THE LOGIC ORGANIZER AND DIAGNOSIS PROGRAMS

Sundaram Seshu

REPORT R-226   JULY, 1964

COORDINATED SCIENCE LABORATORY
UNIVERSITY OF ILLINOIS
URBANA, ILLINOIS

The research reported in this document was made possible by support extended to the University of Illinois, Coordinated Science Laboratory, under the Joint Services Electronics Program by the Department of the Army, Department of the Navy (Office of Naval Research), and the Department of the Air Force (Office of Scientific Research), and by the Advanced Research Projects Agency under Department of the Army contract DA-28-043-AMC-00073(E) and Department of the Air Force (Office of Scientific Research) contract AF 49(638) – 1383
THE LOGIC ORGANIZER AND DIAGNOSIS PROGRAMS

by

Sundaram Seshu
This report is divided into two parts. The first part is a user's manual for those wishing to use the system of programs for the diagnosis of sequential switching circuits. The second part is a programmer's manual. Here detailed descriptions of the individual subroutines, including calling sequences and peculiarities are given. This system of programs was developed as a tool in the experimental study of self-diagnosis in electronic digital computers. The system consists of two programs—the organizer (5000 instructions) and the sequential circuit analyzer (10,000 instructions) with about 1500 instructions in common. The organizer is a program for arranging the logic in levels and identifying any feedback loops not identified in the input. The sequential circuit analyzer is a program for producing the testing procedure for any given sequential circuit. The organizer prepares the input for the analyzer. The entire system is written in machine language for the CDC 1604 computer and uses the Illinois system subroutines for input-output.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>PART I. USER'S MANUAL</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. GENERAL</td>
<td>2</td>
</tr>
<tr>
<td>2. INPUT FORMAT</td>
<td>3</td>
</tr>
<tr>
<td>3. CONTROL PROGRAM</td>
<td>7</td>
</tr>
<tr>
<td>4. USE OF ORGANIZER</td>
<td>13</td>
</tr>
<tr>
<td>5. USE OF SEQUENTIAL CIRCUIT ANALYZER</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PART II. PROGRAMMER'S MANUAL</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6. GENERAL STRUCTURE AND CONVENTIONS</td>
<td>23</td>
</tr>
<tr>
<td>7. THE CONTROL PROGRAM</td>
<td>31</td>
</tr>
<tr>
<td>8. EDITING</td>
<td>39</td>
</tr>
<tr>
<td>9. ORGANIZER</td>
<td>41</td>
</tr>
<tr>
<td>10. STRUCTURE OF SEQANALZ</td>
<td>45</td>
</tr>
<tr>
<td>11. THE COMPILER AND HUFFMAN ANALYZER</td>
<td>47</td>
</tr>
<tr>
<td>12. BOOKKEEPING AND ARITHMETIC SUBROUTINES</td>
<td>53</td>
</tr>
<tr>
<td>13. OTHER INTERMEDIATE LEVEL ROUTINES</td>
<td>60</td>
</tr>
<tr>
<td>14. STRATEGY ROUTINES</td>
<td>69</td>
</tr>
<tr>
<td>15. CONTROL ROUTINES FOR SIMULATION AND DIAGNOSIS</td>
<td>73</td>
</tr>
</tbody>
</table>

APPENDIX OF EXAMPLES ................................................. 77

INDEX OF SUBROUTINES (PART II) ................................. 88
PART I

USER'S MANUAL
1. General

The system of programs described here was developed as a tool in the experimental study of self-diagnosis in electronic digital computers. This system performs the following overall function. Given the description of a sequential switching circuit, the system produces a testing procedure for detecting and isolating logical malfunctions. The basic model and assumptions have been described in detail by Seshu and Freeman [1] and is not to be repeated here. The present program differs from the Seshu-Freeman program primarily in increased flexibility and power. A general description of the program and its relationship to the Seshu-Freeman program is being issued as a separate report [2].

The function of the logic organizer program is to rearrange the input information (logic circuit description) into a form suitable for the analyzer. It thus serves as a buffer between the engineer and the analyzer. The analyzer demands that the logic be arranged in levels and that all inputs, outputs, and feedback be identified. The organizer arranges the logic and locates all unidentified feedback loops. It is left up to the engineer to locate the feedback first and last blocks. An editing subroutine permits the latter function to be performed on-line if desired.

Both programs contain a rather elaborate program (READCNTR) which reads and obeys control statements. This program is similar to the CDC FORTRAN resident. The program READCNTR is described in Section 3. In the organizer, the subroutine READCNTR and the editing subroutine are the only programs that read control information. In the analyzer control information may be read by the subroutines READCNTR, OPTION and SIMULATE.
The control medium may be assigned as paper tape reader, typewriter or any mag tape on cabinet 2. The input medium may be any mag tape on cabinet 2. Several output media may be specified simultaneously and may be any mag tape on cabinet 2, paper tape, typewriter or printer.

In addition to the operational routines, both programs contain several auxiliary routines which were originally inserted for debugging convenience or program preparation. Some of these have been retained in the final version as potentially useful, and are described in Chapter 2.

This system of programs was started on December 28, 1962, and was considered operational on April 10, 1964.

2. Input Format

The input to the program is a description of the logic circuit as one file of 88 to 120 character BCD records on a low density magnetic tape. The input tape may contain several files (describing several logic circuits). The end of information is signified by an extra end of file. A special program called MERGE is available in the organizer for converting 80 character records into 120 if the input information is made up on cards. This format is discussed at the end of the section.

The first record of the file is the identifier. Columns 10-14 contain IDENT and columns 20-27 contain the name of the circuit, left justified and with no internal spaces. If synchronous simulation is desired, columns 41-51 contain SYNCHRONOUS. Otherwise, it is left blank.
Comment records are identified by an asterisk in column 1 and may appear anywhere after the identifier.

All other records are divided into 11 fields of 8 characters each in columns 1-8, 9-16, 17-24, 25-32, 33-40, 41-48, 49-56, 57-64, 65-72, 73-80, 81-88. In all fields the information must be left justified, and there may be no internal spaces.

The first group of records following the identifier are the heading records. These records contain input, output, etc. information and must precede the block information. These heading records have the following format:

Columns 1-8

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>OUTPUTS</th>
<th>FEEDBK</th>
<th>NUMITER</th>
<th>PRVTSTD</th>
<th>RESETS</th>
</tr>
</thead>
</table>

Columns 9-16

Decimal integer, left justified.

The decimal integer specifies number of inputs, outputs, feedback lines, number of iterations (for oscillation check), number of previously tested blocks or number of resets, respectively. For the INPUTS, OUTPUTS, FEEDBK and PRVTSTD directives, the immediately following records contain the relevant names, 10 per record in the fields 9-12, 17-20, 25-28, 33-36, 41-44, 49-52, 57-60, 65-68, 73-76, 80-83. (Names of blocks and inputs are restricted to 4 characters.) Reset vectors are given one per record, immediately following the reset directive. Reset vectors are given packed, left justified octal. Reset feedback state starts in column
9 and reset input in column 25. Any one or more of the heading records may be omitted. They may appear in any order except that FEEDBK and INPUTS directives must precede the RESETS directive. We may note here that resets may also be entered through the control medium in either the organizer or the analyzer.

The heading group is followed by the block (logical element) information records. Each block is described by a group of records in the following format:

Record 1 identifies the block:

Cols 1-4: Name of block (left justify)
Cols 9-12: Type (AND, OR, NOT etc. Left justify)
Cols 17- : Number (or list of numbers) of inputs.

The immediately following records contain the list of inputs, 10 per record in columns 9-88. Columns 1-8 must be blank. In each 8 character field, the name of the input occupies first four characters. This may be followed by pin number if desired. If the input is a feedback input, the last non-blank character is an *. The input list may be followed by output information if desired. (The reading routine skips until it finds a record with cols. 1-8 non-blank).

The programs recognize both ordinary and two-level macro blocks. The ordinary blocks currently recognized are: AND, OR, NOT, NAND, NOR, EXOR, IFNF. (EXOR is "exclusive or" and IFNF is "if and only if"). Macro-blocks currently recognized are:
AON  And-or-not (AND followed by NOR)
AOR  And-or
ORA  Or-and
OAN  Or-and-not (OR-NAND)

For macro blocks the input count field (cols 17- ) contain a list of numbers separated by commas (no internal spaces) and terminated by space. The input list is in the following records and must be ordered according to the list of numbers of inputs. Optionally, a separator may be used to separate inputs to the first level subblocks. This separator is 8 characters consisting of 4 blanks, + sign, 3 blanks. The separator is disregarded by the reading routine.

An end of file mark (tape mark, not double periods) terminates the block information and the logic circuit description.

Provision has also been made for input information on 80 character records (if originally made up on cards). The list of blocks (inputs, outputs, feedback, list of inputs to a block) are the only ones that go beyond column 80. These are to be listed 5 per card in the fields 9-16, 17-24, 25-32, 33-40, and 41-48. The program MERGE in the organizer will merge these into 120 character records for processing. The control statement is

```
Merge, x,y
```

where x is input tape number (80 char) and y is output tape number (120 char). Neither tape is rewound by MERGE. Only one file is merged by one control statement. End of file is copied automatically.
3. Control Program

The control programs in the organizer and the sequential circuit analyzer are nearly identical and are therefore described together in this section. The name of the program is READCNTR. This program reads and obeys control statements from the control medium. The control medium is initialized to paper tape. The input medium is initialized to mag tape 2. In the analyzer, the output medium is initialized to printer. In the organizer the output medium is initialized to printer, mag tape 3. The clock is cleared and started.

Control statements must begin in column 1, contain no internal spaces and are terminated by a carriage return. The parameters in control statements are restricted to 8 characters (except for reset vectors) and are separated by commas. In addition to control statements, one can also have comment and message records. Comments are distinguished by a + sign in column 1 and are copied onto the normal output media (the + is blanked out). Messages are distinguished by a - sign in column 1 and are typed out on the console typewriter (- is suppressed).

The various control statements that are currently available are discussed in the remainder of the section. If any control statement is in error, it is copied on the typewriter and control is transferred to the typewriter.

3.1 Assignment of Media

The input, output and control media may be reassigned as desired by the following control statements:
Control, $z$: Assign $z$ as the control medium. Permissible values of $z$ are $F$ (paper tape), $T$ (typewriter), 1,2,3,4 (mag tapes).

Input, $z$: Assign $z$ as the input (logic circuit input) medium. $z = 1,2,3$ or 4.

Output, $x,y,z,w,...$: Assign $x,y,z,...$ as output media. In the organizer, the organized logic is written on the last mag tape in the list and input errors and feedback loops are written on the first output medium. $F,T,P$ (printer), 1,2,3,4 are all valid output media.

3.2 Tape Handling

Rewind, $z$ \hspace{1cm} $z = 1,2,3,4,F,P$. Rewind tape $z$ or put out leader or eject printer page.

Bksp,$z,n$ Read backspace tape $z$ by $n$ records. $z = 1,2,3,4$.

Skiprecd,$z,n$ Skip $n$ records on tape $z$. $z = 1,2,3,4$.

Skipfile,$z,n$ Skip $n$ files on tape $z$. $z = 1,2,3,4$.

Wef,$z,y$ Write end of file on tape $z$. $z = 1,2,3,4$.

If $y = 1$, increase no. of files of output by 1.

Unload, $z$ Unload (rewind with interlock) tape $z$.

Autoload,$z$ Simulate autoload for tape $z$. (Control goes away to the bootstrap.)

In addition, there are executable tape handling programs COPYFILE and COPYRECD described in Section 3.3.
3.3 Execution

Any subroutine may be executed by the control statement

\[ \text{Xeq, Prog, A, Q} \]

Prog is either a recognized symbolic name or an absolute octal address. In the case of recognized names, the directive may also be given in the format:

\[ \text{Prog, A, Q} \]

omitting the Xeq. A and Q are (signed) decimal integers of eight characters or less (including sign) other than \(-\phi\). These are loaded into A and Q at entry into Prog. If A and/or Q is not given, it is taken as \(\phi\). For certain programs, these parameters are interpreted differently.

The following programs may be referred to symbolically in either the organizer or the analyzer:

- **Copyfile, \(x,y,n\)**: Copy \(n\) files from tape \(x\) onto tape \(y\).
- **Copyrecd, \(x,y,n\)**: Copy \(n\) records from tape \(x\) onto tape \(y\).
- **Setkey, \(n\)**: Ask operator to set jump switch \(n\) and wait for it to be set.
- **Resetkey, \(n\)**: Ask operator to reset (to normal) jump switch \(n\) and wait for it.
- **Keytest, \(n,z\)**: Wait for jump switch \(n\) to be set if \(z\neq\phi\) and reset if \(z=\phi\).
- **Flexload** : Flex load program in ILLRES.
- **Load** : System loader in ILLRES.
Octaload, n

Compact octal loader used for debugging.
n≠0 for no check sum. Input from control medium. Record 1 has 5 character address. Succeeding records have 16 character octal digits (left packed) to be loaded into successive words. Double periods for new origin. Two successive double periods terminate loading.

Octldump, x,y

Octal dump on paper tape from x to y (octal) in the format of Octaload, with check sum.

The following programs may be referred to in the organizer only:

INPUTS, OUTPUTS, FEEDBK, CHANGE, PRVTSTD, MERGE

See write-up of Edit in Section 4 for these directives. (MERGE is described in Section 2.)

The following programs may be referred to in the analyzer only:

DIAGNOSE, COMPILE, SIMULATE, OPTION, STRADUMP, STRAREAD, INITLZN

See Section 5 for these directives.

3.4 Logic Input Handling

The directive for reading the logic description has slightly different forms in the organizer and the analyzer. In the organizer the format is:
Logic $x,z$

Here $x$ is the name (identifier) and $z$ is input tape number. If $z$ is omitted, the input medium is used. The format in the analyzer is

Logic, $x,y,z$

Parameters $x$ and $z$ have the same significance as before. The parameter $y\neq 0$ to copy all the comment records (asterisk in column 1) onto the output media and $y=0$ otherwise.

Note that the input tape is rewound before and after reading. Thus the control medium may not be the same as the input medium.

