Digital logic simulation.

George Sotlrlos Maroudas

Follow this and additional works at: http://preserve.lehigh.edu/etd
Part of the Electrical and Computer Engineering Commons

Recommended Citation
DIGITAL LOGIC SIMULATION
AND
MACRO PROCESSING

by

George Sotirios Maroudas

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Electrical Engineering

Lehigh University
1975
This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

May 9, 1975
(date)

Professor in Charge

Chairman of Department
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>v</td>
</tr>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>Preface</td>
<td>2</td>
</tr>
<tr>
<td>Chapter 1: DIGITAL LOGIC SIMULATION</td>
<td></td>
</tr>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Digital Logic Simulators</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Application of Simulation Techniques</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Types of Digital Logic Simulators</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Fault Analysis Simulation</td>
<td>21</td>
</tr>
<tr>
<td>Chapter 2: IMPLEMENTATION OF A DIGITAL SIMULATOR</td>
<td></td>
</tr>
<tr>
<td>2.1 The Digital Logic Simulator DLS</td>
<td>26</td>
</tr>
<tr>
<td>2.2 Capacity of DLS</td>
<td>26</td>
</tr>
<tr>
<td>2.3 Data Structure</td>
<td>26</td>
</tr>
<tr>
<td>2.4 Basic Element Set</td>
<td>29</td>
</tr>
<tr>
<td>2.5 Time Mechanization</td>
<td>30</td>
</tr>
<tr>
<td>2.6 Propagation Delay</td>
<td>30</td>
</tr>
<tr>
<td>2.7 Signal Values</td>
<td>32</td>
</tr>
<tr>
<td>2.8 Synchronous and Asynchronous Simulation</td>
<td>32</td>
</tr>
<tr>
<td>2.9 DLS Evaluation</td>
<td>34</td>
</tr>
</tbody>
</table>

- 111 -
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Circuit generated by the Boolean processor for A</td>
<td>10</td>
</tr>
<tr>
<td>2.</td>
<td>Timing considerations in simulation</td>
<td>16</td>
</tr>
<tr>
<td>3.</td>
<td>Parallel Fault Simulation</td>
<td>24</td>
</tr>
<tr>
<td>4.</td>
<td>Data Tables in DLS</td>
<td>27</td>
</tr>
<tr>
<td>5.</td>
<td>Delay scheme in DLS</td>
<td>31</td>
</tr>
<tr>
<td>6.</td>
<td>Recursive macros</td>
<td>47</td>
</tr>
<tr>
<td>7.</td>
<td>5-bit Asynchronous counter</td>
<td>89</td>
</tr>
<tr>
<td>8.</td>
<td>Space iterative circuit</td>
<td>92</td>
</tr>
</tbody>
</table>

### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Logic elements in DLS</td>
<td>29</td>
</tr>
</tbody>
</table>
Abstract

Two distinct but not exclusive topics are included in this thesis. The first part of the material deals with Digital Logic Simulation. A concise survey of the current trends in digital logic simulation is presented together with a particular example of a digital simulator. This survey covers the main concepts on which simulation is based today.

The second part of the thesis deals with macro programming. As in the first part, an introduction is presented in the beginning and then the design and implementation of a macro processor is included. The particular macro processor developed here is of general use and powerful enough to allow most of the desired features of macro programming.

Finally, examples of the application of the macro processor are given to illustrate that the latter can be used not only with assemblers, but also for extending the capabilities of digital logic simulators.
Preface

The purpose of undertaking the tasks dealt with in this thesis was twofold. I wanted to be exposed to modern problems in both hardware and software systems.

Digital logic simulation is the development of software systems with the goal of optimizing hardware. The survey in digital logic simulation presented in Chapter 1, as well as the particular simulator discussed in Chapter 2, gave me the opportunity to be involved in both areas of interest. So did the analysis and synthesis of a macro processor presented in Chapters 3 and 4, respectively. The development of a macro processor, whose purpose is to make hardware systems more convenient to (software) programmers, served well my original expectations. The interesting applications of macro modules, presented at the end of this thesis, show how macro processing can be used as a tool in digital logic simulation, not only for substituting MSI elements, but also for generating iterative circuits of variable length.

Dealing with the above topics has proved to be quite enlightening, as well as rewarding, in both the areas of hardware and software.
Chapter 1

Digital Logic Simulation

Introduction

The basic idea underlying computer simulation is to transform a real system into some mathematical model that the computer can operate on. The particular case of digital logic simulation is no exception to this principle. It refers to modeling a digital logic circuit into some computer executable form. A digital logic simulator is a computer program that maps the real circuit on to some form that the computer can process appropriately. An ideal model simulating a circuit would be one whose behavior matches exactly the behavior of the real circuit. It is not possible to construct such a model. For most practical cases though, less precise models are applicable without excessive loss in accuracy. Preciseness, as well as efficiency, are of prime importance in modeling a real circuit with the aid of computers. The best theoretical model, badly implemented, can be of no practical use. Unfortunately the reverse is also true.

1.1 Digital Logic Simulators

The digital logic simulator is a program written in some computer language. Its input is a digital logic cir-
circuit description coded in some appropriate form by the user. The simulator translates the input circuit description into some internal form. Once the compilation phase is completed, the model is exercised according to some external input sequence specified by the user. This is the execution phase of the simulation. Hence, an input-output behavior of the real circuit is generated by the computer rather than by the actual components of the circuit.

1.2 Application of Simulation Techniques

The main goal of a digital logic simulator is to remove the user from the need for a circuit and enable him to work with a more convenient model. The verification of the user's circuit by the computer is both cheaper and faster. Small errors in the original design may readily be uncovered and corrected without rebuilding the circuit. The logic designer works in a very efficient environment. Fault detection analysis is now available with the computer being the oscilloscope of the designer.

Generally, one would use digital logic simulation for one of the following purposes:

1. To check logic before commitment to hardware.
2. To test alternative designs and compare efficiency.
3. To obtain detailed reference data for a circuit.
4. To verify fault test procedures.
Fault test procedures include:
a. Analysis of circuits in the presence of faults.
b. Generation of fault detection tests.
c. Verification and evaluation of fault detection tests.
d. Generation of detailed fault dictionary for a circuit.

1.3 Types of Digital Logic Simulators

There are various techniques the designer of a digital simulator may employ in developing his program. Usually the prospective application of the simulator plays an important role in the designer's decision. What follows is a concise introduction of the various options available to the designer of a modern digital logic simulator.

1.3.1 Data Structures

There are generally two ways of forming a data base for storing all the necessary information about a circuit:
a. Compiled-code method.
b. Table data base.

In the compiled-code method, the circuit is compiled directly into computer executable code which performs the Boolean function realized by the circuit. If the simulator is to be used for pure logic verification where timing
considerations are of no interest, then this is an efficient technique because of its simplicity. Efficiency may be further increased by proper rearrangement of the compiled code or readily applied code simplifications. If this technique is to be used for sequential circuits, the elements should be sorted in levels with proper breaking of feedback lines. Hence, a sorting processor should be available.

The compiled-code simulation is of little, if any, use today because of its inherent drawbacks. Due to the way it mechanizes time, only synchronous simulation is possible. This is very restrictive and such a method should not be implemented unless a very simple and specialized simulator is desired.

Table-driven simulators are by far the dominant type of simulators used today. Their data base consists of tables which store all the pertinent information about the circuit. The tables should be concise, with no redundant information. At the same time, they should be easy to access. Main considerations are speed and storage. It should be emphasized at this point that the performance of the simulator will greatly depend on the structure and implementation of the data tables. The designer should be aware of the hardware of the host computer since the form
of his tables may depend on it. It is generally a good approach to design a simulator, which may cost several hundred thousand dollars, in a way such that it will not depend on any particular computer. The resulting program is then more "transportable", but generally its efficiency is thereby decreased. Hence, the designer usually limits his program to a particular "family" of computers and takes into account their hardware characteristics.

Digital logic is currently based on table-driven techniques. They can handle both synchronous and asynchronous circuits. Also, small changes in the original circuit do not necessarily imply recompilation of the whole circuit, since a few index changes in the tables may take care of these modifications.

1.3.2 Elements Used in the Simulator

In the discussion that follows, some non-standard terms are used, which do not appear in the literature. Although they should be self-explanatory, their definition is presented below to avoid possible misinterpretation.

Basic or primitive element: A logic element that may be simulated without being translated in terms of other elements.

Basic element set: The set containing all the basic elements of the simulator.
Capacity of a simulator: Maximum size of acceptable circuit to be simulated. The capacity of a simulator is usually given in terms of number of elements in the circuit and/or number of leads. The number of total inputs or outputs of the circuit may also be included.

Resolution of a simulator: Accuracy or precision of the simulator.

A simulator is built in such a way that it understands only certain basic logic elements. The designer usually decides which logic elements his simulator should be able to recognize. He then supplies his simulator with the computer code which describes the behavior of these basic units. For an OR gate for example, he would specify in his code that the output of the gate would have the value of 0 iff none of its inputs have the value 1. In this way, the computer will be able to simulate the OR gate correctly.

Once the basic elements are defined, the computer will be able to simulate their behavior only. Consider the case in which the basic elements are the AND, the OR and the NOT gates. If these are the only gates the simulator understands, then every circuit to be simulated should be written in terms of OR's, AND's and NOT's. Although this is theoretically possible, it presents some practical prob-
lems. The user will have to write every component of his circuit in terms of OR, AND and NOT gates. This is not an easy task for the user. Even if the computer would do it for him, the resulting circuit would include many gates.

An obvious solution to the above problem is to increase the number of elements in the basic element set. NAND and NOR gates should be included. Flip-flops should also be implemented as basic elements, so that it would not be necessary to write them in terms of combinational logic. Thus, an adequately broad set of basic elements should be one of the designer's main considerations.

Although the efficiency of the simulator is increased by the expansion of the basic element set, the user may desire to work with higher-level elements such as counters, registers, etc., that are not included in the basic set. This often occurs when the user is simulating large circuits. Without changing the internal structure and the operation of his simulator, the designer can implement some programming techniques to make it appear more powerful from the user's point of view.

The first technique is to have a Boolean preprocessor operate on the user's input circuit to the simulator. The Boolean preprocessor should be able to read and understand Boolean equations, and generate the corresponding circuit.
Thus, let the user input a desired Boolean equation rather than building and inputing the corresponding circuit. The generation of the circuit would be the task of the preprocessor. Thus, if the user wanted lead $A$ to be $A=(B+C)D+E$, he would specify so directly. The preprocessor would generate the corresponding circuit shown in Fig. 1 below and would input it to the simulator.

![Circuit Diagram]

Figure 1: Circuit generated by the Boolean processor for $A$.

