CR 86066

E-2265

ON-LINE LOGICAL SIMULATION (OLLS)

by

R. L. Alonso, H. R. Howie,
J. C. Pennypacker, G. Schwartz,
H. A. Thaler

May 1968

INSTRUMENTATION LABORATORY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge 39, Mass.

E-2265

ON-LINE LOGICAL SIMULATION (OLLS)

by

R. L. Alonso, H. R. Howie,
J. C. Pennypacker, G. Schwartz,
H. A. Thaler

May 1968

INSTRUMENTATION LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS

Approved: Eldon C Hall Date: 29 May 68
ELDON C. HALL, DIRECTOR, DIGITAL DEVELOPMENT
APOLLO GUIDANCE AND NAVIGATION PROGRAM

Approved: R R Ragan Date: 3 Jun 68
RALPH R. RAGAN, DEPUTY DIRECTOR
INSTRUMENTATION LABORATORY

Acknowledgment

The major part of OLLS has been funded through NASA's Electronics Research Laboratory, contract NAS 12-140. Additional non-NASA support has been derived, from within Instrumentation Laboratory, for the development of the FILLIP list processing language. FILLIP is not part of OLLS, however.

Authorship of the various portions is as follows:
Section 2, "OLLS/1800" is by H. Robert Howie
Section 3, "OLLS/360":

- 3.1 "Data Structure", by James C. Pennypacker.
- 3.2 "Device Definition", by James C. Pennypacker.
- 3.3 "Simulation" by Herbert A. Thaler.
- 3.4 "Drawing Algorithms", by H. Robert Howie.
- 3.5 "Program Structure", by James C. Pennypacker and Gary Schwartz.
- 3.6 "ON-LINE System", by H. Robert Howie and Ramon Alonso.

Ramon Alonso is the OLLS project director.

E-2265

ON LINE LOGICAL SIMULATION (OLLS)

ABSTRACT

This report describes techniques for the simulation of logic circuits, combinational and sequential, and for the automatic drawing of circuit schematics. The first part of the report treats an initial system which is somewhat limited in scope in that it is strictly card-oriented and has a selection of logic devices limited to those assembled into the control program. The last part describes a fully expended system in which the user can define and modify his own devices, either on-line via a CRT, or off-line with punched cards. This system enables the designer to perform all phases of logic design: device definition, test, redesign, and retest, with the aid of a computer to free him of the clerical details of drawings, signal lists, retrofits, and wire-wrap control cards.

by: R. Alonso

May 1968

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

Section 1

Introduction

The possibility of using computers to aid designers has been recognized and exploited, in various ways, for the last several years. Designers can have mechanized help in small circuit design (ECAP, NET), and in some forms of mechanical design (SKETCHPAD). As SKETCHPAD showed, the implications of a Cathode Ray Tube system whereby the designer and the computer interact, as opposed to the more prevalent processing systems, are many and exciting.

The possibility of using major data processing aids for logical designs became an important concern to those who had been engaged, for quite some time, in the development of medium-sized computer systems, especially if those systems could be made interactive. But interactive or not, our accumulated experience in logical design indicated the near necessity of mechanized files, drafting aids, and simulations.

The initial objective of MIT/IL was not so much to demonstrate the power of a new approach (Computer Aided Design) as to develop and implement a practical system. We are still short of that goal in that we do not have an operational interactive system; we do have a batch system (OLLS/1800) and major portions of the more ambitious OLLS/360 system. Consequently, the present report is in part a demonstration of achievement and in part a blueprint of present and future developments.

The report is divided into two major parts: Section 2, which deals with an initial, limited, but working system for the Honeywell 1800; and Section 3, which describes a system that attempts a great deal more, and that is designed for execution in an IBM 360/75 machine.

OLLS/1800 is a card system, with very limited file capability, wherein the logical device models are an integral part of the program. It was written without recourse to a list-processing language, and has been in use for about six months.

In early 1967 a decision was made by the Digital Computation group (which runs the data processing system of Instrumentation Laboratory) to implement a major list-processing language called FILLIP, and it was decided then that OLLS/360 should be based on FILLIP. As of this writing, FILLIP is still under development for its overall system aspects, and, consequently, most of OLLS/360 is untested. The major features of FILLIP, and its power, are described in an as yet unpublished report by Charles A. Muntz and J. Halcombe Laning, Jr.

Section 2

The Honeywell 1800 Working System

H. Robert Howie

2.1 Objective

When it became apparent that the IBM 360, the on-line CRT, and the list-processing language would not be available much before 1968, a more limited system designed to operate on an available Honeywell 1800 computer was developed and demonstrated in 1967. It was hoped that this system would provide (in addition to an operating digital simulator) some experience with drawing interconnection algorithms, simulation algorithms, and, through feedback from in-house users of the system, a better understanding of what input-output techniques are most acceptable and convenient to the digital circuit designer.

This chapter describes the concepts and operation of a schematic drawing program and of a simulation program currently available for the Honeywell 1800 computer.

2.2 General Description

Two main programs comprise our system:

a) program DRAWSCHEMATIC which reads cards describing devices and device interconnections, creates and maintains a logic file, and, through subroutine DRAWLINES, computes computer interconnection paths and produces a finished schematic on a CALCOMP plotter.

b) program SIMSCHEMATIC which reads the logic file created and stored by DRAWSCHEMATIC and simulates the circuit using input signal values supplied on cards by the user at execution time. Output traces of the history of any signals specified are plotted, oscillograph style, on a CALCOMP plotter.

Figures 2-1 and 2-2 are typical of the quality and complexity of designs the programs are capable of handling. The Arithmetic Unit in Fig. 2-1 contains about 150 gates and required about 20 minutes of computer time to create the logic file and produce the drawing plot tape. (About 5 minutes are spent in system management - rewinding and labeling tapes, etc.) The plotting was done off-line and required about 30 minutes on the CALCOMP plotter. The simulation of the Arithmetic Unit included testing the ability of the circuit to add, to shift, and to shift-and-add. Five logic-design errors were discovered along the way and several clerical errors were corrected before the simulation shown in Fig. 2-2 was successful. The simulation ran for 200 simulation time units (the equivalent of 4 microseconds if a time unit of 20 nanoseconds is used as the typical gate delay), or required 7 minutes of real time on the H1800.

2.2.1 DRAWSCHEMATIC

Program DRAWSCHEMATIC is intended to run under a batch-processing operating system. Its sole input is from cards and its output is written on magnetic tape for off-line printout and plotting. (An optional output facility punches a deck of cards for automated wirewrapping.)

The input deck can be in either or both of two card formats. The preferred card format was designed specifically for this program and was intended to be simple, easy to punch, easy to read, and organized the way a circuit designer would find convenient. For compatibility with wirewrap programs developed 3 years ago, wirewrap cards are also accepted. These cards, although difficult to punch manually or to scan, are logically equivalent to the new card format. Detailed descriptions of card formats are given with the examples in a later section.

The program operates in three modes: DRAW, REVISE, and REPRINT.

In the DRAW mode, a new logic file is created by the input deck. Cards which describe devices to be included are read first. The designer may select

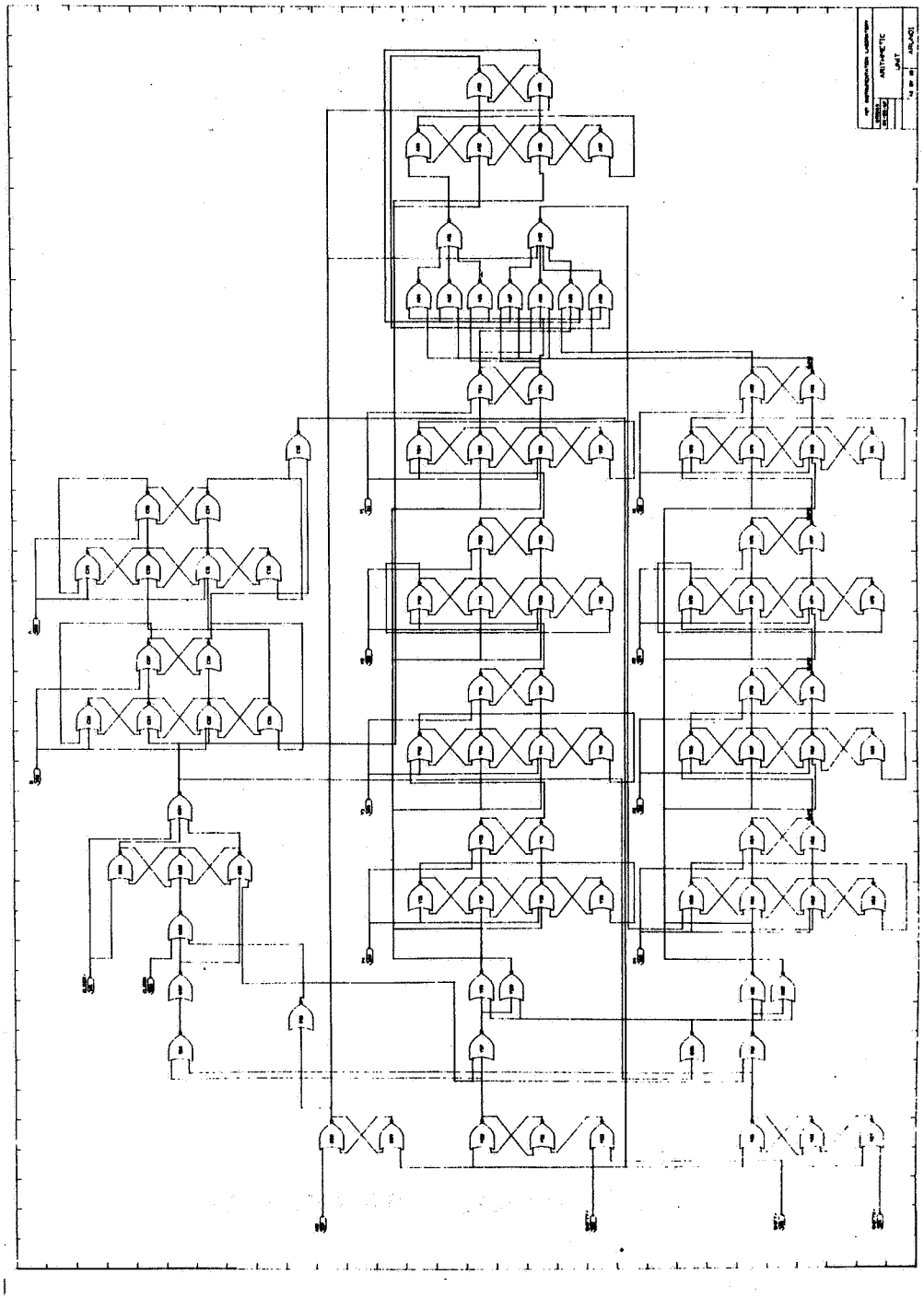


Fig. 2-1

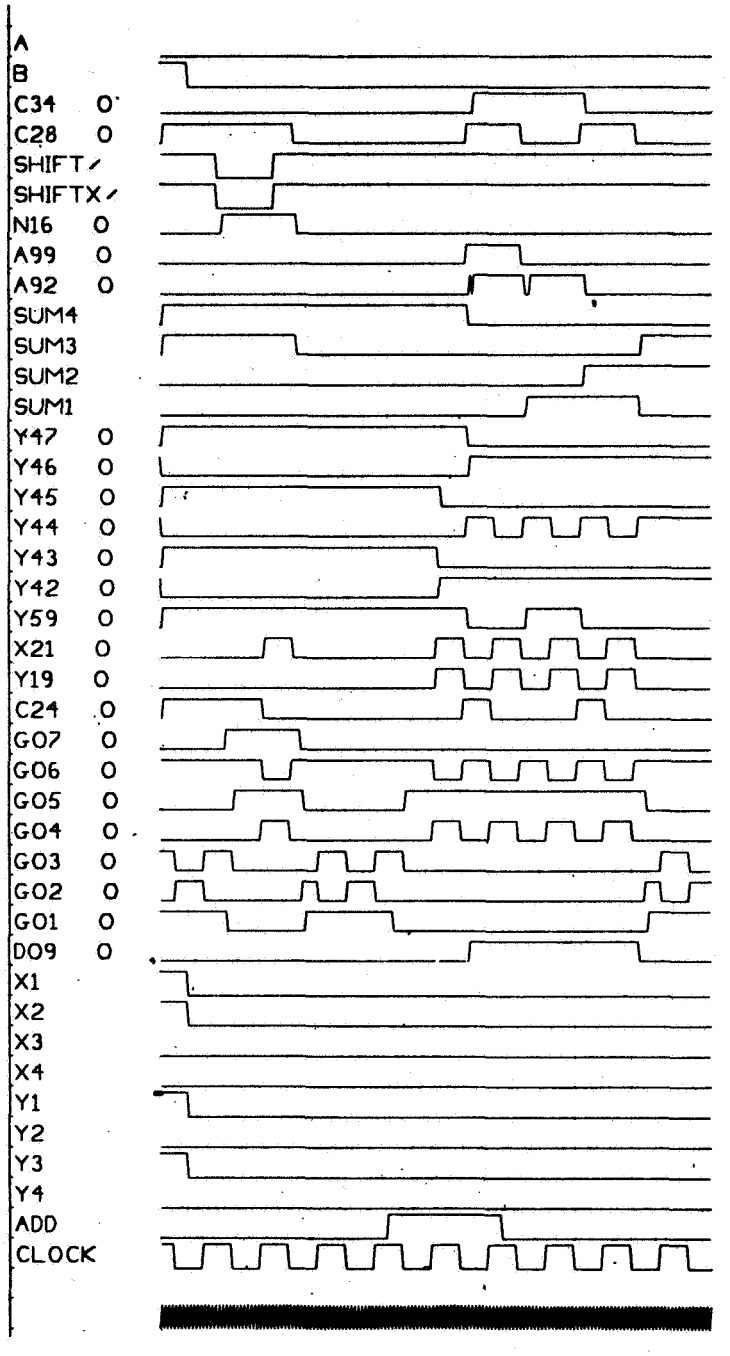

A
B
C34 O
C28 O
SHIFT/
SHIFTX/
N16 O
A99 O
A92 O
SUM4
SUM3
SUM2
SUM1
Y47 O
Y46 O
Y45 O
Y44 O
Y43 O
Y42 O
Y59 O
X21 O
Y19 O
C24 O
GO7 O
GO6 O
GO5 O
GO4 O
GO3 O
GO2 O
GO1 O
DO9 O
X1
X2
X3
X4
Y1
Y2
Y3
Y4
ADD
CLOCK

Fig. 2-2

from 25 available device types, such as 3 input NOR gate, set-reset flip-flop, etc., (see Fig. 2-3). He may include up to 400 devices in any one logic file (drawing). Each device card describes the identification number of the device, its location on the drawing (optional if no drawing is to be produced), and the device type.

Cards which describe the logical interconnections are read next. The designer may specify the interconnections in either of two ways, depending on whether or not he wishes to assign a signal name to the connection. He may write:

[connect] signalname [to] devicename devicepin

or he may write:

[connect] devicename devicepin [to] devicename devicepin.

This provides a convenient way of describing any connection without cluttering up a drawing with unnecessary signal names, and yet allows connections to be named where desired. The actual path which the signal will take on the drawing is not specified by the user but rather is computed for him by subroutine DRAWLINES. This computed path becomes part of the logic file and is not recomputed for every REVISION or REPRINT.

Some clerical cards mark the beginning and end of the input stream; give the drawing a name, number, and author, and size; and specify what type of output is desired, i.e., full listing, error messages only, a drawing, a wirewrap deck, etc.

In the REVISE mode, an old logic file created by an earlier run in either the DRAW or REVISE mode is updated by cards in the input stream. This mode operates exactly as the DRAW mode with the addition that cards are accepted which allow the designer to DELETE or MOVE devices which are already in the file. In both cases, all signals that enter or leave a device which was deleted or moved are themselves deleted or moved automatically, and the logic file is updated accordingly. Similarly, the designer is allowed to REMOVE signal interconnections. The operation of this mode saves the time and expense of resubmitting the entire input deck just to correct minor errors. Only those signals that are directly affected by the change are recomputed for the file.

In both DRAW and REVISE modes, checks are made on the validity of the input stream as much as possible, and diagnostic error messages are always printed when an error is discovered. This service has been found to be as valuable to the designer as is the drawing or the simulation. Examples of such errors are finding non-unique device identification numbers or signal names,

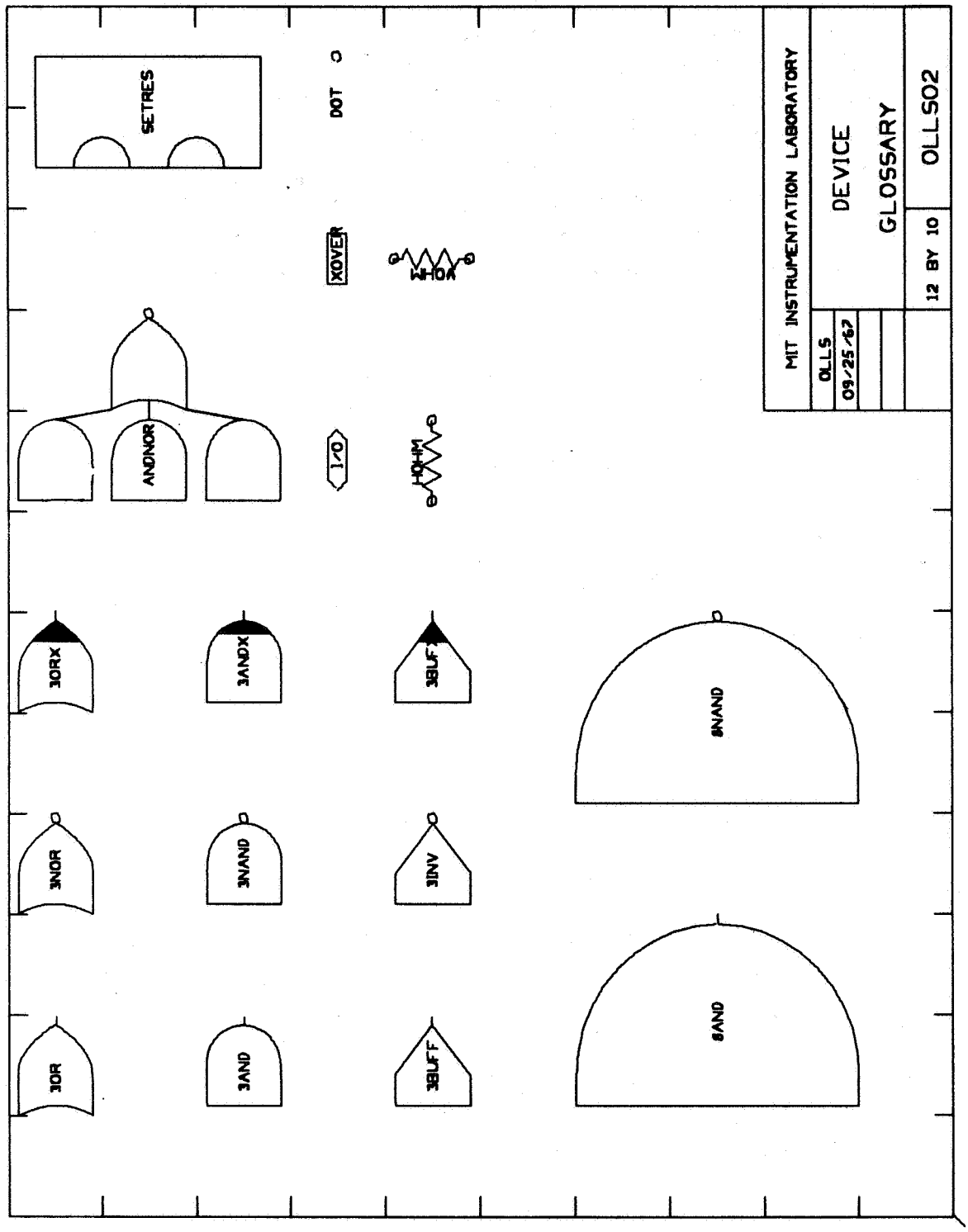

MIT INSTRUMENTATION LABORATORY
OLLS
09/25/67
DEVICE
GLOSSARY
12 BY 10
OLLS02
SETRES
DOT
XOVER
1/0
ANDNOR
1ORX
3ANDX
3BUFF
8NAND
3NOR
3NAND
3INV
8AND
1OR
3AND
3BUFF

Fig. 2-3

finding two different signals connected to the same point, or finding a signal attempting to connect to a non-existent device.

In addition to producing whatever hard copy was specified by the designer, the logic file is always catalogued and stored on magnetic tape with other logic files for later REVISION, REPRINTS, or simulation.

The REPRINT mode allows the designer or any interested person to extract specified information from the file without changing its contents. A listing or a drawing scaled to any convenient size is made available at minimum cost.

2.2.2 SIMSCHEMATIC

Program SIMSCHEMATIC enables the designer to simulate the logic file created and stored by program DRAWSCHEMATIC. Since this step is intended to replace or augment the laboratory breadboard of the circuit, the designer is provided with the means of specifying or changing the effective signal delay of any devices in the file, the input signal timing, and the topology of the circuit itself without going all the way back to program DRAWSCHEMATIC to test minor changes.

The input to the program consists of the logic file to be simulated, which was stored on magnetic tape by DRAWSCHEMATIC, and a short deck of cards which describe such things as:

a) The list of devices which are to be considered active for this portion of the simulation and what the signal delay of each device shall be. Gates can be activated or deactivated any time during a simulation run or the delay can be changed to a new value as desired.

b) The input waveforms necessary to fully test the circuit and initial conditions for signals which might otherwise be indeterminate. It is often desirable in testing a laboratory breadboard to inject inputs to portions of the circuit which might otherwise be unused or unavailable. The capability is provided here to PATCH existing or non-existing signals to any portion of the circuit or to REMOVE undesired connections of existing signals. Input waveforms can now be injected to the new connections which exist only for the duration of the simulation and do not affect the permanent logic file.

c) The list of signals to be traced for output on the CALCOMP plotter.

d) The timing of events to follow, such as a snapshot of the file, magnification of a certain portion of the output, how long to simulate, and when to stop or restart.

2.3 Logic-File Organization

Figure 2-4 shows the structural organization of the logic file. It consists of two major parts, the device list and the signal list. A third list, the list of drawing interconnection points, strictly speaking, does not belong in the logic file. The interconnection list (produced by subroutine DRAWLINES) is kept to save time on revisions and reprints. These three lists contain enough information to draw and simulate a logic diagram. For storage reasons the logic file produced by DRAWSCHEMATIC does not contain the interlocking pointers necessary for rapid simulation. These pointers are added to the logic file by SIMSCHEMATIC as the logic file is read in.

2.3.1 DRAWSCHEMATIC

The DEVICE LIST is an alphabetically organized list with each entry containing the device identification (5 characters), the device coordinates, and a code number which indicates the device type. This code number enables the program to fetch from a glossary of devices such information as device shape, terminal locations, and simulation behavior.

The SIGNAL LIST is an alphabetically organized list with each entry containing the signal name (8 characters) and a sublist for each connection of the signal. The sublist contains the device and pin number to which the signal is connected, the (X,Y) coordinates of the device and the device type code, the signal load of the device, a flag which indicates whether the signal was affected by some operation in the REVISE mode (the OLD/NEW entry), and an entry which either points to the list of drawing interconnection points (if any), or indicates that the connection is to the signal source (STATUS = 2 or 3), or that the connection is to be labeled only (STATUS = 1). (A signal which has no source on the drawing is labeled at each input where it is used and no interconnections are made on the drawing, although for simulation purposes all inputs with the same label are logically interconnected.)

Figure 2-6 shows what the logic file would look like for the sample circuit shown in Fig. 2-7. Notice that the logic file is made up of three signals: "LABEL", "OUTPUT", and "6Z4 0".

The signal named LABEL has no source on this drawing, i.e. it is nowhere connected to the output of a gate. It therefore has a STATUS = 1 in the logic file, indicating to the output plotting routine that no interconnection lists are to be found and its name is merely to be labeled where it is used.

The signal named OUTPUT is defined at a source, the output of a gate 1A1. It therefore has a STATUS = 2 for the connection to device 1A1, indicative that the signal name is to be plotted. The rest of the connections for signal

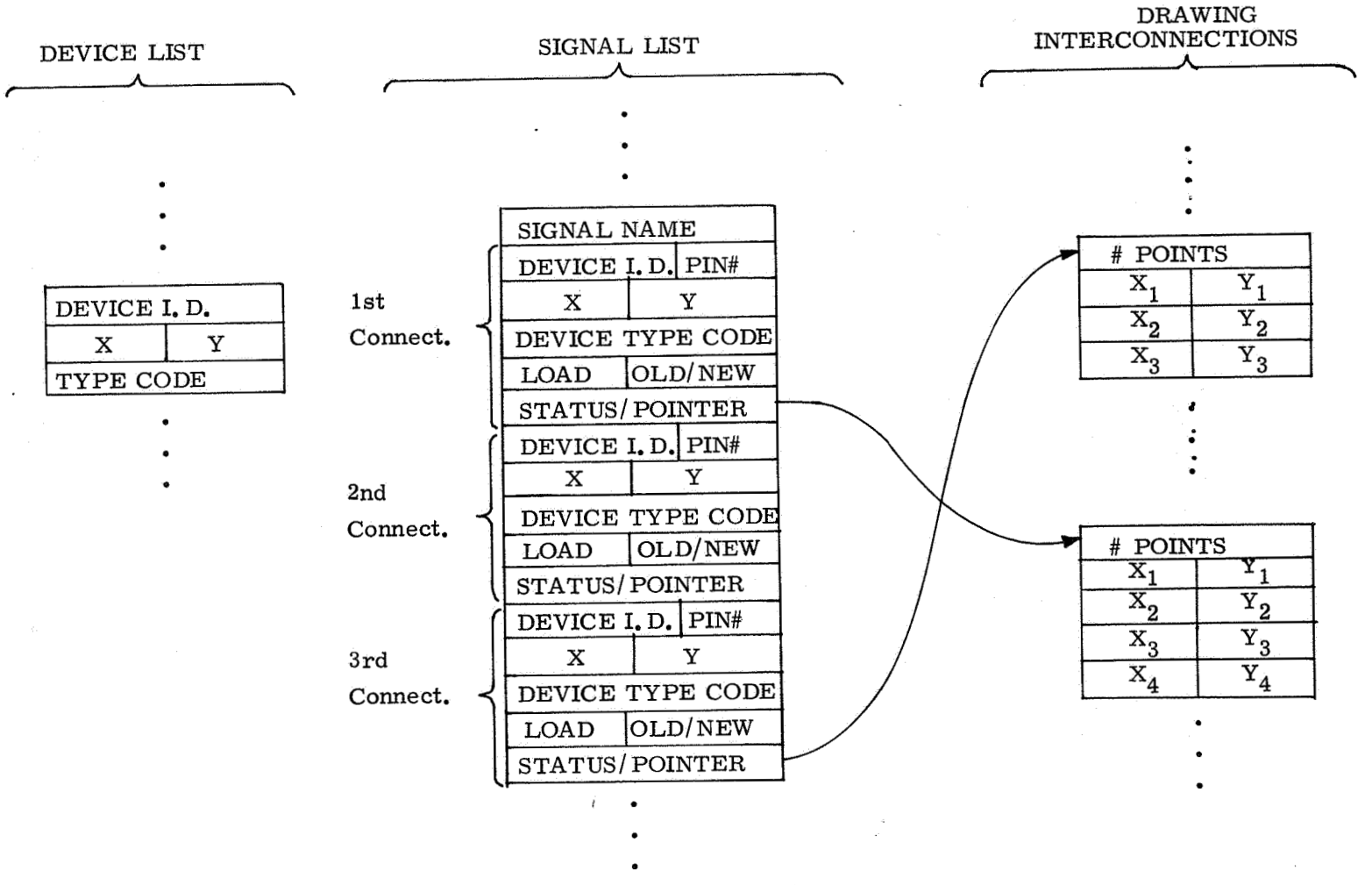


Fig. 2-4 Logic File Organization.

```
D       DRAW      SCHEMATIC FIG2.7                OLLS     10 BY 08
S       SCALE TO  09 BY 07
S       NAME IT   SAMPLE CIRCUIT
S       PRODUCE   SCHEMATIC
G       ADD       1A1             1       4        3NOR
G       ADD       401             6       5.5      I/O
G       ADD       3B7             4       2        SETRES
G       ADD       6Z4             4       5        3NAND
W       DEFINE    OUTPUT          1A1     0        -30
W       SIGNAL    OUTPUT          6Z4     3        005
W       SIGNAL    OUTPUT          3B7     2        007
W       SIGNAL    LABEL           3B7     8        005
W       SIGNAL    LABEL           6Z4     1        005
W       CONNECT   6Z4       0     674     0        -30
W       CONNECT   6Z4       0     401     0        005
W       CONNECT   6Z4       0     1A1     1        005
D       PLOTNOW
```

Fig. 2-5 Input Deck to Produce the SAMPLE Circuit in Fig. 2-7.

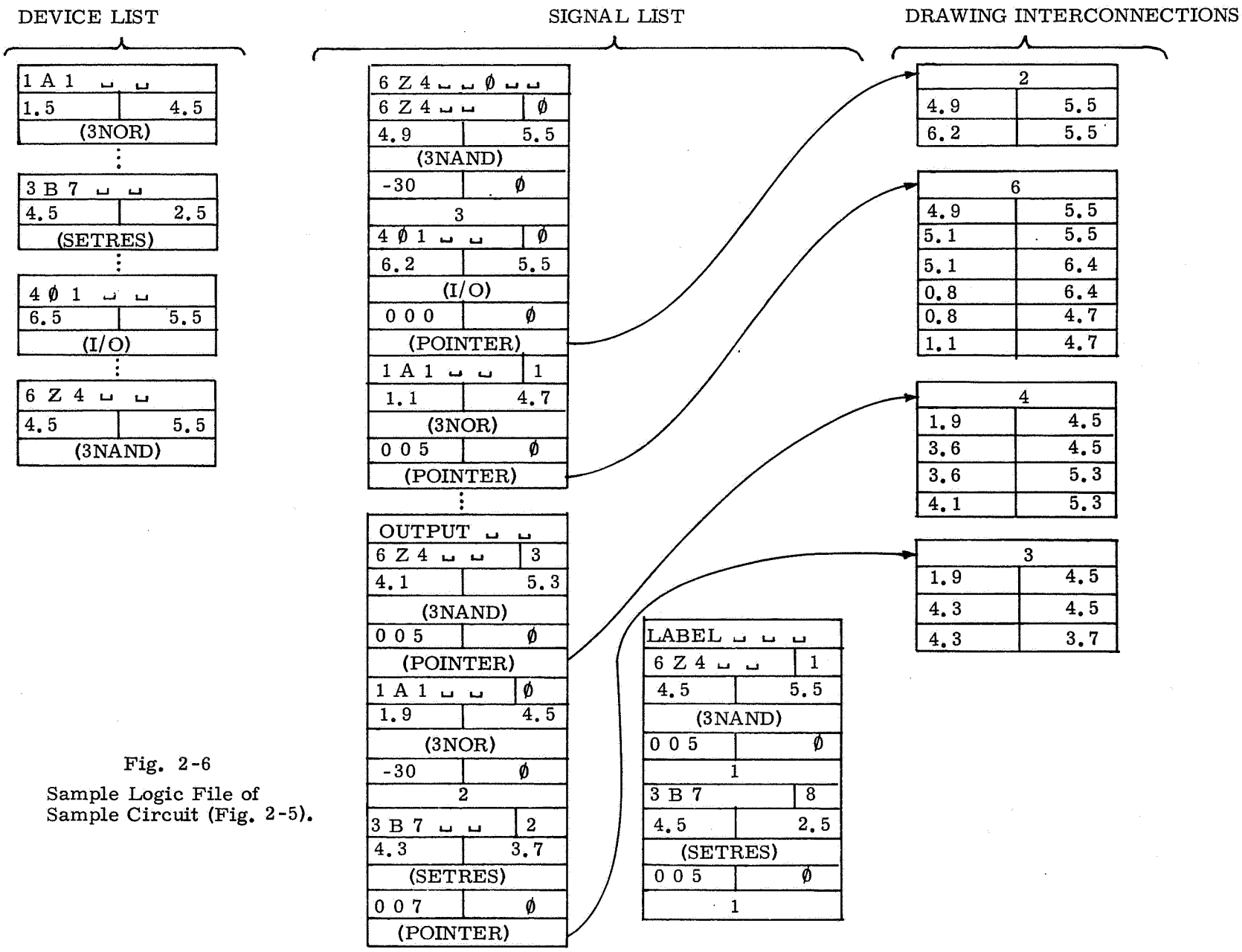
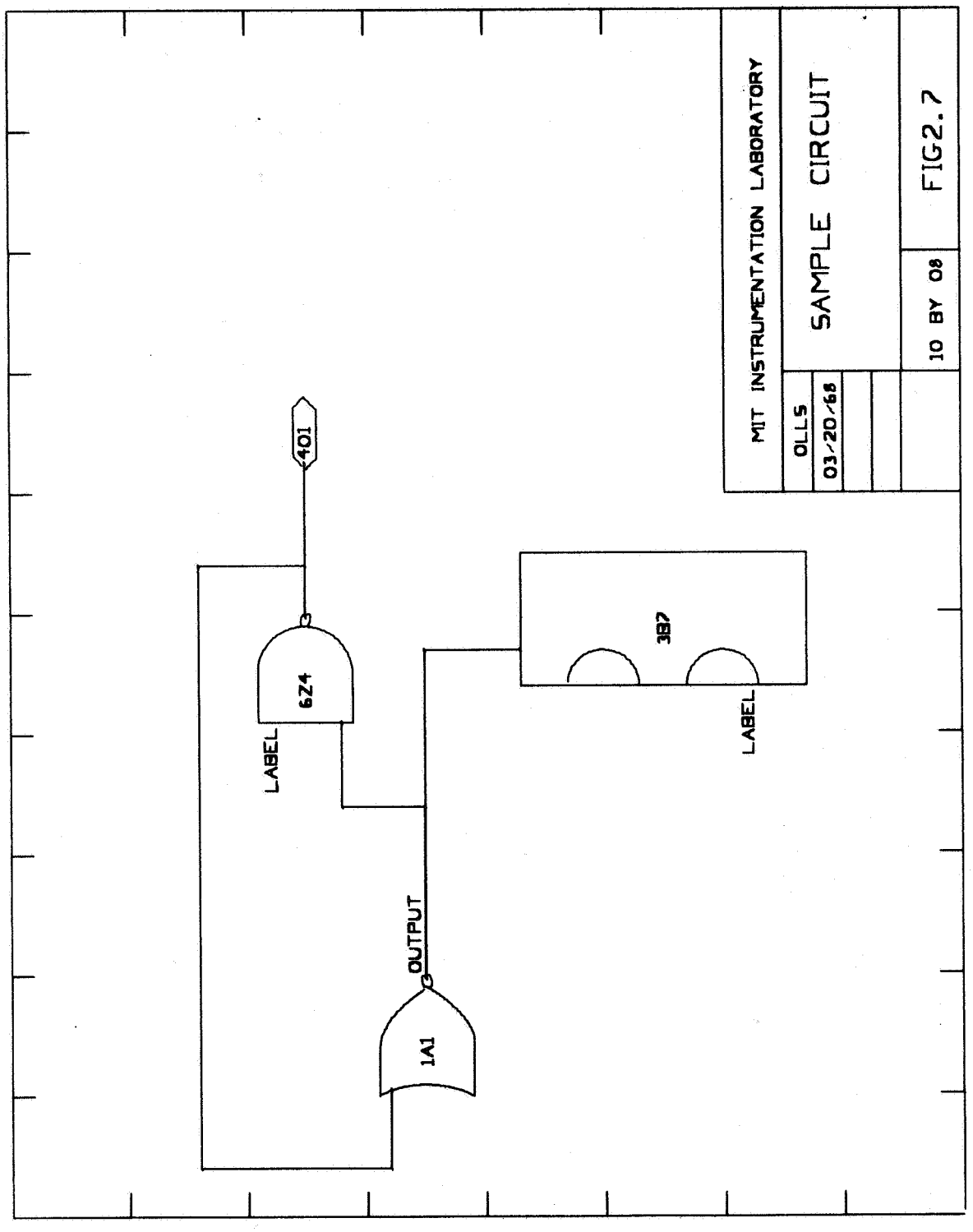


Fig. 2-6

Sample Logic File of
Sample Circuit (Fig. 2-5).


MIT INSTRUMENTATION LABORATORY
OLLS
03/20/68
SAMPLE CIRCUIT
10 BY 08
FIG2.7
401
6Z4
LABEL
387
LABEL
OUTPUT
1A1

OUTPUT contain pointers to the drawing interconnection lists produced by DRAWLINES. The signal at the output of gate 6Z4 pin 0 also has a well defined source, but, because of the way in which it was defined (see below), its signal name will not appear on the drawing and it is flagged with STATUS = 3.

Figure 2-5 shows the input deck which created this drawing and logic file.

Each card contains a descriptor in column 1 for ease in sorting the deck if necessary. (D = director card, S = subdirector, G = gate card, W = wiring card). Director and subdirector cards are clerical and do not affect the logic file (except to abort the run if incorrectly specified). Gate cards can ADD, or MOVE gates. The first field after the verb is always the device ID, the second and third fields are the X and Y position (for ADD and MOVE), and the fourth field specifies the device type (ADD only). All devices except I/O pins and resistors are placed at X, Y coordinate values of integer + 1/2 for ease in gate-to-gate registration. Gate 6Z4, for example, will be placed at X = 4.5, Y = 5.5 instead of X = 4, Y = 5 as specified on the ADD card.

Wiring cards are of two varieties: those which concern signals whose names appear on the drawing and those which concern signals whose names do not appear. Of the first variety are cards which specify DEFINE, SIGNAL, or REMOVE. DEFINE specifies that there exists a source for the signal and its location and it triggers DRAWLINES to compute interconnection paths for all SIGNAL cards specifying the same signal name. The DEFINEd source is flagged in the logic file with STATUS = 2. If no DEFINE card can be found for a signal, DRAWLINES does not attempt to compute interconnection paths. Instead, the signals are added to the logic file and flagged as "label only" (STATUS = 1). REMOVE allows the designer to selectively disconnect signals as desired. If the DEFINEd source of a signal is REMOVEd and not reDEFINEd, the remaining connections of the signal revert to the status of labels only (STATUS = 1). The first field on the card after the verb specifies the signal name, the second and third specify the device ID and pin number, and the fourth specifies optional load information. (Loads of -30 and 005 are assumed for sources and non-sources, respectively, if this field is left blank.)

Of the second variety of W cards are CONNECT and DELETE cards. These specify signals which are to be connected in the logic file and which are to be drawn on the schematic but whose signal names are not to appear. The intent here is to relieve the designer of the necessity of dreaming up a unique signal name for an uninteresting connection and allow him to specify the connection by device ID and pin number to device ID and pin number. Each CONNECT card specifies the connection of one end of an equipotential in the logic file. Thus the requirement that a device ID and pin number be CONNECTed to itself to specify a source. See Fig. 2-7.

W CONNECT 6Z4 0 6Z4 0 -30

This card specifies the source for a signal (actually named 6Z4 0) and flagged it with STATUS = 3 in the logic file to inhibit plotting its signal name. It also triggers DRAWLINES to compute interconnection paths for all other occurrences of CONNECTions of the same device ID and pin number. If the device ID and pin number is not CONNECTed to itself, no source is defined and all CONNECTions of the signal are entered in the logic file and flagged for label only (STATUS = 1). DELETE acts like REMOVE except that device ID and pin number are specified instead of signal name.

2.3.2 SIMSCHEMATIC

Program SIMSCHEMATIC creates a temporary logic file of its own by adding information to the logic file produced by DRAWSCHEMATIC. The new information consists of gate delays, signal values, direct pointers from signals to devices, and some additional status flag words. Figure 2-8 shows the general structure of the logic file after modification by SIMSCHEMATIC. The INPUT LIST and the TRACE LIST are expanded as required by the input deck.

The SIMULATION DEVICE LIST contains a flag which indicates whether or not the gate is active. If the gate is active (ACTIVE = 1), the TYPE CODE is used to branch to a section of programmed coding which logically combines the signal values at PIN00 through PIN12 (most devices only have 3 or 4 pins and the last 8 or 9 pins are ignored for those devices) to produce an output value (s). This output value (usually PIN00) is then propagated through the delay line using the DELAY value and the delay indices for this instance of the device to produce a final output value which replaces the former output value (usually PIN00). (See Fig. 2-9.)

A true digital delay line should insert new values at one end and, by shifting the values in the line, extract the value at a later time either at the other end of the line or at a midpoint. Since shifting is slow for programs written in a high level language, the index scheme shown in Fig. 2-9 was adopted. Values are inserted into the line at index location J and extracted at index location I. Both indices are incremented (modulo 16- the maximum delay) but maintain a constant difference equal to the value specified in DELAY.

Gates are activated, inactivated, and delays specified at any time desired by the designer by cards with a G in Column 1. The four forms of these cards are:

G ACTIVE ALLGATES 3

specifies all gates are to be active with a delay of 3

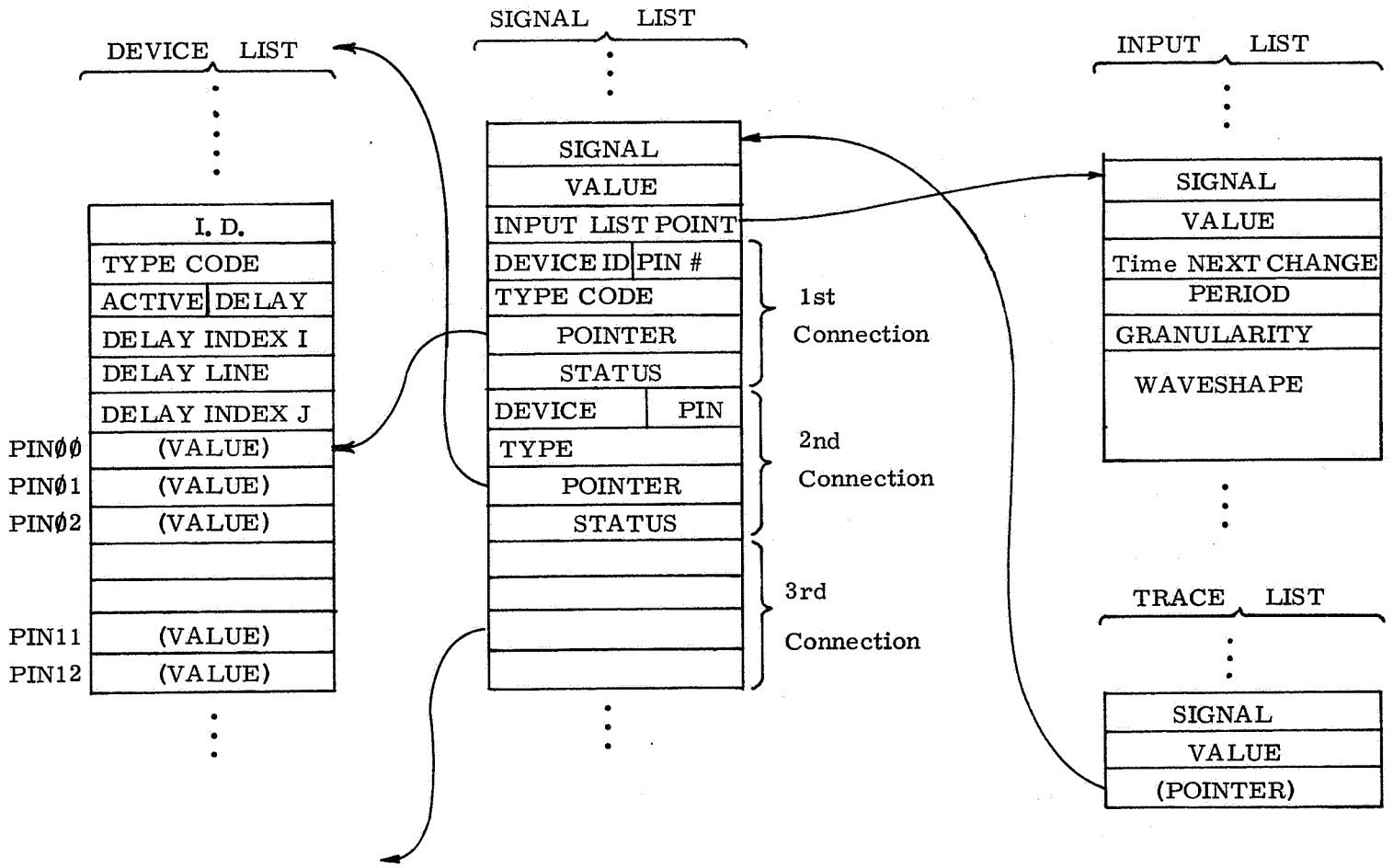


Fig. 2-8 Simulation Logic File Organization.