Reset information may also be inputted from the control medium. The logic must be read (and in the case of the organizer, any required editing of feedback and inputs done) before resets are read in. The directive format is:

Resets, n, add

where $n$ is the number of resets (maximum 100). Resets follow immediately, one per record, in the format $O_m, l_x, O_n$ where $m=$no. of fb lines$/3$ and $n=$no. of primary inputs$/3$, both rounded up. (See examples in the appendix.) If "add" is omitted, list of resets is initialized.
3.5 Miscellaneous

Other control statements available are tabulated below:

End Exit from program

Call, z Call program z from master tape

Time, φ Clear location φ and start clock

Time, P Print time

Time, S Stop clock

Dump, x,y,z,w Dump from x to y (octal) in operation mode z and control mode w (see SNAPSHOT in ILLRES). If z omitted, octal dump given. If w omitted, self-control used.

Pause Type Pause (J2) and wait for jump switch 2 to be set (momentarily).

Transfer, z Unconditional transfer to location z (octal)

Save, z Save all cores and partial output (if on tape) on tape z with bootstrap. Can be read back by autoload, z.

Tdump, z Dump contents of location z (octal) on typewriter

Store, x,y Store y (decimal integer) in location x. x may be a recognized symbol or an octal address. In organizer, recognized symbols are NUMITER, CUTFLAG. In analyzer, recognized symbols are NUMITER, MSGFLAG, CRITERON, CUTFLAG, WANDER, INVTFAIL, SAFEFLAG.
4. Use of Organizer

4.1 General

The purpose of the organizer is to arrange the input logic description as made up by the engineers into a format suitable for the sequential circuit analyzer. For convenience of compilation, the analyzer requires that the logic be organized into levels and all feedback loops be identified. Except in the case of very simple circuits, this is a major task and is not easy to do by hand. The organizer program simply locates the undefined feedback loops and prints them out. It does not attempt to identify the feedback first and last blocks. Special fine editing procedures are provided, which allow one to define the feedback loop from the control medium instead of having to retype the tape or edit the mag tape through standard tape editing routines. Other information such as resets and previously treated blocks may also be added.

The normal input medium is tape unit 2. Normally the organized output is written on tape unit 3. Undefined inputs and undefined feedback loops are normally listed on the printer.

4.2 Editing

Fine editing of the input information is done by a set of directives. These are written as control statements, followed by any required information. The editing procedure is not intended for correcting typographical errors. Therefore, special care has to be observed if it is used for this purpose. The logic must be read before any editing is done. As the block list is read, the inputs to a block are coded to
signify whether they are primary, feedback, or internal. Therefore, if the list of primary or feedback inputs is edited, it is necessary to also edit the input lists where these names appear. The following directives are available.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change, B,I,N</td>
<td>Change the $i^{th}$ input (count from 1) of block B to N.</td>
</tr>
<tr>
<td>PRVTSTD,N,ADD</td>
<td>These three directives add to the list of previously tested blocks, primary inputs and feedback inputs. If ADD is omitted, lists are initialized. The new information follows the directive, max 10 names per record, separated by commas (no internal spaces). Records need not be full (there may be less than 10 names/record).</td>
</tr>
<tr>
<td>INPUTS,N,ADD</td>
<td></td>
</tr>
<tr>
<td>FEEDBACK,N,ADD</td>
<td></td>
</tr>
</tbody>
</table>

The addition of reset vectors is accomplished through the control program (see Section 3).

4.3 Organizing

The organizer proper is executed through the control statement—Organize. The logic must be read and editing completed before this directive is given. The organizer first checks to see that all inputs to all blocks have been defined. If not, the undefined inputs are listed on the printer (or the first output medium if reassigned) and control is
switched to typewriter. It then attempts to organize the logic. If successful, the organized logic is written on the last output tape (normally 3). If undefined feedback loops are found, these are printed on the printer. In the first case, the typeout "organized" occurs and in the second case, the typeout "fb loops" occurs. Control is not switched in either case.

The output tape is not rewound and an end of file is always written after the organized logic.

5. Use of the Diagnosis Program

5.1 General

The diagnosis program (name SEQANALZ) cannot be used effectively without a familiarity with Seshu and Freeman [1]. In broad outlines, SEQANALZ operates as follows. The organized logic is read from magnetic tape through the control program. Resets may be either on the magnetic tape or read from the control medium or both. At this point one may either simulate a given sequence or ask the program DIAGNOSE to produce a testing procedure. In the latter case, any desired options may also be entered from the control medium. It is also possible to transfer back and forth between the two programs (SIMULATE and DIAGNOSE).

5.2 Diagnose

The subroutine OPTION, which reads the options may be entered in one of two ways. The control statement

OPTION
executes the subroutine. Alternatively, if diagnosis is desired, may also write

\text{DIAGNOSE, 1}

In this case the control program DIAGNOSE first executes the subroutine OPTION. In either case, the options are given in the control records immediately following the control directive. Options are written as strings with commas as separators, and with no internal spaces. Each record may contain 1 to 15 options. Termination of the list of options is by the option ENDOPTN or a period (separated from the last option by a comma). Currently the following options are available:

\begin{itemize}
  \item \text{DIAG} \quad \text{Complete diagnosis}
  \item \text{DETECT} \quad \text{Failure detection only. (If given in a subset without machine 1, the top branch is carried as far as possible.)}
  \item \text{WANDER, N} \quad \text{Wander N steps}
  \item \text{SHORTSIM} \quad \text{Simulate shorted diodes}
  \item \text{NOSHORT} \quad \text{Do not simulate shorted diodes}
  \item \text{SUBSET, I,J} \quad \text{Start from subset (i,j)}
  \item \text{MAINTAIN} \quad \text{Maintain previous partition. Do not initialize.}
  \item \text{INFO} \quad \text{Use information gain as figure of merit}
  \item \text{CHECKOUT} \quad \text{Use the checkout gain as figure of merit}
  \item \text{INVTPFAIL} \quad \text{Simulate inverter shorts}
  \item \text{NOINVT} \quad \text{Do not simulate inverter failures}
\end{itemize}
SAFETY Insert safety extension
NOSAFETY Do not insert safety extension
STRAINIT Initialize strategy indices
STRACONT Continue strategy indices from previous problem.
MONITOR Cut out to next problem if errors occur
ENGINEER Do not cut out. Come to typewriter if errors occur.
INITOPTN Initialize options to DIAG, WANDER, 10, SHORTSIM, SUBSET, 0, MONITOR.

If errors are discovered on option records, INITOPTN is performed, record in error is printed and control medium is moved to ENDOPTN or period.

The diagnosis program normally operates as follows. The logic must be read in and all resets read in before DIAGNOSE is executed. The logic simulator is compiled (including failures as specified by options) and the dictionary of failures is printed. The dictionary is self-explanatory except for macro elements. The inputs to the second level of a macro are referred to as

\[ \text{XYZ INT N OPEN (OR SHORT)} \]

where XYZ is the name of the macro block and N is the number of first level block counting from 0.

Following the dictionary, the orders of the inputs, outputs and feedback lines are printed to permit correlation with later statements.
The test procedure follows in the same format and notation as in Seshu and Freeman [1] except that the input set is identified by its index only (the list is not printed). The other differences are slight. If MSGEFLAG is set (positive) the strategy that is currently in use is also printed.

The strategy indices may be dumped at the end of a run by the control statement

STRADUMP

and read in the next time by the control statement

STRAREAD

The normal medium for both statements is paper tape. To get any other medium, the decimal equivalent of BCD code should be used as argument, eg.,

STRADUMP, 39

prints the indices on the line printer. If strategy indices are read in, the strategy continue flag is always set.

For a general discussion of the operation of this program, see companion report [2].

5.3 Simulate

SIMULATE is the name of the program for simulating a sequence of inputs. This program is used, for example, to complete the detection of failures by manually-generated tests in cases where the program DIAGNOSE is unable to do so. The program is executable by its name. It reads all directives from the control medium. The following directives are recognized.
Subset, i,j  
Choose subset (i,j) for simulation. If i=0, initialization occurs (compile, print dictionary).

Sequence  
Simulate the following inputs, following the top branch at partitions. Inputs follow, packed left justified octal, one per record. Any directive terminates simulation.

Reset, k  
Reset current subset to $k^{th}$ reset in list of resets (count from 1).

Interpolt  
Interpolate between inputs that differ in more than one bit.

Nointer  
Do not interpolate

Endsiml  
Exit

The program initializes to interpolation if the circuit is not synchronous. If errors are found in directives, the directive in error is typed out and a one record switch to the typewriter occurs. To disregard directive, type

SKIP

However, if the error occurs in the SEQUENCE branch, the message INPUT FORMAT ERROR is printed on the normal output media followed by the record, and it continues in this mode. If any machines misbehave, the list of misbehaving machines is printed out, the typeout MISBEHAVIOR occurs and a one record switch is made to the typewriter. In the MONITOR mode, these switches are replaced by cut-out to next problem.
5.4 Precaution

It must be noted that several programs may be reading from the control medium. It is, therefore, necessary to know which program one is talking to. If OPTION, SIMULATE or OCTALOAD is entered, it is necessary to exit properly from the program before one can talk again to the control program.

Options remain where set unless a new OPTION entry is made. They do not get initialized for each problem. On the other hand, the interpolation flag is initialized at each entry onto SIMULATE.

PART II

PROGRAMMER'S MANUAL
6. Structure and Conventions

6.1 General

The purpose of this part of the manual is to describe the programs in sufficient detail to permit other programmers to modify the program or add to it. Since the programs are rather large, it is essential to agree to certain broad rules and conventions in order to minimize confusions.

6.2 Structure

One of the structural rules which has been carefully obeyed is that of subroutine isolation. The programs have been separated into subroutines, of small size where possible. The subroutines communicate only through calling sequence parameters carried in A and Q and through data and flags in COMMON. All subroutines save and restore any index registers used. No subroutine saves or restores A or Q. Interrupt is not used. There are several exceptions to the "small size" rule which are unavoidable. The exceptions are:

- READCNTR (about 2000 instructions)
- COMPILE (about 400 instructions)
- READINPT (about 400 instructions)
- HUFFMAN (about 800 instructions)
- IMPLY (about 1500 instructions)

The names above are the only proper entry points into these subroutines. Within a subroutine index registers may get used in calling sequences or otherwise. Hence, if one enters (from the outside) a subroutine inside IMPLY for example, anything may happen.
All programs have been written in such a way that one can take over control at the typewriter whenever one chooses. This facility is provided since the program is basically an experimental program. Thus, every time an input is to be read from the control medium, the programs look to see what the control medium is.

Each subroutine is documented in the listing. The calling sequence, result of executing the subroutine and any error action, is stated in front of each subroutine. The listing should, therefore, be consulted before any changes or additions are made.

6.3 Conventions

Each of the major subroutines in this program was checked carefully by using detailed dump routines (which are not in the final assembled version). However, no program of 10,000 instructions can be considered "debugged." For this reason, a special subroutine called "GOOF" has been provided. At numerous places in the program, checks are made. If any check fails, a return jump to GOOF is executed. For example, if one is searching for an ordered pair index, the sequence goes as

```
EQS 5 INDXPAIR
RTJ GOOF
...
```

Similarly if one is getting ready to set up the number of items-1 as the address of an ISK instruction:
The subroutine GOOF types out "Program goof at location ...." and gives a console scoop. This subroutine has been a major debugging aid and all future additions should keep up this practice.

All subroutines, except for the compiled simulator, treat computed data in common as lists arranged sequentially. Everytime a subroutine is entered, it looks up the name of the list in common and sets up all required addresses. The names are all in one standard format. The upper address is the number of items and the lower address is the start of the list. Notice that each item on the list may occupy part of a word or several words. The compiled simulator refers to the lists PRIMINPT, FEEDINPT, BLOKVALU, FEEDOUTP, PRIMOUTP absolutely. This exception is made for speed.

This programming convention was made to permit one to follow a sequence of events on a trial basis and restore the original state later. Not all of the lists need to be this flexible. However, a uniform convention is always useful.

Almost all of the lists start from the first location. There are a few exceptions made for indexing convenience in subroutines that have to be executed many times. The exceptions are;
all in the diagnosis program. The organizer also uses the data as lists, but this feature is not essential. It is done mainly to preserve conventions.

There are many communication flags in COMMON. The flags are identified by their names. Each flag is a full word. Each flag is set when positive and not set when negative. For example, if MSGEFLAG is positive, special messages are to be printed (e.g., current strategy).

A list of all the tables and flags follows:

List of tables in ORGANIZE:

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOKLIST</td>
<td>List of blocks. First four characters give name of block. Last four give type.</td>
</tr>
<tr>
<td>BLOKINPT</td>
<td>List of input lists. Corresponds 1:1 with BLOKLIST. Entry i is the name of the input list of ith block.</td>
</tr>
<tr>
<td>PINLIST</td>
<td>List of all inputs to all blocks. Indexed by BLOKINPT. Each input occupies a word. The lowest six bits are coded as octal 20 for block, 40 for primary and 00 for feedback.</td>
</tr>
<tr>
<td>PRINLIST</td>
<td>List of primary input names, one per word.</td>
</tr>
<tr>
<td>PROTLIST</td>
<td>List of primary output names, one per word.</td>
</tr>
<tr>
<td>FEEDLIST</td>
<td>List of feedback last blocks, one per word.</td>
</tr>
<tr>
<td>LEVLBLOK</td>
<td>List of blocks arranged by levels.</td>
</tr>
<tr>
<td>LEVLINDX</td>
<td>Set of indices to LEVLBLOK (not a list).</td>
</tr>
</tbody>
</table>
RESFEED List of reset feedback states (one/word, left justified).

INPTLIST List of reset primary inputs (one or two words per input vector, left justified).

PREVTSTD List of previously tested blocks.

SYMBTABL List of symbols (for writing).

SYMEQUIV Memory address of symbols.

List of flags in ORGANIZE:

CUTFLAG Flag for cut-out on error (skip to next logic).

EDRSFLG Resets edited.

EDINFLG Inputs edited.

EDPOFLG Primary outputs edited.

EDPRFLG Primary inputs edited.

EDFDFLG Feedbacks edited.

List of tables in SEQANALZ:

BLOKLIST
BLOKINPT
PINLIST
PRINLIST
PROTLIST
FEEDLIST
RESFEED
INPTLIST
PREVTSTD

These tables are identical to those in ORGANIZE of the same name.
RACEOSCL
OSCILLTN
CRITRACE

Lists of malfunctioning machines, set up by HUFFMAN. Machine numbers in lower address, one per word. CRITRACE also contains an index to all final states, which are stored in CRITFEED. First six bits of word in CRITRACE give number of final states. Upper address is start of list. If more than 47 branches, upper 24 bits are 0.