This technique helps the user to decrease his input considerably, and allows him to think in terms of Boolean relations rather than circuit elements. It would be a useful tool for the logic designer who wants to test his circuit in its Boolean-equation form. It should be noted that this technique does not increase the final storage and speed efficiency of the simulator.
Another programming technique that the designer of the simulator should consider is the feature of macro-processing. The idea in macro-processing is to let the user build new elements he wants to insert in several places in his circuit and which are not available in the simulator. These elements should only once be described in terms of the elements in the simulator. Once this is done, the user may insert them in his circuit in the same way he inserts the primitive elements provided by the designer.

This technique saves considerable time for the user. He defines a new logic block only once, rather than every time he uses it. Now the user may think in terms of larger blocks like counters and registers. The designer may even provide a library of commonly used elements so that the user can employ them directly, without having to define them.

It should be noted that the capabilities of the simulator seem to have increased since it understands more elements than before. This is true from the user's point of view. For the designer of the simulator, however, no additional changes to his program are necessary. A preprocessor is needed to translate the user's blocks in terms of the primitive elements of the simulator. The program performing this type of processing is usually called a
macro-processor. One such program was developed by the writer and is described in Chapter 4. It is used as a preprocessor of a particular logic simulator for extending the latter's applicability. The reader should be cautioned that careful use of logic blocks (macros) is recommended. It is very easy to surpass the capacity of the simulator by using only a few large blocks, since one such block may consist of hundreds of primitive elements.

Finally, a third programming technique is available which overcomes the storage problem of the macros discussed above. The designer originally includes in his basic set those elements he considers necessary. Then, he allows the user to include new elements by describing them in computer code rather than primitive gates as in the case of macros. The description of the new elements is similar to the one that the designer used to implement his, i.e., computer executable code that performs the Boolean function of the logic element. This approach of letting the user describe the behavior of a large logic element in terms of computer instructions is usually referred as "functional simulation". In the functional simulation, the basic element set of the simulator is left open to the user to insert his own logic elements.

This technique is more efficient compared with the use
of macros since both storage and execution time are greatly reduced. One large logic block may be treated as a single element, possibly with multiple inputs and/or outputs. The use of an 'equivalent' macro might take considerable storage and its processing time would be significantly longer. The main disadvantages of functional simulation are:

1. An additional requirement is imposed on the designer to make the data structure of his simulator modular enough so that it may accommodate a wide range of user specified elements. This imposes a big problem on the designer, who is obliged to think in terms of general elements rather than some fixed ones with known characteristics such as number of inputs and outputs. He may have to decrease his simulator's efficiency and/or impose restrictions on the user's new functional elements.

2. The user must become familiar with the simulator. He can no longer use it completely as a black box. He has to write the proper computer instructions in the proper language. This is generally a difficult task for the user, harder than writing a macro to do the same job.

3. Timing considerations of the functional elements should be taken into account by the user. With macros this is not, generally, necessary since individual gate delays
are inherent within the logic block. The user of functional elements has to specify delay times for his newly defined block. For large elements, this is a tedious task to impose on the user.

The reader should also notice that the use of functional elements decreases the accuracy of the simulation, since the fine gate structure of the circuit is no longer preserved. Hence, the simulator's resolution is decreased with the addition of functional elements. In demanding simulation applications such as in Bell Labs, different types of simulators are used during the development of a system. A functional simulator is used to verify initial system design expressed in terms of registers, memories, etc., where gate-level logic is not available yet. Once the circuit is verified to work properly on the functional level, its gate-level description is tested by a lower-level but more accurate gate simulator.

To summarize the above ideas, the following guidelines should be considered by the designer of a modern simulator:

a. For simulating small circuits, a basic element set with the standard logic gates and flip-flops should be sufficient for the simulator.

b. For medium-to-large circuits, an enlarged version
of the simulator in part (a) above should be used together with a macro facility. The simulator's capacity should be increased and new elements can be added, either as primitive ones in the basic set, or as defined macros in a macro library.

c. For handling large circuits, functional simulation is necessary. Finally, the Boolean preprocessor is a practical accessory applicable in all simulator schemes.

1.3.3 Time Mechanization

A very important decision for the designer of a digital logic simulator is how to mechanize time in his program. The effectiveness of his simulation, both in correctness of modeling and in speed, greatly depend on the time-flow mechanism.

Because of inherent characteristics of computer operation, only one (or very few) elements may be evaluated at a time. The designer would like to avoid having a gate with an updated input and a non-updated one, as shown in Fig. 2.
One way to solve the problem is to arrange the input circuit into cascade levels of logic. This is a rather primitive technique though, being both time-consuming and inaccurate.

Another way is to store more than one value for each element. This is usually called state variable approach. In the example shown in Fig. 2, if all gates had a delay of one time unit, then two values would be necessary for each lead: the new value and the old value. The new value of the output lead of a gate is evaluated from the old values of its input leads. Thus, the inconsistency illustrated in Fig. 2 is avoided. Although the state-variable method offers a solution to the above problem, it creates a new one by requiring multiple values for each lead in the circuit. It is a straightforward time-flow mechanism, though, and, if efficiently implemented, it may be
practical. This is clearly illustrated in its application to the DLS program, discussed in Chapter 2.

Quantization of time into discrete units is generally employed in all simulation schemes. Signals may change only at discrete time instants. A fixed-increment model would simply increment time by a fixed unit (of relative size), and at each increment, it would determine whether or not any activity is to take place. Another model employs the next-event mechanism, where the simulator proceeds from one scheduled event to the next. Scheduling of events is done according to changes in element inputs and propagation delays assigned to the elements in the circuit. An event is defined to be that time instant when an output line of an element (or elements) is to change value. Both of the above techniques have implementation drawbacks. The fixed-increment model requires extensive tables and element evaluations. The next-event model needs to store considerable data for each event. Modern simulation systems employ various combination schemes of these two models in an effort to achieve maximum efficiency in speed and storage.

Some simulators use the selective trace technique to improve even more the overall efficiency. This technique is based on the principle that, if an element's output does not change when the inputs are considered, then the fan-out
of that element is unaffected by the excitation that caused the evaluation. Hence, updating of elements is substanti-
tially reduced.

1.3.4 Propagation Delay of Elements

A major problem that the designer faces is what types of delays he should allow for the basic elements of his sim-
ulator. The more realistic the delay model, the more com-
plex his implementation becomes. Thus, it is again a mat-
ter of compromising between accuracy and efficiency. The use of the simulator is the most important consider-
ation. Zero and unit delays are easily implemented, but such sim-
ulators are restricted to logic verification and are not suitable for design verification.

The design verification needs a more precise delay scheme. Assignable delays should be considered. This mod-
el permits average gate delays which may be a multiple of a unit delay. Assignable delays result in a much more pre-
cise modeling of the circuit. With such delay schemes, spike analysis as well as a limited amount of hazard detec-
tion can be accomplished.

More refined delay models are also employed in some simulators. These allow the user to specify a time range in which the device may switch value. More precision is achieved in some simulators by the addition of different
rise and fall time, so that the delay depends on whether the signal transition is from 0 to 1 or from 1 to 0. With each additional refinement, a more precise model is achieved, but complexity of the program increases and its speed suffers.

1.3.5 Signal Values

Signal values in real digital logic circuits can be either 0 or 1. The designer of a digital simulator should decide whether his simulator should use only these two values, or more. It is usually his choice on the delay scheme that determines his decision here. For a zero, unit or assignable delay models his choices are either two-valued or three-valued simulation. Two valued simulation is considered obsolete by today's standards because of serious drawbacks inherent in its implementation, which come about from initialization and consistency problems.

These problems can be resolved to a large extent by the addition of a third value X, where X stands for 'inde-terminate' or 'don't-know'. (In the LAMP system four values are used: 0, 1, 2, and 3. Values 0 and 1 are the regular 0 and 1 of Boolean algebra. Values 2 and 3 represent non-propagating and propagating "don't-know" conditions, respectively. Value 2 is used solely to allow efficient
initialization of the circuit. At the beginning of a simulation run all gates are initially set to the value 2.
The non-propagation property is necessary to prevent destroying a prespecified initial state of the circuit. Value 3 is a propagating "don't know" representing indeterminate propagation signals. It corresponds to the X value in the previously discussed three-value simulator.

In the case of simulators with more precise delay schemes (minimum-maximum range) a six-valued simulation could be used: 0, 1, X, U (signal rising), D (signal falling), E (potential spike, hazard or race).

1.3.6 Synchronous or Asynchronous Simulation

The terms synchronous and asynchronous simulation refer to the manner in which timing considerations and responses are modeled. In synchronous simulation each operation takes place under the control of a clock. In asynchronous simulation, the signal values of each element are available as soon as state changes occur and not on fixed clock times.

Synchronous simulation, although simpler and faster, because fewer events are to be accounted, is obsolete today, since it is restricted to synchronous circuits and it is associated with techniques no longer favored, like com-
piled-code simulation and zero or unit delay schemes. On the other hand, asynchronous simulation is preferred, being capable of simulating both synchronous and asynchronous circuits.

The designer of a modern simulator should implement asynchronous simulation since it is more general than synchronous. It can be used both for logic and design verification, whereas synchronous simulation is restricted to logic verification with less rigid time flow mechanism.

1.4 Fault Analysis Simulation

The important role of digital logic simulation in fault analysis necessitated the development of specialized simulators for this purpose.

If digital simulation is the process of modeling the behavior of a real circuit by the computer, digital fault simulation is the modeling of a digital circuit in the presence of faults. There are some standard forms of faults that are usually simulated, such as stuck-at faults and shorted leads, but for our purpose a fault will generally indicate some physical defect in the circuit.

There are two classes of simulators, not necessarily exclusive: the fault-free or true-value simulators, and the fault simulators. The latter are more general than the
true value simulators since they are capable of handling both fault-free and faulty circuits. True-value simulators may sometimes be used for fault analysis, but their application is both limited and inefficient, as we shall see.

Fault simulators need additional data bases to keep track of special activities such as fault specifications, fault propagation, fault insertion and fault detection. A fault simulator can be used for logic verification and design verification. Its prime application, though, is verification and evaluation of fault diagnostic tests. There are generally three schemes of fault simulators:

a. single fault simulators
b. the deductive fault simulators
c. parallel fault simulators

The single fault simulators are similar to the fault free ones. A single fault at a time is inserted into the circuit and simulation begins. This scheme is very inefficient and slow. Unless a small number of faults is to be examined, or only a fault-free simulator is at hand, single fault simulation should not be considered.

The deductive method of simulation for fault analysis was first introduced by Armstrong. This technique explicitly simulates the behavior of the fault-free logic and
simultaneously deduces all faults that can be detected by a given input vector.

