1972

Digital system design and simulation

Ronald William Borgstahl
Iowa State University

INFORMATION TO USERS

This dissertation was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again – beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

University Microfilms

300 North Zeeb Road
Ann Arbor, Michigan 48106

A Xerox Education Company

Digital system design and simulation

by

Ronald William Borgstahl

A Dissertation Submitted to the

Graduate Faculty in Partial Fullfillment of

The Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major: Electrical Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University

Ames, Iowa

1972

TABLE OF CONTENTS

INTRODUCTION

During the past eight to ten years, a great deal of effort has been exerted in the general area of computer-aided design of hardware systems. This effort has taken place both at the university level, to satisfy purely research interests, and at the industry level, to satisfy actual manufacturing needs. As might be expected, this general area is a very broad one. It can imply a circuit described at the component level in order to analyze its transient characteristics, or it could be an entire digital computing system described in such a manner such that the overall system's characteristics can be analyzed.

The motivations for having these systems are well known and well published, however, in the interest of completeness, they will be repeated here. Some of them are: (1) reduction in over-all system development cost, (2) increase in problem analysis capabilities, (3) increase in the speed of design, (4) improved documentation, (5) logic and implementation errors can be detected prior to hardware construction, and (6) allows more flexibility to obtain more detailed information, and indeed, to obtain information that is not obtainable under actual hardware operation due to the lack of control over initial and external stimuli. In addition, when used as a teaching aid in the university, they provide the student with information that simply is not practicable when only hardware is available.

Computer-aided design can be broken down into basically three areas. They are: (1) synthesis, (2) analysis, and (3) simulation. Typically, once a system exists to handle any or all of these three areas, and once

the user's system has been put into a form which is acceptable to a computer, then other "by-products" are relatively easy to obtain. A few of these might be: (1) total system documentation in a standardized format, (2) partitioning of the system's elements, (3) assignment and placement of elements, (4) board layout and wire routing, and (5) parts lists of required material for constructing the system.

These "by-products" are not of interest to this research project mainly because they consist basically of book-keeping type algorithms, and also because they require a very well-defined set of hardware elements.

Instead, the first objective of this research effort was to provide a means by which a user can accurately and concisely model his digital system, independent of any predefined logic families or present-day state of the art hardware elements. Once an accurate model has been defined, then the second objective was to provide an accurate simulation of his system. Of almost equal importance was a secondary goal, which was to provide a simulation system which would require minimal amounts of both computer time and capacity.

As has been pointed out by Breuer (1) and others, there are basically five levels at which simulation of digital systems is performed. They are:

(1) System Level: At this level, the over-all system's properties are studied. The primary interest here is timing analysis of the system.

(2) Register Transfer Level: At this level, the functional design is evaluated. Typically, this is done by running actual programs on

the system being simulated.

(3) Logic Level: At this level, the system is described using logic equations (i.e., Boolean equations). The logical consistency of the design is then verified.

(4) Gate Level: At this level, the system is described by the set of actual hardware elements from which it is to be constructed, along with their interconnections. Typically, the output at this level is a state-time map of the system's logic signals.

(5) Circuit Level: At this level, the system is described at the diode, transistor, and resistor level. This is usually used to study the transient behavior of circuits.

The area of concern in this work is that of simulating at the gate level. This particular area was chosen for several reasons. (1) For one reason or another, the bulk of work already performed in the area of computer-aided design, has been done in the other four areas. (2) Most of the existing simulators in this area have had varying degrees of inadequacy such as, inaccurate models, models which are not user definable, inefficient simulation, and in some cases, they are merely analyzers and not simulators. (3) A simulator operating at the gate level, as stated earlier, can be a very useful instructional aid since it simply is not possible, nor necessarily desirable, to allow students to bread-board up even a modest size digital system.

The first two reasons for motivation will be elaborated upon later in the literature review.

Basically, a simulator must consist of three parts: (1) a means by which the system's building blocks can be defined, (2) a means by which the total system model can be described based upon these building blocks, and (3) a means of exercising the model and monitoring its responses. This then implies the necessity of a computer language which can describe both the structural properties (1 and 2), and the behavioral properties (3). A new language is desirable because the existing general purpose simulators do not have a convenient means of specifying systems at the gate level, and secondly, as will be shown later, the language required to adequately exercise such a system can be much more simple than the general purpose simulators, and hence, more desirable and usable from the viewpoint of a design engineer.

Briefly, the following unique results have been obtained in this work:

1. Any integrated circuit family can be exactly modeled. For obvious reasons this is necessary, but it also means the simulation system will not be obsoleted by changes in hardware technology.
2. Actual element modelling is facilitated because of the high level primitive macro set.
3. The speed of operation of the simulator was not sacrificed by having flexibility in the input specifications. In fact, the initial evidence indicates it is faster than the existing software simulators.
4. The mechanics of specifying macros, networks, and exercisers are easy to learn since they have been tailored to meet the specific requirements of gate level simulation.

5. The system's output is in a form which can easily be interpreted by the design engineer.
6. A relatively large amount of logic can be simulated with a reasonable amount of IBM 360/65 space and time.
7. The entire system is built modularly to facilitate future modification and expansion.

LITERATURE REVIEW

A brief description of this simulating system will be given at this time in order to give a better feel for its relationship with respect to other existing systems.

First of all, the building blocks (i.e., system elements such as gates) are defined by a series of macro definitions. These macros are defined in terms of a set of predefined, relatively high-level set of primitive macros, or in terms of macros defined elsewhere by the user. With the building blocks defined, then the system network is defined based upon these macros. This network description is input into a network compiler which generates a PL/1 program that represents the logical characteristics of the network. The third step is to define the simulation control program. This description is fed into another compiler which also generates a PL/1 program. These two programs are then compiled by the PL/1 compiler and are used to control and complete the definition of the simulator. The simulator is of the event directed type, which will be elaborated on later.

Since this simulation system is based upon a set of defining macros, it has been given the acronym (since all programs need acronyms) of MACSIM. In addition, the simulation phase will be referred to as MACSIM2, and all of the preceding steps will be referred to as MACSIM1.

Ideally, it would be desirable to have a simulation system that includes at least the first four areas of simulation previously described. The system described in (2) at IBM is such a system. The fifth area, circuit level simulation, is actually in an area of its own and is not pertinent when total digital systems are to be evaluated.

As stated earlier, there were several reasons why such a task was not undertaken in this research project. First, a system of this magnitude typically involves the expenditure of several thousand man-hours to implement. Secondly, a great deal of effort has already been expended in the first three areas that deal primarily with system synthesis.

When simulating at the system level, usually a general purpose simulator can be used. Some of the more common ones are GPSS, SIMSCRIPT, SIMULA, and GASP. As their name implies, these simulators are not restricted to simulation of solely digital systems.

One of the most active areas of research in the field of simulation has been at the register transfer level, to wit (3,4,5,6,7,8,9,10,11). Not all of these can be classed as simulators, rather, they should be called analyzers. Both analyzers and simulators however, require the system's structure be defined in terms of registers, memories, interconnecting paths, and combinational networks. Typically, the machine being modeled is a stored program machine. Hence, a timing analysis of the design can be made by executing sample programs on the simulation model of the proposed machine. A common output from such a system, in addition to the analysis and simulation results, is a set of boolean equations describing the system. In a few of these systems, such as (2), a schematic is also generated.

The second active area has been in synthesis at the logic level, to wit (12,13,14,15,16). Quite often, the system is described by a set of combinational logic expressions, i.e., Boolean equations. In some of these systems, the input has leaned more towards the areas of truth tables and

functional arrays. The output then is typically a description of the system implemented in predefined NAND and/or NOR logic. Regardless of the actual implementation however, only a limited amount of actual gate timing can be evaluated. Some of the more recent ones do, however, perform the implementation with elements having limited fan-in and fan-out characteristics, which was not the case in the earlier versions.

Gate level simulation has also attracted a great deal of effort (17,18,19,20,21,22,23,24). It is the author's opinion, however, that all of these for varying reasons and in varying degrees have not resulted in nearer to optimum solutions. Some of the attributes that are desirable are: (1) ease of specifying the system model and simulation controller, (2) flexibility in specifying the system model, such as a good selection of primitives, (3) being able to simulate and not merely analyze the model, (4) have a simulator that executes rapidly, and (5) ease of interpreting the results.

For the most part, all of these systems except (19 and 22) have only combinational elements making up their set of primitives. That is, NOT, OR, AND, NOR, and NAND. This, it would seem, would make the job of modelling much more difficult. In (19) the primitive set also includes JK flip-flop, AC flip-flop, and an 18-bit 4096-word memory element. In (22) the primitive set also includes a majority gate, four-phase logic elements, and a 6-bit 8-word read only memory element. In this area, MACSIM's primitive element set includes NOT, AND, OR, NAND, NOR, RS flip-flop, JK flip-flop, D flip-flop, and an arbitrary m-word by n-bit read/write memory element.

An analyzer, and not a simulator, is incorporated in (17). The analysis section is made up of evaluation, race resolution, and control sections. The simulators in at least (18,19,24) are not event directed simulators. That is, the status of each logic block is reevaluated at every clock interval. There have been some schemes used to speed up this process, however, it still would seem to be the slowest means of simulating. Event directed simulation was first introduced by Ulrich (20). The premise supporting this method is that, for a typical digital network, only about one per cent of the system's blocks are active (i.e., changing state) at any given point in time. To implement this scheme, first a timing wheel is generated which is used for scheduling and controlling all activities. Then, after examining the inputs to a gate, if it is determined that the output will change state, then that output, or event, is placed on the timing wheel at some future time. This time is usually whatever the propagation delay through the gate happens to be. The more recent simulators are of this type. In particular, (22) is such a simulator. MACSIM is also an event directed simulator. However, the actual item that is scheduled on the timing wheel is different, and hence, the over-all action of the simulator is changed considerably. In particular, Ulrich's method is to first determine if an output node will change. If it will, then all of gates in the fan-out list of that node are scheduled to be simulated. If, however, the gate has already been scheduled at that time, then he has to go through a procedure he calls indirect scheduling. In the case of MACSIM, nodes rather than gates are scheduled and no special algorithm is required to handle this case. Either the node's new value is the same as that already

scheduled, which is no problem, or else it is the complement, which can be flagged as a race condition.

Almost all of the simulators investigated are of the table driven variety as contrasted to the compiled variety. MACSIM and (18) operate using compiled code to perform the simulation. Basically, the table driven simulators store the information concerning each gate in a large table with a series of pointers representing their fan-out list, and bit strings representing their input and output states. One additional item is necessary which points to a standard subroutine that represents its logical configuration. It is this subroutine then that gets executed during simulation. This, in effect, means the simulator is operating in the interpretive mode rather than being able to operate on totally compiled code. Of course, interpretive processors in general operate considerably slower, and in this case, more storage is required. For instance, in Ulrich's case, if given a gate with four inputs and one output, then six pointers are necessary for the nodes and macro subroutine, in addition to the at least five bits necessary to store the actual states of the nodes. In addition, before simulating a particular gate, the status of all four input nodes have to be updated. In the case of compiled simulators, storage requirements for all nodes are handled just as for any simple variable in a program. That is, the resulting amount of storage required for any given node is not a function of its fan-out. Also, at the time of simulating each gate, complementing a single bit is all that is required, as opposed to going through a node's fan-out list and performing all of the complementations.

It should be pointed out at this point, however, that compiled simulators are not without their faults also. As in the case of MACSIM, "cascaded" compiling (25) is done, i.e., the network is first compiled into a high level language (PL/1). This in turn must ultimately be compiled into the object language of the host computer. This penalty would be most severe in the case where large networks are compiled often but simulated rarely, a case which should normally not exist. However, these compilers are quite often constructed modularly, as is MACSIM, and hence lend themselves to relatively easy modification and expansion.

As mentioned earlier, one last feature of desirablity is the ease of usage. Many of the systems investigated had languages which would be either difficult to learn and use, or else lacked flexibility. It is felt that these shortcomings have been overcome in MACSIM, since the basic simulation control can be defined by the use of merely two instructions. The total instruction set contains ten instructions, most of which were included to facilitate input/output operations.

SYSTEM OVERVIEW

An overview and brief description of the total system will be given at this time in order to give a better understanding of the functions performed by the component parts which are described in further detail later.

Figure 1 contains the block diagram of MACSIM1, whose function, as stated earlier, is to define the macros, and network and simulation control programs. A cross-reference listing can also be generated in this phase. The solid lines represent the normal flow of operation when all of MACSIM1 is executed. The dashed lines represent options which are available for secondary runs.

Figure 2 contains the block diagram of MACSIM2, whose function is to perform the actual simulation.

Figure 3 contains an abbreviated sample of the key-words required, and the order in which they are organized for running of the MACSIM1 phase.

The operation is as follows. First the input is punched on cards according to Fig. 3. This corresponds to blocks 1,4, and 12 in Fig. 1. The macro descriptions are then input into the macro table generator (block 2) which generates all of the tables that are required for the network compiler. After all of the macros have been defined, the card DMP_MAC (dump macro tables) can be inserted which causes the macro tables to be punched on cards. Hence, any future networks which use these macros can define the macro tables by merely reading in these cards which is much faster than recreating the tables from their descriptions. Also, note that the macro tables can be generated by a combination of both a previously defined table

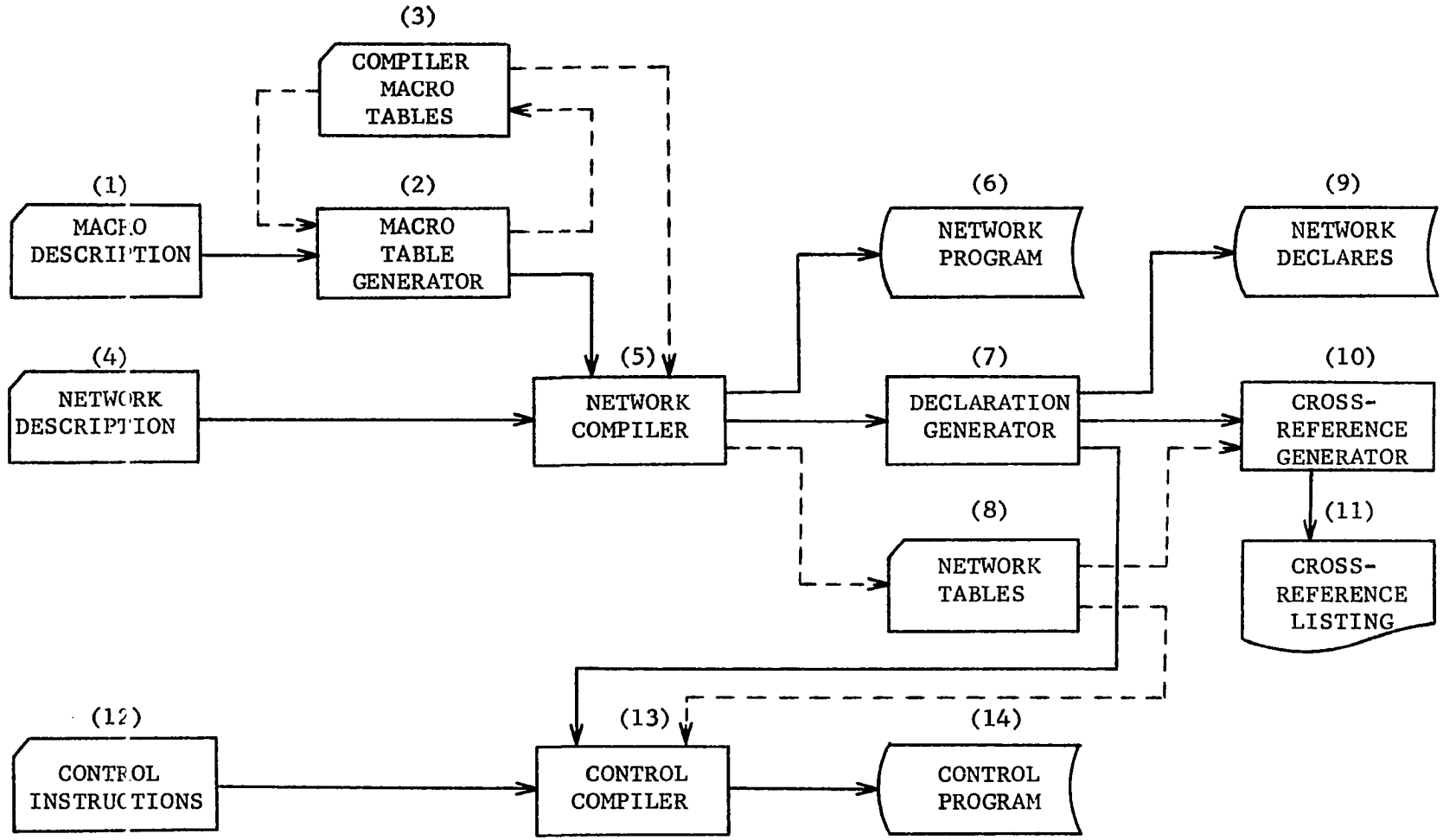


Figure 1. MACSIM1 - macro, network, and control description phase.

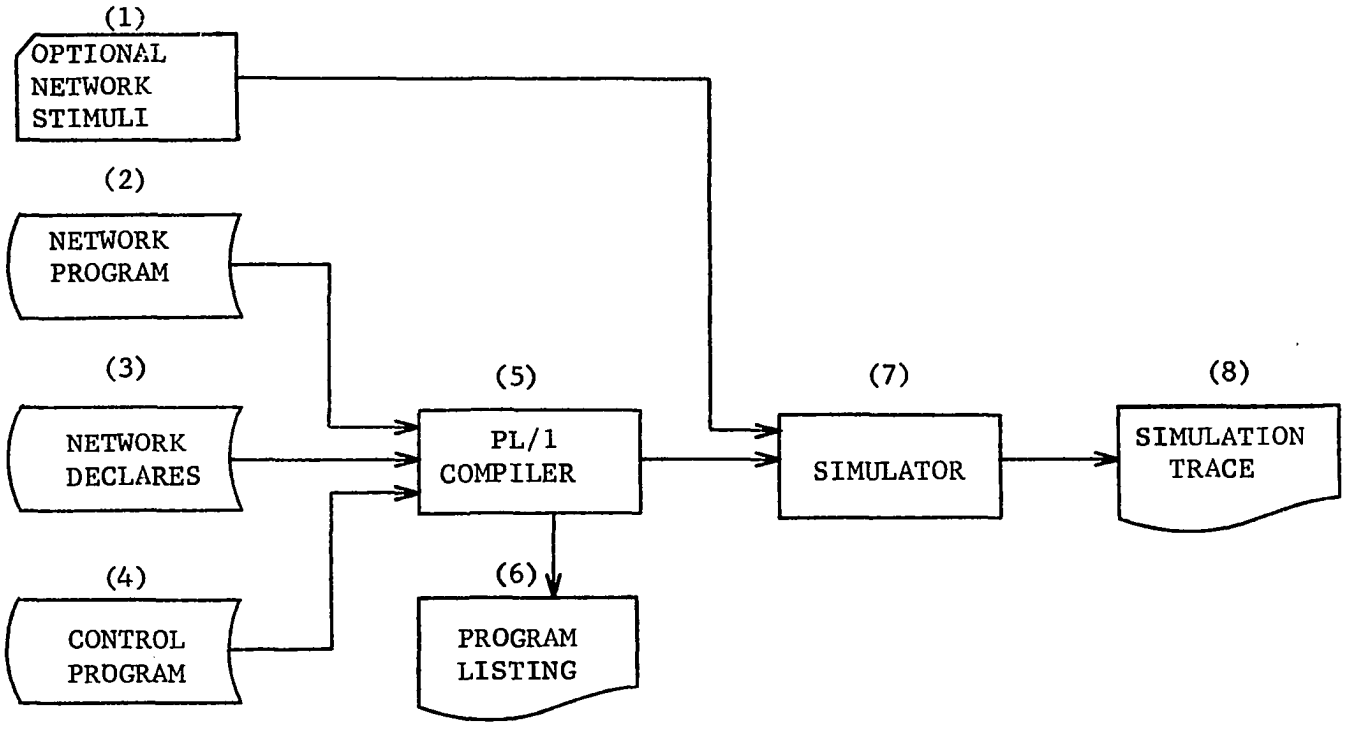


Figure 2. MACSIM2 - Simulation phase.

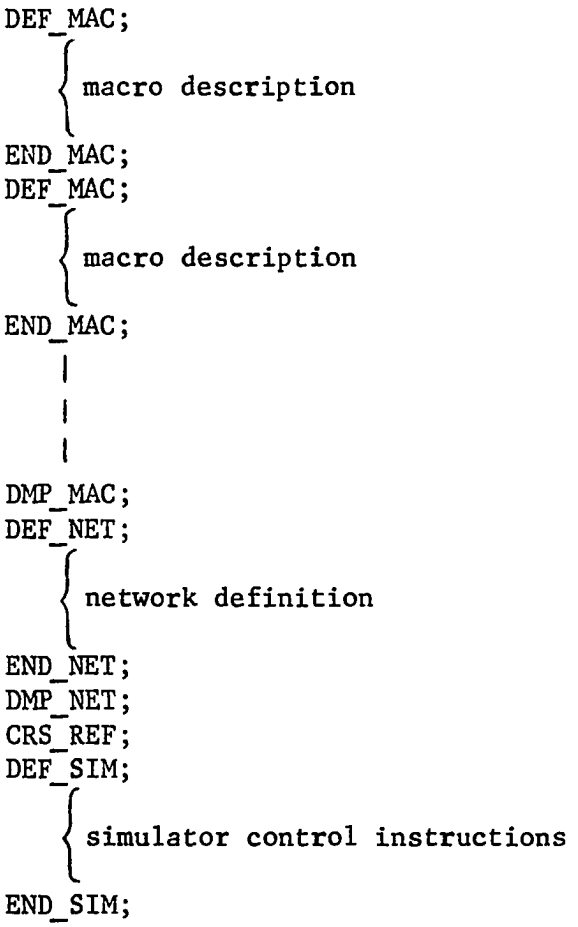


Figure 3. Key-words and structure of input cards to MACSIM1 phase.

and newly defined macros, i.e., the two inputs into block 2.

Once the macro tables have been generated, then the network description (block 4) is input into the network compiler (block 5). The first output of the compiler is a PL/1 program which contains the logical description of the network. This is stored on temporary disk space. The second output is the tables which are required to generate the PL/1 declaration statements for the network and to generate the cross-reference listing. Also at this point, the card DMP_NET (dump network tables) can be included which will cause these tables to be punched on cards (block 8) for later

use in generating a cross-reference listing or for defining a new simulation control program.