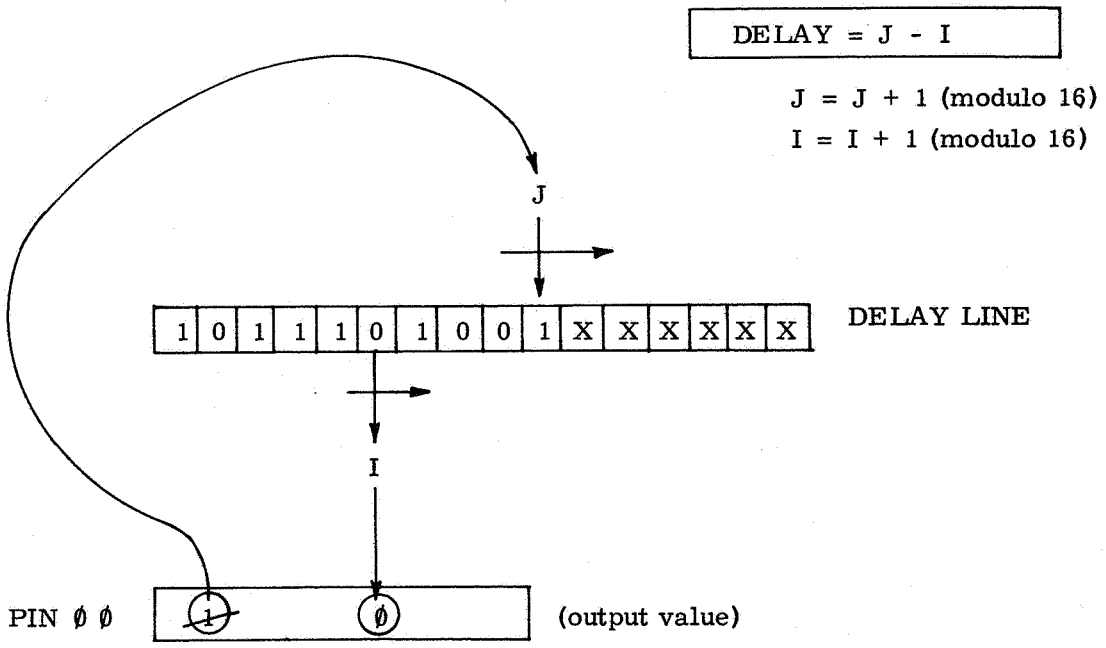


Fig. 2-9 Operation of the Delay Line.

G ACTIVE 6Z4 7

specifies that gate 6Z4 is to be active with a delay of 7

G INACTIVE ALLGATES

specifies that all gates are to be inactive (not very useful)

G INACTIVE 1A1

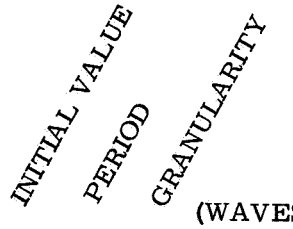
specifies that gate 1A1 is to be considered inactive.

The purpose of providing the capability of inactivating gates is to speed up the simulation when a designer knows that a certain portion of the circuit is working correctly and wants to spend time simulating a separate portion of the same circuit. Gates which have been inactivated can be reactivated in any time during the run.

The SIMULATION SIGNAL LIST contains the current VALUE of the signal which was obtained by following the pointer to the device which was indicated as the signal source (by STATUS = 2). The pointer points not only to the device but also to the source pin on the device so that no computations must be made in extracting the signal value from the device. The current VALUE of the signal is then spread to all of the devices which use that signal as an input (indicated by STATUS = 1) by following the pointer to those devices.

Any signal may be driven by an external input at any time. When a signal is driven by an external input, the status flag of each connection of that signal is increased by +2 and a pointer to the input list entry is constructed. Connections which formerly were to the signal source (STATUS = 2) receive a new STATUS = 4. Connections which formerly were to device inputs (STATUS = 1) receive a new STATUS = 3. This scheme allows signals to be changed from free independent signals to signals driven by an external input and back again to free independent signals with a minimum of computation.

External inputs are caused by the following card

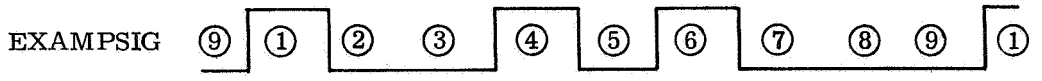


T INPUT SIGNALNAME (WAVESHAPE)

The SIGNAL NAME must be a valid signal in the logic file or a temporary signal created by PATCH explained below. The PERIOD specifies the number

of simulation time units over which the signal is periodic. (All inputs must be periodic but, as we shall see, this does not present a limitation.) The GRANULARITY specifies the number of simulation time units per entry in the WAVESHAPE field of the card. The WAVESHAPE is a field up to 32 characters long which, by punching 1's or 0's, specifies the waveshape of the periodic signal.

Consider the following signal which is desired to be used as an external input.



This signal could be described in several ways depending on the time scale desired:

a) T INPUT EXAMPSIG 0 9 1 100101000

after 9 time units the signal would become periodic and each 1 or ∅ in the waveshape lasts 1 time unit.

b) T INPUT EXAMPSIG 0 27 3 100101000

after 27 time units the signal would become periodic and each 1 or ∅ in the waveshape lasts 3 time units.

c) the same result as b) above could be obtained by the following cumbersome method but the signal would not be periodic.

```
T INPUT   Signal   0 9    3   100
D SWEEP   9
T INPUT   Signal   0 12   3   1010
D SWEEP   12
T INPUT   Signal   0 6    3   00
D SWEEP   6
```

The sweep cards cause the simulation to proceed the number of time units specified.

As each INPUT card is read, a check is made to see if the signal is already in the EXTERNAL INPUT LIST and a new entry is made if it is not. The PERIOD, GRANULARITY, and WAVESHAPE fields are then filled in for the entry. An explanation of how the EXTERNAL INPUT LIST signal values are maintained is postponed until the section on SIMULATION ALGORITHM.

External inputs may be removed at any time. This causes 2 to be subtracted from each STATUS entry in the SIMULATION SIGNAL LIST thus restoring the signal to its status before the input was applied. Inputs are removed by a card which specifies

T NOINPUT signal

The designer can cause signals to be traced by a card which specifies

T TRACE signal

and, conversely, cause a trace to be turned off by

T NOTRACE signal

Only signals which have a source (STATUS = 2 or 4) can be traced.

In addition to the control cards described above which directly affect the simulation logic file, there are director cards (D in column 1) and subdirector cards (S in column 1) which control the simulation timing and the form of the output, respectively.

There are four subdirector cards.

S YSIZE number

specifies the finished size of the output plot in inches.

S TSCALE number

specifies the number of simulation time units per horizontal inch of output.

S MAGNIFY number

magnifies the horizontal scale of any portion of the output.

S SNAPSHOT time

takes a snapshot of the file at the time specified.

There are four director cards.

D SIMULATE filename

This card must be first. It causes the specified file to be read in, constructs the necessary pointers, and initializes all signal values of unused inputs as required.

D SWEEP number of time units

This card causes simulation to proceed using all information previously made available by control cards described above. After the specified number of time units has elapsed, control cards are read in again until the next SWEEP card is encountered. There is no limit to the number of SWEEP cards in a run.

D RESTART

This card clears all input lists, trace lists, and reinitializes the file.

D ENDSIM

This card tells the program that simulation is complete and to begin output plotting.

2.4 Example (Westinghouse Circuit)

Presented below is an example of how a very elementary circuit might be drawn and simulated.

Figure 2-10 shows the input deck required to DRAW the schematic shown in Fig. 2-11. Notice that only three signal names appear at the output of the devices. These signals (CLOCKIN, MASTER, and SLAVE) were all DEFINEd as sources and have a STATUS = 2 in the logic file. The rest of the outputs of devices were all CONNECTed to themselves to indicate a source but that no signal name is to appear. These signals have a STATUS = 3 in the drawing logic file. The two signals ENABLE and RESET were not defined as sources anywhere (and in fact they are not intended to have sources on this drawing). They appear as labels only and have a STATUS = 1 in the drawing logic file. All other connections except those mentioned above with a STATUS = 1, 2, or 3 have, instead of a status word, a pointer to the list of interconnection points produced by subroutine DRAWLINES.

Figure 2-12 shows the input deck required to REVISE the drawing. The result of this run is shown in Fig. 2-13. Notice that the two devices MOVEd and DELETEd have caused the signals connected to them to be rerouted or deleted as required with no further information from the designer. One signal was REMOVEd. More devices could be ADDed or more signals connected, but this would have cluttered up the schematic which is now ready for simulation.

Figure 2-14 shows the input deck required for the simulation produced in Fig. 2-15 through 2-17. The input sequence in this example is a bit unusual from a logic designer's point of view, but it serves to demonstrate many of the functions available in program SIMSCHEMATIC.

```
D       DRAW      SCHEMATIC  OLLS01          OLLS     12 BY 10
S       PRODUCE   SCHEMATIC
S       SCALE TO  09 BY 07
S       NAME IT    DEMONSTRATION     CIRCUIT
G       ADD       DUM1              6          5
G       ADD       0A0               1          5          3AND
G       ADD       1A1               3          8          3NOR
G       ADD       1A2               3          6          3NOR
G       ADD       1A3               3          4          3NOR
G       ADD       1A4               1          2          3NOR
G       ADD       1B1               5          6          3NOR
G       ADD       1B2               5          4          3NOR
G       ADD       401               7          6.5        I/O
G       ADD       2C1               8          8          3NOR
G       ADD       2C2               8          7          3ORX
G       ADD       3F                10         8          3BUFF
W       DEFINE    CLOCKIN           0A0        0
W       DEFINE    MASTER            1A2        0
W       DEFINE    SLAVE             1B1        0
W       CONNECT   1A1        0      1A1        0
W       CONNECT   1A3        0      1A3        0
W       CONNECT   1A4        0      1A4        0
W       CONNECT   1B2        0      1B2        0
W       CONNECT   2C1        0      2C1        0
W       SIGNAL    ENABL             0A0        3
W       SIGNAL    CLOCK             0A0        1
W       SIGNAL    CLOCKIN           1A2        2
W       SIGNAL    CLOCKIN           1A3        2
W       SIGNAL    MASTER            1A1        3
W       SIGNAL    MASTER            1A3        1
W       SIGNAL    MASTER            1B1        2
W       SIGNAL    MASTER            2C1        3
W       SIGNAL    SLAVE             1A1        1
W       SIGNAL    SLAVE             1B2        1
W       SIGNAL    SLAVE             401
W       SIGNAL    RESET             1B2        3
W       SIGNAL    RESET             2C2        1
W       SIGNAL    SLAVE             2C1        2
W       SIGNAL    SLAVE             2C2        2
W       CONNECT   1A1        0      1A2        1
W       CONNECT   1A3        0      1A2        3
W       CONNECT   1A3        0      1A4        1
W       CONNECT   1A3        0      1B2        2
W       CONNECT   1A4        0      1A3        3
W       CONNECT   1B2        0      1B1        3
W       CONNECT   1B2        0      1A4        3
W       CONNECT   2C1        0      2C2        0
W       CONNECT   2C1        0      3F         2
D       PLOTNOW
```

Fig. 2-10

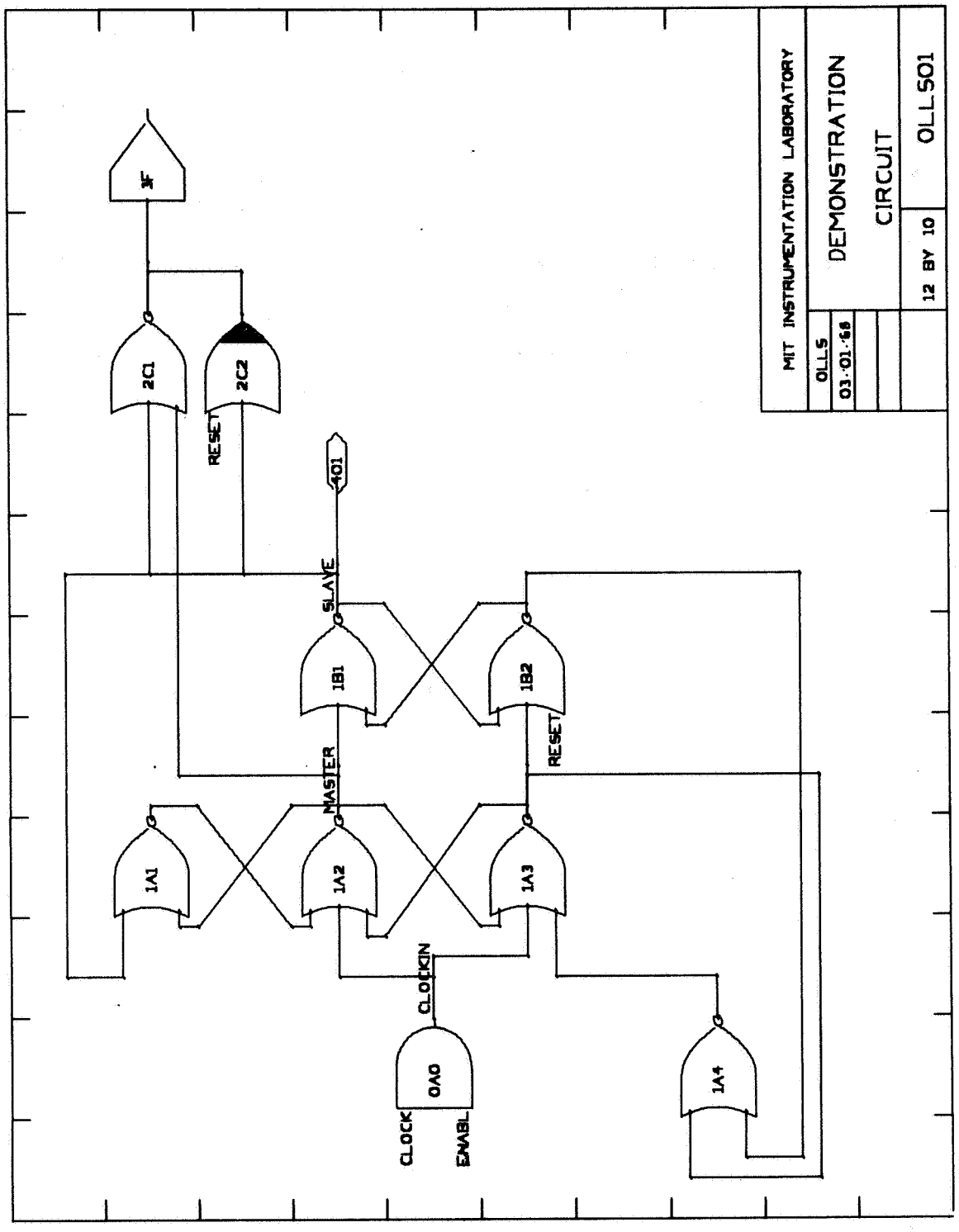

MIT INSTRUMENTATION LABORATORY
OLLS
03-01-68
DEMONSTRATION
CIRCUIT
12 BY 10
OLLS01
1F
2C1
2C2
RESET
401
SLAVE
1B1
1B2
MASTER
RESET
1A1
1A2
1A3
CLOCKIN
0A0
CLOCK
ENABL
1A4

Fig. 2-11

```
D       REVISE   SCHEMATIC  OLLS01          OLLS    12 BY 10
S       PRODUCE  SCHEMATIC
S       SCALE TO 09 BY 07
G       MOVE     1A4            3        2
G       DELETE   2C2
W       REMOVE   MASTER         2C1      3
D       PLOTNOW
```

Fig. 2-12

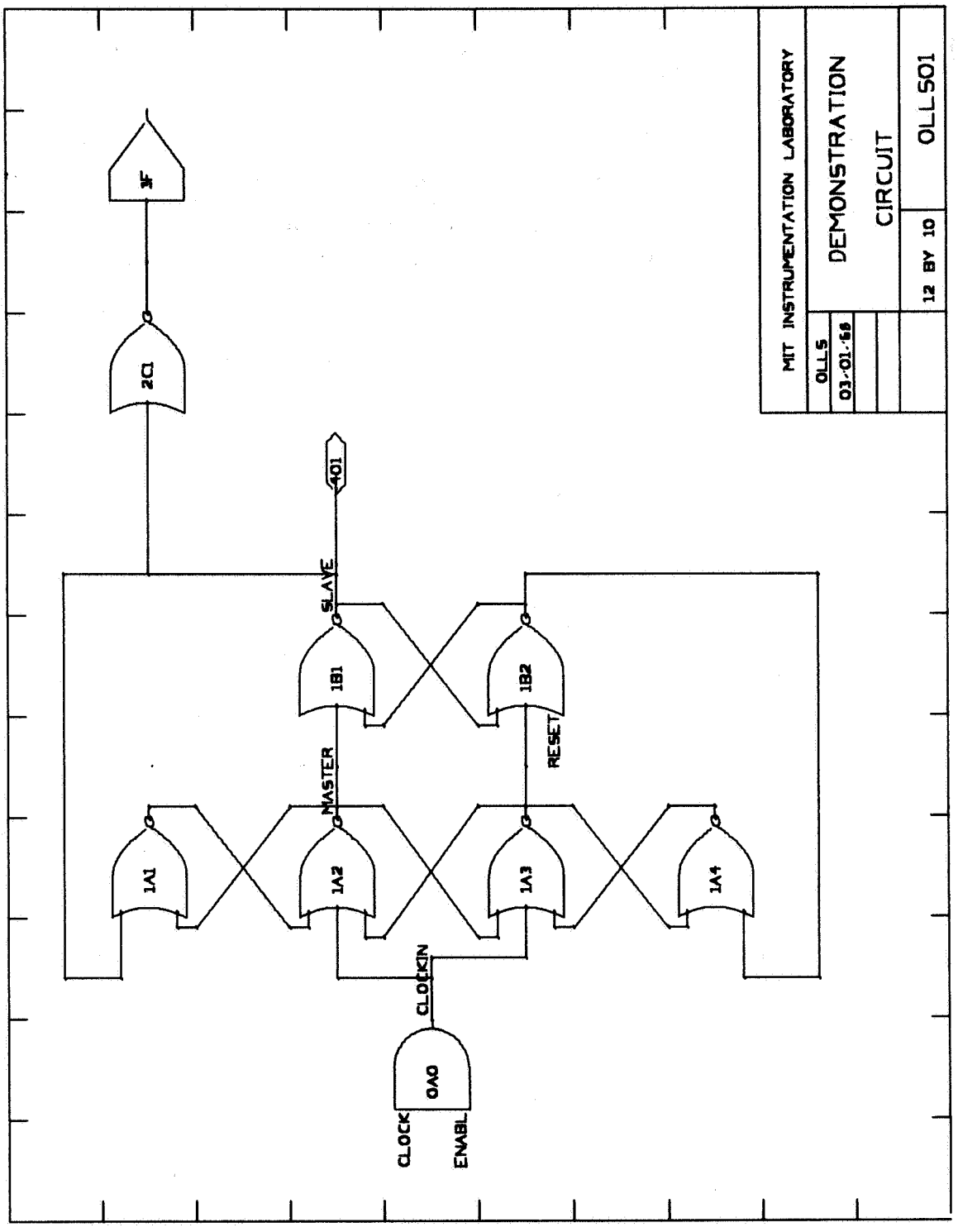

MIT INSTRUMENTATION LABORATORY
OLLS
03-01-68
DEMONSTRATION
CIRCUIT
12 BY 10
OLLS01
1F
2C1
401
SLAVE
1B1
1B2
MASTER
RESET
1A1
1A2
1A3
1A4
CLOCKIN
0A0
CLOCK
ENABL

Fig. 2-13

```
D       SIMULATE SCHEMATIC OLLS01
S       YSIZE    4
S       TSCALE   10
G       ALLGATES          1     ALL GATES ACTIVE WITH DELAY OF 1
T       TRACE    CLOCKIN
T       TRACE    MASTER
T       TRACE    1A1  0
T       TRACE    SLAVE
T       TRACE    2C1  0
T       INPUT    CLOCK    1 10   5    10
T       INPUT    RESET    1 16   8    10
D       SWEEP    10         TO INITIALIZE SLAVE FLIP-FLOP
T       NOINPUT  RESET
D       SWEEP    60
```

See Fig. 2-15

```
D       RESTART
S       TSCALE   50
G       ALLGATES          1     ALL GATES ACTIVE WITH DELAY OF 1
T       TRACE    CLOCKIN
T       TRACE    MASTER
T       TRACE    1A1  0
T       TRACE    SLAVE
T       TRACE    2C1  0
T       INPUT    CLOCK    1 10   5    10
T       INPUT    RESET    1 16   8    10
D       SWEEP    10         TO INITIALIZE
T       NOINPUT  RESET
D       SWEEP    80
W       PATCH    (1)             2C1       1
D       SWEEP    80
W       REMOVE   (1)             2C1       1
W       PATCH    1A1  0          2C1       1
D       SWEEP    80
W       REMOVE   1A1  0          2C1       1
W       PATCH    (0)             2C1       1
W       PATCH    CLOCKIN         2C1       3
D       SWEEP    80
W       REMOVE   CLOCKIN         2C1       3
```

See Fig. 2-16

```
D       RESTART
G       ALLGATES          1     ALL GATES ACTIVE WITH DELAY OF 1
T       TRACE    CLOCKIN
T       TRACE    MASTER
T       TRACE    1A1  0
T       TRACE    SLAVE
T       TRACE    2C1  0
T       INPUT    CLOCK    1 10   5    10
T       INPUT    RESET    1 16   8    10
D       SWEEP    10
T       NOINPUT  RESET
D       SWEEP    80
S       MAGNIFY  5
D       SWEEP    20
G       ACTIVE   1A1      3
G       ACTIVE   1B1      2
G       ACTIVE   2C1      5
S       MAGNIFY  1
D       SWEEP    80
S       MAGNIFY  5
D       SWEEP    20
D       ENDSIM
```

See Fig. 2-17

Fig. 2-14

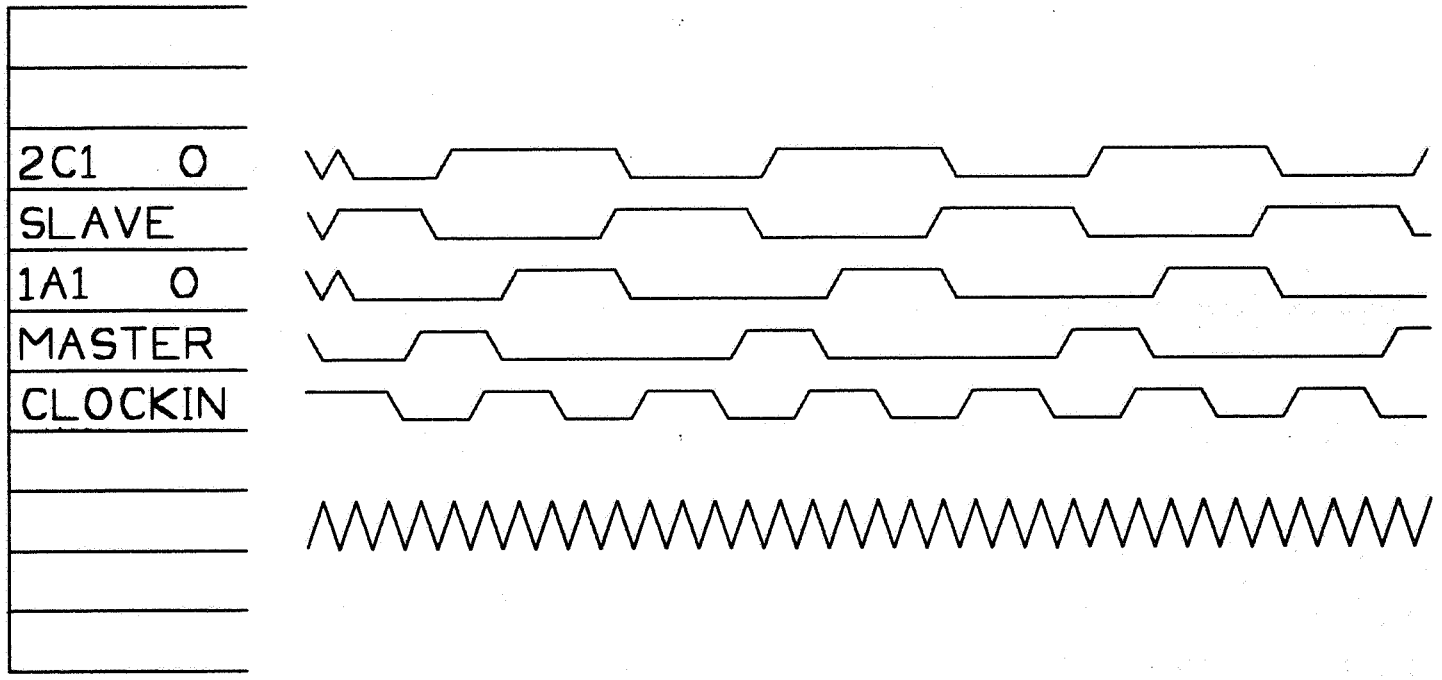
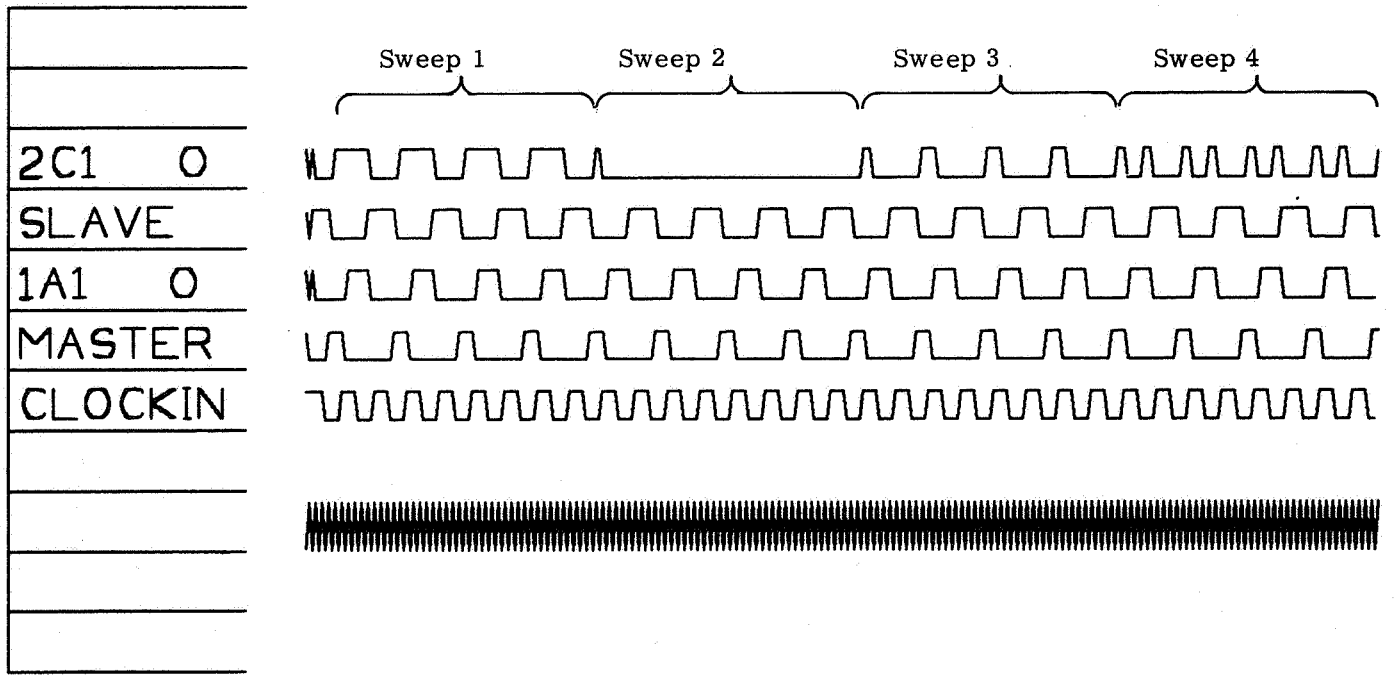


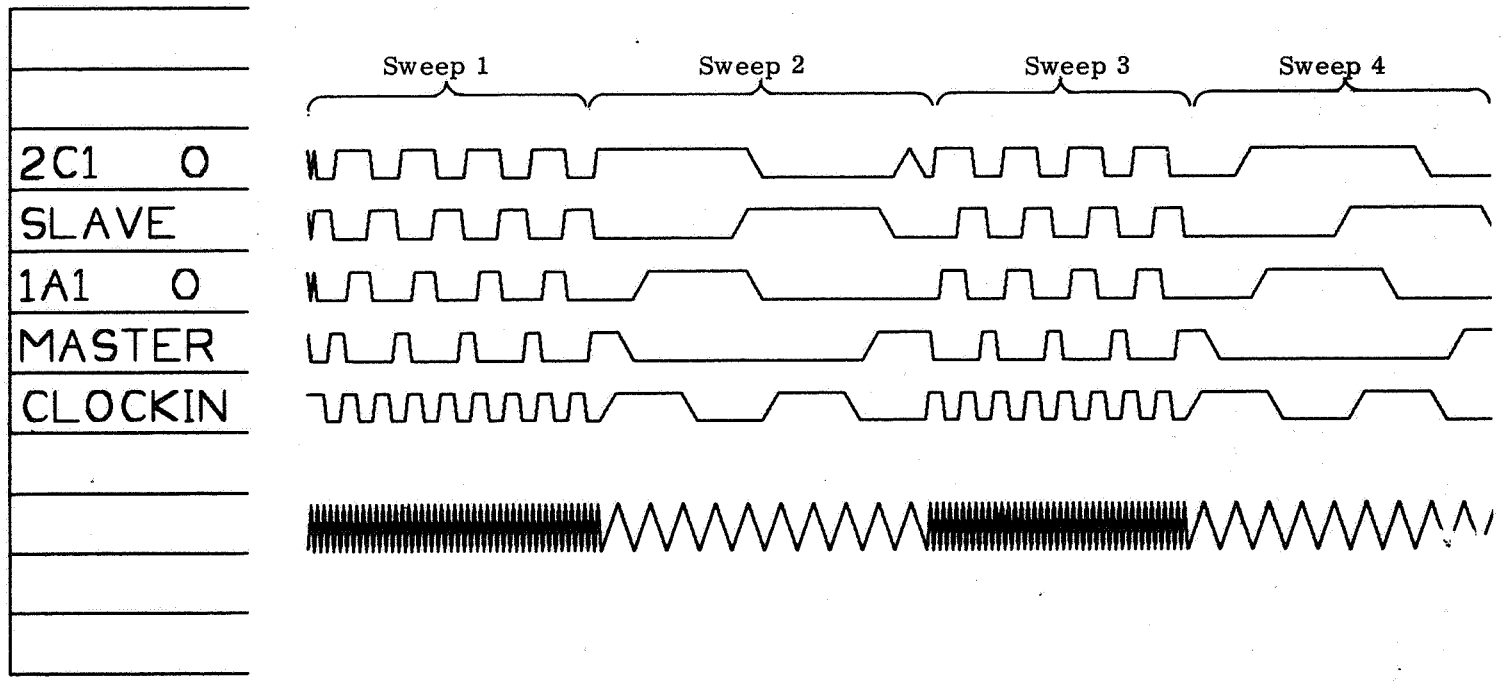
Fig. 2-15 Simulation of unmodified DEMONSTRATION CIRCUIT (Fig. 2-13).
This illustrates the basic simulation procedure
using "TRACE", "INPUT", "NOINPUT", and "SWEEP".



Sweep 1 shows the unmodified circuit performance
Sweep 2 shows the effect of "(1)" (logical one) PATCHed into 2C1 pin 1
Sweep 3 shows the effect of 1A1 pin Ø PATCHed into 2C1 pin 1
Sweep 4 shows the effect of CLOCKIN PATCHed into 2C1 pin 3

Note: "(0)" grounds the unused pin 1 on 2C1

Fig. 2-16 Illustrates the use of "PATCH" and "REMOVE" to make temporary changes to the logic during simulation.



Sweeps 1 + 2 show the circuit in operation with all gates having a delay value of 1.

Sweeps 3 + 4 show the circuit in operation with the delays of gates 1A1, 1B1, and 2C1 changed to 3, 2, and 5 respectively.

Fig. 2-17 Illustrates the use of "MAGNIFY" and "ACTIVE".

2.5 Drawing Interconnection Algorithm

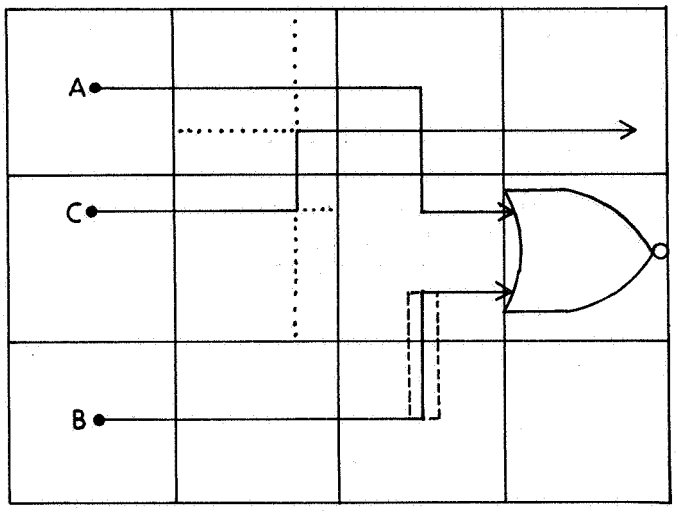
One very important product of H-1800 System has been our experience with the drawing interconnection algorithm used by subroutine DRAWLINES to produce accurate schematics which are fairly pleasing to the logic designer's eye.

Since the goal of any circuit routing algorithm is to combine a very limited set of input parameters into something which is measured by purely human standards, any algorithm will naturally have a great many checks and balances or fine tuning adjustments which can only be described and set by observing a great many examples and readjusting the algorithm as necessary.

2.5.1 General Procedure

The input to DRAWLINES consists of the X, Y coordinates of the pin on the device which is the source of a signal and a list of X, Y coordinates of the pin on each device which is a connection of that signal. DRAWLINES works on each connection (one at a time) trying to find the "best possible" route around obstacles such as devices, crowded areas of the schematic, or areas reserved by the designer. To aid in this task, DRAWLINES maintains several maps which describe in detail the layout of the schematic. When all connections of a particular signal have been successfully routed, the various maps used are updated with the new information, the interconnection points are written into the logic file as described above, and control is returned to the main program DRAWSCHEMATIC.

To keep storage requirements reasonable, a drawing is mapped as a checkerboard on one-inch squares, and each square contains information about 10 horizontal lines and 10 vertical lines through it. For a 50 x 30-inch schematic only 1500 words each are required for the horizontal and vertical maps. When a connection has been routed completely or partially through any square, the line occupied is marked "filled" on the map and no other connection may be routed through that square on that particular line. Thus connections may cross other connections anywhere at right angles, but no two colinear connections may ever use the same line in the same square at any time. It should be noted that the connection of a signal to one gate does not in any way affect or impair connections of the same signal to any other gates, since the maps are only updated after all connections of any one signal name are routed. In the example below, if the connection from A were made first, the connection from B as shown could not occur since segments of different signals would occupy the same square at the same vertical line number.



The one vertical segment of the connection from B would instead be routed at least 1/10-inch either to the right or to the left of its present position as indicated by the dashed lines. The connection from C is permissible, and the extent of the line segments which are reserved for C and forbidden to any other signal is shown again by the dotted lines.

Devices as well as segments of connections cause lines to be reserved, and in the above example the gate shown reserved all 10 of the possible lines through the square it occupies in both the vertical and the horizontal map.

More maps:

The qualities which we considered most important in finding the "best possible" route for a connection were long straight runs and minimum number of line segments in the run. In order to encourage long straight lines, four 50 x 30-square maps were maintained in addition to the layout maps described above. The additional "look-ahead" maps, as they are called, contain information in each square which describes the map index of the farthest square which may be reached before an obstacle is encountered in the direction (up, down, left, or right depending on which of the four maps is being read). An obstacle may be either a device or a square on the layout map which has all 10 lines filled with the routed connections.

Figure 2-18 shows the four look-ahead maps superimposed. The key shows which map contains the entry shown in each square. For example, if the algorithm found itself at square (3,1), it would know by asking the appropriate map that a straight-line path exists up as far as (3,3), down as far as (3,0), right as far as (5,1), and left as far as (1,1).

The algorithm proceeds to route connections one-at-a-time, using the maps described above, in three general phases:

a) Phase I tentatively constructs up to seven unique paths emanating from the source device and up to seven unique paths emanating backwards from

KEY:

	up	
left		right
	down	

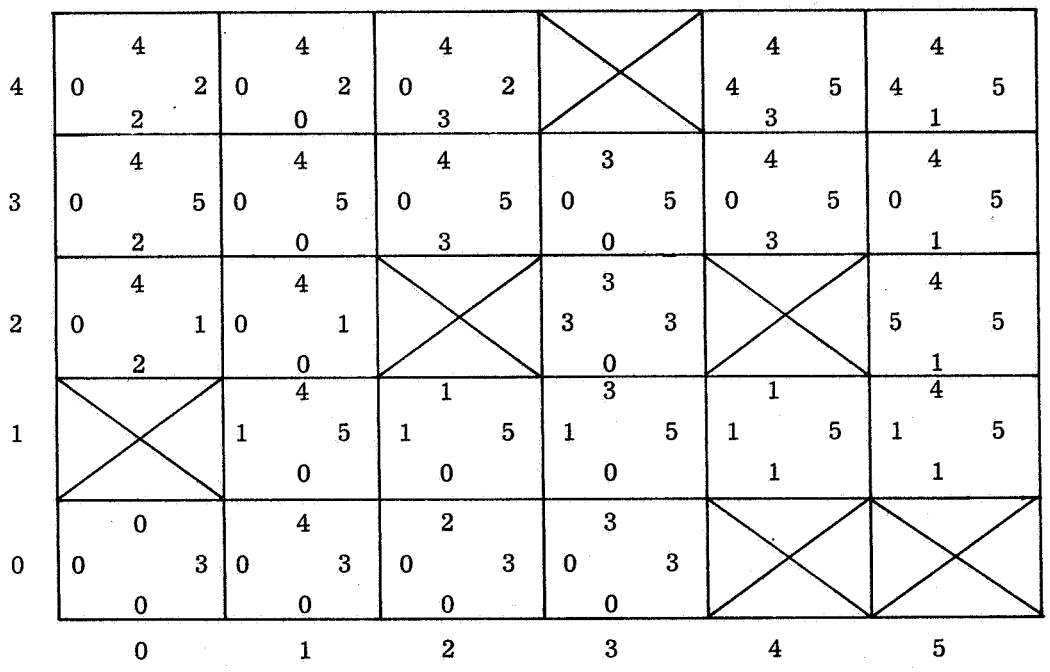


Fig. 2-18

The four look-ahead maps would be constructed as shown for a 5 × 3 square drawing with the six obstacles shown.

the destination device. No path has more than nine segments and the segments are all found on the look-ahead maps; very little attempt is made here to connect the source to the destination in either direction. Each forward line is then checked with every backward line looking for intersections. All possible combinations are tried and the combination with the fewest number of segments is saved for each forward line. The program never fails to find between one and seven such connections.

b) Phase II takes a more detailed look at the connections found in Phase I. It first checks each connection for loops on itself and removes any it finds. It then checks each connection for possible shortcuts from one point in its path to another point in its path and takes any shortcut it finds. These two steps use only the look-ahead maps with no regard to the detail on the layout maps. It then measures the length of each connection, counts the number of segments in each, and picks one "best" connection based on these two numbers.

c) Phase III works out the detailed position of each line segment in the connection selected by Phase II, using the layout maps which indicate to the nearest 1/10-inch which lines in the squares along the path are unoccupied. This phase contains algorithms for jogging around minor obstacles within a particular square; but, if the jogging becomes excessive, Phase III gives up in disgust and asks Phase II for its second-best connection.

2.5.2 Phase I in greater detail:

A test is made to detect flip-flop-like diagonal connections before main routing techniques are applied. If two devices of the same type are at the same horizontal coordinate and two squares apart on the vertical, and if the output of the upper gate is connected to the upper input of the lower gate or the output of the lower gate is connected to the lower input of the upper gate, the connection is made flip-flop style. Phase II and III are bypassed. If no flip-flop connection is called for, the program begins drawing forward paths and backward paths as mentioned above.

Figure 2-19 shows seven forward paths which might tentatively be taken by Phase I using the look-ahead maps of Fig. 2-18. Two things are worthy of note here: First, the algorithm had choices at squares (1,1), (1,0), (4,3), (0,3), and (1,4). Whenever it encounters a choice for the first time, it decides in favor of the direction of the goal and places a flag in a list to remind itself that if it still hasn't found seven unique paths, it should go back and try the other direction. Secondly, we have found that nine segments and seven lines in each direction are more than sufficient even on very large drawings. It is not necessary that a forward path find the destination or that a backward path find the source; it is only necessary that the paths cross somewhere.

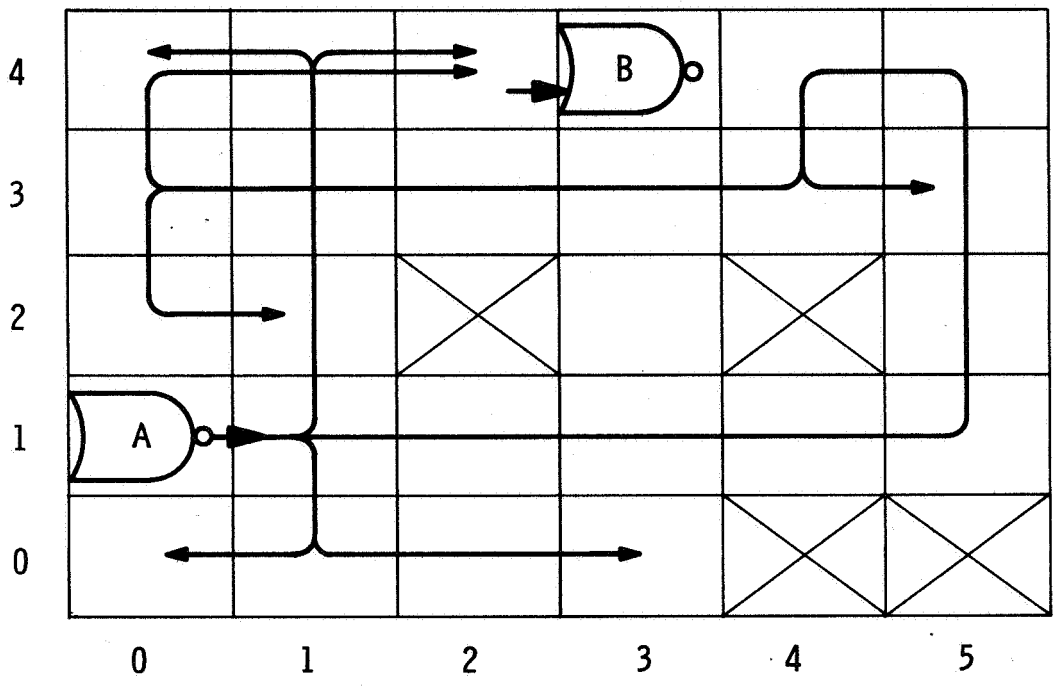


Fig. 2-19

Shows forward paths which Phase I might produce using the look-ahead maps of Fig. 2-18.

Phase I is prepared to recognize when it somehow has led itself into a dead end, such as the left branch taken in the decision at square (1,0) on Fig. 2.19. It also recognizes when it is retracing itself in a loop, such as the potential decision at square (5,3). In both cases it stops trying and proceeds to the next path, or goes back to the last-flagged decision and takes the opposite alternative to the decision made when the flag was placed.

The backward paths from the destination at the input of device B are not shown, but it is obvious that many intersections exist. Figure 2-20 shows two tentative connections which would be typical of what Phase I might decide after testing all forward/backward intersection combinations.

2.5.3 Phase II

Phase II can usually make significant improvements on the connections produced by Phase I. As shown by the dashed lines in Fig. 2-21, Phase II does find shortcuts in both paths; and, in fact, both paths reduce to the same final path. The path selected by Phase II in this case measures 6-unit inches and has 3 segments. These two numbers would be used in making the decision if there were more than one candidate for the connection. In practice we have found that Phase II is usually able to reduce all connections made by Phase I to a single path. In those cases where a single path is not the result, most logic designers would usually consider both (or all) the paths suggested as acceptable.

2.5.4 Phase III

Phase I and Phase II used only the look-ahead maps for directions and, as a result, the path selected by Phase II is known only to the nearest inch in any square. Phase III now takes over and, using the detailed layout maps, selects the final detailed line through each of the squares in the path. The routine tries to pick a line which requires no jogging. Crowded conditions and poor layout on the part of the designer sometimes force minor jogging which can be straightened out in a later REVISION of the schematic. If program complexity were measured in pounds of punched cards, Phase III would be very complex compared to Phase I and II. But such is not the case. Phase III is a very unprofound routine to clean up all unfinished details.