The third type of fault simulators make use of the parallel simulation concept. In this scheme M copies of the circuit are simulated on each run, the fault free one together with ones in which faults are inserted. M is usually the word length of the host computer. Each bit of a word corresponds to a signal value in one of the M versions of the original circuit. Faults of interest are inserted in each copy by using appropriate fault masks. In the CDC-6000 series of computers with a word length of 60, up to 60 copies of the same circuit may be simulated on each run. In Figure 3 below, bit 0 would hold the true value of lead Y in the fault free circuit M0. Then, to simulate circuit M1 which is the same as M0 but whose Y is s-at 0, the first mask shown would be used. The second mask would be used for simulating a circuit M2 with Y s-at 1. The first mask (for s-at-0 faults) is ANDed with the correct value of Y. The second mask (for s-at-1 faults) is ORed with Y.
Parallel simulation, unlike deductive, is not restricted to single faults. It may easily be extended to multiple faults by using more than one mask for each circuit. It should also be noticed that the scheme of parallel simulation is not limited to fault analysis only. Each bit of the computer word could correspond to the same fault free circuit with a distinct input vector. Hence, up to $M$ different input vectors can be tested on the same circuit on only a single run. This increases considerably the efficiency of the simulation process.

More than one of the above schemes may be used during fault analysis. Bell's LAMP simulator, for example, in-
cludes a distinct true value simulator for fault free circuits. It also includes a fault simulator to test circuits with classical stuck-at faults. There is a separate simulator, similar to the one above, but which employs the technique of parallel fault simulation. Finally, there is still another simulator which allows testing of circuits with non-classical faults, such as shorts between adjacent paths and crossovers.
Chapter 2

IMPLEMENTATION OF A DIGITAL SIMULATOR

2.1 The Digital Logic Simulator DLS

In this Chapter, we consider a simulation program called DLS (Digital Logic Simulator), which is a Fortran program written at Air Force Institute of Technology. The simulator was completed at the end of 1971 and, hence, does not include many of the modern features discussed in Chapter 1.

2.2 Capacity of DLS

The available version of DLS can simulate circuits consisting of up to 450 logic elements. The maximum allowable number of distinct leads in the circuit is 675. There may be up to 154 external inputs. Signals on up to 25 leads may be printed on each run. Some of these upper bounds can be readily extended.

2.3 Data Structure

DLS is a table-driven simulator. It employs four data tables: LAB, LL, LLT and OUTPUT, as shown in Fig. 4.

LAB is a 675x1 array containing the CDC alphanumeric codes for all signal names in the circuit. The element output labels are stored in the first portion of the table, and the external inputs are stored in the second portion.
Figure 4: Data tables in DLS
Each entry of LAB contains one lead label.

The Value Table LL is a 675x2 array. LL is parallel to LAB and contains two entries for each label. Column one of LL contains the delay register and column two contains the delay mask for each signal. If the latter has a delay of n time units, the n+1 first bits of the 60-bit word are used. The left-most bit 0 contains the current output value of the signal evaluated n times earlier. The value calculated at the current time increment is entered into bit n. The delay mask is used to enter the new value into the proper bit of the delay register for each signal. A signal with delay n would have a delay mask of all 0's with 1 only in the n'th bit. This will become clearer when the delay scheme of DLS is considered.

The element table ELT is a 450x10 array. Element outputs, element types properly coded, and element inputs are stored in ELT. Outputs and inputs are stored as pointers to corresponding LAB entries.

Finally, table OUTPUT is a 25x1 array storing the labels whose values are to be printed. The elements of OUTPUT are also indices to the corresponding entries in LAB.
2.4 Basic Element Set

The available elements in the basic element set of DLS are:

Table 1: Logic elements in DLS

<table>
<thead>
<tr>
<th>Combinational Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>AND</td>
</tr>
<tr>
<td>OR</td>
</tr>
<tr>
<td>INV</td>
</tr>
<tr>
<td>NAND</td>
</tr>
<tr>
<td>NOR</td>
</tr>
<tr>
<td>EOR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flip-Flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>TFF</td>
</tr>
<tr>
<td>JKFF</td>
</tr>
<tr>
<td>RSFF</td>
</tr>
<tr>
<td>RSFFU</td>
</tr>
<tr>
<td>RSTFF</td>
</tr>
<tr>
<td>DFF</td>
</tr>
<tr>
<td>JKFFN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>CLOCK</td>
</tr>
<tr>
<td>DELAY</td>
</tr>
<tr>
<td>SWITCH</td>
</tr>
</tbody>
</table>

The code numbers correspond to the internal representation of the elements in the ELT table.
2.5 **Time Mechanization**

Time is quantized into discrete time units. Time increments are of relative size; the user may interpret them as nanoseconds, microseconds, etc. All gates are evaluated at each time increment. The state-variable approach scheme discussed in Chapter 1 is used. Because more than one unit delay is allowed, the present output value of combinational elements is a function of their inputs at n time units earlier. For sequential elements with delay n, the output at time $t_1$ is a function of the inputs at $t_1-n$, and the state at $t_1-n$. These relations are formally stated below:

Combinational logic: $y(t_1) = f(x(t_1-n))$

Sequential logic: $y(t_1) = f(x(t_1-n), y(t_1-n))$

Both synchronous and asynchronous simulation is possible with this timing scheme.

2.6 **Propagation delays**

Assignable delays are available in DLS. These are multiples of a unit delay corresponding to a delay of one time increment. The user may specify an integral number of delay units for each element. A default value of 1 is automatically assumed for unspecified delays. Consider the case in which the user assigns a delay of 5 time units for an element with output lead X. The delay mask for X would
have a 1 in bit 5 as shown in Fig. 5 below:

![Delay Mask for X](image)

![Delay register for X](image)

Figure 5: Delay scheme in DLS

The present value of X is determined from bit 0. In the above example, X computed from its present inputs, is entered into bit 5. The delay mask is used to enter the present value in the correct bit of the delay register. On each time increment, the delay register is shifted to the left by one bit. So the entered value of X will show up at the output (bit 0) after 5 time increments. Thus a delay of 5 time units is correctly simulated. The maximum assignable delay is 59, limited by the word length of the CDC-6600 computer. It should be noted that, although the state variable approach was termed impractical in Chapter...
1, this particular implementation here is fast and straightforward. Very few computer operations are performed. Storage is not wasted because only one word per label is used. Finally, the propagation of time simply takes the form of a single shift of the entries in LL(1). This updating scheme by single shifts is both efficient and conceptually simple. This illustrates the case in which a generally bad technique, when efficiently implemented, may be practical.

2.7 Signal Values

Only two values, 0 and 1, are allowed in DLS. The don't know value X is not employed, neither can it be easily implemented without changing significantly the data structure of the simulator. This presents a very serious drawback of the program, especially if it is to be used for fault analysis.

2.8 Synchronous and Asynchronous Simulation with DLS

DLS can simulate both synchronous and asynchronous circuits. It includes four modes of operation: Gate, Event, Synchronous and Asynchronous. In the beginning of his simulation the user specifies under which mode he wants to test his circuit. Gate mode is the basic mode of opera-
tion in DLS. Circuit inputs are examined every time increment and all elements are evaluated. With Gate mode, the user can investigate the fine structure of signals in his circuit. Event mode is similar to Gate mode with some provisions to reduce the output that is printed by the computer. Synchronous mode is basically used to verify transitions of synchronous sequential circuits. Flip-flops are not evaluated unless the combinational logic becomes steady. Once this has been achieved, the state of the circuit is updated and the next input is read. Thus, to the user, it appears that the input and the flip-flops are ANDed with a master clock internal to the simulator which goes to 1 whenever the combinational logic becomes stable. The Asynchronous mode differs from the Synchronous one, in that flip-flops are now evaluated every time increment regardless of whether the combinational logic is stable or not. The next input is read after the circuit is stabilized, i.e. enough time is automatically provided by the simulator to the input to propagate its effects through the circuit and settle it to a new state before the next input is read.

Synchronous and Asynchronous modes simplify the testing of sequential circuits. A master clock in Synchronous mode, as well as continuous checks for circuit stability in both modes are implemented in the simulator and pro-
vided to the user.

2.9 DLS Evaluation

The digital logic simulator DLS does not contain many of the desired features included in a modern simulator. This should be attributed to the fact that the state of art of digital simulation has changed considerably since the design and completion of DLS.

DLS is table-driven and this is the current trend. It employs a fixed-time incrementation scheme. This is inefficient compared with faster and improved techniques. However, it uses the state variable approach in a very powerful way, allowing up to 59 unit delays for each signal without requiring any extra storage. Each time increment and updating of the circuit is performed by a simple left-shift operation of the entries in the value table LL. This saves considerable execution time, since a minimum of changes are performed on the data tables. Also it makes the time routine of the program conceptually easy.

The main drawback of DLS is its two valued simulation, which is considered obsolete now. If only two values are allowed, then problems with the initialization of the circuit arise. Unrealistic cases, such as both the input and the output of an inverter being 0 may initially occur.
Such problems may be corrected by having the user set consistent initial values. This is generally impractical even for circuits of moderate size. Another limitation of the two valued model is that don't-knows due to spikes, unknowns etc. cannot be used. This imposes restrictions on the application of the simulator, which is now basically limited to logic verification. Fault analysis capabilities of the simulator are also considerably decreased. It is basically this drawback that makes DLS a rather primitive simulator according to present-day standards.

The four modes of operation included simplify, to a large extent, its use. It makes simulation easier for the user and solves some of the problems created by the absence of don't-know value X, by checking for stability of the circuit (Synchronous and Asynchronous modes).

Some other advantages of the DLS are its modularity, which makes the program easy to understand and modify, and the free-format input, which simplifies its use. The basic element set includes a sufficient number of standard combinational and sequential elements.

In general, DLS is a powerful enough simulator if one considers the time of its development and the fact that it was written by one individual rather than by a group of engineers.
To increase its applicability in simulating medium-size circuits with MSI elements, the feature of macros, as discussed in Chapter 1, was developed. This feature is described in the next two chapters. It should be emphasized at this point that the macro processor is entirely independent of the particular simulator DLS. It can be used to reinforce any digital logic simulator, as well as other systems programs, such as assemblers.
Chapter 2

MACRO PROCESSING

Introduction

A macro processor is a text (or code) processor that facilitates programming in low-level languages. Its input is some text and its output is a processed version of the input text according to some prespecified rules. Usually the output of the macro processor serves as the input to an assembler. Thus one way of looking at the macro processor is to consider it to be a preprocessor of the assembler. Although this is its standard use, it is not restricted, as we shall show, to this application only.