CRITFEED

List of final states for machines with critical race, one per word left justified. Indexed by CRITRACE.

DICTION

Dictionary of failures. Head of list empty.

DICTION + K is entry for machine K. Set up by COMPILE. Upper half of word is block name and lower half is failure. Lower half is 0 or 1 for output at 0 or 1. For input failures, lower half is number of failing input (count from 0). Lower operation code is 46 (BCD code for 0) for open circuit failures and 22 (BCD code for S) for short circuit failure. For macro internal failures, lower index is 1 (otherwise 0).

FEEDINPT

List of feedback input values.

PRIMINPT

List of primary input values.

PRIMOUTP

List of primary output values.

FEEDOUTP

List of feedback output values.

BLOKVALU

List of block output values (memory shared).

FAILLIST

Current list of failures to be simulated (maximum 48).
OUTVECTR List of distinct output vectors, primary or feedback (one or two words per vector, left justified) in present subset of machines. Set up by PARTMACH.

OUTMACH List of machine numbers in present subset, ordered according to output vectors. 12 bits per machine (4 to a word). List indexed by OUTINDX. Set up by PARTMACH (not a list).

OUTINDX List of indices to OUTMACH. Each entry is the name of the list of machines with this output. Corresponds 1:1 with OUTVECTR. Set up by PARTMACH.

LISTMACH List of all machines arranged according to present partition, one per word in lower address. Indexed by INDXMACH. Set up by UPPARTIT.

INDXMACH List of indices to LISTMACH. Corresponds 1:1 with INDXPAIR. Set up by UPPARTIT.

INDXPAIR List of ordered pair indices of all subsets of machines. Set up by UPPARTIT.

LASTINPT List of last inputs, one per subset (one to two words per input). Corresponds 1:1 with INDXPAIR. Set up by UPPARTIT.

LASTFEED List of last feedback states, one per machine in natural order. Head empty. Set up by UPFEED.
LASTPAIR
List of last used second indices (in ordered pair indices) arranged by first index. Head empty.

PSNDMCHN
PSLSTMCH
PSLSTNPT
PSNPAIR
Psuedo list names which replace INDXMACH, LISTMACH, LASTINPT and INDXPAIR in the try mode. Set up by SAVEDATA, restored by RSTRDATA.

SAVEFEED
List of last f.b. states saved in try mode. Order as in present subset.

APRIORI
List of aprori probabilities of failure.
If not available as input, set up by SETPROB as identical. Floating point numbers in order corresponding to machine numbers.

MERITABL
List of figures of merit for all next inputs.
Floating point numbers. Set up by BESTNEXT.
Entry is $-\phi$ if input is invalid.

PATH
List of bits to change to go from last input is desired input. Bit number counting from left (count from zero) is stored in lower address.
Set up by FINDPATH.

DECISION
List of decisions made in combinational test generation. $K^{th}$ entry gives decision made for $K^{th}$ block. Zero if no decision. Otherwise upper address gives no. of input fixed. Lower half is block name (for debugging convenience).
Set up by IMPDECSN in IMPLY.
TRACE Backtrace from given output or forward trace from given block to given output for combinational test generation. Set up by BACKTRACE.

TRACE1 Temporary list used by BACKTRACE. (Shared memory).

STRALIST Strategies in current order (not a list). Lower address is address of strategy subroutine.

Communication and Option Flags in SEQANALZ:

CUTFLAG Cut-out on error flag.

MSGEFLAG Special messages flag.

QUITFLAG Detect only (no diagnosis).

INVTFAIL Simulate inverter shorts.

SHORTFLG Simulate shorted inputs.

INITFLAG Initialize (compile, print dictionary, Subset 0).

SAFEFLAG Insert safety extension.

COMBFLAG Combinational circuit.

SYNCHFLG Synchronous circuit.

The flag descriptions apply when the location is positive.

7. The Control Program

7.1 General

The control program is essentially the same in ORGANIZE and SEQANALZ. It operates in a relatively simple fashion. The program initialization consists of setting up control medium as paper tape, input medium as 2 and output medium as printer, clearing location 0 and starting the clock. It then enters the loop point RDCNTROL. The control
statements are read through the FORTRAN input routine INPUTCDI in the format (10A8) into the locations CNTRBFFR. Checks are performed for +, and -, in column 1. The subroutine PARAMETR is asked to unpack the parameters into PARSTR. The first location is compared with the list of control directives RDCNSTAT. If on the list the address of action is taken from the switch RDCNSWT1, an unconditional transfer occurs. If not on this list, the list of symbolically executable programs CNTRPROG is searched. If on this list, the list PARSTR of parameters is moved down 1, and we jump to XEQ action. If the given directive is not to be found, the message "control statement error" followed by the statement in error is typed out, and control transferred to RDCNTYPR.

The detailed actions taken for the different directives are discussed in succeeding sections.

For external programs, the entry point is READCNTR. Manual entry is at RDCNDRVR. Local subroutines may enter (by ordinary jump) at RDNCUT or RDCNTER. This point is an error return. Control goes normally to the typewriter. If however, the flag MONITOR is set, the control tape is moved to the next "LOGIC" statement (or END).

7.2 Problem Oriented Directives

We first discuss the special directives which are oriented toward the problem at hand. These are

LOGIC, RESETS, CUTOUT

The directive LOGIC is given when one wishes to read a circuit description from magnetic tape. The parameters are interpreted differently in
ORGANIZE and SEQANALZ. The action for the directive is as follows. We initialize the logic tape (location RDCNLTP) to the input medium. In SEQANALZ the comment flag is initialized to no comments. The parameter list is searched to see if these two are given. If they are given, they replace the initialized values.

We now enter the subroutine FINDSORC to locate the logic description on tape. The calling sequence parameters for FINDSORC are:

A: Name of logic desired
Q: Input tape number (1, 2, 3, or 4)

The subroutine FINDSORC operates as follows. The given tape is rewound. We read the first record and compare columns 20-27 with given name. If different, we skip a file and try again. If the first record is either a FINIS in columns 10-14 or an end of file, we exit with A negative. If the given name is found, the input tape is backspaced one record in ORGANIZE so that IDENT may be read again. In SEQANALZ, columns 41-48 of the IDENT record are checked for SYNCHRON. If yes, SYNCHFLG is set positive, otherwise negative. At exit, accumulator is positive if logic is found.

If logic was not found by FINDSORC, READCNTR goes to error return RDCNTRER. Otherwise, we enter the subroutine READINPT to read the input logic description.

The calling sequence parameters for READINPT are:

A: Comment flag
Q: Input tape number

The comment flag is not interpreted in ORGANIZE. In SEQANALZ this flag is positive to print comments and negative to omit them.
The subroutine READINPT operates differently in SEQANALZ and ORGANIZE. The general action is, however, the same in both programs. The main differences between the two programs are the following. In ORGANIZE the input records are stored in memory and a symbol table is made up. A symbol is by definition any entry in columns 1-8 except for an asterisk in column 1. In SEQANALZ, the block input lists are checked to see whether they have already occurred (level check). Special action is taken in SEQANALZ for macros. Otherwise the two routines are identical.

The heading group of records are handled differently from the block information records. As each record is read, columns 1-8 are checked for INPUTS, OUTPUTS, FEEDBK, RESETS, PRVTSTD, NUMITER, asterisk in column 1.

If NUMITER occurs, columns 9-16 are simply converted to binary and stored in location NUMITER. If the directive RESETS occurs, the number of resets (in cols 9-16) is converted to binary for end test. The following records are read for reset vectors. The reset f.b. states and inputs are converted to binary and stored in RESFEED and INPTLIST, respectively. This conversion is done by a special subroutine READCNBN which converts two BCD octal words to one binary word. The other directives are all handled in the same way. The number of items is stored in the upper address of the appropriate list name. The succeeding input records are read, and the items stored in the lists.

The switch from heading group to block information occurs whenever we encounter an entry in columns 1-8 that is not recognized as a header. READINPT first checks the block type to see whether it is a
macro block or an ordinary block. The two types are handled by different parts of the program. In either case, the block name and type are packed into one word (upper half, name; lower half, type) and stored in the list BLOKLIST. Also the name of the input list (no. of inputs/starting address in PINLIST) is stored in the list BLOKINPT.

Each input to the block is first checked by the subroutine READCKIN. This subroutine first checks the list of primary inputs. Then it checks to see whether the last non-blank character is an asterisk. If yes, the name is checked against FEEDLIST. In SEQANALZ, if it is neither primary nor feedback, the input is checked against BLOKLIST. (This is the level check.) This subroutine codes the lowest six bits as octal 00 if feedback, octal 40 if primary, octal 20 if neither.

If the block is ordinary, no special action takes place. Each input, after checking by READCKIN is simply stored in PINLIST.

If the block is a macro, the program READINPT in ORGANIZE handles it as follows. The list of inputs to the first level blocks is packed into one word (6 bits per block left justified) and stored in the first location of the input list. The given number of inputs is read from succeeding records (separators are simply skipped) and stored in PINLIST.

In SEQANALZ, the macros are handled in a more complicated way. They are effectively converted into ordinary blocks. The first level blocks are given pseudo-names. These are binary numbers starting from 00000001 (octal). The numbers are sequential. There is, thus, no confusion with the other names which are BCD. Also, the first six bits
specify that the block is the first level of a macro. This fact is used in both the compiler (for failure injection) and the subroutine that prints the dictionary (DICTONRY). Otherwise, all subroutines treat the block as if it were an ordinary block. Special macro procedures are thus avoided.

After the given number of inputs have been read, the subroutine READINPT in SEQANALZ simply skips until it finds another record with columns 1-8 non blank or an end of file is encountered. In ORGANIZE the records are stored in memory, but no other action takes place.

The directive RESETS, N (with or without a, ADD) may only be given after the logic has been read. In ORGANIZE, primary and feedback input editing must also be completed first. The following action takes place when this directive is encountered.

The list of resets is initialized or maintained depending upon whether ADD is absent or present in the directive. Next, from the number of primary and feedback inputs, the format statement for reading the resets is made up. The resets are now read through INPUTCDI.

The CUT-OUT directive is acted on as follows. The argument given is stored in the location CNTRMEDM (control medium). We jump into location RDCNNXLG which simply moves the control medium until it finds a LOGIC or END directive.

7.3 Tape Handling

The tape handling directives operate very simply. The required actions are directly executed except for SKIPRECD and SKIPFILE. For these two directives, we enter appropriate system subroutines.
(in ILLRES). For all others, the given tape number is substituted into the select instruction, and the desired action takes place. Note that all selections are "read" selections except for the directive WEF (write end of file).

Special subroutines handle REWIND, P (eject printer page) and REWIND, F (put out leader on paper tape).

7.4 Medium Assignment

The directives CONTROL, Z and INPUT, Z are handled in similar ways. The argument is checked against the list of permissible media. If the given medium is permissible, it is stored in the appropriate location (CNTRMEDM or INMEDIUM) and we return to the common point RDCNTROL. The action of OUTPUT, X, Y, Z,... is similar except that there may be several output media given. The number of these is counted (from 0) and stored in OUTNMBR.

7.5 Execution

The directive XEQ is acted on as follows. First we look to see whether the given first argument is on the list of symbolically executable programs (CNTRPROG). If yes, the entry point address is taken from the table RDCNSWT2 and substituted in the return jump instruction. The upper address in this table is zero for normal programs and nonzero if the other arguments in the directive should not be interpreted by the control program. If the given name is not on the list, it is taken as octal and we enter BCOOCT to convert to binary. Any other arguments are taken to be decimal integers (unless they are not to be interpreted).
The first argument is loaded into A and the second in Q before the return jump is performed.

7.6 Dump, Store

The directives DUMP and TDUMP are mainly for debugging purposes. TDUMP is executed simply by asking OUTPUTCD to dump the contents of the given location. DUMP is executed in a more complicated way. The arguments are first converted into the proper calling sequence parameter for SNAPSHOT (in ILLRES). For each output medium given, we have to check whether it is the printer or a tape since the parameter for SNAPSHOT has to be set up properly. Then we enter SNAPSHOT for the actual dump.

STORE is also primarily intended for debugging convenience. The action is very simple. The first argument is checked against the list RDCNSTSY. If not on list, it is taken as octal. The second argument is taken as a decimal integer.

7.7 Miscellaneous

END simply causes an exit.

CALL, Z. This directive is executed as follows. Z is converted from BCD to flex code. We store this name in PROG (in Block 2 of Illinois system) and enter FINDANY (also Block 2).

TIME, P. The subroutine TIME is entered with the output medium in A.

TIME, φ. Store zero in location φ.

TIME, S. Clock is stopped.

PAUSE. We enter OUTPUTCD to type Pause (j2) on the typewriter and enter KEYTEST to wait for j2.
SAVE, Z. We enter the driver routine DRVSAVE which drives
PROGSAVE, to save all cores and output (if available on tape).
TRANSFER, S. Transfer to given location.

8. Editing and Output From Organizer

8.1 General Format

In the logic organizer, in addition to storing the input
information in the format described in the last chapter, the subroutine
READINPT stores the input information as 120 character images and makes
up a symbol table. Each input record is stored in memory as 15 words.
If columns 1–8 are non-blank, the first word is stored in SYMBTBL and
the location in memory where this record is stored is entered into
SYMEQUIV. An asterisk in col. 1 is not considered as a symbol. The
input records themselves are stored in FULLBFFR. In the initialization
of the program READINPT, a number of flags are cleared to indicate
that no editing has been done yet. These are EDRSFLG (resets), EDINFLG
(input lists), EDPOFLG (outputs) and EDFDFLG (feedback). Clear as usual
means –0. If any editing is done the appropriate flag is set (positive).
These flags are tested before writing out the organized output.

8.2 Operation of Subroutine EDITSORC

The subroutine EDITSORC basically operates as a control program.
As each edit directive is read, the first parameter of the record is
checked against the list of available directives and a switch is used to
jump to appropriate action.
The directive CHANGE is executed as follows. We first locate the block in BLOKLIST and in the symbol table (for correcting the source). The input list for the block is found and corrected if necessary for macro. We take the given input and check whether it is primary or feedback or ordinary. It is coded appropriately and stored in PINLIST. The source is also corrected appropriately. The following error checks are made: no new name, input number too high, feedback input and name not on feedback list. In all cases a return jump is made to EDITERRI, where a one record switch to the typewriter occurs.

