T H E U N I V E R S I T Y O F M I C H I G A N

Memorandum 26

THE DISCRETE, LOGICAL DESIGN, SIMULATION SYSTEM

J.R. Guskin
T.J. Dingwall

CONCOMP: Research in Conversational Use of Computers
F.H. Westervelt, Project Director
ORA Project 17449

supported by:

ADVANCED RESEARCH PROJECTS AGENCY
DEPARTMENT OF DEFENSE
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

April 1970

TABLE OF CONTENTS

1. INTRODUCTION

The Discrete, Logical Design, Simulation System is a relatively straightforward, high-speed method for simulating combinatorial and delay logic. Such a programming system would provide a relatively quick way for the logic designer to determine any flaws in his network. He can, thus, debug his design and immediately test his corrections.

The programs are structurally divided into four main parts: the Command Language Interpreter, the Data Structure Manipulation Routines, the Simulator and various specialized input and output routines which communicate directly with the data structure routines. The primary programming language used was the System/360 G-level assembler language; however, we used the FORTRAN IV G-level compiler along with an extended runtime system to write the Command Language Interpreter because it enabled us to quickly and inexpensively make alterations to the program.

The algorithm which allowed the construction of a very fast simulator is described under the section dealing with the "Ordering Algorithm."

2. COMMAND LANGUAGE INTERPRETER

A highly interactive and versatile command language interpreter was included in the package as an interface between the programs themselves and the user. The commands are designed to be simple to use and quickly expandable. A complete error-detection system has been incorporated so that the user needn't worry about "blowing the system down." He is reminded, if he forgets, for example, that certain signal and package names have been used before. Besides this interactive mode there is a stored command mode and a batch mode, which allow a whole data-structure (network) to be efficiently loaded into the system.

The CLI consists of three primary parts: the Parser, the Input Control Program, and the Command Execution Control, all of which are interrelated and which communicate with each other through a series of subprogram calls.

The parser is the only one of the routines which is fully input-device-independent. It retrieves an input line from the Input Control Program (INSEPT) and breaks it up into a four-character command name and a set of four-character arguments, all of which are either padded with trailing blanks or truncated at four characters if necessary. Commands are decoded using a table supplied by the Command Execution Control. Matches between entries in the table and commands entered are attempted using the

the number of characters entered as long as that number is four or less. Ambiguities which arise are not resolved and are returned to the user as an error. For example:

Real command Names

(entries in the table)

INPUT INIT

Ambiguous matches

I IN

If three characters are entered

INP => INPUT

INI => INIT

if an argument list is null, a count of zero is returned to the execution control, otherwise the total number of arguments (up to thirty) is returned with the argument array. If a null argument is entered, it is replaced by four blanks. If a number is requested as an argument, then it must be an unsigned integer between 0 and 9999. Any violation of this syntax (in all cases it is actually similar to the syntax of a FORTRAN CALL statement) halts parsing immediately (except for numerical conversion) and generates an error comment.

The Input Control Program provides for the entry of a command from the master console (the teletype or card-reader) or from a file or other device (see SOURCE command). If a line is being read in the interactive Master Command Mode, then each line (except the first) is preceded by the

characters FEED ME ?. Otherwise the prefix character is a "?". If this program is in Store Command Mode, and end of file will perform an automatic TERM command which restores the program to Master Command Mode. The statute of the program is controlled by a set of subroutines which provide addresses for the reading routines and status switches.

The Command Execution Control is the prime unit of the CLI. It is here that the commands are really executed and most errors are detected. This is really the interface to the data structure and simulation package routines.

The parser returns the decoded number of the command and an array of arguments. The control program then performs a branch to the appropriate routine to execute the command.

For complete descriptions of all the available commands see Appendix B.

There is one thing in particular to note about the error detection facility in the Command Execution Control: as soon as an error is detected the command execution is aborted and, in general, if the user is in Stored Command Mode he is returned to Master Command Mode (commands are read on the master console).

3. THE DATA STRUCTURE

The data structure used to describe a network internally consists of three chained lists with various interconnections. The lists are as follows:

A. Package Definition Chain

Each element of this chain describes one possible package type in the circuit, such as an AND or a DELAY gate. Fields in a package definition are as follows:

NEXTDEF - fullword pointer to the next package definition.

SWITDEF - 1-byte set of switches used to initialize the switches in a package instance of this type.

TYPEDEF - a 1-byte code indicating the type of package this element represents. Possible values are:

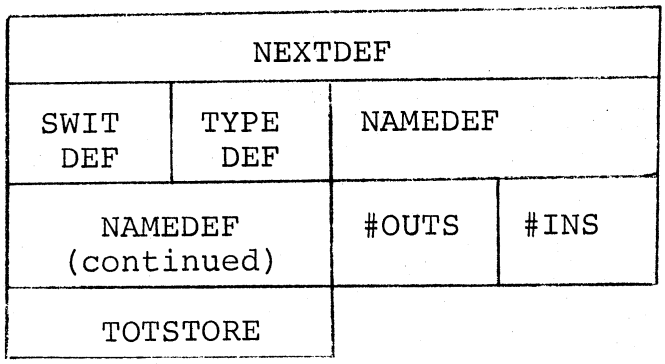
CODE	GATE TYPE
4	AND
8	OR
12	NAND
16	EXCLUSIVE OR
20	NOT
24	UNIT DELAY

NAMEDEF - 4-character name of the package type.

#OUTS - 1-byte field giving the number of outputs of the package.

#INS - 1-byte field giving the number of inputs to the package.

TOTSTORE - Halfword integer giving the total amount of storage needed for each package instance of this type.



A package definition may be referenced through the subroutine 'DEFTN'.

B. Package Instance Chain

Each element of this list represents an actual instance of a combinational logic module in a circuit. The chain is arranged so that simulation of the packages may be performed in the given order. In addition to regular logic modules, the chain includes special blocks for defining primary inputs. These special blocks are identical to instances with a type code of zero, 8 outputs, and no inputs.

Fields in a package instance are:

NEXTINST - fullword pointer to the next package instance.

SWITINST - 1-byte set of switches; presently only the high-order bit is used during ordering of the network.

TYPEINST - 1-byte type code indicating the kind of gate. This field is identical to the type code in a package definition.

NAMEINST - 4-character name of the package instance.

ORDER - halfword integer giving the relative order in which this gate should be simulated. This information is redundant with the ordering of the chain itself, but is convenient to use when ordering the network. Note that more than one instance may have a given numerical order.

BACKINST - fullword back-pointer to the previous instance.

PKGDEF - fullword pointer to the corresponding package definition for this instance.

