETNUM - sets the value of global variables dsanum, maxstate,
            num2n, and num2n_1                      */

setnum()
{
int j, k, n;

n = 0;
while ( (power(n) < nstates) ) n++;
num2n = power(n);        /* Calculate the number of state variables required */
maxstate = n;
num2n_1 = power( (n-1) );
num2R_1 = power( (nstates-1) ) - 1;
num2R = power( nstates );

dsanum = 0;
for ( j = ( nstates - num2n_1); j <= num2n_1; j++)
    dsanum += bionom( nstates, j);
dsanum = dsanum/2;
if (debug)
    {
    fprintf(stderr,"SETNUM: n %d, num2n %d, maxstate %d,", n, num2n, maxstate);
    fprintf(stderr," num2n_1 %d, num2R_1 %d n", num2n_1, num2R_1);
    fprintf(stderr,"       num2R %d, dsanum %d \n", num2R, dsanum);
    }
}




/*********************************************************************/


/* BIONOM - calculates the numerical result of x things taken y at a time  */

bionom(x,y)
int x, y;
{
int j, diff, n, m;

n = x;
```

E - 26

```
        m = y;
        diff = x - y;
        for ( j = 1 + diff; j < x; j++) n *= j;
        for ( j = 1; j < y; j++) m *= j;
        if( debug )
            fprintf(stderr,"BIONOM: %d things taken %d at a time is %d\n",x,y,n/m);
        return ( n/m );
        }




/***************************************************************************/

/* PRNT() - print a single dimension array until value EOR or -2 is reached */

prnt(p)
int *p;
{
int *p1;
p1 = p;
fprintf(stderr,"Partitions:  n");


    for ( p1 = p; *p1 != EOR; p1++)
        fprintf(stderr," %d", *p1);
    fprintf(stderr,"\n");
}




/***************************************************************************/

/* FORMAT - prints an text file of global variables Nstates, Ninputs,
            Noutputs, Symbol, Lambda, and the two arrays: Stable and Otable
                                                                        */

format(stable,otable)
int stable[MAXNUMBER][MAXCOMBIN];
char otable[MAXNUMBER][MAXCOMBIN];
{
int j, k, i;

if ( debug )
    {
    fprintf(stderr,"Hello from FORMAT!  Prints the state ");
    fprintf(stderr," transition and output arrays. n");
    }
fprintf(stderr,"%d %d %d %d %1.1f",nstates,ninputs,noutputs,symbol,lambda );
fprintf(stderr," %d\n\n", errorstate);
for ( j = 0; j < nstates; j++)
    {

    for ( k = 0; k < numinp; k++)
```

```
                    fprintf(stderr,"%d ",stable[j][k]);
                fprintf(stderr,"\n");
            }
        fprintf(stderr,"\n");
        if ( ( numinp * noutputs ) > NUMCOL )
            {
            fprintf(stderr,"Warning - the output array will print funny since ");
            fprintf(stderr,"numinp * noutputs is longer than %d.", NUMCOL);
            }
        for ( j = 0; j < nstates; j++)
            fprintf(stderr, "%s\n",otable[j]);
        }




/**********************************************************************/

/* FORM - prints just the transition table    */

form ( strans )
int strans[MAXNUMBER][MAXCOMBIN];
{
int j, k;

for ( j = 0; j < nstates; j++)
    {
    for ( k = 0; k < numinp; k++)
        fprintf(stderr,"%d ",strans[j][k]);

    fprintf(stderr,"\n");
    }
fprintf(stderr,"\n");
}




/**********************************************************************/

/* ERROR - Prints error messages to stderr    */

error(n)
int n;
{
char *p1 = "ERROR - Usage: assign [-d] [-sa,-gc] outfile.ext < infile ",
*p2 = "ERROR - Can't open the output file!",
*p3 = "ERROR - One of the state table parameters is incorrect.",
*p4 = "ERROR - Next state input file error!",
*p5 = "ERROR - Output string incorrect. ",
*p6 = "ERROR - Output string is not equal to numinp ",
*p7 = "ERROR - Wrong number of distinct State Assignment Columns",
*p8 = "ERROR - Gen0 called with n less than 1. ",
*p9 = "ERROR - Gen1 called with n less than 1. ";

switch(n)
    {
```

```
        case 1: fprintf(stderr,"%s\n\n", p1); break;
        case 2: fprintf(stderr,"%s\n\n", p2); break;
        case 3: fprintf(stderr,"%s\n\n", p3); break;
        case 4: fprintf(stderr,"%s\n\n", p4); break;
        case 5: fprintf(stderr,"%s\n\n", p5); break;
        case 6: fprintf(stderr,"%s %d. n n", p6, numinp); break;
        case 7: fprintf(stderr,"%s\n\n", p7); break;
        case 8: fprintf(stderr,"%s\n\n", p8); break;
        case 9: fprintf(stderr,"%s\n\n", p9); break;
        default: fprintf(stderr,"ERRONEOUS call to ERROR!!!\n\n");
        $
    exit(1);
    $
```

CFORM.C