2.5.5 An Appraisal

Among the things we learned in experimenting with various other interconnection algorithms before we arrived at the above description was that the initial overhead required to setup and maintain the look-ahead maps was well worth the storage and time spent. Those maps do indeed encourage long straight lines and they are very fast to operate. The concept of the look-ahead

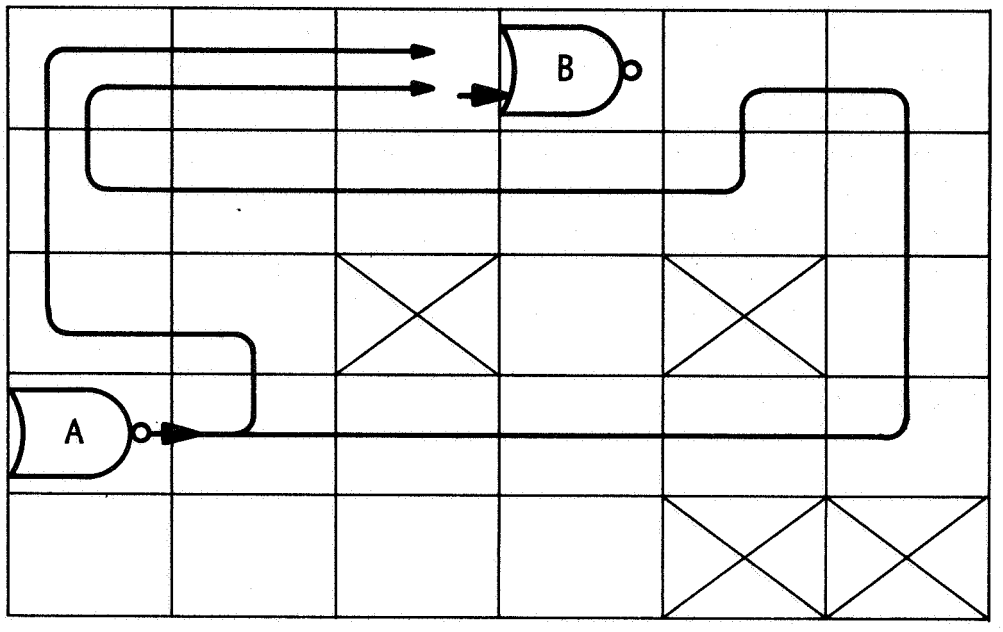


Fig. 2-20 Two Tentative Connections Produced by Phase I.

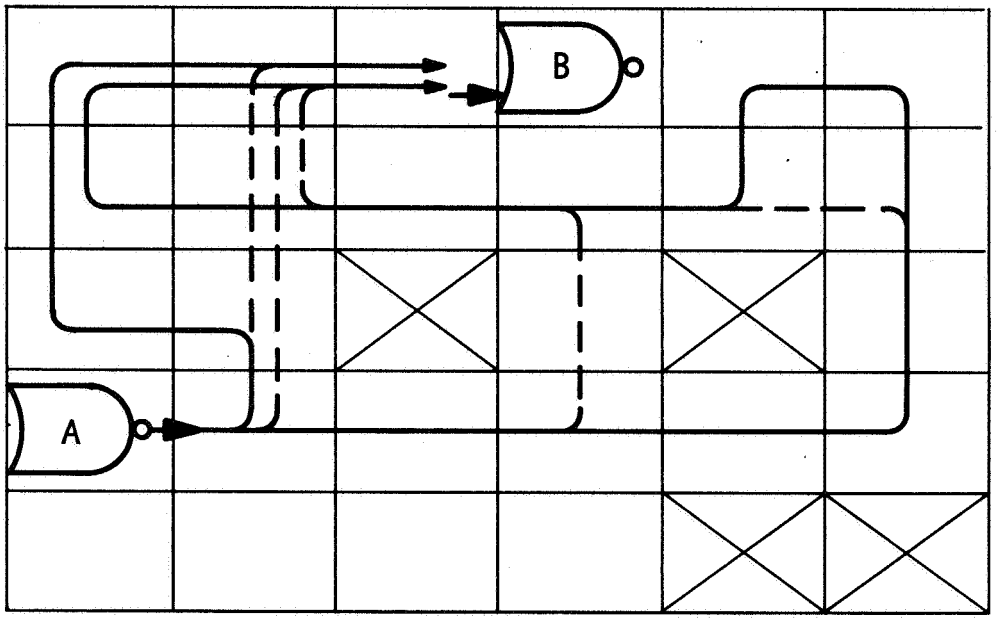
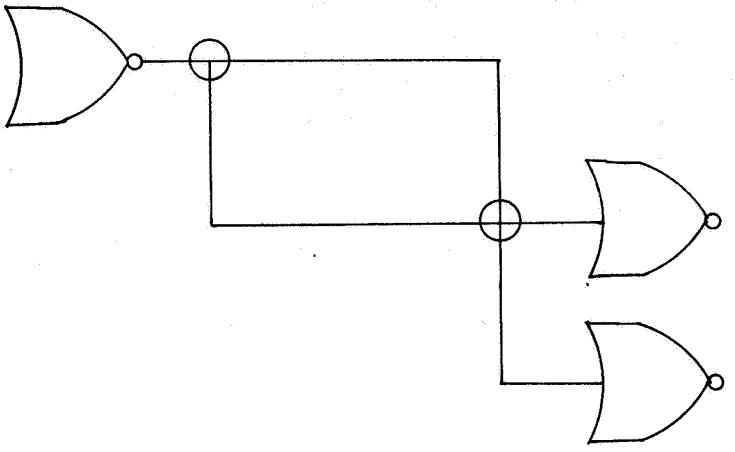


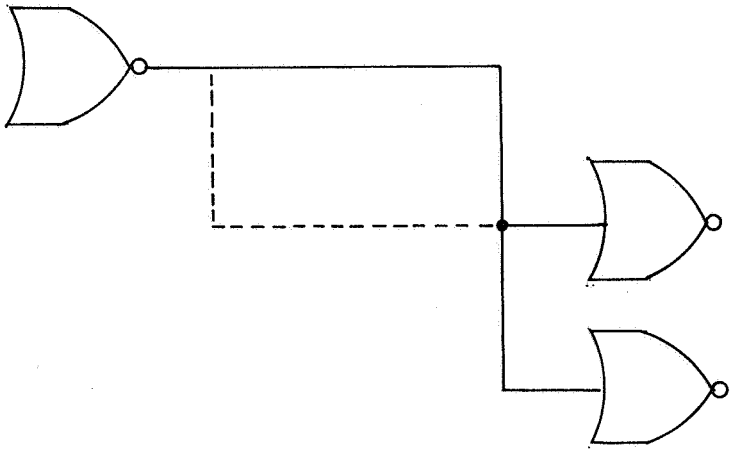
Fig. 2-21 Shortcuts which might be found by Phase II.

maps is not limited to the type of algorithm discussed above and we intend to put it to use in the 360 system.

Probably what the H-1800 drawing algorithm needs most is a Phase IV which can take over after previous phases have selected tentative routes for all the connections of one signal. It should then adjust some connections to perhaps make use of another connection of the same signal. Shown below is a typical example of how two interconnection routes which are perfectly acceptable if taken alone are unacceptable together.



Phase IV should recognize that the circled intersections are the same signal and connect one of them with a dot and eliminate two line segments. It should be smart enough to know that the combination shown below has fewer segments than the other alternative (shown in dotted lines).



2.6 Simulation Algorithm

Unlike the drawing interconnection algorithm, the simulation algorithm is quite simple. The card reading routines were described above. Cards are read in and the simulation logic file is modified as requested until a SWEEP card causes the routine in Fig. 2-22 to assume control, or until an ENDSIM card causes output plotting to begin.

As far as it was designed, the simulator provides satisfactory results, but it does not go far enough to be generally useful as a design tool. There is not enough flexibility in the kind or amount of output produced. There is no diagnostic facility as described in the 360 system. The designer is not free to define or modify the devices available to him (with the important exception of gate delays).

This algorithm differs from the 360 system algorithm in only one very major respect. The H-1800 algorithm evaluates the logic equation for every device once every time increment, regardless of whether or not the input signal values have changed. Due to the logic file organization chosen, the simulation would run slower if the inputs of each device were checked for changes and a decision to bypass satisfying the logic equations were made on the result of the check. In the 360 system deliberate checking of device inputs is not required. We have reached no definite conclusions about this difference in algorithm yet.

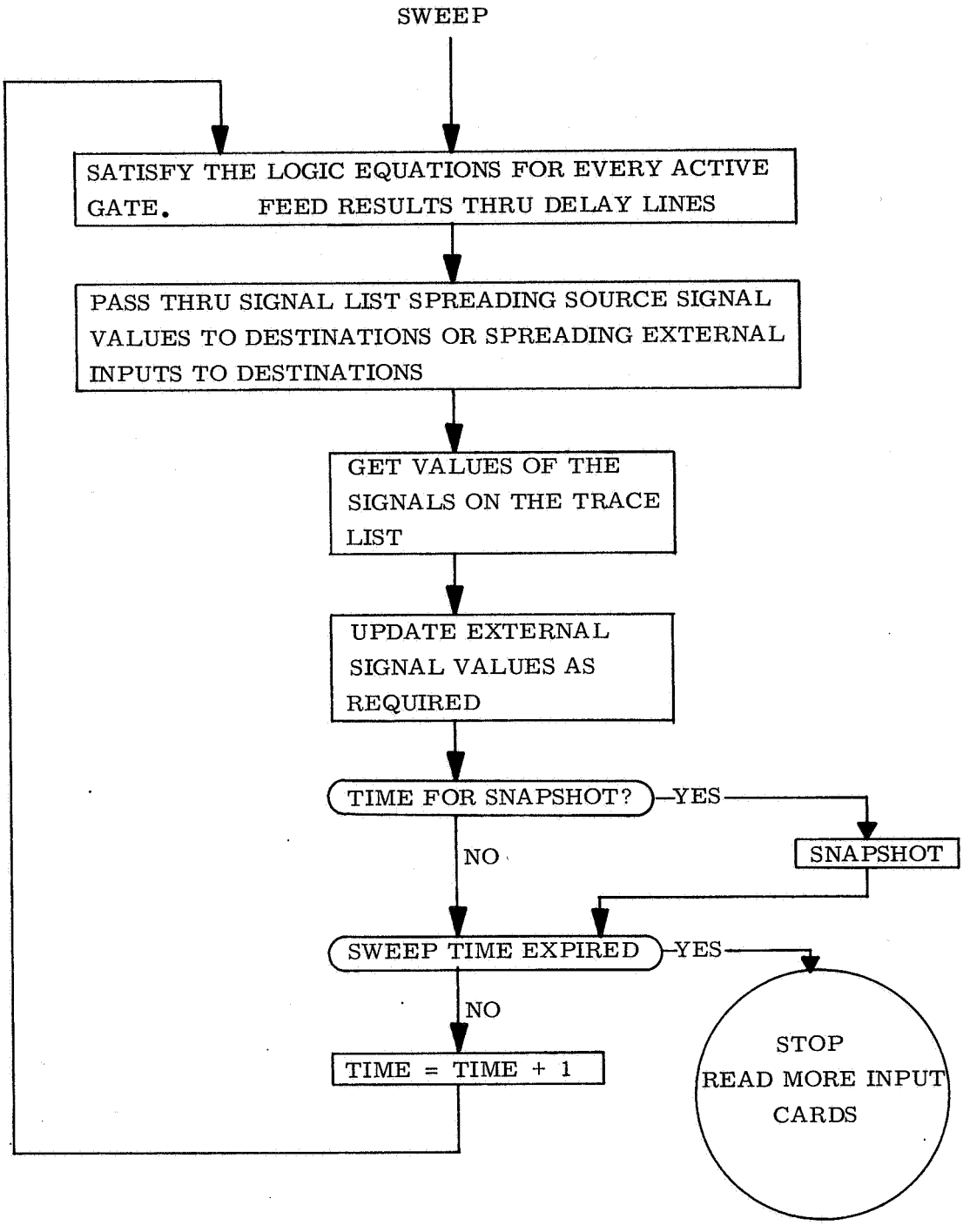

SWEEP
SATISFY THE LOGIC EQUATIONS FOR EVERY ACTIVE GATE. FEED RESULTS THRU DELAY LINES
PASS THRU SIGNAL LIST SPREADING SOURCE SIGNAL VALUES TO DESTINATIONS OR SPREADING EXTERNAL INPUTS TO DESTINATIONS
GET VALUES OF THE SIGNALS ON THE TRACE LIST
UPDATE EXTERNAL SIGNAL VALUES AS REQUIRED
TIME FOR SNAPSHOT?
YES
NO
SNAPSHOT
SWEEP TIME EXPIRED
YES
NO
TIME = TIME + 1
STOP
READ MORE INPUT CARDS

Fig. 2-22

Section 3

OLLS/360

3.1 Data Structure

James Pennypacker

The data structure is a complex organization of data items and pointers to the data. The structure is list-oriented and incorporates features that are designed to take full advantage of FILLIP. One overriding consideration in the development of the data structure was the requirement for rapid insertion, retrieval, and manipulation of data. A second governing requirement was that the data structure must be expandable and flexible in handling data of different attributes.

The data structure is far too complex to be presented meaningfully in a single diagram; the approach taken in this section is to present individual sections of the structure in detail and then attempt to show how the various sections are integrated into the whole system. The descriptions will be most easily understood if reference is made to the appropriate figures. It should be noted that details of the structure are subject to change, for the data file is a continuously evolving structure. To fully understand the implications of the data structure, some knowledge of FILLIP is assumed.

3.1.1 Binary Tree

Two basic types of structures are found throughout the data structure: the "binary tree" and the "196 structure".

The binary tree is a structure in which the individual data items are not organized as a linear list. The tree consists of nodes which contain the data and which are interconnected by pointers. A node is illustrated in Fig. 3.1a. From

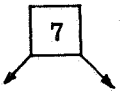


Fig. 3-1a

each node, a left-hand pointer and a right-hand pointer point to two different nodes, both of which are one level lower in structure than the node which points to them. To insert a data item in the tree, the data is compared with that of the top node of the tree. If the data to be inserted is smaller in value than that of

the top node, the left-hand pointer of the top node is followed and the node to which it points is now used in the comparison. If the data to be inserted is greater in value than that of a node, the right-hand pointer is followed. The data is always inserted at the bottom of the tree at the point found on the basis of the comparisons.

To illustrate the concepts of the binary tree, assume the sequence of numbers 7, 3, 5, 1, 9, 4, 13, 11 is to be organized into a binary tree. The first number, 7, becomes the top node of the tree as shown in Fig. 3-1a. The next entry, 3, is compared with 7. Because it is less than 7, it becomes the node pointed to by the left-hand pointer emanating from node 7, as shown in Fig. 3-1b.

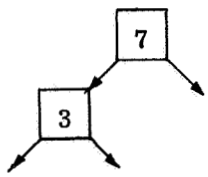


Fig. 3-1b

The next entry, 5, is compared first with node 7. Being less than node 7, it is then compared with node 3. Being greater than node 3, it is inserted as shown in Fig. 3-1c.

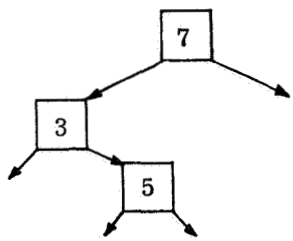


Fig. 3-1c

The next entry, 1, is similarly compared against node 7 and node 3. Because it is less than 3, it is inserted as shown in Fig. 3-1d.

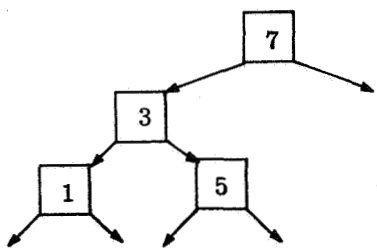


Fig. 3-1d

The next entry, 9, is greater than 7 and hence is inserted as shown in Fig. 3-1e.

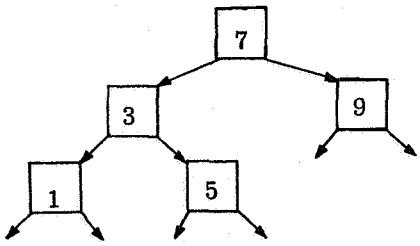


Fig. 3-1e

The remaining items of the list are inserted in a similar fashion, resulting in the tree shown in Fig. 3-1f.

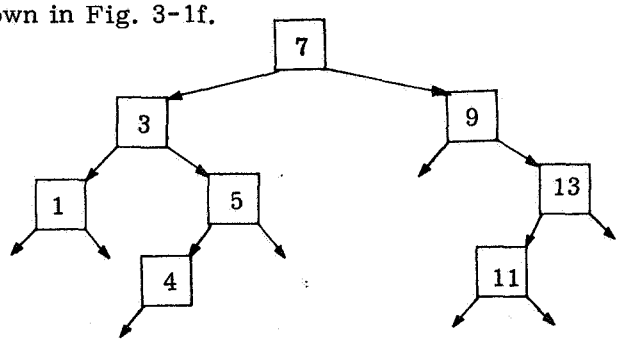


Fig. 3-1f

The primary advantage of the binary-tree structure is that a large amount of data can be searched rapidly for a particular item. In an ideal tree, the number of comparisons in such a search is a logarithmic function of the number of nodes, rather than a linear function required for the typical linear list. A second advantage of the tree is that order is introduced into a set of random data. In Fig. 3-1f, a scanning of the nodes from left-to-right produces the ordered list of data. Removal of a node from the tree is more difficult than from a linear list but is a straightforward process, especially if the binary tree is modified slightly as is actually done in the data-file structure.

To be optimally useful, the binary tree must be created by adding nodes of random value; if, for example, the data is entered in order of value, the binary tree degenerates into a linear list and the advantage of quick retrieval of data is lost. Furthermore, every deletion of a node tends to linearize the remaining tree structure; however, the tree structure can never become worse in terms of retrieval time than a linear list. The impact of the binary-tree structure is so great in a system in which a large amount of data is stored that a single FILLIP instruction was designed to search the tree for a requested piece of data.

In the data-file structure, there are many independent binary trees. Every node on the tree is a FILLIP data cell with a standardized definition of

the first four subfields. A typical node is shown in Fig. 3-2.

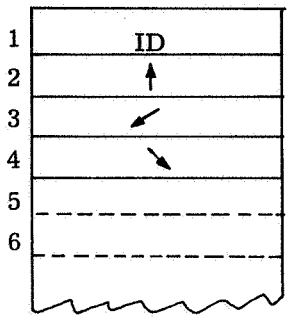


Fig. 3-2

The first subfield of a node contains the ID of value of the node. Thus, the numbers shown on the nodes of Fig. 3-1f are actually contained in subfield of data cells. Subfields 3 and 4 contain the left-hand and right-hand pointers from each node; each pointer points to another node. When a subsequent node does not exist, that is, when the bottom of the tree is reached, the associated pointer is NIL. Subfield 2 contains an upward pointer which points to the previous node in the tree; in Fig. 3-1f, subfield 2 of node 9 contains a pointer to node 7 as does subfield 2 of node 3. This upward pointer facilitates deleting a node from the tree.

The pattern of pointers in subfields 2, 3 and 4, which is illustrated in Fig. 3-2, is always to be understood as representing a binary-tree structure and any data cell containing such a pattern is understood to be a node on the tree. Note that subfields 5 through 14 of the data cell may contain other data which is associated with the node of the tree; specifically, they may contain pointers to data cells which are nodes on other binary trees, resulting in an interleaving of the trees.

3.1.2 196 Structure

The second basic structure which appears throughout the data file is the "196 Structure". The structure consists of a data cell of fourteen subfields, each of which contains a pointer. Each pointer points to a separate data cell of fourteen subfields, each of which in turn contains a pointer. At this point there are 196 independent pointers. Each pointer points to a data cell of the same type but containing different data. The "196 structure" is illustrated in Fig. 3-3a but only fourteen of the ultimate 196 pointers are shown. The 196

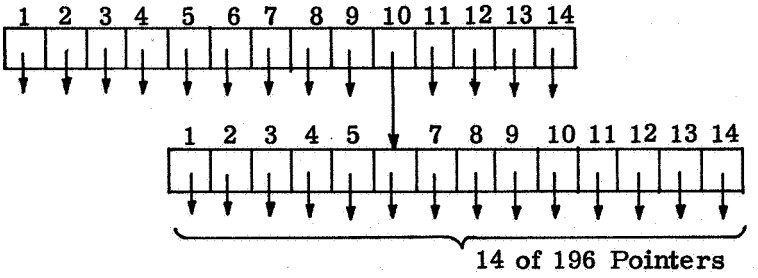


Fig. 3-3a

structure as represented in the data file is illustrated in Fig. 3-3b; this pattern, when it appears, is understood to represent the complete structure illustrated in Fig. 3-3a. The 196 structure is the reason for the limitation on the number

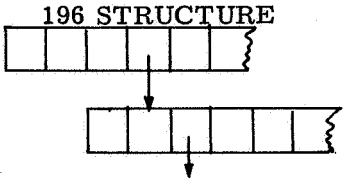


Fig. 3-3b

of terminals of a device which was mentioned earlier. By making the structure of three levels instead of two, up to $14^3 = 2744$ terminals could be handled.

3.1.3 Classifications of Data

The data structure as currently defined is designed to handle data of four major classifications; additional classes of data can be added as future requirements dictate. After a general introduction, each class of data will be described in detail.

The first class of data includes all information relating to signals or interconnections between the individual components. Included in the classification is information required for the logical simulation of devices, such as signal history of logic levels, signal-load factors, and simulation times. The information pertaining to signals is organized in a binary-tree structure in which each node is called a signal-head cell.

The second major classification of data includes the complete specification of every different type of logical element which is used in any specific design. Included in this set of data are the name of the device type, the number of each type of terminal (e.g., inputs, outputs) names for each terminal, specification of terminal-logic behavior, the shape of the device, and additional information for simulation purposes. The information concerning each

individual type is contained in a glossary structure, which is essentially a 196 structure. The glossaries in turn are organized into a 196 structure; thus, up to 196 different types of devices can be utilized in any particular design.

The third major classification of data pertains to the individual instances of each type of element. All the detailed information about a particular instance is contained in the instance structure, which is a hybrid structure consisting of a partial 196 structure and FILLIP data cells. Typical data stored in this structure identify which signals are connected to the terminals of the device, the drawing number specifying on which drawing the device is to be found, drawing coordinates, and the identification of the device. The hybrid-instance structure is associated with two independent binary trees, as will be explained shortly.

The final classification of data which has so far been identified includes all the graphic information required for CRT display and hard-copy output. This data is not confined to one structure but is rather inter-related with the rest of the data file. One structure which does occur, however, includes data relevent to physical drawings. This structure is a binary tree whose nodes are drawing-head cells. Data stored in the drawing-head cell includes the drawing number, drawing size, signature information, scale size, instances, and signals which appear on the drawing.

3.1.4 Instance Structure

The first structure which must be understood is the instance structure. There is one instance structure (or cell) for every individual logical element maintained in the data base. The purpose of the instance cells is to show which signals appear on the terminals of the instance and where the instance is located; the location information is specified by drawing number and coordinates. This information is the only information currently used which is unique to each individual device.

The instance structure and the associated instance-drawing structure are illustrated in Fig. 3-4. In this and subsequent drawings, the letter P and arrows are both used to represent direct FILLIP pointers. The instance cell itself is a FILLIP data cell consisting of eleven subfields. The first four subfields indicate that the instance structures are organized as a binary tree; there is, in fact, one tree for each type of logical element and the instances of each type represent the nodes of the type tree. Subfields 5 through 9 are pointers to terminal structures and will be discussed shortly; it is sufficient at this time to say that the terminal structure contains pointers to signals which appear on the respective terminals of the instance. Subfield 10 contains a pointer to a FILLIP data cell of 8 subfields which contain graphic information pertaining to the instance; this data cell is called the instance-drawing structure.

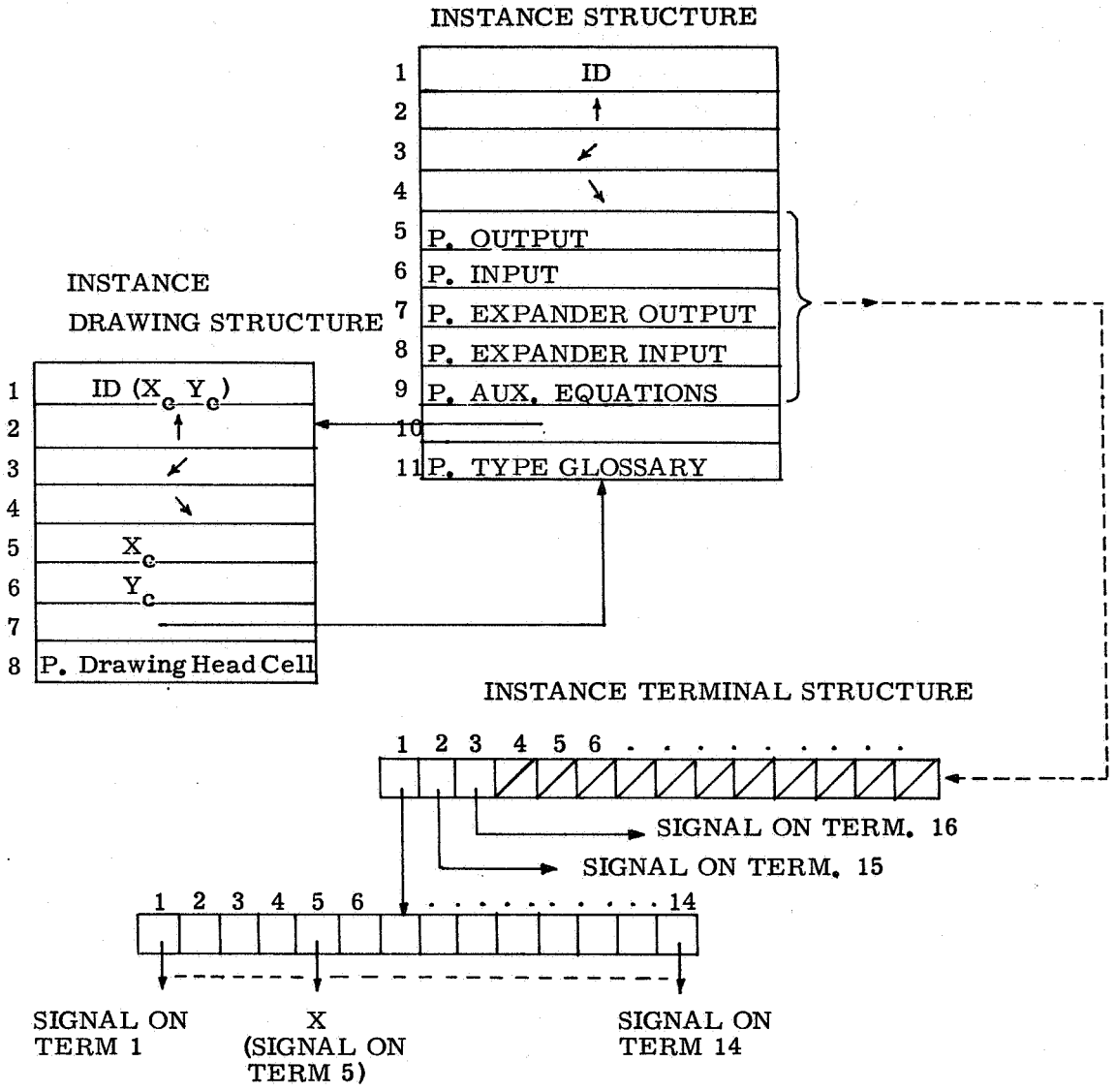

INSTANCE STRUCTURE
1 ID
2
3
4
5 P. OUTPUT
6 P. INPUT
7 P. EXPANDER OUTPUT
8 P. EXPANDER INPUT
9 P. AUX. EQUATIONS
10
11 P. TYPE GLOSSARY
INSTANCE
DRAWING STRUCTURE
1 ID (X_c Y_c)
2
3
4
5 X_c
6 Y_c
7
8 P. Drawing Head Cell
INSTANCE TERMINAL STRUCTURE
1 2 3 4 5 6 14
SIGNAL ON TERM. 16
SIGNAL ON TERM. 15
SIGNAL ON TERM 1
X (SIGNAL ON TERM 5)
SIGNAL ON TERM 14

Fig. 3-4

Subfield 11 of the instance cell contains a pointer to the glossary which defines the type of this instance.

The instance-drawing structure is also organized into a binary tree, with one tree for each different drawing. The identification (subfield 1) of the instance-drawing structure is a function of the graphic coordinates, making it possible to retrieve information about an instance which is specified by drawing number and coordinate, as well as by type and ID of element. Subfields 5 and 6 contain the x and y coordinates of the instance. Subfield 7 contains a pointer to the instance structure. Subfield 8 contains a pointer to the head cell of the drawing on which this particular instance appears.

One of the requirements in the design of OLLS was that the user be able to define his own logical devices. In practice, this means that different types of elements will have different numbers of input terminals, output terminals, etc., but that all instances of the same type will have the same number of inputs, the same number of outputs, and so on. Up to 196 terminals of each class (including auxiliary equations) can be accommodated. The instance structure must thus be capable of pointing up to 5 x 196 = 980 different signal-head cells, one signal-head cell for each terminal of the device. One way to realize this capability would be to set up a 196 structure for each class of terminals, with a one-to-one correspondence between pointers and terminals. Each pointer would then point to the head cell of the signal which is connected to the corresponding terminal. In other words, the pointers in subfields 5 through 9 of the instance structure could each point to a separate 196 structure.

Such a structure, however, is wasteful of storage; an instance with only one useful output would result in a structure possessing 195 unused pointers; this waste of storage occurs for every instance of the logic type.

To overcome this problem, a flexible instance-terminal structure is utilized. For convenience, only one class (outputs) of terminals will be discussed; the structure design is identical for the other classes of terminals. If the instance contains only one output, subfield 5 of the instance structure contains a pointer directly to the head cell of the signal which appears on that terminal. If the instance contains more than one but fewer than fifteen output terminals, subfield 5 of the instance structure contains a pointer to a FILLIP data cell of fourteen pointer subfields; these pointers point the head cells of the signals which appear on the respective terminals. If the instance cell contains more than 14 but fewer than 28 output terminals, the situation is as shown in Fig. 3-4. Subfield 5 of the instance structure points to a data cell (call it cell Z) of fourteen pointers. The first of the pointers in cell Z points to another data cell of 14 pointers which point to the head cells of signals on the first fourteen terminals. The second pointer of cell Z points to the signal-head

cell for the fifteenth terminal, the third pointer of cell Z points to the signal-head cell for the sixteenth terminal, and so on until the terminals are exhausted. (Figure 3-4 illustrates the case of 16 terminals). The instance-terminal structure shown in Fig. 3-4 is expanded by pointing from cell Z to additional cells of 14 pointers as needed. Only when a device has more than 182 terminals of the same class does a true 196 structure result.

3.1.5 Glossary

The flexible structure just described raises one important question: given a pointer to an instance, how does one locate the signal on the n^{th} terminal, since, in general, the location of the pointer to the desired signal depends upon the total number of terminals of the particular class? The answer to this question is that one must use a glossary to interpret the instance structure for each particular type.

The glossary is a complex structure which, at least indirectly, contains the complete description of a logic device; there is one glossary per type of element. Provision is made for including 196 glossaries in each data file. The glossary enables all instance structures to be treated identically by the program routines, even though instance structures of different types are not identical. The structure of the glossary is illustrated in Fig. 3-5.

The entry point or root of the glossary is a FILLIP data cell of 14 pointers. The first pointer points to a data cell which contains basic alphameric information about the type, such as type name.

The second pointer of the glossary root points to the top of the tree of instances of this type; each node on the tree is an instance structure with a unique instance ID. Thus, all instances of the same type are grouped together on a tree which may be addressed through the type glossary.

The third and fourth subfields of the glossary root are presently not utilized.

Subfields 5 through 9 pertain to the terminal structure with the same relationship between subfield number and terminal class as is used in the instance structure; i.e., subfield 5 corresponds to output terminals, subfield 6 is associated with input terminal, etc. Again, for simplification, the following discussion will be in terms of only one class of terminals. If the device has no terminals of a certain type, the pointer in the associated subfield of the glossary root is NIL; otherwise the pointer points to a 196 structure regardless of how many terminals of that class are actually defined. For each defined terminal of the device, a terminal cell is created and pointed to by a known pointer from the 196 structure; thus, regardless of how many terminals are defined for the device, the method of addressing the terminal cell associated with the n^{th}

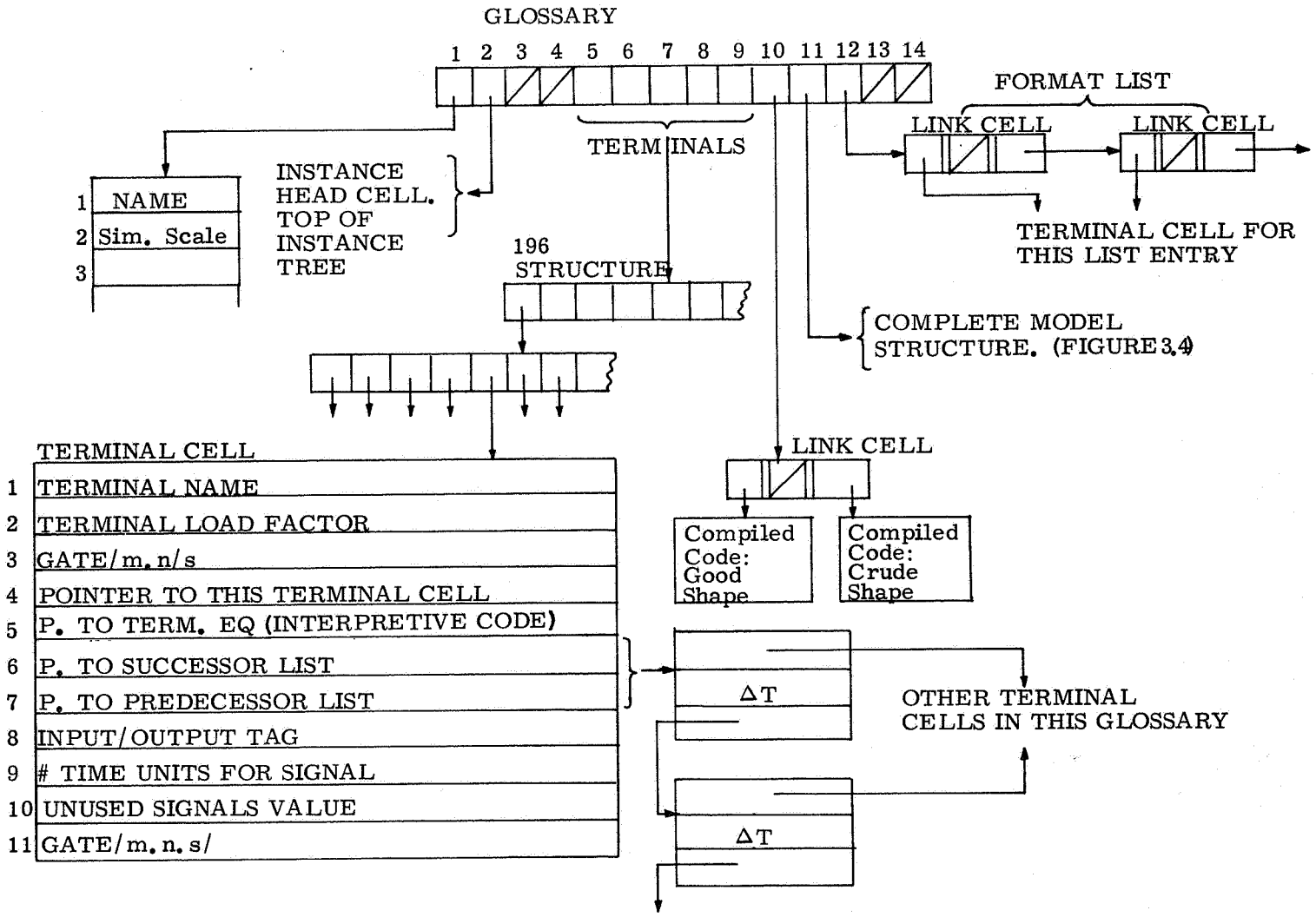

GLOSSARY
1 2 3 4 5 6 7 8 9 10 11 12 13 14
FORMAT LIST
LINK CELL
LINK CELL
TERMINALS
INSTANCE HEAD CELL. TOP OF INSTANCE TREE
1 NAME
2 Sim. Scale
3
196 STRUCTURE
TERMINAL CELL FOR THIS LIST ENTRY
COMPLETE MODEL STRUCTURE. (FIGURE 3.4)
TERMINAL CELL
1 TERMINAL NAME
2 TERMINAL LOAD FACTOR
3 GATE/m, n/s
4 POINTER TO THIS TERMINAL CELL
5 P. TO TERM. EQ (INTERPRETIVE CODE)
6 P. TO SUCCESSOR LIST
7 P. TO PREDECESSOR LIST
8 INPUT/OUTPUT TAG
9 # TIME UNITS FOR SIGNAL
10 UNUSED SIGNALS VALUE
11 GATE/m, n, s/
LINK CELL
Compiled Code: Good Shape
Compiled Code: Crude Shape
ΔT
ΔT
OTHER TERMINAL CELLS IN THIS GLOSSARY

Fig. 3-5

terminal is pre-specified and invariant. As just indicated, there is one terminal cell associated with each defined terminal of the device.

From this point on, the description of the data-file structure becomes hopelessly complicated if strict grammatical rules are followed. To facilitate a clear description of the concepts involved, certain linguistic liberties will be taken. Because of the one-to-one relationship between a terminal cell and a terminal, the phrase "this terminal" will often be used to mean "the terminal which is associated with this terminal cell". Another difficulty now arises; it is obvious that there is nothing physical or real in the data structure, yet it is convenient and clear to refer to signals, terminals and drawings as if they were part of the data structure. For example, it is far clearer to refer to "the signal which is connected to this terminal" rather than "the signal whose head cell is pointed to be the subfield in the terminal structure which is associated with this terminal cell"; the former phrase is easy to understand, the latter is a precise statement.

A slightly different area in which it is sometimes clearer to take certain liberties, rather than being precise, is in reference to signal values. Properly speaking, the terminal of a device is at some logic level and the signal on the conductor connected to the terminal is said to have a logic value. It is occasionally convenient, however, to refer to terminal values and signal values interchangeably.

Returning now to the description of the glossary, the first subfield of the terminal cell contains the terminal name.

The second subfield of the terminal cell contains the signal load factor for the associated terminal.

The third subfield of the terminal cell is essentially a road map in the form of a FILLIP operand pointer showing where to find, within the instance structure, the pointer to the signal which is on this terminal. The subfield contains the bug expression GATE/m.n/s, where normally $5 < m < 9$, $0 < n$, $s < 14$. If the bug named GATE contains a direct pointer to a particular instance, then the operand pointer GATE/m.n/s will locate a subfield of the instance-terminal structure; this subfield will contain a pointer to the signal. As an example, assume the terminal cell describes the 5^{th} input terminal of an instance. The instance is illustrated in Fig. 3-4. Further, assume that Bug GATE points to the instance. Subfield 11 of the terminal cell (in the glossary) will then contain the bug expression GATE/6.1.5/. If bug GATE points to the instance shown in Fig. 3-4, the expression GATE/6.1.5/ locates the signal connected to the 5^{th} input terminal, signal X. Thus the variable instance-terminal structure is interpreted by the glossary. The glossary in turn is a fixed structure. Note that subfield 11 of the terminal cell is similar to subfield

3, except that the bug expression is evaluated as a pointer to the relevant signal-head cell. This redundancy is solely for convenience.

Subfield 4 of the terminal cell contains a direct pointer to the same terminal cell; this is again a convenience in setting up other portions of the data file.

Subfield 5 points to interpretive FILLIP coding of a routine to evaluate the logic level of this terminal; this coding evaluates the equation which the designer utilizes to describe the behavior of the terminal. The coding is used only when an instance of the type is included in a circuit which is to be simulated. Equations exist only for output terminals and for auxiliary equations.

Subfields 6 and 7 of the terminal cell point to list structures which are also useful for simulation purposes. The successor list of a terminal - say terminal A - contains pointers to other terminals (terminal cells) of the device whose logic values are influenced by the value of terminal A.

The predecessor list for terminal A contains pointers to those terminals (terminal cells) of the same device which influence the logic behavior of terminal A. Each entry of both lists also contains the value of time (ΔT) between the time terminal A changes value and the influencing (or influenced) terminal changes state. These lists are used in simulating device behavior.

Subfield 8 of the terminal cell contains a tag which identifies the class of terminal; input, output, etc. The contents of this subfield are simply a number $5 < n < 9$ with the same association employed in the terminal structure, i.e., 5 means input, 6 means output, 7 means expander input, 8 means expander output, and 9 means auxiliary equations.

Subfield 9 contains the largest value of ΔT found in either the predecessor or successor list; this is used only in simulation to define the length of signal history tapes.

Subfield 1Ø of the terminal cell identifies a default signal to which the terminal of the instance is to be connected if the designer fails to specify a signal for the terminal of an instance. UNUSEDØ and UNUSED1 are permitted.

Subfield 1Ø of the glossary root points via a LINK cell to two blocks of compiled codes, each of which describes the shape of the device. Because the buffer which drives the CRT display is of limited size, provision is made for displaying a crude graphical representation of the device on the CRT. This crude shape will require a minimal amount of buffer storage. For hard-copy drawing, however, buffer size is irrelevant, so provision is made for drawing the device as any (reasonable) shape desired by the designer. In either case, the designer draws the desired shape on the CRT with the light pen; this drawing is transformed into the compiled code and stored in the glossary. (See Section 3.6).

Because instance structures of different types will be added frequently during the formation of the data file, it is desirable to have a "master" instance structure stored away which can be duplicated as desired. Such a model structure is pointed to by subfield 11 of the glossary root. The model section includes the instance cell, the instance-drawing structure, the instance-terminal structure and the interconnecting pointers; all data within the model structure is blank, or NIL. Thus, if all the subfields shown in Fig. 3-4 were empty or NIL, except for the pointers between the instance-drawing structure and the instance cell and between the instance cell and the instance-terminal structure, the resulting structure would be identical to the model structure pointed to by subfield 11 of the glossary root.

Subfield 12 of the glossary root points to a format list which is used only when operating in the card or batch-processing mode. The format list is simply a list of LINK cells, where there is one LINK cell for every defined terminal of the device. The format list provides a one-to-one correspondence between signal names on the input cards and terminals of a device. The first LINK cell identifies the terminal to which the first signal on the input card is connected, the second LINK cell identifies the terminal to which the second signal on the input card is connected, etc. If a signal is not specified on the input card (its absence is indicated by a dollar sign on the input card), the unused value of the terminal cell indicates the appropriate default signal.

3.1.6 Signal Structure

All the logical information pertaining to signals is contained in the signal-head cell and associated lists. The signal-head cell structure is illustrated in Fig. 3-6. There is one signal-head cell for every signal in the data file; there is also one signal-head cell for every auxiliary equation of each individual instance.

The signal-head cells are structured in a binary tree as indicated by the first four subfields.

Subfield 5 of the signal-head cell contains a pointer to a list of FILLIP LINK cells, where there is one LINK cell for every instance terminal (except output) to which the signal is connected. The source pointer of each LINKing cell points to the instance to which the signal is connected. The destination pointer of each LINKing cell points to the terminal cell of the connected terminal.

The source qualifier of the LINKing cell identifies the class of terminal to which the signal is connected; it contains the same tag as is found in subfield 8 of the terminal cell.