The directives FEEDBACK, INPUTS, OUTPUTS and PRVTSTD are all handled in the same way. We first check to see if the directive ADD has been given. The index register for storing the new names is set up appropriately, depending on whether ADD has been given or not. Next, we set the appropriate flag to say that the list has been edited and jump to common action. The common action is simply to add to or make up PRINLIST, PROTList or PREVTSTD and store the new number of items. The source is undisturbed in this case. The subroutine WRITEOUT checks the flags to decide whether the source is to be copied or a new source made up.

8.3 Organizer Output

The output from the organizer is written by the subroutine WRITEOUT according to the following conventions. It is always written on the last tape in the output medium list. The ident record and any following comments are written first. The comment records (asterisk in column 1) are always assumed to be attached to the immediately preceding records and not the following records. Next, we check to see whether
primary inputs, outputs and feedback information were given in the source. If not, we make up the appropriate list. If yes, we check the appropriate flag to see if any editing was done. If yes, we again make up the source from the memory list. Otherwise the source is copied onto the last output tape given. Resets and previously tested blocks are also treated similarly. Notice that if any editing was done on any of these lists, the source program is made up. Thus if any comments were originally included, they are now lost.

After the heading records are completed, we go into the point WRTBLOKS to write the information for each block. The blocks are picked up by levels using the index LEVLINDEX. The level of the block is converted to decimal and stored in words 14, 15 of the record containing the block name. After all the levels are finished, an end of file is written and routine exits. Notice that comments are not lost in this case.

The actual handling of the output is done by the subroutine WRTCPCOM.

9. Organizer

9.1 General Structure

The basic control program in the system ORGANIZE is the subroutine of the same name. This is the program for arranging the input information into levels. It is mainly designed for asynchronous sequential circuits.

This routine assumes that the input has been read into memory by the subroutine READINPT in the proper format and all required editing has been done. Immediately on entry, the organizer jumps into the
subroutine CKINPUT. The subroutine CKINPUT has the function of checking that all inputs to all blocks have been defined. The subroutine CKINPUT operates in a very simple fashion. It takes the list of inputs to each block and checks the inputs one by one. It first checks to see whether the input has been coded as primary or feedback. If yes, no further checking is done. If not, the input name is checked against the list of blocks. If any input is not found, the subroutine prints the message on the first output medium given. As usual exit parameter is a positive accumulator if all inputs have been defined and a negative accumulator otherwise.

The organizer does not attempt any organization if some inputs have not been defined. The organizer has two main functions. In the normal mode of operation, if all the feedback loops have been defined by the programmer, it simply organizes the logic into levels and writes the output tape. The second main function (or mode) is to locate undefined feedback loops. If any feedback loop has not been defined, the organizer locates the loop and prints the list of blocks which are in the feedback loop. It does not attempt to define which part of the loop is to be considered as feedback.

9.2 Normal Organization

The normal organization of a logic onto levels is a relatively simple process. The level of a block is defined as follows. Primary inputs and feedback inputs are considered to be of level zero. A block is of level n if all the inputs to the block are of level n-1 or less and at least one input is of level n-1.
The procedure for organization is therefore as follows. We initialize the level to 1. We initialize the list LEVLBLOK to contain no entries. The level index LEVLINDEX is the list of names of the block at each level. For each level the loop point is ORGLVLP. We first set a flag ORGFLAG2 to indicate that no new blocks have been picked up. We pick up the name of each block and check whether it is already on the list LEVLBLOK. If it is already there, no action is necessary. If not, we get the list of inputs to the block and check them one by one. If it is primary or feedback input, we pass to the next input. If neither, the input is checked against the list LEVLBLOK. If it is there, the block is added to the list LEVLBLOK at the current level. Notice that we check only the part of LEVLBLOK corresponding to the previous or lower level. We also reset the flag ORGFLAG2 to indicate that a new block has been picked up. If any input to the block is not primary or feedback and is not on the list of blocks at lower levels, we leave it alone and go to the next block. When all blocks have thus been tested, we check the flag ORGFLAG2 to see if any new blocks have been added to the list LEVLBLOK in the last pass. If yes, we step the level count, initialize the list of the new level and go back to the loop point.

If no new blocks were picked up in a pass, organization has been carried as far as possible. We now check (at the point ORGFDCK) to see if all the blocks have been included in the level structure. If any block is not found in the list LEVLBLOK, then a feedback loop is present in the logic. If all blocks have been levelled, we enter WRITEOUT to write out the organized logic as described in the previous chapter.
The typeout "organized" occurs and we exit. In the first case (where a feedback loop exists) we jump to ORGFDFD to locate the loop as discussed in the next section.

9.3 Location of Feedback Loops

The location of an undefined feedback loop is performed by a backtrace followed by forward trace technique. We start with the block that was not found on the LEVLBLOK list. This block name is stored in a new list. We now go into a loop at ORGFDSL. For each block on the list, we find the list of inputs. Each input is checked to see whether it is primary or feedback. If yes, we go to the next input. If no, we first check to see if it is the block we started with. If not, we add to the list of blocks. If yes, the backtrace process ends.

When the backtrace process ends, we have localized the feedback loop but have not identified it. For identifying the blocks in the loop, we forwardtrace from the first block, using only the blocks in the backtrace list. To do this, we start a new list with the name of the last block. We go to the previous block in the backtrace list, find its input list and check whether the last block is an input to the previous. If not, we go back up the backtrace one step and try again. The forwardtrace again ends when the loop closes. At this point the feedback loop has been identified. We print the list of blocks that are in the forwardtrace, and add them to the list LEVLBLOK. We set a flag to say that feedback loops have been found and go back to ORGFDCF for more feedback loops, if any.
Notice that the process of adding all the blocks in feedback loop to the list LEVLBLOK may result in some of the feedback loops not being located in the first pass. If two feedback loops feed the same first block and there is no fan-out at this point, the second feedback loop may not be identified. However, if we now identify the first feedback loop (by editing) and go back into the organizer, it will find the other. If the organizer is successful, all feedback loops have been identified. If the organizer is unsuccessful, it will find feedback loops, but may not find all of them in the first pass.

When editing is done to identify a new feedback loop (Feedback, 1, add), it is necessary to also edit the input list for the blocks that are fed by the feedback loop.

10. Structure of SEQANALZ

The essential structure of the program in SEQANALZ is shown in Figure 1. The description of the logic circuit to be processed is read by the subroutine READINPT discussed earlier. The compiler now compiles a logic simulator. This simulator is simply a sequence of instructions that compute the Boolean functions corresponding to the feedback output variables and the primary output variables. The compiler also inserts additional instructions for the purpose of simulating failures as specified by the list of options. As the figure shows, there are two basic driver routines that can drive the entire program. The external communication as before is through the control program READCNTR. The driver routine simulate is used for the purpose of patching manually generated tests to program-generated tests. This driver routine simply
simulates any given sequence of inputs. After each input has been applied, the top branch of the resulting partition is followed. All of the bookkeeping is common to the two driver routines. The diagnosis driver is the complicated driver and this is the program that generates a test procedure for the given logic circuit. At any given time, the diagnosis driver may be connected to one of the four strategy subroutines. The program is written in such a way that one can add additional strategy subroutines, if desired. The four strategy subroutines are described in Chapter 14. Each of these subroutines attempts to find a next input or a sequence of next inputs in such a way as to refine the present partitioning of the machines. Each strategy subroutine is empowered to take action if it is able to refine the partition. If it is unable to refine the partition, it returns control to the calling program. The calling program may then go to the next strategy subroutine. These strategy subroutines drive a bookkeeping subroutine called SIMULSET. The basic function of the subroutine SIMULSET is to cut up the present set of machines into sets of 48 machines for purposes of simulation, set up the failure injection words for each of the sets of 48 machines, and enter both the compiled simulator and the subroutines for producing the partition of the machines based on either the primary output or the feedback output as desired. There are other bookkeeping subroutines which permit one to calculate the information gain of each test or to select the best next input on the basis of either information gain or failure detection. These subroutines are discussed in Chapter 12. The combinational strategy, the return strategy, as well as the straight simulation
driver may need to interpolate between the present input and the next desired input in case these two inputs differ by more than one bit. This interpolation is done by means of a subroutine FINDPATH described in Chapter 13. If the subroutine FINDPATH is successful in finding a path between the present input and the next desired input without any malfunction, then it stores the path in a list PATH. One may now use the subroutine straight sequence simulation (STRTSEQ) to simulate this path rapidly.

11. The Compiler and the Huffman Analyzer

11.1 General

This chapter describes the programs which form the core of the logic simulation. The compiler is the subroutine which converts the input information as read by READINPT into a subroutine for computing the Boolean functions including failure simulation. The Huffman analyzer is the program which closes the feedback loops on the simulator and checks for various malfunctions.

The two programs described in this chapter also share the feature of "tightness." The compiler has been written tightly to conserve space. The Huffman analysis is a rather complicated process since we deal with 48 machines in normal simulation and with 48 branches of one machine in race analysis and these are shared. In both programs subroutines make use of index registers, A and Q as required in the calling sequences. Therefore the only proper entry points from other programs are COMPILER and HUFFMAN, respectively.
11.2 Compiler

The compiler performs two main functions. It makes up the compiled simulator and it makes up the dictionary of failures. The two functions are naturally related. The basic process of compilation is very simple since we assume that the logic has already been arranged in levels (this fact is checked while the input is read by READINPT). The complications arise mainly from the fact that we have to perform many tasks repeatedly for each type of logical element and in the interest of conserving space, we extract as many of these functions as possible into subroutines. The other complication of a minor nature is that there are two instructions per word in the 1604. This difficulty is overcome by using a left/right flag, asking the subroutines to compile the instructions one per word in a temporary store and packing them in the main part of the compiler. There has to be an entry point and exit from the compiled simulator, of course. There is also the minor complication of having to index the failure injection words. This difficulty is overcome by using an index register (number 6) for this purpose in the individual subroutines. The main part of the compiler replaces the indexed instructions by absolute addresses (for speed). We also introduce the further convention that the compiler will suppress (while packing the instructions) the instruction ZRO 0. This instruction is used for a filler which becomes necessary because of certain conventions in the local subroutines.

The basic operation of the main part of the compiler is as follows. We initialize the dictionary index to 2. We initialize the compiled simulator to the entry point (SLJ 0 0) and initialize the left/
right flag to right. We now go into a loop for each block. For each block, we find the block type and use a local switch (COMPLTYP) to find the address of the appropriate subroutine. We also check to see whether the block is in the list of previously tested blocks (PRVTSTD). We now enter the appropriate subroutine with the block name in A and mode parameter in Q. Q is positive to omit failure injection and negative to include failures. When we exit from the subroutine back to the main compiler, the instructions for the block are in INSTSTOR (memory shared) and the number of instructions is in A. These instructions are now packed according to the previous conventions. Namely, we suppress ZRO 0, execute indexing instructions instead of packing them and pack the rest, two to a word in SIMULATR. When all the blocks are finished, we store the number of machines in NUMMACH as well as the upper address of DICTON, pack an exit instruction in the simulator and we are all done.

The local subroutines used by all the subroutines for the different block types are briefly described below.

**COMPBLOK:** Entered first by each block subroutine. Stores block name in COMPBLNM, locates the block in BLOKLIST leaving index in B5, and stores input list name (entry of BLOKINPT) in COMPINLS. Also stores the block type in COMPBLTP. Enter with block name in A left justified.

**COMPADDR:** To find the absolute address of an input. Enter with name of input in A with coded last character. The input name is stored in COMPINPT. The proper absolute address (in PRIMINPT, FEEDINPT, or BLOKVALU depending on the coding of the lowest character) is in A at exit.
COMPSTA: This subroutine stores the STA instructions at the end of the compiled sequence of instructions. For all blocks, this subroutine stores an STA BLOKVALU instruction. In addition it stores instructions for storing A in FEEDOUTP and PRIMOUTP if the block is a feedback last block or primary output block, respectively. It may be both, of course. B4 is assumed to index INSTSTOR and B5 is assumed to be the index of the block in BLOKLIST.

COMPOUTS: This subroutine sets up the output failures and shorted input failures (if required) and calls on COMPSTA to set up the STA orders for each block. It also checks to see whether the block is a macro first block, in which case no output failures are set up. The dictionary entries for these failures are also made up and stored in DICTON.

COMPMACK: This subroutine checks to see whether the block (name in A left justified at entry) is part of a macro block. At exit Q is positive if macro and negative otherwise. If the block is the second level of a macro, lower half of A is cleared and the lower index field is set to 1. Otherwise lower half of A is simply cleared.

COMPLOOP: This is the basic loop subroutine which inserts the sequences of instructions required for each input to a given block. This subroutine is to be entered after the LDA instruction for the first input has been set up. Enter with input list name in A and instruction list name in Q. The third instruction in the list of instructions is assumed to be the LDA instruction for each input. (The filler ZRO 0 is used to make it the third if necessary). Dictionary entries for open input
failures are made up if COMPFLAG is positive. Dictionary entry is made up for the first input as well. The outputs are not set up by this program. B4 is assumed to index INSTSTOR.

The individual subroutines for the various block types are very similar. Each subroutine sets up COMPFLAG, enters COMPBLOK, sets up the first LDA instruction and enters COMPLOOP with the proper list names. It then enters COMPOUTS if output failure is to be included and COMPSTA otherwise. There is also considerable sharing between subroutines of similar types.

11.3 Huffman Analyzer

This is one of the most complicated subroutines in the program SEQANALZ, which unfortunately cannot be broken up into smaller subroutines. The calling sequence parameter for the subroutine is a positive accumulator to cut out on malfunction. Before the subroutine is entered, the primary and feedback inputs to the simulator must be set up. At exit accumulator is positive if malfunction occurs (convention backwards from other programs) and negative if no malfunction occurs. Malfunctioning machines are listed in the tables CRITRACE, RACEOSCL, OSCILLTN as described in Chapter 10.

The initialization of HUFFMAN consists of initializing the malfunction flag WELLBEHV and race flag HFRACFLG to -0, and iteration count to 0. The major loop point is HFSIMLTR. At this point we set a flag HFCYCFLG to -0 to indicate whether the circuit has settled down. We now enter the compiled simulator. The first check is to see whether
the circuit is combinational or synchronous. If combinational, no action is necessary. If synchronous, we simply move the feedback output values to FEEDINPT and exit. If asynchronous, complicated checking is necessary. This process is described below.