To a certain extent, the macro processor fills the gap between high and low-level languages, enabling the programmer to write in a more powerful style while still in an assembly language environment. As such, the macro processor can be thought to be a programming tool in the service of the user.
3.1 Macros

Writing in assembly language, a programmer often faces the problem of repeating exactly the same code in several places throughout his program. One rather primitive way of solving this problem is to type (or punch) multiple copies of the same code and insert these copies wherever it is necessary. The need of a programming technique to relieve the programmer from such a tedious task is obvious. This is how the idea of macros and macro processors originated. The programmer would abbreviate the repeated code by assigning it some name, and whenever he wanted to use that particular code, he would just insert the name. It would be the task of the macro processor to substitute the corresponding code in the program whenever the name of a macro was encountered.

Consider the following instructions as being part of an assembly language program. The mnemonics of the machine instructions in this example, as well as in the ones that follow, are taken from the IBM-360 instruction set, but the choice is entirely arbitrary. The same idea would apply for any other assembly language, and thus the reader need not be familiar with the IBM-360 assembly instructions to be able to understand the example.
L 1,A Load register 1 with the contents of location A
A 1,3 Add to contents of reg. 1 the contents of location B
ST 1,C Store contents of reg. 1 in location C

---

Example 1.

In both cases, exactly the same three instructions are repeated to evaluate C=A+B. If that were done several times throughout the program, it would be a rather degrading task to have the programmer repeat the same code. It is usually when such embarrassing situations arise that new hardware or software techniques are devised.

3.2 Macro Language [20]

The above problem is solved by the software technique of using macros. That is, have the programmer name that
portion of the code which is to be used repeatedly and use the assigned name instead of the code thereafter. In the above example the programmer could write:

:MACRO ........ Beginning of macro
ADD ........ Name of macro
L 1,A
A 1,B. ....... Definition body of macro
ST 1,C
MEND ........ End of macro

ADD ........ First call to macro ADD

ADD ........ Second call to macro ADD

The first card MACRO indicates the beginning of a macro definition. The next card is the name that the programmer assigns to his macro. What follows is the definition of the macro, i.e the code to be abbreviated by the
macro name. The END card indicates the end of the macro.

Now, the programmer may call the macro ADD wherever he wants the corresponding code to be inserted. The macro processor saves the macro definitions and inserts the corresponding code on every occurrence of its name.

3.3 Macros with Dummy Arguments

The problem of repeating the same code is now overcome, but consider also the following example:

```
  L 1,A
  A 1,B
  ST 1,C
  L 1,A1
  A 1,B1
  ST 1,C1
```

Example 2.

A call to ADD, as previously defined, will work for the first case but not for the second. Both sets of instructions are similar in that they both evaluate $Z=X+Y$. In the first case, $X=A, Y=B, Z=C$ whereas in the second,
The programmer would like to define his macro using dummy arguments or parameters. He then would call it using the actual arguments. Thus a new version of ADD with this additional feature of dummy arguments would be:

MACRO
ADDXYZ 'X, 'Y, 'Z
L 1, 'X
A 1, 'Y
ST 1, 'Z
MEND

Dummy arguments are preceded by an apostrophe ('). The macro calls to ADDXYZ would now be ADDXYZ A,B,C, and ADDXYZ Al, Bl, Cl to generate the code in Example 2. Note that there is a 1-to-1 positional correspondence between the dummy and the actual arguments appearing on the macro name card and the macro call respectively. This is usually referred as a call by position. An alternative way is the call by name or keyword in which both the dummy and actual arguments are used in the macro call, e.g ADDXYZ 'X=A, 'Y=B, 'Z=C.
3.4 **Conditional macros**

Suppose the programmer wants to write the following code:

```
L 1,A
A 1,B
ST 1,C
.
.
A 1,B1
ST 1,C1
.
.
ST 1,C2
.
.
```

Example 3.

With the introduction of conditional macro pseudo-instructions, the programmer can write a new macro `ADDXYZN` as follows:
MACRO

ADDXYZN 'X,'Y,'Z,'N
IF 'N=1 GO .ONE
IF 'N=2 GO .TWO
IF 'N=3 GO .THREE

.ONE L 1,'X
.TWO A 1,'Y
.THIRE ST 1,'Z

MEND

The macro processor is now capable of performing arithmetic comparisons and following corresponding actions just like in a higher level language. GO is an unconditional jump. Processing continues from the line having as label the label that appears in the GO statement. Macro labels are preceded by a period to indicate to the processor that they are only for internal use (within the macro processor) and should not be generated into the expanded code, i.e. into the output of the macro processor.

The IF pseudo-op performs an arithmetic comparison. If the condition is satisfied, the pseudo-op that follows on the same card is processed. Otherwise the IF line is ignored and processing continues with the next line.

The calls ADDXYZN A,B,C,1
ADDXYZN A1,B1, C1,2
ADDXYZN A2,B2, C2,3
will generate the code shown in Example 3.
3.5 Nested Macro Definitions

In many occasions the programmer wants to define a new macro within the definition of another macro so that the inner one may share the dummy arguments of the outer one. Consider the following example:

MACRO
OUT 'A
.
.
.
MACRO
IN 'B
.
.
.
L 'A,'B → Macro definition of IN → Macro definition of OUT
.
.
.
MEND
.
.
.
MEND

Definition of macro IN will take place when macro OUT is called. Once OUT is called, the programmer may call IN
anywhere in his program including inside the definition of OUT after the definition of IN. The macro processor must now be able to recognize and process properly nested macro definitions.

It should be noted that during an expansion of OUT, the definition of IN is not generated in the expanded output but it is saved by the macro processor to be used for expanding any subsequent calls to macro IN.

3.6 Macro Calls within other Macros

Once a macro is defined, the programmer would like to be able to call it anywhere in his program, even within a macro definition. Consider for example the case in which the programmer wants to evaluate $Z=2(X+y)$ in several places in his program. He could of course write a new simple macro to do this, but he notices that the consecutive calls:

```
ADDXYZ A,B,C
ADDXYZ C,C,C
```

of his already defined macro ADDXYZ, have the same effect. Instead of repeating both lines each time, he can define a new macro ADDXYZ2 as follows:

```
MACRO
ADDXYZ2 'X,'Y,'Z
ADDXYZ 'X,'Y,'Z
```
If he then wants to evaluate $C=2(A+B)$ he may call:

```
ADDXYZ Z, Z, Z
```

The processor must now be able to process multiple levels of macro calls. To do this, a push down stack of arrays is employed where all pertinent information about a macro call is saved until it is fully expanded.

### 3.7 Recursive macros

With the two features of conditional macro processing and macro calls within macros, the programmer is capable of writing in a very powerful macro language. He may define a macro $A$ which calls itself, or which calls a macro $B$ which calls $C$ which calls $A$ again. These two cases are illustrated on Fig. 6 below:

![Figure 6: Recursive macros](image)

These macros forming a closed calling loop are standard examples of recursive macros. The loop is allowed to be formed because the macro processor may properly handle
macro calls within macros. The conditional macro processing is necessary in this case to break the closed loop when certain conditions specified by the programmer are satisfied. Without the conditional pseudo-ops, the macro processor would not be able to get out of the closed loop.

3.8 Macros vs. Subroutines

Most of the functions realized by macros can also be realized by subroutines. These two features are similar in functional use, but their execution differs. Macros are referred to as 'open' subroutines in contrast to subroutines which are referred to as 'closed' subroutines. The terms open and closed become meaningful when one examines the way each one is processed. The code for a macro is inserted whenever a call to this means is encountered. The subroutine is a closed entity somewhere in the program, to which transfer is made whenever it is called. The inherent disadvantage of macros is waste of space since a copy of the code defining a macro is physically inserted at each macro call. Execution time of macros, though, is shorter than that of subroutines, since the program is executed sequentially without time consuming jumps back and forth the main program and the subroutine. The assembly time, though, is increased when macros are used, since the re-
sulting program is longer than in the case of subroutines.

The main advantage of macros is the detachment it offers to the programmer from the tedious low level language. It elevates the programming style almost to a high level language, although the programmer still works in an assemble language environment. He does not have to worry about correct transfer of subroutine arguments and return addresses. Macros enable him to work with a language he understands better and which the computer does not have to translate through a compiler.
Chapter 4

MACRO PROCESSOR IMPLEMENTATION

Introduction

The macro processor MACRO described here is a FORTRAN program written, tested and run on the CDC-6400 computer at Lehigh University. It includes most of the standard macro processing features as well as some additional ones. It was made to be independent of any particular assembler, thus being of more general use. With no particular changes, it was used in three different applications: as a macro processor for the IBM-360 assembler simulator (LUIAS) written at Lehigh; as a macro processor for the PDP-8 assembler (LEPAL) also written at Lehigh; finally as a preprocessor to the Digital Logic Simulator (DLS) program written at Air Force Institute of Technology.

Macro is written in a modular way, being composed of a main program and ten subroutines. A user may easily make any additions or modifications to adapt MACRO to his needs. It should be a rather easy task to incorporate the macro processor into the first pass of an assembler, making both programs more powerful, since the macro processor could have access to the assembler's Symbol Table, thus eliminating some inherent limitations MACRO suffers for its
4.1 Macro Processor Implementation

With the theoretical background developed in Chapter 3 we are ready to introduce the implementation of the particular macro processor \textsc{Macho} as it was developed by the writer. \textsc{Macho} is a \textsc{Fortran} program, compiled by the \textsc{Ftn} compiler and run on the CDC-6400 at Lehigh University. Some of its main characteristics are the following:

1. It is a one pass macro processor.
2. It includes three conditional pseudo-ops which allow a wide range of conditional macro programming.
3. It handles nested macro definitions.
4. It handles nested macro calls (including recursive macros).
5. It includes a complete set of diagnostic messages that make the processor fail-safe.
6. It is format-free (no format restrictions imposed on the user).

In the current version of \textsc{Macho}, up to 100 macros are allowed in each program, with their total definitions not to be longer than 1000 lines. These upper bounds can be easily changed by increasing the \textsc{Dimension} statement in the
common block. They should however be sufficient for most average programs.

4.2 Data Structure

The data bases of MACHO mainly consist of the following tables:

a. The Macro Name Table, MNT.
b. The Macro Definition Table, MDT.
c. The dummy Argument List Array, ALA1.
d. A stack of arrays, S.

two separate tables whose entries correspond to those of the MNT which store indices:

e. The Macro Definition Table Index, MDTI.
f. The dummy Argument List Array Index, ALA1I.

There are also counters and pointers associated to the above data base tables. Namely:

a. The Macro Name Table Counter, MNTC.
b. The Macro Definition Table Counter, MDTC.
c. The dummy Argument List Array Counter, ALA1C.
d. The Stack Pointer, SP.