```
/* CFORMAT - changes PRESTO's output file to the format
        required by PLAGEN.  It also adds the CIF number and lambda
        size, supplied in the command line, to the file.
        CFORMAT uses the standard input.  Input files are supplied
        by I/O redirection.  CFORMAT assumes that the input and output
        arrays are separated by a space.  The output file is supplied
        in the command line.
        Usage: cform symbolic_state.table  < assign.out
                                                            */


#include <stdio.h>
#define READ "r"
#define WRITE "w"
#define NULL 0
/*              MAIN PROGRAM                    */


main(argc, argv)
int argc;
char *argv[];
{
char string[2], c;
int numinp, numout, numprod, symbolnum, j, k ;
float   lambda;
FILE *f1 ;


if (argc != 2 ) error(0);
if ((f1 = fopen(argv[1],READ )) == NULL) error(4);
numinp = numout = numprod = 0;     /* Initialize array size variables for
                                      later error check that the PLA has
                                      no zero parameters          */
fscanf( f1,"%*d %*d %*d %d %f", &symbolnum, &lambda );


for (j = 0; j < 3; j++)
    {                               /* Get the PLA parameters   */
    scanf("%2s", string);
    if ( string[0] != '.') error(1);
        switch (string[1])
        {
        case 'i': scanf("%d", &numinp); break;
        case 'o': scanf("%d", &numout); break;
        case 'p': scanf("%d", &numprod); break;
        default: error(1);
        }
    }


if (numinp * numprod * numout == 0) error(1);
                /* Write the PLA parameters to the output file */
printf( "%d, %d, %d, ",numinp, numprod, numout);
printf( "%d, %1.1f\n", symbolnum, lambda);
                /* Get the PLA specifications and transfer them to the output
```

F - 1

```
                         file.  The input values are not changed.  The output
                         values are changed as shown in the switch.        */

        for (j = 0; j < numprod; j++)
            $
            for (k = 0; k < numinp; k++)                    /* Handles Input Array */
                $
                scanf("%1s", &c);
                putchar( c );
                $
            putchar(' ');
            for (k = 0; k < numout; k++)                    /* Handles Output Array */
                $
                scanf("%1s", &c);
                switch (c)
                    $
                    case '1': putchar('-'); break;
                    case 'x': putchar('0'); break;
                    case '-': putchar('0'); break;
                    case '0': putchar( c ); break;
                    default: error(3);
                    $;
                $
            putchar( ' n' );
            $
        $


error(n)
int n;
$
switch (n)
    $
    case 0: fprintf(stderr,"Usage: cformat symbolic_state.table");
            fprintf(stderr,"\nStandard call: cform sym_st.table < ");
            fprintf(stderr," a.out | plagen\n\n");
            break;
    case 1: fprintf(stderr,"ERROR - PLA parameters \n\n"); break;
    case 2: fprintf(stderr,"ERROR - The number of characters in the input");
            fprintf(stderr," string did not match the number associated\n");
            fprintf(stderr,"with .i.  The required space between the input");
            fprintf(stderr," and output arrays may be missing.\n\n");
            break;
    case 3: fprintf(stderr,"ERROR - The output array had an illegal charac");
            fprintf(stderr,"ter.\n       Legal characters are '-', '1', 'x'");
            fprintf(stderr,", and '0'.\n"); break;
    case 4: fprintf(stderr,"ERROR - UNABLE to open output file.\n\n");
            break;
    default: fprintf(stderr,"ERROR - erroneous call to error! \n\n");
    $
exit(0);
$
```

```
/*****************************************************************/
/*
        Make_sfsm - Creates the SFSM CLL file
                                                    */
#include <stdio.h>
#define FALSE 0
#define TRUE 1



main(argc, argv)
int argc ;
char *argv[] ;
{
FILE *fin1, *fin2, *fout ;
int n0, n1, n2, n3, n4, n5, n6, numinp ;
int j, k, maxstate, found;
char c, store[ 80 ], string[ 20 ] ;

strcpy( string, "external" );
fin1 = fopen( argv[ 1 ], "r" );
fin2 = fopen( argv[ 2 ], "r" );
fout = fopen( "sfsm_pla.clf", "w" ) ;



fscanf( fin2, "%d %d", &j, &numinp );
k = 1;
maxstate = 0;

while ( k < j )
    {
    k *= 2 ;
    maxstate++ ;
    }

found = FALSE ;
while ( (c = getc( fin1 )) != EOF )
    {
    if ( ( c == '(' )&&( found == FALSE ) )
        {
        fscanf( fin1,"%s", store  );
        if ( strcmp( string, store ) == 0 )
            {
            j = strlen( store )  ;
            store[ j++ ] = ' ' ;
                        /* Make sure that there is a space between words */
            while( (c = getc( fin1 ) ) != ')' )
                store[ j++ ] = c ;
            store[ j++ ] = c ;  /* Get the last character            */
            store[ j ] = 0 ;    /* Terminate the string              */
            while ( (c = getc( fin1 ) ) != '\n') ;
            found = TRUE ;
```