The declaration generator (block 7) then outputs onto temporary disk space (block 9), the required PL/1 declare statements for the network (i.e. nodes and memory elements). Several lists are also output which are needed later in the simulation phase.

If the card CRS_REF (cross-reference) was included, then this listing is created and output on the line printer.

At this point, all of the required network information has been created and only the network exerciser remains to be defined. The cards (block 12) are then input into the simulation control language compiler (block 13). The output of this compiler is another PL/1 program which is also placed on temporary disk space (block 14). This completes the generation of everything that is unique to the given network and exerciser.

Assuming no errors were encountered in the preceding steps, the actual simulation can now be performed (Fig. 2). The first step is to compile into a single program the three sections of PL/1 code generated in the first phase. These are blocks 2,3, and 4, which correspond to blocks 6,9, and 14 in Fig. 1. They are input into the standard PL/1 compiler as a procedure which is compiled (block 5) and the source code for the subroutine is listed (block 6). The object code is then link edited with the other object code subroutines to complete the definition of the simulator (block 7).

Finally, the simulator is executed which generates a trace on the line printer (block 8), at the specified time intervals, the 0/1 bit patterns of all the specified nodes.

All of the algorithms required to perform these tasks are constructed from a series of external subroutines written in PL/1. The routines are stored in object code on disk. Because of the large size of MACSIM1 (approximately 240,000 bytes of object code), it has been subdivided into nine parts and an overlaid structure is used. That is, only the routines that are required to perform the current task are brought in from disk and placed in main memory. Only the tables which are common to all the routines remain resident for the duration of the job. This then allows all of MACSIM to run in a region size of 124,000 bytes. Of course, this required region size becomes larger as the network becomes larger.

NETWORK DESCRIPTION

The first step in defining a network is to define all of the building blocks (i.e., macros) that are used in modeling the network. The syntax for defining macros is shown in Table 1.

The notation used here is called a modified Backus Normal form. The symbols ::= are read as "is defined by", the symbol | is read as "or", the brackets [ ] denote "zero or more occurrences of", the brackets { } denote "one or more occurrences of", and the brackets < > are used to enclose non-primitive terms. All other symbols and all terms typed in upper case appear literally in the macro description text. As usual, the term on the left-hand side of ::= is the defined item, and the term on the right-hand side is the defining item(s).

A semantic description of the syntax shown in Table 1 is as follows: First of all, the extent of a macro description is defined to be any description contained within the key words DEF_MAC and END_MAC. The macro description is in effect a word description of its output nodes, input nodes, and its drive and load characteristics. Its general form is as follows: (output node list) (input node list) macro name; The only requirement on the names used to construct the node lists is that they be unique for a given macro. The actual network node names are substituted in place of these names later in the network description when the macro is used. This is analogous to the relationship of parameters and arguments in subroutine calls.

Table 1. Macro definition syntax

<macro> ::= DEF_MAC; <macro desc> END_MAC;

<macro desc> ::= {<defined macro> | <prim macro>}[<comment>]

<comment> ::= [blank] * [<alphameric>]

<defined macro> ::= (<list>) (<list>) <macro name> ; <drive-load desc>

<prim macro> ::= (<list>)(<list>) <prim macro name> ; <timing desc>

<prim macro name> ::= $NOT|$OR|$AND|$NOR|$NAND|
$RSFF|$JKFF|$DFF|$MEM

<list> ::= <identifier> [, <identifier>]

<macro name> ::= <identifier>

<drive-load desc> ::= <drive-load key> = <integer> [, <integer>];

<drive-load key> ::= DRIVE|LOAD

<timing desc> ::= <timing key> = <integer>;

<timing key> ::= DEL_0|DEL_1|MIN_CLK|MIN_CLR|
MIN_SET|MIN_HLD|CYCLE|ACCESS|
MIN_ADR|MIN_INP|MIN_RED|MIN_SEL

<identifier> ::= <letter> [<alphameric>]

<alphameric> ::= $|#|@|_|<letter>|<digit>

<integer> ::= <digit> [<digit>]

<letter> ::= A|B| --- |Y|Z

<digit> ::= 0|1| --- |8|9

The drive characteristics of the macro are defined by the key word DRIVE= followed by a list of integer numbers corresponding to the drive of each output node. A similar description is used to denote the load characteristics of each input node.

The remainder of the description conveys the logical and timing characteristics. The logical properties are defined by referencing macros defined elsewhere (not necessarily previously defined) or by referencing the set of primitive macros. The schematic representations of the primitives are shown in Figure 4. The first five elements are the standard combinational logic gates. The three flip-flop primitives have the standard truth tables. In addition, the JK and D flip-flops have direct-clear and preset inputs. The most complex of the primitives is the memory element, as it can be used to represent any arbitrary m-word by n-bit read/write memory. A more detailed discussion of the rules which must be followed to use these macros is given in the MACSIM User's Manual (Appendix 1).

Only primitive elements can convey timing information. Note the distinction here, the load and drive characteristics are a function of the macro being defined, whereas the timing, i.e., the time required for the output lines to respond to the input lines, is a function of the primitives that make up the macro description. For all but the memory element, the propogation delay, which is the time required for the output to respond to a change in the input, must be specified by DEL_0 and DEL_1 (delay for the output to change to a 0 and 1, respectively). For the JK and D flip-flops, the minimum clock (MIN_CLK) must be specified. In addition,

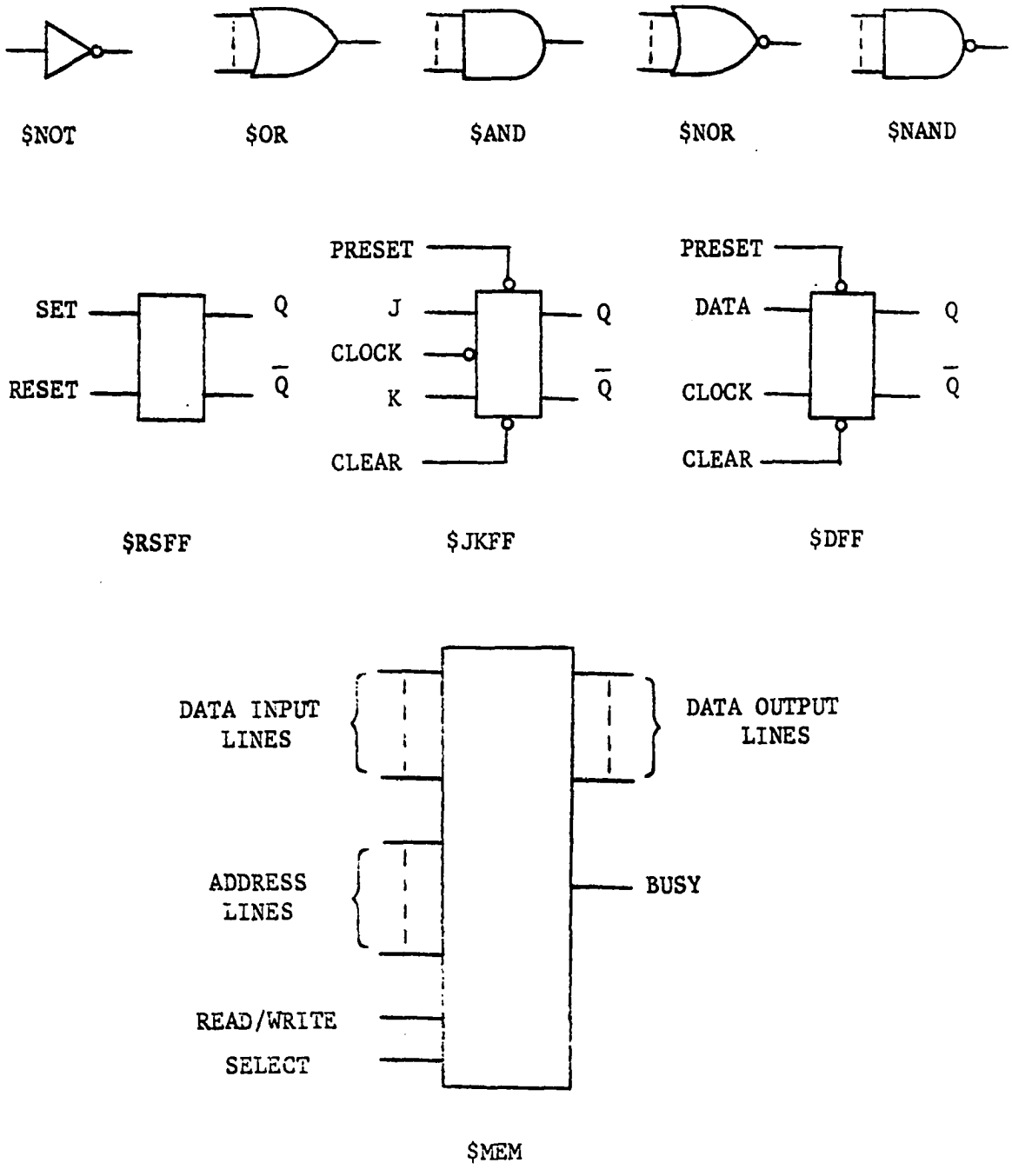


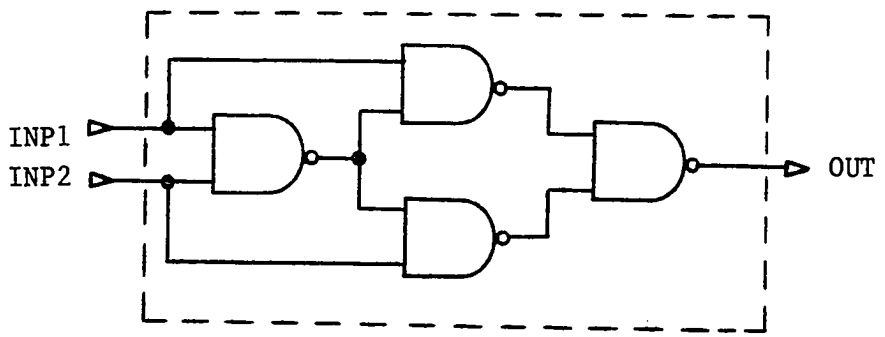
Figure 4. Schematic representation of primitive macros.

for the D flip-flop, the minimum amount of time that the data input must be stable prior to and after the leading edge of the clock pulse must be given (MIN_SET and MIN_HLD, respectively). For the memory element, the minimum times that the address, data input, read/write line, and select line must be held stable is given (MIN_ADR,MIN_INP,MIN_RED,MIN_SEL, respectively). And finally, the cycle and access times are specified. These time values are normalized units of time and can be whatever the user chooses, so long as their relative values are consistent throughout the design. For instance, when using TTL integrated circuits, one unit of time might be used to represent one nanosecond.

For the most part, all names used in MACSIM can be anything that is acceptable to the PL/1 compiler with the exception they must not exceed 16 characters. In addition, Appendix 1 contains the few reserved words that cannot be used.

A very simple-minded example of defining a two-input exclusive - or macro is given in Figure 5. For this example, the circuit (EXCL_OR) is defined using two-input NAND elements (NAND2). These, in turn, are defined using a primitive NAND element. It should be noted here that the order in which the terms are shown as well as their general formats (spaces, position on the page, etc.) is not important since MACSIM's input format is quite flexible. Again, this is discussed further in the User's Manual.

Schematic representation of EXCL_OR



Macro definition of EXCL_OR

```
DEF_MAC;
     (OUT)(INP1,INP2)EXCL_OR;
         DRIVE = 10;
         LOAD  = 2,2;
     (DUM1)(INP1,INP2)NAND2;
     (DUM2)(INP1,DUM1)NAND2;
     (DUM3)(INP2,DUM1)NAND2;
     (OUT)(DUM2,DUM3)NAND2;
END_MAC;
DEF_MAC;
     (OUTPUT)(INPUT1,INPUT2)NAND2;
         DRIVE = 10;
         LOAD  = 1,1;
     (OUTPUT)(INPUT1,INPUT2)$NAND;
         DEL_0 = 8;
         DEL_1 = 12;
END_MAC;
```

Figure 5. Example of a macro definition for a two-input exclusive-or circuit.

Macro Table Generator

Now that the semantics and syntax of the macro descriptions have been given, a more detailed examination of the actual implementation of these algorithms can be discussed.

Two of the external subroutines mentioned previously are loaded from disk to perform the task of generating the macro tables that are used in the network compilation phase. They are DEF_MAC and PARSE. DEF_MAC is called whenever an input card contains "DEF_MAC;". This in turn calls PARSE once for each statement in the macro definition that contains an output list, input list, and macro name.

The PARSE subroutine uses a CONO table to perform the parsing operation. In this scheme, the action that is taken is a function of both the current character being examined as well as the preceding one. The CONO table is shown in Table 2. This routine returns to DEF_MAC both a vector containing the names of the output nodes and the input nodes, as well as a pointer indicating either which primitive macro is being used or else indicating that a non-primitive is being used or defined. PARSE is also called when the network is defined, hence, this pointer indicates whether or not the network is using a previously undefined macro.

The subsequent action taken by DEF_MAC can be best understood by first examining the four tables that it generates which make up the macro description tables. These are represented pictorially in Figure 6 along with a brief description of each variable name's function.

DSTRNG and NSTRNG are vectors, but can be thought of as modified lists. Actual list structures are not used because they require more space

Table 2. CONO table for parsing macro definitions.

previous character \ current character	space	,	(	)	;	X
space	1	1	1	1	10	2
,	1	6	8	6	6	2
(	1	6	7	6	6	2
)	3	8	3	7	9	4
;	10	10	10	10	10	10
X	5	5	8	5	5	1

The actions to be performed are as follows:

1. Increment character pointer; get new card if necessary.

2. Set pointer indicating first character of name.

3. Increment the phase counter (i.e., 1=output list, 2=input list, 3 = macro name).

4. Perform both actions 2 and 3.

5. Get name and place in name list; if phase is 3 then get macro pointer.

6,7,8,9. Error conditions.

10. Finished with scan; return from PARSE.

where, X is any other character which is not specifically mentioned elsewhere.

and more time to chain through. However, the same flexibility is obtained since no space is "wasted" depending upon whether a macro might have two nodes or 20 nodes. After each of the list elements has been entered consecutively, the list is terminated with either a zero or blank entry depending upon which vector is being filled. The sizes of these tables are set to accomodate "modest" size networks (approximately twenty "average" sized macros), however, any of these tables will automatically be expanded by DEF_MAC in the event insufficient space was initially allocated.

The description given in Figure 6 for MAC_NAME should be fairly self-explanatory, however, the structure of MAC_DESC (macro description) warrants furture discussion. As mentioned in Figure 6, the forward pointer (FPTR) of MAC_NAME points to the row in MAC_DESC that contains the first description element. If this is a primitive element, then a number between 1001 and 1009 is entered in the FPTR column of MAC_DESC. If, however, the descriptor macro is not a primitive, then FPTR is set to a value corresponding to the row in MAC_NAME that contains this macro's name. If it has not yet been defined, then the name is entered in the next available row of MAC_NAME and FPTR of MAC_DESC is set up accordingly. All subsequent description macros are similarly loaded consecutively into MAC_DESC all of their backward pointers (BPTR) would point to the same row in MAC_NAME.

One additional function is performed in DEF_MAC. This is the generation of dummy names for all nodes that are internal to the macro. For instance, in the example given in Figure 5, dummy names must be created for the nodes DUM1, DUM2, and DUM3. Each internal node of all the macros must have a unique name and each time a given macro is used in the network,

Figure 6. Macro description tables.

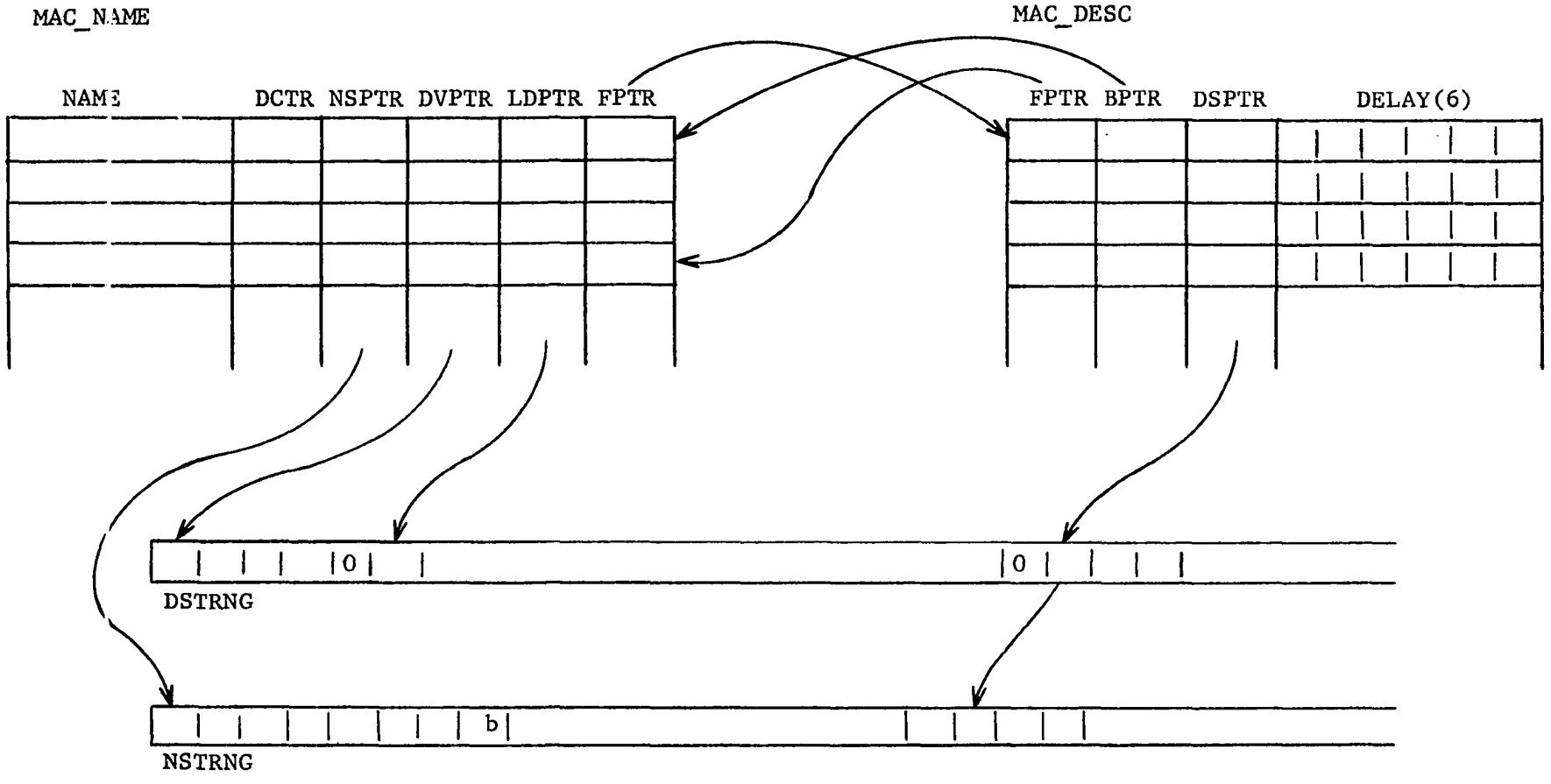
Definition of terms:

MAC_NAME = macro name table
NAME = name of macro
DCTR = dummy name counter
NSPTR = name string pointer, points to first name in NSTRNG
DVPTR = drive pointer, points to first drive value in DSTRNG
LDPTR = load pointer, points to first load value in DSTRNG
FPTR = forward pointer, points to first description in MAC_DESC

MAC_DESC = macro description table
FPTR = forward pointer, if primitive macro then FPTR>1000, else points to MAC_NAME
BPTR = back pointer, points to the macro it is describing in MAC_NAME
DSPTR = points to DSTRNG which in turn points to the macro names in NSTRNG
DELAY = contains up to six timing entries

DSTRNG = contains either drive values, load values, or pointers to NSTRNG, strings are terminated by a zero entry

NSTRNG = contains node names, strings are terminated by a blank entry


MAC_NAME
MAC_DESC
NAME
DCTR NSPTR DVPTR LDPTR FPTR
FPTR BPTR DSPTR
DELAY(6)
0
0
DSTRNG
b
NSTRNG

another unique name must be generated. The technique used to perform this function is as follows. As the tables are filled, each internal node is numbered consecutively $000, $001, etc. and the dummy name counter (DCTR) is set to zero for each macro. Later, in the network definition, when a given macro is used, its DCTR is incremented and concatenated with the dummy name. That is, the first time a macro is called, the names would be $000$000, $001$000, etc.; the second time $000$001, $001$001, etc.; and so on. This is the reason why a user cannot define node names whose first character is a "$", since the uniqueness of names could no longer be assured.

Network Compiler

Once the macros have been defined, the network description can then be input and converted to a logically equivalent PL/1 program. This task is performed by a table driven compiler called DEF_NET. This routine uses the same parsing routine (PARSE) that was described earlier. Hence, the general format for defining the network is the same as that for defining the macros. That is, the network description is as follows:

```
DEF_NET;

      (output list) (input list) macro name;
        |
        |
END_NET;
```

where, the lists are as defined earlier and the macro name is any macro previously defined.

DEF_NET has basically two parts to it. The first is a recursive subroutine (GEN_CODE) which "steps" through the macro tables, and the

second part is a section containing internal routines which generates the appropriate PL/1 code for each of the primitive macros.

As mentioned, GEN_CODE is a recursive routine, which means it can either be called from DEF_NET or it can call itself. The operation is as follows. First a network statement is input, and then parsed to get the two lists into a single list of node names and to determine the pointer which corresponds to the row in MAC_NAME for the macro being used. Then GEN_CODE is called and the appropriate descriptor macros are accessed in MAC_DESC. For a given descriptor, if it is a primitive macro, then the appropriate code generating routine is called and the PL/1 code describing that gate is output. If, however, the descriptor is not a primitive, then GEN_CODE calls itself and the process begins all over at the new row in MAC_NAME. This process continues until the entire macro has been described in the set of primitives and the corresponding PL/1 code generated, at which time a new network statement is input. Eventually END_NET is encountered and control is returned back to the main routine from DEF_NET.