We first form the exclusive or of the present state and memory excitation and store in temporary store. This array is transposed to bring the vector for each machine into one word (originally in the simulator each word is one feedback variable for 48 machines). We now count the bits in each word of the transposed matrix. If it is zero, the machine is stable. If it is one, recycling is necessary, and if it is more than one, we have a race. If exactly one variable is unstable, we check to see if the machine has already misbehaved. If yes, we skip it. If no, we set the recycle flag HFCYCFLG positive close the loop and move to the next machine. If a race occurs, we go to the separate branch at HFRACE discussed later. After all machines have been tested, we check the recycle flag. If recycling is necessary, we check the iteration count. If it does not exceed NUMITER we go back to the simulator. If it has, we set the malfunction flag. If cut out has been specified, we exit. If not, we store the numbers of the malfunctioning machines in OSCILLTN or RACEOSCL depending on whether we are in normal or race analysis.

The race analysis is the worst part of the subroutine. We have to first check to see whether we are already in race analysis. In this case, we split the branches further to include the new race. If we are not, we have to switch the mode of operation from normal to race analysis. We change the list of failures FAILLIST to consist of only
one machine (by saving the original and moving the link as usual).
We reset the simulator to no failures and call on SETFAIL to set up all
48 bits for the one machine we are analyzing. We have to set a flag to
say that no race branch has settled yet, set up an initial list of number
of branches and go back to the loop point. The feedback state for the
machine must also be set up and the cycle count saved.

When the first race branch settles, we store the final state
and clear the "first settle" flag. When subsequent branches settle, the
final state is compared with the branch that settled first. If same, no
action is necessary. Otherwise, the critical race must be noted.

A separate part of the program HFRACNRML restores the simulator
to normal after the race has been processed.

Modifications to this routine are not advisable.

12. Bookkeeping and Arithmetic Subroutines

12.1 Conversion Routines

In this section we describe the various conversion routines
that are used for input-output. In several places in the program we can­
ot use the standard FORTRAN subroutines for conversion since we do not
know what type of data is coming in until we read the first parameter in
the record, which is the directive.

OCTBCD: This subroutine converts an octal integer to the bcd
equivalent. Enter with octal integer in A. BCD equivalent in A at exit,
right justified and with spaces to the left. Range \(-3^7\) to \(3^8\). If an
error is found (number out of range) accumulator has \(-0\) at exit.
LFOCTBCD: Driver for OCTBCD which left justifies the integer rotating the spaces to the right. Same entry and exit as OCTBCD.

BCDOCT: Converts BCD integer to octal equivalent. Enter with bcd number in A left justified and with spaces to the right (number terminated by space). Octal equivalent in A at exit. A contains -0 at exit if illegal characters found.

BCOOCCT: Converts an integer in bcd octal format to octal. Enter with bcd octal number in A left justified. Octal number in A at exit. If illegal characters found, A contains -0 at exit.

OCTBCO: Octal to bcd octal integer conversion routine. Enter with octal number in A. BCD octal number in AQ at exit. No leading zero suppression.

12.2 Arithmetic Subroutines

This section contains brief descriptions of the various arithmetic subroutines in SEQANALZ.

COUNTBIT: Counts the number of bits in a word. Enter with number in Q. Count of bits in A at exit. Uses table look up with 6 bit bytes. Takes about 270 microsecs.

SPRAYBIT: Used for setting up primary inputs and in the case of race analysis, feedback inputs as well. Enter with list name of store to spray in A and packed word in Q. Highest order bit goes into first word. This subroutine stores 48 bits in the word identical to the one given.

PEELBITS: Subroutine to select a column from a binary matrix in packed form. Matrix assumed to be stored one row per word. Enter
with matrix address in lower address of A, number of rows as upper address of A and desired column number in Q. Count sign bit as column 0 and lowest order bit as 47. Maximum size of matrix 96 x 48. Desired column in AQ at exit, left justified and with zeros packed to right.

TRANSPOS: To transpose a packed binary matrix. Matrix assumed to have been stored one or more words per row, 48 bits to a word, left justified. Enter with parameter list address in accumulator. Parameters: Address of matrix, number of rows, number of columns, address of transposed matrix. The last address must be different from the address of the original matrix. Exits with -0 in A if calling sequence errors occur.

FLIPBIT: To complement selected bit of a vector in packed form. Enter with address of first word of vector in A and index of bit to flip in Q. Count sign bit of first word as zero.

FLOATINT: To convert an integer to floating point format. Enter with integer in A. Floating point form in A at exit.

LOGBASE2: To compute the logarithm of a floating point number to base 2. Enter with floating point number in A. Floating point logarithm in A at exit. Uses a standard series for the logarithm to base e, truncated to 9 terms.

12.3 Bookkeeping and Auxiliary

Various updating subroutines and simulator control subroutines are described in this section.

SETFAIL: This subroutine sets up the desired failures in the simulator. It is assumed that the location FAILSIMUL+k corresponds to
machine number \( k \). Calling sequence: set up the list \( \text{FAILLIST} \) of machine numbers to simulate. Normally, this subroutine sets up words in \( \text{FAILSIML} \) which have one bit each, for each of the entries in \( \text{FAILLIST} \). Bit position left to right corresponds to order in \( \text{FAILLIST} \). If, however, \( \text{FAILLIST} \) contains only one entry, and the accumulator has -0 at entry into this subroutine, all 48 bits are set up for the given failure. This feature is for race analysis. Logic assumed to be free of failures at entry.

**RSTFAIL**: Clears failures in \( \text{FAILSIML} \). (Simply stores zero in all locations)

**PARTMACH**: This is the main partitioning subrouting for partitioning the present set of machines into equivalence classes. This subroutine must be entered after each set of 48 machines have been simulated in order to set up the partitions. To initialize the output from this subroutine (for each subset of machines) set upper address of location \( \text{OUTVECTR} \) to zero. Enter the subroutine with accumulator positive to partition on the basis of primary outputs and negative to partition on the basis of feedback outputs. Outputs from this program are \( \text{OUTVECTR} \), \( \text{OUTMACH} \) and \( \text{OUTINDX} \). \( \text{OUTVECTR} \) lists the distinct output vectors that occur (one or two words per entry depending on number). As usual, the upper address of location \( \text{OUTVECTR} \) gives the number of distinct output vectors and lower address is the start of the list of vectors. \( \text{OUTMACH} \) is the set of machine numbers, arranged according to the partition obtained. Machine numbers are packed four to a word( 12 bits each). This list of machines is indexed by \( \text{OUTINDX} \). Entries in \( \text{OUTINDX} \) correspond one to one with
entries in OUTVECTR. That is, each entry in OUTINDX is the name of the list of machines with this output vector. In making up these lists, PARTMACH disregards malfunctioning machines (they are not on the list).

Note: PARTMACH uses TEMPRRY for storing the transpose of FEEDOUTP or PRIMOUTP as the case may be. Thus, the block values after simulation are destroyed.

UPFEED: This subroutine updates the feedback states. That is, it transfers the feedback states from FEEDINPT to LASTFEED. Of course, the entries are transposed and only the entries of LASTFEED corresponding to machines in FAILLIST are affected.

Note: UPFEED also uses TEMPRRY to store the transpose of FEEDINPT. Also note that UPFEED must be entered after each set of 48 machines have been simulated (if the updating of feedback states is desired).

UPPARTIT: This subroutine does the major bookkeeping of machine numbers and partitions and produces the normal output of the program. The following lists are updated by this program:

LISTMACH, INDXMACH, LASTPAIR and LASTINPT.

Calling sequence: PARTMACH must be entered to set up the partitions. This subroutine uses the output of PARTMACH as input data. All machines are assumed to be well behaved and all inputs are assumed to be identical. Ordered pair index of subset of machines simulated should be in PRSTINDX (index of input set). If this index is zero, the index of the input set is not printed.

Note: In the subroutines PARTMACH and UPPARTIT, there is some shuffling of storage to maintain the sequential ordering of lists. In both
programs we need to insert items in a list. We do this by actually moving the lower part of the list and creating space in memory for insertion in the proper place.

SETFEED: This subroutine sets up the feedback states in the simulator (FEEDINPT). Set up FAILLIST before entry. The feedback states are transferred from LASTFEED to TEMPRRY and then transposed into FEEDINPT.

SAVEDATA: This is the critical subroutine which permits one to follow any course of action on a trial basis and then recover the original state. Calling sequence: Store ordered pair index of subset to save in PRSTINDX. Note that only one subset can be saved. Note also that this subroutine does not suppress output. The following data are saved and can be restored by RSTRDATA:

- LISTMACH, INDXPAIR, LASTINPT, LASTPAIR, LASTFEED, INDXMACH, CYCLCNT, ANYCNT.

All the lists, with the exception of LASTFEED are saved by saving the list name and replacing it by the name of a pseudo-list. In the case of LASTFEED, however, we save the actual contents in a local list. The present index is saved and is replaced by \((k+1,1)\) where \(k\) is the last level encountered so far.

Note: Only one level of saving is possible because of memory limitations. An alternation flag SVDATFLG is used to denote whether one is in the SAVE mode or the normal mode. This flag is set negative by SAVEDATA and positive by RSTRDATA. Both subroutines check the flag and indicate (by calling on GOOF) if two successive saves or restores have been called for.
RSTRDATA: Restores the data that has been previously saved by SAVEDATA. Checks that the data has been saved, before restoring.

SETPROB: Sets up identical entries in the a priori probability of failure list APRIORI.

12.4 Figure of Merit

There are currently only two figures of merit available. These are the information gain and check-out figures of merit respectively. The addresses of the subroutines which compute the figure of merit are listed in MERITSWT in common storage. The figure of merit, specified by the programmer by the option card, is stored as an index to this switch in CRITERON. (Currently this location has 0 for information gain and 1 for check-out.)

INFOGAIN is the subroutine which computes the information gain from a test. The input data to this subroutine is the partition set up by PARTMACH and the a priori probabilities listed in APRIORI. Note that this subroutine actually consults the list APRIORI and does not assume that all failures are equally probable. There are no calling sequence parameters. At exit the information gain for the test is in the accumulator as a floating point number. Q contains 0 if there are no malfunctioning machines (upper addresses of RACIOSCL, OSCILLTN and CRITRACE are all zero) and Q contains -0 otherwise. The gain is computed even if there are malfunctioning machines—uses the standard definition of information gain. For a binary partition, if P and Q are the total probabilities of the machines in the two subsets, normalized so that \( P + Q = 1 \), the gain is
An $n$-ary partition is equivalent to a sequence of $n-1$ binary partitions.

The figure of merit for the checkout criterion is calculated by the subroutine CHECKOUT. Normally this subroutine is used in the failure detection mode, to minimize the number of tests required to detect failures. However, the subroutine is set up so that it can also be used in branches not containing the good machine if desired. The figure of merit calculated as:

$$-(P \log_2 P + Q \log_2 Q)$$

If the first machine is the good machine, this figure of merit is larger for greater failure detection. There are no calling sequence parameters. The subroutine uses the output of PARTMACH. The figure of merit is in the accumulator at exit as a floating point number.

13. Intermediate Level Routines

13.1 Combinational Test Generation

One of the four major strategies used by the program SEQANALZ is the so-called combinational strategy. The basic tool used by the strategy subroutine is the combinational test generation program called COMBTEST. This program is described in the present section.

COMBTEST assumes that the input has been read into memory in the standard format. It does not have any input-output. The purpose of the program is to produce a test for a given machine. It treats feedback
inputs and primary inputs the same way, thus effectively opening up the feedback loops and considering the circuit to be combinational. Calling sequence: enter with machine number in A and mode flag in Q. The mode parameter occupies the lowest three bits of Q. The lowest bit is 0 for the first test (for the given machine) and 1 for next. Bit 46 is a 1 if the test must be observable immediately at the primary output. Bit 45 is 1 if we are to minimize the dependence of the test on feedback inputs. The outputs from the program are stored in TESTINPT, TESTFEED and TESTOUTP. The location TESTOUTP contains the name of the test output. The last character is an * if the output is a feedback output. TESTINPT (4 words) and TESTFEED (2 words) contain the test inputs, primary and feedback respectively. Each variable is given two bits, and is coded as:

00 - Free (x)
10 - Must be 0
11 - Must be 1

If no tests (or no more tests) are found, A contains -0 at exit.

The main part of COMBTEST operates as follows. We first interpret the mode parameter. If bit 45 is a 1 (minimize feedback dependence), we generate all tests, one by one. We count the number of free variables in FEEDINPT, and choose the one which has a maximum. Otherwise, we check for first test. If first, we initialize. Then we try primary outputs first before trying feedback outputs. If no feedback outputs are specified by mode parameter, the second half is skipped. The part of the program that actually drives the test generation subroutine is a local subroutine called COMBTDRV. The local subroutine that packs the test information that is in the simulator into the proper coded form
for TESTINPT and TESTFEED is called COMBPACK. For each output, the
driver COMBTDVR first calls on the subroutine FWDSET to set up the
test for the block. The calling sequence for FWDSET is: machine number
in A and output name in Q. The subroutine FWDSET first calls on the
subroutine BACKTRCE to check whether the given block (in which the failure
is located) feeds the given output. If not, it exits with A negative.
The block corresponding to the given machine is obtained by entering the
subroutine LOCMACHN which looks up the dictionary and gives the block
name in A and input list in Q. The calling sequence parameters for
BACKTRCE are: output name in A and block name in Q. Back tracing is,
of course, an elementary operation. The subroutine BACKTRCE may be
asked to backtrace to a given block or backtrace to inputs. For the latter
mode we enter with zero in Q. When it is asked to backtrace to inputs,
the subroutine simply stores the back trace in the list TRACE. When
asked to trace to a given block, it checks to see whether the given block
occurs in the backtrace. If it does not, it exits with A negative. If
it occurs, the subroutine BACKTRCE stores the forwardtrace starting from
the given block and going to the given output in the list TRACE.

FWDSET next checks the type of failure to take appropriate
action. The type of failure is also identified by LOCMACHN as follows.
The dictionary entry is stored in CURRFAIL. For input failure, name of
failing input is stored in FAILINPT. Location FAILSHRT is set positive
for short-circuited inputs. These two locations (FAILINPT and FAILSHRT)
are meaningless if output failure. FWDSET takes different actions depend­
ing on the type of failure. In all cases we first clear all locations in
the simulator (FEEDINPT, PRIMINPT, BLOKVALU) to the value "FREEVALU" by entering the subroutine DONTCARE. We now set up the inputs for the block appropriately for each failure. The value of the output of the block is made "TESTVALU". The zeros and ones set up by FWDSET are "TESTZERO" and "TESTONE", to distinguish them from the values set up by the subroutine IMPLY. These values are the corresponding bcd equivalents in memory. Thus, the block output entry in BLOCVALU is octal. 2365222325614324. The subroutine FWDSET next sets up the outputs of all blocks in the forward-trace from the failing block to the output as TESTVALU.