```c
                        $
                    else
                        $
                        putc( c, fout ) ;
                        fprintf( fout, "%s", store );
                        $
                    $
                else
                    putc( c, fout ) ;          /* Write all characters in the file except the
                                        external pla message - it can appear anywhere
                                        in the file and disrupt the cif file!  */
                $

        printf( "#include \"/usr/lib/local/s_ext.cil\" \n") ;
        printf( "%s \n", store ) ;
        printf( " nSFSM  n$  n npla(0,0); \n") ;

        n1 = maxstate + numinp ;
        n2 = n1 / 2  ;
        n3 = 28 + n1 * 16 ;
        n4 = 12 ;
        n5 = 22 ;
        n6 = 7 ;

        printf( "iterate %d, 1  16, 0\n", n1 ) ;
        printf( "    PlaClockIn ( 15, --58 ); \n" ) ;
        printf( "iterate %d, 1 16, 0\n", n2 ) ;
        printf( "    PlaClockOut ( %d, --53 ); \n", n3 );

        n3 += 3 ;
        for ( j = 0; j < maxstate ; j++ )
            $
            printf( "wire poly  %d, --53 w 2 d %d l %d ", n3, n4, n5 );
            printf( "diff u %d ;\n", n6 );
            n3 += 8 ;
            n4 += 10 ;
            n5 += 24 ;
            n6 += 10 ;
            $
        printf( "\n$ \n \n" ) ;

        fclose ( fout ) ;
        $
```

VITA

Darrell Clarke Pelan was born on 10 June 1957 at Larson Air Force Base in Moses Lake, Washington. He graduated from high school in Bothell, Washington in 1975 and attended the University of Washington from which he received the degree of Bachelor of Science, Electrical Engineering in June 1980. Upon graduation, he received a commission in the USAF through the Officer Training School program. He served as a Computer Software Engineer in the Embedded Computer Branch of the Air Force Acquisition Logistics Division, Wright-Patterson Air Force Base, Ohio until entering the School of Engineering, Air Force Institute of Technology, in June 1982.

Permanent Address: 6112 NE 154th St

Bothell, Washington 98011

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS			
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE				

4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/EN/83D-57	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL (If applicable) AFIT/EN	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City. State and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433		7b. ADDRESS (City, State and ZIP Code)

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER			
8c. ADDRESS (City, State and ZIP Code)		10. SOURCE OF FUNDING NOS.			
		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
11. TITLE (Include Security Classification) See Box 19					

12. PERSONAL AUTHOR(S)
Darrell C. Pelan, BSEE, 1Lt, USAF

13a. TYPE OF REPORT MS Thesis	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) 1983 December	15. PAGE COUNT 137

16. SUPPLEMENTARY NOTATION

Approved for public release: IAW AFR 190-17.
7 Feb 84
LYNN E. WOLAVER
Dean for Research and Professional Development
Wright-Patterson AFB OH 45433

17.	COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: PLAFST Programmable Logic Array From State Table

Thesis Chairman: Harold W. Carter, Lt Col, USAF

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION
22a. NAME OF RESPONSIBLE INDIVIDUAL Harold W. Carter, Lt Col, USAF	22b. TELEPHONE NUMBER (Include Area Code) 513-255-3633	22c. OFFICE SYMBOL AFIT/EN

AFIT/GE/EE/83C-57

Abstract

Programmable Logic Array From State Table (PLAFST) is a computer
aided design (CAD) tool that takes a symbolic state table as input and
produces a very large scale integrated (VLSI) circuit implementation of
the symbolic state table. The state table is first reduced symbolically
using equivalence partitioning. A near optimal binary state assignment
is made based on the Story, Harrison, and Reinhard procedure as modified
by Noe and Ryhne. Distinct state assignment variables are sorted based
on cost estimates obtained by increasing the number of adjacencies in
the state transition table. Once sorted, the actual costs of valid
state assignments made from the state variables are calculated. Since
state assignments with the lowest cost estimates are investigated first,
an optimal solution is found with a small number of iterations. This
binary state assignment is demonstratably less costly than either simple
or gray code assignments of the state variables. The VLSI circuit
consists of a programmable logic array (PLA) and clocked buffers. The
state buffers are properly interconnected. The final outputs are Chip
Layout Language (CLL) and Caltech Intermediate Format (CIF) descriptions
of the integrated circuit. PLAFST also plots the final integrated
circuit.

END

FILMED

3-84

DTIC