There are basically five routines that generate the PL/1 code. One routine generates the code for all five combinational primitive macros, while the other four routines generate the code for the remaining four primitive macros.

These are four basic parts to each section of code; (1) a label, (2) either the logical description or else the appropriate subroutine call, (3) a call to the event scheduler, and (4) a return to the calling routine statement.

Later on, in the actual simulation, these labels are used to determine which section of code to execute in order to, effectively, excercise a particular gate.

For combinational primitives and RS flip-flops, logical expressions are inserted as in-line code as opposed to a procedure call since this requires no more memory space and will execute much faster. In-line code is also generated for the memory primitive. In this case, more program space is required as opposed to a procedure call, however, it would be extremely difficult, if not impossible, for the system to have a single routine which could handle all types of memory elements that a user might wish to define. If such a routine could be written, it certainly would execute quite slowly. Even the in-line, "tailored" code that is generated is the slowest of all the primitives. This is due primarily to the fact that, for every memory cycle, eventually all of the inputs will need to be checked to insure they have been stable for the appropriate amount of time.

For the JK and D flip-flops, three labels and three subroutine calls are generated for each flip-flop. These correspond to a change in the CLOCK input, a change in the CLEAR input, and a change in the PRESET input. One additional label is generated for the D flip-flop. For this one, a dummy data input node name is generated which is used for determining whether or not the data input was stable for the specified period of time (i.e. MIN_HLD).

The reasons for using subroutine calls in these cases were as follows: (1) all flip-flops of a given type are, by definition, the same, which means writing a standard set of routines is no problem, (2) a

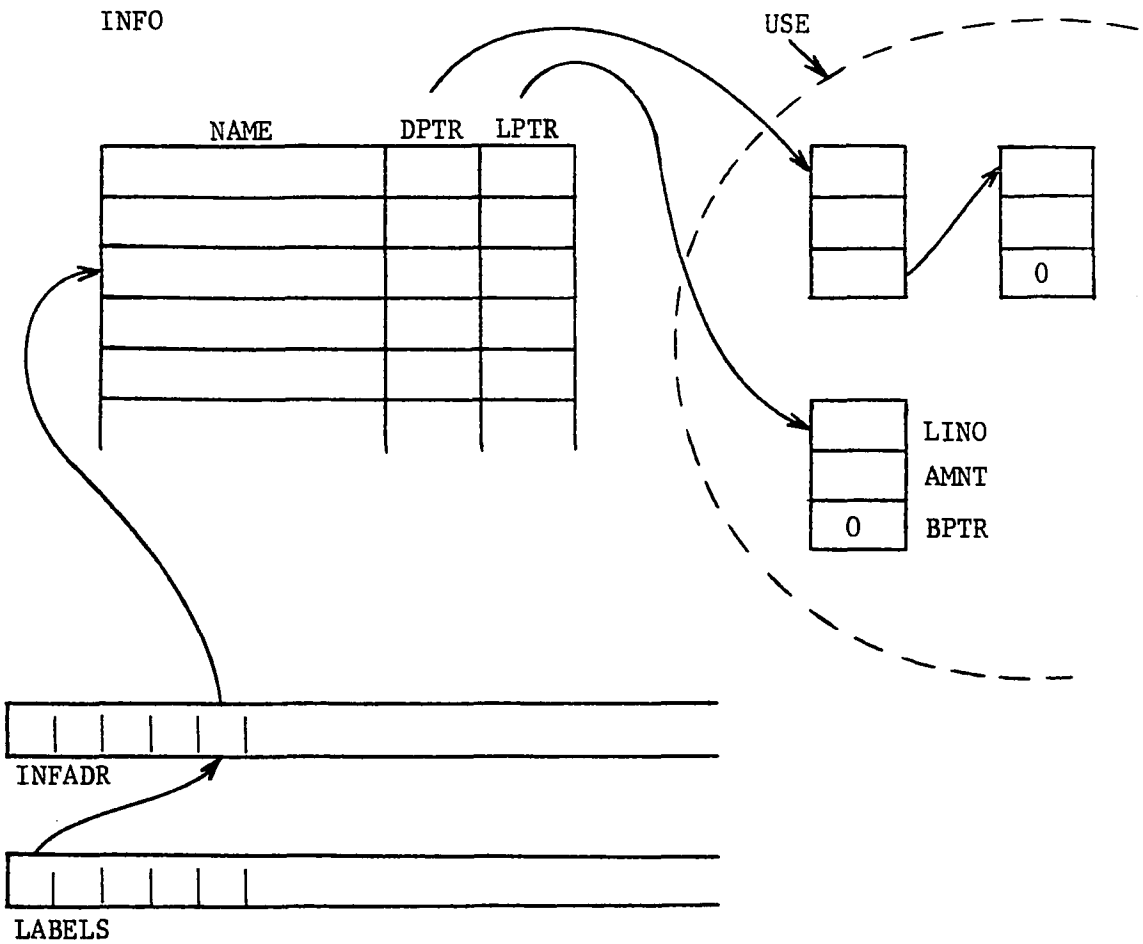
considerable amount of code is required to accurately model these flip-flops, and (3) since a network would typically contain many of these flip-flops (e.g. registers), a considerable savings in over-all program size can be realized.

The third item to be placed in the generated code is a call statement to the event scheduler. This procedure ($CHED) has four arguments. First is the output node name of the gate being simulated, then a bit value representing the new output value after simulation of the gate, and finally DEL_0 and DEL_1, one of which is chosen as the time to schedule the output change.

The last item to output is a simple return statement which returns control back to the event simulator routine ($IMUL8).

As mentioned earlier, the combinational gates and the RS flip-flop each have one label (i.e., entry point) and the JK and D flip-flops each have three and four labels, respectively. The memory element, however, has six labels. They correspond to the actions performed at the beginning of a memory cycle and at the end of the access time. The other four labels are used for checking the stability of the address lines, data input lines, read/write input, and the select line. These last four checks are included, even though they slow up the simulation, because it is felt they are valuable pieces of information to the user.

In addition to generating PL/1 code, the compiler must perform one additional task. That is, the appropriate tables must be generated which can be passed on to the declaration and cross-reference generators. The four tables that are generated are shown in Figure 7.



Definition of terms:

INFO = network information table
NAME = node name
DPTR = drive pointer, points to USE
LPTR = load pointer, points to USE

USE = node usage list table
LINO = line number in source text of usage
AMNT = amount of drive or load
BPTR = brother pointer, points to another USE element

INFADR = INFO table address, is indirect chain connecting LABELS to INFO

LABELS = contains pointers to INFO for each input node at each label.

Figure 7. Network description tables.

The node usage table (USE) is a list structured table which contains, in effect, the fan-in/fan-out information for each node. This information is used only by the cross-reference generator. If a node has one or more destinations then LPTR is set to the next available element in USE. At this point, LINO is set to value corresponding to the line number in the source text where the node was used. Also, the amount of load is entered in AMNT. At this time, BPTR is set to zero. If the node is used again, then BPTR is set to the next available element in USE. The same operation takes place when a node is defined (i.e., DPTR is set up) or multiple-defined. It should be noted, that only user defined network nodes have entries in the USE table. The only function of USE is to provide a cross-reference listing. Any nodes that are internal to macros are given dummy names by the system and therefore will not appear in this listing. This approach was somewhat arbitrarily taken in an effort to "clean up" the cross-reference listing.

The node names are entered in INFO simply by serially searching through the table to see if it has already been entered. If it has not, then it is entered in NAME at the next available space. Admittedly, this is a very slow technique and probably warrants changing to something else, such as a hashing scheme. Actually, this would be quite easy to do, since only the routine called LOOKUP would need to be changed.

In order to understand the information that is stored in LABELS, a description of what is needed by the simulator must first be given. When a node changes state, two separate actions must be performed. First, all of the destination gates whose output could be affected by this change must

be determined. Secondly, the appropriate inputs to these gates must be checked to determine if they are, as of yet, still undefined. These inputs which need checking are tabulated in Table 3. Note in particular for the JKFF, DFF, and MEM macros, the fact that any arbitrary input node might change, does not necessarily mean the associated gate will in fact be simulated. For instance, the output of a JK flip-flop will not change when either the J or K inputs change state and the clock input remains stable, hence there is no need to simulate the flip-flop in such a case.

With these two actions defined, the contents of LABELS can now be explained. At the time when the code for each label is being generated, the following action is taken. First the index value for the node being processed is obtained (this corresponds to the row subscript value indicating its location in INFO). If the action to be performed on that node is defined by both columns (a) and (b) of Table 3, then its index value is placed in LABELS as a positive number. If, however, the action is defined only by column (b), then its index is entered as a negative number. After this has been performed for each of the required nodes at the given label, then a zero is entered to indicate the end of that particular label. For instance, if the code at label L(i+2) is being generated for a JK flip-flop, then all of the gates input node index values would be entered as negative numbers with the exception of the clock input, which would be positive. This seemingly obscure solution was taken in order to contain both types of information in a single table.

If the network definition includes the use of one or more memory macros, then one additional small table is generated which contains the memory names and their respective sizes (i.e., the number of bits per word

Table 3. Gate inputs to be checked by the simulator

macro	label	(a) inputs whose change will effect the output	(b) inputs that must be defined before simulating
COMB	L(i)	all inputs	all inputs
RSFF	L(i)	all inputs	all inputs
JKFF	L(i)	clear input	clear input
	L(i+1)	preset input	preset input
	L(i+2)	clock input	all inputs
DFF	L(i)	clear input	clear input
	L(i+1)	preset input	preset input
	L(i+2)	clock input	all inputs
	L(i+3)	dummy data input	none
MEM	L(i)	select input	all inputs
	L(i+1)	dummy access input	none
	L(i+2)	dummy address input	none
	L(i+3)	dummy data input	none
	L(i+4)	dummy read input	none
	L(i+5)	dummy select input	none

where, COMB = all combinational gates
RSFF = RS flip-flop
JKFF = JK flip-flop
DFF = D flip-flop
MEM = memory element

and the number of words).

It should be noted that these tables, like the macro tables, are automatically expanded in the event insufficient space was initially defined.

After all of the network has been defined, the INFO table is sorted by the NAME entries and placed into alphabetical order before the declaration and cross-reference listing generators are called. This action obviously leaves all of the index values stored in LABELS pointing to the wrong locations in INFO. Therefore, one addition vector (INFADR) is defined which is the same length as INFO and contains the new locations of the node names after the sort is completed. For instance, if node X was initially at row 3 in INFO, and after the sort was placed in say row 15, then the third element of INFADR would contain a 15.

All of the preceding PL/1 code is written on disk in a temporary data set whose file name is NETWORK.

Declaration Generator

The first items declared are the memory elements. For every m-word by n-bit memory that is used in the network, a corresponding bit matrix is declared. That is, the matrix has rows from zero to (m-1) and columns from zero to (n-1).

Secondly, all of the network nodes are declared, including of course, both user names and system defined dummy names. Each node is defined to have three parts: (1) one bit which contains its current binary value, (2) an integer number that indicates the last time the node changed states,

and (3) an integer number which is an index pointer. The first two items are straight forward, however, the third requires some elaboration.

The simulator is constructed of a number of external subroutines, one of which is a subroutine generated by the network compiler. This compiler generates the code using the actual node names that are defined by the user. Hence, in order to perform the simulation, the other routines need to have access to these nodes which could be performed by externally declaring them in each of the routines. There are facilities in PL/1 to do this, however, it would be very time consuming since it would necessitate recompiling all of the simulator routines for every simulation routine.

An alternate solution of overlaid defining of the node names was taken. A vector ($NODE) whose element structures are defined to be the same as those of the nodes is declared in each of the routines, including the generated network routine. Then at the time the network routine is compiled, the node names are declared and overlay defined onto $NODE. What all of this means is that any node can be referenced by using its name (i.e., in the network routine) or by referencing some element of $NODE (i.e., in any of the other routines) without any penalty of added compile time, execution time, or program space requirement.

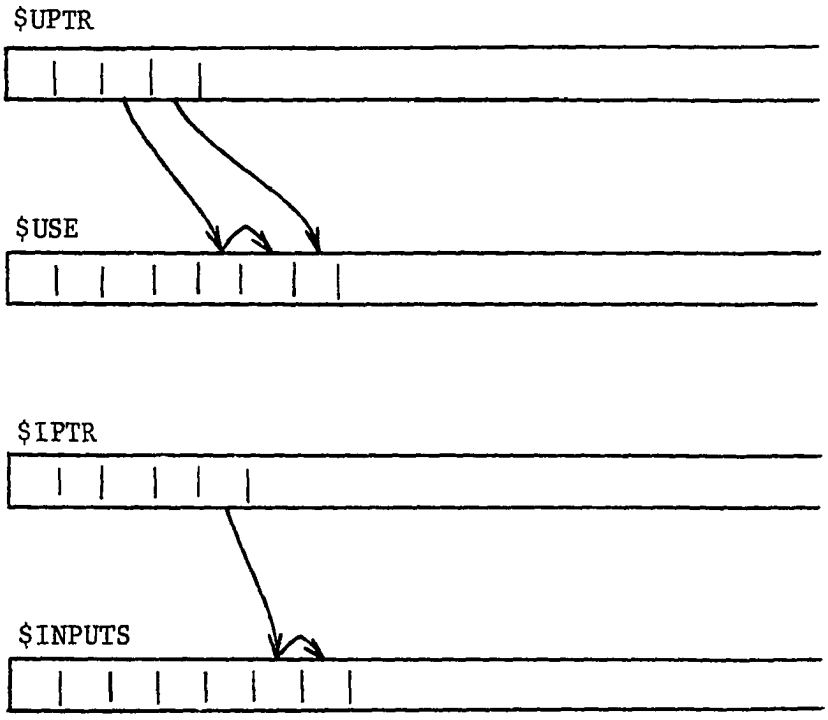
Therefore, a node name's index is given a value corresponding to its location in the INFO table. For instance, if the node named ABC is the third entry in INFO (after the sort is completed), then its index would be set to three. Later on in the actual simulation, it can then be referenced either by its name ABC, or by referencing $NODE(3).

Because of peculiarities in PL/1, the actual implementation of this scheme required the use of based storage which requires that pointers be defined and set up initially at execution time. Therefore, in addition to generating the PL/1 declare statements for the nodes, the declaration generator also outputs the necessary pointer information.

Four vectors are also generated. In effect, they are used to convey the information contained in columns (a) and (b) in Table 3. They are shown pictorially in Figure 8. As noted in the figure, there is one entry in $UPTR for each node in the network. The contents of $USE are index numbers corresponding to the labels in the network program. These two vectors define the function given in column (a) of Table 3. The meaning of the arrows shown in Figure 8 is as follows: if the third node changes states, then go to the labels indicated by the contents of $USE(5) and $USE(6) and perform a simulation.

A similar function is performed by $IPTR and $INPUTS which define column (b) in Table 3. The meaning of the example shown in Figure 8 is: if simulation is to be performed at the fifth label in the network program, then, before simulating, check the two nodes indicated by the contents of $INPUTS(6) and $INPUTS(7) to see if they have been defined. Again, there is one entry in $INPUTS for each label in the network program.

Incidentally, it is possible to have zero entries in both $UPTR and $IPTR. For instance, all network output nodes would have zero entries since they would have no destinations, and hence, no simulation need be performed when they change states.



where, $UPTR contains pointers to $USE, there is one entry for each node name.

$USE contains label values.

$IPTR contains pointers to $INPUTS, there is one entry for each label.

$INPUTS contains node index values.

Figure 8. Simulation control vectors generated by the declaration generator.

All of the information needed to construct these four vectors is derived from the previously defined tables LABELS and INFADR.

All of the preceding declaration code is placed on disk in a temporary data set whose file name is DECLARE.

Cross-Reference Generator

The actual algorithms used to generate the cross-reference listing is of little interest or concern to this thesis. Rather, the listing itself is all that will be discussed here.

It is felt that the main attribute of MACSIM's cross-reference listing is the fact that all of the information regarding each node is presented in a single listing. This is often not the case in other systems of this sort.

The listing has the following form. First of all, it is divided into three columns across the page. The left-hand column contains the node name preceded by a line number which is its reference number within the listing.

The center column contains all of the information concerning the node's definition. That is, the line number back in the network source listing on which it was defined as a gate output node, the amount of drive specified for its gate type, the amount of remaining drive after subtracting all of its destination loads, and finally, all of the node names that go into the defining or generation of this node. These names are also preceded by their respective listing index numbers. In the event the node was multiple-defined, this same type of information is listed

pertaining to the secondary definitions. Of course, if a node is used but never defined, then only the negative amount of remaining drive is listed which corresponds to the amount of loading of the node. This is the case for network input nodes.

The right-hand column contains all of the usage information concernint the node. That is, for a given destination, the amount of load is listed along with all of the output node names of that gate which are generated at this point. Again, each of these names is preceded by its listing index number. And finally, after all of the destinations have been listed, then the total net load of the node is listed. In the event the node is never used, such as a network output node, then the only entry in this column is a zero value under net load.

SIMULATOR DEFINITION

Once the PL/1 program has been generated which defines the network and after the appropriate declaration statements have been generated, the remaining task is to generate one more PL/1 program that will control the action of the simulator. Some of these functions are to apply new stimuli to the network input nodes, load one or more words in the memory elements, and supply a trace of the activity of the desired nodes in the network.

Before describing the syntax of the control language and its associated compiler, a description of the control instructions, shown in Table 4, will first be given.

Control Language Semantics and Syntax

All of the instructions in Table 4 have been shown in their complete form. That is, all of the instructions have degenerate forms. In particular, all of the IF clauses in the first nine instructions are optional. That is, when an instruction is written in the form shown, then the specified action is taken only when the condition is satisfied. However, when the IF clause is omitted, the action is unconditionally taken.

The following is a semantic description of each of these ten instructions.

1. Define check: check all input nodes to the gate before performing the simulation on it.
2. No define check: do not check for undefined input nodes.
3. Terminate the simulation.

Table 4. List of simulation control instructions.

1.	DEF_CHECK	IF (condition);
2.	NODEF_CHECK	IF (condition);
3.	STOP	IF (condition);
4.	GOTO (label)	IF (condition);
5.	COMMENT (string)	IF (condition);
6.	HEADING (x_1, x_2, ---, x_n)	IF (condition);
7.	TRACE (x_1, x_2, ---, x_n)	IF (condition);
8.	READ (x_1, x_2, ---, x_n)	IF (condition);
9.	LOAD ($mem_1(i)$, $mem_2(i)$, ---)	IF ($i <= M$);
10.	x_1, x_2, ---, x_n = v_1, v_2, ---, v_m (t_1, t_2, ---, t_k);	

where, M = integer number
x_i = node name
v_i = binary value
t_i = integer number indicating time
$mem_j(i)$ = i-th word of memory mem_j
label = any label name
string = any string of characters
condition = any logical condition which returns a boolean true or false value

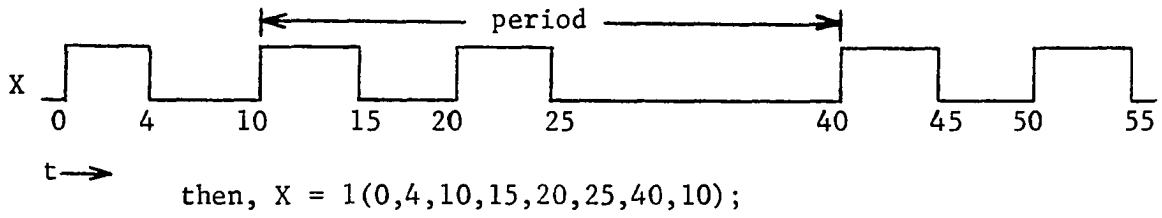
NOTE: any of the above statements can be preceded by a label name.

4. Transfer control to the specified label within the control program.
5. Print the comment in the output trace denoted by the specified string.
6. Print the specified node names in the heading at the top of each page of output trace.
7. Print the binary values of the specified node names. These node names must be contained within the list of node names specified in the heading statement. Not all of the preceding names need be present, however, they must be in the same order. If no node name list is specified, then all of nodes specified in the heading are traced.
8. Read the binary values from punched cards and assign them to the specified nodes. All of the binary values (0 and 1) corresponding to the execution of each read statement must be punched in consecutive card columns beginning in the first card column.
9. Perform, in effect, the same function as READ. However, is used to facilitate the loading of memory elements only, where the bit strings for each word are punched on a separate card. In the form shown, all of the first 0 through M words of mem_1 would be loaded, and then similarly for any other memory elements specified. In the case of this instruction, if no IF clause is present, then the subscript i must be an integer number, or else no subscript need be specified at all. In the first case, the specified word in memory would be loaded, whereas in the second case, the entire memory would be loaded.

10. Assignment statement: the elements in the node list are assigned the corresponding values in the value list at time t_1. In the event $m < n$, then the last (n-m) nodes are set to the value of v_m. At time t_2, etc., the value list is complemented and the assignments made again. Time t = 0 is assumed if no time list is specified. A periodic signal can also be defined (i.e., a clock signal) by specifying

$$t_k = t_{k-1} - \text{period}.$$

An example is shown below. This example is not intended to be a practical or useful one, rather it is chosen to emphasize the power of the assignment statement.



then, X = 1(0,4,10,15,20,25,40,10);

With this general discussion completed regarding the overall characteristics of the control language, a more formal syntax definition can be given. This syntax definition is shown in Table 5. The same modified Backus Normal form is used here as in the definition of the macro syntax.

It should be noted that the control program, like the rest of the MACSIM specification, can contain comments at any point so long as the comments first non-blank character is an asterisk (*). Actually, comments can also be placed following the semicolon (;) since the appearance of a semicolon, with two exceptions, terminates the scan (parse) of a statement. The exceptions are if they are contained within a comment statement or within the string specification of a COMMENT instruction.