OUTSIGS - 1 or more 8-byte fields defining the outputs of this module. The field

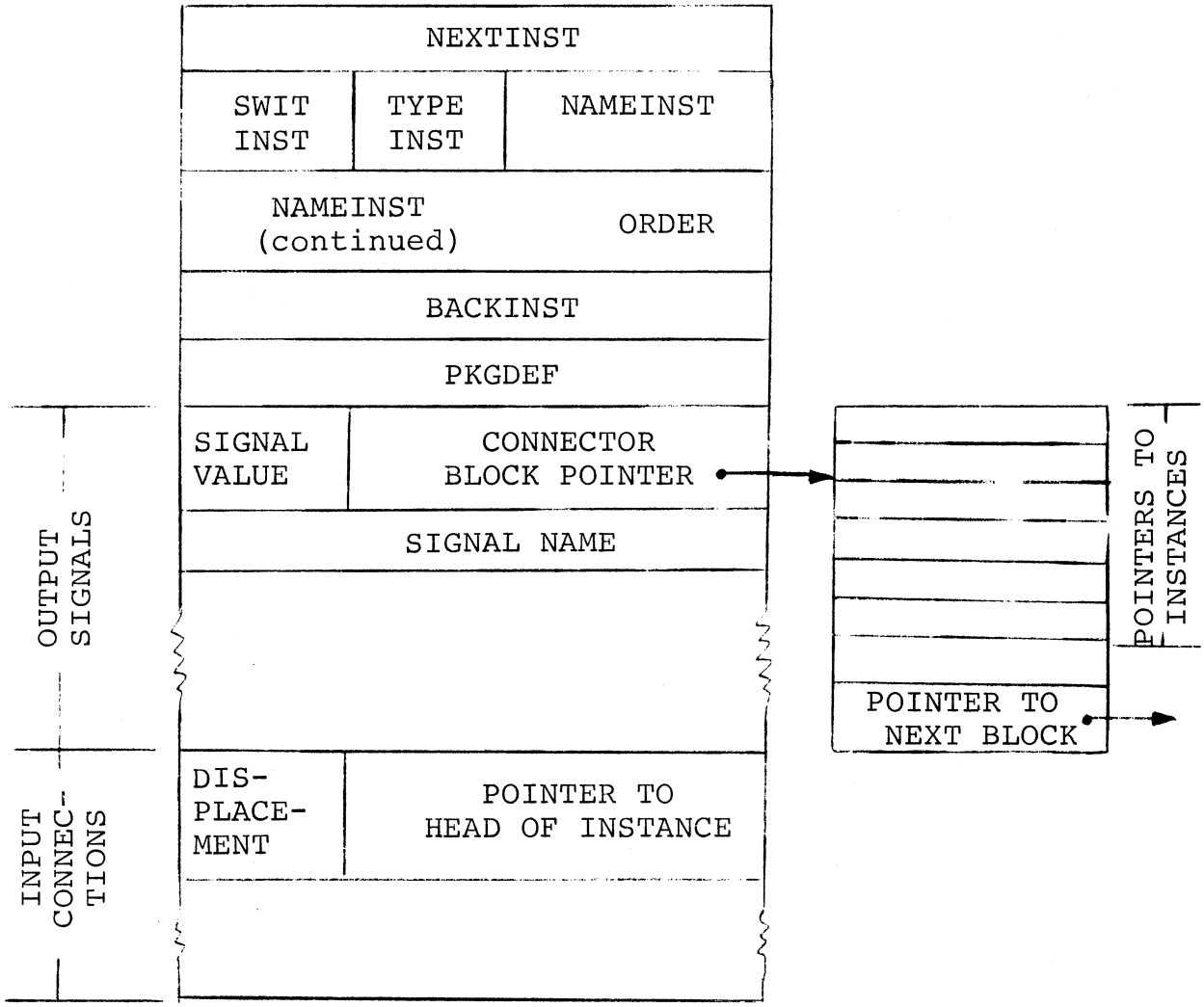
consists of a 1-byte signal value, a 3-byte pointer to a connection block, and a 4-character signal name. The signal field has the possible values 0, 1, or 2, standing for 0, 1, and x respectively. Also the high-order bit is used to indicate whether or not the signal was changed in the last time-interval.

The 3-byte pointer is an anchor for a chain of connector blocks indicating what packages this signal is connected to. Each connector block consists of 7 fullword pointers to package instances, and an eighth fullword pointing to the next connector block. Zeros are used in the pointers to indicate no connection.

These connector blocks are used only during ordering, and not during simulation.

After the output signals is one four-byte field for each input to the instance. The field is split into a 3-byte pointer to the head of a package instance, and a 1-byte displacement from the start of that instance to the signal connected to the given input. If no connection is made,

the field is zero. The input pointers are used only during simulation and not during the ordering of the network.



Instances may be referenced via the subroutine PACKAG.

C. Delay Instance Chain

All delays in a network are on this chain. The chain elements have exactly the same format as instances in the combinational chain. Delays are put on a separate chain only for convenience

during simulation.

There is no ordering of delays in the chain.

4. THE ORDERING ALGORITHM

The network is ordered dynamically as connections between packages are made. In this way cycles are detected immediately and an ordering need not be done before each simulation.

All delays and blocks defining primary inputs have order 0. All combinational modules are given the order 1 initially and have their orders increased as connections are made.

Assume an output of package A has just been connected to an input of package B. The following algorithm is applied to A and B:

(1) If B is a delay, the network is ordered, and we may stop.

(2) If the order of A is less than the order of B, then we are done.

(3) Otherwise, set the order of B equal to the order of A+1.

(4) Apply this algorithm recursively to B and each package connected to an output of B.

Cycles are discovered in the following manner: at step 4 in the algorithm a switch is set in B indicating that B has been reordered.

If at some lower level of application, a switch is found to be on, then a cycle exists in the network. When

a cycle is discovered, normally the newest connection is broken, and an error comment is printed. It is possible, however, that before a cycle is discovered, many packages may be given too high an order; this does not hurt anything, since only too low an order will cause errors in simulation.

5. THE SIMULATOR

Simulation is performed in three separate passes.

In pass one, the outputs of all delays are set. The value is obtained from an internal storage area which was set in the previous pass 3.

In pass two, all combinational logic modules are evaluated. Since the chain has the proper ordering, each package can be evaluated without concern as to whether its inputs have been set. Because the types of packages are so similar, the evaluation of combinational packages is almost totally table-driven. The evaluation is also arranged so that an unconnected input is considered to be "tied down" and hence has no effect on the simulation.