Subfield 6 of the signal-head cell points to a single LINK cell which is identical to those just described. The source pointer of this LINKing cell,

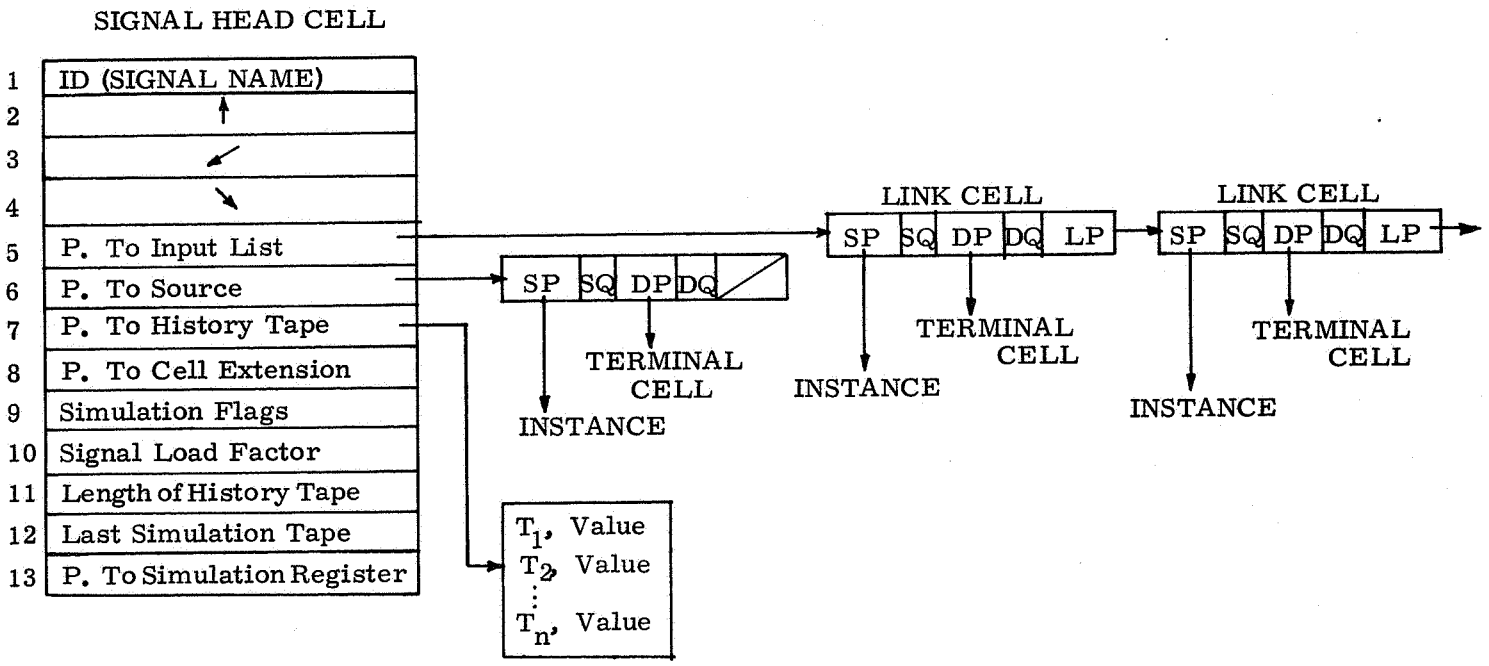

SIGNAL HEAD CELL
1 ID (SIGNAL NAME)
2
3
4
5 P. To Input List
6 P. To Source
7 P. To History Tape
8 P. To Cell Extension
9 Simulation Flags
10 Signal Load Factor
11 Length of History Tape
12 Last Simulation Tape
13 P. To Simulation Register
SP SQ DP DQ
TERMINAL CELL
INSTANCE
LINK CELL
SP SQ DP DQ LP
TERMINAL CELL
INSTANCE
LINK CELL
SP SQ DP DQ LP
TERMINAL CELL
INSTANCE
T1, Value
T2, Value
Tn, Value

Fig. 3-6

however, points to the instance which is the source of the signal; i.e., the instance whose output terminal is connected to the signal. It is important to note that each signal can be connected to only one output terminal; thus, each signal can be generated by only one source.

Subfield 7 of the signal-head cell points to a signal history tape. The history tape is used in simulations and is simply a block of machine coding which contains a history of signal values and the times at which the signal values occurred. The history tape provides the data necessary to evaluate the terminal equations.

Subfield 9 of the signal-head cell contains flags for simulation purposes. Such flags indicate whether a logical error has occurred, whether or not a signal is of interest to the simulation, etc.

The original load factor contained in subfield 10 of the signal-head cell is the algebraic sum of the load factors of all the terminals to which the signal is connected.

Subfields 11, 12, and 13 of the signal-head cell are useful only for simulation purposes. Subfield 11 indicates the maximum time length of history which is maintained for the signal. Subfield 12 contains the most recent time when the signal was evaluated during the simulation. Subfield 13 points to a temporary structure which groups signals for simulation; if any signal of the group changes value, all signals of the grup must be re-evaluated.

3.1.7 Drawing Structure

In the OLLS data file, most of the drawing information is arranged according to drawings, where a drawing can be either the plot which appears on the face of the CRT or a piece of hard-copy output. Each drawing is assumed to have a unique drawing number (ID). For every drawing there is a drawing-head cell in the data file; the structure of the drawing-head cell is illustrated in Fig. 3-7.

As indicated by the first four subfields, the drawing-head cells are arranged on a binary tree according to drawing number.

Two of the types of information which appear on each drawing are the set of devices and the signals or logical interconnections of these devices; this data, in fact, comprises the essence of the drawing. Subfield 5 of the drawing-head cell contains a pointer to the top of the binary tree of the instance drawing structures (see Fig. 3-4) which belong to the drawing. By means of this tree all instances which appear on the same drawing are grouped together.

Subfield 6 of the signal-head cell points to a FILLIP data cell containing editorial information about the drawing. At the present time, this data cell is not strictly defined. It is certain, however, that the drawing size and scale

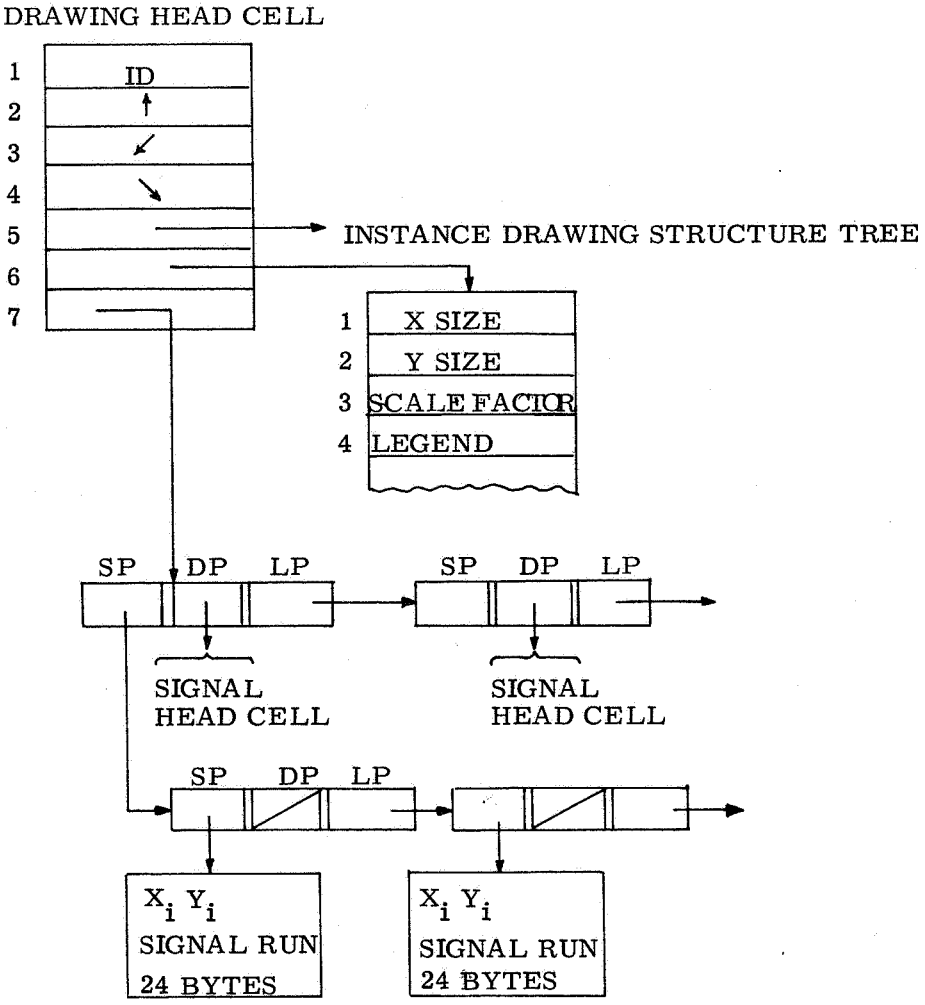

DRAWING HEAD CELL
1
2
3
4
5
6
7
ID
INSTANCE DRAWING STRUCTURE TREE
1 X SIZE
2 Y SIZE
3 SCALE FACTOR
4 LEGEND
SP DP LP
SP DP LP
SIGNAL HEAD CELL
SIGNAL HEAD CELL
SP DP LP
X_i Y_i SIGNAL RUN 24 BYTES
X_i Y_i SIGNAL RUN 24 BYTES

Fig. 3-7

factor will be contained in the data cell, as will information required for the legend or signature of the drawing.

Subfield 7 of the drawing-head cell points to a structure containing graphical information about the signal runs which appear on the drawing. The pointer in subfield 7 points to a list of FILLIP LINK cells, where there is one LINK cell for each signal that is drawn on the drawing. The LINK cells are connected by the LINK pointers of each cell. The destination pointer of each LINK cell points to a different signal which appears on the drawing. Each source pointer of the LINK cells points to a different list of LINK cells; these cells are used to store and recover the graphic coordinates for each run of the signal. The source pointers of these LINK cells each point to a block of machine coding containing the coordinates of the signal run. There is one block of coding for each interconnection of two different terminals. Thus, if Fig. 3-7 is used as an example, the first signal on the drawing - which is obtained via subfield 7 of the drawing-head cell - connects three terminals on this drawing. It is known that three terminals are connected since two blocks of coordinates are shown and each block of coordinates describes one signal run or interconnection.

It should be noted that the entire drawing structure of the data file enables one to immediately reproduce drawings which have been created. This is a necessary requirement if different drawings are to be called up to appear on the CRT during an on-line process.

3.1.8 Data-File Root

The binary tree of signal-head cells, the binary tree of drawing-head cells, and the 196 structure of glossaries are all addressable through the data-file root as shown in Fig. 3-8. To facilitate discussion, it is convenient to consider the root of the file as being the cell of 14 pointers, rather than the NOP instruction which is the actual FILLIP file root.

As shown in Fig. 3-8, subfield 1 contains a pointer to the tree of signal-head cells, subfield 2 contains a pointer to the 196 structure of glossaries, and the third subfield contains a pointer to the tree of drawing-head cells.

Subfield 4 of the data-file root is normally NIL. During simulation runs, however, a structure consisting of the simulation event list, of initialization conditions, and of signal registers is temporarily constructed and pointed to by a pointer in subfield 4. Because the simulation is temporary and unique to the simulation program, the structure will not be described in further detail at this time.

In general, there will be signals in the data file which have no source; i.e. are not connected to the output terminal of any device. These signals are called

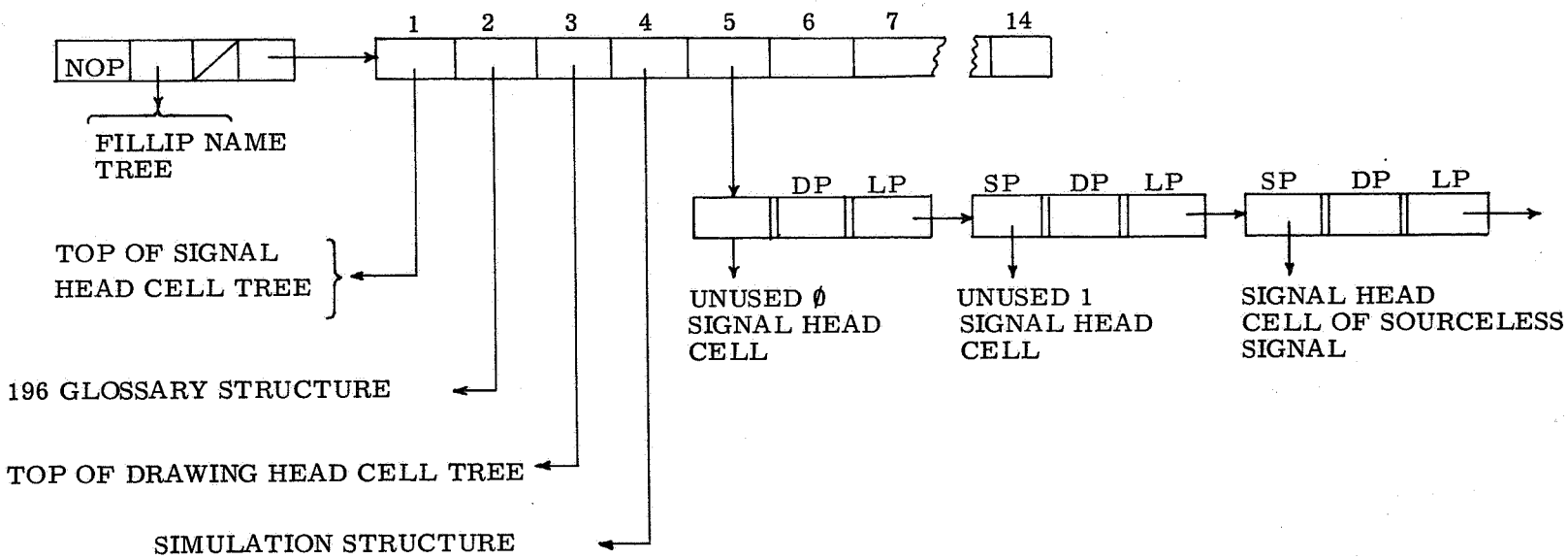

NOP
1
2
3
4
5
6
7
14
FILLIP NAME TREE
TOP OF SIGNAL HEAD CELL TREE
196 GLOSSARY STRUCTURE
TOP OF DRAWING HEAD CELL TREE
SIMULATION STRUCTURE
DP
LP
SP
DP
LP
SP
DP
LP
UNUSED ∅ SIGNAL HEAD CELL
UNUSED 1 SIGNAL HEAD CELL
SIGNAL HEAD CELL OF SOURCELESS SIGNAL

Fig. 3-8

sourceless signals. Two specific examples of sourceless signals are UNUSEDØ and UNUSED1. Other occurrences of sourceless signals result during the modification of existing drawings when signal interconnections are changed. For simulation, as well as for error-checking purposes, it is desirable to group all sourceless signals together. For this purpose, subfield 5 of the data-file root contains a pointer to a list of FILLIP LINK cells. The source pointer of each LINK cell points to the signal-head cell of a sourceless signal. More specifically, the first LINKing cell contains a pointer to the UNUSEDØ signal-head cell and the second LINKing cell contains a pointer to the UNUSED1 signal-head cell. The remainder of the LINKing cells contain pointers to head cells of any other sourceless signals which might exist.

3.1.9 Integrated Data Structure

The various parts of the data-file structure have now been described in detail. Figure 3-9 illustrates how the various component parts are tied together to yield a highly interwoven data structure. In Fig. 3-9 much of the detail has been omitted to avoid unnecessary confusion.

A few words about the overall data structure will help to clarify some of the important concepts. First, there are four independent classes of binary-tree structures in the data-file structure. Each tree represents a grouping of data according to some common characteristic. Each tree is normally addressed through the top node of the tree, although individual nodes of the tree are often addressed from external structures. Two of the trees, the tree of different drawings and the tree of signals, are addressed directly by the root of the data-file structure. Up to 196 different types of logical devices can be incorporated into one data file; the glossaries for the types are also addressed by the file-structure root. Each glossary points to a binary tree of all instances of that particular type. Each drawing-head cell contains a pointer to a binary tree of all instances which appear on the drawing. Each signal-head cell points (indirectly) to every instance to which the signal is connected and, similarly, each instance points (indirectly) to the head cell of every signal which is connected to the instance. Each instance also points to the head cell of the drawing on which the instance is located. The drawing-head cell also points to the head cell of every signal which appears on the drawing.

At first glance, it might appear that the data structure is unnecessarily complex. A few illustrations might illustrate the capabilities of the file structure. Consider the problem of removing a signal from an instance. In the batch-processing mode, it is most convenient to identify a particular signal by its name; it is also easiest to identify an instance by its type and identification (identifications need be unique only within a type class). The identified signal

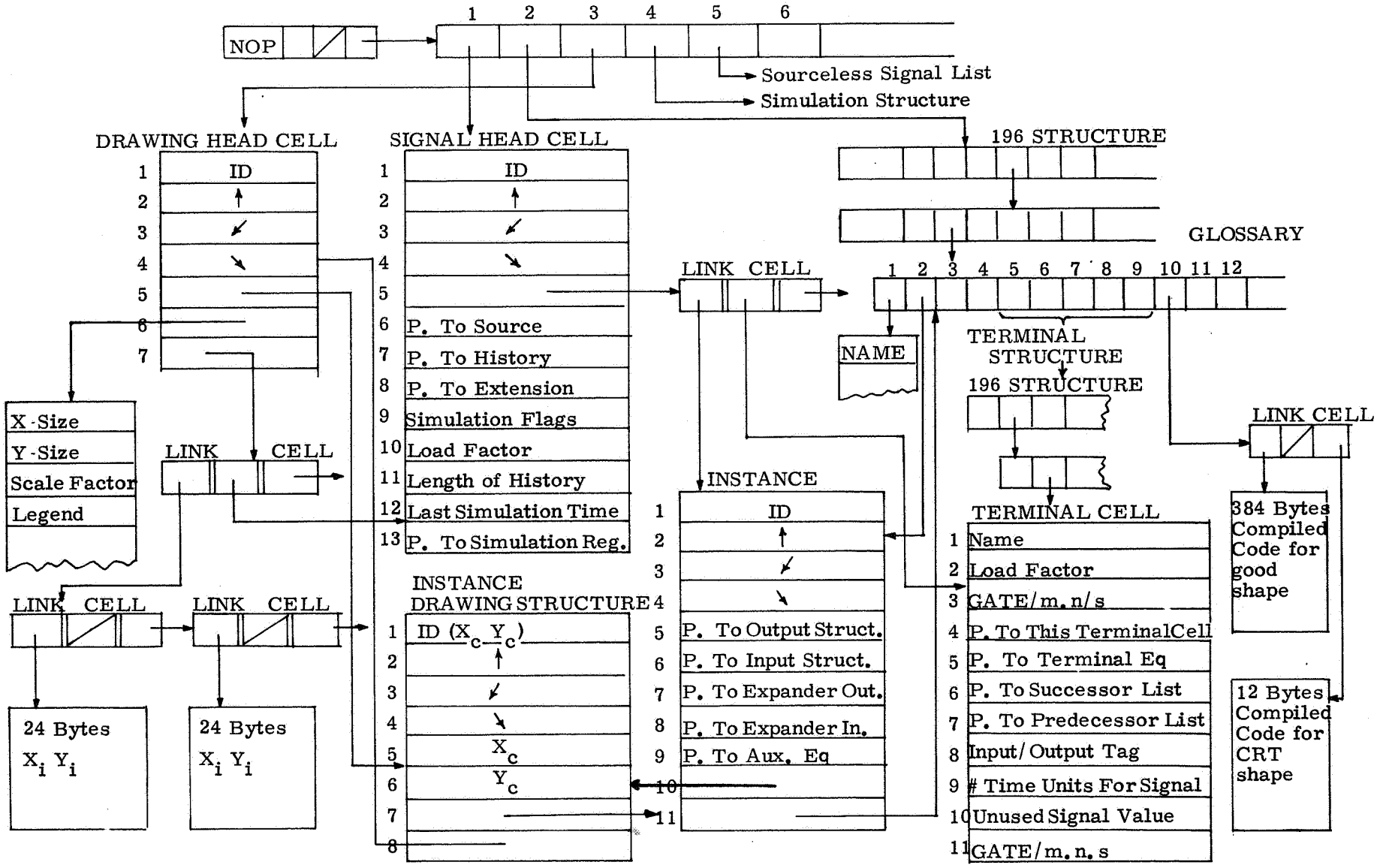

NOP
1 2 3 4 5 6
Sourceless Signal List
Simulation Structure
DRAWING HEAD CELL
SIGNAL HEAD CELL
196 STRUCTURE
GLOSSARY
ID
P. To Source
P. To History
P. To Extension
Simulation Flags
Load Factor
Length of History
Last Simulation Time
P. To Simulation Reg.
LINK CELL
NAME
TERMINAL STRUCTURE
196 STRUCTURE
X-Size
Y-Size
Scale Factor
Legend
LINK CELL
INSTANCE
INSTANCE DRAWING STRUCTURE
ID (X_c Y_c)
X_c
Y_c
P. To Output Struct.
P. To Input Struct.
P. To Expander Out.
P. To Expander In.
P. To Aux. Eq
TERMINAL CELL
Name
Load Factor
GATE/m.n/s
P. To This TerminalCell
P. To Terminal Eq
P. To Successor List
P. To Predecessor List
Input/Output Tag
Time Units For Signal
Unused Signal Value
GATE/m.n.s
384 Bytes Compiled Code for good shape
12 Bytes Compiled Code for CRT shape
24 Bytes X_i Y_i
24 Bytes X_i Y_i

Fig. 3-9

is readily found in the data file by a FILLIP search of the tree of signal-head cells; this tree is directly addressable through the data-file root. Having found the head cell of the specified signal, the list of instances (actually LINK cells) which are connected to the signal is searched again by one FILLIP instruction until the specified instance is found. The LINK cell pointing to this instance also points to a terminal cell which tells where in the instance the pointer to the signal is to be found; the changing of this pointer and the pointer in the LINK cell constitutes removing the signal from the instance.

To accomplish the same result when operating in the on-line mode, the drawing containing the instance will be called up for display on the CRT. To identify the signal which is to be removed, the designer will point with the light pen to the terminal of the device to which the signal is connected and command the system to remove the signal. In this case, the tree of drawings is searched (one FILLIP instruction) until the head cell for this drawing is located. The tree of instances (instance-drawing structures) on this drawing is then searched by coordinates (coordinates of device and the specific terminal are computed from the light-pen position) until the correct instance structure is found. The instance itself is addressed through the instance-drawing structure and the process continues as in the batch-processing node.

A different type of problem arises during simulation; here it is necessary to know what instances are affected by changes in signal value. The head cell of the signal is again located and the list of instances is searched. For each instance, the glossary is consulted to determine the equation for the specific terminal. If the terminal is an input terminal, the terminals on the successor list (output terminals on the same instance) are examined and their terminal equations are examined. If the successor terminals change state, the corresponding signals are examined in a recursive process; these signals are addressed from the instance-terminal structure using the glossary to identify the location of the address pointer. The simulation process is described in detail in Section 3.3 of this report; the sole intent here is to show how the data structure can be utilized.

It is felt that the data structure provides sufficient flexibility to be useful in a wide variety of design problems, either in a batch-processing or on-line mode. It is further felt that the structure can be readily expanded to include new classes of data when they become identified. Although the properties of FILLIP influenced the development of the data structures, the data structure stands by itself, and can be implemented in any reasonable language.

3.2 Device Definition

James Pennypacker

One of the most important features of OLLS is the provision for the designer to define his own logical devices. The user is not confined to using a set of pre-defined elements but within broad limits can use any logical element, combinational or sequential, he cares to define as a device. Rather than having to code separate programs to handle each new logic element, the user can in a straightforward manner define the functional behavior of the model for the new element. This means, for example, that the advent of large-scale integration (LSI) will necessitate no programming modifications; an LSI chip and a single logic gate are handled with equal ease. The data and program structures are organized to accept the user-defined device without additional programming effort. Once a device type is defined, instances of the device type can be used whenever desired.

3.2.1 Contents of Definition

The definition of a device includes the name of the device type, the number of each class of terminal for the device, logic equations relating output terminal behavior to input terminal behavior, the load factor for each terminal, and, when the device is defined at the on-line graphic console, the shape of the device symbol.

Each defined logic device can have four different classes of terminals, input, output, expander input, and expander output. Expander input and expander output terminals are electrical points of the device which provide for fan-in and fan-out capability. In addition to the four types of terminals, the definition of a logic device may include internal logic states which are neither input nor output terminals; the definition of any sequential circuit would include such internal variables. These internal variables, hereinafter referred to as auxiliary equations, are treated identically to the other four classes of terminals; they are expressed as - and included in - equations specifying logic behavior of other terminals.

At the present time, a defined device may contain up to 196 terminals of each type, including auxiliary equations. The number of terminals is the only hard restriction in the definition of a device; without extensive reprogramming, the data structure could be modified to provide for up to 2744 terminals of each type. Each terminal of the device (and auxiliary equations are to be included as terminals) is given a unique name that is used in the specification of the logic operation of the device.

3.2.2 Concepts of Definition

The logic operation of a device is specified by the equations which relate each output, expander output, or auxiliary equation to other terminals of the device. For example, the NOR gate shown in Fig. 3-10 has two inputs, A and B,

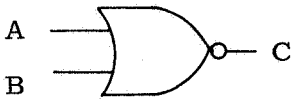


Fig. 3-10

and one output, C, where A, B, and C are understood to be the respective terminal names. The logic operation of this device is specified by the equation.

$$C = \bar{A} \cdot \bar{B}$$

A subscript notation is employed to show time delays. Using this notation, a subscript $\emptyset$ indicates the time "now", a subscript 1 indicates time one unit previous, a subscript 2 indicates two time units ago, etc. The time unit is dimensionless but is often understood to be one gate delay, the time it takes for a logic element to respond to the input excitation. Subscripts indicating relative time dependency must be integers (see Appendix B).

For the two-input NOR gate shown in Fig. 3-10, the logic operation can be more completely specified by the equation

$$C_0 = \bar{A}_1 \cdot \bar{B}_1$$

which indicates that the behavior of terminal C at the present time is influenced by the behavior of both terminals A and B one time unit ago. If the excitation does not influence the output terminal equally rapidly for both input terminals, the equation might be expressed as

$$C_0 = \bar{A}_1 \cdot \bar{B}_2$$

To more fully illustrate the concepts under discussion, consider a device with two inputs, S and R, one output, Q, and one internal state or auxiliary equation, P, as illustrated in Fig. 3-11.

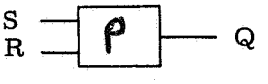


Fig. 3-11

The operation of the device is specified by the equations

$$Q_0 = \bar{S}_1 \cdot \bar{P}_1 \tag{3-1a}$$

$$P_0 = \bar{Q}_1 \cdot \bar{R}_1 \tag{3-1b}$$

Note that the behavior of every terminal is specified at time "now"; i.e., the subscript of variables on the left-hand side of the equations is always $\emptyset$. Further, note that the behavior of internal states must be specified in addition to behavior of output terminals.

Equation set 3-1 is completely acceptable to OLLS as a definition of the device. It is possible, however, to simplify Eq. 3-1 by noting that if $P_0 = \overline{Q_1} \cdot \overline{R_1}$ describes the behavior of state P at time now, then the behavior of P one time unit ago is specified by the relation $P_1 = \bar{Q}_2 \cdot \bar{R}_2$. Substituting this expression into Eq. 3-1a, followed by logical simplification, yields

$$Q_0 = \bar{S}_1 \cdot (R_2 + Q_2) \tag{3-2}$$

which is also sufficient to define the device. Note that, in Eq. 3-2, the behavior of terminal Q is a function of its own state two time units ago. This will, in fact, be the case when any sequential circuit is defined as a logic device.

Either Eq. 3-1 or 3-2 may be used to define the logic properties of the device; both result in the same logic operation. The simulation of this particular device, however, will run somewhat faster if Eq. 3-2 is used. It is difficult to generalize as to whether or not equations should be simplified when the device is defined; it is only during simulation that any difference is observable and the difference is one of running time only. If most of the defining equations are expressed as functions of all other terminals, then simplification of the equations will speed up the simulation. On the other hand, if most of the defining equations are each expressed as a function of only a few of the other terminals, then simplification of the equations will slow down the simulation program. The reason for this is that the simplified equation for a terminal, say terminal Q, includes many terminals in the expression. If any of these terminals changes state, the entire expression for terminal Q must be re-evaluated to determine whether or not terminal Q has changed state. Because of the large number of variables which are maintained in the equation for Q, the equation will have to be evaluated more frequently than if fewer terminals were included in the defining relationship. Without actual operating experience with OLLS, however, it is virtually impossible to analyze the difference in running time of the simulation program as a function of the defining equations.

It was previously mentioned that, once a device was defined, instances of the device could be used whenever desired. In other words, the concept of a NOR gate must be made known to the data file before individual NOR gates are used in the design process. The definition of the device type and the utilization of individual devices are normally independent operations except that definition must precede utilization. As will be discussed later, the utilization of an individual instance includes specifying what signals appear on the different terminals of the instance. In this manner, instances are interconnected to form circuits. Part of the process of defining a device includes the option of specifying the order in which the terminals of the device are to be connected to signal interconnections. The definition of a device also includes specifying the default signals to which unmentioned terminals are to be connected.

3.2.3 Definition by Terminal Behavior

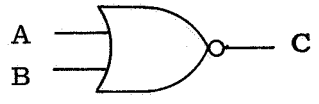
There are two methods by which a device may be defined: by specification of terminal behavior and by circuit design. Because the difference between the two methods concerns only the manner of defining the logic behavior of the device, the discussion will concern only this phase of device definition.

The process of defining a device by specifying terminal behavior was essentially described in the preceding section. The process involves identifying each of the terminals, including auxiliary equations, by a unique name. Logical equations are then written for each terminal, specifying the behavior of one terminal as a function of all the terminals of the device. Equations specifying the behavior of input and expander input terminals are of course not required. Each logic equation must be of the form of Eq. 3-1 or 3-2. In fact, Eq. 3-1 or 3-2 comprise definition by terminal behavior.

3.2.4 Definition by Circuit Design

A considerably easier method of defining a device is by circuit design. While this method of definition is currently available only for operation in the on-line mode, it is planned to incorporate the procedure into the batch-processing mode.

To illustrate this method of definition, assume that the NOR gate shown in Fig. 3-12 has been defined by terminal behavior. Further, assume that the



$$C_0 = \bar{A}_1 \cdot \bar{B}_1$$

Fig. 3-12

designer has drawn on the CRT with the light pen a circuit which is an interconnection of only devices of the type shown in Fig. 3-12. The circuit which has been drawn is shown in Fig. 3-13. In this circuit, the letters are names of

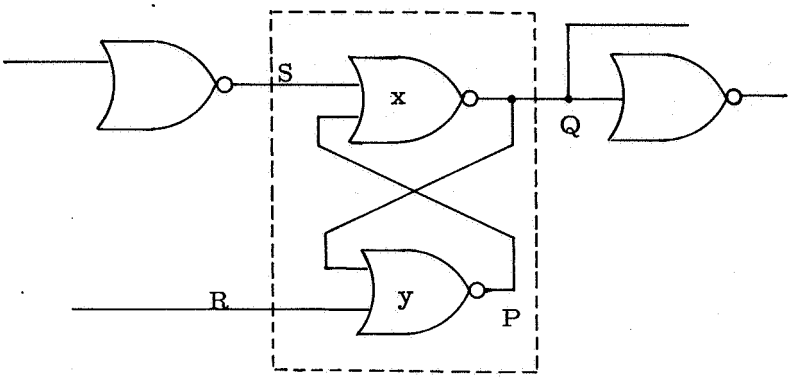


Fig. 3-13

signals which appear on the various terminals of the interconnected devices; the terminals of each individual NOR gate are still called A, B and C as per the definition of the device. Having drawn the circuit, the designer recognizes that the portion of the circuit enclosed within the dotted lines is a flip-flop which is used so frequently that it is desirable to define this portion of the circuit as a single logic device. Because the NOR gate has been specified by terminal behavior and because the drawn circuit represents logic interconnections of the NOR gates, the logic behavior of the flip-flop is already contained in some form in the data-file structure. Thus the designer can define the flip-flop as a new device by simply identifying the devices which are included in the flip-flop (gates x and y) and the terminals of those gates which are also to be terminals of the new device. The only other information which the designer must provide is the name of the new device, FLIP-FLOP. Identification of the gates and terminals is performed with the light pen; the name is entered via the graphic console keyboard.

In general, when a device is defined, the terminals of the device must be given names which are unique among themselves. When a device is defined by circuit design, the terminals of the new device are named after the signals which are connected to the respective terminals. Thus, the output terminal of the flip-flop shown is named Q and the two input terminals of the flip-flop are S and

R. (These terminals names can be modified by the designer, if desired.) Because the output of NOR gate y is not considered to be an output terminal of the flip-flop, it gives rise to an internal state or auxiliary equation, P, of the newly defined flip-flop. Note that, even though none of the individual NOR gates has an internal state, the interconnection of these devices yields a new device which has an auxiliary equation associated with its definition.

After the elements and interconnections which comprise the new device have been identified, the model of the new device is automatically generated by OLLS and equations specifying the terminal behavior of the new device are automatically generated. For the flip-flop shown, the generated equations are:

$$P_0 = \bar{Q}_1 \cdot \bar{R}_1 \qquad (3\text{-}3a)$$

$$Q_0 = \bar{S}_1 \cdot \bar{P}_1 \qquad (3\text{-}3b)$$

These equations are identical to Eq. 3-1; in fact, the black box defined by Eq. 3-1 is a flip-flop.

The load factor for each terminal of the new device is automatically generated from the load factors of the individual components of the device.

Currently, the shape of a device which is defined by circuit design is automatically generated as a rectangle; this shape can be modified as desired via the on-line graphic console.

The process of defining a device by circuit design is simple and fast for the designer, requiring only that the individual components be previously defined. The designer can thus define devices in a boot-strap manner; the only restriction is that the new device not exceed the restriction of 196 terminals of each type.

3.2.5 Impact on Data File

The impact on the data structure is identical when the device is defined by terminal behavior as when the device is defined by circuit design. The following discussion assumes definition by terminal behavior unless otherwise specified. Reference to Fig. 3-9 might facilitate an understanding of the material in this section.

When a new device is defined, a glossary root is constructed and connected to the 196 structure of glossaries. The name of the device is inserted in a FILLIP data cell which is pointed to by the glossary root.

For every class of terminal which is defined for a device type, a 196 structure of terminal cells is constructed and connected to the appropriate subfield of the glossary root. For each terminal, the terminal name and load factor are inserted in the terminal cell.

The equation specifying the logic behavior of the terminal is compiled into interpretive FILLIP code which is addressed by the terminal cell. The equation is also scanned for the largest subscript which is inserted in subfield 9 of the terminal cell. The equations are also used to construct the predecessor and successor lists.

One of the important results of device definition is the construction of a model-instance structure which is pointed to by the glossary root. The instance-cell and the instance-drawing structures are of fixed format and therefore easily constructed; the instance-terminal structures are of variable format and are more difficult to create. The instance-terminal structure depends upon the number of terminals which are defined for the device; the contents of subfield 3 of the terminal cell is also a function of the number of defined terminals.

The shape of the defined device is translated into machine coding which is addressed indirectly through the glossary root.

To summarize what has been stated so far, the definition of a device by terminal behavior results in a new glossary, in the formation of the associated structures, and in the insertion of the contents of every defined terminal cell.

When the device is defined by circuit design, the end result is the same as if the device were defined by terminal behavior; the process is, however, considerably different since most of the required information is not explicitly available. Without going into the complexity of detail which is required, it is sufficient to state that the logic-data file is searched to provide the information required to construct the glossary and its associated structures. The designer need not specify the logic behavior of the device; OLLS subroutines process the existing relevant interpretive code which describes behavior of the individual component terminals to yield interpretive coding for each terminal of the new device.

3.3 Simulation

Herbert Thaler

3.3.1 Circuit Formation

a.) Device Definition

The formation of a logic circuit for simulation by OLLS can be divided into two phases. The first phase is the definition of the terminal behavior of the logical devices to be included in the circuit. This is accomplished through the "Define Device" subprogram of OLLS. Those properties of a device which are essential to simulate it are:

1. A classification of its terminals into at least two categories - input and output.
2. Boolean equations to relate the output terminals to the inputs (and outputs).

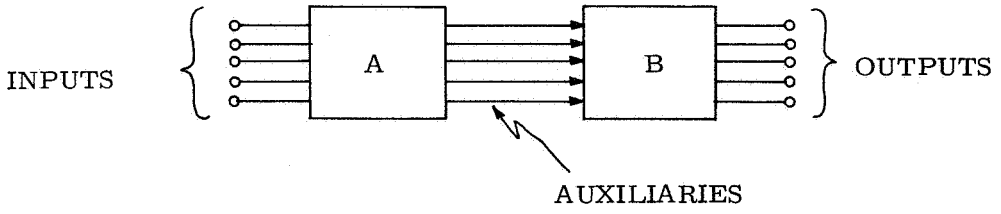
A device-input terminal is one whose logic value cannot be affected by the device itself. That is, there is no Boolean equation within the device to change a logic signal value at that terminal. It can only follow the value of an applied signal.

An output terminal, on the other hand, does have a generating equation. In fact, all output terminals of all devices must have their own independent generating equations. Therefore, output terminals of devices may not be interconnected, since there is no unique way to resolve the resulting competition between independent Boolean generating functions. This fact, and a desire to make OLLS as general as possible, gives rise to two other classifications of device terminals - expander outputs and inputs.

In certain types of logic families (e.g., RTL), it is common practice to tie device outputs together to achieve higher fan-in and/or fan-out capabilities than the individual devices provide. In other types of logic families (e.g., TTL), higher fan-in is achieved by connecting specially designed expander devices to expansion terminals provided solely for this purpose. Both techniques are logically correct, but the TTL approach is more general. Therefore, OLLS adopts that view of fan-in augmentation. If an output-terminal equation can be expanded logically by either technique, an expander-input terminal is provided on the device. The expander-input signal value must then be included in the Boolean equation for the expandable-output terminal. However, only signals which originate on expander-output terminals may be connected to expander inputs. Thus two classes of signal runs are found in an OLLS logic circuit. Normal signals originate on the output terminal of some device, and may be

connected only to the normal input terminals of other devices. Expander signals originate on the expander-output terminal of some device and may be connected only to the expander-input terminals of other devices. A generalized device may possess all four terminal categories if the proper equation set and terminal classification are given to it.

There is a fifth class of terminal that, in the interest of generality, a device may possess. It is often necessary to define more equations to specify the behavior of a device than there are output terminals to assign them to. This usually occurs when defining a sequential circuit with many stable states and few outputs. Since OLLS places almost no restrictions on the complexity of definable devices, equations which affect output terminals but do not themselves represent real terminals must be accommodated. These equations are classed as auxiliary terminals in the definition of the device. Auxiliary equations are equally as important to the behavior of a device as its output equations, and differ from them only in that they are not available as signals outside the device. Therefore, an auxiliary-equation variable is treated in the OLLS logic-circuit file as an output signal which cannot be connected to any external inputs, but which nevertheless may influence other terminals within the device. As an example of one use of an auxiliary variable, consider the following device:



Let us assume that the two elements A and B are complex combinational devices, but that their individual equation sets are known. The OLLS user may choose to define their cascade in either of two ways. He may substitute into the equations for B the functions appearing at the outputs of A. This would eliminate the intermediate variables from the overall cascade equations, and express the outputs of B in terms of the inputs to A. The alternative is to submit the equation sets for both A and B, retaining the intermediate variables as auxiliary terminals. This achieves the identical simulation behavior for the cascaded circuit, but saves the user much effort in reducing the total equation set he must provide. Further example of the trade-off in effort and efficiency between minimized equation sets (a few complex equations) and the more easily generated gate-by-gate sets (many simple equations) are given in Appendix B. Often the preferred choice is more one of personal taste than engineering necessity, but certain cases of necessity can be defined. These occur near the limits of device

complexity, when the number of independent variables exceeds 196. Then one may be forced to eliminate redundant equations simply to fit the device definition within the specified limits.

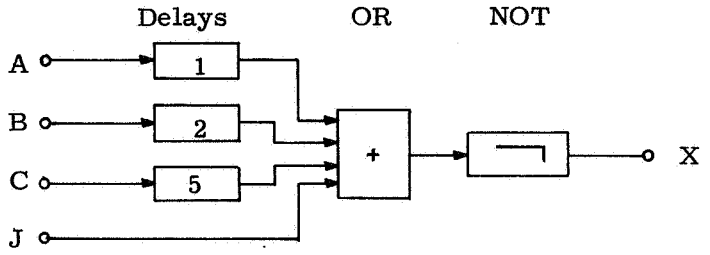
Any logic device, which can be modeled by using the following basic set of ideal Boolean elements, can be defined in OLLS. The ideal elements are:

a. delayless multi-input OR
b. delayless multi-input AND
c. delayless NOT
d. ideal delays.

Since this set of elements is complete, any finite combinational or sequential circuit can thus be defined. For example, an expandable 3-input NOR gate can be modeled as follows:

```
*   Define Device   NOR
    OUTPUTS         X
    INPUTS          A,B,C
    EXPINS          J
    EQUATION        X = ¬(A + B + C + J)
    DELAY           0      1   2   5   0
```

This set of input data represents the following device model:



Examples of the method used in OLLS to define a logic device by its terminal Boolean equations have been given, but the power and generality of the method have not yet been explored. As has been stated, the only restriction applied to the device definition is on the number of each type of terminal which may be used on one device. This limit is currently set at 196 for our own convenience. Hence, a device may have up to 196 inputs, 196 outputs, and 196 auxiliary variables in addition to expander terminals. The form of the output and auxiliary equations themselves need only be deterministic - that is, there must be a unique equation which specifies the current value of each terminal at all times. The equation forms are:

$$OUT_o = F \text{ (inputs, outputs, auxes, expins, expouts)}$$

or:

$$AUX_o = F \text{ (inputs, outputs, auxes, expins, expouts).}$$

There is no limit on the complexity of the functions involved, no limitation on what variables may appear in them, and none on what delay values may be used (zero delay is permissible but should be avoided). An equation may even mention the variable it is defining in the function for that variable.

With these rules, most reasonable system functions can be incorporated as a single device. This includes, for example, storage arrays with up to 587-bit capacity, sequential circuits with up to 2^{588} states (not all stable!), delay lines of any length (with up to 195 taps), combinational circuits of virtually any complexity, binary counter-divider up to 196 stages long, complete computer arithmetic units, computer input/output control units, and just about anything that the near future will see implemented by LSI technology. This generality is, we feel, necessary for the next generation of logic simulation programs, and is one of the strong points of OLLS.

b.) Interconnections

Once the OLLS user has successfully defined or copied those logic devices he wishes to utilize, he may begin to form a circuit. For simulation purposes, an OLLS circuit (or file) consists of the device definitions, one or more instances of the logic devices, and interconnections. Any mixture of different device types may reside in one file, since OLLS assumes they are compatible. The simulation program treats all devices alike, and therefore relies on the device definitions (glossary entries) to differentiate behavior. The mechanics of interconnecting devices and assigning names to the signals created are explained in Section 3.2 of this report. All that need be said at this point is that it is easy to create the total file in terms of instances of the defined logic devices and signal names.

3.3.2 Desired Capabilities

a.) Logical Initialization

Once an OLLS user has specified his circuit topologically, he may turn his attention to its simulation. The first step in this process is the establishment of initial logic values at all the terminals and signals of interest. This is necessary because OLLS recognizes three signal logical values - zero, one, and undefined; and because the circuit is first automatically set so that all signals are undefined. This is done to force the user not only to initialize every signal of interest to him, but also to be aware of all other signals in his file

which have a direct effect on the test circuit. He need not initialize every signal in a large circuit to test only a small subsection of it, but he must satisfactorily isolate the test circuit logically.

A particular signal is initialized in the OLLS card system by specifying the intended value and the signal name. In the CRT-oriented system, the signal of interest may be designated either by name or by touching with a light pen a device terminal on its displayed run. This is especially useful since the names of some signals on circuits created at the CRT may not be known to their creator. This occurs because the process of gate interconnection is more easily handled graphically than by card entry, and hence the card system artifact of the user manufacturing signal names for every run is unnecessary. Some signal names, therefore, may have been created by OLLS to fill the void left by the user. These names are derived from the ID, type, and terminal name of the source device, and generally have no nmemonic value to the user. They do, however, exist in the file and can readily be determined.

However a signal to be initialized is designated, the user still has the option of stating an initial value for every signal, or of trying to minimize such effort. OLLS has the ability to propagate logic levels through devices (from inputs to outputs) subject to the logic constraints given in the device definitions, but independent of the delay values therein. This causes the logic circuit to behave as if all its delays were zero, but doesn't affect the Boolean equations within it. Thus, for example, if a logic "one" is specifically placed on one input to an OR gate, then the output terminal signal will also become specified through propagation. This is particularly useful since many logic circuits are designed with an unconditional preset signal distributed through the circuit. When such a signal is initialized and then propagated, many additional signals become defined gratis. An example of such a situation can be found in the MIT Apollo Guidance Computer where a single preset signal can initialize every significant section of the computer control logic.

The intent of initialization is to establish a static configuration of logic signals on which to base further simulations. When signal propagation is used, the static nature of the signal set may be disturbed. During the course of propagation a signal previously declared to be $\emptyset$ (1) may become re-evaluated as a 1 ($\emptyset$). This generally indicates oscillatory circuits, or sequential circuits in which the control variables are improperly defined. Since the phase and frequency of such oscillation is indeterminate within the context of static initialization, the logic values of the signals involved are forced to revert to undefined. The user is informed of such behavior and is expected to provide the necessary circuit or initialization changes to correct it. Thus, for example, oscillators should have control lines which can quench their activity during initialization but can release them afterward for the dynamic run.

b) Circuit Stimuli

The user must be able to insert stimuli for his test circuit once the dynamic simulation run begins. There should be both "one-shot" and repetitive-waveform stimulus capability, with all parameters under his control. The "one-shot" stimulus is in the nature of an event, and is so classified by OLLS. To insert such an external event, the user would prepare a card image (in either the card-oriented or CRT OLLS system) with the following format:

EVENT Signame Value Time.