Table 5. Syntax definition of control language.

```
<control program> ::= DEF_SIM; {<control statement> }
                      [<comment statement>] END_SIM;

<comment statement> ::= [blank] * [<string>]

<control statement> ::= [<label>:] <assignment statement>;|
                        [<label>:] <operation clause> [<if clause>];

<assignment statement> ::= <node list> = <value list> [( <time list> )]

<if clause> ::= IF ( <condition> )

<operation clause> ::= DEF_CHECK|NODEF_CHECK|STOP|<go to>|
                       <comment>|<heading>|<trace>|<read>|<load>

<go to> ::= GOTO ( <label> )

<comment> ::= COMMENT ( <string> )

<heading> ::= HEADING ( <node list> )

<trace> ::= TRACE [( <node list> )]

<read> ::= READ ( <node list> )

<load> ::= LOAD ( <memory list> )

<node list> ::= <name> [, <name>]

<value list> ::= <binary digit> [, <binary digit>]

<time list> ::= <integer> [, <integer>]

<memory list> ::= <name> [( <subscript> )] [, <name> [( <subscript> )]]

<condition> ::= <string>

<label> ::= <name>

<subscript> ::= <integer>|<name>

<name> ::= <alpha> [<alphameric>]

<string> ::= {<alphameric>}

<integer> ::= {<digit>}

<alpha> ::= A|B --- |Y|Z

<digit> ::= 0|1 --- |8|9

<binary digit> ::= 0|1
```

One further comment should be made regarding the syntax definition. Note that the definition of a condition is simply that it be a string of characters. This is the way it is handled in the control language compiler, even though such a description does not necessarily create a valid conditional statement. That is, it has been left as a responsibility of the user to supply a valid conditional statement. Such a statement is, by definition, any clause which, upon execution, will return a Boolean true or false value. In this case, any valid PL/1 conditional statement is legitimate, since the string is inserted literally into the PL/1 code being generated. The following are some examples of IF clauses.

1. IF (TIME < = 100)
2. IF (¬STABLE)
3. IF (MOD(TIME,10) = 0)
4. IF (A & B)

where, TIME = reserved word indicating current time

STABLE = reserved word indicating the entire network is presently in a stable condition (i.e., no events are scheduled)

MOD = built-in PL/1 modulo arithmetic function

A,B = network node names

The interpretation of these examples is as follows:

1. if current time is less than or equal to 100
2. if the network is not stable
3. if current time, modulo 10, equals 0
4. if nodes A and B are both true

With the semantics and syntax defined, the overall form of a MACSIM control program warrants discussion. The reader who is familiar with

programming languages will notice that the general form of the control language bears a strong resemblance to the forms of other languages. This, of course, was intentional. The use of labels and go-to statements, however, implies more system action than in the conventional sense.

To understand this, a more detailed examination of the function that needs to be performed during simulation must be given. It seems to this author, that in order to exercise a network, the user would first like to apply some set of stimuli, and then simulate until some predetermined condition results. A flow-chart of this operation is shown in Figure 9. Therefore, the total control program could be constructed from a series of these loops. This is not unlike the DO-END and BEGIN-END blocks of other languages. Notice, however, that the calling of the simulator and the incrementing of time operations are not included in the syntax definition. This was done for two reasons. First, the general form of the control program just discussed is assumed, because it is believed to be a valid and usable assumption. Secondly, given this form and since the user would always want these two functions performed, then the control language compiler can automatically insert these functions into the generated code, thereby relieving the user of having to repeatedly specify these two operations.

Since the reserved word TIME is automatically incremented by the system for the user, then it follows that it must also be initialized automatically. It was felt that TIME would be more useful to a user if it was reinitialized each time before entering a new loop group. That is, TIME can be thought of as a loop counter, and hence, the user need not

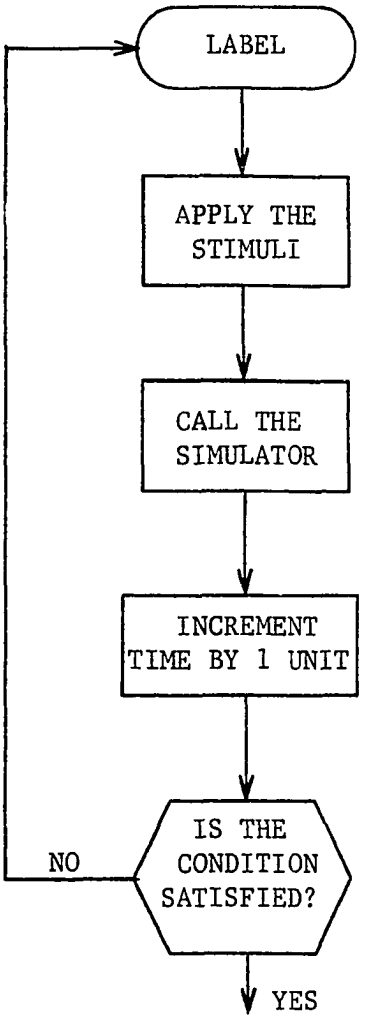


Figure 9. Action of the simulation control program.

have any knowledge of the total elapsed time for specifying an action to occur at the time he is specifying the instructions. In other words, the scope of the reserved word TIME and all specified actions is defined to be local or internal to a given loop. This, of course, has a disadvantage when the user wishes to have a particular action with a scope that is global to the entire control program. An example of this would be if a synchronous network is defined where it would be convenient to have a periodic clock signal defined once for the entire program. As presently

implemented however, the one assignment statement required for defining periodic signals would have to be included within each loop. This is not an optimum solution in this case, however, it was felt that the other conveniences outweighed this particular disadvantage. Of course, if it were deemed necessary, a scheme for defining the scope of actions could be implemented, even though it is not presently felt to be necessary.

Control Language Compiler

The control language compiler, like the other routines described in MACSIM1, is an external PL/1 subroutine. The appearance of a DEF_SIM card in the input stream causes it to be loaded into core from disk and executed. The appearance of END_SIM causes its termination.

The operation of the compiler (DEF_SIM) is as follows. Each time a new instruction is read from punched cards, the routine called PARSE is called, which is a different routine than the one discussed previously. First it scans down the instruction and obtains the first word. If this word is a label, then it saves it and continues the scan to the next word. This word is used to determine the type of instruction being decoded. If it is a reserved word, then the appropriate number is set up (i.e., 1 through 9). If it is not a reserved word, then the assignment statement (10) is assumed. At this point, any of three routines are called. One is IFCHECK, which scans the remainder of the instruction looking for an IF clause and saving it if one is found. A second is GETLIST, which scans the instruction and obtains the elements of either a node list, value list, time list, or memory list. The third routine is GETSTNG, which scans the instruction and obtains either the label of a GOTO statement

or the string argument of a COMMENT statement. After any or all of these routines have been executed, control is returned to DEF_SIM and a new instruction is entered.

In IFCHECK, once the reserved word IF has been found, the scan continues and saves all of the input string beginning at the first left parenthesis through its matching right parenthesis. Similarly, in GETSTNG, the scan saves all of the input string between the matching left and right parentheses.

The routine GETLIST, however, is considerably more complex due to the varying types of lists it has to decode. A CONO table similar to the one discussed previously is used to perform this task. This is shown in Table 6. The details of this algorithm will not be explained, however, because as can be seen in the table, there are 18 different actions that are performed depending upon the relationship of the last and current characters being scanned.

There are basically five different types of lists that can be parsed by GETLIST. They are given below along with the sources of these lists. It is assumed that X, Y, and Z are network names.

1. X, Y, Z)

   node list for instructions 6, 7, 8, or 9

   time list for instruction 10

2. X, Y, Z =

   node list for instruction 10

3. X, Y, Z(

   value list for instruction 10

Table 6. CONO table for the GETLIST routine.

last character \ current character	space	,	(	)	=	;	X
space	1	14	10	8	6	12	2
,	1	ERR 1	ERR 2	ERR 1	ERR 1	ERR 1	3
)	15	ERR 1	ERR 2	ERR 1	ERR 2	ERR 2	15
(	1	16	ERR 2	8	ERR 2	ERR 2	ERR 2
=	ERR 2	ERR 2	ERR 2	ERR 2	ERR 2	ERR 2	ERR 2
;	ERR 2	ERR 2	ERR 2	ERR 2	ERR 2	ERR 2	ERR 2
X	4	13	9	7	5	11	1

where, ERR1 and ERR2 are error conditions, 1 through 16 are different types of actions to be performed, and X is any other character which is not specifically mentioned elsewhere.

4. X, Y, Z;

   value list for instruction 10

5. X(i), Y(i), Z(i))

   node list for instruction 9

Basically then, the appropriate type number is first set, and then GETLIST is called. The actions taken by the CONO table are then a function of the type of list being decoded. Depending upon this type number, the decoded list elements are placed into one of three vectors called NODLIST, VALLIST, or TIMLIST.

At this point, the parsing of the instruction is complete and all of the instruction elements have been placed in the appropriate save areas (i.e. ALABEL, NODLIST, VALLIST, TIMLIST, and IFCLAUS). The only remaining function to be performed is to create the PL/1 code for the instruction being processed. This is done by calling the appropriate internal procedure. There are nine, rather than ten, of these since the code for instructions 1 and 2 can be generated by the same routine. The actual contents of these routines as well as the code they generate is of little interest here, and hence will not be discussed in any further detail. However, for the reader who is interested in the actual PL/1 code generated by both the network and control language compilers should refer to Appendix 3 where a MACSIM sample run is given. A listing of the subroutine containing these two sections of code has been included.

All of the PL/1 code generated by DEF_SIM is placed on disk in a temporary data set whose file name is CONTROL. This data set now contains three files, i.e.,NETWORK, DECLARE, and finally CONTROL.

SIMULATION

As mentioned earlier, the simulator, MACSIM2, is defined by a set of external PL/1 subroutines. Functionally, they can be divided into three groups. These are listed below along with their subroutine names which will be referred to in the discussion of the simulator that follows.

1. Main driving routines:

   $PROGRM - control program

   $NETWRK - network program

   $CHED - event scheduler

   $IMUL8 - event simulator

2. Common network routines:

   $CLEAR - performs the clear function for JK and D flip-flops

   $PRESET - performs the preset function for JK and D flip-flops

   $JKFF - performs the logical switching function for the JK flip-flop

   $DFF - performs the logical switching function for the D flip-flop

3. Assorted output routines:

   $TRACE - performs the output trace

   $HEAD1 - sets up the headings to be printed

   $HEAD2 - prints the headings at the top of each trace page

   $COMNT - prints comments in the trace output

   $ERROR - prints all of the various types of errors that might arise during simulation.

All of these routines, except $PROGRM and $NETWRK, reside in object code form on disk. Therefore, for a given simulation run, only the two

routines have to be compiled and then link edited with the others.

Actually, $NETWRK is not a separate routine, but rather is an entry point within $PROGRM. The reason for this is because they both need to have access to the declarations which were generated earlier and it would not make sense to have to insert them twice, which is what would be necessary if separate routines existed. It is possible, however, that during simulation, $PROGRM will need to call $NETWRK. This problem was solved by simply defining %PROGRM to be a recursive subroutine which, in this case, means it can be reactivated even though it may currently be active.

To combine the three files which were previously generated (i.e., NETWORK, DECLARE, and CONTROL) into the single routine called $PROGRM, the PL/1 compile time facility known as preprocessing was used. This is defined to be the process in which the PL/1 source code can be modified prior to compilation.

The routine called $PROGRM, when its original source code is loaded from disk into core, prior to the preprocessing stage, has the following form.

```
$PROGRM: procedure recursive;
        { external (i.e., common) declarations
    %INCLUDE SYSLIB(DECLARE);
    %INCLUDE SYSLIB(CONTROL);

    $NETWRK: entry;

    %INCLUDE SYSLIB(NETWORK);

end $PROGRM;
```

where, SYSLIB is simply the name of the data set containing the three files DECLARE, CONTROL, and NETWORK. The preprocessor instruction %INCLUDE

performs the function of fetching the PL/1 code from the three files and literally inserting it in the appropriate places in the already existing source code of $PROGRM. The combined source code is then compiled and link edited with the object code for the remainder of MACSIM2.

The flow of control during simulation has been pictorially shown in Figure 10. The numbers on the lines connecting the routines show the order in which the routines are called, assuming there is currently activity in the network. First, if new stimuli are to be applied to the network, as indicated by the control program, then the event scheduler ($CHED) is called. After all new events have been scheduled, then the simulation routine ($IMUL8) is called. If events are scheduled on the clock at the current time (i.e., nodes are changing states now) then the network routine ($NETWRK) is called. The appropriate label is branched to and the gate is simulated. $CHED is again called which determines if the simulation has caused the gate's output to change. If it has, then the node is scheduled on the timing clock at the appropriate time slot in the future. That is, at the current time plus the propogation delay through the element. In the event the output node will not change states, control is simply returned back via paths 6 and 7. If a JK or D flip-flop is to be simulated, then the paths marked 5.1, 5.2, 6.1, and 6.2 would be followed, rather than paths 5 and 6.

If more events are scheduled to take place at the current time, then steps 4 through 7 are repeated. Finally control returns back to $PROGRM. If a trace is to be performed at this time, then $TRACE is called (paths 9 and 10). This completes the activities to be performed, therefore, time is incremented and the process is repeated.

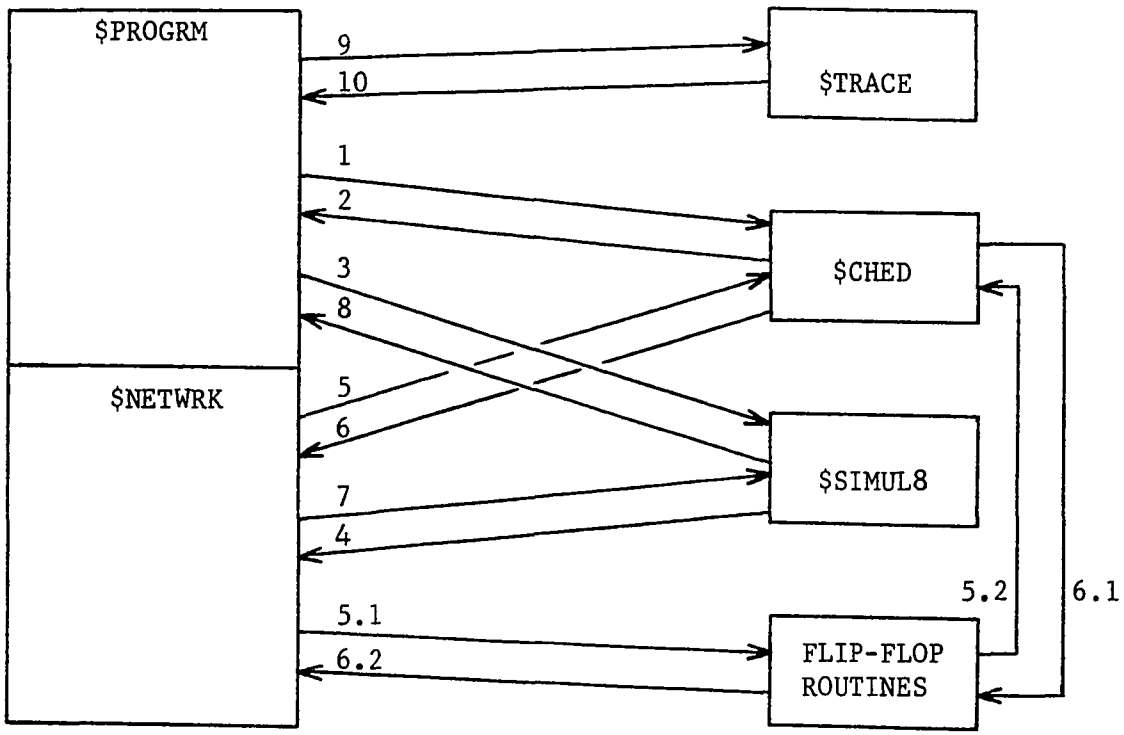


Figure 10. Flow of control during simulation.

As stated earlier, however, during most of the simulating time there is no activity in the network. This means that most of the operations are performed by traversing paths 3 and 8, and possibly 9 and 10. That is, typically no new events are to be scheduled in the control program, so $IMUL8 is called. Then it is determined that no events are scheduled at the present time, so control returns to $PROGRM and time is incremented, and so forth.

The output routines will not be examined in detail since they are relatively straightforward. However, the error messages that are printed by $ERROR have been shown in Table 7 so as to give a better feel for the

Table 7. List of simulation error messages.

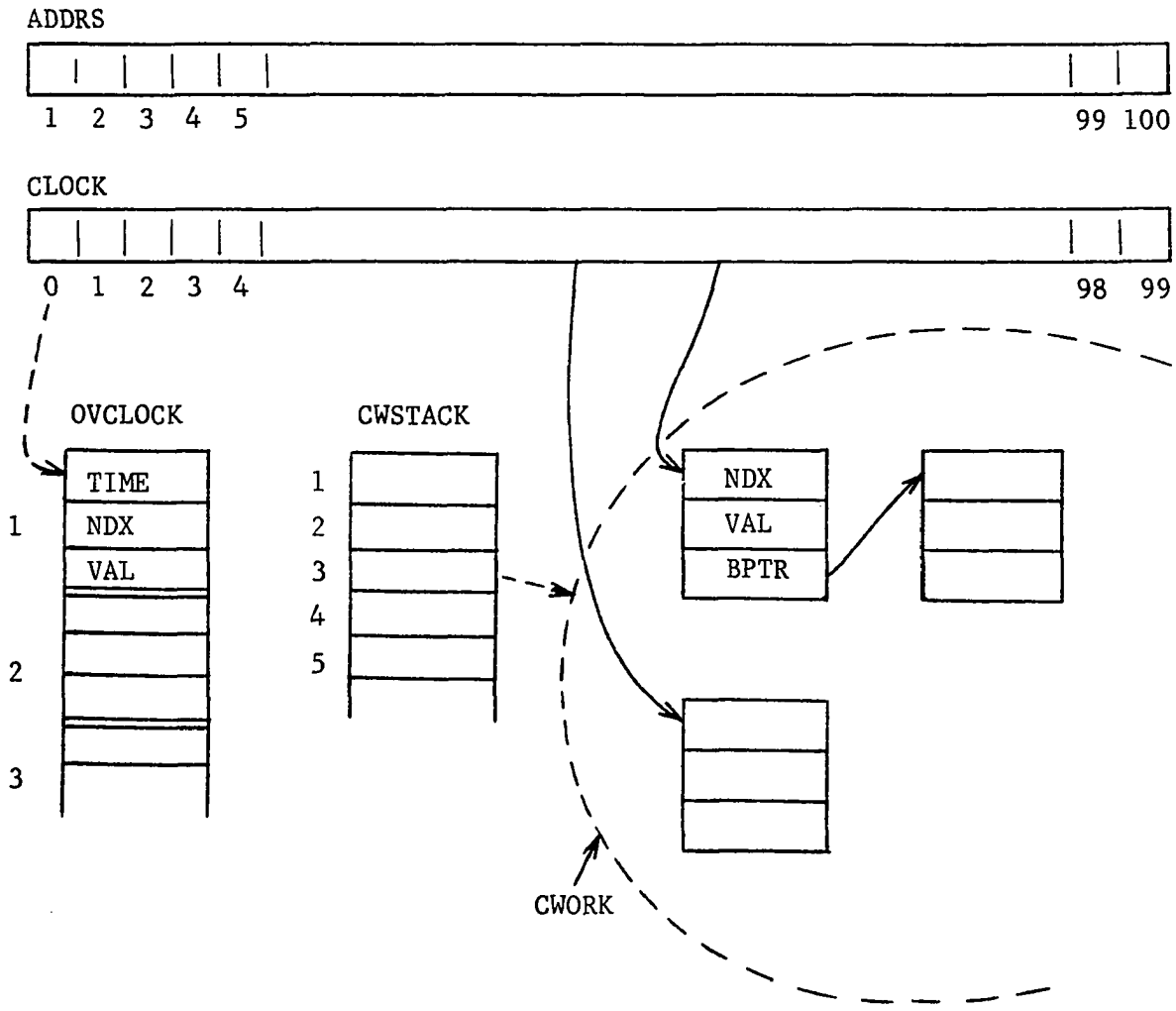
1.	NODE name IS BEING DRIVEN TO CONFLICTING STATES AT TIME = time
2.	NODE name IS BEING USED BUT IS UNDEFINED, IS ASSUMED 0
3.	BOTH SET AND RESET INPUTS ARE HIGH INTO RSFF name
4.	CLEAR INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
5.	PRESET INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
6.	CLOCK INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
7.	J INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
8.	K INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
9.	Q-SIDE INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
10.	QBAR-SIDE INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
11.	DATA INPUT INTO FLIP-FLOP name NOT STABLE LONG ENOUGH PRIOR TO CLOCK
12.	DATA INPUT INTO FLIP-FLOP name NOT STABLE LONG ENOUGH AFTER CLOCK
13.	CLOCKING OF FLIP-FLOP name ATTEMPTED DURING CLEAR OR PRESET OPERATION
14.	ACCESS TO MEMORY name ATTEMPTED WHILE STILL BUSY
15.	ADDRESS LINE(S) NOT STABLE ON INPUT TO MEMORY name
16.	DATA-IN LINE(S) NOT STABLE ON INPUT TO MEMORY name
17.	READ LINE(S) NOT STABLE ON INPUT TO MEMORY name
18.	SELECT LINE(S) NOT STABLE ON INPUT TO MEMORY name

types of conditions that are checked during simulation. These messages can be thought of as warning, rather than error, messages since simulation is never terminated by the raising of any of these conditions.

The only routines remaining to be discussed are the event scheduler ($CHED) and the simulate routine ($IMUL8). Actually, these two routines form the nucleus of the simulator. The tables which are common to these two routines are shown in Figure 11, along with a brief description of each table's function. Almost all of the elements in these tables require half-word (2 bytes) integer storage. The exceptions are ADDRS which is a 100 position character string, and VAL in OVCLOCK and CWORK which is one bit.

As can be seen, CLOCK is a vector containing positions 0 through 99. This clock size was chosen as a compromise between a very large one which would waste a considerable amount of storage, and a smaller one which would require that a lot of the events to be scheduled would first have to be scheduled on the overflow clock. That is, the simulation runs faster when the clock size is larger than the largest propogation delay of any element in the network. Of course, this can not always be achieved, particularly when memory elements are included where the access and cycle times are typically much greater than the gate delays. During simulation then, all elements of CLOCK are set to zero except for the ones corresponding to the time when an event is scheduled to occur. At these locations, a pointer is entered which points to some position within the clock work area (CWORK).

The actual nature of ADDRS is peculiar to the fact that the implementation is in PL/1. There is a built-in function in PL/1 called INDEX which takes the form,



where, CLOCK = event clock
ADDRS = indicates the location of where events are scheduled on CLOCK
OVCLOCK = overflow clock
CWORK = clock working area
CWSTACK = stack containing addresses of unused areas in CWORK

Figure 11. Common tables used for scheduling and simulating events.

result = INDEX(string, configuration);

where, both of its arguments can be character strings, and the result is an integer number. When the INDEX function is executed, the specified configuration is looked for within the string. The resultant value then is the index number corresponding to the first occurrance of the configuration. If the configuration is not found, a zero value is returned.