In pass three, the input to each delay package is stored in a field inside the package instance. This method of handling delays makes it unnecessary to order delays.

All simulations are three-valued, i.e., a signal may take on the value 0, 4, or X. This allows one to make sure certain events occur independently of other conditions. For instance, a register should be able to be cleared regardless of its contents. Another advantage of three-valued simulation which was not implemented is the ability to propagate an X between any signal changes and thereby

determine any possible SPIKES. This feature would not be too difficult to implement in our system.

APPENDIX A. SYNTAX OF INITIALIZATION

STATEMENTS

A special mode for the definition of a network is entered under the command INIT. The format of the statements accepted is given below:

Each statement defines one package instance and all its connections. The statements are in free-format in that the four fields may be separated by any number of blanks.

The first field is the name of the instance this statement refers to. It may be 1 to 4 characters and is padded with blanks to 4 characters.

The second field is the name of the package type. Presently, the only acceptable entries are: DELY, AND, OR, NOT, EXOR, NAND, and NOR.

The third field defines the signals connected to the inputs of the package. It consists of a list of signal names, separated by commas and all enclosed in parentheses:

(SIG1, SIG2, ...)

The named signals are connected to the inputs of the package in the order given. Two consecutive commas indicate the corresponding input is to be left unconnected.

The fourth field gives the names of the output signals associated with the package. As in the case of inputs, the field consists of a list of signal names separated by

6. POSSIBLE ADDITIONS AND EXTENSIONS TO THE SYSTEM

The following are suggestions for possible additions and extensions to the system.

1. The use of a CRT for interaction both in defining a network and simulating it.
2. The use of a plotter for output of both circuit diagrams and simulation output.
3. Addition of partitioned logic, so that extensively used circuits could be defined once and then used in many different places. Modules such as flip-flops could then be used.
4. Addition of generalized combinatorial logic modules, defined only by a truth table.
5. Inclusion of such package types as core storage and microprogram storage.
6. The ability to generate various reports, such as cross reference listings, propagation delay and fan-out listings, etc.
7. Addition of further design aids, as packaging, placement, and wire routing algorithms.
8. The implementation of a parser to allow networks to be defined in a higher level language.
9. The inclusion of more aids during simulation, such as more sophisticated conditionals and looping, and the ability to dump the entire structure.

commas and enclosed in parentheses. It should be noted, however, that in the present version no package has more than one output.

A special format is used to define primary inputs. It consists of the word INPUTS, followed by a list of signal names, separated by commas and enclosed in parentheses. There can be no confusion with an ordinary statement, since INPUTS contains more than 4 characters.

The initialization process takes place in two passes. In the first pass, the statements are read in and the proper packages created. Also in this pass, a numbered listing of the statements is produced and any errors found at that time are printed under the proper statement.

In the second pass, the various connections are made. Any errors in this pass are printed with the number of the statement at fault. At the end of each pass, the number of statements flagged in that pass is printed.

Below is an example of a set of statements defining a simple set-reset flip-flop. The signals ON and OFF, respectively, set and reset the flip-flop, and the signal OUT is the output of the circuit. All other signals are internal.

```
INPUTS  (ON,OFF)
NOT1  NOT   (OFF)(T1)
AND1  AND   (T1,T3)(T2)
OR1   OR    (ON,T2)(OUT)
DEL1  DELY  (OUT)(T3)
```

APPENDIX B. BASIC INFORMATION COMMANDS

NAME: HELP

ARGUMENTS: The number of arguments is variable; each is the full four-character name of the command about which a description is desired.

PURPOSE: To provide information about commands.

PROTOTYPE: HELP(CREA,HELP)

. . . . . .

NAME: TYPES

ARGUMENTS: The number is variable; each is the name of a package definition.

PURPOSE: To provide information about package types.

PROTOTYPE: TYPE(ØR,XØR)

APPENDIX C. DATA STRUCTURE MANIPULATION COMMANDS

NAME: CREATE

ARGUMENTS: At least three

(1): the package type

(2): name to give instance

(3): name of an output

(4)etc., same as (3)

PURPOSE: To create a gate, name it, and name its outputs.

POSSIBLE ERRORS: (1) Type doesn't exist.

(2) Package name already exists.

(3) Output name already exists.

. . . . . .

NAME: DESTROY

ARGUMENTS: One

(1): package name

PURPOSE: To remove a gate from the network and "garbage collect" its storage.

NOTE: This is the only way to delete a gate or signal name from the system.

POSSIBLE ERRORS: (1) Package name doesn't exist.

(2) Everything is not disconnected

PROTOTYPES: CREA(AND,AND1,ØUT1)

DEST(AND1)

NAME: CONNECT

ARGUMENTS: Three

(1): name of output signal.

(2): name of package which has input.

(3): sequence number of input signal.

PURPOSE: To connect one gate to another and perform necessary reordering of the network.

POSSIBLE ERRORS: (1) Signal or package doesn't exist.

(2) Cycle detected in structure.

(3) Bad sequence number.

. . . . . .

NAME: DISCONNECT

ARGUMENTS: Two

(1): name of package which has input.

(2): sequence number of input signal.

PURPOSE: To disconnect one input lead from an output signal.

POSSIBLE ERRORS: (1) Package doesn't exist.

(2) Bad sequence number.

PROTOTYPES: CØNN(ØUT1,AND2,1)

DISC(AND2,1)

NAME:	SIMULATE
ARGUMENTS:	Variable (up to 30)
(n):	Name of signal to print out after simulation.
PURPOSE:	To actually perform the simulation of combinatorial and delay logic over one clock period.
NOTE:	If a trace is active (see TRACE command), then the signals named here are added to the others for one simulation only.
POSSIBLE ERRORS:	Signal name does not exist.
PROTOTYPES:	SIMULATE SIMU(ØUT1)

NAME:	TRACE
ARGUMENTS:	Variable (up to 30)
(n):	Signal name.
PURPOSE:	To print out the values of signals after each call to SIMULATE automatically. If no arguments are present this feature is turned off. Every call to this command deletes the previous ones.
POSSIBLE ERRORS:	No error checking is performed here, although errors might pop up when the signal values are printed out.
PROTOTYPE:	TRACE(ØUT2,ØUT3) TRACE

NAME: INIT

ARGUMENTS: None (however a file or device name must be given when it is asked for).

PURPOSE: To allow the bulk entry of gates and the connections between them (for further information see Appendix A and the subroutine description for INITAL).

PROTOTYPES: INIT

. . . . . .