Naturally this causes the signal designated to assume the given value at the given time during the dynamic simulation run. The signal remains at that value (Ø or 1) until some other external or internal event causes it to change.

It is also desirable to be able to specify that an external event should occur based on purely internal circumstances. With such a capability one would not be concerned with the time at which the event occurs, but rather with the internal events leading up to the desired effect. For example, one might desire to turn on (or off) an oscillatory circuit control line if some other signal level in the file became a 1 (or Ø). This almost has the effect of making a temporary logical connection for simulation purposes between unconnected signals in the circuit, and would be expressed by the user as:

EVENT Signame Value IF Expression.

Whenever the given Boolean expression becomes true, the specified signal assumes its stated value. The expression need not be a single-signal name. It can be any Boolean combination of signal names (and their complements) which appear in the file being simulated.

This example of a conditional event allows the designated signal to assume only one of the possible logic values whenever the Boolean expression becomes true. In order to accommodate the other value, another conditional event has to be given. Thus, for example, the pair of cards:

EVENT	ALPHA	1	IF	BETA
EVENT	ALPHA	Ø	IF	¬ (BETA)

slaves signal ALPHA to expression BETA both for BETA ← Ø and for BETA ← 1. Therefore, ALPHA and BETA are the same for the simulation run. If BETA is only one signal rather than a combination of signals, the two signals are logically connected together for the duration of the simulation run. Since this enables the

user to make temporary connections in his circuit without actually affecting the circuit itself (only the simulation thereof), it should prove to be a very useful feature. A single additional command to combine the two conditional event cards given above into one card is:

EVENT ALPHA EQUALS BETA

where BETA can be either a single-signal name or an expression. Note that there is a definite direction implied by this card - ALPHA follows BETA, not vice versa. Therefore, BETA must be a signal with a real source, while if ALPHA has a source it is always competing with BETA for dominance. Generally it is best if ALPHA has no real source (no device output terminal) in its run.

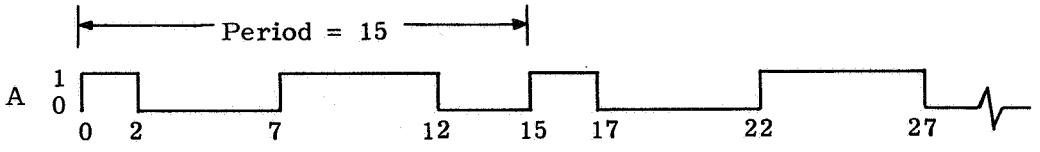
The second type of circuit stimulus allowed in OLLS is a general repetitive waveform. This is actually an unending sequence of unconditional events all directed at some signal. A shorthand technique for specifying the parameters of the waveform is provided.

SEQUENCE Signame Value Period TList

as an OLLS input card, defines a waveform to be applied to the designated signal. The first new value and the period of time for repetition are given explicitly. The last entry (TLIST) is actually a list of discrete times of events within the first period. Each one causes a transition of the designated signal at the given time. As an example of such a sequence, consider

SEQUENCE A 1 15 0 2 7 12

which describes a waveform as follows:



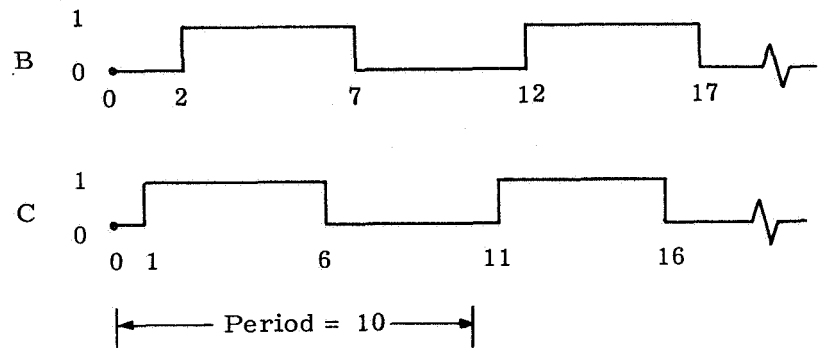
The signal A would execute the transitions shown at the times indicated in the sketch above. The number of events which may appear in the TList is limited only by the total period given. In this way the OLLS user can build up very complex repetitive waveforms, or he may generate simple square waves as, for example:

```
SEQUENCE    B    1    10    2    7
```

This describes a period-10 square wave starting at time = 2 with value 1. Signal B would be delayed one time unit from signal C if:

```
SEQUENCE    C    1    10    1    6
```

were the definition of C. The waveforms defined as B and C would appear as follows:



c) Output Features

It almost goes without saying that a good logic-simulation program must have good output capability. This is especially true if the intent of the program is to mimic a circuit rather than to analyze it. OLLS falls into the former category - it is intended as a very realistic replacement for the real circuit, especially in the early stages of computer design. This in no way prejudices its value in later stages when the design is more mature, since the record keeping capabilities of the OLLS system will then be truly invaluable to the designer.

OLLS, therefore, possesses a broad output capability, reflecting its use not only in all stages of logic design but also as a console interactive system. These output features are:

1. Trace.
2. Sample.
3. Hazard Detection.
4. Inquiries.
5. Summary Prints.
6. Dumps.

The first three categories of output involve data collected piece-by-piece during a simulation run. Then, after the run terminates, its history is available in these various forms.

Tracing a signal is equivalent to monitoring a real circuit with an oscilloscope probe. A complete time history of the signal is made available as output. Every time a signal specifically marked for tracing undergoes a transition, the value, time, and other circumstances surrounding the transition are recorded. Several alternatives are given the OLLS user in the choice of output medium for tracing results. The high-speed line printer can be used to show the trace of up to 20 signals simultaneously on the same time base. Each line of print output represents a time at which one or more of the traced signals changed state. The time is noted on the line, and each signal's value is noted by printing a number 1 in one of two particular columns. Thus if the first signal across the page has value zero, a number 1 is printed in column 14, and if the logic value is one it appears in column 16. Transitions are marked with a dash at the appropriate line in column 15. This type of output results in a common nonlinear timeline for each signal if the output is turned sideways for reading. The names of the signals being traced head their respective column groups on each page.

If the user is interested in elegance of appearance rather than large volume of output, he may choose to draw the trace output on an X-Y plotter. Samples of how this would appear are presented elsewhere in the report. This type of output is done with a linear time base for appearance's sake since generally fewer transitions are being displayed. Identical linear-trace output can also be displayed via the on-line CRT console if desired.

Signal sampling is different from tracing in the format of output and in the factors which cause the sampled data to be outputted. When a signal is being traced, the event which causes output to be created is a transition of that same signal. Sampling, on the other hand, is intended to produce output from one signal upon the occurrence of a particular transition in another. For example, one might choose to view the contents of a flip-flop by sampling it whenever a reading pulse occurs. Another reason to sample could be the transition of a timing pulse, thus establishing a linear history of the sampled signal for display. It would also be desirable to generalize the conditions causing the sample and to link more than one signal in a single sample event. Thus a general sample control card would be:

SAMPLE Siglist IF Expression

The Siglist term above is meant to be a list of any number of signals to be simultaneously sampled whenever the given signal expression becomes true.

The output format for a sample event as recorded on the line printer or CRT includes the time of the sample, a restatement of the control card, and the sampled logic values. In this way different samples and traces can be intermixed in the output medium without confusion.

The detection of logic-race conditions is an important adjunct to an OLLS simulation. Since OLLS is intended to mimic a circuit rather than to analyze it, no attempt is made to perform an algebraic-hazard analysis of the test circuit. Instead, hazard testing is performed dynamically as the simulation run occurs. Static and dynamic hazards manifest themselves as short-duration pulses on signals which should not have such pulses. Essential hazards are observed when a sequential circuit fails to execute the intended state-variable transformation and instead executes another. The first error condition is checked for on every signal every time it has a transition. The OLLS user declares what is the minimum acceptable pulse width in his circuit. Any violations of this figure are outputted as suggesting hazards. The signal at fault, time of occurrence, and causative factors are indicated. In addition, OLLS continuously tests for a potential source of such trouble by monitoring for coincidence and near-coincidence of signal transitions. Whenever two or more input signals to a device equation have coincident or near-coincident transitions but no actual hazard occurs in the output signal, a potential hazard area is outputted. This type of hazard might develop into a real one if the circuit were to be built out of real gates with wide variance in delay times - hence the interest in this situation.

Essential hazards are merely static or dynamic hazards which cause memory functions in the circuit to operate improperly. Hence, the occurrence of an essential hazard will generally follow the detection by OLLS of one of the other types of hazards.

In order to avoid bombarding the OLLS user with reams of hazard-detection data, the ability to ignore either type of hazard situation is built into OLLS. The user may specifically choose to exempt certain signals in the circuit from participating in the test procedures. All signals participate unless specifically excluded.

The subject of inquiries arises because OLLS is to be operated on-line from a CRT console. Between segments of dynamic simulation runs the user may raise questions about the immediate history and current state of the file. The most obvious inquiry is for the current and immediately past values of designated signals, and along with those values the times at which those transitions occurred. OLLS keeps at least the last ten transitions for each signal at all times, so meaningful data is usually available. The form of output can be either numerical print on the CRT face or sketches such as would be done for CRT trace display.

Queries, such as which signals are being traced, or which appear in sampling commands, are also useful. In addition, the user should be able to get information about his devices and circuit interconnections. He should be able to read the glossary for a device at this time, and may be interested in the extent of a signal run. The general intent of the inquiry operations (which have not all been defined as yet) is to provide the on-line CRT user with as much information retrieval as possible short of writing his own FILLIP coding. Considerable work still needs to be done in this area as new requirements arise in practice.

Summary printing, like inquiries, takes place between or after run segments or initialization procedures. The difference between these two operations is more quantitative than qualitative. Summary prints are oriented more to outputting entire lists of data such as the values of the entire signal set. However, the flexibility built into the inquiry-command structure should also be present for summaries. For example, a summary of all undefined-value signals should be possible, as well as one for all signals which experienced transitions over a given span of time. Again, the scope of commands has not been defined as yet, and future effort will be directed to make a useful set of summary-print commands.

It is worth noting that the OLLS programs are very modular in their form, and new features can be easily added to existing programs. Therefore, it is possible to postpone definition of these operations until more is learned about user requirements through experience.

One more useful output operation for OLLS is a total-signal dump. All data in a logic circuit necessary to define the simulation status of the circuit is recorded. This includes all signal values and histories, all trace, sample, and hazard test data, and the current image of the "event list". If this data is recorded permanently, it is possible at a later date to restore the logic circuit to the exact state it had at the instant the dump occurred. Thus the time-consuming processes which led up to its current state need not be repeated merely to reach the same state again. The dumped data would be sufficient to initialize the file completely and the run could be resumed at that point. This rollback facility is also reserved for future instrumentation since it is truly useful only when large circuits and long simulation runs are involved.

d) Run Control

The subject of run control applies primarily to dynamic-simulation segments. These are run segments in which time is a factor as opposed to the initial-value propagation segment in which time stands still while signals propagate. The initialization segment terminates when no further propagation

can occur, either because all signals have defined values or because those signals which remain undefined mask the effects of the defined signals. There is always a logical conclusion to that segment because of the algorithm chosen. The number of steps taken is always finite for a finite circuit.

This finiteness does not apply to the dynamic algorithm. It is perfectly feasible to have an infinite sequence of transitions occur at any signal. In fact, the periodic signals which can be inputted to OLLS logic circuits are themselves potentially infinite in their effective span of time. Thus a dynamic run must always have a definite finite limit to its time scope provided by the user each and every time it begins. If the user fails to provide a numeric time limit, OLLS derives one from the time sense of his circuit. Although the sequence of events in the circuit is potentially infinite, at any moment OLLS plans only a finite future portion of it. This partial plan of future events is stored on the circuit "event list". If the user fails to specify his time limit, the maximum extent of the event list is determined and becomes his limit.

It is also possible to terminate a run segment prematurely. To do this, the user specifies a Boolean signal expression which, when it becomes true, will halt the run. One reason to use such an expression is to halt on the occurrence of an alarm condition in the circuit under test. This preserves the signal values and histories at the moment the alarm occurs so that inquiries and summaries can be fruitfully performed. Thus, the run segment control card is as follows:

RUN Δt OR exp

where the segment terminates either after Δt or when the given expression becomes true, whichever comes first. If the number Δt is omitted, the event list default value is taken. If the "OR exp" is omitted, the given Δt value (or its default value) is the only condition for halting the run segment.

Every time a run segment begins, the circuit it applies to must also have some time history sense. This history can come from one of three sources -

a. Initialization.
b. Continuation.
c. Rollback.

When the simulation program is first called up for a circuit, the time sense of that circuit is initialized to zero. All runs following this normal sequence, therefore, have a common initial time value. After a run segment terminates, the final time sense is kept intact in anticipation of the next run segment being a continuation of the first. Only when a circuit is finally released from simulation

by the user and returned to bulk storage is its time sense destroyed. The rollback time initialization as described in Section 3.3.2.c can be performed before or between run segments, provided only that the necessary signal-value dump data is available to the user. Whichever source is the time sense of a run, the value of Δt given in the run control card simply follows it.

3.3.3 Program Details

a) Data Structure Review

The simulation programs operate on a circuit in an OLLS data file, the structure of which has been explored in depth. It is appropriate, however, to review those aspects of the data structure which are vital to the simulation program. There are four types of structures of particular interest:

1. Glossaries.
2. Instances.
3. Signals.
4. Simulation lists.

1) Glossaries

There is a glossary structure in the data file for every different type of logic device which the user has placed into the file. A glossary structure is formed when a logic device is defined, either through the Define Device program or through the amalgamation of several interconnected devices to form one larger device. It contains information pertaining to the device as a whole and to each individual terminal of the device. The simulation program uses only the terminal-oriented information. In particular, it refers to the following pieces of information for each terminal:

a. A FILLIP pointer which can be interpreted to locate the signal cell which is attached to the terminal on any instance of the device type. It takes two pieces of information acting in concert to use this pointer - one must know which particular instance of the device type is of interest and which of its terminals are involved.

b. A pointer to a FILLIP list which indicates the other terminals within the device that are successor to this one. When dealing with multi-input, multi-output devices, as OLLS allows, it is useful to know which terminals can possibly be affected by logic values on others. The successor list of a particular terminal indicates which other device terminals can be affected by changes in its value. The delays associated with these successor relationships are also given for each pair of terminals.

c. A pointer to a FILLIP list which is the result of compiling interpretable code for the terminal driving equation if one was given to the Define Device program. The form of this code is a parenthesis-free rephrasing of the given equation. Interpretation of the code is done with a pushdown store interpreter.

For example, if the given equation is

$$A = B + C + D$$

then the code list placed onto terminal A is

B, C, D, +, +.

When this is executed in order during simulation, the desired values of the signals on B, C and D are stored into successive positions in the pushdown list. Then C and D are logically OR'ed and the result placed where C was stored. Finally, the intermediate (C+D) is OR'ed with B and the result placed where B used to be. The total operation results in the correct value for the signal at terminal A being placed into the first cell of the pushdown list. Figure 3-14 illustrates the various steps in the process for the example.

Operation Performed	Fetch B	Fetch C	Fetch D	+	+
State of Pushdown List After Operation	B	C B	C B	(C+D) B	(B+(C+D))

Fig. 3-14

The value fetch portions of the compiled code actually consist of three pieces of related information. To be more specific, the operation implied by "fetch B" is actually stored in the code list as:

1. Fetch - activate the fetch routine.
2. Pointer to terminal B of this device.
3. Delay (Δt).

The fetch routine locates the signal actually on terminal B for this instance. It then searches the past history of that signal for a time appropriate

to the delay (Δt) given above, and retrieves the value associated with that time. This finally is the value to be stored into the pushdown list.

It should be noted that in the example where A = B + C + D, terminal A appears on the successor lists for terminals B, C, and D. There may be other entries on those successor lists, but they would result from other equations within the same device. Thus if, for example, E = B * C defines yet another terminal (E), then E also appears on the successor lists for B and C (but not for D).

2) Instances

There is an instance structure in an OLLS logic file for every physical element in the circuit described by the file. This includes all the logic devices in the circuit. The form of every instance structure is an exact replica of the model structure found in the glossary for its type; thus the forms of several instances of the same device type are identical. The contents of the instance structures are different, however, reflecting the fact that each individual instance is actually distinguishable from its brothers.

The instance structure consists of two parts - a FILLIP cell called the Instance-Head cell, and a group of cells for each terminal classification which indicates the interconnections from this device to other instances. The Instance-Head cell contains a unique identification number which distinguishes this instance from all others of the same type. This is referred to as the device ID. The cell also contains a FILLIP pointer back to the proper glossary structure for this type of device. This is the main link from instances back to glossaries, and is used extensively during simulation. The Instance-Head cell also contains pointers to each of the groups of cells from the various terminal classifications (input, output, expin, etc.), and is therefore the nucleus of the whole instance structure.

The terminal-cell groups, as described in Section 3.1 of this report, are variably structured FILLIP-pointer cell arrays. The size of each group, and therefore the amount of computer core storage occupied, is totally dependent on the numbers of each kind of terminal actually defined for the device. Each separately defined terminal requires room for a single FILLIP pointer to be available on every instance of the device type. When a particular instance of a device is incorporated into the circuit, each of its pointers is made to indicate the signal actually connected to that terminal. This is in the form of a direct FILLIP pointer to the Signal-Head cell which represents that signal. Note that the glossary for a device contains data for each of its terminals that uniquely locates the pointer to a Signal-Head cell on any of its instances. Hence, the location of any signal cell attached to any terminal of any instance of any type

can be determined through the proper glossary and instance structure, and knowledge of which terminal is involved.

3) Signals

An OLLS file contains a signal structure for every unique signal in it. This includes not only normal runs between a device output and input, but also expander-signal runs and auxiliary-terminal signals. A signal structure consists of:

a. Signal-Head Cell.
b. Source Link.
c. Destination-Link List.
d. Set Inclusion List.
e. Simulation History-Tape List.

a. Signal-Head Cell

As in the instance structure, the Signal-Head cell contains pointers to all of the other constituent elements of the signal structure, and is therefore its nucleus. In addition, the head cell contains three data items pertaining to simulation of the circuit. They are a set of simulation flags, the maximum required time span of the history tape, and the time at which the signal last changed value in the simulation. The use of these data items will be explored later in this section.

b. Source Link

The source link of a signal structure contains an indication of the device and terminal on that device where the signal originates. This would necessarily be an output, expander-output, or auxiliary-variable terminal of the source device. The form of the indication is a pair of FILLIP pointers; one directed at the appropriate instance structure, and one directed at the proper terminal cell within the source device glossary. The source link, therefore, provides all the information necessary to exploit the glossary to evaluate a signal's driving equation.

c. Destination-Link List

The destination-link list consists of cells identical to the source link. The data items indicated are all the terminals and devices on the signal run, except the source terminal. Thus the pair of items indicates all glossaries, instances, and terminals which are associated with a signal.

d. Set-Inclusion List

The set-inclusion list and the simulation flags are used to indicate and define membership of this signal in higher-level sets of signals. The kinds of sets indicated include:

1. Those signals being traced.
2. Those signals which participate in sampling expressions.
3. Those signals which participate in run control expressions.
4. Those signals which are not to be tested for hazards.
5. Those signals which are included in register set definitions.

The function of these various sets are clarified elsewhere in this report.

e. Simulation History-Tape List

The history-tape list of a signal consists of one or more special FILLIP cells. Each cell contains enough room for up to ten transition records. Each record is a pair of data elements - value and time. Whenever the signal under consideration changes value during the dynamic simulation run, this fact is recorded in the history-tape list. The existing tape contents at the time of the transition are pushed down one level to make room for the current transition record. The new value and time of occurrence are thus recorded at the top of this tape (or list).

As previously mentioned, the signal-head cell contains a numerical limit to the time span which must be covered by the history tape. As an older transition gets pushed down further from the current signal value, its time of occurrence is tested for obsolescence. Such obsolete segments of value history are discarded to prevent the growth of very long history tapes, and hence to conserve core storage for useful data.

4) Simulation Lists

Simulation lists are created immediately before and/or during a simulation run. They do not constitute a permanent addition to an OLLS data file, and so are jettisoned when the file is returned to an inactive state. This is done to reduce the required storage area for the file when the user is done simulating it.

All simulation lists are addressable from the root of the logic file, and are appended to the first FILLIP cell below that root. The kinds of lists which are involved are:

a. Sampling control commands.
b. Run control expression.
c. Event list.
d. Trace and sample output.
e. Hazard detector output.
f. Dump output.
g. Propagation error output.
h. Register set definitions.
i. Conditional input events.

a. Sampling Control Commands

The general sample control card is:

SAMPLE Siglist IF Expression

When this is inserted into an OLLS logic simulation, three data elements are added to the sample control list. They are the card image (for subsequent print output), a list of pointers to the signals mentioned in the Siglist, and a list of compiled interpretable code generated from the given signal expression. This code is slightly different from that compiled for device output terminals because it fetches only current signal values directly from signal-value history tape without passing through any glossaries enroute.

Each of the signals mentioned in the expression is marked by a flag bit indicating that fact, and is given a set-inclusion pointer back to this entry on the sample control list. Thus, when any signal mentioned in an expression changes, the simulation program can easily locate and evaluate the proper IF clause to control sampling output.

b. Run-Control Expression

The run-control expression is created when a Run-Control card containing a termination expression is inserted. This card appears as:

RUN Δt OR Expression

in the input decks. This expression is processed exactly like the sample control expressions and is stored in the same format. Naturally, maturation of a run-control expression has a different effect on the run than a sample control expression.

c. Event List

This list is the center of all simulation activity during a dynamic run. Its structure and use is explored in detail beginning on page 99 of this report.

d. Trace and Sample Output

When trace and sample events actually occur during a simulation run, the pertinent data is first outputted to this central list for later editing and printing. The output is in the form of one data cell per trace event, with indications by pointer of the signal being outputted and by value of the time and logic value being recorded. Sample events are recorded in the form of lists with a pointer indicating the sample control card initiating the output. The time of the event is recorded along with an ordered list of logic values corresponding to the desired sample-signal list.

e. Hazard-Detector Output

When hazard conditions occur during a run segment, the output indicating these events is placed into this list. Output is in a form which, for each recorded hazard, gives the signal on which the hazard occurred, the time of occurrence, and the nature of the hazard (real or potential).

f. Dump Output

Whenever a rollback dump is ordered by the user, this list is created from the data file. It consists of an ordered set of all signal-value history-tapes plus the current event list. The signal tapes are collected in a certain order and, when the file is re-initialized properly, are replaced onto the proper signal-head cells in the same order. The success of this operation requires that the signal-structure tree not be changed at all between dumping and re-initializing the file.

g. Propagation-Error Output

Errors which occur during the pre-dynamic run setup are recorded in this list. The output for each error is a pointer to the signal that reverted to an undefined status. Knowledge of the order in which the errors occurred is vital to any corrective action the user may take, so the list is carefully ordered in sequence of occurrence.

h. Register-Set Definitions

Although this feature of OLLS will not be included in the initial program release, a brief discussion of its merits is in order. The OLLS

user would be able to declare any set of signals to be a register. This fact would be recorded in this simulation list and in each included signal-head cell. Then he would be able to refer to the complete register set by name in any of his simulation control cards. The net effect would be, first, to free the user from defining sets of signals more than once in his input statements and, second, to provide more readily interpretable formats of trace and sample outputs for him. If a defined register should be traced or sampled during a run segment, output would be presented more concisely than for other signal lists which are not defined as registers. This would go far towards upgrading OLLS from a bit-by-bit simulator towards a register-by-register simulator.

i. Conditional-Input Events

The general form of a conditional-input event card is:

EVENT Signal Value IF Expression

When such a card is presented to OLLS the expression is complied exactly like the expression on a Sample-Control card. An entry containing the compiled code, a pointer to the signal to be changed, and the new value for that signal is placed in this list. Each signal mentioned in the expression is so marked and given a pointer back to this entry. Henceforth, whenever those signals change, the expression is evaluated to determine if the conditional event is to occur.

b) Predecessor and Successor Signals

The concepts of predecessor and successor signals are used in the OLLS simulation algorithms, and therefore should be fully understood. The two terms are closely related and can be defined together.

One must recall that there are two levels of logic construction in an OLLS circuit - the interconnections among devices and the devices themselves. A signal which is attached to an output terminal of a device is a successor of all those signals which enter into the output-terminal driving function directly. Similarly, a signal which is used directly to compute the driving function for some output terminal is a predecessor of that output signal. In the example of a simple NOR gate device, the NOR gate output signal is successor to all its input signals, since all the input terminals appear in the output-driving function.

Next consider a multi-input multi-output device in which only some of the inputs affect each output. Then only those signals that are connected to device terminals which interact directly through the driving equations have a successor-predecessor relation.

In the special case of a device wherein an output-terminal function includes itself in its equation, the signal connected to that terminal is its own successor and predecessor. Should the interaction between a signal and its past history be expressed through an intermediate equation, the self-successor property disappears. Thus, in the example of an oscillating terminal:

$$A_0 = \overline{A}_{10} \cdot B_2$$

The signal on A is successor to both itself and the signal on B. This same behavior can be modeled by a pair of terminals:

$$A_0 = \overline{C}_5 \cdot B_2$$

$$C_0 = A_5$$

In this case A is successor to B and C, while C is successor to A. Note that, although the behavior of terminal A for these two models during simulation is identical, the successor-predecessor relationships are different.

c) Initialization Algorithm

Initialization of an OLLS logic circuit usually consists of two separate phases. The first is the planting of explicitly stated initial-signal values throughout the signal-head cell tree. This is done in response to input cards such as:

INITIALØ Siglist

or

INITIAL1 Siglist

As each of the mentioned signals is set to the indicated static value, it is also entered into a list to provide data for the next phase.

The second phase of the process is initiated by entering the command card:

PROPAGATE

The state of the logic file at that moment is as follows:

1. The explicitly initialized signals have the correct values.
2. All other signals are currently uncontrolled in value. They may be all undefined or may have been left in some other logic state by an earlier run segment.

3. There is a list (Setlist) of all the signals which have heretofore been explicitly initialized.

The propagation proceeds by using the Setlist as a source of data and also as a destination of computed results. For every signal mentioned in the Setlist, the following operations are performed:

1. Each of the signal's successor signals is located and evaluated by its terminal driving function. The evaluation assumes time to be frozen so no account of delay is made. All signal values are taken to be current and unchanging.

2. If a successor's newly computed value agrees with its previous value, no further effort need be made. If, on the other hand, the old and newly computed values differ, then one of two additional steps must be taken. If the old value was undefined and the new value is Zero or One, a normal initialization has occurred. In this case the new value is placed in the successor signal's value history tape. Since the successor signal has now changed value, its name must be added to the original Setlist. This is performed before going on to evaluate the next successor of the signal being propagated. The Setlist thus grows to indicate implicit signal-value initializations.

The second case to be considered for propagation is the one in which data is actually lost during the process. If the old value of a successor signal is Zero or One but the newly computed value disagrees with this, an indication of error has occurred. Generally this happens because the user has requested an inconsistent or inadequate initial-value array. Such a signal is forced to revert to an undefined value to prevent unending computations. The signal's name is added to a list of other such initialization errors kept in the simulation-list area of the file. Finally, the signal at error is also added to the Setlist to propagate, if possible, the newly undefined signal value still farther into the file.

3. When all successors of a signal mentioned on the Setlist have been evaluated by the rules of 1 and 2 above, that signal is deleted from the Setlist. The propagation program then steps on to consider whatever signal happens to be next in the list. The order in which signals are considered has some effect on the length of time that the entire process takes, but none on the final value configuration achieved.

d) Event List

As far as the initialization procedure in OLLS is concerned, the passage of time is not a factor. However, the dynamic-simulation technique used to achieve realistic circuit behavior requires a time base to operate successfully. The function of the event list is to provide the necessary time base to the simulation.

The event list is a time-ordered list of all events which are scheduled to take place during simulation. Its structure can best be described as like that of a comb - with time advancing along the back of the comb and events recorded on its teeth. Simultaneous events, if there are any, are recorded on the same tooth. The spacing between teeth is not uniform, however, since the presence of a tooth indicates a time at which an event is actually scheduled rather than a time at which one might be placed. Each tooth position is marked to indicate the simulation time it represents, and it is possible to interject new teeth between existing ones should the need to do so arise.

There are presently two sub-lists on each tooth which may each contain an unlimited number of simultaneous related events. Of the two lists presently utilized (out of a possible twelve lists), one is concerned only with evaluation of signal values and the other only with the actual transitions experienced by signals. They are called, respectively, EVLIST and DOLIST.

The dynamic simulation proceeds by stepping along the comb. All scheduled events at each tooth must be executed before advancing to the next tooth in sequence. When time does advance after successfully executing all events on one tooth, it does so by jumping directly to the next time tooth in place. Time thus increases monotonically, but not necessarily in unit increments. The gap between two teeth may be any positive integer on a total available time scale of 1,048,576.

When a user first calls for the simulation program to operate on his circuit, the event list is necessarily empty. Any unconditional input events or repetitive sequences (see page 82), which he may supply, form the initial event list. These are placed on the DOLIST in their correct tooth positions. Repetitive sequences have all discrete transitions which will occur during only the first period placed into the event list at this time. Subsequent transitions on these and other periodic signals will be added to the list no more than one period before they occur. The event list is, therefore, always finite in length, even in the presence of repetitive sequences.

During the course of a run segment the event list grows into the future and contracts from the past as signal-related events occur and in turn breed new events. Thus, at any moment, the list indicates the schedule of imminent events as generated by the simulation algorithm.

e) Dynamic-Simulation Algorithm

The dynamic-simulation run follows a course dictated by entries in the event list and by the logic constraints imposed by the user's circuit. All computation and output occurs with time frozen at a particular tooth in the event list. The behavior of the logic circuit, propagation of signals into the future, and sampling and tracing snapshots all take place with time frozen. The algorithm through which the behavior of a logic circuit is simulated thus is actually an ordered sequence of computations which take place with time halted. Whenever an event must take place in the future rather than instantaneously, it is seeded into the event list at a point in time which is not yet under consideration. Any references to the past must be made within the signal structures themselves (history tapes), since those portions of the event list which have already been processed are discarded.

The dynamic-simulation algorithm begins by interrogating the current event-list tooth. First, the DOLIST is processed. Each entry on the DOLIST is a data-pair-signal pointer and new value. For each such entry on the current DOLIST the program:

1. Locates the indicated signal's value history tape.
2. Pushes down old values to make room for the new value and current time.
3. Discards any obsolete old values for the signal. A value is obsolete only if its time of occurrence is older than the oldest which needs to be kept for that particular signal.
4. Locates, through the destination-link list and the glossary for each destination device, the successor signals of the one being changed.
5. Each successor signal has a specified delay associated with its response to the particular predecessor signal being changed. This information is also extracted from the destination-device glossaries.
6. An entry is prepared for the event list for every successor of the signal just changed. Each entry is made into the EVLIST of the tooth appropriate to the delay extracted from the glossary. If an extracted delay happens to be zero, the EVLIST entry is made on the event-list tooth being processed. If no tooth exists at the appropriate time in the near future to accommodate an EVLIST entry, one is created and the entry made. The entry presently consists only of a pointer to the successor signal-head cell, although additional data could be included. For instance, the predecessor signal which stimulated the placement of the successor into the event list could be named again, thus allowing cause-and-effect behavior to be studied more closely.

When the DOLIST is processed, it is jettisoned as excess baggage. The data file has been changed only to the extent that those signals which were mentioned on the DOLIST have their new values. The event list has many new EVLIST entries on various teeth, representing all the successors of all the signals which have actually changed.

Next the current EVLIST is examined for any entries. If there are any, the program must compute their current logic values. For each entry on the current EVLIST, the program:

1. Locates the mentioned signal-head cell and source-link cell.
2. Refers to the glossary for the source device of that signal and executes the output driving function for the terminal to which this signal is connected.
3. The computed value is compared with the present value on top of the signal's value history tape. If they are the same, no further action takes place.
4. If the two values differ, an entry is prepared for the current DOLIST, indicating the signal to be changed and the computed new value.

When all entries in the current EVLIST are processed, the EVLIST is jettisoned as obsolete. The program then examines the current DOLIST to see if any entries have been created in it. If there are any, it repeats the processing described for the DOLIST. If not, it has finished with the current time tooth and steps on to the next one in the sequence.

The sequence of operations just described is the heart of the dynamic-simulation algorithm. The program separates all nonsimultaneous events into distinct processing stops. Then, within a time step, it further separates events into evaluation of signal values and execution of computed changes. This separation permits us to handle zero-delay logic so long as there are no oscillatory logic loops. If such a situation exists, the program will never be able to finish its computations at one time step. It will be rapped back and forth between EVLIST and DOLIST in a nonconverging process. The presence of any delay in the loop eliminates this problem entirely.

f) Tracing, Sampling and Hazard Detection

The preceding section described that portion of the dynamic-simulation algorithm which causes signals to change value and time to advance. There are, however, a number of frills that accompany the signal-changing process which are required to make OLLS a useable program. The subjects of output processing, run control, and hazard detection must be merged into the dyanmic

algorithm to create a user interface and to present results of a run to the user for inspection.

Since all these extra features of OLLS are based on actual signal-value changes, we need only consider the DOLIST processing that occurs at an event-list time tooth. Each time a signal value is actually changed, the program consults the simulation flag bits in that signal's head cell. These indicate whether the signal is to be traced, tested for hazards, or considered as part of a sampling, run termination, and/or conditional-input event-control expression. If any of these flag bits indicates the need for additional processing, the set-inclusion list of the signal being changed is extracted and added to a third list (in addition to EVLIST and DOLIST) on the current-event tooth. When all processing of EVLIST and DOLIST at a particular tooth is complete, this third list is consulted before actually permitting the program to step along to the next event time. All the operations indicated are performed now. The required trace-output data is placed into the simulation lists. Control expressions are evaluated to determine if sampling is to occur, if input events are to be added into the event list, or if the run is to terminate prematurely. Hazards are tested for by measuring the time that elapsed between transitions on a changing signal, and comparing pulse widths with prescribed limits. Finally, after all the extra processing is complete, the program examines the time of the next-event tooth and compares it to the established run-segment limit value. If the Δt limit will be exceeded by stepping on to the next time tooth, the run is stopped, thereby returning control over the simulation to the user.

3.4 The OLLS 360 Drawing Algorithm

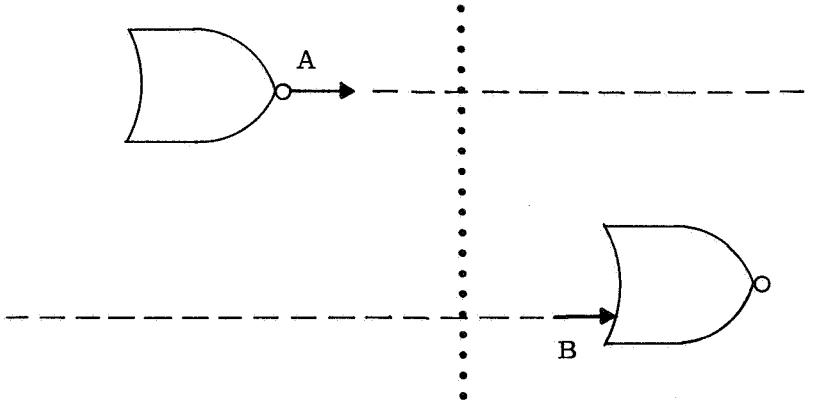
H. Robert Howie

Experience with the H1800 system described in Chapter 2 has shown that very few interconnections are ever made with more than five straight-line segments. Most connections, in fact, are made with one or three straight-line segments. The H1800 system was designed to find any connection with as many as nine straight-line segments and then try to reduce that connection to as few segments as possible. This naturally requires computer time and sometimes doesn't reduce the connection far enough.

With this in mind, a better approach would be to try the simplest pattern first and proceed upward through more complicated patterns as necessary until a connection is made or until more than five straight-line segments are required for the connection. The limit of five segments is purely arbitrary. Complicated connections are easier for the designer to follow if labeled with the signal name than if connected by a maze of twisting line segments. To aid in this approach all possible connections are classified by the patterns shown in Fig. 3-15.

In Fig. 3-15 notice that the five basic patterns correspond to the number of segments required to connect the source at A to the destination at B, and that each pattern has two forms depending on whether the exit direction from A is horizontal or vertical. The required pattern to connect a particular pair of devices can be determined by elimination fairly rapidly by asking a few simple questions about the terminal characteristics of the device.

To illustrate this, consider the example below.



The exit direction from terminal A is horizontal so the Y forms of all five patterns are eliminated. The entrance direction to terminal B is also horizontal so patterns II X and IV X which require a vertical entrance to B are also eliminated. A first attempt is made to fit pattern I X. The Y coordinate of A and B are not identical so

we proceed to try pattern III X. This is done by extending A as far forward as possible, extending B as far backward as possible, and then looking for a single straight line which connects them. If such a line can be found, the algorithm ends; otherwise proceed to pattern V X. If pattern V X fails, the connection is simply labeled with the signal name.

Had terminal B been vertical (such as the Direct Set or Reset on a JK Flip-Flop), patterns I X, III X and V X would have been eliminated, and only patterns II X and IV X would be considered.

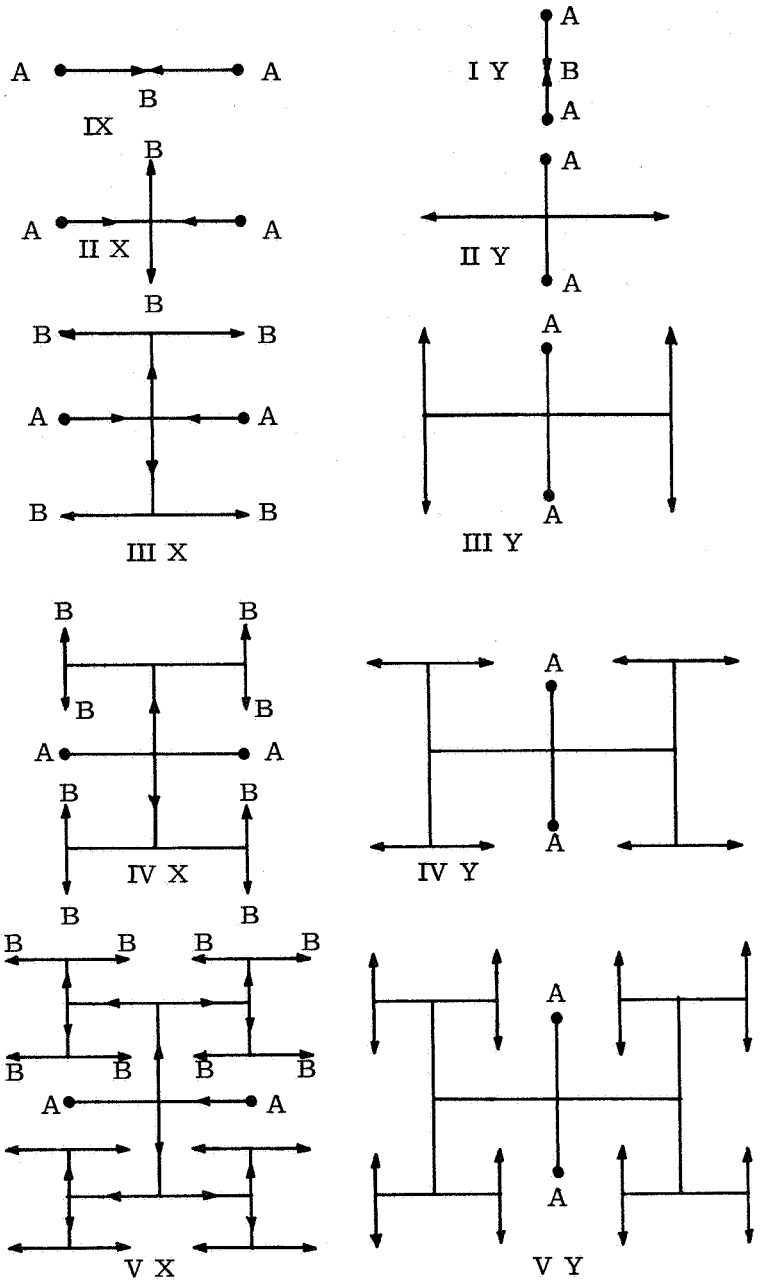


Fig. 3-15

3.5 Program Structure

James Pennypacker
Gary Schwartz

To manipulate the data in the data file, a number of separate programs have been written. Each of these programs is independent of the others but there is a structure which ties the various routines together; this structure is shown in Fig. 3-16. The intent of this section is to describe the functions of each of the routines and to indicate how the various programs are logically inter-related.

Briefly, there is one main program of OLLS; this main program reads and interprets all OLLS input cards and, depending upon the input-card contents, branches to one of the working programs. Figure 3-16 depicts five different working programs, each of which is identified by an asterisk; other working programs, such as drawing generation and deletion, are not explicitly shown. In general, the working programs provide the means for manipulating data in the data-file structure.

In Fig. 3-16, there are two references to a program called DEFINE DEVICE. Even though two different programs are indicated in the illustration, both rectangles refer to the same physical program; there is only one program called DEFINE DEVICE. The same applies to the programs ADD TO LOGIC FILE, CHANGE LOGIC FILE, SIMULATE, and CARDFILE.

3.5.1 Types of Input Cards

As mentioned in the previous section, there are really three types of input cards which are of interest to OLLS. First, there are the macroinstruction cards which call into operation either the working programs or the on-line CRT system. The macroinstruction cards are recognized by the presence of an asterisk (*) in the left-most column; for this reason they are often referred to as asterisk cards. Examples of macroinstruction cards are

```
*   DEFINE DEVICE type
*   ADD logic file name
*   CHANGE logic file name
*   SIMULATE
*   CRT
```

The second class of cards is composed of subinstruction cards which describe the specific operations to be performed by the working programs. Subinstruction cards all contain certain key words which are recognized by the appropriate working program.

The third class of input cards is formed by the actual data cards which are required for specific operations.

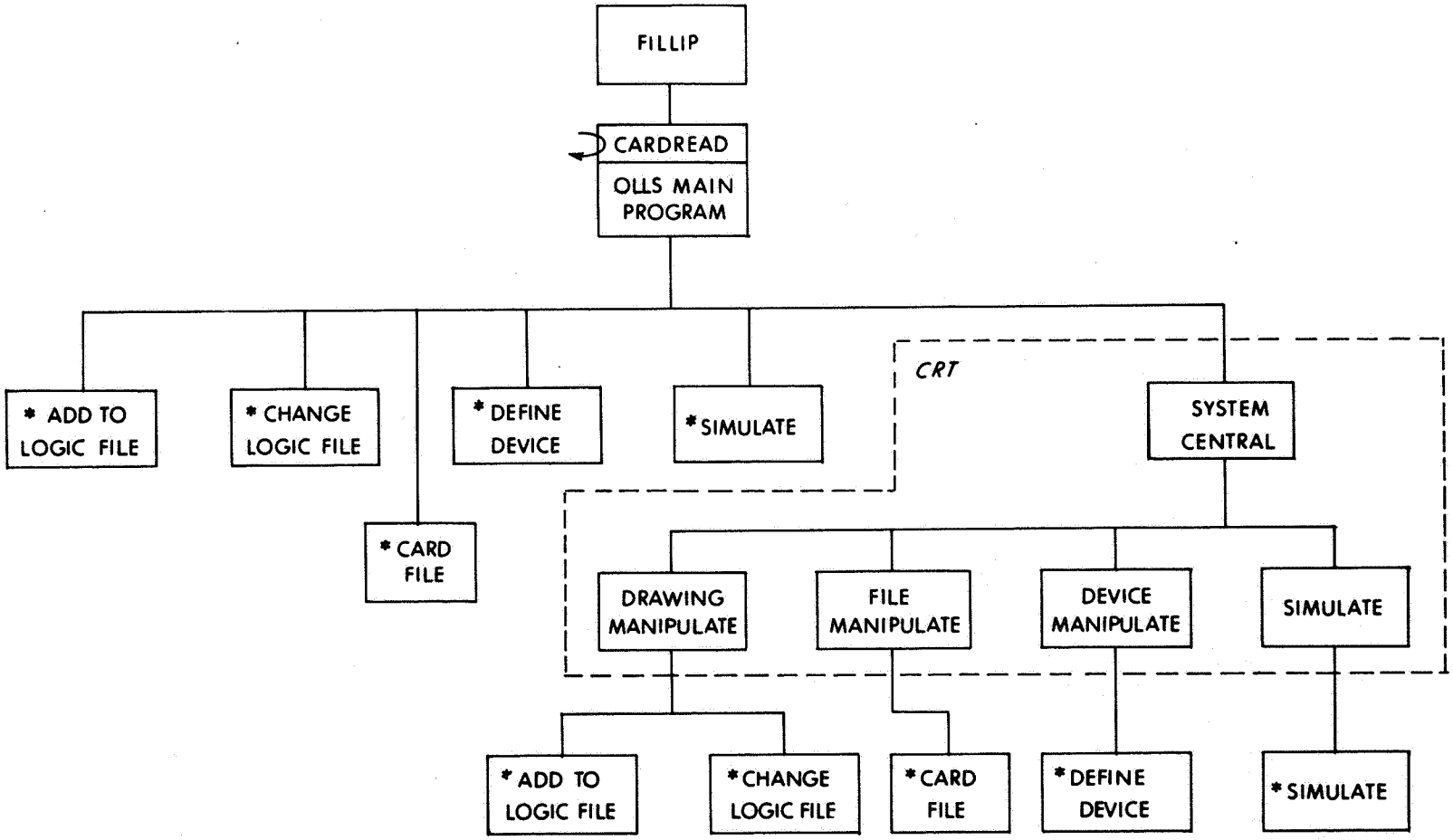

FILLIP
CARDREAD
OLLS MAIN PROGRAM
* ADD TO LOGIC FILE
* CHANGE LOGIC FILE
* DEFINE DEVICE
*SIMULATE
CRT
SYSTEM CENTRAL
* CARD FILE
DRAWING MANIPULATE
FILE MANIPULATE
DEVICE MANIPULATE
SIMULATE
* ADD TO LOGIC FILE
*CHANGE LOGIC FILE
*CARD FILE
*DEFINE DEVICE
*SIMULATE

Fig. 3-16

Generally speaking, a number of different data cards immediately follow each subinstruction card; all data cards are serviced by the subinstruction card until a new subinstruction card is read. Furthermore, a number of different subinstruction and data cards normally follow each macroinstruction card; the macroinstruction card maintains control until the next macroinstruction card is read.