ADDRS is then set up in the following manner. All of its positions contain the character 0 everywhere except the positions corresponding to the times in CLOCK where events are scheduled. These positions contain the character 1. In addition, the variable EMPTY is set to a binary value of 1 whenever OVCLOCK is empty. Therefore, it can be determined whether or not the entire network is in a stable condition simply by executing one statement. That is,

result = INDEX(ADDR,'1').

Hence, if the resultant value is zero and EMPTY is true, then the network is stable. Naturally, this executes far faster than serially searching the clock and overflow clock to determine if any events are scheduled.

Since it is not possible to predict beforehand what the peak network activity is going to be, the clock work area (CWORK) is a list structure, rather than scheduling events directly on the clock itself. The list structure also lends itself to conveniently being able to add and delete events. Once an element in CWORK has been simulated however, the area needs to be released so that it can be used later. To implement this, a push-down stack (CWSTACK) was incorporated which contains all of the available addresses (i.e., subscripts) in CWORK. That is, the top of the stack

always contains the address of the next available CWORK area. Then after an event has been simulated, the stack is effectively pushed down and the address of the just released area is placed at the top of the stack. In the event the level of activity becomes too high, i.e., more work areas are needed than what were initially allocated, then CWORK and CWSTACK are automatically increased in size.

Notice that each element of CWORK contains three items. First NDX, which is the index value of the node that is scheduled. It will be recalled that each node can be referenced either by its name or else by its index value. Secondly VAL, which is one bit indicating the new value that will be assigned to the node. Thirdly BPTR (brother pointer), which points to another location in CWORK containing a node that is scheduled to change at the same time. BPTR is set to zero to indicate the tail of the list.

The last table, the overflow clock (OVCLOCK), contains any events whose delay is greater than 100. A simple serial table has been used here, as opposed to setting up cascaded timing wheels (similar to CLOCK) which was proposed by Ulrich (20). The reason for this is that, typically, there is virtually no activity in OVCLOCK and hence does not warrant the added table and program space that would be required in the cascaded approach. In fact, as implemented here, the storage for OVCLOCK is never allocated unless a specific request by the scheduler is made. On the other hand, if for some reason a large amount of activity exists in OVCLOCK, additional storage is allocated when the original table size is exceeded.

Each element of OVCLOCK contains three elements. The first is TIME, which indicates the time that the event is to take place, relative to CLOCK(0). The other two, NDX and VAL, have the same meanings as those given for CWORK.

The dashed line between CLOCK(0) and OVCLOCK is shown to indicate that the status of OVCLOCK is only checked when the CLOCK passes through 0. As mentioned earlier, if OVCLOCK is empty, then EMPTY is set to a binary 1 which means OVCLOCK does not have to be checked to determine if anything is scheduled. If the CLOCK(0) state exists and if EMPTY is a 0, then all of TIME entries in OVCLOCK are decremented by 100, and the ones whose times are less than or equal to 100 are then scheduled appropriately onto CLOCK. Whenever an element of OVCLOCK is released, then its TIME quantity is set to -1, indicating it is free to be used again. It is also at this point where the status of EMPTY is defined.

With the contents and actions of these tables defined, the functions of $CHED and $IMUL8 can now be examined. The functional flow charts of these two routines are shown in Figures 12 and 13 respectively. It is believed that these flow charts, with the aid of the preceding discussion, should be self-explanatory and will not be discussed further.

One final item, however, does warrant discussing. This is the scheme used for determining if a node is undefined. Or, if it is defined and is being used, then determine if it has remained stable for the specified amount of time. It will be recalled that each node is defined to have three items which are:

1. VAL - bit containing the node's current value.
2. NDX - integer number indicating its index.
3. LAST - integer number indicating the last time the node changed states.

LAST is the variable that conveys these two pieces of information. That is, when the simulator is initially loaded, all of the nodes, LAST values are

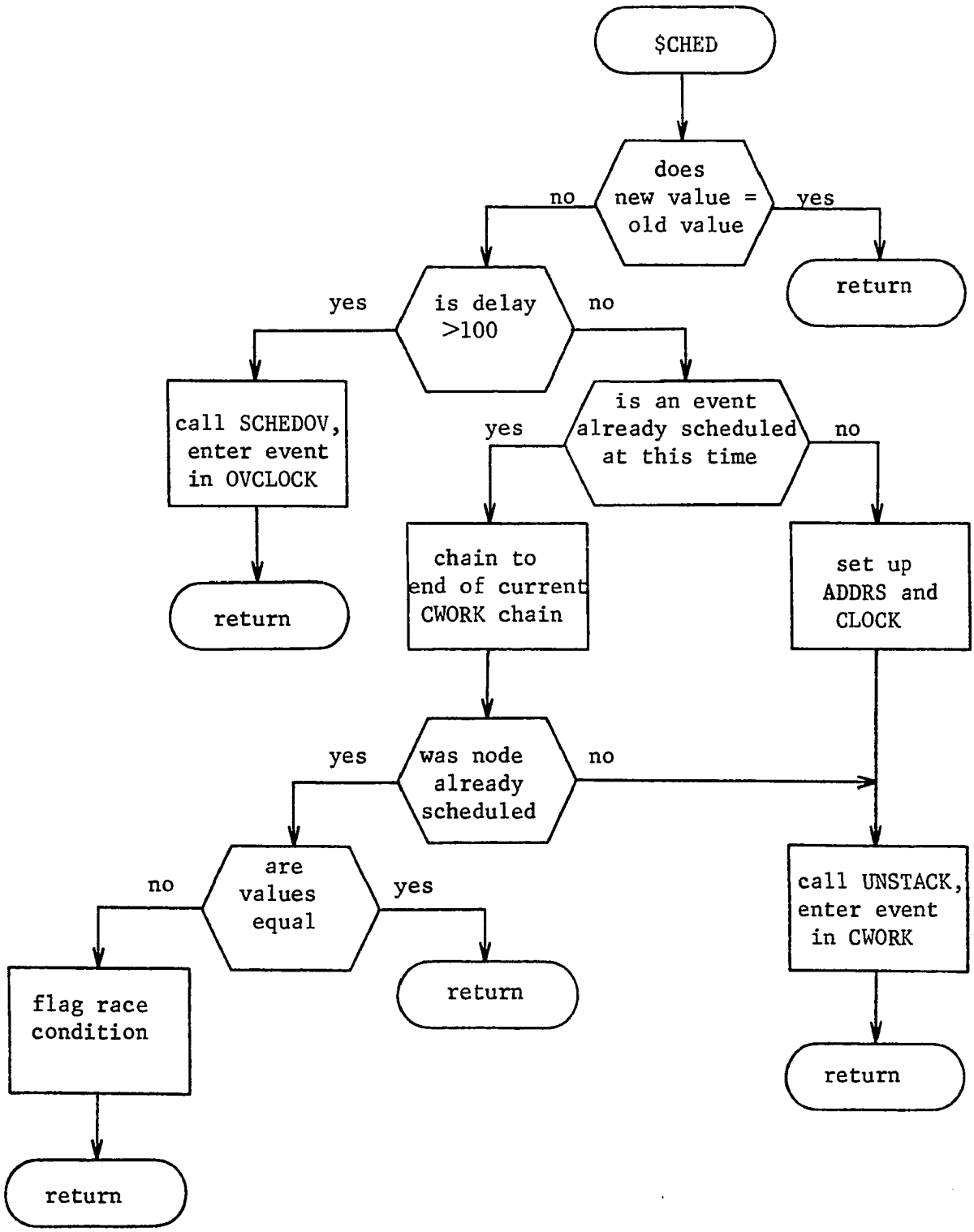


Figure 12. Event scheduler ($CHED) flow chart.

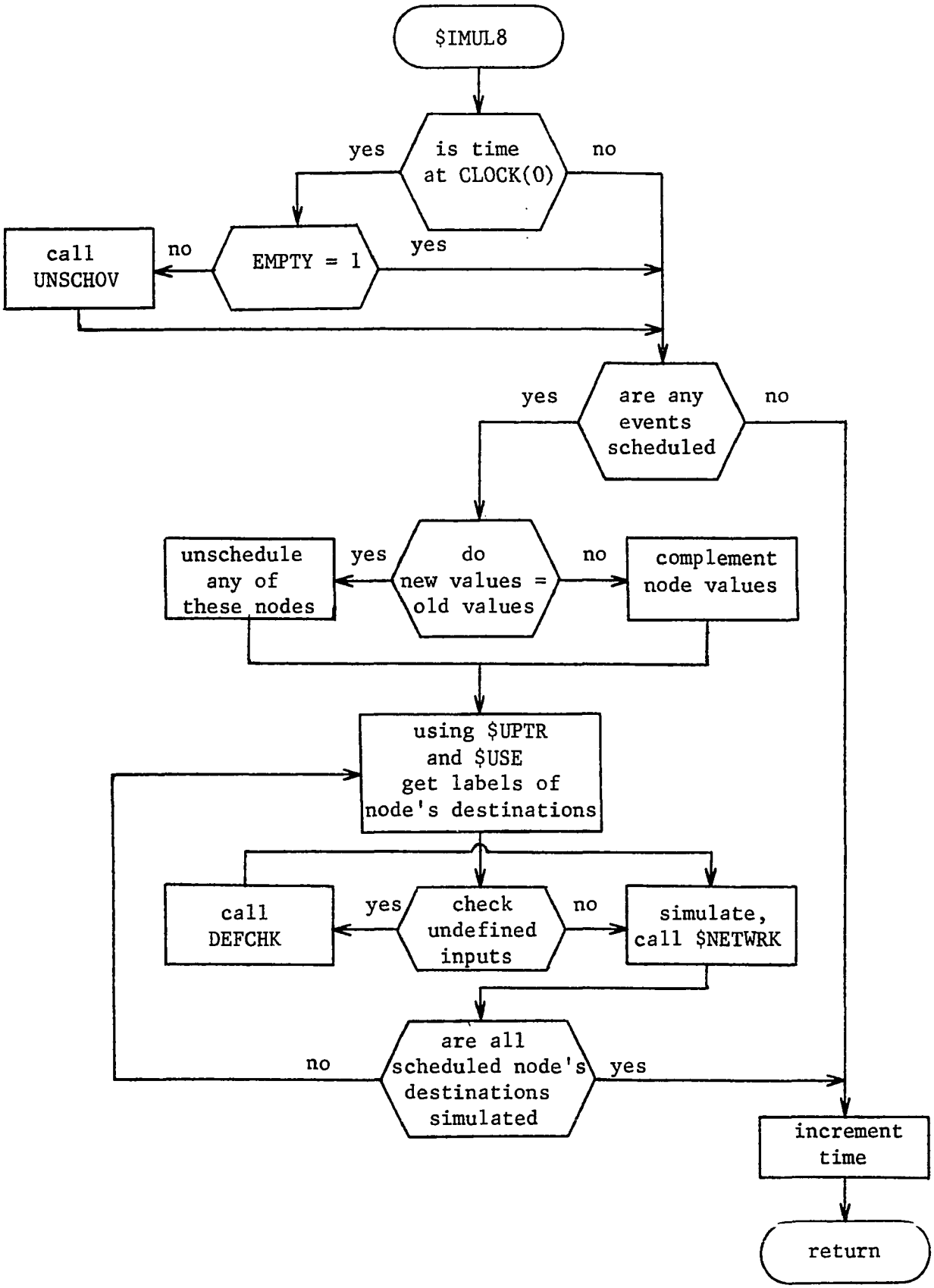


Figure 13. Simulator ($IMUL8) flow chart.

set to -1. Then, after all of the destinations of a node have been simulated, its LAST value is set to the current time. Therefore, if the gate's inputs are checked prior to simulation, any input nodes whose LAST values are -1 are flagged as having undefined inputs. Similarly, the amount of time that a node has been stable can also be checked simply by finding the difference between current time and the node's LAST value. Hence timing problems associated with flip-flops and memory elements can be detected and flagged for the user's attention.

The general philosophy adopted in MACSIM regarding the question of what to do in the event timing problems arise is as follows. If a request for a specific action is made, then that action is carried out to completion, independent of whether or not something might occur in the future to alter the end result. If and when something does occur, then the appropriate flags are raised at that time. To be more specific, take for example the case of a memory element. Suppose the select line for the element comes true, which initiates a memory cycle. Suppose further that it is determined sometime later that say the address lines did not remain stable long enough to insure a proper memory cycle. In this case, the memory cycle would be completed as usual, however, the unstable address lines would be detected and flagged for the user. This philosophy does not, however, affect such timing problems as are encountered when, say the input pulse to a gate can be masked out or suppressed at the output due to unequal 0 and 1 propogation delays through the gate.

It could be argued, of course, that this approach does not accurately model the actual hardware situation. This approach was taken for at least

two basic reasons. First, it is not likely that one general purpose algorithm can be defined which will handle all logic families properly in the case where timing rules are being violated. Secondly, even if such an algorithm could be generated, it almost certainly would result in a significantly slower simulator. This is due to the fact that once a timing problem has been detected, then the event, which has already been scheduled to occur, has to somehow be unscheduled. This, of course, would more than likely be a time consuming operation, or at least require that additional information be defined for each node, such that the time, or times, at which it has been scheduled can be known.

CONCLUSIONS

The construction of MACSIM has demonstrated, among other things, that a gate level simulator can be developed whose input is flexible enough to allow the use of any integrated circuit family without paying the penalty of simulation slow down. In fact, the initial evidence indicates that MACSIM may very well be at least an order of magnitude faster than the present software simulators. McKay (26) defines a term called slow-down ratio (SDR) which is the ratio of simulation time to actual device time. In this article, he cites existing software simulators which have SDR's in the range of 400,000:1 to 1,200,000:1. It is very difficult to compare the SDR's of different simulators unless they are actually running the same problem because they could vary considerably depending upon the activity level of the network being simulated. However, based upon the networks that have been simulated with MACSIM, it appears that an SDR of no more than 100,000:1 is obtained. McKay also points out that a special purpose processor (i.e. hardware simulator) is being constructed which is expected to have an SDR of 400:1, which is obviously a more optimum solution, provided that the initial hardware investment of such a processor can be justified.

In addition to a special purpose processor being faster, it can also simulate larger networks. McKay's processor is expected to be capable of handling networks with 36000 nodes which is about an order of magnitude better than the software simulators. When running MACSIM on the IBM 360/65 at Iowa State University, the largest practical region size that should be allocated is 256K, where K stands for 1024 bytes. The exact network size that can be simulated in 256K is a function of the types of elements which

make up the network, since MACSIM is a compiled code simulator. However, any of the following estimates appear to be conservative:

1. Approximately 600 four-input NAND elements, or their equivalent. This would correspond to about 2400 nodes and 300 packages.
2. Approximately 350 eight-input NAND elements, which corresponds to about 2800 nodes and 350 packages.
3. Approximately 140 JK flip-flops, which corresponds to probably 70 packages.
4. More realistically, the following mix could be accomodated: 1024 bit memory, 8 eight bit registers constructed from JK flip-flops, and 150 four-input NAND gates. This would correspond to about 125 packages.

What all of this means is the following. Given a board size of about 12 inches by 12 inches, which is a typical size of board presently being used, approximately 150 packages could be placed on a single board (assuming 14 pin dual-in-line packages). Hence a user could almost certainly simulate at least one board of logic at a time, and quite possibly two boards. This, of course, represents a significant amount of logic circuitry.

The actual computer time costs for making MACSIM runs are not cheap if compared to the typical costs of, say, students' Fortran runs. However, the costs can not really be compared. First, because a MACSIM program is typically much more complex in terms of the nature of the problem being solved. Secondly, the simulation costs must be compared to the costs of the alternative solution. That is, the cost of bread boarding and debugging the hardware. The actual cost for a typical student defined network appears to be somewhere between $5 and $10.

These costs, at least for the relatively small networks that have been run to date, can be broken down as follows:

1. Compile the macros, network, and simulation controller into a PL/1 subroutine - 19%.
2. Compile the above subroutine into IBM 360/65 object code - 46%.
3. Link edit the above object code with the object code of the other simulator subroutines - 27%.
4. Actual simulation - 8%.

This cost break-down is in the proportion that one would hope it to be, since only 27% of the total job cost is spent on compiling the network into a computer recognizable form and simulating (parts 1 and 4). Hence, the bulk of the cost comes from the PL/1 compiler and linkage editor. Therefore, the cost of a MACSIM run could probably be at least cut in half by simply changing the appropriate routines in MACSIM1 to output the equivalent assembler language code as opposed to the PL/1 code that they presently generate. This would actually be quite an easy task since all of MACSIM has been modularly constructed, and would in fact mean modifying 14 relatively small subroutines which do nothing more than generate the appropriate character strings. That is, all of the parsing and table generating subroutines would not have to be modified in any way.

Not only can any integrated circuit family be modeled, but it can be modeled quite easily for at least two reasons. First the basic primitive macro set includes logic elements which are of a higher degree of complexity than those found in existing gate level simulators, and are of the type most commonly used in network design. Secondly, very complex macros can

easily be defined because macros can be defined based upon other macros, and not merely upon the primitive macro set. Hence, modelling MSI and LSI circuits is greatly facilitated. In addition, MACSIM has been defined in such a way as to allow, without any modifications, the usage of a data base which would contain the set of commonly used macros in the event a group of users would wish to construct their networks from the same types of circuit elements. This, of course, would mean that the user would not have to define macros at all, or at most, he would only have to define a few extra ones to add to the data base for his particular network. Hence, the user would only need to define his network and his network exerciser.

Because MACSIM's macro definitions are flexible, it also means it will not become obsolete simply because some logic family might become obsolete.

It is felt that the actual mechanics of writing a MACSIM program should be quite easy for a designer to learn because of both the nature of the instructions and because the input specifications are virtually completely free format. In fact, MACSIM was presented to a class containing both electrical engineering and computer science students, none of whom had any previous simulator experience of any kind, and many of whom had virtually no experience in actual network design. Following approximately a two hour discussion, these students seemed to have very little trouble in any of the three phases of defining macros, networks, and exercisers.

In addition, the output is in a form that is easily recognizable and understandable to the designer. MACSIM does not, however, make any attempt to tell the user what the "correct" answer is, nor to fix up probable design errors. All of these judgement-type decisions are left to the user. The

primary reason this is done is because no specific logic family is assumed and hence, actual violations of a family's rules can not be detected.

For reasons mentioned previously, a simulator was the goal of this work. If however, the detection of races, hazards, and faults is desired, then simulation is not necessarily the best solution. Harrison and Olson (27) have an alternate solution which is faster than simulation. For certain types of fault detection, three valued simulation may be desirable (28), where the values are 0, 1, and undefined. In this scheme, instantaneous switching times are not assumed as is the case with two valued simulation. One other scheme is often used in fault detection which is called parallel simulation. Basically, this means that each node is defined to have several binary values, rather than just one. For instance, suppose all nodes are defined to have eight binary values. Then, eight initial conditions of the network could be set up, which when simulated, would result in up to eight different solutions, all for approximately the same cost as one solution. This has not been extensively investigated; however, due to MACSIM's structure, it is believed that such a scheme could quite easily be added to MACSIM at some future time if, in fact, it were desired.

In concluding, it is believed that MACSIM contains several features which make it both unique and preferable over the existing gate level simulators which are known to date. In addition, because of the way in which it was constructed, it should prove to be a valuable tool and building block for any future work in development of a more fully automated design system.

BIBLIOGRAPHY

1. Breuer, M. A., "Recent Developments in the Automated Design and Analysis of Digital Systems," Proceedings of the IEEE 60, No. 1: 12-27 (1972).

2. Rath, J. R., "Systematic Design of Automatic," AFIPS Fall Joint Computer Conference 27, Part 1: 1093-1100 (1965).

3. Gorman, D. F. and J. P. Anderson, "A Logic Design Translator," AFIPS Fall Joint Computer Conference Proc. 22: 251-261 (1962).

4. Schlaeppi, H. P., "A Formal Language for Describing Machine Logic, Timing, and Sequencing (LOTIS)," IEEE Trans. on Computers EC-13, No. 4: 439-448 (1964).

5. Procter, R. M., "A Logic Design Translator Experiment Demonstrating Relationships of Language to Systems and Logic Design," IEEE Trans. on Computers EC-13, No. 4: 422-430 (1964).

6. Schorn, H., "Computer-aided Design Systems and Analysis Using a Register Transfer Language," IEEE Trans. on Computers EC-13, No. 6: 730-737 (1964).

7. Duley, J. R. and D. L. Dietmeyer, "A Digital Design Language (DDL)," IEEE Trans. on Computers C-17, No. 9: 850-861 (1968).

8. Friedman, T. D. and S. C. Yang,"Methods Used in an Automated Logic Design Generator (ALERT)," IEEE Trans. on Computers C-18, No. 7: 593-614 (1969).

9. Potash, H., A. Tyrrill, D. Allen, A. Joseph, and G. Estrin, "DCDS Digital Simulating System," AFIPS Fall Joint Computer Conference 35: 707-720 (1969).

10. Stabler, E. P., "System Description Languages," IEEE Trans. on Computers C-19, No. 12: 1160-1173 (1970).

11. Pumplin, B. A., "A Programming System for the Simulation of Digital Machines," unpublished Ph.D. thesis, Ames, Iowa, Library, Iowa State University of Sciences and Technology (1971).

12. Ellis, D. T., "A Synthesis of Combinational Logic with NAND or NOR Elements," IEEE Trans. on Computers EC-14, No. 5: 701-705 (1965).

13. Dietmeyer, D. L. and S. Y. H. Su, "Logic Design Automation of Fan-in Limited NAND Networks," IEEE Trans. on Computers C-18, No. 1: 11-22 (1969).

14. Schultz, G. W., "An Algorithm for the Synthesis of Complex Sequential Network," Computer Design 8, No. 3: 49-55 (1969).

15. Dietmeyer, D. L., "Automated NAND Network Sythesis," Computer Design 10, No. 3: 53-58 (1971).

16. Su, S. Y. H. and C. W. Nam, "Computer Aided Synthesis of Multiple-output Multilevel NAND Networks with Fan-in and Fan-out Constraints," IEEE Trans. on Computers C-20, No. 12: 1445-1454 (1971).