NAME: SAVE

ARGUMENTS: None (same as for INIT)

PURPOSE: To save a data structure on a file or other device (for further information see the subroutine description for SAVE).

PROTOTYPES: SAVE

NAME:	INPUTS
ARGUMENTS:	Variable (up to 30)
(n):	Name of signal to create.
PURPOSE:	To create primary input signals into the network.
POSSIBLE ERRORS:	(1) Signal already exists.
PROTOTYPE:	INPUTS(INP1,INP2)

. . . . . .

NAME:	SET
ARGUMENTS:	Variable (up to 15) (1): name of signal to set. (2): value to set signal to: 9,1, or x ETC.
PURPOSE:	To set the value of a signal.
POSSIBLE ERRORS:	(1) Signal doesn't exist. (2) Value is not 0, 1, or X.
PROTOTYPE:	SET(ØUT1,0,ØUT5, X)

NAME:	PRINT
ARGUMENTS:	Variable (up to 30)
(n):	Name of signal.
PURPOSE:	To print out the value of the desired signal.
POSSIBLE ERRORS:	(1) Signal doesn't exist
PROTOTYPE:	PRINT(ØUT1,ØUT2)

. . . . . .

NAME:	OUTPUT
ARGUMENTS:	None
PURPOSE:	To print out the values of all those signals whose value has changed during the last simulation. All the package names currently in the data structure are printed out also.
POSSIBLE ERRORS:	None
PROTOTYPE:	ØUTPUT

APPENDIX D. STORED PROGRAM CONTROL COMMANDS

NAME: SØURCE

ARGUMENTS: None (a file or device name is requested).

PURPOSE: To alter the source stream so that commands may now be read from any file or device.

PROTOTYPES: SØURCE

. . . . . .

NAME: TERM

ARGUMENTS: None.

PURPOSE: To switch back to master command mode (i.e., to read commands from GUSER).

PROTOTYPES: TERM

. . . . . .

NAME: CØNT

ARGUMENTS: None.

PURPOSE: To switch back to stored command mode. If SOURCE has never been given then this has no effect.

PROTOTYPES: CØNT

NAME: GØTØ

ARGUMENTS: At least 1 (up to 29).

(1): line number to transfer to

(2): name of signal

(3): value to use in comparison

(any number of signals and values may be given)

PURPOSE: To provide an unconditional transfer to the line number specified on (1) if only 1 argument is present.

If more than 1 argument is present then a conditional transfer is made if and only if all the signals' current values are equal to those stated in the GOTO.

POSSIBLE ERRORS: (1) Signal name doesn't exist.

(2) Invalid value specified.

PROTOTYPES: GØTØ(1000)

GØTØ(5,A,1,B,X)

NOTE: Line number is an unsigned integer between 0 and 9999.

NAME: ECHØ

ARGUMENTS: 1 argument

(1): switch

PURPOSE: To turn on and off a global switch which determines whether command lines read in stored command mode are to be printed out.

If switch=0 the echo is turned on.

If switch=1 the echo is turned off.

PROTOTYPE: ECHØ(0) (this is the default).

. . . . . .

NAME: PREFIX

ARGUMENTS: None.

PURPOSE: To turn prefixing off in general command mode.

PROTOTYPE: PREFIX

APPENDIX E. OTHER COMMANDS

NAME: MTS

ARGUMENTS: None.

PURPOSE: To return to the system with the option of reentering the command mode of this package by issuing a $RESTART.

PROTOTYPES: MTS

. . . . . .

NAME: END

ARGUMENTS: None.

PURPOSE: To return the system and say goodbye to this package.

PROTOTYPES: END

APPENDIX F. SYSTEM SUBROUTINES

NAME: PACKAG

PURPOSE: To search through the data structure and return the pointer to the package requested.

CALLING SEQUENCE: FORTRAN IV

PTR=PACKAG(NAME)

NAME=4-character name of gate to look for

PTR=value returned by routine PTR to the package found).

RETURN: RC=0 everything O.K.

RC=4 didn't find NAME

PTR is set to 0

NOTE: Must be declared INTEGER*4

NAME: DEFTN

PURPOSE: To search through the data structure and return the pointer to the package definition requested.

CALLING SEQUENCE: FORTRAN IV

PTR=DEFTN(NAME)

NAME=4-character name of package definition

PTR=pointer to the definition (returned by routine).

RETURN: RC=0 everything O.K.

RC=4 didn't find name NAME, PTR is set to 0.

NOTE: Must be declared INTEGER*4

NAME: SIGNAL

PURPOSE: To search through the data structure for a desired signal name and return a pointer to the instance which contains it with a displacement to the particular signal found.

CALLING SEQUENCE: FORTRAN IV

PTR=SIGNAL(NAME)

NAME=4-character name of signal to look for.

PTR = return value - the low-order 24 bits is the address of the gate containing the signal. The high-order 8 bits provides the displacement to the signal.

RETURN: RC=0 everything O.K.

RC=4 didn't find NAME; PTR is set to 0.

NOTE: Must be declared INTEGER*4

NAME: CREATE

PURPOSE: To enter an instance of a package definition (i.e., a gate) into the system data structure.

CALLING SEQUENCE: FØRTRAN IV

CALL CREATE (TYPE,NAME,PTR,ARRAY)

TYPE=pointer to the package definition as returned by DEFTN.

NAME=4-character name of this gate.

PTR =fullword PTR to the package created.

ARRAY=array of 4-character output signal names.

RETURN CODES: RC=0 everything is O.K.

RC=4 bad arguments. (TYPE is equal to 0)

NAME: SETSIG

PURPOSE: To set the values of signals already defined in the data structure.

CALLING SEQUENCE: FØRTRAN IV

CALL SETSIG(PTR,VALUE,...)

PTR=pointer (as returned by SIGNAL) to signal desired.

VALUE=0,1, or 2 which stand for 0, 1 or X.

PTR & VALUE may be repeated as many times as desired.

RETURN: RC=0 always.

NAME: GETSIG

PURPOSE: To return the value of a particular signal.

CALLING SEQUENCE: FØRTRAN IV

VALUE=GETSIG(PTR)

PTR=PØINTER to signal desired (as returned by SIGNAL).

VALUE=0, 1 or 2 - this is the current value of the signal.

RETURN: RC=0 always.

NOTE: This must be declared INTEGER*4.

. . . . . .

NAME: OUTS1

PURPOSE: To output all the values of all those signals whose value had changed since the last simulation. Also all the names of all the current combinatorial gates are printed.

CALLING SEQUENCE: FØRTRAN IV

CALL ØUTS1

RETURN: RC=0 always.