Also the Macro Definition Level Counter MDLC is used to keep track of nested macros.

Each line that is processed, either from the input deck or from the MDT, is copied to the array LINE. LINE is eight
words long, sufficient to store the eighty BCD characters on each card. Each BCD character of a card is also stored in array C1 (left justified). All tables and variables forming the data base of MACRO are shared among the various subroutines through a COMMON block.

4.2.1 The Macro Name Table MNT

The Macro Name Table is the table where the names of the various macros are sequentially stored as they are encountered in the user's input deck. For each macro name there are two indices associated with it. These are stored in two separate arrays:

a. The Macro Definition Table Index MDTI. It is a pointer to the beginning of each macro in MDT.

b. The dummy Argument List Array Index ALAI. It is a pointer to the first dummy argument of each macro in ALAI.

Each of MNT, MDTI and ALAI arrays occupy a 1000x1 array. Macro names are stored in MNT as left justified BCD characters.

4.2.2 The Macro Definition Table MDT

The Macro Definition Table stores the definitions of all macros. The MACRO card is not entered in MDT. The macro definition begins with the macro name card followed
by the main body of the definition and ends with the END card. Lines in MDT are exact copies of the cards in the user's input source deck defining a macro. MDT is a 1000x8 array. Eight words are needed per card to store its 80 BCD characters.

4.2.3 The Dummy Argument List Array ALA1

The dummy Argument List Array holds the dummy arguments of each macro as they are entered by the user. It is not necessary to have one general static argument list array but it was used to simplify the implementation of the program. The ALA1 is a 1000x1 array. Dummy arguments are stored as left justified BCD characters, with the first character always being the apostrophe (').

4.2.4 The Stack S

The reason for using the push down stack $S$ of arrays is to be able to handle nested macro calls, e.g. macro $A$ calling $B$. Obviously recursive macros also make use of this data base since they form a particular group of macros calling macros, with their calling sequence being a closed loop.

The information stored in an array in $S$ for each macro call is called a stack frame. Every macro call generates a stack frame. Each frame contains all the pertinent infor-
nation associated with that call. Thus each frame will contain in sequential order:

a. A pointer to the beginning of the previous frame.
b. A pointer to MDT indicating the next line of this macro to be processed.
c. The actual arguments appearing in the macro call in the order of their appearance.

The stack is a 1000x1 array. Pointers in parts (a) and (b) above are stored as integers. The actual arguments in part (c) are entered as left justified BCD characters.

Each of the counters MNTC, MDTC and ALAIC are used to keep track of the number of entries in each of the corresponding three data tables. They always point to the next free entry in the corresponding table.

The stack pointer SP points to the beginning of the current macro frame, i.e. the frame associated with the last macro call. If a new frame is to be added in S because a new macro call is encountered while expanding a previous one, SP changes properly to point to the first free entry in S where the new frame is to begin. If a macro expansion is completed, the corresponding frame is wiped out of the stack by simply decreasing the SP value to point to the beginning of the previous frame.
4.3 Example Showing the Use of the Various Data Bases

Consider the macros ADDXYZ and ADDXYZ2, defined in Chapter 3, whose definitions are repeated here.

MACRO
ADDXYZ  'X,'Y,'Z
L  1,'X
A  1,'Y
ST  1,'Z
MEND

MACRO
ADDXYZ2  'X,'Y,'Z
ADDXYZ  'X,'Y,'Z
ADDXYZ  'Z,'Z,'Z
MEND

The data bases formed by these two macro definitions will look like:

- 56 -
<table>
<thead>
<tr>
<th>HNTC</th>
<th>HNT</th>
<th>MDTI</th>
<th>ALA1I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADDXYZ</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ADDXYZ2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current value of HNTC=3

<table>
<thead>
<tr>
<th>HNTC</th>
<th>HNT</th>
<th>MDTI</th>
<th>ALA1C</th>
<th>ALA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADDXYZ</td>
<td></td>
<td>'X,'Y,'Z</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>L 1,'X</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>A 1,'Y</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>ST 1,'Z</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>MEND</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>ADDXYZ2</td>
<td></td>
<td>'X,'Y,'Z</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>ADDXYZ</td>
<td></td>
<td>'X,'Y,'Z</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>ADDXYZ</td>
<td></td>
<td>'Z,'Z,'Z</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MEND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current value of ALA1C=7

Current value of MDTC=10
The stack $S$ is not used yet since no macro call has been encountered. Assume that the call ADDXYZ2 A,B,C, is just being read from the input deck. The following stack frame will be created:

This regularly points to the beginning of the previous frame. Since there isn't any, the value -1 is used to indicate the bottom of the stack.

$SP=1 \rightarrow -1$

6 $\leftarrow$ Points to the beginning of ADDXYZ2 in MDT

A

B Actual arguments appearing on the macro call

C

$SP$ always points to the beginning of the current stack frame

$S(SP)$ points to the beginning of previous frame. If $S(SP)=-1$ then there is only one frame in stack.

$S(SP+1)$ points to the line in MDT to be processed next.

$S(SP+2)$ contains the first actual argument; rest, if any, follow sequentially in $S(SP+3), S(SP+4)$...
Once the stack frame is set up, the processor starts expanding the macro. It takes and processes the line in MDT pointed by S(3P+1). Here S(3P+1)=i(1+1)=i(2)=6. Therefore line MDT(6) is to be processed. The processor recognizes this as a macro name card and increments S(3P+1) which now points to MDT(7). The processor substitutes the actual arguments for the corresponding dummy ones, so that MDT(7) becomes ADDXYZ A, B, C. It recognizes this as a macro call, and thus a new frame is set up in stack S which now becomes:

```
SP=6  ---  1
     \  |
     7---\---A
      \   |
       \  |    B
        \|
         C
```

The new value of SP is 6, and S(3P+1)=i(7)=1 points to the beginning of ADDXYZ in MDT. ADDXYZ may now be processed, while the current information about the previous call to
ADDXYZ2 is saved in the first stack frame. When the expansion of ADDXYZ is completed, SP will become -1 again, and the expansion of ADDXYZ2 will continue.

All the changes taking place in the stack while ADDXYZ2 is expanded are shown below.

\[
\begin{matrix}
SP=-1 & \quad \text{SP}=-1 \\
-1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
6 & 7 & 7 & 7 & 7 & 7 & 8 & 8 & 8 & 8 & 9 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 & 1 & 2 & 5 & 1 & 2 & 5 \\
\end{matrix}
\]

**States:**

1 2 3 4 5 6 7 8 9 10 11 ... 14 15 16

State 1: No macro calls yet; stack empty; SP=-1.
State 2: Frame associated with call to ADDXYZ2 A,B,C.
State 3: Read call ADDXYZ A,B,C.
State 4: Set up a new frame for ADDXYZ A,B,C. Get ready to expand it.
State 5: Generate line L 1,A and pass it to the expanded output.

State 6: Generate line A 1,B and pass it to the expanded output.

State 7: Generate line ST 1,C and pass it to the expanded output.

State 8: MEND card in encountered; done with expanding ADDXYZ A,B,C.

State 9: Continue expansion of ADDXYZ2; call to ADDXYZ C,C,C, is encountered.

State 10: Set up a new frame for the above call.

States 11-14: Expand ADDXYZ C,C,C

Lines L 1,C
A 1,C
ST 1,C
are generated and passed to the expanded output.

State 15: The MEND card of ADDXYZ2 is encountered. Expansion is completed.

State 16: Initialize SP to -1 again, denoting completion of macro call expansion.
4.4 The MACRO Program

MACRO is a FORTRAN program compiled by the FTN compiler. It is written in a modular way to facilitate debugging and simplify possible modifications by future users. MACRO consists of the following subprograms.

1. Main program called MACROP
2. Subroutine READ
3. Subroutine DUMMY
4. Subroutine ARGUMNT
5. Subroutine SUBS
6. Subroutine WORDS
7. Subroutine IF
8. Subroutine SET
9. Subroutine GO
10. Subroutine NUMBER
11. Subroutine ERROR

Subprograms 1 to 5 form the main core of MACRO.
Subroutine WORDS is a utility subroutine used as a text scanner.
Subroutine 7, 8 and 9 handle the conditional macro pseudo-ops
Subroutine NUMBER is a utility subroutine used by subroutines 7, 8 and 9. Finally subroutine ERROR generates the diagnostic messages whenever an illegal input to MARCO is attempted.
4.4.1 Main Program

The main program is responsible for controlling the proper processing of macros. It is the nucleus of the program, calling pertinent subroutines during macro definitions or macro expansions.

The basic functions of the main program are:

1. Initialize counters and pointers.
2. Call subroutine READ to read a line (either from the input or from MDT).
3. If it is neither a MACRO card nor a macro call, pass it to the expanded output.
4. If it is a MACRO card:
   a. Call subroutine READ to read the macro name card either from the input deck or from MDT.
b. Enter the macro name into MDT and save the corresponding indices MDTI and ALAI.
c. Enter the macro name line into MDT.
d. Enter the dummy arguments in ALAC.
e. Increment macro definition table counter MDTC and macro definition level counter MDLC.
f. Start entering definition lines into MDT.
   i/ If a MEND card, decrement MDLC; if MDLC=0 go to step 2; else go to 4f.
   ii/ If a MACRO card, increment MDLC and go to 4f.
5. If it is a macro call set up the appropriate stack frame and go to 2.
Step 4 handles macro definitions. Step 5 handles macro calls.

4.4.2 Subroutine READ

Subroutine READ reads a line either from the user's input deck or from the MDT depending on the value of SP.

1. If SP=-1 (no macro call), read a line from the user's input deck and return to the main program.
2. If SP≠-1, it means that a macro is under expansion. Therefore:
   a. Increment S(SP+1) which now points to the line in MDT to be processed.
   b. Call subroutine SUBS to substitute the actual arguments appearing in the macro call for the dummy ones.

   1/ If it is a MEND card and MDLC=0, change SP to previous value (pop-up one frame in S since expansion of last macro is done); go to 1.

   11/ If it is not a MEND card, check if it is conditional macro pseudo-op; process it if it is; return to main program.

4.4.3 Subroutine DUMMY

Subroutine DUMMY stores in the dummy argument list array, ALA1, the dummy arguments appearing on each macro name
card. Only those arguments preceded by an apostrophe (') are considered dummy.

4.4.4 Subroutine ARGUMENT

Subroutine ARGUMENT stores in consecutive entries of stack S, starting from S(SN+2), the actual arguments appearing in a macro call.