Once the forwardtrace has been set up, we go into the loop point in the combinational test driver. Here we call on the subroutine IMPLY to set up the values in the simulator by backtracing from the output to the inputs, setting up all values as required for the test.

The subroutine IMPLY is rather involved. It is 1000 instructions long and cannot be entered anywhere except at IMPLY. The basic procedure in IMPLY is the following. We take each block in TRACE (which now must contain the backtrace from the desired output to primary and feedback inputs) find its type and jump to the appropriate subroutine to set up the inputs to the block. Each block subroutine checks the value required for the block. If free, no action is taken. Otherwise the inputs have to be set up properly. Two situations occur. In some cases (e.g., AND output zero) not all inputs need be fixed. In this case, a decision must be made on which input to fix. The second situation is where we are unable to set up the inputs properly because of conflicts. In the second case, we must try to see whether it is possible to modify some decisions
to resolve the conflicts. For this purpose, as well as for the purpose of getting the "next" test, we make up a decision list. This list has one entry for each block and corresponds to BLOKLIST. The name of the list is DECISION. The entry is a zero if no decision was made for the block. Otherwise, the lower half word is block name and upper address is the index of the input which was fixed. (The lower half was intended mainly for debugging convenience). When the first test for a failure is called for, we clear the list of decisions. Otherwise it is modified as required. When we are successful in setting up the inputs as required, we first save the contents of PRIMINPT, FEEDINPT and BLOKVALU by coding the values suitably and packing them, in the stores PRIMSAVE, FEEDSAVE and BLOKSAVE respectively, before exiting. Thus, when we are called upon to move the decision forward (generate the next test), we can restore the previous values. The detailed operation of the subroutine IMPLY is too complicated for description.

13.2 Simulation Control and Reset Control

This section contains brief outlines of the subroutines for simulating the behavior of a subset of machines both for an input and for a reset.

SIMULSET: This is the subroutine that divides up the set of machines in the present subset into sets of 48 machines and drives the partitioning and updating routines and the Huffman analyzer. Calling sequence: store ordered pair index of present subset of machines in PRSTINDEX (may be zero), and desired primary input vector in PRSTINPT. Enter with mode parameter in A. The mode parameter uses the three lowest order
bits in A as follows:

Lowest order bit: 1 to try input only, 0 to use.

Second lowest bit: 1 to complete simulation of all machines and 0 to cut out of simulation on misbehavior.

Third lowest bit: 1 for partition based on feedback outputs and 0 for partition based on primary outputs.

If the input is to be used, feedback states are updated and so are the partitions. Also in the use mode, the normal program output occurs.

When the input is to be tried and cut out on misbehavior has been specified, simulation is stopped as soon as a misbehavior occurs and the program exits with -0 in the accumulator. Note that if cut out has not been specified there is no indication of misbehavior (use COMPSIML for this purpose). Accumulator normally has 0 at exit. All the initializations necessary for bookkeeping are done by this program.

COMPSIML: This is a special driver for SIMULSET which simulates a complete set of machines and still gives indication of misbehavior in accumulator at exit (-0 if error, 0 otherwise). Calling sequence as for SIMULSET. This subroutine drives SIMULSET with the mode parameter 3 (primary output partition, complete simulation, try only).

SETRESET: Subroutine to store the desired feedback state in LASTFEED for a given subset of machines. Calling sequence: store ordered pair index of subset in PRSTINDX and enter with desired feedback state in A. No output from this program.

RESTSBST: To reset a subset of machines, both feedback and primary input, and put out the reset message on the output media. The reset
is also simulated. Enter with ordered pair index in PRSTINDX (may be zero) and index of reset in A. PRSTINDX contains the ordered pair index of top branch at exit.

13.3 Subroutines for Strategy Programs

This section describes the subroutines which are used by the Strategy Routines described in the next chapter.

FINDPATH: This is a basic subroutine to find a path from the present input to a desired input. Calling sequence: store ordered pair index of subset in PRSTINDX and desired input in PRSTINPT. Zero is not permissible in PRSTINDX. Enter with accumulator nonzero to find a path only, and zero to use the path if found. At exit, accumulator is positive if path found and negative otherwise. Also at exit, Q contains the number of machines left in the top branch at the end. If there was no gain, Q contains minus zero. The following conventions are used in finding the path. Only one bit is changed at a time and no bit is changed more than once. Thus, only paths within the subcube defined by entry in LASTINPT and the desired input (in PRSTINPT) are tried. The path is a simple path (no loops). Otherwise all paths are tried. The first valid one, if found, is given as output. The path is stored in the list PATH as sequence of bits to change. The entry in the list PATH is the index of the bit to be changed, counting the sign bit of the first word as 0.

Note: This program uses SAVEDATA to store intermediate steps. If MSGEFLAG is set (positive), a message is put on the output medium whenever a path cannot be found.
STRTSEQ: This is a straight sequence simulation program which is normally used to simulate a sequence of inputs after a path has been found. Calling sequence: store ordered pair index of subset in PRSTINDX (may not be zero). Enter with name of bit list in A and try flag in Q. The bit list format is the same as the one described for PATH in the write up of FINDPATH. The try flag is negative to try each input before use. Otherwise it is directly used. If try is specified and some input is invalid, routine exits with A negative and index of invalid input in Q. If try flag is zero, the given input list is used in its entirety. If try flag is a positive number, the simulation of the list is carried until the number of machines left is equal to the given number and no further. At exit A contains total gain from all inputs and Q contains the number of steps used.

BESTNEXT: This is the subroutine which finds the best next input. This subroutine tries each input which differs from the entry in LASTINPT in one bit, and computes the figure of merit according to the present criterion. The figures of merit for all the next inputs are stored in the list MERITABL using -0 as the figure of merit of an invalid input. Calling sequence: store ordered pair index of subset in PRSTINDX and enter with 0 in A to partition primary outputs and 1 in A to partition feedback outputs. At exit accumulator contains the figure of merit of the best next input and Q contains the index of the bit changed. The best input is not used. If no next input is valid, accumulator contains -0 at exit.
CYCLWAND: Cyclic wandering subroutine. The subroutine operates as follows. It first enters BESTNEXT to see if any input partitions feedback states. If one does, it is used. If none of them do, it changes bits cyclically until it finds a valid input. This input is used. Note that this subroutine actually updates data. The input used is in PRSTINPT at exit. Calling sequence: store ordered pair index of subset in PRSTINDX. At exit index of top branch is in PRSTINDX, A is positive and Q contains index of bit changed. If no next input is valid, A contains -0 at exit.

ANYNEXT: This is a subroutine to find any reasonable next input and use it. This subroutine changes input bits cyclically until it finds one for which the figure of merit exceeds amount specified in INFOMIN. If one is found, it is used. Calling sequence: store ordered pair index of subset in PRSTINDX. If no next input is satisfactory, routine exits with -0 in A. If satisfactory input is found and used, index of top branch resulting is left in PRSTINDX and Q contains index of bit changed.

Note: Currently no strategy uses ANYNEXT. It is an unused subroutine.

BSTRESET: This is a subroutine to find the best reset for a given subset of machines. Resets are investigated to a given depth. Calling sequence: store ordered pair index of subset in PRSTINDX and enter with depth to investigate in A. The best reset is by definition the one that minimizes the number of machines left in the top branch after the given number of steps. At exit, A contains the index of the best
reset (in the list of resets) and Q contains the number of machines left in the top branch. If no reset is permissible, A contains -0. If no gain results from any reset, Q contains -0.

Note: No output results from this program. It is advisory. SAVEDATA is used by this program. All data saved and restored. PRSTINDEX may be zero at entry.

14. Strategy Routines

14.1 Best Next or Return to Previous Good Input (BESTRETN)

This is the strategy subroutine that is generally most efficient (in terms of computing time) and generally the most useful. The calling sequence consists of storing the index of the present subset of machines in PRSTINDEX. At exit accumulator is positive if the program is successful and negative otherwise.

The program operates as follows. We first check the name of the logic in LOGCIDNT to see if it is the same as the logic at previous entry. If not, we set the number of vectors in the local list (BESTRENPN) to zero. If yes, we leave the number alone. Next, we enter BESTNEXT to find the best next input. If some next input is good (accumulator nonzero at return from BESTNEXT), we save the index of the bit and update the strategy indices (see Sec. 14.5 for these). Next we look up MERITABL to see if several next inputs resulted in a positive gain. If yes, we save the unused ones in the local list of previous good inputs BESTREPI after checking for duplicates. Now we use the best next input and exit.
If, on the other hand, no next input resulted a positive gain, we look up the list of previously good inputs. If this list is empty, we exit. If not, we try each of these in turn (using FINDPATH) to see if any of them is still a useful input. The first useful one is used (using STRTSEQ), if yes. Otherwise we exit.

14.2 Wandering Strategy (TRYUWAND)

This is the strategy subroutine that tries to see if wandering is a useful strategy. The operation of the program is as follows. We first save all data and suppress output. We enter BESTNEXT to see if any next input is useful. If not, we enter CYCLWAND to take a step. We repeat these two operations until the number of steps in WANDER is exhausted, or no next input is valid or we find a good input from BESTNEXT. At each step we also save the bit changed in the list PATH. If we do find a good input at some point, we restore the simulator (RSTRDATA) and call on STRTSEQ to simulate the sequence of inputs saved in PATH. We update the strategy index and exit with A positive. If, on the other hand, we do not find a good input, we restore the simulator and exit with A negative.

14.3 Combinational Strategy (TRYUCOMB)

The combinational strategy subroutine tries to see whether tests derived on a combinational basis can be used on the sequential circuit. For each machine in the current subset of machines TRYUCOMB tries the following. It first checks to see if it is the good machine. If so, we go to the next machine. Otherwise, we call on COMBTEST to generate the first test for the machine. Now we call on COMBMERG to merge
the test input with the present input (i.e., replace all the x's in the test by the present values). Next, we call on FINDPATH to find a path from the present input to the test. If a path is found and results in a partition, we use it by calling on STRTSEQ. We also update the strategy indices. If either no path is found (malfunction in some machine) or no partition occurs (since feedback states are not controllable), we call on COMBTEST to produce the next test for this machine and try again. When all tests are exhausted, we move to the next machine. When all machines are exhausted, we give up and exit with A negative.

TRYUCOMB is currently not being used as efficiently as possible. That is, we do not get as much benefit from the combinational test generation procedure as we might. If it is possible to reset the current subset of machines on a trial basis to a reset that is close to the test value, more partitioning might be obtainable. However, it is not possible to use savedata, because of the fact that FINDPATH needs to use SAVEDATA and only one level of saving is possible.

14.4 Reset Strategy (RESETSTR)

This is a rather simple subroutine. It simply calls on BSTRESET to produce the best reset. If no reset is good, we quit. Otherwise, we call on RESTSBST to simulate the reset and put out the reset message. Then we update the strategy indices and exit. Reset is counted as 10 steps for the strategy indices.

14.5 Strategy Indices

The main strategy index that is used as the figure of merit of the different strategy subroutines is "average gain per step". To compute
this number we need to keep track of total number of steps and total gain. This computation is done inside each strategy routine. The strategy index handling subroutines are STRATSET, STRATDMP, STRATRD, STRATINT.

**STRATSET:** This is the subroutine which sorts the strategies in decreasing order of figure of merit. It looks up the strategy indices in STRATDX and sets up the list of strategy names (address of corresponding subroutines) in the list STRALIST for use by the control program. This is a simple $n^2$ sort routine.

**STRATDMP:** The subroutine STRATDMP is used for dumping the strategy indices for later reading by the subroutine STRATRD, for continuing the indices from one run to the next. This subroutine can be executed by the control directive STRADUMP. If A is nonzero at entry, the contents of A are taken to be the output medium desired. Otherwise, the contents of location STRAOMED (currently paper tape) are used as the desired output medium.

**STRATRD:** This subroutine which can be executed by the directive STRAREAD is used for reading the data previously dumped by STRATDMP. The input medium is taken from A if nonzero, otherwise from STRAOMED.

Note: The format statement used for dumping and reading strategies uses the fact that there are 4 strategies. If this number is changed, the format statement needs to be changed also. The data that is dumped is: average gain, total number of steps, total gain, and number of times used.

**STRATINT:** The subroutine STRATINT is used to initialize the list STRALIST of strategies for use by the control programs. If the strategy
continue flag STRACONT is not set, this subroutine transfers the standard list from STRATLS1 to the list STRALIST and clears all indices to zero. Otherwise, it enters STRATSET to sort the strategies, for continuing the strategy indices.

15. Control Programs for Simulation and Diagnosis

15.5 Simulation Driver

The control program SIMULATE which can be executed by name, is a control program for simulating a given input sequence. It reads all control statements and input data from the control medium. The logic and all necessary resets must be read before this program is executed. Basically, it operates like any other control program. It reads the directive given and uses a switch to jump to the appropriate open subroutine to execute the given directive. The actions that take place for the different directives are described below.

NQINTER: Stores -0 in SMLTFLAG to signify no interpolation required.

INTRPOLT: Stores 0 in SMLTFLAG for interpolation.

SUBSET,I,J: The parameters I and J are first converted to octal. We check to see if I = 0. If yes, the initializing subroutine INITLZN is executed and we exit. If no, we check to see if the index I,J is in the list of ordered pair indices. If it is, we exit. If not, we enter the error branch.

RESET,I: We check if the present index is zero. If yes, we try the reset input after setting up the feedback states. If well-behaved use it and we exit. If not, we print out the list of malfunctioning
machines. If the present index is not zero, we first save the data and try the reset as before. If valid we use it and exit. Otherwise, we print all malfunctioning machines. The trial simulation is done using the driver COMPSIML in order to get complete information on malfunctioning machines. In case of misbehavior at reset, we type out MISBEHAVIOR and go to the error branch.