NAME: DELAY

FUNCTION: To create a delay package.

CALLING SEQUENCE: FØRTRAN IV

CALL DELAY(PKGNAME,SIGNAME,PKGPTR,INSTPTR)

ARGUMENTS: PKGNAME - 4-character name of package to be created.

SIGNAME - 4-character name of output signal package.

PKGPTR - Pointer to delay package definition.

INSTPTR - Pointer to instance created (returned).

RETURN CODES: None.

COMMENTS: The instance is created but no connections are made. The output signal is given the value X.

NAME: INPUT

FUNCTION: To create primary input signals to the network.

CALLING SEQUENCE: FØRTRAN IV

CALL INPUT(SIG1,SIG2,...)

ARGUMENTS: SIG1,SIG2,... Each argument is a 4-character signal name. Any number of arguments may be given.

RETURN CODES: None.

COMMENTS: Each signal is created but not connected. Each signal is initialized to the value X.

. . . . . .

NAME: REORDER

FUNCTION: To update the ordering of a network after a connection is made.

CALLING SEQUENCE: GR1 points to the output package supplying the signal.

GR2 points to the package receiving the signal in the new connection.

RETURN CODES: RC=4 - A cycle was found in the circuit. Some orderings may have been updated.

COMMENTS: 1. This routine is for internal use only.

2. This subroutine calls on itself.

NAME:	INSERT
FUNCTION:	To insert an instance in the proper place on the instance chain, according to its order.
CALLING SEQUENCE:	NON-STANDARD
ARGUMENTS:	GR1 points to the instance to be inserted.
RETURN CODES:	None.
COMMENTS:	This routine is for internal use only.

NAME: CONECT

FUNCTION: To connect two package instances and update the network ordering.

CALLING SEQUENCE: FØRTRAN IV

CALL CONECT(SIGPTR,PKGPTR,SEQNO.&1,&2)

ARGUMENTS:

SIGPTR - A 4-byte pointer and displacement to a signal as returned by 'SIGNAL'.

PKGPTR - Fullword pointer to the instance receiving the signal.

SEQNO - Fullword integer indicating which input is to be used (0 is the first input, 1 the second, etc.).

RETURN CODES:

&1 - Cycle found in circuit, connection not made.

&2 - Invalid sequence number of input already connected.

COMMENTS: To connect a signal named 'SIG' to the third input of a package named 'PKG', the following call would be used.

CALL CONECT(SIGNAL('SIG'), PACKAG('PKG'),2)

NAME: SIMULT

FUNCTION: To perform a simulation through one time interval.

CALLING SEQUENCE: FORTRAN IV

CALL SIMULT

ARGUMENTS: None.

RETURN CODES: None.

COMMENTS: This routine updates all signal values for the next time interval.

. . . . . .

NAME: SAVEDS

FUNCTION: To preserve the data structure in a form suitable for input to INITAL.

CALLING SEQUENCE: FORTRAN IV

CALL SAVEDS(FDVB)

ARGUMENTS: FDVB - Pointer to file or device usage block onto which statements are to be written.

RETURN CODES: None.

COMMENTS: For a description of the syntax of the statements, see Appendix A.

NAME: DISCNT

FUNCTION: To break the connection between two instances.

CALLING SEQUENCE: FORTRAN IV

CALL DISCNT(PKGPTR,SEQNO,&1)

ARGUMENTS: PKGPTR - Pointer to package receiving signal.

SEQNO - Fullword integer sequence number of input.

RETURN CODES: &1 - Invalid sequence number or input not connected.

COMMENTS: To destroy a connection to the third input of a package named 'PKG' the following call would be used.

CALL DISCNT(PACKAG('PKGE')2)

(0 is the first input, 1 the second, etc.)

NAME: DESTRY

FUNCTION: To destroy a package instance.

CALLING SEQUENCE: FORTRAN IV

CALL DESTRY(PKGPTR,&1)

ARGUMENTS: PKGPTR - Pointer to instance to be destroyed.

RETURN CODES: &1 - All connections not broken.

COMMENTS: All connections to a package must be broken before it can be destroyed.

. . . . . .

NAME: INITAL

FUNCTION: To read in definitional statements and to create the corresponding network.

CALLING SEQUENCE: FORTRAN IV

CALL INITAL(FDUB)

ARGUMENTS: FDUB - Fullword pointer to file or device usage block from which statements are to be read.

RETURN CODES: None.

COMMENTS: For a description of the syntax of statements read, see Appendix A.

APPENDIX G. SAMPLE RUN

Given below is a simulation of a four bit accumulator. The circuit has four input signals, BIT0 through BIT3, four output signals ACC0-ACC3, and three control signals, CLR, ADD, and CARY. The CLR signal clears the accumulator in one time step. An add is performed in two cycles, the first with ADD on, and the second with CARY on.

In the run below, the system is initialized with the network from a file. Then the network is simulated with various inputs. Statements entered by the user are underlined, while responses from the system are not.

S Y S T E M 3 6 0 / 6 7 S I M U L A T I O N
I N L O G I C A L D E S I G N P A C K A G E

PRODUCED BY: TOM DINGWALL AND JACK GUSKIN

DIRECTED BY: PROFESSOR EUGENE L. LAWLER

SUBMITTED IN FULLFILLMENT
OF THE REQUIREMENTS FOR THE
C . I . C . E . LOGICAL DESIGN COURSE
NO. 5 6 5

WELL HERE IT GOES

FEED ME

HELP

COMMAND HELP

TO GET INFORMATION ABOUT COMMANDS
TYPE HELP(COMMAND,...)
COMMANDS AVAILABLE ARE: CREATE,CONNECT,INPUT,
HELP,DISCONNECT,DESTROY,MTS,END,OUTPUT,
PRINT,SET,INIT,SAVE,GOTO,SOURCE,TERM,CONT,READ,TYPE

TYPE

AVAILABLE PACKAGE DEFINITIONS ARE:
AND , OR , NOR , NAND, NOT , DELY, XOR,

INIT

&ENTER FILE NAME ?