4.4.5 Subroutine SUBS

This subroutine performs the substitution of the actual arguments for the dummy ones. Its input consists of lines from MDT which may include dummy arguments. If a dummy argument is found, SUBS finds the associated actual one and replaces the former by the latter. Since these may be of different length, the line may have to be expanded or shrunk accordingly. There is a routine in SUBS that takes care of this. If, while expanding the line, non-blank characters surpass column 80, a fatal error is generated. If shrinking of the line occurs, the routine fills the right end of the line with blanks.

4.4.6 Subroutine WORDS

This is the text scanner of the program. Its input is a card image either from the input deck or from the MDT. A pointer, called INDEX in the program, is also provided by the calling program. WORDS starts reading from the column
specified by INDEX until it has read a word. Leading
blanks are ignored. A word is assumed to have been read
if, after reading at least one non-blank character, a
delimeter is encountered. It is the delimiters (stored in
array DELIM) that indicate the end of the word. The de-
limiters used in WORDS are: < = > + - * / , ( ) blank.
New ones may be added and/or old ones deleted to suit the
user's needs. Delimiters are also considered as valid
words.

Once a word is being read, it is placed in variable
WORD as BCD left justified characters. Subroutine WORDS
returns WORD to the calling program, together with its
length (in LARG). Also INDEX is updated to point after the
end of WORD, so that a next call to WORDS will return the
next word in the line. The calling program usually ini-
tializes INDEX to 1 and then starts calling WORDS. On each
call a new word is returned, starting from left to right.
If WORD=10HENDOF CARD, then there are no more words to read
on the current line.

Subroutine WORDS was a breakthrough during development
of MACRO. Without it, the rest of the subroutines would
have had to scan each line individually and the result
would have been rather repetitive and confusing. With the
development of WORDS, this function was considerably sim-
Once FORTRAN was available, designing MACAO to be
format free rather than fixed-column-oriented was straight
forward. At the same time the implementation of the condi-
tional pseudo-ops was made a manageable task to realize.

4.4.7  **Subroutine IF**

Subroutine IF handles user specified arithmetic com-
parisons. There are three valid comparisons in MACAO: <,
= and >. If the condition is satisfied, the pseudo-op
on the same line is processed. Otherwise the IF line is
ignored.

4.4.8  **Subroutine SET**

This subroutine allows arithmetic modification of the
current numeric value of one of the actual arguments. It
is a pseudo-op instructing the macro processor to perform
the arithmetic operation and replace the old value of the
argument by the new one. A more formal description of the
SET pseudo-op is given on paragraph 4.5.2.5 in the user's
manual.

4.4.9  **Subroutine 30**

This subroutine alters the sequential processing of
lines in MDT. An unconditional jump is made to process
that line with the same label as that appearing on the GO
card.
4.4.10 Subroutine NUMBER

NUMBER is a utility subroutine for the IF and SET pseudo-ops. It translates BCD numbers to their corresponding integer values.

4.4.11 Subroutine ERROR

This subroutine generates the appropriate diagnostic messages whenever it is called. An effort was made to make MACRO fail-safe i.e. not to let the user get stuck with a mode error within his program, if an illegal input is attempted. Instead, the user gets a complete account of what went wrong. If the error is non-fatal, an informative message is printed in the user's output listing and execution continues.

If the error is fatal, the line generating the fatal error is printed, together with a message of what is illegal about it. Also the name of the macro that this line belongs to is provided to the error message to avoid ambiguities and facilitate debugging. The expanded code up to the fatal error is also provided to the user in an effort to decrease debugging time.
4.5 User's Manual

4.5.1 Accessing and Running the Macro Processor

The macro processor is on a file called MACRO with ID=GSM. File MACRO contains the compiled version of the original Fortran program. The FTN compiler was used to obtain the compiled code of MACRO. The same compiler should be used if recompilation of the Fortran program is desired. The input to MACRO should be provided by the user on the INPUT file. The generated expanded code is saved on a file called EXPAND which the user may apply as input to some other program (e.g., an assembler).

Thus if the user wants to access and run MACRO his deck should look like:

Job card
ATTACH, MACRO, ID=GSM.
MACRO.
7/8/9
User’s code including macro definitions and macro calls
6/7/8/9

If the user already has his program on some other file, he can input it to MACRO by using the control card: MACRO(file name) instead of MACRO. Filename is the name of the user’s file.

After either of the above cards is encountered, execution of MACRO takes place, and the expanded code is generated into the file EXPAND. The user may then catalog EXPAND or
Consider for example that the user has the input for MACAO on a file called FILE1. He wants MACAO to process FILE1 and then pass the processed output to an assembler called LUIAS. He should use the following control cards.

Job card
ATTACH,MACAO,ID=3SM.
ATTACH,LUIAS,ID=HPS.
MACAO(FILE1)
LUIAS( EXPAND )
7/8/9
User's program with macro definitions and macro calls
6/7/8/9

4.5.2 Macro Programming

4.5.2.1 Macro Definition

The user may define his macros anywhere in his program. Macro definitions should be preceded by a MACAO card, followed by the macro name card with the name of the macro and the dummy arguments. Dummy arguments should be preceded by an apostrophe (') and be separated by commas. The macro definition is ended with the MEND card.
The macro pseudo-ops MACRO and MEND as well as the macro name and the dummy arguments may be placed in any column.

4.5.2.2 Macro Calls

Once a macro is defined, it may be called by its name followed by the actual arguments to be substituted for the dummy ones. Dummy and actual arguments should be in one-to-one correspondence on the macro name card and in the macro call. If more actual arguments are provided than are needed, the extra ones are ignored. If fewer actual arguments are provided than are needed, the missing ones are assumed blank. A blank argument is also assumed if nothing appears between the corresponding two commas.
4.5.2.3 Macro Names and Arguments

Macro names, dummy and actual arguments may be strings of alphanumeric BCD characters not including the following delimiters:

\(<\, >=\, +\, -\, *\, /\, (\, )\, \) blank

They should be of length 9 or less.

4.5.2.4 Continuation Cards

If dummy or actual arguments cannot fit on one card, continuation cards may be used by entering a plus sign (+) anywhere at the end of the filled card. If a plus sign is encountered on a macro name card or on a macro call, the next card is taken to be continuation of the previous one. There is no limit on the number of continuation cards. An argument should be placed on a single card; splitting of an argument into two cards is illegal.

4.5.2.5 Conditional Macro-Expansion

The following three macro pseudo-ops are available for conditional macro expansion

a. SET (arithmetic replacement)
b. IF (arithmetic comparison)
c. GO (unconditional jump)

These may alter the numerical values of arguments as well as the sequence of macro processing, but they do not gen-
The SET Pseudo-op

The general form of the SET pseudo-op is:

\[ \text{SET (operand1) = (operand2) (operation) (operand3)} \]

Examples:

\[ \text{SET } 'A' = 'B' + 'C} \]
\[ \text{SET } 'A' = 'A' + 1 \]
\[ \text{SET } 'A' = 0 \]

Operand 1 must be a dummy argument. Operands 2 and 3 may either be dummy arguments or integer constants. If they are dummy arguments they should have a numerical value during the macro expansion.

The arithmetic operations allowed are: +, -, *, /

Only one operation per card is allowed. Consecutive SET pseudo-ops must be used for more complicated replacements. Integer constants may be positive or negative and should be limited to 9 digits (8 if signs are used).

The IF Pseudo-op

The general form of the IF pseudo-op is:

\[ \text{IF (operand1) (arithmetic comparison) (operand2) (JO or SET pseudo-ops)} \]

Example:

\[ \text{IF } 'A' < 'B' \text{ SET } 'A' = 'B} \]

- 73 -
both operands should be either dummy arguments or integer constants. If they are dummy arguments, their corresponding actual arguments should be numeric during the macro expansion.

There are three valid arithmetic comparisons allowed:

- Less than - typed as `<`
- Equal to - typed as `=`
- Greater than - typed as `>`

If the IF condition is satisfied, the following (on the same card) SET or GO pseudo-op is processed, otherwise the IF line is ignored.

The GO Pseudo-op

The general form of the GO pseudo-op is:

```
GO (macro label)
```

GO is a macro pseudo-op allowing non-sequential processing of the macro definition lines in KDT. A macro label is any alphanumerical string proceeded by a period (.) and placed anywhere in the beginning of a card. Labels may be placed on any card within the macro definition. They are used only for the conditional macro processing and are not generated into the expanded code.

The pseudo-op GO may refer to a macro label placed on a card preceeding or following the GO statement.
4.5.2.6 User's Comments in Macro Definition

The user may insert comments anywhere within the macro definition. Comments should be preceded by C. followed by at least one space and can be placed in any column. C. tells the processor that what follows are user comments. The processor ignores the comments and does not include them in the macro expansion.

4.5.2.7 Symbols to be changed on each macro call

In many cases the programmer does not want to generate the same symbol everytime he calls a macro. Consider, for example, the following macro:

```
MACRO
ANY
BEGIN L 1, 'A
        :
        :
BR    BEGIN C BR=Unconditional branch (jump to BEGIN) .
MEND
```

Every time macro ANY is called the label BEGIN will be generated in the expanded code. Although this is acceptable by the macro processor, there would be an assembly error if the expanded code was subsequently assembled. This would occur simply because assemblers do not allow multiply
defined symbols. To avoid this problem, a special routine was included in subroutine SJBS. Symbols that the user wants to be different on every call are placed within parenthesis (no blanks allowed). A two digit counter is added to the end of the symbol*, thus making it distinct on each call expansion. The user would re-write the previous macro as:

MACRO
ANY 'A
(BEGIN) L 1,'A

END

On the first call to ANY, label BEGIN would be BEJ00
On the second call, it would be BEJ01. So up to 100 times the user may call ANY, and each time a distinct label substitutes BEGIN.

If the user has to use parenthesis in some macro definition and does not want the processor to change the included contents, he should leave one blank after each left parenthesis.

*Symbols to be changed in parenthesis should be of length 3 or less. If greater than 3, only the first 3 characters are retained.
4.5.2.8 The END Card

MACHO expects an END card at the end of the user's source code for proper termination. If an END card is not encountered, MACHO assumes its existence and prints an informative message. This message may easily be removed from the program, if the user does not want his code to terminate with an END card.

4.5.3 Restrictions

a. MACHO is a one pass macro processor and as such it requires that a macro call should appear after the macro is defined.

b. All dummy and actual parameters, macro names and integer constants should be limited to an alplanumeric (or numeric) string of length 9 or less.

c. There should not be more than 100 macros on a single program. The length of the macro definitions should not total more than 1000 cards (comment cards are ignored). There should be up to 1000 dummy arguments all together in the macro cards. These bounds may easily be extended.