17. Shalla, L., "Automatic Analysis of Electronic Digital Circuits using List Processing," Communications of the ACM 9, No. 5: 372-380 (1966).

18. Hardie, F. H. and R. J. Suhocki, "Design and Use of Fault Simulation for Saturn Computer Design," IEEE Trans. on Computers EC-16, No. 4: 412-429 (1967).

19. Hays, G. G., "Computer-aided Design: Simulation of Digital Design Logic," IEEE Trans. on Computers C-18, No. 1: 1-10 (1969).

20. Ulrich, E. G., "Exclusive Simulation of Activity in Digital Networks," Communications of the ACM 12, No. 2: 102-110 (1969).

21. Scheff, B. H., "A Machine Aids System for Digital Designers," Computer Design 8, No. 10: 76-81 (1969).

22. Kofard, J. and R. Walker, "A Modular Fairchild Computer Aided Design Program," Fairchild Semiconductor, Palo Alto, California (1969).

23. Szygenda, A., D. Rouse, and E. Thompson, "A Model and Implementation of a Universal Delay Simulator for Large Digital Nets," AFIPS Spring Joint Computer Conference 36: 207-215 (1970).

24. Austin, B. J., "Use of a Macro Processor in Logical Design," IEEE Trans. on Computers C-19, No. 11: 1085-1089 (1970).

25. Glass, R. L., "An Elementary Discussion of Compiler/Interperter Writing," Computing Surveys 1, No. 1: 55-77 (1969).

26. McKay, A. R., "Comment on "Computer-aided Design: Simulation of Digital Design Logic"," IEEE Trans. on Computers C-18, No. 9: 862 (1969).

27. Harrison, R. A. and D. J. Olson, "Race Analysis of Digital Systems without Logic Simulation," 8th Annual Design Automation Workshop, Atlantic City, New Jersey (1971).

28. Breuer, M. A., "A Note on Three Valued Simulation," IEEE Trans. on Computers C-21, No. 4: 399-402 (1972).

Additional References

Beardsley, C. W., "Computer Aids for IC Design, Artwork, and Mask Generation," IEEE Spectrum 8, No. 9: 63-79 (1971).

Breuer, M. A., "General Survey of Design Automation of Digital Computers," Proceedings of the IEEE 54, No. 12: 1708-1921 (1966).

Breuer, M. A., "Functional Partitioning and Simulation of Digital Circuits," IEEE Trans. on Computers C-19, No. 11: 1038-1046 (1970).

Dlugatch, I., "The Applicability of Computer-aided Design as a System Engineering Tool," IEEE Special Issue on Computer Aided Design 55, No. 11: 1940-1945 (1967).

Haney, F. M.,"ISDS: A Program that Designs Computer Instructions," AFIPS Fall Joint Computer Conference 35: 575-580 (1969).

IBM Corporation. IBM System/360 PL/1 Reference Manual. IBM Corporation, White Plains, New York (1968).

Iverson, K. E., A Programming Language, John Wiley and Sons, Inc., New York (1962).

Lake, D. W., "Logic Simulation in Digital Systems," Computer Design 9, No. 5: 77-83 (1970).

Lawson, H. W., Jr., "PL/1 List Processing," Communications of the ACM 10, No. 6: 358-367 (1967).

Maley, G. A. and J. Earle, The Logic Design of Transistor Digital Computers, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1963).

McDougall, M. H., "Computer System Simulation: An Introduction," Computing Surveys 2, No. 3: 191-210 (1970).

Rosen, S., Programming Systems and Languages, McGraw-Hill Book Company, New York (1967).

Smith, W. R., "Fairchild Experimental Logic Documentation System," Fairchild Semiconductor, Palo Alto, California (1970).

The Integrated Circuits Catalog for Design Engineers. Texas Instruments Incorporated, Dallas, Texas (1972).

Ulrich, E. G., "Time Sequenced Logical Simulation Based on Circuit Delay and Selective Tracing of Active Network Paths," Proc. of the ACM National Conference 20: 437-448 (1965).

Waxman, R., M. T. McMahon, B. J. Crawford, and A. B. DeAndrade, "Automated Logic Design Techniques Applicable to Integrated Circuitry Technology," AFIPS Fall Joint Computer Conference 29: 247-265 (1966).

Young, S., "A Microprogram Simulator," 8th Annual Design Automation Workshop, Atlantic City, New Jersey (1971).

ACKNOWLEDGEMENTS

The author wishes to thank both A. V. Pohm and R. J. Zingg for their many helpful suggestions and criticisms which resulted in the inclusion of several desirable features into this system which might have otherwise been overlooked. The author also thanks W. B. Boast for his generous support of computer time, and to Sheila for her typing of this manuscript. And finally, the author thanks Carol, Leigh, and Kaye for their many years of patience.

This work was partially supported by the Iowa State Affiliates Program in Solid State Electronics.

APPENDIX 1. MACSIM USER'S MANUAL

This document, referred to as Appendix 1, 2, and 3, is intended to be a stand-alone document whereby the prospective MACSIM user can become sufficiently familiar with the language structure such that he will then be capable of modeling his network using macro definitions and ultimately supplying the necessary instructions to fully simulate his system. Therefore, an apology is given to the reader who has already read the thesis and hence may find some of the following information to be redundant.

A MACSIM program must consist of basically three parts. They are: (1) definition of the macros, (2) definition of the network that is to be simulated, and (3) definition of the control program that will exercise the network.

The first part, defining the macros, is the phase in which the network building blocks are defined. A macro must consist of a list of the input and output nodes, the drive capability of the outputs, the loading caused by the inputs, the timing information denoting the amount of time for the output to respond to input stimuli, and finally, the logical switching function performed by the macro.

The timing and logical functions can be defined in two different ways. First, by using a set of predefined primitive macros, or secondly, by using macros which are defined elsewhere in the macro definitions. The primitive macro names along with their required timing quantities are shown in Table 8. The definitions of these timing quantities are also included in Table 8. It should be noted here that a specification of

Table 8. List of primitive macros and their timing specification requirements.

primitive function	primitive name	timing
combinational	$NOT, $OR, $AND, $NOR, $NAND	DEL_0, DEL_1
sequential	$RSFF	DEL_0, DEL_1
	$JKFF	DEL_0, DEL_1 MIN_CLK, MIN_CLR
	$DFF	DEL_0, DEL_1, MIN_CLK, MIN_CLR, MIN_SET, MIN_HLD
memory	$MEM	CYCLE, ACCESS, MIN_ADR, MIN_INP, MIN_RED, MIN_SEL

DEL_0, DEL_1 - amount of delay time for the output to go to a 0 and 1, respectively, after the appropriate input has been applied.

MIN_CLK - minimum amount of time that the clock input must be applied and stable at the input to a JK or D flip-flop.

MIN_CLR - similarly for the clear and preset inputs. Note that both inputs are assumed to have the same requirements.

MIN_SET - minimum amount of time that the data input must be set prior to the leading edge of the clock pulse for a D flip-flop.

MIN_HLD - minimum amount of hold time that the data input must be stable after the leading edge of the clock pulse for a D flip-flop.

CYCLE, ACCESS - cycle and access time for a memory element.

MIN_ADR, MIN_INP, MIN_RED, MIN_SEL - minimum amounts of time that the address, data inputs, read/write input, and select input must remain stable, respectively, for a memory element.

zero delay time is not valid. That is, all timing information must have a value of at least one.

The schematic representations for the primitive macros are shown in Figure 14. Notice that the JK and D flip-flops have direct set and reset (i.e., clear and preset) inputs. The appearance of the small circles on these as well as the other nodes has the conventional meaning of inversion. That is, a logical 0 at these inputs will perform the required function. Similarly, it is the trailing edge of the clock input pulse which causes the JK flip-flop to change states. Also, it should be noted here that all timing specifications are given in normalized units of time.

The format used for describing the macro nodes is as follows:

(output node list) (input node list) macro name;

where, the list elements are separated by commas. This same format is used later in the network definition also.

In general, all of the input specifications to MACSIM are free format. That is, items never have to appear in any specific card column, blanks can be used freely to improve readability, and instructions can extend beyond card boundaries for as many cards as is required to specify the instruction. The only restriction is that only one instruction can appear on a single card. Comments can always be inserted anywhere so long as its first, non-blank character is an asterisk (*). (Note, a comment cannot extend beyond a single card unless the asterisk is repeated on the subsequent cards). Since the appearance of a semicolon (;) always terminates the decoding of an instruction (not a comment), then comments can also be placed to the right of a semicolon if the user so desires. Card columns 73 through 80 are reserved for identification numbers.

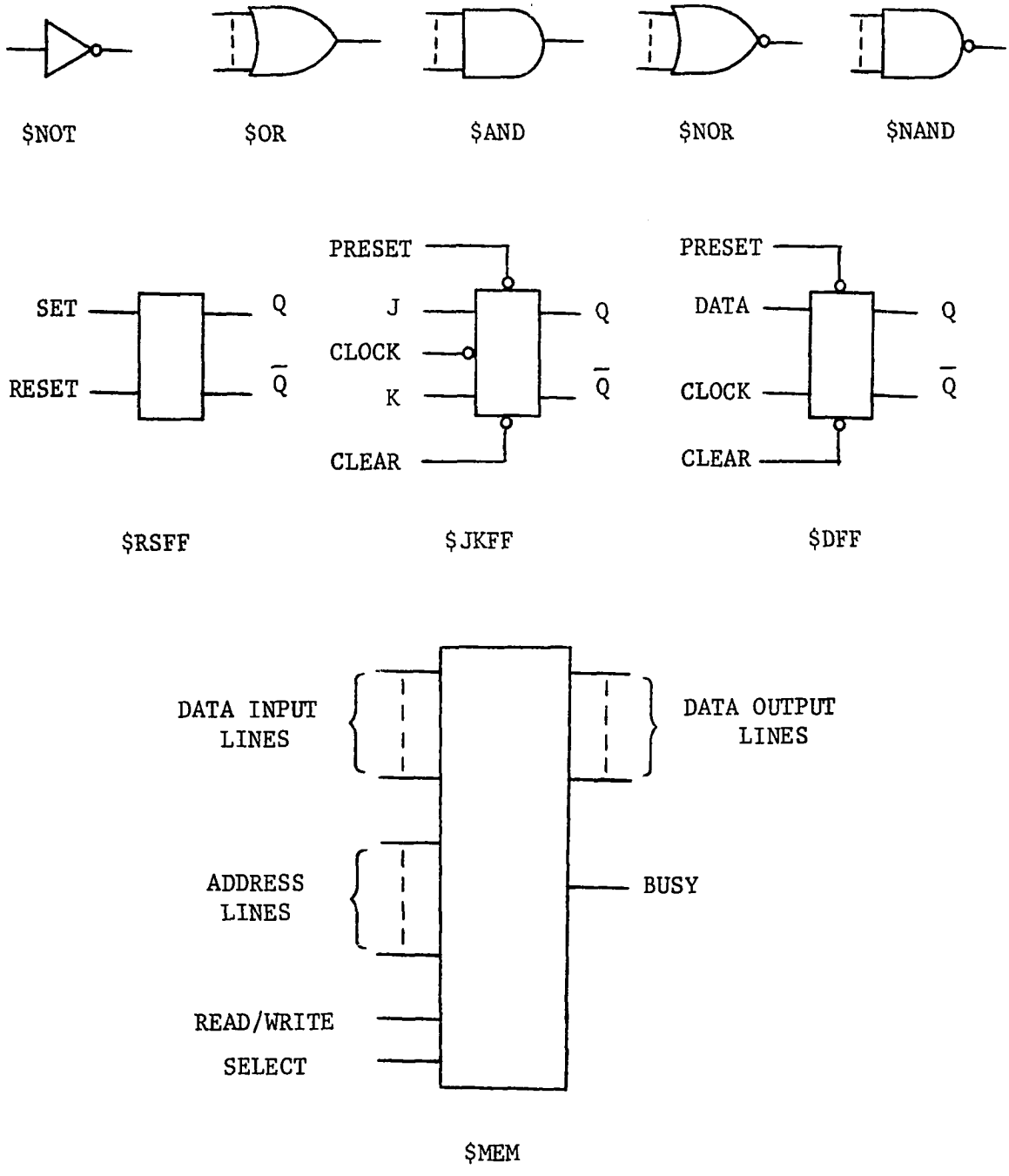


Figure 14. Schematic representation of primitive macros.

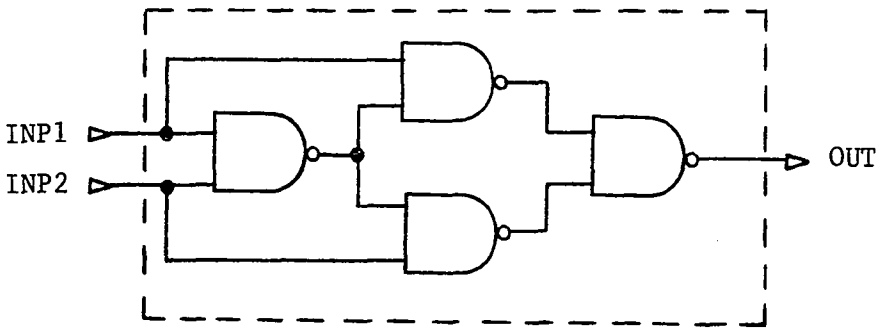
With these introductions given, the actual mechanics of defining a macro can probably best be explained through the use of the example given in Figure 15. First of all, notice all macro definitions begin with DEF_MAC; (define macro) and end with END_MAC; (end of macro). In this example a two input exclusive-or is defined. First, it (EXCL_OR) is defined using two input NAND gates (NAND2). The NAND2 gate is in turn defined using the primitive macro $NAND. The actual names used in the input and output lists are of no importance, so long as their usage remains consistent within a macro definition. Similarly, no conflict arises in the event different macros contain the same names. Next, in the case of EXCL_OR, the output is defined to have a drive capability of 10 units (normalized) and the two inputs each have a load of 2. The physical location of these two statements within the macro is of no importance, however, the ordering of the integer numbers must be the same as the ordering of the nodes for which they correspond. That is, suppose INP1 has a load of 15 and INP2 has a load of 4, then the specification would be

LOAD = 15, 4;

Again, notice that only primitive macros can convey timing information. Here, the sequence in which the timing statements are specified is not important, however, they must follow the macro statement that they are describing.

In general, the ordering of specific node names within input and output lists does not matter because the user is doing the defining. However, when using primitive macros, the following ordering of the inputs is always assumed:

Schematic representation of EXCL_OR



Macro definition of EXCL_OR

```
DEF_MAC;
     (OUT)(INP1,INP2)EXCL_OR;
        DRIVE = 10;
        LOAD  = 2,2;
     (DUM1)(INP1,INP2)NAND2;
     (DUM2)(INP1,DUM1)NAND2;
     (DUM3)(INP2,DUM1)NAND2;
     (OUT)(DUM2,DUM3)NAND2;
END_MAC;
DEF_MAC;
     (OUTPUT)(INPUT1,INPUT2)NAND2;
        DRIVE = 10;
        LOAD  = 1,1;
     (OUTPUT)(INPUT1,INPUT2)$NAND;
        DEL_0 = 8;
        DEL_1 = 12;
END_MAC;
```

Figure 15. Example of a macro definition for a two-input exclusive - or circuit.

(out, outbar) (set, reset) $RSFF

(out, outbar) (J, K, clock, clear, preset) $JKFF

(out, outbar) (D, clock, clear, preset) $DFF

(memory name, busy, data-out$_n$, ---, data-out$_0$)

(select, read, data-in$_n$, ---, data-in$_0$, adr$_k$, ---, adr$_0$) $MEM

where, out and outbar are the true and complement flip-flop outputs, respectively, and adr$_i$ is an address line.

With one exception, all macros can be defined upon any number of other macros. The exception is in the case of defining a memory macro. Due to problems of being able to decode the input and output lists, a memory macro's definition can be based upon only one macro, which is the primitive $MEM.

Once the macros are defined, the defining of the network in terms of the macros is quite straightforward. The network definition begins with DEF_NET; and ends with END_NET;. For instance, if the network consisted of simply a 4-bit parity checker, then the description might look as follows:

```
DEF_NET;
  (A0) (BIT0, BIT1) EXCL_OR;
  (A1) (BIT2, BIT3) EXCL_OR;
  (PARITY) (A0, A1) EXCL_OR;
END_NET;
```

where, the network inputs have arbitrarily been named BIT0 through BIT3 and an output called PARITY.

It should be noted here that two additional commands exist. They are DMP_MAC (dump macro tables) and DMP_NET (dump network tables). These instructions can be placed after the last END_MAC or after the END_NET instructions, respectively. Their function is to cause the appropriate tables to be output on punched cards. The cards generated by DMP_MAC can then be used either alone or with other macro definitions for subsequent network definitions. If they are used with other macros, then these cards should be placed ahead of the macro definition cards. The first card that is punched is a LOD_MAC (load macro table) command. The actual deck of punched cards that the user will receive quite possibly will begin and end with a number of blank cards, however, they can either be left on or taken off as the user wishes. As presently implemented, the cards generated by DMP_NET can only be used for obtaining a crossreference listing. Here, the first punched card contains LOD_NET (load network table). If the user is interested in making several different simulation runs on a single network, then he will need to apply for permanent disk space on which his network description can be stored. In this case then, the deck of cards generated by DMP_NET would be used to redefine a new simulation control program.

If a cross-reference listing is desired, then the command CRS_REF; is placed following either the END_NET instruction, or else following the DMP_NET instruction if it is included.

The last specification that must be included, provided a simulation is in fact wanted, is to define the means by which the network is to be exercised. That is, the simulation control instructions must be specified. The instructions that make up this language are shown in Table 9.

Table 9. List of simulation control instructions

1.	DEF_CHECK	IF (condition);
2.	NODEF_CHECK	IF (condition);
3.	STOP	IF (condition);
4.	GOTO (label)	IF (condition);
5.	COMMENT (string)	IF (condition);
6.	HEADING (x_1, x_2, ---, x_n)	IF (condition);
7.	TRACE (x_1, x_2, ---, x_n)	IF (condition);
8.	READ (x_1, x_2, ---, x_n)	IF (condition);
9.	LOAD ($mem_1(i)$, $mem_2(i)$, ---)	IF ($i <= M$);
10.	x_1, x_2, ---, x_n = v_1, v_2, ---, v_m (t_1, t_2, ---, t_k);	

where, M = integer number
x_i = node name
v_i = binary value
t_i = integer number indicating time
$mem_j(i)$ = i-th word of memory mem_j
label = any label name
string = any string of characters
condition = any logical condition which return a Boolean true or false value

NOTE: any of the above statements can be preceded by a label name.

All of the instructions in Table 9 have been shown in their complete form. That is, all of the instructions have degenerate forms. In particular, all of the IF clauses in the first nine instructions are optional. That is, when an instruction is written in the form shown, then the specified action is taken only when the condition is satisfied. However, when the IF clause is omitted, the action is unconditionally taken.

The following is a semantic description of each of these ten instructions.

1. Define check: check all input nodes to the gate before performing the simulation on it; is the default condition.
2. No define check: do not check for undefined input nodes.
3. Terminate the simulation.
4. Transfer control to the specified label within the control program.
5. Print the comment in the output trace denoted by this specified string.
6. Print the specified node names in the heading at the top of each page of output trace.
7. Print the binary values of the specified node names. These node names must be contained within the list of node names specified in the heading statement. Not all of the preceding names need be present, however, they must be in the same order. If no node name list is specified, then all of nodes specified in the heading are traced.
8. Read the binary values from punched cards and assign them to the specified nodes. All of the binary values (0 or 1) corresponding

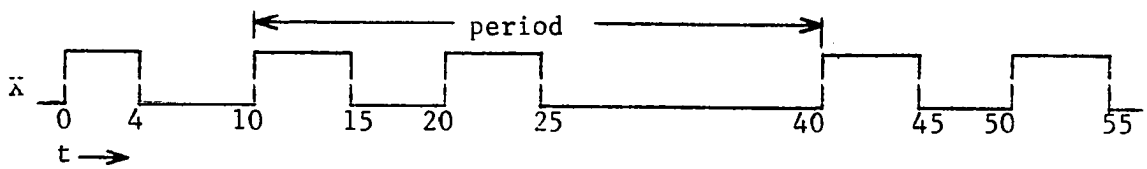
to the execution of each read statement must be punched in consecutive card columns beginning in the first card column.

9. Perform, in effect, the same function as READ. However, is used to facilitate the loading of memory elements only, where the bit strings for each word are punched on a separate card. In the form shown, all of the first 0 through M words of mem_1 would be loaded, and then similarly for any other memory elements specified. In the case of this instruction, if no IF clause is present, then the subscript i must be an integer number, or else no subscript need be specified at all. In the first case, the specified word in memory would be loaded, whereas in the second case, the entire memory would be loaded.

10. Assignment statement: the elements in the node list are assigned the corresponding values in the value list at time t_1. In the event $m < n$, then the last (n-m) nodes are set to the value of v_m. At time t_2, etc., the value list is complemented and the assignments made again. Time t = 0 is assumed if no time list is specified. A periodic signal can also be defined (i.e., a clock signal) by specifying

$$t_k = t_{k-1} - \text{period}.$$

An example is shown below. This example is not intended to be practical or a useful one, rather it is chosen to emphasize the power of the assignment statement.



then, X = 1(0,4,10,15,20,25,40,10);

As stated previously, the IF condition on all of the first nine instructions is optional. The nature of the condition for instruction 9 has already been explained, however, the types of conditions that can be specified in the first eight instructions warrant further explanation. This condition is, by definition, any clause which, when executed, will return a Boolean true or false value. In the case of MACSIM, any valid PL/1 conditional statement is legitimate. The following are a few types of conditional statements.

1. IF (TIME <= 100)
2. IF (¬ STABLE)
3. IF (MOD (TIME, 10) = 0)
4. IF (A & B)

where, TIME = reserved word indicating current time

STABLE = reserved word indicating the entire network is presently in a stable condition

MOD = built-in PL/1 modulo arithmetic function

A,B = network node names

Note, the standard Boolean operators of NOT, OR, and AND are indicated by the symbols ¬, | , and & respectively. The interpretation of these examples is as follows:

1. if current time is less than or equal to 100
2. if the network is not stable
3. if current time, modulo 10, equals 0
4. if nodes A and B are both true

The meaning of and the nature in which the reserved word STABLE is used should be apparent from the preceding example. However, the general nature in which the simulator functions needs to be explained before the exact meaning of the reserved word TIME can be defined.