SOURCE

SOURCE LISTING

```
 1  INPUTS (CLR ,ADD ,CARY,BIT0,BIT1,BIT2,BIT3)
 2  NO15 NOT  (D6  ) (C15 )
 3  AN16 AND  (ADD ,BIT3,,,,,,) (N16 )
 4  NO14 NOT  (CLR ) (C14 )
 5  NO12 NOT  (D5  ) (C12 )
 6  AN11 AND  (ADD ,BIT2,,,,,,) (N11 )
 7  NO10 NOT  (CLR ) (C10 )
 8  NOT8 NOT  (D3  ) (C8  )
 9  AND6 AND  (ADD ,BIT1,,,,,,) (N7  )
10  NOT5 NOT  (CLR ) (C5  )
11  NOT4 NOT  (D2  ) (C4  )
12  AND3 AND  (ADD ,BIT0,,,,,,) (N3  )
13  NOT2 NOT  (CLR ) (C2  )
14  NOT1 NOT  (D1  ) (C1  )
15  AN18 AND  (CARY,BIT2,C15 ,,,,,) (N18 )
16  NOT3 NOT  (N3  ) (C3  )
17  AND1 AND  (N3  ,C1  ,,,,,,) (N1  )
18  AND2 AND  (C2  ,C3  ,D1  ,,,,,) (N2  )
19  OR1  OR   (N1  ,N2  ,,,,,,) (ACC0)
20  NOT7 NOT  (ACC0) (C7  )
21  AND8 AND  (CARY,BIT0,C7  ,,,,,) (N8  )
22  OR3  OR   (N7  ,N8  ,,,,,,) (N6  )
23  NOT6 NOT  (N6  ) (C6  )
24  AND4 AND  (N6  ,C4  ,,,,,,) (N4  )
25  AND5 AND  (C5  ,C6  ,D2  ,,,,,) (N5  )
26  OR2  OR   (N4  ,N5  ,,,,,,) (ACC1)
27  NO11 NOT  (D4  ) (C11 )
28  AN12 AND  (CARY,D4  ,N6  ,,,,,) (N12 )
29  AN13 AND  (CARY,BIT1,C11 ,,,,,) (N13 )
30  OR5  OR   (N11 ,N12 ,N13 ,,,,,) (R1  )
31  AN17 AND  (CARY,D6  ,R1  ,,,,,) (N17 )
32  NOT9 NOT  (R1  ) (C9  )
33  AND9 AND  (R1  ,C8  ,,,,,,) (N9  )
34  OR7  OR   (N16 ,N17 ,N18 ,,,,,) (R2  )
35  AN10 AND  (C9  ,C10 ,D3  ,,,,,) (N10 )
36  NO13 NOT  (R2  ) (C13 )
37  AN14 AND  (R2  ,C12 ,,,,,,) (N14 )
38  OR4  OR   (N9  ,N10 ,,,,,,) (ACC2)
39  AN15 AND  (C13 ,C14 ,D5  ,,,,,) (N15 )
40  OR6  OR   (N14 ,N15 ,,,,,,) (ACC3)
41  DEL6 DELY (ACC2) (D6  )
42  DEL5 DELY (ACC3) (D5  )
43  DEL4 DELY (ACC1) (D4  )
44  DEL3 DELY (ACC2) (D3  )
45  DEL2 DELY (ACC1) (D2  )
46  DEL1 DELY (ACC0) (D1  )

0 STATEMENTS FLAGGED IN PASS 1

0 STATEMENTS FLAGGED IN PASS 2
```

```
TRACE(ACC0,ACC1,ACC2,ACC3)
READ(CLR)
  CLR = 1
SET(ADD,0,CARY,0)
SIMULATE
ACC0 = 0
ACC1 = 0
ACC2 = 0
ACC3 = 0
SET(CLR,0)
READ(BIT0,BIT1,BIT2,BIT3)
  BIT0= 1
  BIT1= 0
  BIT2= 0
  BIT3= 0
SET(ADD,1,CARY,0)
SIMULATE
ACC0 = 1
ACC1 = 0
ACC2 = 0
ACC3 = 0
SET(ADD,0,CARY,1)
SIMULATE
ACC0 = 1
ACC1 = 0
ACC2 = 0
ACC3 = 0
```

READ(BIT0,BIT1,BIT2,BIT3)

BIT0= 1

BIT1= 0

BIT2= 1

BIT3= 1

SET(ADD,1,CARY,0)

SIMULATE

ACC0 = 0

ACC1 = 0

ACC2 = 1

ACC3 = 1

SET(ADD,0,CARY,1)

SIMULATE

ACC0 = 0

ACC1 = 1

ACC2 = 1

ACC3 = 1

READ(BIT0,BIT1,BIT2,BIT3)

BIT0= 1

BIT1= 1

BIT2= 1

BIT3= 1

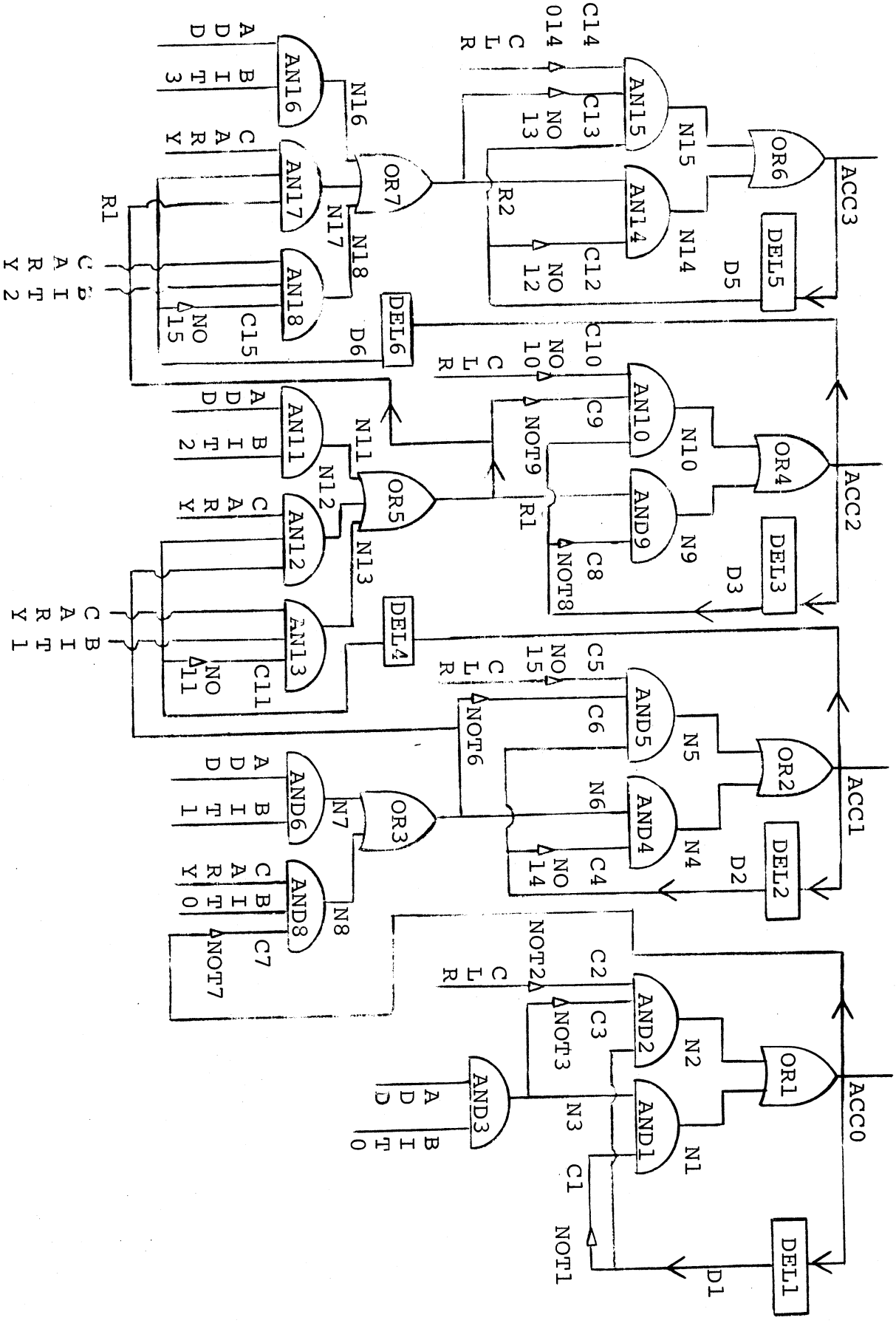
SET(ADD,1,CARY,0)