3.5.2 CARDREAD and Main Program

Insofar as it controls the operations of the various working programs, the central, or main program, of OLLS is the program CARDREAD. This program operates in two modes; as the main, or controlling, program or as a subroutine called by the working programs.

All input cards, regardless of type, are read by the CARDREAD program. When operating as the main OLLS program the operation of CARDREAD is essentially as follows: CARDREAD scans each input card until a macroinstruction card is found. The image of the card is remembered by CARDREAD and some minor bookkeeping operations are performed. The appropriate logic-data file is fetched from the FILLIP monitor. Control is transferred to the appropriate working program; this transfer includes transferring the image of the macroinstruction card and the root of the required data file. CARDREAD remembers which data file is being processed and which operating program has control.

Each working program may in turn call CARDREAD as a subroutine to read and deliver the subinstruction cards and the data cards required by the working program. When operating in this mode, CARDREAD scans the input card to determine whether or not the card is a macroinstruction card. If it is not, the image is delivered directly to the working program. If the card is a macroinstruction card, control of operation is transferred to a different entry point of the working program which performs only clean-up operations for the working program. No card image is transferred by CARDREAD to the working program at this time; however, the card image is remembered by CARDREAD. When the clean-up operations are completed, the working program must transfer control to the CARDREAD in its main program mode. CARDREAD now scans the image of the macroinstruction card which has just been read; the appropriate data file is fetched, the bookkeeping is updated, and control is transferred to the appropriate working file as before.

If the macroinstruction card does not contain a logic file name, the file which was named on the previous macroinstruction card is assumed. The bookkeeping operations include keeping an up-to-date list of all OLLS files and users, remembering which data file was last named, and remembering which operating program is in control.

3.5.3 DEFINE DEVICE

The device definition program has been described in Section 3.2 of this report.

3.5.4 CARDFILE

One of the service routines incorporated into OLLS is the CARDFILE system. Using the system, the designer can store in the FILLIP data files lists of input card images. The CARDFILE system will deliver the card images, one at a time and upon demand, for further processing. The CARDREAD program is designed not only to read cards from the computer card reader but also to read card images from the CARDFILE system; thus, either entire computer runs or portions of runs can be performed, using data and control cards which have been stored in memory prior to the run.

A CARDFILE EDIT program enables the user to modify the image of any particular card which has been stored. Cards may be deleted from the image list and new card images may be inserted into the list at arbitrary points. The editing of the card files may be performed either in the batch-processing mode, using input cards, or in the on-line mode, using the typewriter keyboard of the on-line graphic terminal. This feature permits the designer to control from the on-line keyboard those runs which are made up of batch-processing segments interspersed with on-line activity; the control cards of the segments can be modified as desired, depending upon results of previous activity.

There are two foreseeable major uses of the CARDFILE system. When a data-file structure is composed of thousands of input cards, as is the typical case for a practical design, it becomes unwieldy for the user to work with such large input decks. Using the CARDFILE system, the designer can in one job store all the card images; he can in the future edit any of these images to make corrections, additions, or deletions. If the designer then wants to make a trial design which is a slight variation of the first design, he can duplicate the card image list in memory and make the necessary modifications to the duplicated list. The designer now has two input lists in memory, one for each design; these input lists become controlling documents for the design.

The second major use of the card file system occurs during simulation conducted on-line. The control cards for performing simulation can normally be prepared prior to the on-line run and stored as an image list. After the user has designed a logic circuit in the on-line mode, additional data-card images must be prepared and inserted into the image list if the circuit is to be simulated. This can be done at the graphic terminal keyboard; the card-file image list will then control the simulation.

3.5.5 ADD

The ADD TO LOGIC FILE program provides the means of inserting design information pertaining to a particular device into the logic-data file. Prior to inserting design data, a glossary for the device type must have been constructed by the DEFINE DEVICE program. For each individual device, the following information may be inserted into the logic-data file:

1. Device type.
2. Device identification.
3. Identification of drawing on which device appears.
4. Coordinates of drawing where device is located.
5. Names of signals which are connected to each of the device terminals.

It is not yet known how graphical data is to be represented on punched data cards.

Generally speaking, one card image is used to insert all the data relevant to a particular device; continuation cards may be used to contain additional signal names if required.

The program checks each input data card for consistency with the existing data structure. The device identification is checked against the identification of all other devices of the same type; if it is a duplicate identification, an error message is produced in hard copy or on the CRT, depending upon the mode of operation. The coordinates of the drawing number are checked to assure that other devices have not been positioned at the same location. If the drawing number does not yet exist in the data file, a new drawing-head cell with the desired number is automatically generated and inserted in the data structure. The signals on the output terminals of the device are checked to determine whether or not the signal is connected to any other output terminal; this is not allowed. Signal-head cells for new signals are automatically created and inserted into the data file. Finally, signals which are connected to no output terminals are identified as sourceless signals and brought to the attention of the designer.

3.5.6 CHANGE

Where the ADD program provides for inserting additional data into the data file, the CHANGE LOGIC FILE program enables the designer to modify via punched cards any piece of design data which has been inserted into the data-file structure. The following types of modifications can be made under control of the CHANGE routine:

1. The name (identification) of a drawing, signal or device instance may be changed.
2. A device may be moved to a different drawing or to a different location on the same drawing.
3. A device may be changed to one of a different type.
4. Signal connections may be made or broken.
5. The size of a drawing can be changed.

In all relevant cases, consistency checks are applied to the modified data as was described for the ADD routine. All data not specifically mentioned on the input cards to the CHANGE program remain unchanged.

When a device type is changed to a new type, the old and new types must have the same number of inputs, outputs, expander inputs, and expander outputs. Signals for auxiliary equations are automatically added (or deleted) as needed for the new device type.

When a device is moved from one position to another (or to a different drawing at the same positions), the graphic data which described the former signal run is deleted from the file structure. Graphic information for all runs of the signal on the original drawing are deleted.

3.5.7 SIMULATE

The simulate program is discussed in detail in Section 3.3 of this report.

3.5.8 CRT

The CRT system is described in detail in Section 3.6 of this report. It is sufficient to state here that, in order to change or insert data from the CRT, the corresponding control and data card images are generated by the CRT systems and read by CARDREAD exactly as if the cards were coming from a card file or the computer card reader. Thus, the same working programs are called into operation from the CRT system as are used in the batch-processing system.

3.5.9 DELETE

The DELETE FROM LOGIC FILE program enables the designer to remove unwanted data from the data-file structure. Because the CHANGE routine allows individual data to be modified by the designer, the DELETE program operates in toto on all the design data included on the original ADD input-data cards. Specifically, the DELETE program requires only that the type and identification of the device which is to be deleted be given. The device and all data pertaining to the device are deleted from the data file. Signals which are connected to the outputs of the device are eliminated from the data file rather

than being classed as sourceless signals. If the device is the last device on a drawing, the drawing itself is eliminated from the data file. Auxiliary signals required by the device are also deleted. The schematic position of the device becomes available for placing another device of the same size. Finally, all signals which were connected to the inputs of the device are appropriately modified and the graphic data - as it relates to the drawing from which the device was removed - for all those signals is deleted from the data structure.

3.5.10 DELETE TYPE

Where the DELETE FROM LOGIC FILE routine enables the user to delete design information relative to a particular device, the DELETE TYPE program provides for deleting a glossary structure from the data file. A prerequisite for this operation is that all instances of the specified type must have been deleted prior to the deletion of the glossary. Otherwise, the glossary structure is not deleted from the data-file structure and an error message is brought to the user's attention.

3.6 On-Line System (CRT)

H. Robert Howie
Ramon Alonso

3.6.1 Introduction

This section describes in detail a portion of the interaction between a designer and OLLS. Because this is a written report, the procedures may appear cumbersome and lengthy, but are, in fact, quite straightforward.

The description carries only up to the point at which the designer has established a working file, and has collected the various devices and circuits he expects to use. Length of text alone makes some limitation of the description mandatory.

As far as implementation is concerned: as of this writing SYSTEM CENTRAL and FILE MANIPULATE menus are working as stand-alone programs, ready to be tied to the rest of the system. The DEFINE DEVICE menus are half-done, with the DRAWING MANIPULATE menus in the planning stages.

3.6.2 Physical System

The equipment system used to implement OLLS is composed of an IBM 360/75 computer and a model 2250 Graphic Display Console. The console has a CRT, a light pen, and a keyboard. An internal memory relieves the computer of the task of regenerating the display the required 30 times per second, for flicker-free appearance.

3.6.3 Concept of User's Role

Experience, both ours and that of others, indicates that it is desirable to use the light pen to choose from among alternative actions, rather than typing in commands. If, at every step of the way, all possible alternative actions are displayed on the CRT, the operator is relieved of the task of memorizing exact spellings and formats, and can choose an action by pointing at it with his light pen. This sort of system has strong overtones of self-instruction and, as will be seen, offers a "natural" progression through the capabilities of OLLS.

Typing cannot be entirely avoided; there are instances, such as when inventing names, when typing is clearly superior to writing characters with a light pen.

3.6.4 Procedures. Setting Up a File

The first thing to happen after calling OLLS is the display of the OLLS SYSTEM CENTRAL menu*, (Fig. 3-17). The designer can at this time select the major OLLS mode by pointing the light pen to the box next to the desired option.

As shown in Fig. 3-17, the designer is confronted with an empty file. Assume he is about to set up shop for the first time. Other designers before him have files, and there are common files from which he can draw information. Still in Fig. 3-17, the designer selects FILE MANIPULATE.

In general, the user can work simultaneously with two distinct data files. The data file which he is constructing or inserting data into is called the Working File. Rather than generate new data, the user can copy information into the working file from any other existing OLLS file. The file from which information is copied is known as the Read Only File; the user is not permitted to write data into this file or to change its contents in any way.

The result of that selection is shown in Fig. 3-18, where the FILE MANIPULATE menu is shown. The selection of possible commands is shown in the upper half, and the active file list is displayed in the bottom half of the screen.

One of the possible commands is RETURN TO SYSTEM CENTRAL, which causes Fig. 3-17 to return to the screen. In general, it is possible at each step to go back to a previous one.

At this time, when Fig. 3-18 first appears, our designer has neither a working file nor a Read Only file. He points to SELECT READ FILE (1), then selects the COMMON file (2), then says EXECUTE (3). The result of the EXECUTE appears in Fig. 3-19. Notice that the top line now shows a Read Only File (common), whereas none was shown before.

The designer now has a set of device definitions to work with, and wishes to set up a working file (Fig. 3-20). He does so by selecting CREATE FILE (1), and typing in the name of the file. He is naming his file SMITH DEMONSTRATION FILE. The name of his working file appears next to the label WORKING FILE (2). When he finishes typing, he can command EXECUTE (3), which enters the new file names in the ACTIVE OLLS FILE LIST (Fig. 3-21). Typing errors can be corrected (before EXECUTE) by way of the keyboard itself.

* The term "menu" describes quite well the character of the control system for OLLS. The designer selects items from the menus in front of him to command desired actions. The menu naturally displays only those which are relevant at that time.

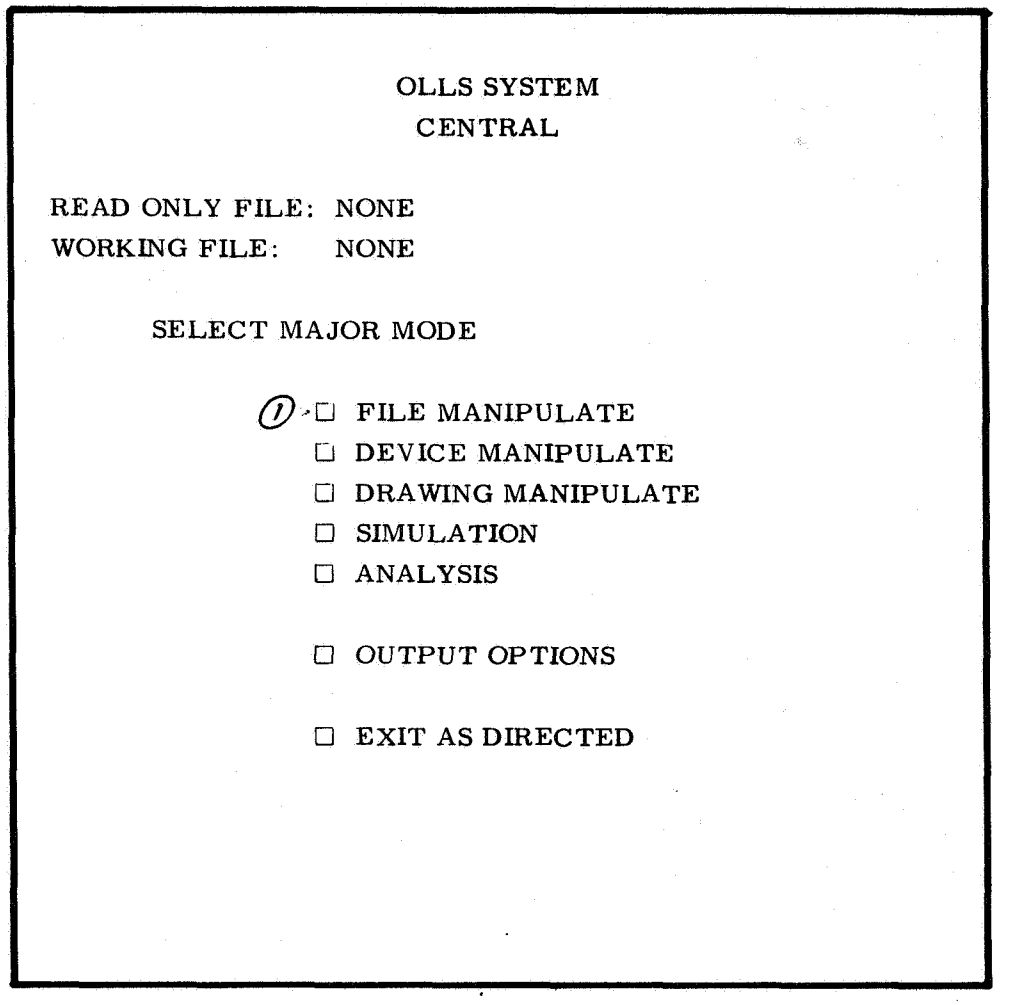

OLLS SYSTEM
CENTRAL
READ ONLY FILE: NONE
WORKING FILE: NONE
SELECT MAJOR MODE
1
FILE MANIPULATE
DEVICE MANIPULATE
DRAWING MANIPULATE
SIMULATION
ANALYSIS
OUTPUT OPTIONS
EXIT AS DIRECTED

Fig. 3-17

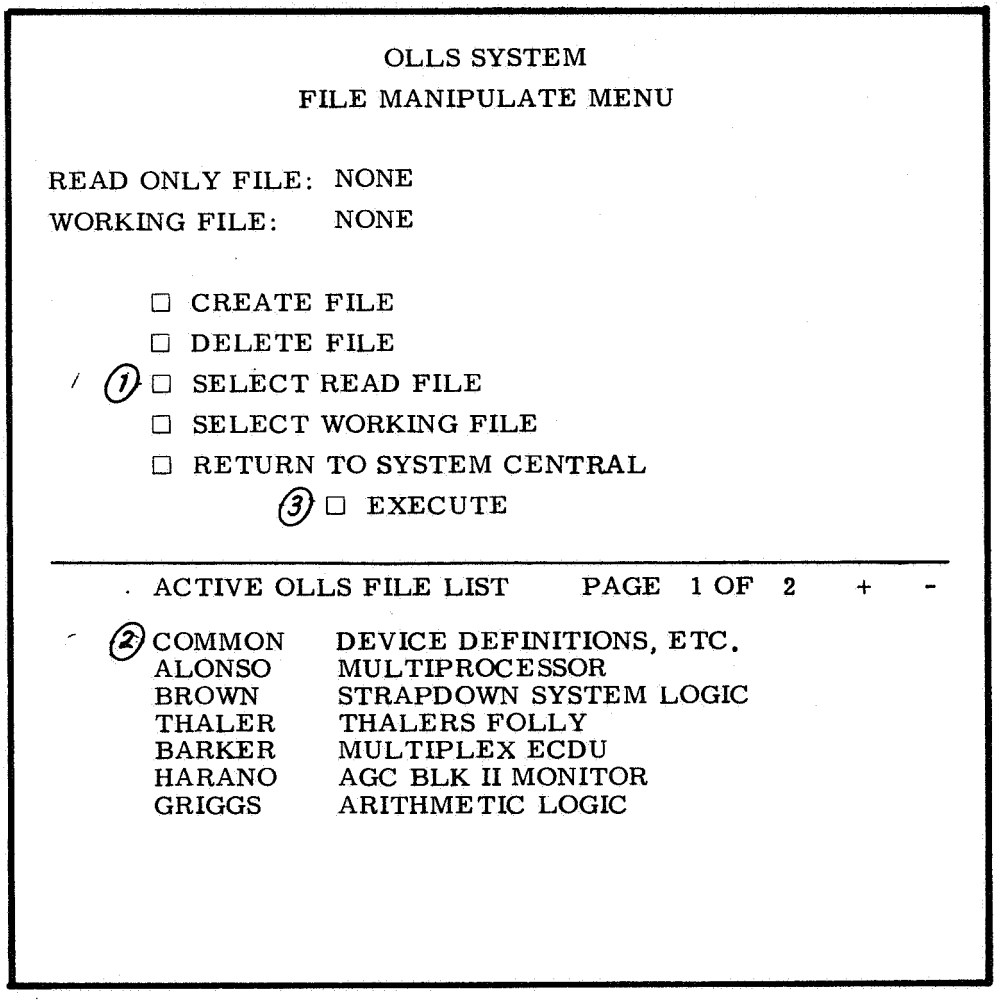

OLLS SYSTEM
FILE MANIPULATE MENU
READ ONLY FILE: NONE
WORKING FILE: NONE
☐ CREATE FILE
☐ DELETE FILE
(1) ☐ SELECT READ FILE
☐ SELECT WORKING FILE
☐ RETURN TO SYSTEM CENTRAL
(3) ☐ EXECUTE
ACTIVE OLLS FILE LIST PAGE 1 OF 2 + -
(2) COMMON DEVICE DEFINITIONS, ETC.
ALONSO MULTIPROCESSOR
BROWN STRAPDOWN SYSTEM LOGIC
THALER THALERS FOLLY
BARKER MULTIPLEX ECDU
HARANO AGC BLK II MONITOR
GRIGGS ARITHMETIC LOGIC

Fig. 3-18

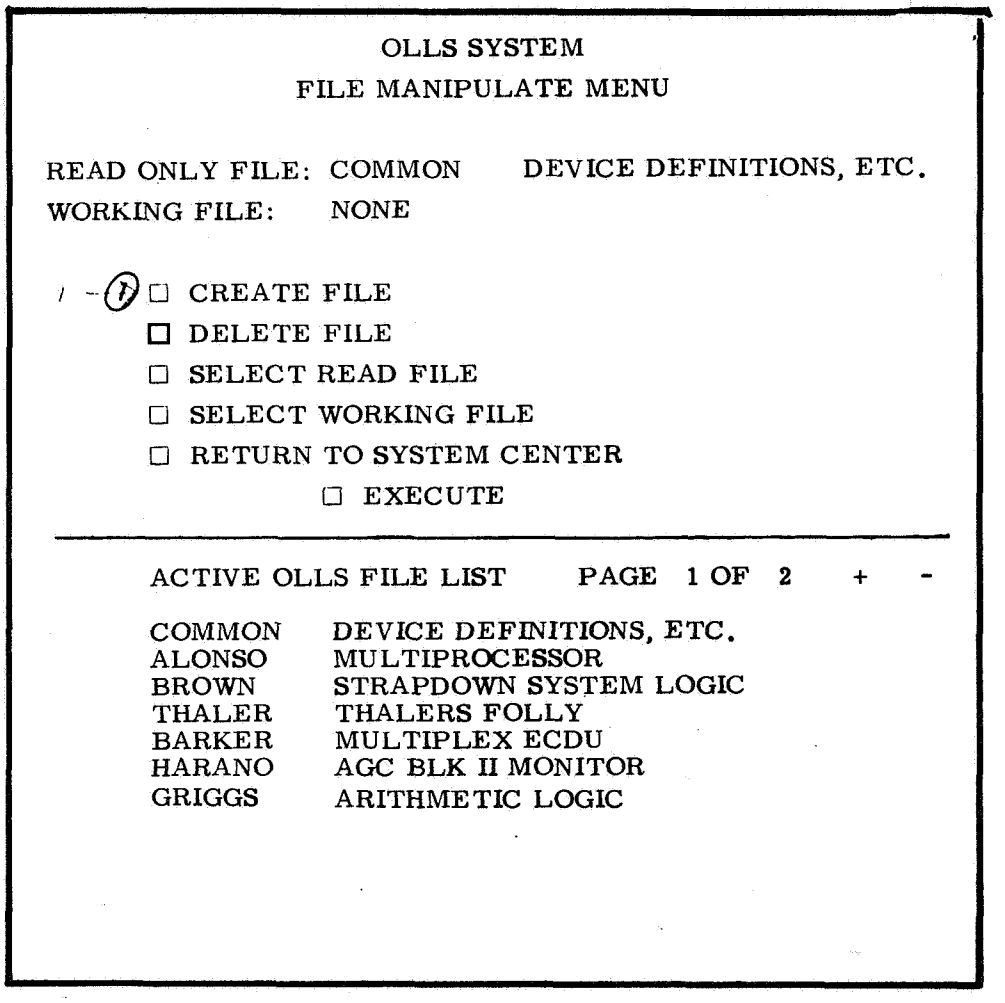

OLLS SYSTEM
FILE MANIPULATE MENU
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: NONE
1
CREATE FILE
DELETE FILE
SELECT READ FILE
SELECT WORKING FILE
RETURN TO SYSTEM CENTER
EXECUTE
ACTIVE OLLS FILE LIST PAGE 1 OF 2 + -
COMMON DEVICE DEFINITIONS, ETC.
ALONSO MULTIPROCESSOR
BROWN STRAPDOWN SYSTEM LOGIC
THALER THALERS FOLLY
BARKER MULTIPLEX ECDU
HARANO AGC BLK II MONITOR
GRIGGS ARITHMETIC LOGIC

Fig. 3-19

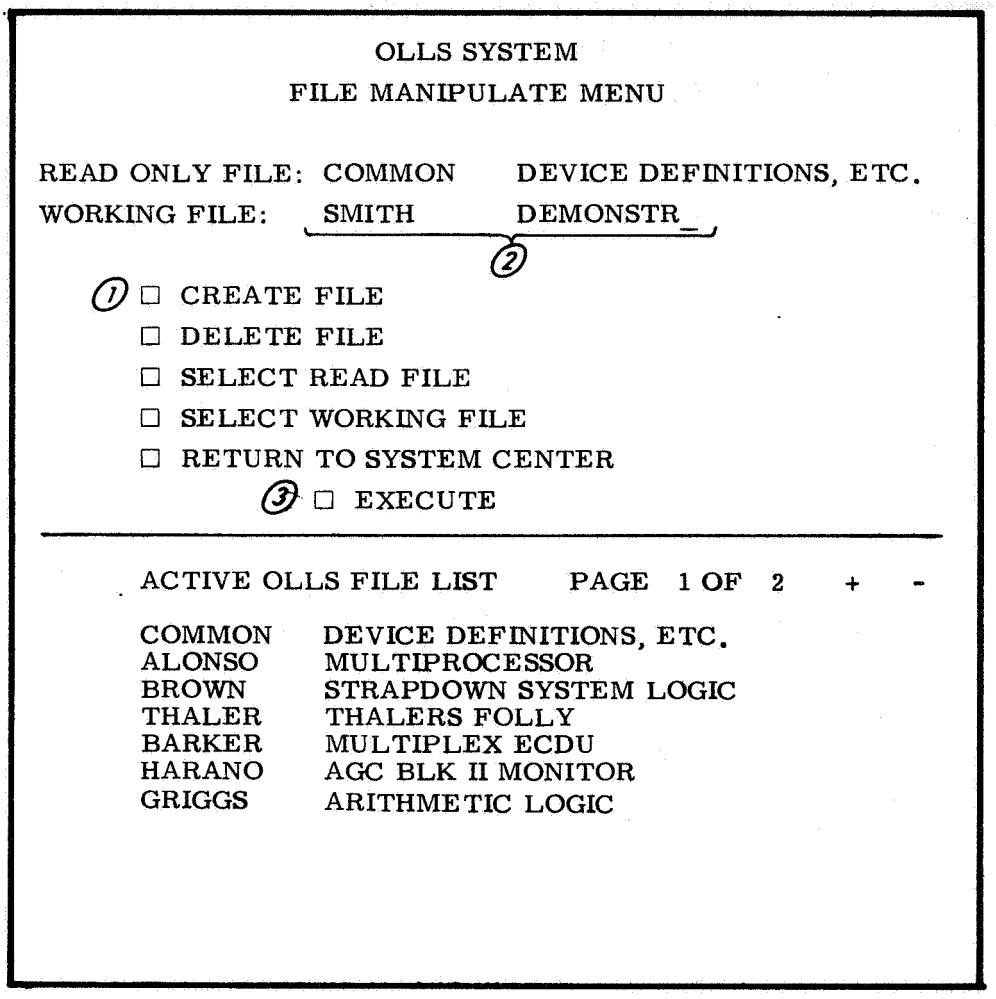

OLLS SYSTEM
FILE MANIPULATE MENU
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: SMITH DEMONSTR_
2
1 ☐ CREATE FILE
☐ DELETE FILE
☐ SELECT READ FILE
☐ SELECT WORKING FILE
☐ RETURN TO SYSTEM CENTER
3 ☐ EXECUTE
ACTIVE OLLS FILE LIST PAGE 1 OF 2 + -
COMMON DEVICE DEFINITIONS, ETC.
ALONSO MULTIPROCESSOR
BROWN STRAPDOWN SYSTEM LOGIC
THALER THALERS FOLLY
BARKER MULTIPLEX ECDU
HARANO AGC BLK II MONITOR
GRIGGS ARITHMETIC LOGIC

Fig. 3-20

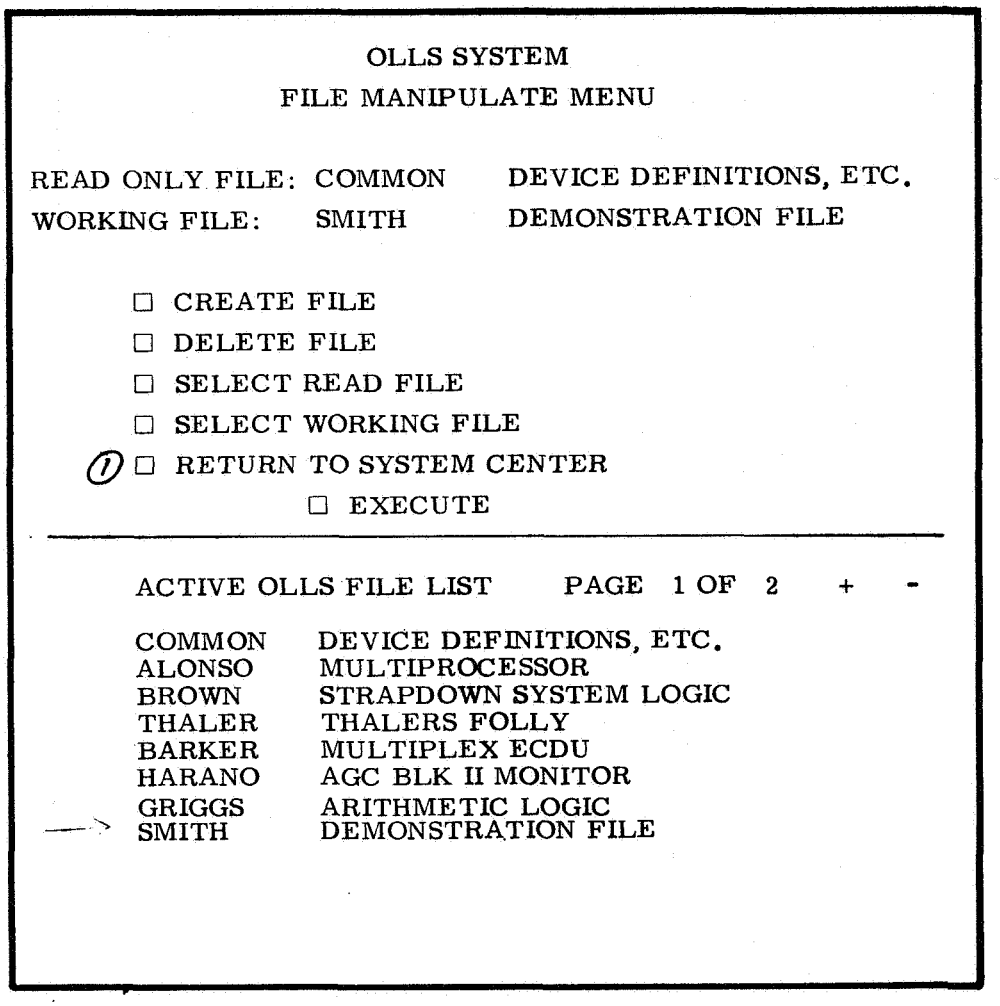

OLLS SYSTEM
FILE MANIPULATE MENU
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: SMITH DEMONSTRATION FILE
☐ CREATE FILE
☐ DELETE FILE
☐ SELECT READ FILE
☐ SELECT WORKING FILE
1 ☐ RETURN TO SYSTEM CENTER
☐ EXECUTE
ACTIVE OLLS FILE LIST PAGE 1 OF 2 + -
COMMON DEVICE DEFINITIONS, ETC.
ALONSO MULTIPROCESSOR
BROWN STRAPDOWN SYSTEM LOGIC
THALER THALERS FOLLY
BARKER MULTIPLEX ECDU
HARANO AGC BLK II MONITOR
GRIGGS ARITHMETIC LOGIC
SMITH DEMONSTRATION FILE

Fig. 3-21

When Fig. 3-21 appears, our Mr. Smith elects to return to SYSTEM CENTER. Smith next selects the DEVICE MANIPULATE mode (Fig. 3-22), whereupon Fig. 3-23 results. He now can elect to DISPLAY READ ONLY FILE DEVICE LIST, a command which refers to the READ ONLY FILE atop Fig. 3-23. At the bottom of the screen is the WORKING FILE DEVICE LIST, presently empty.

When he touches DISPLAY READ ONLY FILE DEVICE LIST (1), and then EXECUTE (2), that list appears below and the original command option is replaced by DISPLAY WORKING FILE DEVICE LIST. Smith can go back and forth alternately displaying each list.

Figure 3-24 shows how Smith selects devices for his working file. He first selects DEFINE DEVICE (1), followed by COPY FROM FILE (2). As will be seen later, the designer can define his own devices.

Following the designer's election to copy from file, he selects, always with the light pen, those devices he needs for his design. These are 3NOR/M (3), 8AND (4), BINARY 1 (5), and DIODE (6). He then EXECUTEs (7), following which he elects to display the working file he has just composed (8). The result is shown in Fig. 3-25. Notice that DISPLAY WORKING FILE LIST has been replaced by DISPLAY READ ONLY FILE LIST.

As of Fig. 3-25 Smith wishes to display a device, the 3NOR/M element, so he commands (1), (2) EXECUTE (3) in Fig. 3-25. The result is Fig. 3-26. The gate symbol, equations, terminal names, and other relevant data are displayed.

At this time the designer elects to RETURN TO DEVICE CENTRAL.

3.6.5 Defining a New Device

Being able to define new devices is, from the user's point of view, a most important property. It frees him from dealing with system programmers, for, if he cannot define his own devices and if the existing list is insufficient, he has to have someone else model the device. The facility to interact with device modeling allows the designer a great advantage. He can define devices operationally, as a black box, or as a circuit, and he can change an existing definition if he so desires. He can readily incorporate new logic elements as they are announced by commercial firms, or he can invent his own, and test them as a system component.

Starting from OLLS DEVICE CENTRAL, Smith can now choose to DEFINE DEVICE (1). Previously he also chose COPY FROM FILE, but this time he does not. He starts typing (2) J-K FLOP DIRECT SET AND RESET COMPONENT. The name proper is J-K FLOP, and the rest are comments. When he EXECUTEs (3), a null device by that name is incorporated into the file.

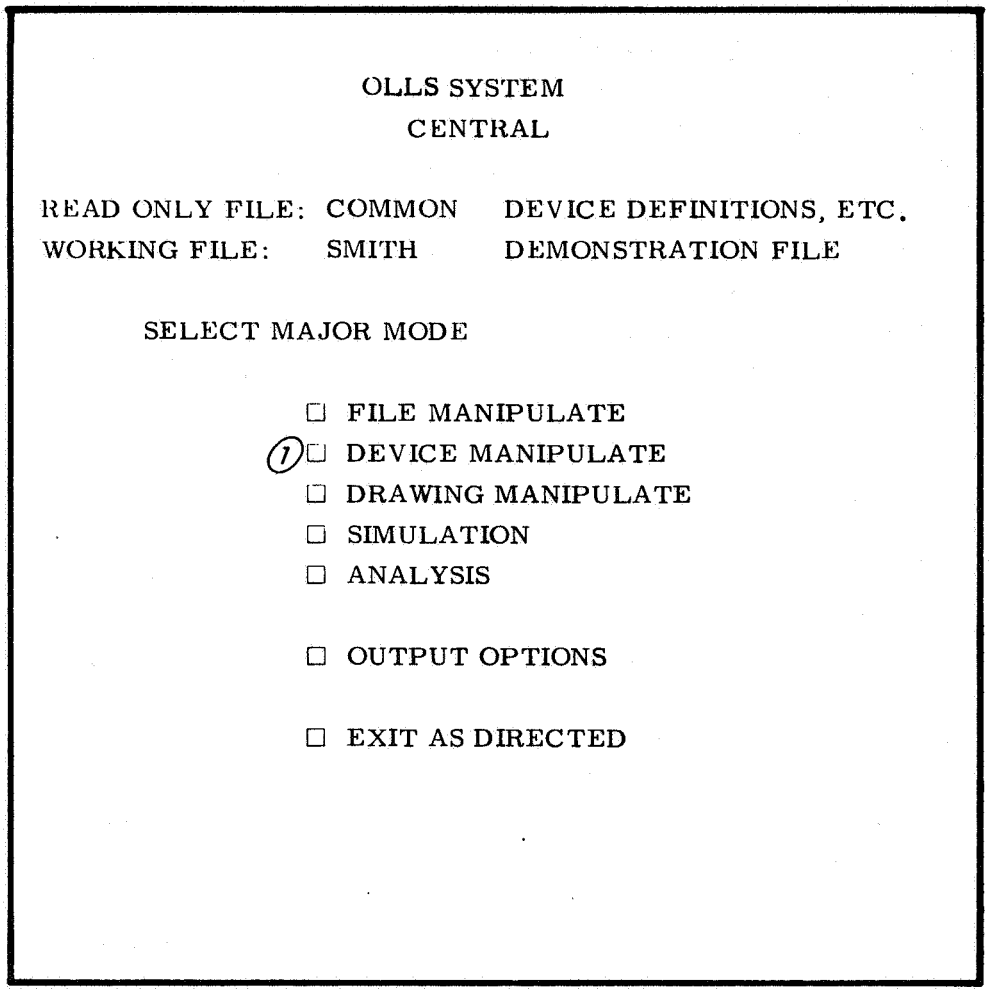

OLLS SYSTEM
CENTRAL
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: SMITH DEMONSTRATION FILE
SELECT MAJOR MODE
FILE MANIPULATE
1
DEVICE MANIPULATE
DRAWING MANIPULATE
SIMULATION
ANALYSIS
OUTPUT OPTIONS
EXIT AS DIRECTED

Fig. 3-22

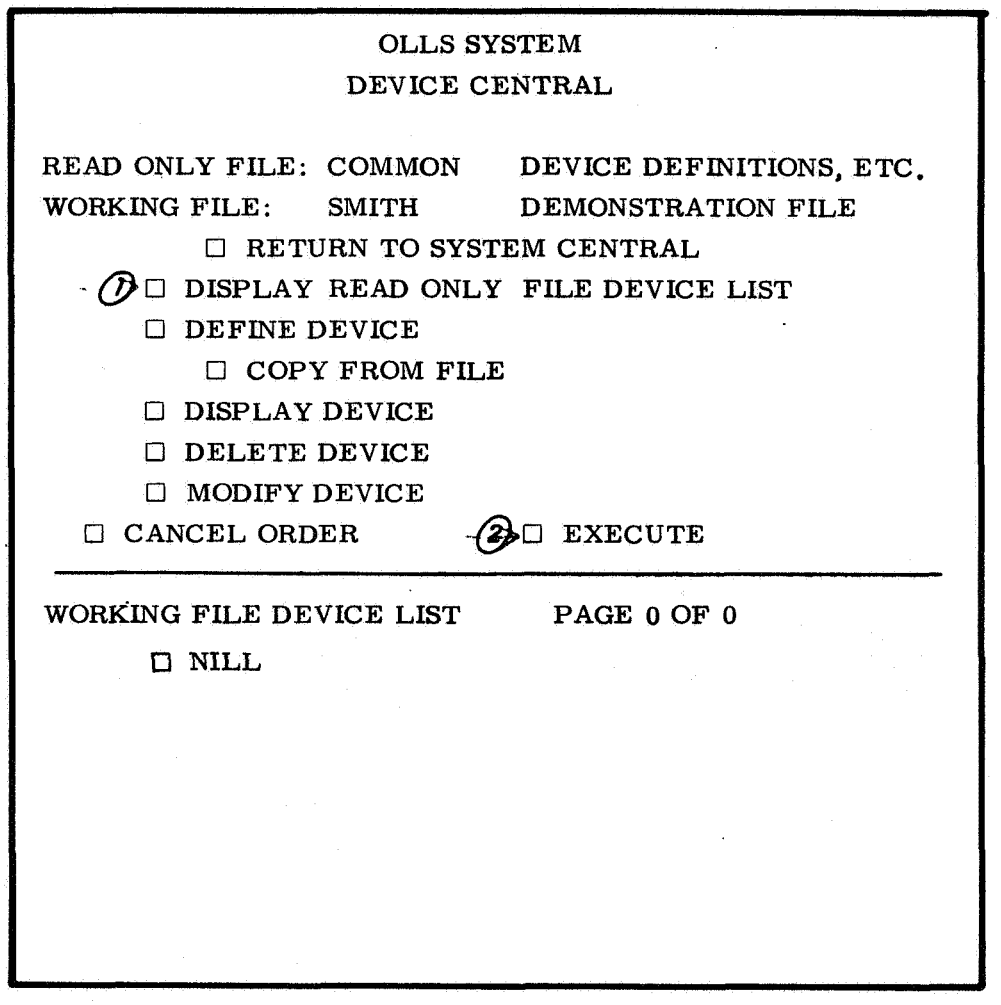

OLLS SYSTEM
DEVICE CENTRAL
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: SMITH DEMONSTRATION FILE
☐ RETURN TO SYSTEM CENTRAL
1 ☐ DISPLAY READ ONLY FILE DEVICE LIST
☐ DEFINE DEVICE
☐ COPY FROM FILE
☐ DISPLAY DEVICE
☐ DELETE DEVICE
☐ MODIFY DEVICE
☐ CANCEL ORDER
2 ☐ EXECUTE
WORKING FILE DEVICE LIST PAGE 0 OF 0
☐ NILL

Fig. 3-23

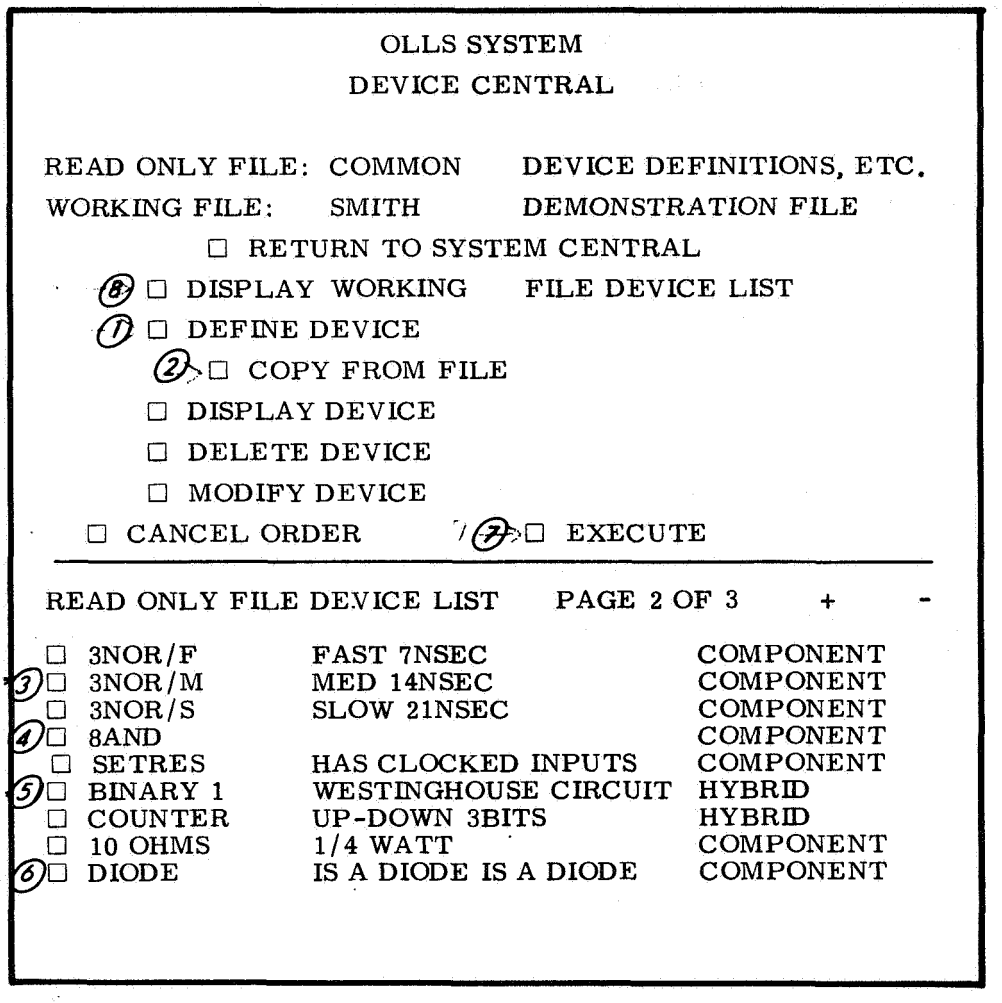

OLLS SYSTEM
DEVICE CENTRAL
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: SMITH DEMONSTRATION FILE
☐ RETURN TO SYSTEM CENTRAL
8 ☐ DISPLAY WORKING FILE DEVICE LIST
1 ☐ DEFINE DEVICE
2 ☐ COPY FROM FILE
☐ DISPLAY DEVICE
☐ DELETE DEVICE
☐ MODIFY DEVICE
☐ CANCEL ORDER 7 ☐ EXECUTE
READ ONLY FILE DEVICE LIST PAGE 2 OF 3 + -
☐ 3NOR/F FAST 7NSEC COMPONENT
3 ☐ 3NOR/M MED 14NSEC COMPONENT
☐ 3NOR/S SLOW 21NSEC COMPONENT
4 ☐ 8AND COMPONENT
☐ SETRES HAS CLOCKED INPUTS COMPONENT
5 ☐ BINARY 1 WESTINGHOUSE CIRCUIT HYBRID
☐ COUNTER UP-DOWN 3BITS HYBRID
☐ 10 OHMS 1/4 WATT COMPONENT
6 ☐ DIODE IS A DIODE IS A DIODE COMPONENT