4.5.4 Remarks

a. MACHO is written in a modular way so that it can be readily modified.
b. Its input is not restricted to any particular assembly language. As it will be shown, it doesn't even have to be an assembly language.

c. MACRO is a free format macro processor. Pseudo-ops and arguments can be placed on any column. A blank serves as a delimiter. Extra ones are ignored.

d. MACRO allows nested macro definitions and nested macro calls.

e. The conditional pseudo-ops allow the user a wide range of very powerful and rather general conditional macro processing. The user may have the macro processor do the arithmetic and generate only one line with the result at the end. This is a rather extended application of macros which are now capable of actually saving space by expanding a minimum number of lines.

f. A complete set of non-fatal and fatal error diagnostics is available which makes the macro processor fail-safe. If illegal input to the MACRO is attempted, the user will get an error message explicitly stating what went wrong, rather than getting stuck with a mode error somewhere in MACRO.
Bibliography


16. S. A. Szygenda and E. W. Thompson, "Digital Logic Simulation in a Time-Based, Table-Driven Environment", Computer Vol 8, March 1975

(continued)
Examples of macro-programming with MACRO

I. Application of MACRO in Assembly Languages

1. MACRO and the IBM-360 Assembly Language

Three examples using the macro facility with the IBM-360 assembly language are given below. In all three cases, a macro that calculates $N!$ for some integer $N$, is described.

Example 1.1

MACRO

FACTORIAL 'N

L 1,=H(1)  C. PUT 1 IN REGISTER 1

.BEGIN    MH 1,=H('N)  C. MULTIPLY REG. 1 BY N
    SET 'N='N-1  C. DECREMENT N
    IF 'N>0 GO .BEGIN

MEND

The generated code for a call to FACTORIAL 10 is:

L 1,=H(1)

MH 1,=H(10)

MH 1,=H(9)

MH 1,=H(8)

.  .  .

- 82 -
The first instruction sets register 1 to 1. Then consecutive multiplications are performed starting with N=10 which is decremented by 1 every time. If the above code was executed by an assembler, 10! = 3628800 would be left in reg. 1. The reader should notice the following:

a. Macro comments are preceded by C. followed by at least one space. They are not generated into the expanded output.

b. One blank follows every left parenthesis to override MACRO's feature of creating new symbols on each call, as described in 2.7 of MACRO's User's Manual.

c. Register 1 is explicitly specified in the definition. The macro would be more general and versatile, if the register number was variable. Another dummy 'REG would be used to specify the variable register.

Example 1.2 N! using a recursive macro.

Consider the following recursive macro:
MACRO
FACTORIAL 'N
IF 'N=0 GO .DONE
MH 1,=H( 'N)
SET 'N= 'N-1
FACTORIAL 'N
.DONE MEND

The above macro generates the following code for the call FACTORIAL 10:

MH 1,=H( 10)
MH 1,=H( 9)
.. ..
MH 1,=H( 1)

The user should initialize reg. 1 to 1 before each call to FACTORIAL. To avoid doing so, he would define a new macro (that's what macros are for after all!) as follows:

MACRO
FACTOR 'N
 L 1,=H( 1)
FACTORIAL 'N
MEND
The code to call FACT 10 would be exactly the same as the one generated in Example 1.1

Example 1.3

In both Examples 1.1 and 1.2, N+1 lines of assembly code were generated. A new version of FACTORIAL is given below, which generates a single line regardless of the value of N:

```
MACRO
FACTORIAL 'N,'TEMP
.BEGIN
SET 'TEMP='TEMP*'N
SET 'N='N-1
IF 'N=0 GO .BEGIN
L 1,.=F('TEMP)
MEND
```

Then, to calculate N!, the user should call FACTORIAL N,1. For example, to find 10!, he would write FACTORIAL 10,1. The generated code of this call would be: L 1,.=F(3628800)

Instead of letting the macro generate the desired code, the user now takes advantage of the arithmetic capabilities implemented in MACRO. It leaves it up to the macro processor to do the arithmetic and then generate only one assembly line with the desired result. This is a rather unusual but very interesting use of macro-programming that MACRO offers to the programmer. The argument that:
'It is the computer that evaluates N! no matter whether it does it during macro-processing or at execution time of the assembled program', is true, but:

1. Total processing time (macro-processing, assembling, execution) is decreased. Assembly and execution times are decreased with macro-processing time virtually invariant.

2. The programmer is enabled to think of his problem in terms of a high-level language, and yet he does not need a compiler! This is exactly what macros should be besides being primitive code substitutors.

The reader should be cautioned at this point that these particular examples are of little practical use since the value of 'N has to be passed by the user on his macro call before assembly time. Hence, in the case of FACTORIAL 'N, he could easily look up N! in some tables or use a desk calculator to find it, rather than calling a macro. However, these examples are presented for demonstrational purposes, and with the hope that they show some of the capabilities (as well as incapacities) of macro-processing.

2. MACHO and the PDP-8 Assembly language

Example 2.1

This example shows the typical application of a macro.
A portion of the assembly program that performs some function is abbreviated by some name so that it can be called whenever it is necessary in the program.

**MACRO**

**MULT** 'A,'3  
CIA CLL C. CLEAR ACCUM. AND LINK  
TAD (A) C. ADD A  
CIA C. TAKE -A  
DCA (TALLY) C. STORE IT IN TALLY

*MULT*, TAD (B) C. START ADDING B  
ISZ (TALLY) C. INCREMENT TALLY AND SKIP IF 0  
JMP (MULT) C. IF NOT 0 ADD B ONCE MORE  
JMP .+4 C. SKIP OVER DATA

(A), 'A
(B), '3  
(TALLY), 0

**MEND**

The user, once he defined the above macro, could call **MULT N,A** to calculate N*M, with the result being left in the accumulator. On each call, the above code would be generated with labels in parenthesis properly concatenated with a counter so to be distinct.
To show again how a macro can be used with minimum expanded code, consider another version of the previous example.

```
MACRO
MULT 'A,'B
SET 'A='A*','B
CLA CLL
TAD 'A
MEND
```

This should make clear the degree of efficiency that a macro may achieve, if properly written according to the MACRO specifications. The user seems to write in some high level language and detaches himself from the rather primitive assembly language instructions of Example 2.1. And yet, he does not need a compiler to do so! As in the case of the FACTORIAL 'N macro, the MULT 'A,'B above seems of little practical use, since the programmer could perform the multiplication operation himself, rather than calling the macro. However, for more complicated functions, such a macro might be defined and used.

Notice that the Example 2.1, with a few changes in the beginning, may be used with one parameter ('B for ex-
ample) to perform a multiplication of '3 with whatever is in the accumulator at the time of the macro call. This would be a standard (and practical) use of macro 
MULT, since the multiplication is going to be performed at execution time (contents of accumulator unknown beforehand) rather than at assembly time (or macro processing time).

II. Application of MACRO in Digital Logic Simulation.

3. MACRO and the Digital Logic Simulator DLS.

Example 3.1

Consider the case in which a user wants to test a circuit with a 5-bit asynchronous counter in several places in his design. The description of such a counter is given below.

Figure 7: 5-bit Asynchronous counter.

Since such modules are generally not available in the basic element sets of simulators, the user would have to write the above flip-flop interconnection whenever he wanted to insert the counter in his circuit. Assuming
that the statement

\[ \text{JKFF, } J, J, K, \text{CL} \]

describes to the simulator a JK flip-flop with output \( Q \). J,K lines called \( J \) and \( K \) respectively, and clock line called \( \text{CL} \), the following macro could have been written to save him the trouble of writing the same code over and over again.

MACRO

\[
\text{COUNTER '25,'24,'23,'22,'21,'CL} \\
\text{JKFF, '25,'J,'K,'CL} \\
\text{JKFF, '24,'J,'K,'25} \\
\text{JKFF, '23,'J,'K,'24} \\
\text{JKFF, '22,'J,'K,'23} \\
\text{JKFF, '21,'J,'K,'22}
\]

MEND

To use this macro, the user would call it with the desired lead names. A call : \text{COUNTER 5,4,3,2,1,CL} would generate the circuit in Figure 1 above. It should be noticed that since inputs \( J \) and \( K \) are always open (logic 1), they do not need to be distinct on each call, so no parenthesis are needed. The user would have to set them to 1 at the beginning of the simulation.
Example 3.2

The macro described above will work properly. It has two disadvantages though:

a. The user can call it only when a 5-bit counter is desired.

b. The length of the macro definition body is equal to the number of the JK flip-flops in the counter. If a 50-bit counter were to be implemented, macro COUNTER would also need the description of 50 distinct flip-flops.

These problems can be overcome with the following version of COUNTER which employs conditional macro-programming.

MACRO
COUNTER 'i,'CL,'TEMP
JKFF,'N,J,K,'CL
.BEGIN   SET 'TEMP='N
         SET 'N='N-1
         JKFF,'N,J,K,'TEMP
         IF N>1 GO .BEGIN
MEND

Then, to generate the counter of Example 1, he would call COUNTER 5,CL,1. 'TEMP is a temporary arithmetic variable which is used locally in the macro. This version of
COUNTER eliminates the problems of the macro in Example 3.1. Example 3.2 illustrates an interesting concept. Even in digital logic simulation, macros can be used not only for direct code substitution (as in Example 3.1), but they can also generate sub-circuits of iterative structure and of variable length. Example 3.2 shows how a one-dimensional iterative circuit may be generated. The generation of two dimensional iterative circuits is conceptually similar but more complex. The iterative unit would be included in two nested loops.

![Space Iterative Circuit Diagram](image)

*Figure 8: Space iterative circuit.*
Example 3.3

The macro in Example 3.2 will generate the desired counter of variable length, but only the first time of its call. Further calls to it would cause problems, since the output lines of all counters would have common names (1,2,3,... etc.). To overcome this problem a new and final version of COUNTER is described below.