SIMULATE

ACC0 = 1

ACC1 = 0

```
ACC2 = 0

ACC3 = 0

SET(ADD,0,CARY,1)

SIMULATE

ACC0 = 1

ACC1 = 0

ACC2 = 1

ACC3 = 1

END
```



Circuit Diagram of Network Used in Sample Run -
A Four-Bit Accumulator

APPENDIX H. MODIFICATIONS AND ADDITIONS TO DATA-STRUCTURE ROUTINES

Modifications to the existing data structure routines and the addition of new subroutines were made to provide the following capabilities: eight-character package and signal names, the simulation of delays in all packages, and a macro definition capability.

The implementation of eight-character names merely involved the expansion of all name fields from four to eight bytes, and corresponding changes to existing subroutines.

Delay simulation encompassed slightly more extensive changes. First, the delay package type was eliminated, along with its chain and creation subroutine 'DELAY'. Next the ordering algorithm was scrapped, resulting in the deletion of the 'REORDER' and 'INSERT' subroutines. The order field in each package was replaced by two single byte fields giving the rise and fall delays of each package. Finally, each input signal area in each package was expanded from four to eight bytes. The extra four bytes hold the value and duration of the signal connected to the given input. Only minor modifications were necessary to the subroutines retained.

MACRO CAPABILITY

The macro capability allows the user to define more complicated packages, such as flip-flops, from the basic package types in the system. Once a user defines a package from a network, he can use the new package definition just like any other.

The concept is similar to that of macros in an assembly language. A macro package consists of one or more component packages and various internal connections. When a macro is "expanded" by the CREATE routine, a dummy package is created to represent the instance, all component packages are created, and internal connections are made between the component packages. All component packages and internal signals have hexadecimal zero names; hence their existence is completely hidden from the user.

DEFINITION OF MACROS

Macros are defined in a manner similar to a network. Whenever the network initialization or single statement routines encounter a "MACRO" statement, the subroutine DEFMAC is called and macro definitional statements are read. The first statement read gives the name of the new package and the output signals of the macro, separated by commas and enclosed in parentheses. The rest of the statements are of the form processed by the network initialization routine. An "EXTERNAL" statement identifies the inputs to the defined package. Each other statement defines

a single component package. Package names in these statements are ignored. Any signal names occurring in these statements which do not occur in the first statement or in an EXTERNAL statement are considered to be internal. A definition is terminated by an end-of-file, a "MEND" statement, or another "MACRO" statement.

The following example defines an R-S flip-flop by the name FLIPFLOP:

```
FLIPFLOP        (ON,OFF)
EXTERNAL        (SET,RESET)
GATE1  NOR      (RESET,OFF)(ON)
GATE2  NOR      (SET,ON)(OFF)
MEND
```

NEW DATA TYPES

Several new data types were required to implement the macro capability. A dummy instance was required for each use of a macro in the system. These dummy instances are similar to normal instances but are held on a separate chain. In a dummy instance, the type and delay fields are zero, and the signal value and connector block pointer are zero for each output signal. The second four bytes of each input field are zero, whereas the first four are anchors for chains of 32-byte blocks. The first 28 bytes of each block are seven four-byte fields containing a three-byte pointer to a component package of the macro and a one-byte sequence number

of an input in the component package. The last four bytes of the block are a chain pointer to the next block. By this means one can determine, given an input to a macro package, what the corresponding connections to component packages are.

Component packages can occur on either the regular chain or the macro chain. They are identified by the low-order bit (X'01') on the switch field, and also by a zero name field and zero signal names for internal signals. A component package may contain one or more non-zero signal names, corresponding to an output of the entire macro.

Finally, additional fields had to be added to package definitions for macros. The normal fields for a definition were left unchanged; and the following entries were tacked on the end: a halfword count of the number of component packages in the macro and a halfword count of the number of internal signals involved in the macro. Next there an entry describing the creation of each component package. Each entry consists of an eight-byte package type name (AND,OR), two one-byte delay entries, a halfword count of the number of output signals for the component package, and a four-byte entry for each output signal. Each four-byte field consists of a one-byte flag and a three-byte sequence number. If the flag is X'00', the sequence number is padded to eight bytes and used as a signal package. If the flag is X'FF', the sequence number is used as an index into the table of output signals provided for the

creation of the macro. This signal name is then used in the creation of the component package. Using this information, each component package can be created. The instance name used is an eight-byte binary integer which starts at one and is incremented for each component entry. Later these names are zeroed out.

Following the component package information, a field is provided for the connections to each component package. The order of connection information is the same as that for each component package creation. Each field contains a halfword count of the number of input connections to the component package followed by a four-byte field for each connection. Each four-byte field consists of two one-byte flags and a halfword sequence number. If the first byte is X'FF' the sequence number is used as an index into the output vector supplied to the CREATE routine. The package input is then connected to the signal found.

If the second byte is X'FF', the component package is to be used as an external input to the macro package. The sequence number tells which macro input is to be used, and an entry is made in the proper input chain on the dummy instance.

If both flags are zero, the sequence number is padded to eight bytes and used as a signal name for connection to the input.