SEQUENCE: This part of the program directly reads the control information for simulating the given inputs. After each record is read, we check to see whether the first parameter on the record is another directive. If yes, we exit back to the main part to obey the directive. If not, we convert the given vectors to binary. Next, we check the interpolation flag SMLTFLAG to see if interpolation is required. If not, we go to COMPSIML to try the given input. If the machines are well-behaved, we use the input and proceed to next record. If not well-behaved, we put out the message that there is misbehavior at this input and go back to the loop for the next record. If interpolation is required, we call on FINDPATH to find the path, and if found, to use the path. Then we go back to the loop.

ENDSIML: Causes simple exit.

Note: The error branch of the control program SIMULATE does a one record switch of control to the typewriter unless CUTFLAG is set, in which case, it goes to RDCNTYPR to cut out and go to next problem. We check the record from the typewriter to see if it says "skip". If yes, we go back to the main part of SIMULATE to read the next control statement. Otherwise, we go to the point where the record has already been read, to interpret it.
15.2 Control Subroutine STRTLINE

This is the main diagnosis control routine. It is normally driven by the control program DIAGNOSE (described in Sec. 15.3). The subroutine STRTLINE starts from any given ordered pair index and carries out diagnosis, following the top branch only, as far as possible.

If at entry into this subroutine the contents of PRSTINDX are zero, the following special action takes place. We first initialize (Compile, Dictionary, Strategy initialization). We then call on BSTRESET to find the best reset. If no reset is permissible, we put out this message and exit with A negative. Otherwise, we call on RESTSBST to simulate the best reset and go into the loop point.

The loop for each subset follows. We first call on STRATSET to sort the strategy subroutines in decreasing order of figure of merit and store their addresses in STRALIST. We then call on INDIST1 to make a simple-minded check to see if the present subset of machines is indistinguishable. If yes, we check MSGEFLAG to see if this message is required. If yes, we put it out. We then exit with the number of machines left in the accumulator. If INDIST1 gives a negative answer, we call on the strategy subroutines in the order in which they appear in STRALIST to take the next step. If one of them is successful, we go back to the loop point. If none of them is successful, we exit with the number of machines left in the accumulator.

15.3 The Diagnosis Driver

The diagnosis driver, which can be executed by the name DIAGNOSE is a very simple program. It first checks to see if INITFLAG is
set or present index is zero. If not, we locate PRSTINDX in the list of ordered pair indices. Then we go into the loop point. We call on STRTLINE to carry the subset as far as possible. Now we check SAFEFLAG to see if a safety extension is required. If not, we check to see if QUITFLAG is set. If yes, (detection only) we exit. Otherwise, we check INDXPAIR to see if there are any indices below the present one. If yes, we go back to the loop point with the next index. If not, we exit. The safety extention branch operates as follows. We take the machine number and check to see whether it is the good machine. If yes, we go to wander five steps. If the good machine is not in the subset, we find a combinational test for the machine. Next, we ask COMBMERG to merge the test and the present input. We check to see if this input is different from the present one. If yes, we call on FINDPATH to find and use a path to this input. If no, we try the next machine. If no more machines left in the subset, we wander five steps.
APPENDIX

Two small examples are worked out in detail in this appendix to illustrate the use of the two programs. The logic circuits of the examples are given in Figures 2 and 3 respectively. The flow tables of the normal circuits (without failures) are given below.

Example 1

<table>
<thead>
<tr>
<th>$y$</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
</tr>
<tr>
<td>1</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
<td>0,0</td>
</tr>
</tbody>
</table>
Example 2

(Array transposed for clarity)

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00,00</td>
<td>00,01</td>
<td>00,00</td>
<td>00,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>00,01</td>
<td>00,01</td>
<td>00,01</td>
<td>00,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>10,01</td>
<td>10,01</td>
<td>10,01</td>
<td>10,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>11,11</td>
<td>11,11</td>
<td>11,11</td>
<td>11,11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>00,00</td>
<td>00,01</td>
<td>00,00</td>
<td>00,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>00,01</td>
<td>00,01</td>
<td>00,01</td>
<td>00,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>00,01</td>
<td>00,01</td>
<td>00,01</td>
<td>00,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>00,01</td>
<td>00,01</td>
<td>00,01</td>
<td>00,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>00,00</td>
<td>00,01</td>
<td>00,00</td>
<td>00,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>00,01</td>
<td>00,01</td>
<td>00,01</td>
<td>00,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>10,01</td>
<td>10,01</td>
<td>10,01</td>
<td>10,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>11,11</td>
<td>11,11</td>
<td>11,11</td>
<td>11,11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>00,00</td>
<td>00,01</td>
<td>10,00</td>
<td>10,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>00,01</td>
<td>00,01</td>
<td>11,11</td>
<td>11,11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>00,01</td>
<td>00,01</td>
<td>10,01</td>
<td>10,01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>00,01</td>
<td>00,01</td>
<td>11,11</td>
<td>11,11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PRINTOUT OF INPUT TAPE

IDENT  EXAMPLE1
INPUTS 3  x1  x2  x3
OUTPUTS 1  R1
FEEDBK 1  R1
A1  AND  2  x1  x2
A2  AND  2  A1  R1*
R1  OR  2  A2  x3

ORGANIZED LOGIC

INPUTS 3  x1  x2  x3
OUTPUTS 1  R1
FEEDBK 1  R1
RESETS 1  x1  x2  x3
LEVEL 1
A1  AND  2  x1  x2
A2  AND  2  A1  R1*
LEVEL 2
LEVEL 3
A2  OR  2  A2  x3
LEVEL 3

CONTROL TAPE FOR EXAMPLE 1

LOGIC:EXAMPLE1
RESETS:1
.0
ORGANIZE
RECMD:3
LOAD
TRANSFER:101
INPUT:3
LOGIC:EXAMPLE1
DIAGNOSE:1
NOSAFETY:CHECKOUT:ENDOPTN
END
# Dictionary of Machine Failures for Logic Circuit Example 1

<table>
<thead>
<tr>
<th>MACH NOS</th>
<th>FAILURE</th>
<th>MACH NOS</th>
<th>FAILURE</th>
<th>MACH NOS</th>
<th>FAILURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GOOD MACHINE</td>
<td>2</td>
<td>x1 TO A1 OPEN</td>
<td>3</td>
<td>x2 TO A1 OPEN</td>
</tr>
<tr>
<td>5</td>
<td>A1 OUTPUT 1</td>
<td>6</td>
<td>A1 TO A2 OPEN</td>
<td>7</td>
<td>R1 TO A2 OPEN</td>
</tr>
<tr>
<td>9</td>
<td>A2 OUTPUT 1</td>
<td>10</td>
<td>A2 TO R1 OPEN</td>
<td>11</td>
<td>x3 TO R1 OPEN</td>
</tr>
</tbody>
</table>

**INPUT ORDER X1 X2 X3**

**OUTPUT ORDER R1**

**FB ORDER R1**

---

**Feedback Input:** 0

**Primary Input:** 000

**Index of Output Set:** (1, 1)

**Output Vector:** 0

**Machine Numbers:** 1 2 3 4 5 6 7 8 10 11 12

**Current Strategy:** BESTNEXT

**Index of Input Set:** (1, 1)

**Primary Input:** 001

**Index of Output Set:** (2, 1)

**Output Vector:** 1

**Machine Numbers:** 1 2 3 4 5 6 7 8 10

**Current Strategy:** BESTNEXT

**Index of Input Set:** (2, 1)

**Primary Input:** 000

**Index of Output Set:** (3, 1)

**Output Vector:** 0

**Machine Numbers:** 1 2 3 4 5 6 7 8 10

**Current Strategy:** BESTNEXT

**Index of Input Set:** (3, 1)

**Primary Input:** 010

**Index of Output Set:** (4, 1)

**Output Vector:** 0

**Machine Numbers:** 1 2 3 4 5 6 7 8 10

**Current Strategy:** WANDER

**Index of Input Set:** (4, 1)

**Primary Input:** 110

**Index of Output Set:** (5, 1)

**Output Vector:** 0

**Machine Numbers:** 1 2 3 4 5 6 8 10

**Current Strategy:** WANDER

**Index of Input Set:** (5, 1)

**Primary Input:** 100

**Index of Output Set:** (6, 1)

**Output Vector:** 0

**Machine Numbers:** 1 2 3 4 8 10
<table>
<thead>
<tr>
<th>INDEX OF INPUT SET</th>
<th>PRIMARY INPUT</th>
<th>INDEX OF OUTPUT SET</th>
<th>OUTPUT VECTOR</th>
<th>MACHINE NUMBERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6, 1)</td>
<td>101</td>
<td>(7, 1)</td>
<td>1 2 3 4 8 10</td>
<td></td>
</tr>
<tr>
<td>(7, 1)</td>
<td>100</td>
<td>(8, 1)</td>
<td>0 2 4 8 10</td>
<td></td>
</tr>
<tr>
<td>(8, 2)</td>
<td>1</td>
<td>(9, 1)</td>
<td>1 2 4 8 10</td>
<td></td>
</tr>
<tr>
<td>(11, 1)</td>
<td>111</td>
<td>(10, 1)</td>
<td>1 2 4 8 10</td>
<td></td>
</tr>
<tr>
<td>(10, 1)</td>
<td>110</td>
<td>(11, 1)</td>
<td>1 2</td>
<td></td>
</tr>
<tr>
<td>(11, 2)</td>
<td>1</td>
<td>(12, 1)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(12, 2)</td>
<td>0</td>
<td>(12, 2)</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Current strategy:** WANDER

**Current strategy:** BESTNEXT
**PRINTOUT OF INPUT TAPE**

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>IDENT</th>
<th>EXAMPLE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>x1</td>
<td>x2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUTS</th>
<th>IDENT</th>
<th>EXAMPLE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>b3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FEEDBK</th>
<th>IDENT</th>
<th>EXAMPLE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>b1</td>
<td>b3</td>
</tr>
</tbody>
</table>

|          |          |          |
| B1       | aor     | 3,2      |
| aor      | x1      |          |
| B2       | not     | x2       |
|         |         | 1        |
| B3       | and     | x3       |
|         |         | b4       |
| B4       | or      | x4       |
| or       | 3       | b4       |

**ORGANIZED LOGIC**

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>IDENT</th>
<th>EXAMPLE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>x1</td>
<td>x2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUTS</th>
<th>IDENT</th>
<th>EXAMPLE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>b3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FEEDBK</th>
<th>IDENT</th>
<th>EXAMPLE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>b1</td>
<td>b3</td>
</tr>
</tbody>
</table>

|          |          |          |
| B2       | not     | x2       |
|         |         | 1        |
| B4       | or      | x4       |
| or       | 3       | b4       |
| B1       | aor     | 3,2      |
| aor      | x2      |          |
| B3       | and     | x3       |
|         |         | b4       |

**CONTROL FOR EXAMPLE 2**

LOGIC,EXAMPLE2
RESETS,2
LEVEL 1
LEVEL 1
LEVEL 2
LEVEL 3

LOAD TRANSFER,101
INPUT,3
LOGIC,EXAMPLE2
OPTION DETECT,
DIAGNOSE
DIAGNOSE,1
MAINTAIN,NOSAFETY,DIAG,
END
### Dictionary of Machine Failures

#### For Logic Circuit Example 2

<table>
<thead>
<tr>
<th>Mach Nos</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Good Machine</td>
</tr>
<tr>
<td>5</td>
<td>B4 Output 0</td>
</tr>
<tr>
<td>9</td>
<td>X2 to B1 Open</td>
</tr>
<tr>
<td>13</td>
<td>B1 Int 1 Open</td>
</tr>
<tr>
<td>17</td>
<td>X4 to B3 Open</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mach Nos</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>B3 to B4 Open</td>
</tr>
<tr>
<td>6</td>
<td>B4 Output 1</td>
</tr>
<tr>
<td>10</td>
<td>X3 to B1 Open</td>
</tr>
<tr>
<td>14</td>
<td>B1 Output 0</td>
</tr>
<tr>
<td>18</td>
<td>B4 to B3 Open</td>
</tr>
</tbody>
</table>

**Input Order**: X1, X2, X3, X4

**Output Order**: W3, W4

**FB Order**: W1, B3

**Feedback Input**: 11

**Primary Input**: 1011

**Index of Output Set**: (1, 1)

**Output Vector**: 11

**Mach Numbers**: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

**Current Strategy**: BestNext

**Index of Input Set**: (1, 1)

**Primary Input**: 1011

**Index of Output Set**: (1, 1)

**Output Vector**: 01

**Mach Numbers**: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

**Index of Output Set**: (2, 2)

**Output Vector**: 00

**Mach Numbers**: 13 14 15 16 17

**Current Strategy**: BestNext

**Index of Input Set**: (2, 1)

**Primary Input**: 1011

**Index of Output Set**: (3, 1)

**Output Vector**: 01

**Mach Numbers**: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

**Index of Output Set**: (3, 2)

**Output Vector**: 11

**Mach Numbers**: 9 10 11 12 13 14 15 16 17

**Current Strategy**: BestNext

**Index of Input Set**: (3, 1)

**Primary Input**: 1111

**Index of Output Set**: (1, 1)

**Output Vector**: 01

**Mach Numbers**: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

**Index of Output Set**: (4, 2)

**Output Vector**: 11

**Mach Numbers**: 11 12 13 14 15 16 17 18

**Current Strategy**: BestNext

**Index of Input Set**: (4, 1)

**Primary Input**: 1110

**Index of Output Set**: (5, 1)

**Output Vector**: 01

**Mach Numbers**: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

**Index of Output Set**: (5, 2)

**Output Vector**: 00

**Mach Numbers**: 3
CURRENT STRATEGY: BESTNEXT

INDEX OF INPUT SET: (5, 1)
PRIMARY INPUT: 1011
INDEX OF OUTPUT SET: (6, 1)
OUTPUT VECTOR: 01
MACHINE NUMBERS: 1 2 6 8 12 18
INDEX OF OUTPUT SET: (6, 2)
OUTPUT VECTOR: 11
MACHINE NUMBERS: 17

CURRENT STRATEGY: BESTNEXT

INDEX OF INPUT SET: (6, 1)
PRIMARY INPUT: 1000
INDEX OF OUTPUT SET: (7, 1)
OUTPUT VECTOR: 00
MACHINE NUMBERS: 1 2 8 12 18
INDEX OF OUTPUT SET: (7, 2)
OUTPUT VECTOR: 01
MACHINE NUMBERS: 6