Fig. 3-24

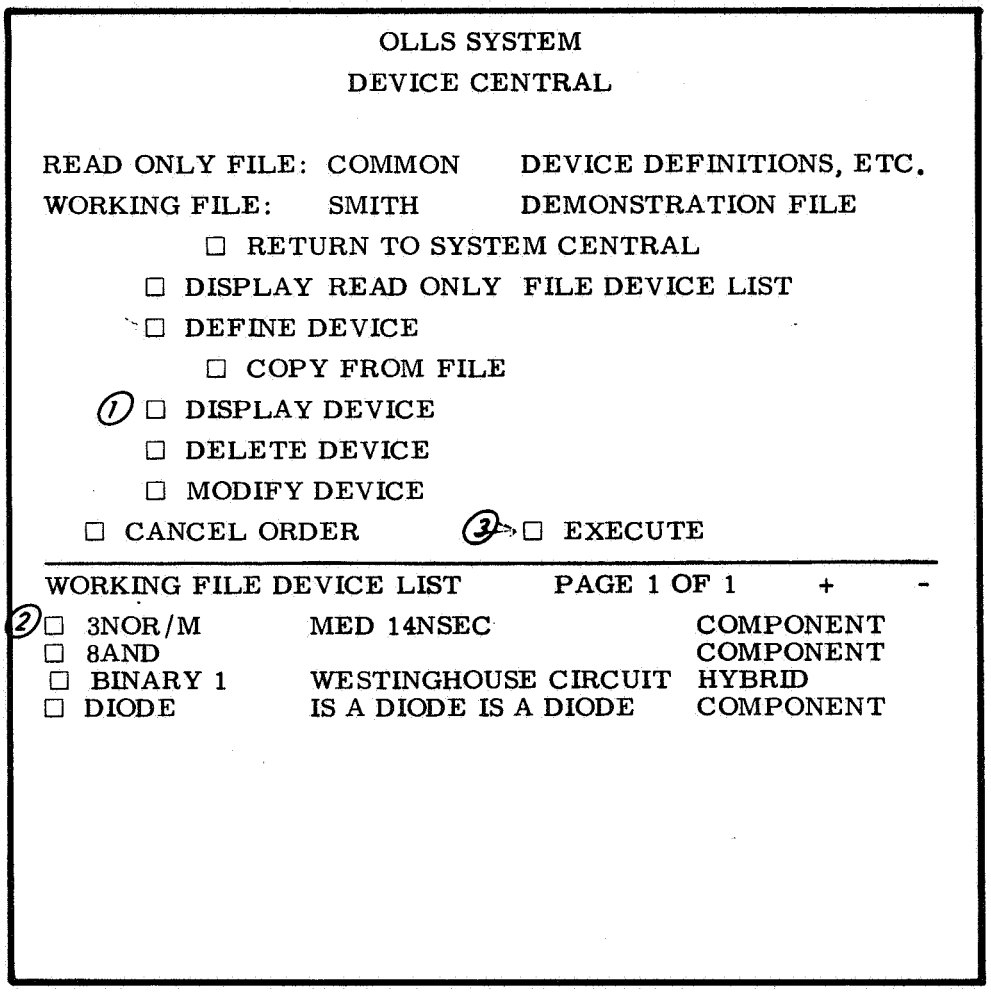
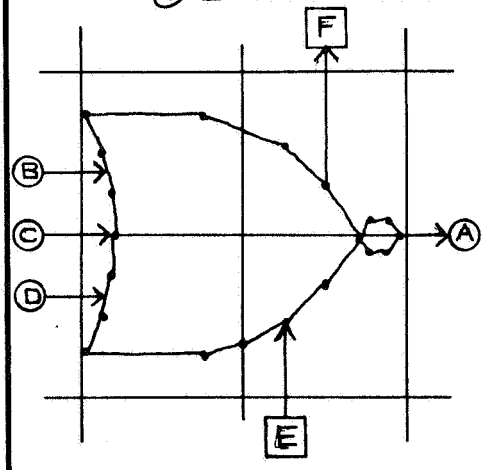

OLLS SYSTEM
DEVICE CENTRAL
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: SMITH DEMONSTRATION FILE
☐ RETURN TO SYSTEM CENTRAL
☐ DISPLAY READ ONLY FILE DEVICE LIST
☐ DEFINE DEVICE
☐ COPY FROM FILE
1 ☐ DISPLAY DEVICE
☐ DELETE DEVICE
☐ MODIFY DEVICE
☐ CANCEL ORDER
3 ☐ EXECUTE
WORKING FILE DEVICE LIST PAGE 1 OF 1 + -
2 ☐ 3NOR/M MED 14NSEC COMPONENT
☐ 8AND COMPONENT
☐ BINARY 1 WESTINGHOUSE CIRCUIT HYBRID
☐ DIODE IS A DIODE IS A DIODE COMPONENT

Fig. 3-25

OLLS SYSTEM
DISPLAY DEVICE

3NOR/M MED14NSEC

(1) □ RETURN TO DEVICE CENTRAL



SIZE	1
PLOTPOINTS	21
INPUTS	3
OUTPUTS	1
EXPANDER INS	1
EXPANDER OUTS	1
EQUATIONS	2

UNSPECIFIED TERMINALS
NONE

EQUATION LIST PAGE 1 OF 1 + -

□ A = E *(B +C +D)/
0 1 14 14 14

□ F = A
0 1

Fig. 3-26

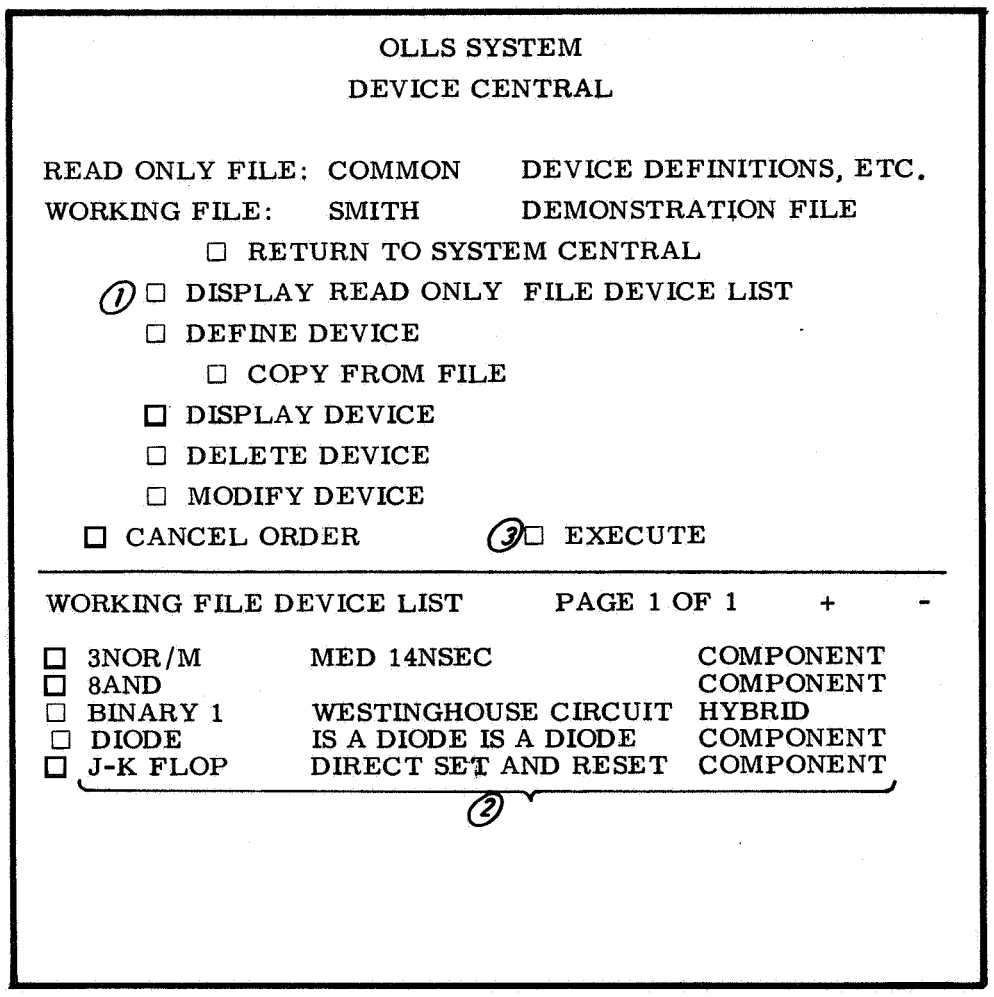

OLLS SYSTEM
DEVICE CENTRAL
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: SMITH DEMONSTRATION FILE
☐ RETURN TO SYSTEM CENTRAL
1 ☐ DISPLAY READ ONLY FILE DEVICE LIST
☐ DEFINE DEVICE
☐ COPY FROM FILE
☐ DISPLAY DEVICE
☐ DELETE DEVICE
☐ MODIFY DEVICE
☐ CANCEL ORDER
3 ☐ EXECUTE
WORKING FILE DEVICE LIST PAGE 1 OF 1 + -
☐ 3NOR/M MED 14NSEC COMPONENT
☐ 8AND COMPONENT
☐ BINARY 1 WESTINGHOUSE CIRCUIT HYBRID
☐ DIODE IS A DIODE IS A DIODE COMPONENT
☐ J-K FLOP DIRECT SET AND RESET COMPONENT
2

Fig. 3-27

The DEVICE CENTRAL display remains. The null device just entered can be changed to the desired one by commanding MODIFY DEVICE. The sequence (1), (2), EXECUTE (3) is shown in Fig. 3-28, whereupon Fig. 3-29, MODIFY DEVICE menu, appears.

The designer must specify a number of things at this point, the most important of which are its logical behavior and graphic symbol. Notice that the name of the device to be modified appears at the top of the MODIFY DEVICE menu.

The graphic symbol must be defined just so that named terminals exist for the equations to refer to.

3.6.6 MODIFY DEVICE

When the MODIFY DEVICE menu appears (Fig. 3-29), the designer can sketch an appropriate symbol. He first selects a size, (1) and (2), then, by alternately pointing to SEGMENT (or SMALL DOT) (3) and transferring the light pen to the sketch area. Segments start and stop where the light pen starts and stops. Small dots are actually small circles centered where the light pen first alights. The ERASE option deletes segments (or dots) on a last-in first-out basis.

The LINE OF SYMMETRY (4) option is used as follows: when half the symbol has been drawn, the light pen can be pointed to LINE OF SYMMETRY, or to two points on either a vertical or horizontal grid line, and to a point on the side to be reproduced. The other half of the symbol then appears.

When satisfied, the designers can ACCEPT SKETCH (5), which is also a PROCEED command.

The MODIFY DEVICE menu changes SKETCH OPTION to PLOTPOINTS OPTION (Fig. 3-30). The original sketch is composed of great many points, which takes a lot of 2250 memory to display. The PLOTPOINTS option allows replacement by a cruder picture.

First, the center of the device symbol must be identified. The location of a device symbol in a drawing refers to the device center.

The PENDOWN option permits replacement of sections of the sketch by short straight segments, to speed up device symbol display. The designer, after touching PENDOWN, touches consecutively the ends of the approximating segments. When satisfied, the designer can ACCEPT PLOTPOINTS which is also a PROCEED command. The end result, when accepted, is as shown in Fig. 3-31.

The graphic symbol now needs terminals assigned to it. Mr. Smith touches ADD TERMINAL (S) (1), then INPUT (2), indicating that, until further notice, all terminals are to be input terminals. He places the five input terminals

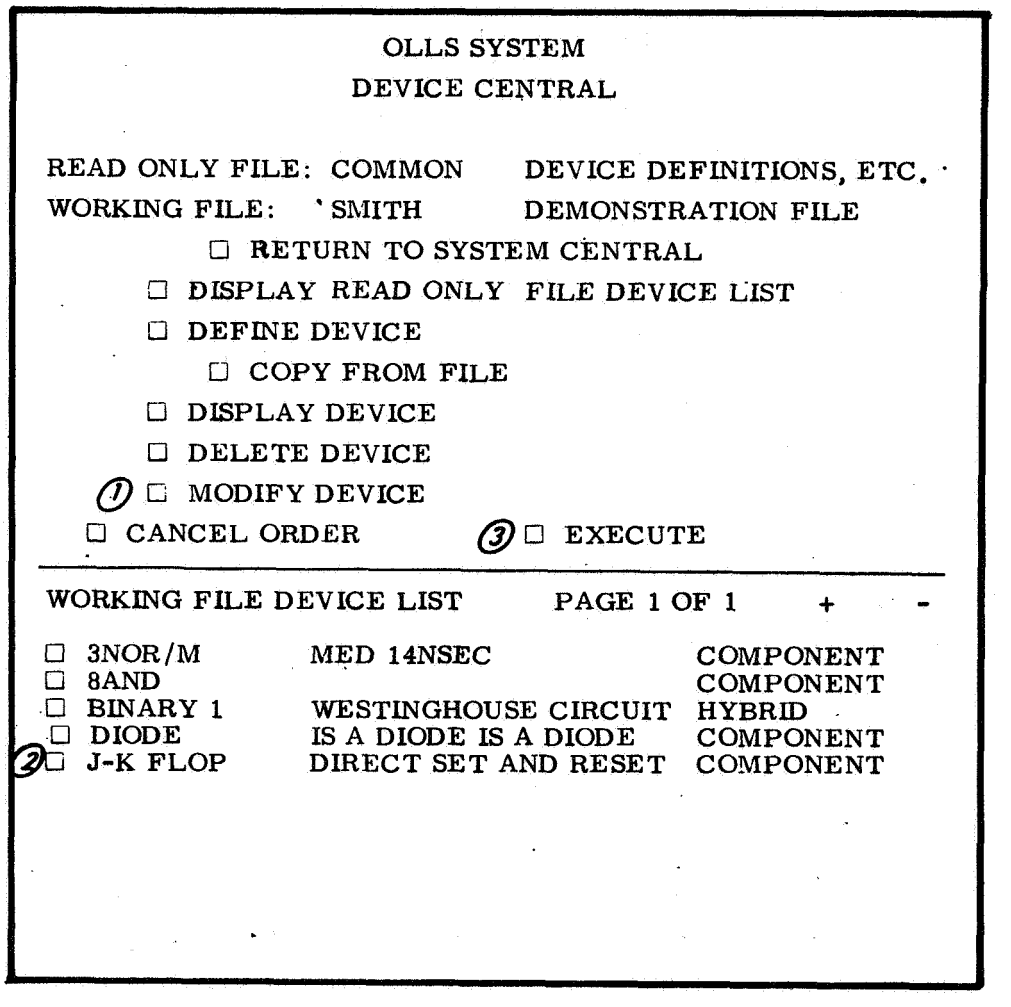

OLLS SYSTEM
DEVICE CENTRAL
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: 'SMITH DEMONSTRATION FILE
☐ RETURN TO SYSTEM CENTRAL
☐ DISPLAY READ ONLY FILE DEVICE LIST
☐ DEFINE DEVICE
☐ COPY FROM FILE
☐ DISPLAY DEVICE
☐ DELETE DEVICE
1 ☐ MODIFY DEVICE
☐ CANCEL ORDER
3 ☐ EXECUTE
WORKING FILE DEVICE LIST PAGE 1 OF 1 + -
☐ 3NOR/M MED 14NSEC COMPONENT
☐ 8AND COMPONENT
☐ BINARY 1 WESTINGHOUSE CIRCUIT HYBRID
☐ DIODE IS A DIODE IS A DIODE COMPONENT
2 ☐ J-K FLOP DIRECT SET AND RESET COMPONENT

Fig. 3-28

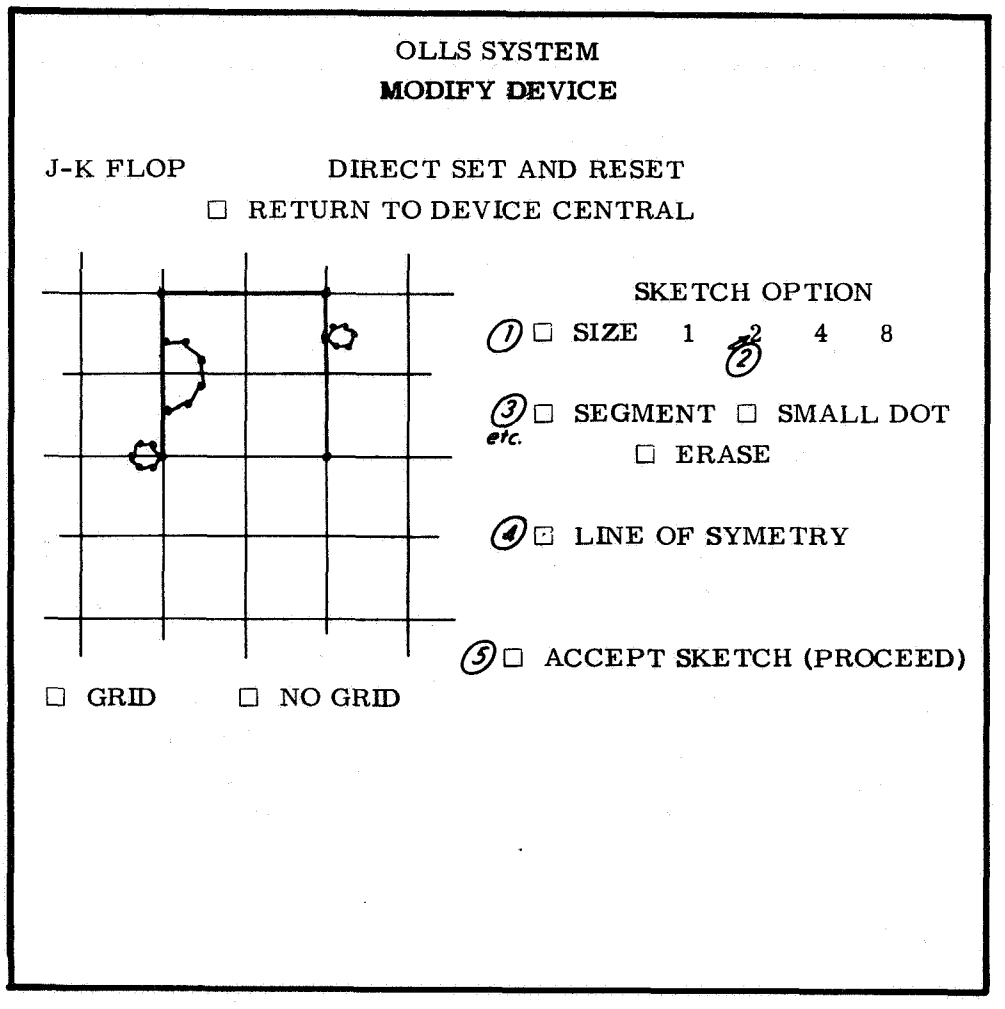

OLLS SYSTEM
MODIFY DEVICE
J-K FLOP
DIRECT SET AND RESET
☐ RETURN TO DEVICE CENTRAL
SKETCH OPTION
(1) ☐ SIZE 1 2 4 8
(2)
(3) ☐ SEGMENT ☐ SMALL DOT
etc.
☐ ERASE
(4) ☐ LINE OF SYMETRY
(5) ☐ ACCEPT SKETCH (PROCEED)
☐ GRID
☐ NO GRID

Fig. 3-29

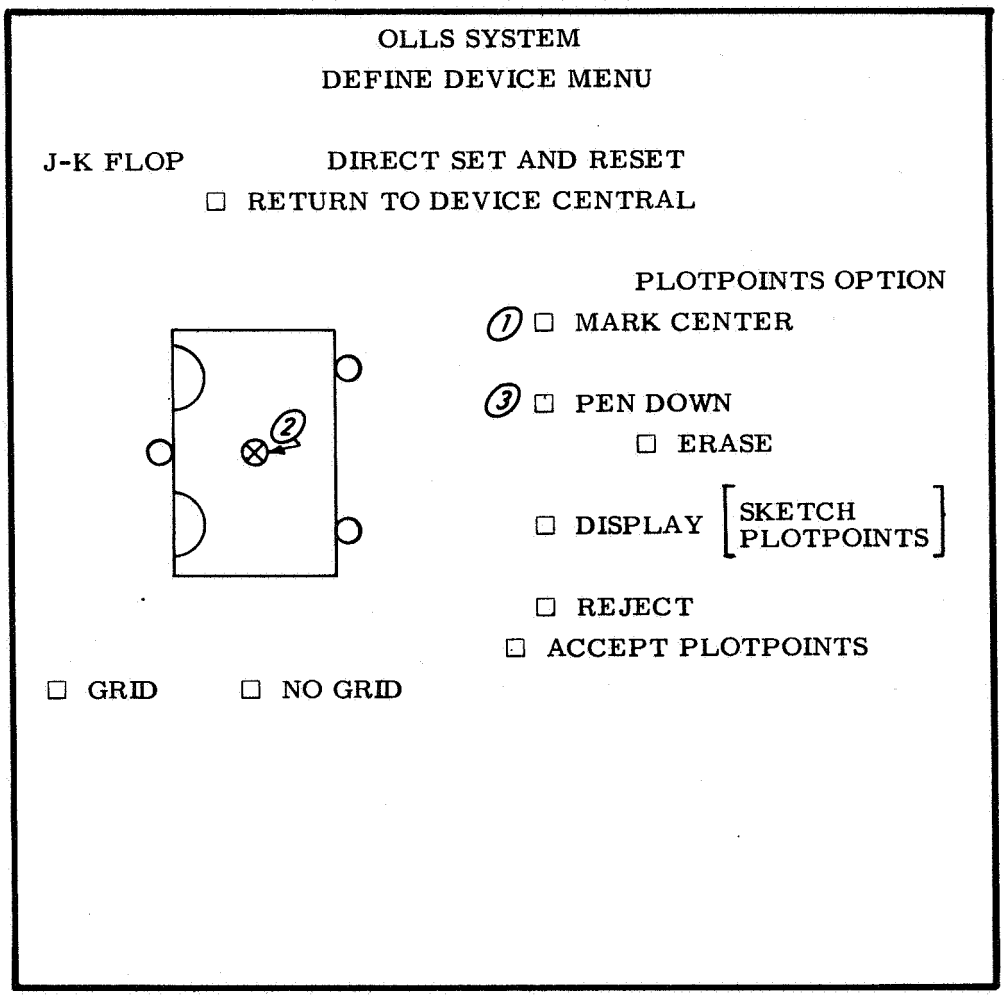

OLLS SYSTEM
DEFINE DEVICE MENU
J-K FLOP
DIRECT SET AND RESET
☐ RETURN TO DEVICE CENTRAL
PLOTPOINTS OPTION
1 ☐ MARK CENTER
3 ☐ PEN DOWN
☐ ERASE
☐ DISPLAY [SKETCH PLOTPOINTS]
☐ REJECT
☐ ACCEPT PLOTPOINTS
2
☐ GRID
☐ NO GRID

Fig. 3-30

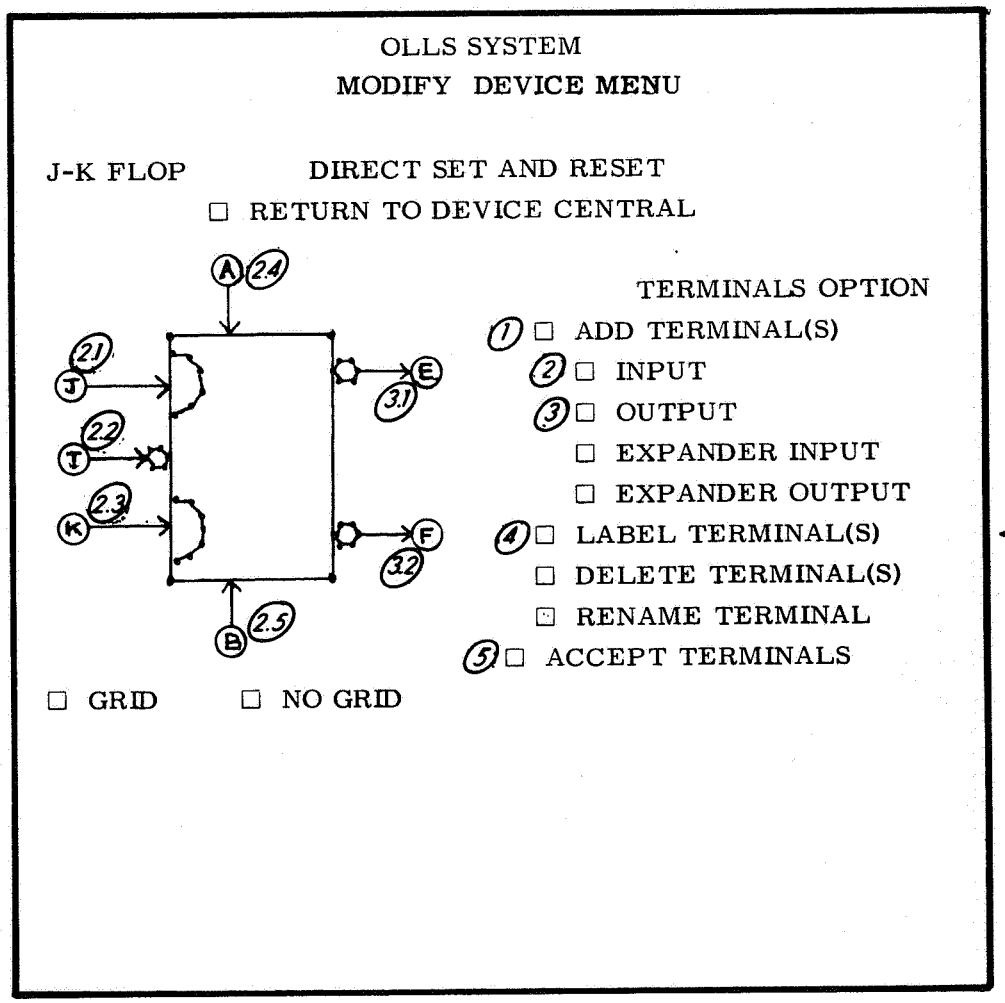

OLLS SYSTEM
MODIFY DEVICE MENU
J-K FLOP
DIRECT SET AND RESET
☐ RETURN TO DEVICE CENTRAL
A 2.4
J 2.1
T 2.2
K 2.3
E 3.1
F 3.2
B 2.5
TERMINALS OPTION
1 ☐ ADD TERMINAL(S)
2 ☐ INPUT
3 ☐ OUTPUT
☐ EXPANDER INPUT
☐ EXPANDER OUTPUT
4 ☐ LABEL TERMINAL(S)
☐ DELETE TERMINAL(S)
☐ RENAME TERMINAL
5 ☐ ACCEPT TERMINALS
☐ GRID
☐ NO GRID
Note:
TYPE TERMINAL NAME
(CURSOR APPEARS IN
ORDER OF CREATION
OF TERMINALS)

Fig. 3-31

(actions (2.1) to (2.5)) where he wants them, by touching some place on the periphery of the symbol. Then he touches OUTPUT (3), then points (3.1) and (3.2). All terminals are still unlabeled. Smith touches LABEL TERMINAL (S) (4), touches the terminal he wants to name and types a letter or a number. When he is satisfied he touches ACCEPT TERMINALS (5), whereupon the MODIFY DEVICE menu changes to the one shown in Fig. 3-32.

3.6.7 Functional Definition

The graphic symbol having been defined and labeled, there now remains the task of defining the device functionally. The algebraic method used is described in detail in Appendix B. In particular, there is a section on the description of a JK flip-flop which applies here.

Smith has presumably worked out a model for his JK (a far from final task). He touches INSERT EQUATION (1) and then types in the first equation (2), but without any delay subscripts. He then touches INSERT DELAYS (3) and using the space bar, types the delay values under and to the right of the corresponding variables (4). The equations can be altered to suit, or modified at a later time. The JK equations used here make no use of the set and reset terminals (A and B). These are consequently ignored in the simulation of the device. When a satisfactory set of equations has been written, the designer can ACCEPT EQUATIONS (5). In our present example he does so and then touches RETURN TO DEVICE CENTRAL (6). Notice that this last option has been continuously available throughout the device modification procedures.

3.6.8 DRAWINGS

Our man Smith returns to System Central by way of Device Central (Fig. 3-33 and 3-34). He is ready to use the available material to design a circuit, which is done by calling the DRAWING CENTRAL Fig. 3-35. The procedures here are quite analogous to the ones for DEFINE DEVICE. A list of currently available drawings (the READ ONLY FILE DRAWING LIST) is displayed (Fig. 3-36), and drawings for the circuits RING 5 and REG 16 are copied into the WORKING FILE. Smith is going to invent an arithmetic unit which he labels ARUN01 (Fig. 3-37). As in the case of defining the JK device, he caused his WORKING FILE DRAWING LIST to be displayed (1) + (2) (it contained, at that time, just RING 5 and REG 16). He then pointed to CREATE DRAWING (3), and typed in ARUN01 ARITHMETIC unit (4), followed by EXECUTE (5). With this action a null (blank) drawing so named is added to his working file. Smith must now elect MODIFY DRAWING (6), and then point to ARUN01 (7), and EXECUTE (8), to start the real design.

The DRAWING MODIFY 1 menu appears (Fig. 3-38), and Smith first

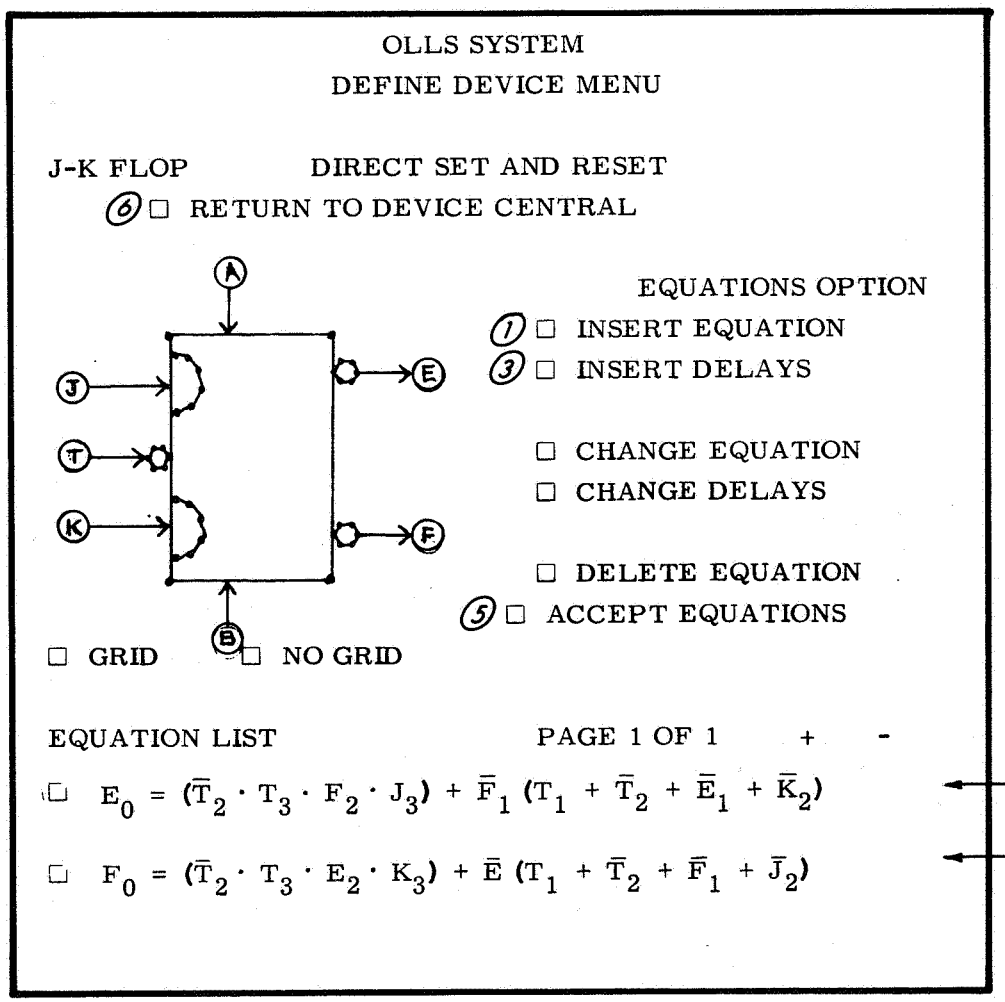

OLLS SYSTEM
DEFINE DEVICE MENU
J-K FLOP
DIRECT SET AND RESET
6 □ RETURN TO DEVICE CENTRAL
A
J
T
K
E
F
B
EQUATIONS OPTION
1 □ INSERT EQUATION
3 □ INSERT DELAYS
□ CHANGE EQUATION
□ CHANGE DELAYS
□ DELETE EQUATION
5 □ ACCEPT EQUATIONS
□ GRID
□ NO GRID
EQUATION LIST
PAGE 1 OF 1 + -
□ $E_0 = (\bar{T}_2 \cdot T_3 \cdot F_2 \cdot J_3) + \bar{F}_1 (T_1 + \bar{T}_2 + \bar{E}_1 + \bar{K}_2)$
TYPE 2
□ $F_0 = (\bar{T}_2 \cdot T_3 \cdot E_2 \cdot K_3) + \bar{E} (T_1 + \bar{T}_2 + \bar{F}_1 + \bar{J}_2)$
TYPE 4

Fig. 3-32

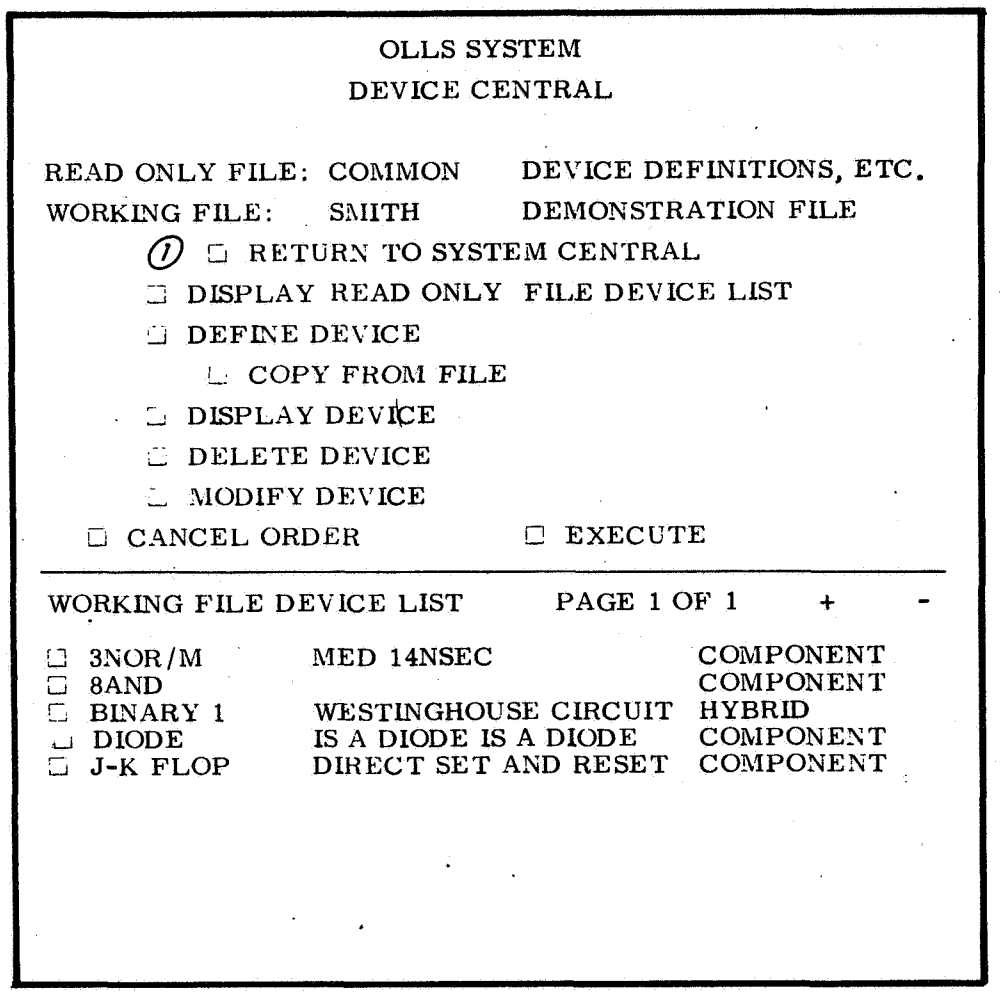

OLLS SYSTEM
DEVICE CENTRAL
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: SMITH DEMONSTRATION FILE
1 RETURN TO SYSTEM CENTRAL
DISPLAY READ ONLY FILE DEVICE LIST
DEFINE DEVICE
COPY FROM FILE
DISPLAY DEVICE
DELETE DEVICE
MODIFY DEVICE
CANCEL ORDER
EXECUTE
WORKING FILE DEVICE LIST PAGE 1 OF 1 + -
3NOR/M MED 14NSEC COMPONENT
8AND COMPONENT
BINARY 1 WESTINGHOUSE CIRCUIT HYBRID
DIODE IS A DIODE IS A DIODE COMPONENT
J-K FLOP DIRECT SET AND RESET COMPONENT

Fig. 3-33

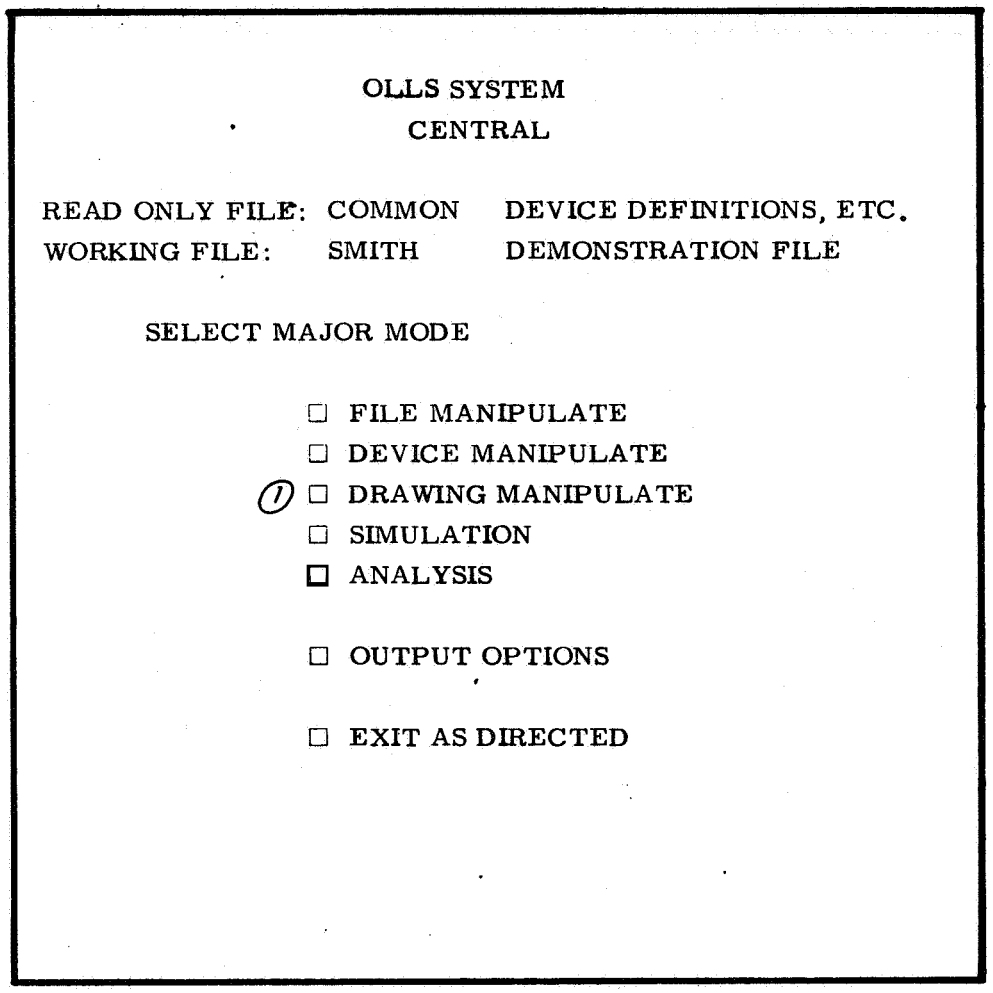

OLLS SYSTEM
CENTRAL
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: SMITH DEMONSTRATION FILE
SELECT MAJOR MODE
FILE MANIPULATE
DEVICE MANIPULATE
1
DRAWING MANIPULATE
SIMULATION
ANALYSIS
OUTPUT OPTIONS
EXIT AS DIRECTED

Fig. 3-34

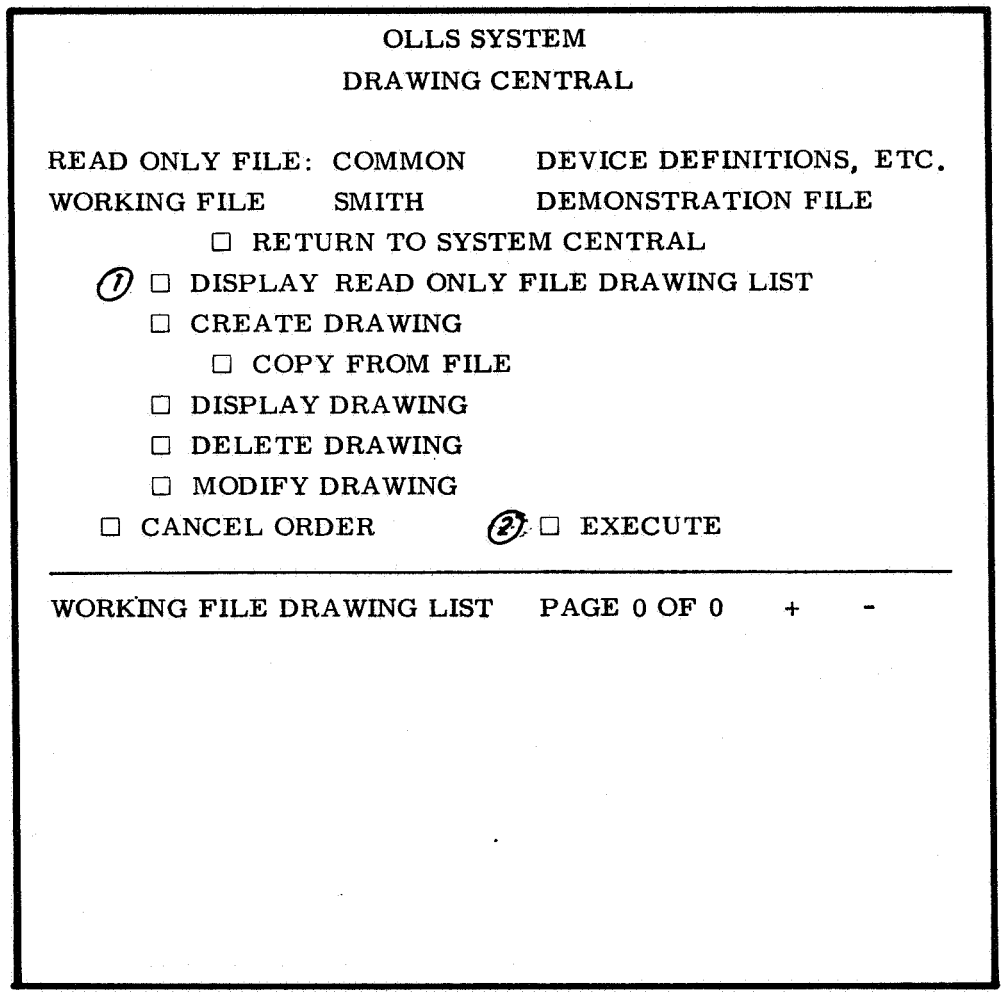

OLLS SYSTEM
DRAWING CENTRAL
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE SMITH DEMONSTRATION FILE
☐ RETURN TO SYSTEM CENTRAL
1 ☐ DISPLAY READ ONLY FILE DRAWING LIST
☐ CREATE DRAWING
☐ COPY FROM FILE
☐ DISPLAY DRAWING
☐ DELETE DRAWING
☐ MODIFY DRAWING
☐ CANCEL ORDER
2 ☐ EXECUTE
WORKING FILE DRAWING LIST PAGE 0 OF 0 + -