In this way, all interconnections between component packages are represented.

NEW SUBROUTINES

Three new subroutines were added to the data structure routines:

PARSE - this subroutine is identical to the old INITAL with the following exceptions:

a) it accepts as arguments all the data structure routines and I/O routines it needs to call. Thus the subroutine calls can be intercepted and PARSE can be used for macro definition as well as initialization.

b) if no package name occurs on a package statement, a binary zero name is generated.

c) the 'MACRO' statement is accepted and a call to DEFMAC made.

d) the 'INPUT' statement was renamed 'EXTERNAL'.

DEFMAC - this subroutine parses the first statement of a macro definition itself and calls PARSE to analyze all following statements. It intercepts necessary subroutine calls and creates a package definition for the given macro.

SINGLE - this subroutine calls PARSE with one package statement. Thus packages may be created and connected individually.

MODIFICATIONS TO EXISTING ROUTINES

The following modifications were made to subroutines already written:

CONECT - if CONECT is called with a macro package as an input the routine calls itself using every component package input associated with the macro input.

CREATE - if a macro package is to be created, the following steps are taken:

a) a dummy instance is created and inserted on the dummy chain.

b) all component packages for the macro are created.

c) all interconnections between component packages are made.

d) all component package names and internal signals are zeroed.

PACKAG - this subroutine searches both the dummy chain and the "real" chain. Also zero package names cause an error return even if a package with a zero name exists.

SIGNAL - zero signal names cause an error return even if a zero signal exists in the network.

DISCNT - if a macro package input is to be disconnected, DISCNT is called recursively for each compon- the package input corresponding to the macro input.

DESTRY - an addition to destroy macros was planned but never implemented.

SAVEDS - component packages are ignored. The dummy chain is also scanned.

INITAL - this routine now calls PARSE to do all the work.

Also included in this extended system is a form of partition, which allows the user to define sets of gates which are to be simulated together. Using this concept the user can simulate large scale parallel processing and other asynchronous logic.

In this system each partition is referred to as a Network. Associated with each Network is a user-defined set of macros, a clock, a scale-factor, a list of signals to be traced when this Network is simulated, a set of package instances (gates) and a unique name. Some new subroutines were needed in order to implement the Network concept. Also additional commands were added to the Command Language Interpreter to provide the facility to the system user.

Other additions to the system included a better signal value printing scheme, a way to print the names of all the package instances currently defined in the active Network, and a way to look at and alter data included in a package instance and its definition.

NEW COMMANDS

LOOP CONTROL

CALL: LSET(NUMBER,MIN,MAX,INCR,LOC)

PURPOSE: To provide looping information to the system.

ARGS:

NUMBER the number of the loop this information is for ($1 \leq$ NUMBER ≤ 25)

MIN the starting number for this loop

MAX the final number for this loop

INCR the increment used to get from MIN to MAX

LOC the MTS line number to transfer to if MAX is not reached

OPERATION: Whenever the LOOP command is given with the same number as NUMBER the INCR is added to MIN and the result is checked against MAX. If the new MIN is larger than MAX no branch is taken. However if it is smaller a transfer in the command stream is made to LOC.

. . . . . .

CALL: LOOP(NUMBER)

PURPOSE: To actually perform the looping incremental updata, compare and transfer. (See LSET for further information.)

ARGS: NUMBER the identification of the loop data to be processed

EXAMPLE OF LOOPING

```
                MIN MAX INCR
 1    LSET(1,1,10,1,5)
 2    SET(A,1,B,1)
 3    CLKSET(0)
 4    SCALE(5)
 5    SIMULATE
 6    PRINT(D,E,F)
 7    GOTO(10,G,0)
 8    READ(G)
 9    GOTO(5)
10    LOOP(1)
```

At statement 10 a transfer is made to statement 5 iff (MIN=MIN+INCR) $\leq$ (MAX)

NETWORK HANDLING

CALL: NCRE(NETWRK)

PURPOSE: To create a new network but not make it the active network

ARGS: NETWRK the 8 character name of the new network

. . . . . .

CALL: NSET(NETWRK)

PURPOSE: To make a network the active network

ARGS: NETWRK the 8 character name of the network

. . . . . .

CALL: NCON(NETWRK,SIGNAL,PACKAG,SEQNO)

PURPOSE: To connect an input signal in the current network to an output signal in another network

ARGS: NETWRK name of network to find output signal in

SIGNAL the name of the output signal

PACKAG the name of the package in the current network which contains the input signal

SEQNO the sequence number for the input signal

CALL: CLKSET(CLOCK)

PURPOSE: To alter the value found in the current network's clock

ARG: CLOCK the new value for the current network's clock

. . . . . .

CALL: SCALE(SCALE)

PURPOSE: To alter the value found in the current network's scale factor

ARG: SCALE the new value for the current network's scale factor

SERVICE COMMANDS

CALL: ALLP

PURPOSE: To list the names of all the instances in the current network

ARGS: none

. . . . . .

CALL: DATA(PACKAGE)

PURPOSE: To print all pertinent data about PACKAGE. This is primarily used for system debugging.

ARGS: PACKAGE the name of the package to get data from

. . . . . .

CALL: DSET(PACKAG,DELY1,DELY2)

PURPOSE: To alter the high to low and low to high input delay "times" in a package. In the case of a SINGSHOT this changes the amount of "time" the signal remains high.

ARGS: PACKAG the name of the package to change

DELY1 the new low to high delay value

DELY2 the new high to low delay value

CALL: TSIM(NUMBER,SIGNALS)

PURPOSE: To simulate a network a number of "time scale" times, during each time unit printing the values for a set of signals.

ARGS:

NUMBER number of times to simulate active network (for each of these times the network is simulated "SCALE" number of times

SIGNALS a string of signal names whose values are to be printed

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) THE UNIVERSITY OF MICHIGAN CONCOMP PROJECT	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3. REPORT TITLE

THE DISCRETE, LOGICAL DESIGN, SIMULATION SYSTEM

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Memorandum 26

5. AUTHOR(S) (First name, middle initial, last name)

J.R. Guskin and T.J. Dingwall

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
April 1970	15	0

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
DA-49-083 OSA-3050 b. PROJECT NO. c. d.	Memorandum 26 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DDC.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Advanced Research Projects Agency

13. ABSTRACT

This paper describes the design and implementation of a programming system for simulating a logical network. It is written in a form usable for a user's guide for this system. The system is intended to be used in the instruction of students in the area of logical design.

DD FORM 1 NOV 65 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
logical design simulation macro facility interactive design						