First of all, it would seem that in order to exercise a network, the user would first like to apply a set of stimuli to the network, and then simulate until some predefined condition results. Therefore, the total control program could be constructed from a series of these loops. This is not unlike the DO-END and BEGIN-END blocks of other programming languages. It can be seen by examining the instruction set in Table 9 that there is no way that the user can specifically call the simulator or initialize and increment TIME. This was done because, if the assumption is correct that the control specification would be made up by a series of these loops, then it is possible for the system to automatically handle these functions for the user, and hence relieve him from having to repeatedly respecify them.

Since the reserved word TIME is automatically incremented by the system, then it follows that it must also be initialized automatically. It was felt that TIME would be more useful to a user if it was reinitialized each time before entering a new loop group, where a loop is defined to begin with a label and end with a GOTO to that label. This was done so that the user need not have any knowledge of the total elapsed time for an action to occur at the time when he is specifying the instructions.

Simply stated, what all of this means is that TIME can be thought of as merely a loop counter that is initialized each time before entering

a new loop (i.e., label), and the scope of TIME as well as all specified actions is always local or internal to that loop. Also, this means that labels, for the most part, can be used in the conventional sense as in other programming languages, however, the user must be aware of the system actions taken on TIME by the appearance of a label. That is, the appearance of a label must have a specific function to perform (i.e., the argument of a GOTO statement) and should not be arbitrarily used on just any instruction.

A general note regarding the nature of network node names should be made. All names can be from one to 16 characters long. However, it is possible that up to four characters will be concatenated onto a user's node name in the event the system needs to create a dummy internal name. Therefore, to insure that all names will always be unique, it would be advisable not to exceed 12 characters per name. All names must begin with an alpha character of A through Z. The subsequent characters can be any of the following:

$|#|@|_|A|B|---|Y|Z|0|1|---|8|9

where, the symbol | is read as "or".

Also with regard to names, all of the words making up the instructions shown in Table 9 as well as all of the preceding instructions such as DEF_MAC, etc. are reserved words and cannot be used by the user to indicate node names.

It should be emphasized that quite a large variety of user's specification errors can be detected by MACSIM at the time the program is being entered, however, it is by no means fool-proof. The technique used to

implement MACSIM is to compile the user's program into a PL/1 program, which in turn is compiled into the object code of the IBM 360/65 at which time the actual simulation can take place. Hence, it is possible that a certain class of errors will not be detected by MACSIM, but rather by the PL/1 compiler. And of course there can also be a class of errors that can exist which are syntactically correct but cause the simulation to operate incorrectly. It is this class of errors, as is true with almost any programming system, that the user is obligated to detect for himself by careful examination of the simulation output trace.

The overall appearance of a MACSIM program is shown in Figure 16. In addition, all of the simulation-time warning messages that might be issued by MACSIM are shown in Table 10.

For the prospective user who is already familiar with other programming languages, and in particular who is familiar with Backus Normal notation, should refer to Tables 11 and 12 where the formal syntax description for MACSIM is given. These tables should also be referred to any time the user finds any of the preceding discussion to be incomplete for his requirements.

The notation used here is called a modified Backus Normal form. The symbols ::= are read as "is defined by", the symbol | is read as "or", the brackets [ ] denote "zero or more occurrences of", the brackets { } denote "one or more occurrences of", and the brackets < > are used to enclose non-primitive terms. All other symbols and all terms typed in upper case appear literally in the macro description text. As usual, the term on the left-hand side of ::= is the defined item, and the term on the right-hand side is the defining item(s).

The job control cards required for running MACSIM at the Iowa State University computation center are shown in Appendix 2, and a complete sample program is shown in Appendix 3.

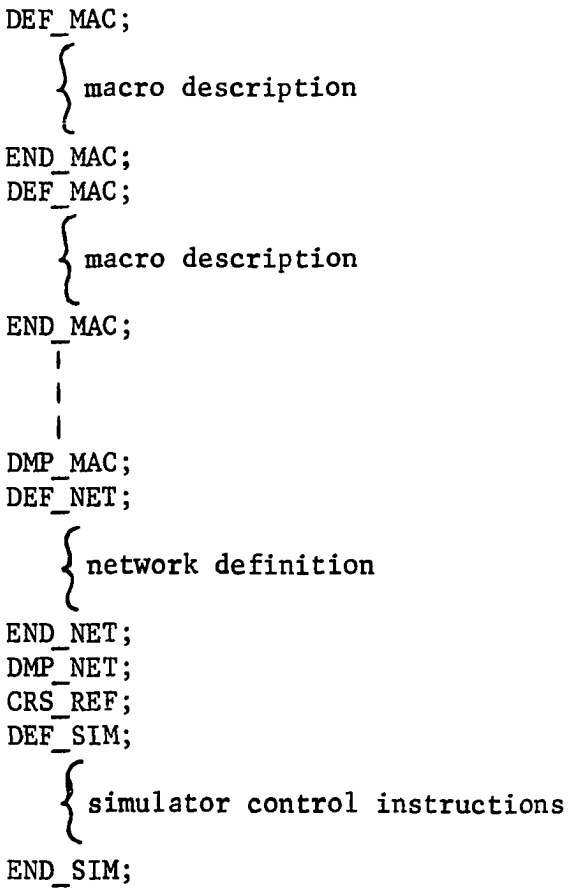


Figure 16. Key-words and structure of input cards for a MACSIM program.

Table 10. List of simulation error messages.

1.	NODE name IS BEING DRIVEN TO CONFLICTING STATES AT TIME = time
2.	NODE name IS BEING USED BUT IS UNDEFINED, IS ASSUMED 0
3.	BOTH SET AND RESET INPUTS ARE HIGH INTO RSFF name
4.	CLEAR INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
5.	PRESET INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
6.	CLOCK INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
7.	J INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
8.	K INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
9.	Q-SIDE INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
10.	QBAR-SIDE INPUT TO FLIP-FLOP name NOT STABLE LONG ENOUGH
11.	DATA INPUT INTO FLIP-FLOP name NOT STABLE LONG ENOUGH PRIOR TO CLOCK
12.	DATA INPUT INTO FLIP-FLOP name NOT STABLE LONG ENOUGH AFTER CLOCK
13.	CLOCKING OF FLIP-FLOP name ATTEMPTED DURING CLEAR OR PRESET OPERATION
14.	ACCESS TO MEMORY name ATTEMPTED WHILE STILL BUSY
15.	ADDRESS LINE(S) NOT STABLE ON INPUT TO MEMORY name
16.	DATA-IN LINE(S) NOT STABLE ON INPUT TO MEMORY name
17.	READ LINE(S) NOT STABLE ON INPUT TO MEMORY name
18.	SELECT LINE(S) NOT STABLE ON INPUT TO MEMORY name

Table 11. Macro definition syntax

<macro> ::= DEF_MAC; <macro desc> END_MAC;

<macro desc> ::= {<defined macro> | <prim macro>}[<comment>]

<comment> ::= [blank] * [<alphameric>]

<defined macro> ::= (<list>) (<list>) <macro name> ; <drive-load desc>

<prim macro> ::= (<list>)(<list>) <prim macro name> ; <timing desc>

<prim macro name> ::= $NOT|$OR|$AND|$NOR|$NAND|
$RSFF|$JKFF|$DFF|$MEM

<list> ::= <identifier> [, <identifier>]

<macro name> ::= <identifier>

<drive-load desc> ::= <drive-load key> = <integer> [, <integer>];

<drive-load key> ::= DRIVE|LOAD

<timing desc> ::= <timing key> = <integer>;

<timing key> ::= DEL_0|DEL_1|MIN_CLK|MIN_CLR|
MIN_SET|MIN_HLD|CYCLE|ACCESS|
MIN_ADR|MIN_INP|MIN_RED|MIN_SEL

<identifier> ::= <letter> [<alphameric>]

<alphameric> ::= $|#|@|_|<letter>|<digit>

<integer> ::= <digit> [<digit>]

<letter> ::= A|B| --- |Y|Z

<digit> ::= 0|1| --- |8|9

Table 12. Syntax definition of control language.

```
<control program> ::= DEF_SIM; {<control statement>}
                      [<comment statement>] END_SIM;

<comment statement> ::= [blank] * [<string>]

<control statement> ::= [<label>:] <assignment statement>;|
                        [<label>:] <operation clause> [<if clause>];

<assignment statement> ::= <node list> = <value list> [( <time list> )]

<if clause> ::= IF ( <condition> )

<operation clause> ::= DEF_CHECK|NODEF_CHECK|STOP|<go to>|
                       <comment>|<heading>|<trace>|<read>|<load>

<go to> ::= GOTO ( <label> )

<comment> ::= COMMENT ( <string> )

<heading> ::= HEADING ( <node list> )

<trace> ::= TRACE [( <node list> )]

<read> ::= READ ( <node list> )

<load> ::= LOAD ( <memory list> )

<node list> ::= <name> [, <name>]

<value list> ::= <binary digit> [, <binary digit>]

<time list> ::= <integer> [, <integer>]

<memory list> ::= <name> [( <subscript> )] [, <name> [( <subscript> )]]

<condition> ::= <string>

<label> ::= <name>

<subscript> ::= <integer>|<name>

<name> ::= <alpha> [<alphameric>]

<string> ::= {<alphameric>}

<integer> ::= {<digit>}

<alpha> ::= A|B --- |Y|Z

<digit> ::= 0|1 --- |8|9

<binary digit> ::= 0|1
```

APPENDIX 2. JOB CONTROL CARD SPECIFICATIONS

Tables 13 through 16 show all of the job control cards which are required for running any type of job related to MACSIM at the Iowa State University computation center.

For the MACSIM user, he need only concern himself with Table 13. If, in fact, he wishes to submit his job at the student submittal area, then he can ignore all of Appendix 2.

Tables 14 through 16 will be of concern to the system's programmer who is either interested in the nature in which MACSIM is handled on disk, or else who is interested in making modifications to any part of the MACSIM source code.

In all cases, jobname, account number, and programmer name must be supplied by the appropriate user.

In the event large networks are to be simulated, and when long simulation times are expected, then the user should contact one of the computation center's system's programmers in order to determine which of the parameters in Table 13 to change in order to permit complete operation.

If the user does not wish to simulate his network, then the cards beginning with, and following,

```
//STEP3 EXEC ---
```

can be omitted.

Table 12. Syntax definition of control language.

```
<control program> ::= DEF_SIM; {<control statement>}
                      [<comment statement>] END_SIM;

<comment statement> ::= [blank] * [<string>]

<control statement> ::= [<label>:] <assignment statement>;|
                        [<label>:] <operation clause> [<if clause>];

<assignment statement> ::= <node list> = <value list> [( <time list> )]

<if clause> ::= IF ( <condition> )

<operation clause> ::= DEF_CHECK|NODEF_CHECK|STOP|<go to>|
                       <comment>|<heading>|<trace>|<read>|<load>

<go to> ::= GOTO ( <label> )

<comment> ::= COMMENT ( <string> )

<heading> ::= HEADING ( <node list> )

<trace> ::= TRACE [( <node list> )]

<read> ::= READ ( <node list> )

<load> ::= LOAD ( <memory list> )

<node list> ::= <name> [, <name>]

<value list> ::= <binary digit> [, <binary digit>]

<time list> ::= <integer> [, <integer>]

<memory list> ::= <name> [( <subscript> )] [, <name> [( <subscript> )]]

<condition> ::= <string>

<label> ::= <name>

<subscript> ::= <integer>|<name>

<name> ::= <alpha> [<alphameric>]

<string> ::= {<alphameric>}

<integer> ::= {<digit>}

<alpha> ::= A|B --- |Y|Z

<digit> ::= 0|1 --- |8|9

<binary digit> ::= 0|1
```

APPENDIX 2. JOB CONTROL CARD SPECIFICATIONS

Tables 13 through 16 show all of the job control cards which are required for running any type of job related to MACSIM at the Iowa State University computation center.

For the MACSIM user, he need only concern himself with Table 13. If, in fact, he wishes to submit his job at the student submittal area, then he can ignore all of Appendix 2.

Tables 14 through 16 will be of concern to the system's programmer who is either interested in the nature in which MACSIM is handled on disk, or else who is interested in making modifications to any part of the MACSIM source code.

In all cases, jobname, account number, and programmer name must be supplied by the appropriate user.

In the event large networks are to be simulated, and when long simulation times are expected, then the user should contact one of the computation center's system's programmers in order to determine which of the parameters in Table 13 to change in order to permit complete operation.

If the user does not wish to simulate his network, then the cards beginning with, and following,

```
//STEP3 EXEC ---
```

can be omitted.

Table 13. User's job control cards.

```
//jobname  JOB  'acct.#,TIME=4,REGION=128K',programmer
//STEP1    EXEC PGM=IEFBR14,TIME=(,30)
//ALLOCATE  DD DSNAME=&&CODE,DISP=(NEW,PASS),
//      UNIT=SPOOL,SPACE=(TRK,(4,4,1)),
//      DCB=(RECFM=FB,LRECL=90,BLKSIZE=400)
//STEP2   EXEC PGM=MACSIM1,REGION=128K
//STEPLIB   DD DSNAME=PROG.I4153MC1,DISP=SHR
//SYSPRINT  DD SYSOUT=A,SPACE=(CYCL,(10)),
//      DCB=(RECFM=VBA,LRECL=137,BLKSIZE=3292)
//PUNCH     DD SYSOUT=B,DCB=(RECFM=FB,LRECL=80,BLKSIZE=1600,BUFNO=1)
//SYSIN     DD *
      { User's MACSIM program cards
/*
//NETWORK   DD DSNAME=&&CODE(NETWORK),DISP=(OLD,PASS),
//      VOLUME=REF=*.STEP1.ALLOCATE
//DECLARE   DD DSNAME=&&CODE(DECLARE),DISP=(OLD,PASS),
//      VOLUME=REF=*.STEP1.ALLOCATE
//CONTROL   DD DSNAME=&&CODE(CONTROL),DISP=(OLD,PASS),
//      VOLUME=REF=*.STEP1.ALLOCATE
//STEP3   EXEC PL1F,PARM.PL1L='MACRO,NOSOURCE2',REGION=128K
//PL1L.SYSLIB DD DSNAME=&&CODE,DISP=(OLD,PASS)
//PL1L.SYSIN  DD DSNAME=PSQ.I4153PRG,DISP=(OLD,KEEP)
//LKED.USE    DD DSNAME=PROG.I4153MC2,DISP=SHR
//LKED.SYSIN  DD *
  INCLUDE USE(MACSIM2)
  ENTRY IHENTRY
/*
//GO.SYSIN  DD *
      { blank card
        { User's input data for READ and LOAD commands
/*
```

Table 14. Job control cards for compiling and loading MACSIM1 onto disk from source cards.

```
//jobname  JOB  'acct.#,TIME=5,REGION=160K',programmer,MSGLEVEL=(1,1)
//STEP1   EXEC MOD
//MOD.SYSIN DD *
  SCRATCH DSNAME=PROG.I4153MC1,VOL=2314=LIBPAK
  UNCATLG DSNAME=PROG.I4153MC1
//STEP2   EXEC PL1LFCL,PARM.PL1L='NOSTMT',TIME.PL1L=3,REGION,PL1L=160K,
//      PARM.LKED='XREF,LIST,LET,OVLY'
//PL1L.SYSIN DD *
      {  MACSIM1 source deck
//LKED.SYSLMOD DD DSNAME=PROG.I4153MC1(MACSIM1),DISP=(NEW,CATLG),
//      UNIT=DISK,VOLUME=SER=LIBPAK,SPACE=(TRK,(33,1,1),RLSE)
//LKED.SYSIN DD *
  OVERLAY ALPHA
    INSERT PARSE,**PARSEA
      OVERLAY BETA
        INSERT DEF_MAC,DEF_MACA
      OVERLAY BETA
        INSERT DEF_NET,DEF_NETA
  OVERLAY ALPHA
    INSERT SORT,***SORTA
    INSERT PUT_DCL,PUT_DCLA
  OVERLAY ALPHA
    INSERT DEF_SIM,DEF_SIMA
  OVERLAY ALPHA
    INSERT CRS_REF,CRS_REFA
  OVERLAY ALPHA
    INSERT DMP_MAC,DMP_MACA
  OVERLAY ALPHA
    INSERT LOD_MAC,LOD_MACA
  OVERLAY ALPHA
    INSERT DMP_NET,DMP_NETA
  OVERLAY ALPHA
    INSERT LOD_NET,LOD_NETA
/*
```

Table 15. Job control cards required for compiling and loading MACSIM2 from its source cards onto disk.

```
//jobname  JOB  'acct.#,TIME=5,REGION=128K',programmer,MSGLEVEL=(1,1)
//STEP1  EXEC MOD
//MOD.SYSIN DD *
  SCRATCH DSNAME=PROG.I4153MC2,VOL=2314=LIBPAK
  UNCATLG DSNAME=PROG.I4153MC2
//STEP2   EXEC PL1LFCL,PARM.PL1L='NOSTMT',REGION.PL1L=128K,
//      PARM.LKED='MAP,LIST,LET,NCAL'
//PL1L.SYSIN DD *
      {
      { MACSIM2 source deck
      {
//LKED.SYSLMOD DD DSNAME=PROG.I4153MC2(MACSIM2),DISP=(NEW,CATLG),
//      UNIT=DISK,VOLUME=SER=LIBPAK,SPACE=(TRK,(7,1,1),RLSE)
/*
```

Table 16. Job control cards required for loading (not compiling)$PROGRM onto disk from its source cards.

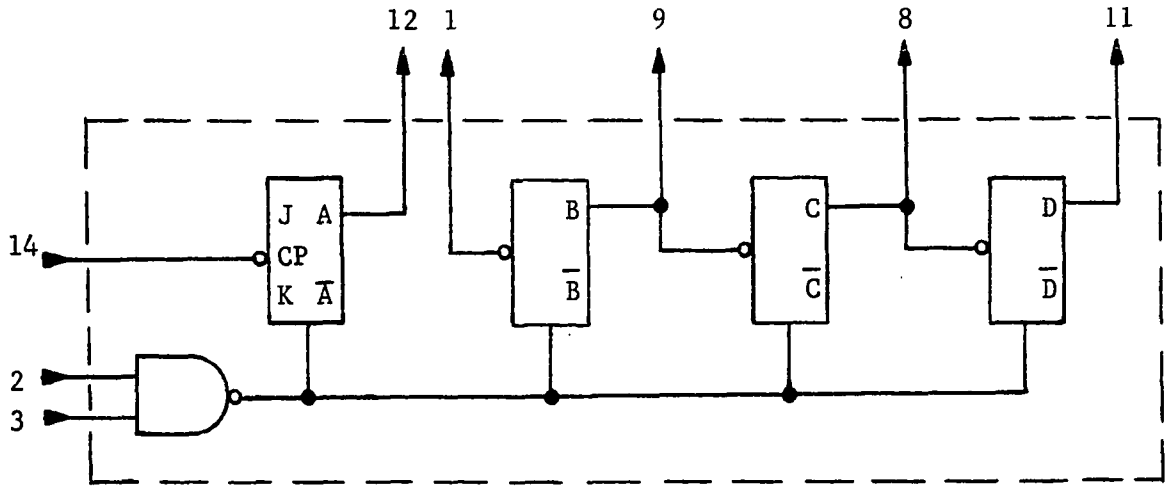
```
//jobname  JOB  'acct.#,TIME=5,REGION=96K',programmer,MSGLEVEL=(1,1)
//STEP1    EXEC MOD
//MOD.SYSIN DD *
  SCRATCH DSNAME=PSQ,I4153PRG,VOL=2314=LIBPAK
  UNCATLG DSNAME=PSQ.I4153PRG
//STEP2    EXEC PGM=IEBGENER
//SYSPRINT  DD SYSOUT=A
//SYSIN     DD DUMMY
//SYSUT2    DD DSNAME=PSQ.I4153PRG,DISP=(NEW,CATLG),
//       UNIT=DISK,VOLUME=SER=LIBPAK,
//       SPACE=(TRK,(1,1)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=1920)
//SYSUT1    DD *
      { $PROGRM source deck
/*
//STEP3    EXEC PGM=IEBGENER
//SYSPRINT  DD SYSOUT=A
//SYSIN     DD DUMMY
//SYSUT2    DD SYSOUT=A,DCB=(RECFM=FB,LRECL=80,BLKSIZE=1600)
//SYSUT1    DD DSNAME=PSQ.I4153PRG,DISP=(OLD,KEEP)
/*
```

APPENDIX 3. A COMPLETE SAMPLE PROGRAM

The following is a complete MACSIM program containing both the required user specifications and the resulting output. Figure 17 contains the schematic representations of the SN5493 which can be used as either a three or four bit ripple counter, and of the actual simulation model, which is defined by the primitive macros $JKFF and $NAND.

The following page then contains the user specification, followed by the cross-reference listing for the network. The next four pages contain the PL/1 program that was generated by MACSIM. The reader will notice that the original source statements are included as comments within the PL/1 code just prior to the code that was generated by that statement. (Note: a PL/1 comment is any string of characters enclosed by the symbols /* and */). And finally, the last two pages contain the trace that was generated during simulation.