CURRENT STRATEGY: WANDER

INDEX OF INPUT SET: (7, 1)
PRIMARY INPUT: 1100
INDEX OF OUTPUT SET: (8, 1)
OUTPUT VECTOR: 00
MACHINE NUMBERS: 1 2 8 12 18
INDEX OF OUTPUT SET: (8, 1)
PRIMARY INPUT: 1110
INDEX OF OUTPUT SET: (9, 1)
OUTPUT VECTOR: 01
MACHINE NUMBERS: 1 2 8 12 18
INDEX OF OUTPUT SET: (9, 1)
PRIMARY INPUT: 1111
INDEX OF OUTPUT SET: (10, 1)
OUTPUT VECTOR: 01
MACHINE NUMBERS: 1 2 8 12 18
INDEX OF OUTPUT SET: (10, 1)
PRIMARY INPUT: 0111
INDEX OF OUTPUT SET: (11, 1)
OUTPUT VECTOR: 01
MACHINE NUMBERS: 1 2 8 12 18
INDEX OF OUTPUT SET: (11, 1)
PRIMARY INPUT: 0111
INDEX OF OUTPUT SET: (12, 1)
OUTPUT VECTOR: 11
MACHINE NUMBERS: 1 2 8 12 18
INDEX OF OUTPUT SET: (12, 1)
PRIMARY INPUT: 0111
INDEX OF OUTPUT SET: (13, 1)
OUTPUT VECTOR: 01
MACHINE NUMBERS: 8
RESET SUBSET: (13, 1)
FEEDBACK INPUT: 11
PRIMARY INPUT: 1111
INDEX OF INPUT SET: (13, 1)
PRIMARY INPUT: 1011
INDEX OF OUTPUT SET: (14, 1)
OUTPUT VECTOR: 11
MACHINE NUMBERS: 1 2 12 18
INDEX OF OUTPUT SET: (14, 1)
PRIMARY INPUT: 1111
INDEX OF OUTPUT SET: (15, 1)
OUTPUT VECTOR: 11
MACHINE NUMBERS: 1 2 12 18
INDEX OF OUTPUT SET: (15, 1)
PRIMARY INPUT: 1111
INDEX OF OUTPUT SET: (16, 1)
OUTPUT VECTOR: 11
MACHINE NUMBERS: 1 2 18
Figure 1. Structure of SEQANALZ
<table>
<thead>
<tr>
<th>INDEX OF INPUT SET</th>
<th>PRIMARY INPUT</th>
<th>INDEX OF OUTPUT SET</th>
<th>OUTPUT VECTOR</th>
<th>MACHINE NUMBERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(16, 1)</td>
<td>1001</td>
<td>(17, 1)</td>
<td>01</td>
<td>1 2 18</td>
</tr>
<tr>
<td>(17, 1)</td>
<td>1000</td>
<td>(18, 1)</td>
<td>00</td>
<td>1 2 18</td>
</tr>
<tr>
<td>(18, 1)</td>
<td>0000</td>
<td>(19, 1)</td>
<td>00</td>
<td>1 2 18</td>
</tr>
<tr>
<td>(19, 1)</td>
<td>0100</td>
<td>(20, 1)</td>
<td>00</td>
<td>1 2 18</td>
</tr>
<tr>
<td>(2, 3)</td>
<td>0001</td>
<td>(3, 3)</td>
<td>01</td>
<td>1 2 18</td>
</tr>
<tr>
<td>(3, 4)</td>
<td>0101</td>
<td>(4, 3)</td>
<td>01</td>
<td>1 2 18</td>
</tr>
<tr>
<td>(4, 4)</td>
<td>0100</td>
<td>(5, 3)</td>
<td>00</td>
<td>1 2 18</td>
</tr>
</tbody>
</table>

CURRENT STRATEGY: BESTNEXT
Figure 1. Structure of SEQANALZ
Figure 2. Example 1

Figure 3. Example 2
INDEX OF SUBROUTINES

SEQANALZ

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANYNEXT</td>
<td>68</td>
</tr>
<tr>
<td>BACKTRCE</td>
<td>62</td>
</tr>
<tr>
<td>BCODCT</td>
<td>54</td>
</tr>
<tr>
<td>BESTNEXT</td>
<td>67</td>
</tr>
<tr>
<td>BESTRETN</td>
<td>69</td>
</tr>
<tr>
<td>BSTRESET</td>
<td>68</td>
</tr>
<tr>
<td>CHECKOUT</td>
<td>60</td>
</tr>
<tr>
<td>COMMERG</td>
<td>70</td>
</tr>
<tr>
<td>COMBTEST</td>
<td>60</td>
</tr>
<tr>
<td>COMPILE</td>
<td>48</td>
</tr>
<tr>
<td>COMPSIML</td>
<td>65</td>
</tr>
<tr>
<td>COUNTBIT</td>
<td>54</td>
</tr>
<tr>
<td>CYCLWAND</td>
<td>68</td>
</tr>
<tr>
<td>DIAGNOSE</td>
<td>75</td>
</tr>
<tr>
<td>DICTONRY</td>
<td>36</td>
</tr>
<tr>
<td>DONTCARE</td>
<td>63</td>
</tr>
<tr>
<td>FINEPATH</td>
<td>66</td>
</tr>
<tr>
<td>FINDSORC</td>
<td>33</td>
</tr>
<tr>
<td>FLIPBIT</td>
<td>55</td>
</tr>
<tr>
<td>FLOATINT</td>
<td>55</td>
</tr>
<tr>
<td>FWDSET</td>
<td>62</td>
</tr>
<tr>
<td>GOOF</td>
<td>24</td>
</tr>
<tr>
<td>HUFFMAN</td>
<td>51</td>
</tr>
<tr>
<td>IMPLY</td>
<td>63</td>
</tr>
<tr>
<td>INFOGAIN</td>
<td>59</td>
</tr>
<tr>
<td>KEYTEST</td>
<td>9</td>
</tr>
<tr>
<td>LFOCTBCD</td>
<td>54</td>
</tr>
<tr>
<td>LOCMACHN</td>
<td>62</td>
</tr>
<tr>
<td>LOGBASE2</td>
<td>55</td>
</tr>
<tr>
<td>OCTBCD</td>
<td>53</td>
</tr>
<tr>
<td>OCTBCO</td>
<td>54</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCTLOADR</td>
<td>10</td>
</tr>
<tr>
<td>OPTION</td>
<td>15</td>
</tr>
<tr>
<td>PARAMETR</td>
<td>32</td>
</tr>
<tr>
<td>PARTMACH</td>
<td>56</td>
</tr>
<tr>
<td>PEELBITS</td>
<td>54</td>
</tr>
<tr>
<td>READCNTR</td>
<td>31</td>
</tr>
<tr>
<td>READINPT</td>
<td>33</td>
</tr>
<tr>
<td>RESETKEY</td>
<td>9</td>
</tr>
<tr>
<td>RESETSTR</td>
<td>71</td>
</tr>
<tr>
<td>RESTSBST</td>
<td>65</td>
</tr>
<tr>
<td>RSTFAIL</td>
<td>56</td>
</tr>
<tr>
<td>RSTRDATA</td>
<td>59</td>
</tr>
<tr>
<td>SAVEDATA</td>
<td>58</td>
</tr>
<tr>
<td>SETFAIL</td>
<td>55</td>
</tr>
<tr>
<td>SETFEED</td>
<td>58</td>
</tr>
<tr>
<td>SETKEY</td>
<td>9</td>
</tr>
<tr>
<td>SETPROB</td>
<td>59</td>
</tr>
<tr>
<td>SETRESET</td>
<td>65</td>
</tr>
<tr>
<td>SIMULATE</td>
<td>73</td>
</tr>
<tr>
<td>SIMULSET</td>
<td>64</td>
</tr>
<tr>
<td>SPRAYBIT</td>
<td>54</td>
</tr>
<tr>
<td>STRATDMP</td>
<td>72</td>
</tr>
<tr>
<td>STRATINT</td>
<td>72</td>
</tr>
<tr>
<td>STRASET</td>
<td>72</td>
</tr>
<tr>
<td>STRRTLINE</td>
<td>75</td>
</tr>
<tr>
<td>STRTSEQ</td>
<td>67</td>
</tr>
<tr>
<td>TRANSPOS</td>
<td>55</td>
</tr>
<tr>
<td>TRYUCOMB</td>
<td>70</td>
</tr>
<tr>
<td>TRYUWAND</td>
<td>70</td>
</tr>
<tr>
<td>UPFEED</td>
<td>57</td>
</tr>
<tr>
<td>UPPARTIT</td>
<td>57</td>
</tr>
</tbody>
</table>
Distribution list as of February 1, 1964

1 Director
Air University Library
Maxwell Air Force Base, Alabama
Attn: CR-4603a

1 Redstone Scientific Information Center
U.S. Army Missile Command
Redstone Arsenal, Alabama

1 Electronics Research Laboratory (Unclassified)
University of California
Berkeley 4, California

2 Hughes Aircraft Company
Florence and Tale
Culver City, California
Attn: N. E. Devereux
Technical Document Center

3 Autonetics (Unclassified)
9156 East Imperial Highway
Downey, California
Attn: Tech. Library, 3041-11

1 Dr. Arnold T. Nordaeeck
General Motors Corporation
Defense Research Laboratories
6767 Hollister Avenue
Goleta, California

1 University of California (Unclassified)
Lawrence Radiation Laboratory
P. O. Box 808
Livermore, California

1 Mr. Thomas L. Hartwick
Aerospace Corporation
P. O. Box 95085
Los Angeles 45, California

1 Lt. Colonel Willard Levin
Aerospace Corporation
P. O. Box 95085
Los Angeles 45, California

1 Professor Zorab Kaprielian
University of Southern California
University Park
Los Angeles 7, California

1 Sylvania Electronic Systems - West
Electronic Defense Laboratories
P. O. Box 201
Mountain View, California
Attn: Documents Center

1 Varian Associates
61 Hansen Way
Palo Alto, California
Attn: Dr. Ira Weissman

1 Houston Den loyal (Unclassified)
Library Supervisor
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

1 Professor Nicholas George
California Institute of Technology
Electrical Engineering Department
Pasadena, California

1 Space Technology Labs., Inc.
One Space Park
Redondo Beach, California
Attn: Acquisitions Group
STL Technical Library

2 Commanding Officer and Director
U.S. Naval Electronics Laboratory
San Diego 52, California
Attn: Code 2808, C. S. Manning

1 Commanding Officer and Director
U.S. Navy Electronics Laboratory
San Diego 52, California
Attn: Library

1 Office of Naval Research Branch Office
1000 Garry Street
San Francisco, California

1 The RAND Corporation
1700 Main Street
Santa Monica, California
Attn: Library

1 Stanford Electronics Laboratories (Unclassified)
Stanford University
Stanford, California
Attn: SEL Documents Librarian

1 Dr. L. F. Carter
Chief Scientific Air Force
Room 4E-324, Pentagon
Washington 25, D. C.

1 Mr. Robert L. Feik
Associate Director for Research
Research and Technology Division
AFSC
Rolling Air Force Base 25, D. C.

1 Captain Paul Johnson (USN-Ret)
National Aeronautics and Space Administration
1520 H Street, N. W.
Washington 25, D. C.

1 Major Edwin M. Myers
Headquarters USAF (APFD)25, D. C.

1 Dr. James Ward
Office of Deputy Director
Department of Defense
Washington 25, D. C.

1 Dr. Alan T. Waterman
Director, National Science Foundation
Washington 25, D. C.

1 Mr. G. D. Watson
Defense Research Member
Canadian Joint Staff
2450 Massachusetts Ave., N. W.
Washington 8, D. C.

1 Mr. Arthur G. Winer
Air Force Systems Command
Andrews Air Force Base
Washington 25, D. C.

1 Director, Advanced Research
Projects Agency
Washington 25, D. C.

1 Air Force Office of Scientific Branch
Directorate of Engineering Sciences
Washington 25, D. C.
Attn: Electronics Division

1 Director of Science and Technology
Headquarters, USAF
Washington 25, D. C.
Attn: AFRST-EL/GU

1 AFRS - SC
Headquarters, USAF
Washington 25, D. C.

1 Headquarters, R & T Division (Unclassified)
Rolling Air Force Base
Washington 25, D. C.
Attn: RTHR

1 Headquarters, U.S. Army Material Command
Research Division, R & D Directorate
Washington 25, D. C.
Attn: Physics & Electronics Branch
Electronics Section

1 Commanding Officer
Diamond Ordnance Fuze Laboratories
Washington 25, D. C.
Attn: Librarian, Room 211, Bldg. 92

1 Operation Evaluation Group
Chief of Naval Operations (OP-03EO)
Department of Navy
Washington, D. C. 20350

1 Chief of Naval Operations (Code OP-ONT)
Department of the Navy
Washington, D. C. 20350

1 Commanding Officer
U.S. Army Personnel Research Office
Washington 24, D. C.

1 Commanding Officer & Director
Code 143 Library
David W. Taylor Model Basin
Washington, D. C. 20007

1 Chief, Bureau of Ships (Code 686)
Department of the Navy
Washington, D. C. 20360

1 Chief, Bureau of Ships (Code 732)
Department of the Navy
Washington, D. C. 20360

1 Chief, Bureau of Naval Weapons
Technical Library, DIJ-3
Department of the Navy
Washington, D. C. 20360

1 Director, (Code 5140)
U. S. Naval Research Laboratory
Washington, D. C. 20390

1 Chief of Naval Research (Code 437)
Department of the Navy
Washington, D. C. 20360

1 Dr. H. Wallace Sinaiko (Unclassified)
Institute for Defense Analyses
Research & Engineering Support Division
1666 Connecticut Ave., N. W.
Washington 9, D. C.

1 Data Processing Systems Division
National Bureau of Standards
Attn: E. H. F. Ward
Washington 25, D. C.

1 Exchange and Gift Division (Unclassified)
The Library of Congress
Washington 25, D. C.

1 NASA Headquarters
Office of Applications
400 Maryland Avenue, S. W.
Washington 25, D. C.
Attn: Mr. A. M. Greg Andrus
Code FC

1 APOC (PG499)
Epin Air Force Base
Florida

1 Martin Company
P.O. Box 5537
Orlando, Florida
Attn: Engineering Library
MP-30

1 Commanding Officer
Office of Naval Research, Chicago Branch
6th Floor, 230 North Michigan
Chicago 1, Illinois

1 Laboratories for Applied Sciences
University of Chicago
6226 South Drexel
Chicago 37, Illinois

1 Librarian
School of Electrical Engineering
Purdue University
Lafayette, Indiana