Fig. 3-35

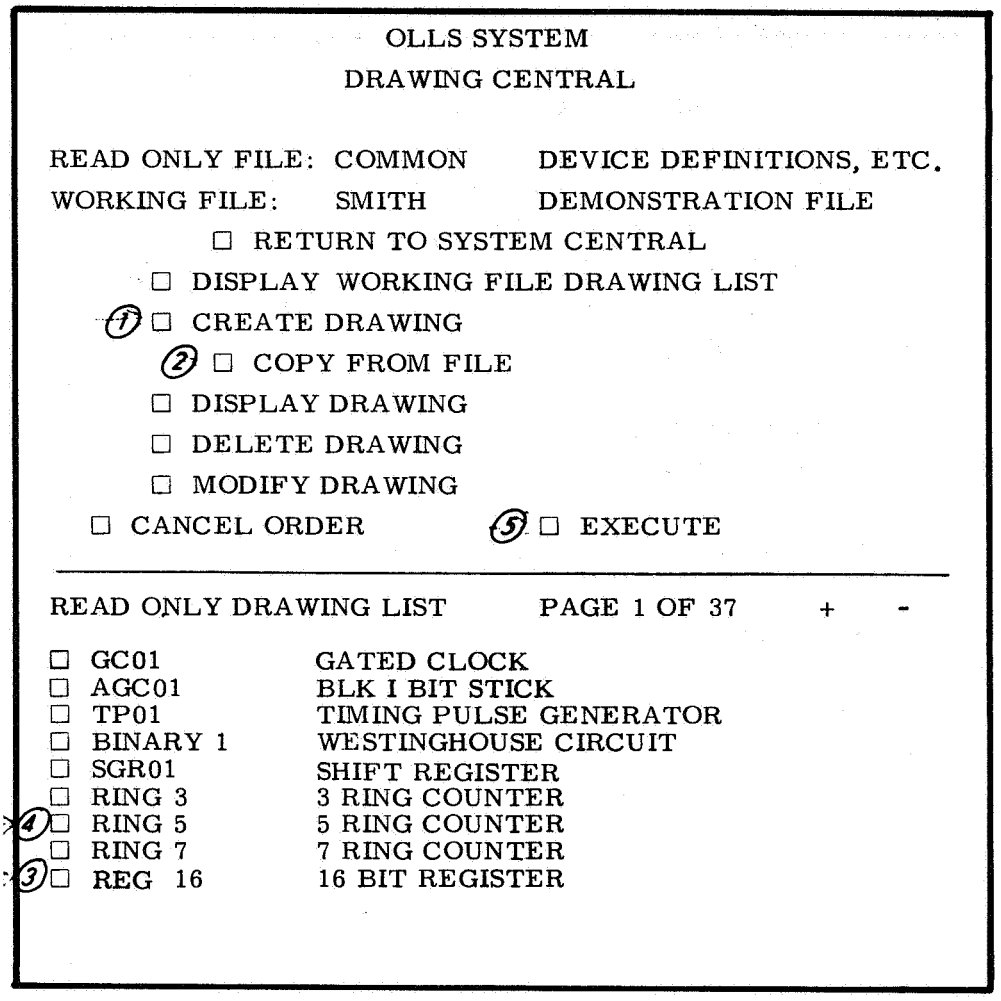

OLLS SYSTEM
DRAWING CENTRAL
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: SMITH DEMONSTRATION FILE
☐ RETURN TO SYSTEM CENTRAL
☐ DISPLAY WORKING FILE DRAWING LIST
1 ☐ CREATE DRAWING
2 ☐ COPY FROM FILE
☐ DISPLAY DRAWING
☐ DELETE DRAWING
☐ MODIFY DRAWING
☐ CANCEL ORDER
5 ☐ EXECUTE
READ ONLY DRAWING LIST PAGE 1 OF 37 + -
☐ GC01 GATED CLOCK
☐ AGC01 BLK I BIT STICK
☐ TP01 TIMING PULSE GENERATOR
☐ BINARY 1 WESTINGHOUSE CIRCUIT
☐ SGR01 SHIFT REGISTER
☐ RING 3 3 RING COUNTER
4 ☐ RING 5 5 RING COUNTER
☐ RING 7 7 RING COUNTER
3 ☐ REG 16 16 BIT REGISTER

Fig. 3-36

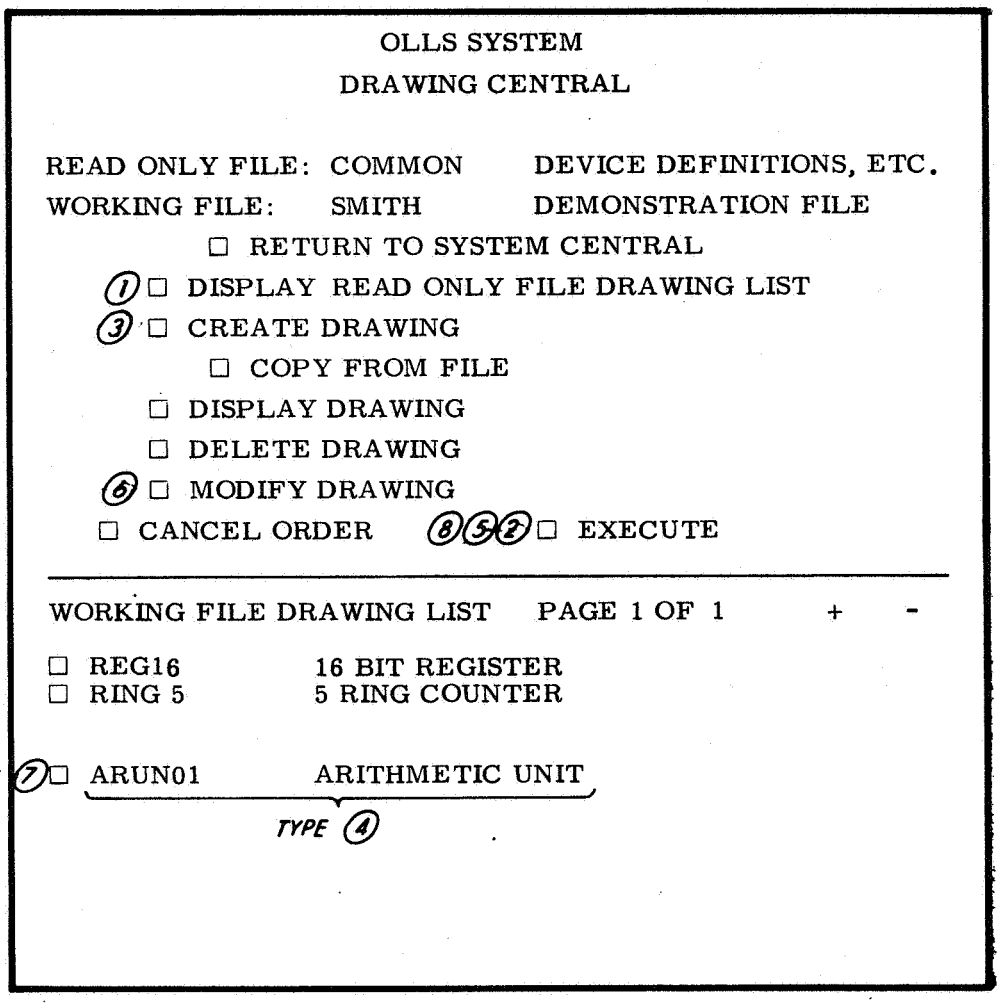

OLLS SYSTEM
DRAWING CENTRAL
READ ONLY FILE: COMMON DEVICE DEFINITIONS, ETC.
WORKING FILE: SMITH DEMONSTRATION FILE
☐ RETURN TO SYSTEM CENTRAL
1 ☐ DISPLAY READ ONLY FILE DRAWING LIST
3 ☐ CREATE DRAWING
☐ COPY FROM FILE
☐ DISPLAY DRAWING
☐ DELETE DRAWING
6 ☐ MODIFY DRAWING
☐ CANCEL ORDER 8 5 2 ☐ EXECUTE
WORKING FILE DRAWING LIST PAGE 1 OF 1 + -
☐ REG16 16 BIT REGISTER
☐ RING 5 5 RING COUNTER
7 ☐ ARUN01 ARITHMETIC UNIT
TYPE 4

Fig. 3-37

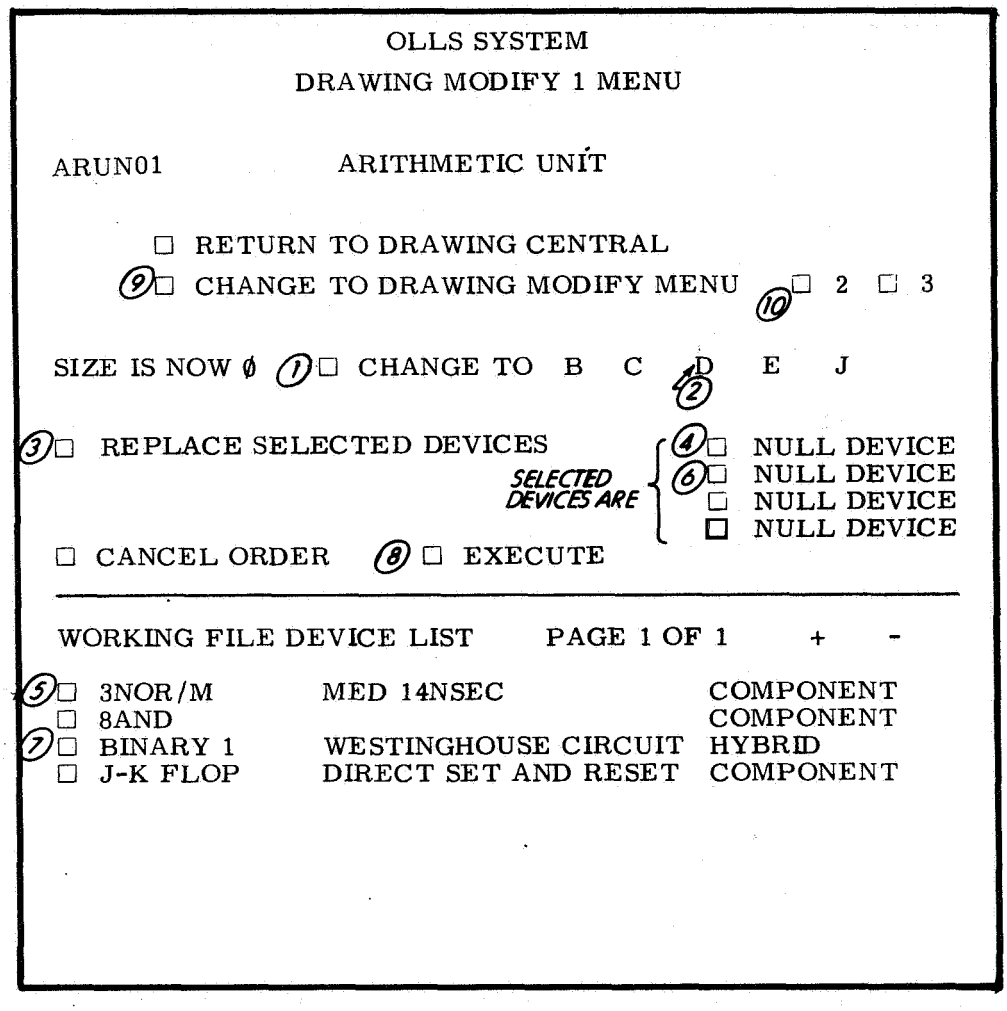

OLLS SYSTEM
DRAWING MODIFY 1 MENU
ARUN01 ARITHMETIC UNIT
RETURN TO DRAWING CENTRAL
CHANGE TO DRAWING MODIFY MENU
2
3
SIZE IS NOW ∅ CHANGE TO B C D E J
REPLACE SELECTED DEVICES
SELECTED DEVICES ARE
NULL DEVICE
NULL DEVICE
NULL DEVICE
NULL DEVICE
CANCEL ORDER
EXECUTE
WORKING FILE DEVICE LIST PAGE 1 OF 1 + -
3NOR/M MED 14NSEC COMPONENT
8AND COMPONENT
BINARY 1 WESTINGHOUSE CIRCUIT HYBRID
J-K FLOP DIRECT SET AND RESET COMPONENT

Fig. 3-38

picks a size by pointing to CHANGE TO (1) and then to D (2), meaning he wants a size D drawing.

The DRAWING MODIFY 1 menu displays the working list of devices previously selected. The task is to select devices and place them in the drawing. The MODIFY 1 menu shows four places where the choice of devices will be displayed; these places initially say NULL DEVICES. Action (3) points to REPLACE SELECTED DEVICES; (4) and (5) indicate that the first null device is to be replaced by 3NOR/M, and (6) and (7) show that the second null device is to be replaced by BINARY 1. EXECUTE follows (8), after which (Fig. 3-39) DRAWING MODIFY (2) menu is selected. That menu contains a blank page with a title block in the lower half, and the menu itself in the upper half. Menu 2 allows choice between a DEVICE SUBMENU (Fig. 3-41) and a SIGNAL SUBMENU (Fig. 3-42), which allow insertion and deletion of devices and connections.

At this point we will leave the present detailed description of OLLS-designer interaction, and point to a likely end result of his efforts (Fig. 3-46). (That figure is actually a circuit drawn with OLLS/1800). We have not discussed many of the necessary actions, such as placing elements or connecting elements automatically or along a designer-selected path. We have also not discussed, nor shall we, what sort of procedures are desirable for CRT simulation. The material presented up to here, however, should give a clear idea of the sort of interaction possible with OLLS/360.

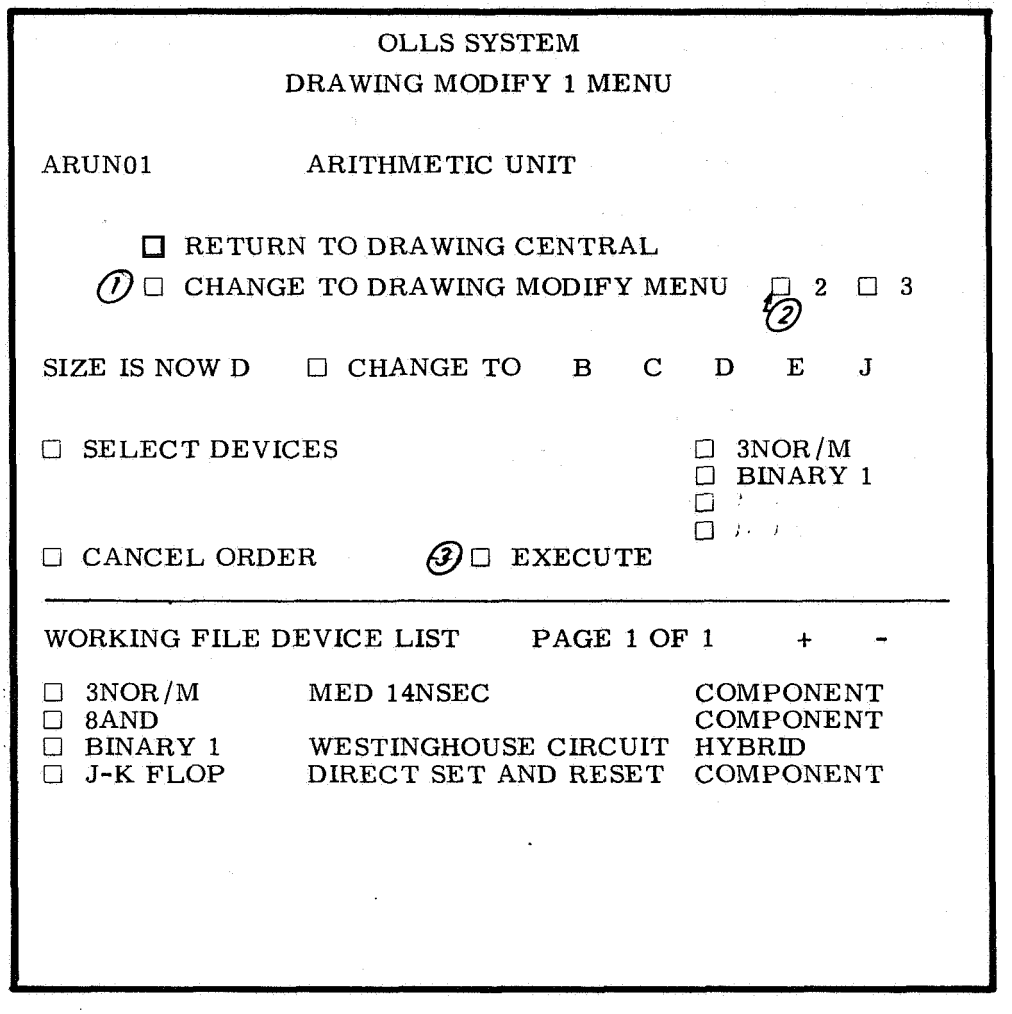

OLLS SYSTEM
DRAWING MODIFY 1 MENU
ARUN01 ARITHMETIC UNIT
RETURN TO DRAWING CENTRAL
1 CHANGE TO DRAWING MODIFY MENU 2 3
SIZE IS NOW D CHANGE TO B C D E J
SELECT DEVICES
3NOR/M
BINARY 1
CANCEL ORDER
3 EXECUTE
WORKING FILE DEVICE LIST PAGE 1 OF 1 + -
3NOR/M MED 14NSEC COMPONENT
8AND COMPONENT
BINARY 1 WESTINGHOUSE CIRCUIT HYBRID
J-K FLOP DIRECT SET AND RESET COMPONENT

Fig. 3-39

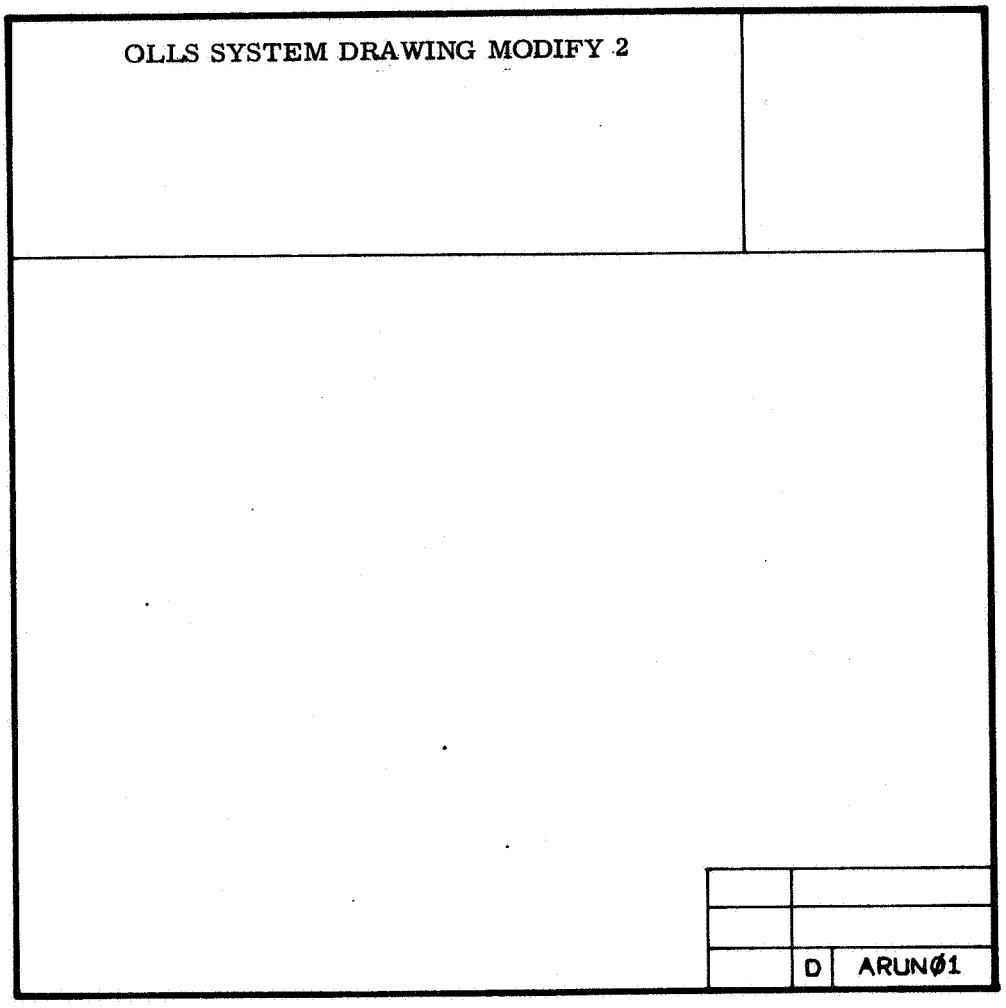


Fig. 3-40

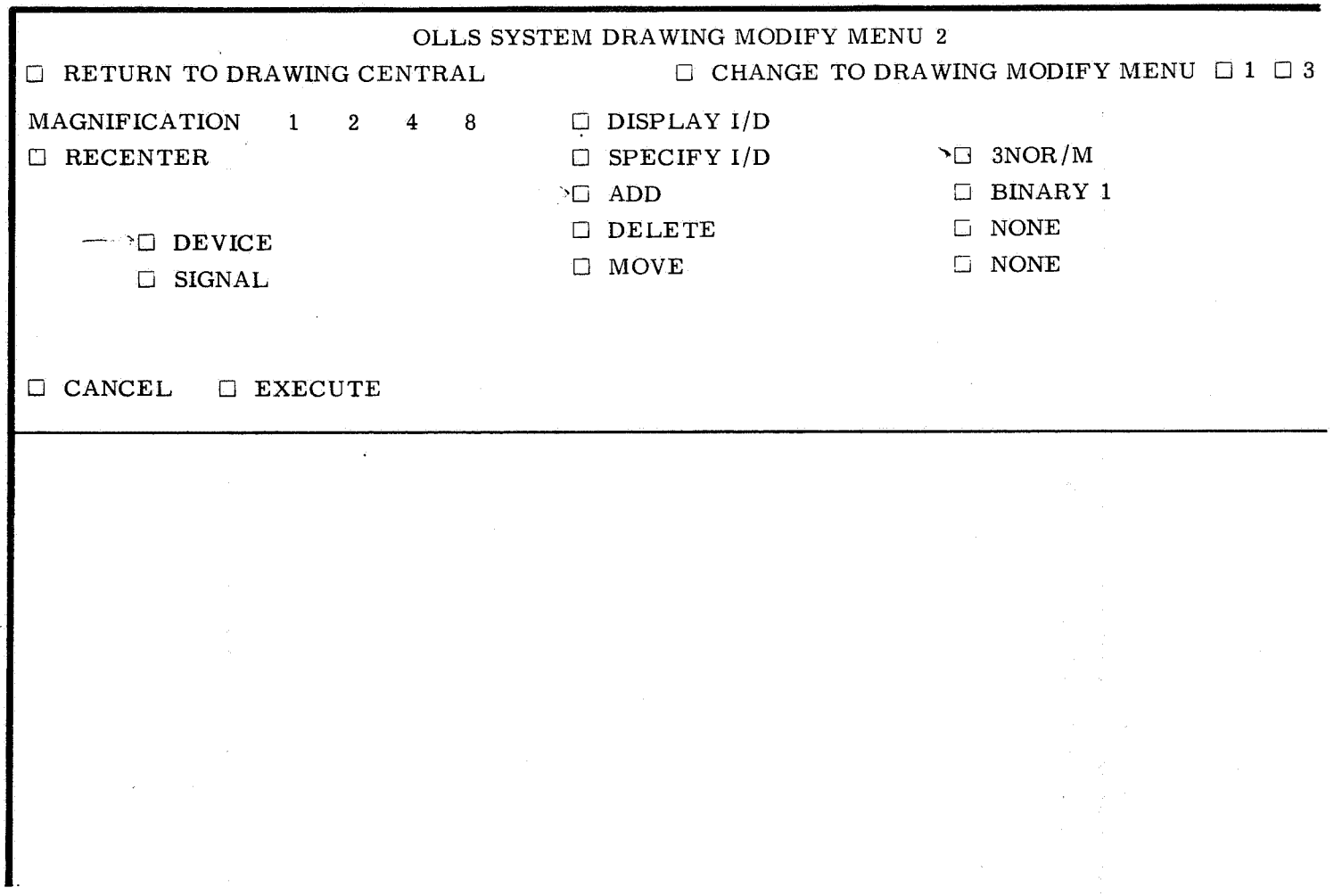

OLLS SYSTEM DRAWING MODIFY MENU 2
☐ RETURN TO DRAWING CENTRAL
☐ CHANGE TO DRAWING MODIFY MENU ☐ 1 ☐ 3
MAGNIFICATION 1 2 4 8
☐ RECENTER
☐ DEVICE
☐ SIGNAL
☐ DISPLAY I/D
☐ SPECIFY I/D
☐ ADD
☐ DELETE
☐ MOVE
☐ 3NOR/M
☐ BINARY 1
☐ NONE
☐ NONE
☐ CANCEL ☐ EXECUTE

Fig. 3-41

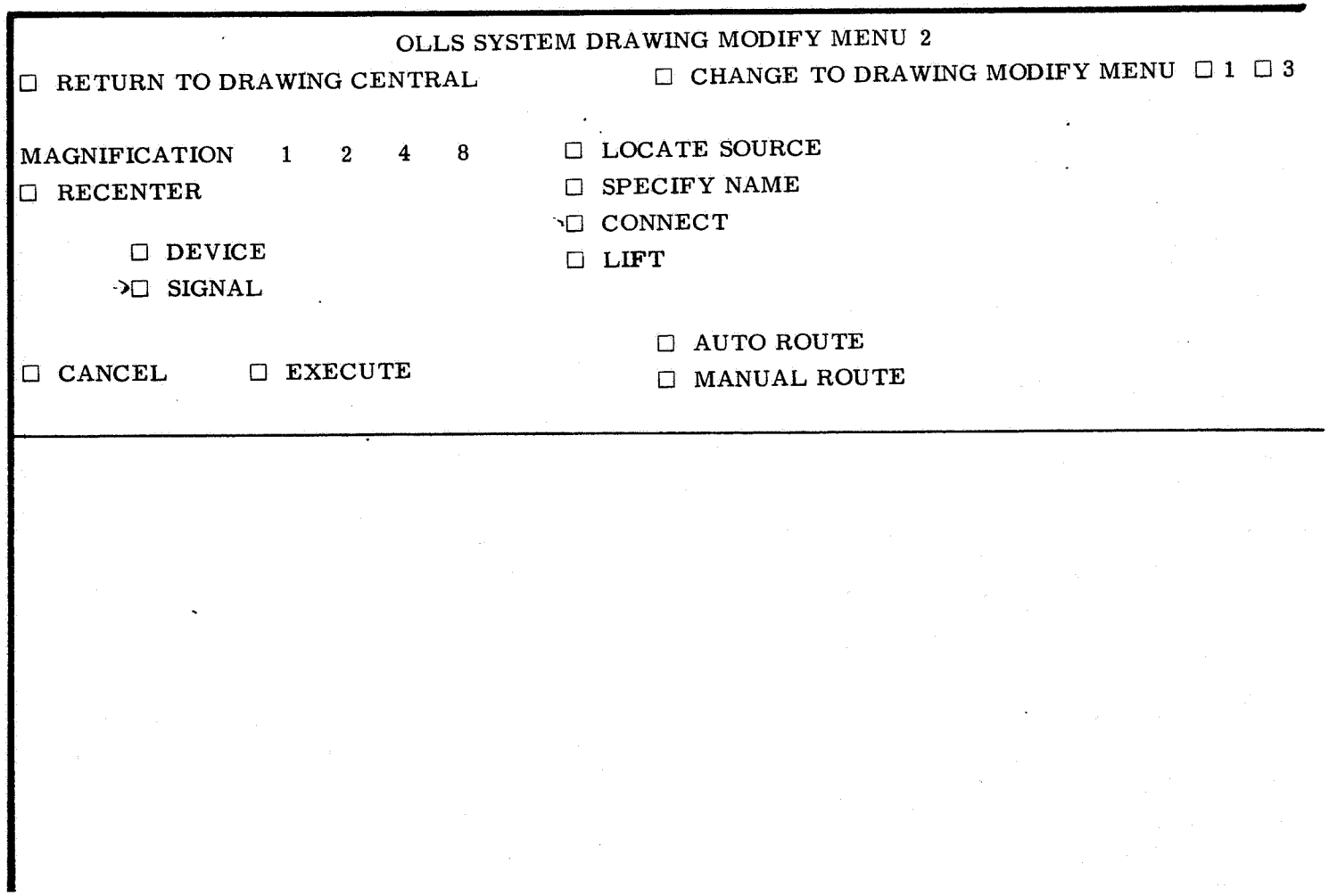

OLLS SYSTEM DRAWING MODIFY MENU 2
☐ RETURN TO DRAWING CENTRAL
☐ CHANGE TO DRAWING MODIFY MENU ☐ 1 ☐ 3
MAGNIFICATION 1 2 4 8
☐ RECENTER
☐ DEVICE
☐ SIGNAL
☐ CANCEL
☐ EXECUTE
☐ LOCATE SOURCE
☐ SPECIFY NAME
☐ CONNECT
☐ LIFT
☐ AUTO ROUTE
☐ MANUAL ROUTE

Fig. 3-42

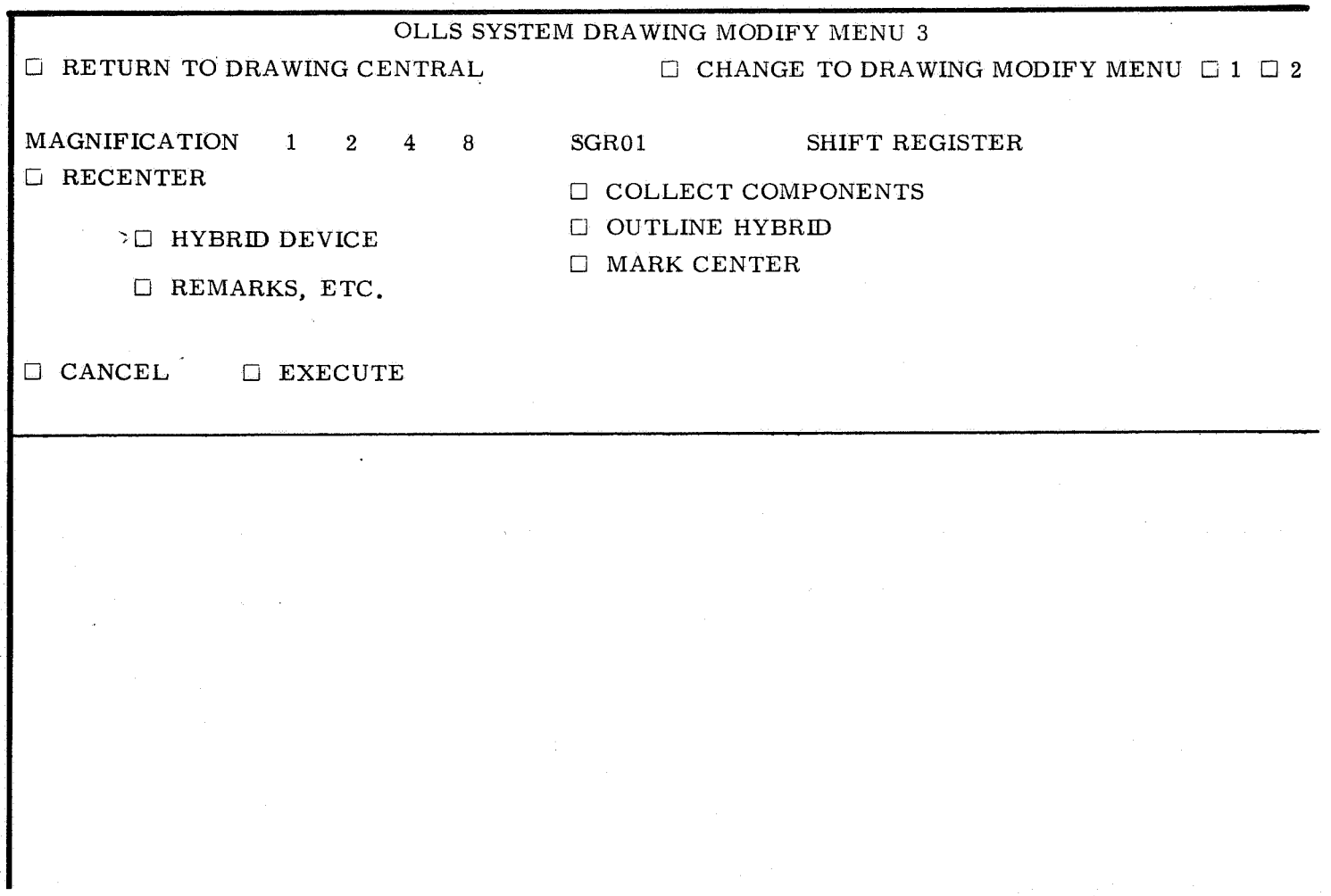

OLLS SYSTEM DRAWING MODIFY MENU 3
☐ RETURN TO DRAWING CENTRAL
☐ CHANGE TO DRAWING MODIFY MENU ☐ 1 ☐ 2
MAGNIFICATION 1 2 4 8
SGR01 SHIFT REGISTER
☐ RECENTER
☐ HYBRID DEVICE
☐ REMARKS, ETC.
☐ COLLECT COMPONENTS
☐ OUTLINE HYBRID
☐ MARK CENTER
☐ CANCEL
☐ EXECUTE

Fig. 3-43

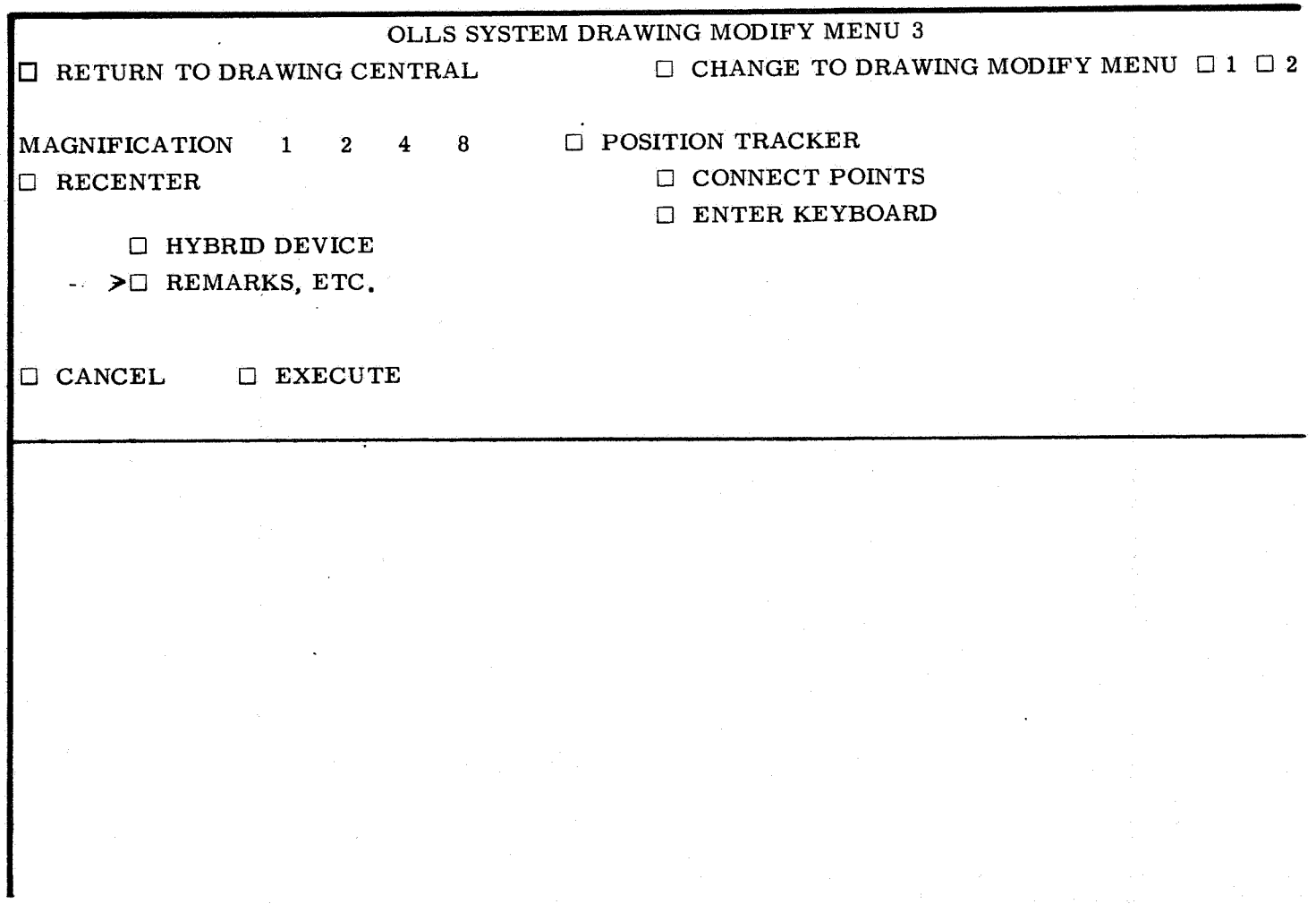

OLLS SYSTEM DRAWING MODIFY MENU 3
☐ RETURN TO DRAWING CENTRAL
☐ CHANGE TO DRAWING MODIFY MENU ☐ 1 ☐ 2
MAGNIFICATION 1 2 4 8
☐ POSITION TRACKER
☐ RECENTER
☐ CONNECT POINTS
☐ ENTER KEYBOARD
☐ HYBRID DEVICE
>☐ REMARKS, ETC.
☐ CANCEL
☐ EXECUTE

Fig. 3-44

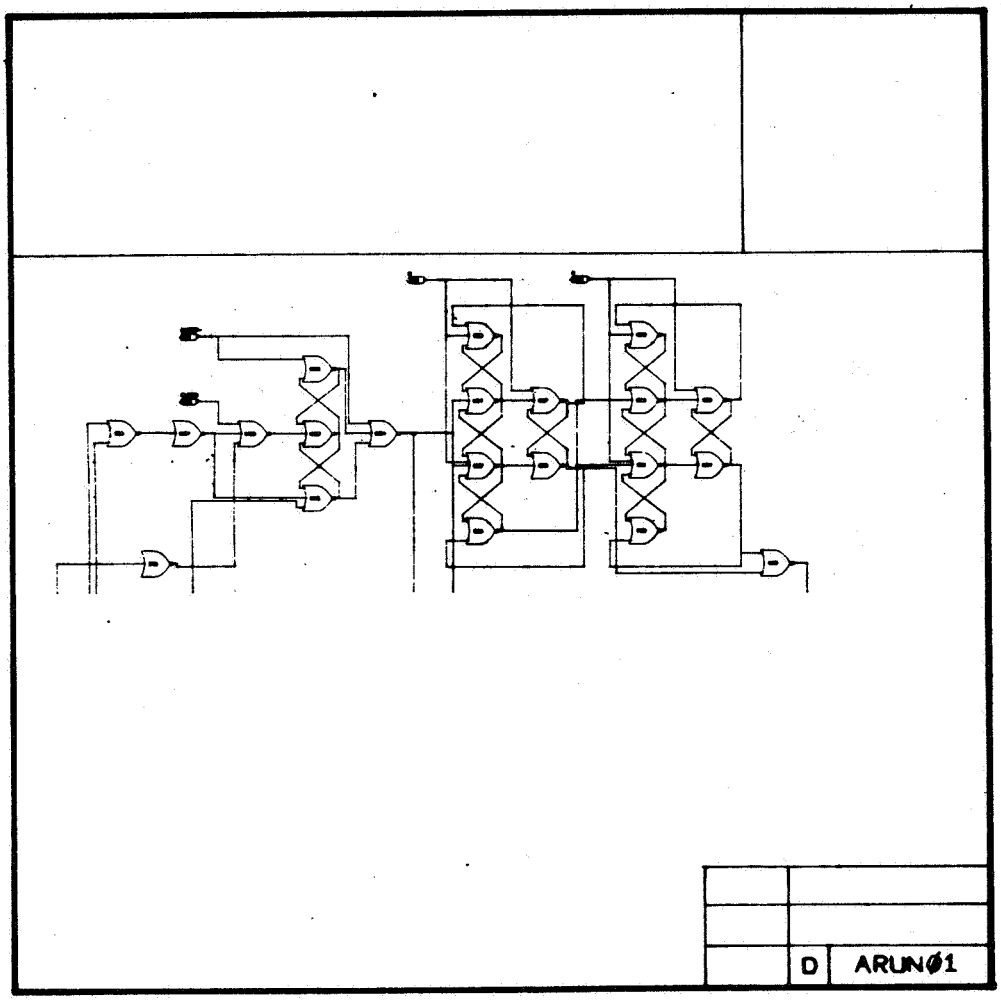


Fig. 3-45

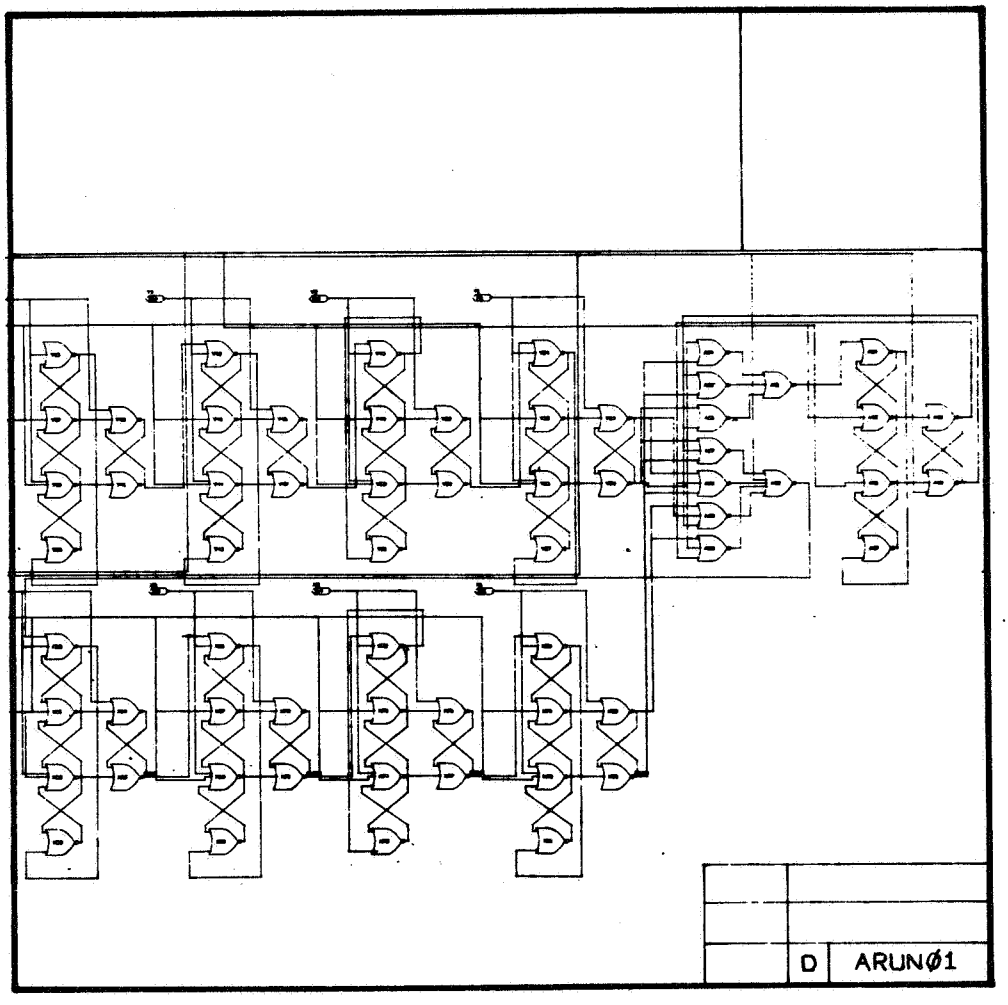


Fig. 3-46

E-2265

DISTRIBUTION LIST

Internal

R. Alonso
R. Battin
R. Crisp
E. Duggan
J.B. Feldman
J. Flanders
S. Forter
W. Grigg
Eldon Hall
A. Harano
D. Hoag
A. Hopkins (25)
F. Houston
H.R. Howie
J. Kingston*
A. Laats
J. Laning
L. Larson
James Miller
John Miller
C. Muntz
J. Nevins
J. Nugent
J. Pennypacker
R. Ragan
J. Sabo
G. Schwartz
H. Thaler
M. Trageser
R. Woodbury
W. Wrigley
Apollo Library (2)
MIT/IL Library (6)

*Letter of transmittal only.

External

NASA/ERC (50 + 1R)
575 Technology Square
Cambridge, Massachusetts
ATTN: KC/Computer Research Laboratory
Mr. D.J. Kelleher (Letter of Transmittal only)