SN5493



SIMULATION MODEL

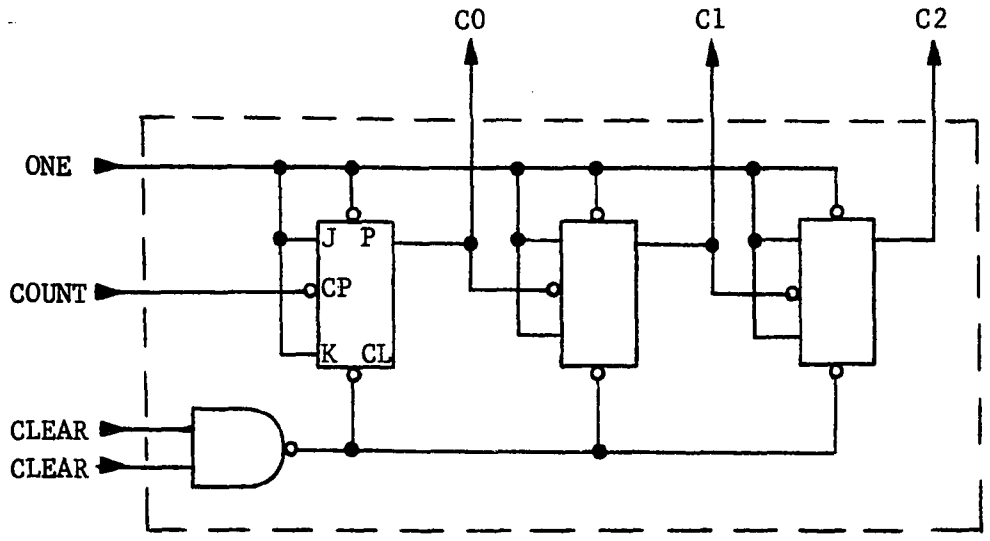


Figure 17. Actual integrated circuit and simulation model for a three-bit ripple counter.

```
 1      *  A COMPLETE SAMPLE RUN ON MACSIM USING A 3-BIT RIPPLE COUNTER
 2
 3      *  DEFINE THE MACROS
 4      DEF_MAC;
 5         (A, B, C) (CP, CL1, CL2, PR) SN5493;
 6            DRIVE = 10, 10, 10;
 7            LOAD = 1, 1, 1, 9;
 8         (A, AB) (PR, PR, CP, NCL, PR) JKFF;
 9         (B, BB) (PR, PR, A , NCL, PR) JKFF;
10         (C, CB) (PR, PR, B , NCL, PR) JKFF;
11         (NCL) (CL1, CL2) $NAND;
12            DEL_0 = 8;
13            DEL_1 = 12;
14      END_MAC;
15
16      DEF_MAC;
17         (Q, QB) (J, K, CLK, CLR, PRT) JKFF;
18            DRIVE = 10, 10;
19            LOAD = 1, 1, 1, 1, 1;
20         (Q, QB) (J, K, CLK, CLR, PRT) $JKFF;
21            DEL_0 = 25;
22            DEL_1 = 16;
23            MIN_CLK = 50;
24            MIN_CLR = 50;
25      END_MAC;
26
27      *  DEFINE THE NETWORK
28      DEF_NET;
29         (C0, C1, C2) (COUNT, CLEAR, CLEAR, ONE) SN5493;
30      END_NET;
31
32      *  GENERATE THE CROSS-REFERENCE LISTING
33      CRS_REF;
```

Cross-reference listing would normally appear here.

```
34      *  DEFINE THE SIMULATION CONTROLLER
35      DEF_SIM;
36         HEADING (COUNT, C0, C1, C2, CLEAR);
37         ONE = 1;
38       L1: CLEAR = 1 (0, 50);
39         COUNT = C (0, 100, 150, 200, 100);
40         TRACE IF ( MOD(TIME, 10) = 1 );
41         GOTO (L1) IF (TIME <= 950);
42         STOP;
43      END_SIM;

END OF NETWORK AND SIMULATOR COMPILATION
```

CROSS-REFERENCE LISTING

LINE-NODE NAME	SOURCE LINE	DRIVE	REM. DRIVE	GENERATING SIGNALS LINE-NODE NAME	LOAD	NET LOAD	GENERATED SIGNALS LINE-NODE NAME
1-CLEAR					1		3-C0 4-C1 5-C2
					1		3-C0 4-C1 5-C2
			-2			2	
2-COUNT					1		3-C0 4-C1 5-C2
			-1			1	
3-C0	29	10		1-CLEAR 1-CLEAR 2-COUNT 6-ONE			
			10			0	
4-C1	29	10		1-CLEAR 1-CLEAR 2-COUNT 6-ONE			
			10			0	
5-C2	29	10		1-CLEAR 1-CLEAR 2-COUNT 6-ONE			
			10			0	
6-ONE					9		3-C0 4-C1 5-C2
			-9			9	

SOURCE LISTING.

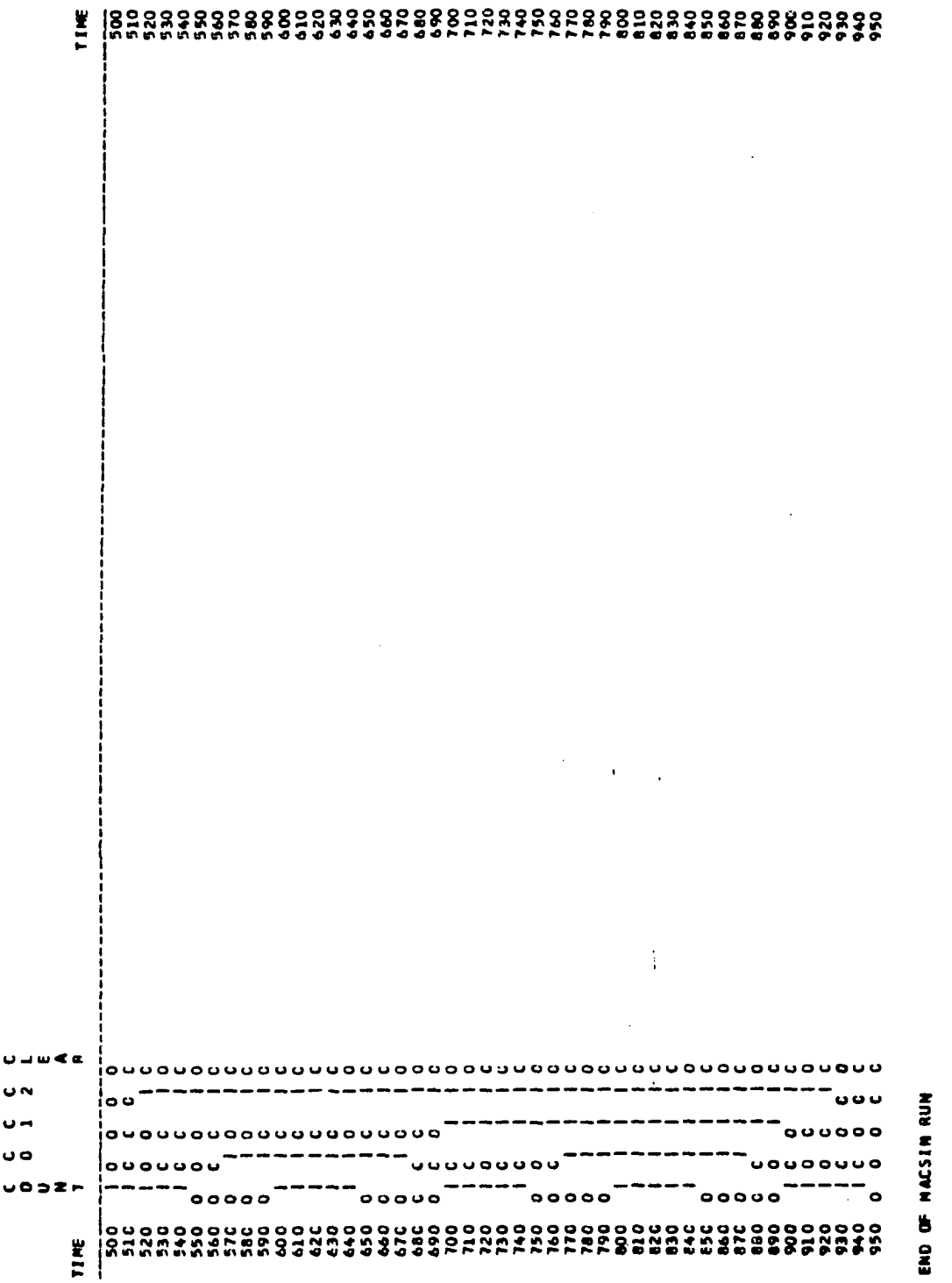
```
                    /*  "MACSIM2" --- NETWORK SIMULATOR  ($PROGRM)      */     1
                                                                               2
1            /*************************/  $PROGRM:  /*****************************/     3
             PROCEDURE RECURSIVE;                                              4
2               DCL 1 $NODE($NDSIZE) CONTROLLED EXTERNAL,                      5
                      2 LAST FIXED BIN,                                        6
                      2 NDX FIXED BIN,                                         7
                      2 VAL BIT(1);                                            8
3               DCL ($NAME($NDSIZE) CHAR(16) VARYING CONTROLLED,               9
                    $UPTR($NDSIZE+1) FIXED BIN CONTROLLED,                    10
                    $USE($USSIZE) FIXED BIN CONTROLLED,                       11
                    $IPTR($IPSIZE) FIXED BIN CONTROLLED,                      12
                    $INPUTS($INSIZE) FIXED BIN CONTROLLED,                    13
                    ($NDSIZE, $USSIZE, $IPSIZE, $INSIZE) FIXED BIN) EXTERNAL; 14
4               DCL (($TTIME, TIME) FIXED BIN INITIAL (0),                    15
                    STABLE BIT(1) INITIAL ('0'B),                             16
                    $CEFCHK BIT(1) INITIAL ('1'B)) EXTERNAL;                  17
5               DCL ($MAX1, $MAX2) FIXED BIN EXTERNAL;                        18
6               DCL @ADDR BIT(15) STATIC,                                     19
                    @I FIXED BIN STATIC,                                      20
                    $LAB FIXED BIN,                                           21
                    $OUT BIT(1) STATIC;                                       22
                                                                              23
7               DCL $NETWRK ENTRY (FIXED BIN),                                24
                    $CLEAR  ENTRY (,,,, FIXED BIN, FIXED BIN, FIXED BIN),     25
                    $PRESET ENTRY (,,,, FIXED BIN, FIXED BIN, FIXED BIN),     26
                    $JKFF   ENTRY (,,,,,,,, FIXED BIN, FIXED BIN, FIXED BIN), 27
                    $DFF    ENTRY (,,,,,,, FIXED BIN, FIXED BIN, FIXED BIN,   28
                                   FIXED BIN, FIXED BIN),                     29
                    $HEAD1  ENTRY ((*) FIXED BIN),                            30
                    $HEAD2  ENTRY,                                            31
                    $COMNT  ENTRY (CHAR (*)),                                 32
                    $TRACE  ENTRY ((*) FIXED BIN, (*) FIXED BIN),             33
                    $ERRCR  ENTRY (FIXED BIN, CHAR(16) VAR),                  34
                    $CHED   ENTRY (, BIT(1), FIXED BIN, FIXED BIN),           35
                    $IMUL8  ENTRY;                                            36
                                                                              37
             /*****    THE FOLLOWING ARE THE USER'S DECLARATION STATEMENTS    *****/  38
                                                                              39
                                                                              62
8               DCL 1 $OCO$OOC LIKE $NODE BASED ($P1),                        63
                    1 $OO1$OOO LIKE $NODE BASED ($P2),                        64
                    1 $CO2$OOC LIKE $NODE BASED ($P3),                        65
                    1 $CO2$OOC LIKE $NODE BASED ($P4),                        66
                    1 #CO LIKE $NODE BASED ($P5),                             67
                    1 #C1 LIKE $NODE BASED ($P6),                             68
                    1 #C2 LIKE $NODE BASED ($P7),                             69
                    1 CLEAR LIKE $NODE BASED ($P8),                           70
```

```
                1 COUNT LIKE $NODE BASED ($P9),                                  71
                1 C0 LIKE $NODE BASED ($P10),                                    72
                1 C1 LIKE $NODE BASED ($P11),                                    73
                1 C2 LIKE $NODE BASED ($P12),                                    74
                1 ONE LIKE $NODE BASED ($P13);                                   75
 9            DCL ($P1, $P2, $P3, $P4, $P5, $P6, $P7, $P8,                       76
                  $P9, $P10, $P11, $P12, $P13)                                   77
                    POINTER STATIC;                                              78
10            DCL $L(10) LABEL;                                                  79
11            $NDSIZE = 13;                                                      80
12            ALLOCATE $NODE;                                                    81
13            ALLOCATE $NAME INITIAL                                             82
                 ('$C00$000', '$C01$000', '$002$000', '$003$000',                83
                  '#C0', '#C1', '#C2', 'CLEAR', 'COUNT', 'C0',                   84
                  'C1', 'C2', 'ONE');                                            85
14            $USSIZE = 11;                                                      86
15            ALLOCATE $USE INITIAL                                              87
                 (1, 4, 7, 10, 10, 3, 6, 9, 2, 5,                                88
                  8);                                                            89
16            ALLOCATE $UPTR INITIAL                                             90
                 (0, 1, 0, 0, 0, 0, 0, 4, 6, 7,                                  91
                  8, 0, 9, 12);                                                  92
17            $INSIZE = 17;                                                      93
18            ALLOCATE $INPUTS INITIAL                                           94
                 (13, 13, 13, 9, 13, 13, 13, 13, 10, 13,                         95
                  13, 13, 13, 11, 13, 8, 8);                                     96
19            $IPSIZE = 11;                                                      97
20            ALLOCATE $IPTR INITIAL                                             98
                 (0, 1, 2, 0, 6, 7, 0, 11, 12, 16,                               99
                  18);                                                          100
21            $P1 = ADDR($NODE(1)); $P2 = ADDR($NODE(2));                       101
23            $P3 = ADDR($NODE(3)); $P4 = ADDR($NODE(4));                       102
25            $P5 = ADDR($NODE(5)); $P6 = ADDR($NODE(6));                       103
27            $P7 = ADDR($NODE(7)); $P8 = ADDR($NODE(8));                       104
29            $P9 = ADDR($NODE(9)); $P10 = ADDR($NODE(10));                     105
31            $P11 = ADDR($NODE(11)); $P12 = ADDR($NODE(12));                   106
33            $P13 = ADDR($NODE(13));                                           107
                                                                                 41
34            DO $I = 1 TO $NDSIZE; $NODE($I).LAST = -1;                         42
36                                  $NODE($I).NDX = $I;                        43
37                                  $NODE($I).VAL = '0'B; END;                 44
                                                                                 45
              /***    THE FOLLOWING IS THE USER'S SIMULATION CONTROL PROGRAM  ***/ 46
                                                                                 47
                                                                                108
              /*     HEADING (COUNT, C0, C1, C2, CLEAR);                    */ 110
39              DCL $HLIST10(5) FIXED BIN STATIC INITIAL                        111
                      (9, 10, 11, 12, 8);                                       112
40              $MAX1 = 5; CALL $HEAD1($HLIST10);                               113
42              SIGNAL ENDPAGE (SYSPRINT);                                      114
                                                                                115
              /*     ONE = 1;                                               */ 116
```

```
43                IF (TIME = 0) THEN DO;                                    117
45                   CALL $CHED(ONE, '1'B, 0, 0);                           118
46                END;                                                      119
                                                                            120
                  /*   L1: CLEAR = 1 (0, 50);                          */   121
47                DCL $$VO(1) BIT(1) STATIC INITIAL                         122
                        ('1'B);                                             123
48                DCL $$TO(2) FIXED BIN STATIC INITIAL                      124
                        (0, 50);                                            125
49                TIME = 0;                                                 126
50                L1:                                                       127
51                DO $I = 1 TO 2 WHILE (TIME ¬= $$TO($I)); END;             128
52                IF ($I <= 2) THEN DO;                                     129
54                   CALL $CHED(CLEAR, $$VO(1), 0, 0);                      130
55                   $$VO = ¬$$VO;                                          131
56                END;                                                      132
                                                                            133
                  /*      COUNT = 0 (0, 100, 150, 200, 100);           */   134
57                DCL $$V1(1) BIT(1) STATIC INITIAL                         135
                        ('0'B);                                             136
58                DCL $$T1(4) FIXED BIN STATIC INITIAL                      137
                        (0, 100, 150, 200);                                 138
59                DO $I = 1 TO 4 WHILE (TIME ¬= $$T1($I)); END;             139
61                IF ($I <= 4) THEN DO;                                     140
63                   CALL $CHED(COUNT, $$V1(1), 0, 0);                      141
64                   $$V1 = ¬$$V1;                                          142
65                END;                                                      143
66                IF $I = 4 THEN DO;                                        144
68                   DO $I = 3 TO 4; $$T1($I) = $$T1($I)+100; END;          145
71                END;                                                      146
                                                                            147
                  /*      TRACE IF ( MOD(TIME, 10) = 1 );              */   148
72                CALL $IMUL8; TIME = TIME+1;                               149
74                IF ( MOD(TIME, 10) = 1 ) THEN DO;                         150
76                   CALL $TRACE($HLIST10, $HLIST10);                       151
77                END;                                                      152
                                                                            153
                  /*      GOTO (L1) IF (TIME <= 950);                  */   154
78                IF (TIME <= 950) THEN DO;                                 155
80                   GO TO L1;                                              156
81                END;                                                      157
                                                                            158
                  /*      STOP;                                        */   159
82                   RETURN;                                                160
                                                                             49
                                                                             50
83                $NETWRK: ENTRY ($LAB);                                     51
                                                                             52
84                   GO TO $L($LAB);                                         53
                                                                             54
                                                                             55
                  /***     THE FOLLOWING IS THE USER'S NETWORK DESCRIPTION PROGRAM    ***/   56
```

```
                                                                                                    57
                                                                                                   161
                    /*     (C0, C1, C2) (COUNT, CLEAR, CLEAR, ONE) SN5493;                    */  163
E5                  $L(1):   IF ($001$000.VAL & ¬ONE.VAL) THEN GO TO $L(2);                       164
87                     CALL $CLEAR(C0, $0C0$000, $0C1$000, ONE, 25, 16, 50);                      165
8E                     RETURN;                                                                    166
89                  $L(2):   IF (ONE.VAL & ¬$C01$000.VAL) THEN GO TO $L(1);                       167
91                     CALL $PRESET(C0, $0C0$000, $001$000, ONE, 25, 16, 50);                     168
92                     RETURN;                                                                    169
93                  $L(3):   CALL $JKFF(C0, $C00$000, ONE, ONE, COUNT, $C01$C00, CNE, #C0,     170
                    25, 16, 50);                                                                   171
94                     RETURN;                                                                    172
95                  $L(4):   IF ($001$C00.VAL & ¬ONE.VAL) THEN GO TO $L(5);                       173
97                     CALL $CLEAR(C1, $C02$C00, $001$000, ONE, 25, 16, 50);                      174
98                     RETURN;                                                                    175
99                  $L(5):   IF (ONE.VAL & ¬$C01$00C.VAL) THEN GO TO $L(4);                       176
101                    CALL $PRESET(C1, $002$000, $0C1$000, ONE, 25, 16, 50);                     177
102                    RETURN;                                                                    178
103                 $L(6):   CALL $JKFF(C1, $002$000, CNE, CNE, C0, $001$C00, ONE, #C1,        179
                    25, 16, 50);                                                                   180
104                    RETURN;                                                                    181
105                 $L(7):   IF ($001$000.VAL & ¬CNE.VAL) THEN GO TO $L(8);                       182
107                    CALL $CLEAR(C2, $003$0C0, $001$000, CNE, 25, 16, 50);                      183
108                    RETURN;                                                                    184
109                 $L(8):   IF (ONE.VAL & ¬$001$000.VAL) THEN GO TO $L(7);                       185
111                    CALL $PRESET(C2, $003$000, $C01$000, ONE, 25, 16, 50);                     186
112                    RETURN;                                                                    187
113                 $L(9):   CALL $JKFF(C2, $C03$000, ONE, ONE, C1, $001$C00, ONE, #C2,        188
                    25, 16, 50);                                                                   189
114                    RETURN;                                                                    190
115                 $L(10):  $OUT = ¬(CLEAR.VAL & CLEAR.VAL);                                     191
116                    CALL $CHEC($0C1$000, $OUT, 8, 12);                                         192
117                    RETURN;                                                                    193
                                                                                                    59
118                 END $PROGRM;                                                                    60
                                                                                                    61
```

```
        C  C  C  C  C
        O  O  1  2  L
        U  0           E
        N              A
TIME    T              R                                                                TIME
-------------------------------------------------------------------------------------------
    0   0              |                                                                  0
   10   0              |                                                                 10
   20   0              |                                                                 20
***** CLOCKING OF FLIP-FLOP C1 ATTEMPTED DURING CLEAR OR PRESET OPERATION
***** CLOCKING OF FLIP-FLOP C2 ATTEMPTED DURING CLEAR OR PRESET OPERATION
   30   0  C  C  0    |                                                                 30
   40   0  0  0  C    |                                                                 40
   50   0  C  C  C  C                                                                  50
   60   0  C  0  C  C                                                                  60
   70   0  C  0  C  0                                                                  70
   80   0  C  0  0  C                                                                  80
   90   0  0  0  C  C                                                                  90
  100      C  C  C  C                                                                 100
  110      0  0  0  0                                                                 110
  120      C  C  C  C                                                                 120
  130      0  0  C  C                                                                 130
  140      C  0  C  C                                                                 140
  150   0  C  C  0  C                                                                 150
  160   0  0  C  C  0                                                                 160
  170   0     0  0  C                                                                 170
  180   0     0  C  C                                                                 180
  190   0     0  C  C                                                                 190
  200      I  C  C  C                                                                 200
  210      I  0  C  C                                                                 210
  220      I  C  C  C                                                                 220
  230      I  0  0  C                                                                 230
  240      I  C  C  C                                                                 240
  250   0     0  C  C                                                                 250
  260   0     0  C  C                                                                 260
  270   0     C  C  C                                                                 270
  280   0  C  0  C  C                                                                 280
  290   0  C  0  0  C                                                                 290
  300      C     C  C                                                                 300
  310      C     C  0                                                                 310
  320      C     0  0                                                                 320
  330      C     C  C                                                                 330
  340      0     C  0                                                                 340
  350   0  C     C  C                                                                 350
  360   0  C     0  C                                                                 360
  370   0     I  C  C                                                                 370
  380   0     I  C  C                                                                 380
  390   0     I  0  0                                                                 390
  400      I  I  C  C                                                                 400
  410      I  I  C  C                                                                 410
  420      I  I  C  C                                                                 420
  430      I  I  0  0                                                                 430
  440      I  I  C  0                                                                 440
  450   0     I  C  0                                                                 450
  460   0     I  C  0                                                                 460
  470   0     I  C  C                                                                 470
  480   0  0     0  C                                                                 480
  490   0  C     C  C                                                                 490
```


TIME
500 510 520 530 540 550 560 570 580 590 600 610 620 630 640 650 660 670 680 690 700 710 720 730 740 750 760 770 780 790 800 810 820 830 840 850 860 870 880 890 900 910 920 930 940 950
C O U N T
C C C C L E A R
TIME
500 510 520 530 540 550 560 570 580 590 600 610 620 630 640 650 660 670 680 690 700 710 720 730 740 750 760 770 780 790 800 810 820 830 840 850 860 870 880 890 900 910 920 930 940 950
END OF NACSIM RUN