MACRO
COUNTER 'N1,'N2,'CL,'TEMP
JKFF,'N1,J,K,'CL

.BEGIN SET 'TEMP='N
SET 'N='N-1
JKFF,'N1,J,K,'TEMP
IF 'N1>'N2 GO .BEGIN
MEND

In this version of COUNTER, the iteration length of the sub-circuit is given in terms of the difference of the variables 'N1 and 'N2. To generate the code of Example 3.1 the user would call: COUNTER 5,1,CL,1 . If another counter of 7 bits was later wanted, the user could call COUNTER 12,6,CL,1 . This counter would use the same clock line CL as before, but would have the distinct output lines: 12,11,10,9,8,7,6. Hence, as many counters of
any length could be generated by a single macro. This
clearly illustrates the power of conditional-macro pro-
grams available with NiC30.
There is no apostrophe (') symbol in the printer. Thus, the apostrophe is printed as the inequality sign (#).
INPUT TO MACRO PROCESSOR

EXAMPLE 1.1 EVALUATION OF FACTORIAL WITH A LOOP

```
L 1,=H( 1)
BEGIN MH 1,=H( #N)
SLT #N=#N-1
IF #N>0 GO .BEGIN
MEND FACTORIAL 10
END
```
**INPUT TO MACRO PROCESSOR**

**EXAMPLE 1.2 EVALUATION OF FACTORIAL WITH A RECURSIVE MACRO**

MACRO FACTORIAL *N
TF *N=0, GO *DONE
MH 1,=H(*N)
SET *N=*N-1
FACTORIAL *N
*DONE
MEND FACTORIAL 10

**EXAMPLE 1.2 COMPLETE VERSION OF EXAMPLE WITH RECURSION**

MACRO FACTOR *N
L 1,=H(1)
FACTORIAL *N
MEND FACTOR 10

C. THIS LINE SHOULD BE GENERATED

END

**EXPANDED CODE FROM MACRO PROCESSOR**

**EXAMPLE 1.2 EVALUATION OF FACTORIAL WITH A RECURSIVE MACRO**

MH 1,=H(10)
MH 1,=H(9)
MH 1,=H(8)
MH 1,=H(7)
MH 1,=H(6)
MH 1,=H(5)
MH 1,=H(4)
MH 1,=H(3)
MH 1,=H(2)
MH 1,=H(1)

**EXAMPLE 1.2 COMPLETE VERSION OF EXAMPLE WITH RECURSION**

L 1,=H(1)
MH 1,=H(10)
MH 1,=H(9)
MH 1,=H(8)
MH 1,=H(7)
MH 1,=H(6)
MH 1,=H(5)
MH 1,=H(4)
MH 1,=H(3)
MH 1,=H(2)
MH 1,=H(1)

C. THIS LINE SHOULD BE GENERATED

END
INPUT TO MACRO PROCESSOR

EXAMPLE 1.3 EVALUATION OF FACTORIAL BY THE MACRO PROCESSOR

MACRO FACTORIAL  
BEGIN  SET *TEMP=*TEMP*^N  
IF *N=*N-1  GO BEGIN  
L 1,*F(*TEMP)  
END

FACTORIAL 10,1

END

EXPANDED CODE FROM MACRO PROCESSOR

EXAMPLE 1.3 EVALUATION OF FACTORIAL BY THE MACRO PROCESSOR

L 1,*F(3628800)
INPUT TO MACRO PROCESSOR

EXAMPLE 2.1 MULTIPLICATION BY PDP-8

MACRO
MULT A, A
CLA CLL
TAD (A)
CIA
DCA (TALLY)
(MULT), TAD (A)
ISZ (TALLY)
JMP (MULT)
(A), DEC A
(TALLY), 0
MEND
MULT 10, 20
MULT 20, 30
MULT 30, 4

END

EXPANDED CODE FROM MACRO PROCESSOR:

EXAMPLE 2.1 MULTIPLICATION BY PDP-8

CLA CL1
TAD A00
CIA
DCA TAL00
MUL00, TAD 900
ISZ TAL00
JMP MUL00
A00, DEC 10
B00, DEC 20
TAL00, 0
CLA CL1
TAD A01
CIA
DCA TAL01
MUL01, TAD 901
ISZ TAL01
JMP MUL01
A01, DEC 20
B01, DEC 30
TAL01, 0
CLA CL1
TAD A02
CIA
DCA TAL02
MUL02, TAD 902
ISZ TAL02
JMP MUL02
A02, DEC 10
B02, DEC 4
TAL02, 0

END
**** FORMAT CONTROL SUPPRESSED ****

INPUT TO MACRO PROCESSOR

EXAMPLE 2.2 MULTIPLICATION WITH PDP-8
(BO THE MACRO PROCESSOR)

```
MACRO
MULT zA,zB
SET zA=zA+zB
CLA CLL (A)
JMP +1
(A)
DEC +4
MEND
MULT 12,13
MULT 14,15
MULT 23,37
END
```

EXPANDED CODE FROM MACRO PROCESSOR:

EXAMPLE 2.2 MULTIPLICATION WITH PDP-8
(BO THE MACRO PROCESSOR)

```
CLA CLL A00
JMP +1
A00
CLA CLL A01
JMP +1
A01
CLA CLL A02
JMP +1
A02
END
```

- 100 -
INPUT TO MACRO PROCESSOR

EXAMPLE 3.1 ASYNCHRONOUS COUNTER (LONG WAY) IN DLS

MICRO COUNTER 05,04,03,02,01,0CL
JKFF,05,J,K,0CL
JKFF,04,J,K,0CL
JKFF,03,J,K,0CL
JKFF,02,J,K,0CL
JKFF,01,J,K,0CL
END

COUNTER 5,4,3,2,1,0CL

EXPANDED CODE FROM MACRO PROCESSOR:

EXAMPLE 3.1 ASYNCHRONOUS COUNTER (LONG WAY) IN DLS

JKFF,5,J,K,0CL
JKFF,4,J,K,0CL
JKFF,3,J,K,0CL
JKFF,2,J,K,0CL
JKFF,1,J,K,0CL
END
*** FORMAT CONTROL SUPPRESSED ***

INPUT TO MACRO PROCESSOR:

EXAMPLE 1.2 COUNTER IN OLS GENERATED BY A MACRO (RESTRICTED USE)

MACRO
COUNTER \#N, \#CL, \#TEMP

*BEGIN
SET \#TEMP=\#N
SET \#N=\#N-1
IF \#N>1 GO .BEGIN
END
COUNTER 5, \#CL,1
COUNTER 7, \#CL,1

******* NON-FATAL ERROR: NO END CARD. END CARD ASSUMED

EXPANDED CODE FROM MACRO PROCESSOR:

EXAMPLE 3.2 COUNTER IN OLS GENERATED BY A MACRO (RESTRICTED USE)

JKFF,5, J, K, CL
JKFF,5, J, K, 6
JKFF,7, J, K, 4
JKFF,2, J, K, 3
JKFF,2, J, K, 7
JKFF,6, J, K, 5
JKFF,4, J, K, 5
JKFF,2, J, K, 6
JKFF,3, J, K, 2

05/06/75 SCOPE 3.4.1 P376 LEHIGH U. 04/12/75
09.24.37 LAERTS, A3036, T18, C70000, *MACRO.
09.24.37 ATTACH, MACRO, ID=GSM.
09.24.37 PEN TO
09.24.37 MACRO
08.24.37 PF CYCLE NO. = 001
08.24.37 MACRO
08.24.47 END MACRO
08.24.47 OP 0000384 WORDS - FILE OUTPUT, OC 40
08.24.44 NR. OF NON-STANDARD (DISK) CIO CALLS = 23
08.24.44 SYSTEM SECONDS USED BY THIS JOB = 1.9
08.24.44 EXECUTION COST OF THIS JOB, NOT INCL I/O COST, IS $ .34
08.24.44 CURRENT AUTHORIZATION BALANCE IS $ 119.76
08.24.44 CP 4.324 SEC.
08.24.44 PP 1.341 SEC.
08.24.44 CH 0.425 SEC.
INPUT TO MACRO PROCESSOR

EXAMPLE 3.3 COUNTER IN OLS GENERATED BY A MACRO

MACRO
COUNTER #N1,#N2,#CL,#TEMP
BEGIN
SFT #TEMP=#N1
SFT #N1=#N1-1
JKFF,#N1,J,K,#TEMP
IF #N1=#N2 GO BEGIN
END
COUNTER 5,1,CL,1
COUNTER 12,6,CL,1
COUNTER 25,13,CL,1

END

EXPANDED CODE FROM MACRO PROCESSOR:

EXAMPLE 3.3 COUNTER IN OLS GENERATED BY A MACRO

JKFF,5,J,K,CL
JKFF,4,J,K,5
JKFF,3,J,K,4
JKFF,2,J,K,3
JKFF,1,J,K,2
JKFF,12,J,K,CL
JKFF,11,J,K,12
JKFF,10,J,K,11
JKFF,9,J,K,10
JKFF,8,J,K,9
JKFF,7,J,K,8
JKFF,6,J,K,7
JKFF,25,J,K,CL
JKFF,24,J,K,25
JKFF,23,J,K,24
JKFF,22,J,K,23
JKFF,21,J,K,22
JKFF,20,J,K,21
JKFF,19,J,K,20
JKFF,18,J,K,19
JKFF,17,J,K,18
JKFF,16,J,K,17
JKFF,15,J,K,16
JKFF,14,J,K,15
JKFF,13,J,K,14

END
APPENDIX B

Flowcharts for MACRO subprograms:

1. MACROP
2. READ
3. DUMMY
4. ASSMNT
5. SUBS
6. WORDS
7. IF
8. SET
9. GO
10. NUMBER
11. ERROR
Start

Initialize counters, variables

Call READ

Is WORD in MNT?
Yes

S(SP+NA+2) ← SP
SP ← SP+NA+2
S(SP+1) ← MDTI for macro name
Call ARGUMENT

No

Is it a MACRO pseudo-op?
Yes

Call READ

MNT(MNTC) ← WORD
MDT(MNTC) ← MDTC
ALA1(MNTC) ← ALA1C
MNTC ← MNTC+1
MDLC ← MDLC+1
MNTC ← MNTC+1
MDLC ← MDLC+1
No

END

Write LINE into EXPAND

END

a

b

(cont.)
MDT(MDTC) macro name card

MDTC ← MDTC + 1

Call READ

MDT(MDTC) ← LINE
MDTC ← MDTC + 1

Is MACRO pseudo-op?
Yes → MDLC ← MDLC + 1
No → Is MEND pseudo-op?
Yes → MDLC ← MDLC - 1
No → MDLC = 0

- 106 -
Subroutine READ.

- 107 -
Subroutine DUMMY

More dummy arguments?

Yes

ALA1(ALA1C) ← WORD

ALA1C ← ALA1C+1

No

Return

Subroutine ARGUMNT

ARGUMNT

NA ← 0

More arguments?

Yes

NA ← NA+1

No

Return

S(SP+NA+1) ← WORD

- 108 -
Subroutine SUBS.

```plaintext
LINE ← MDT(S(SP+1))

Search for a dummy argument in LINE

A dummy argument found?

Yes

Search ALA1 to find relative position L-relative-pos.

Substitute in LINE dummy arg. for S(SP+L+1)

No

Return
```

- 109 -
Subroutine WORDS

I ← INDEX

Is C1(I) a blank?
Yes: I ← I+1
No:

J ← I

J ← J+1

Is C1(J) a delimiter?
Yes:

WORD ← C1(I) to C1(J-1)

LARG ← J-I

INDEX ← J

Return
Subroutine IF

Is arithm. condition satisfied?

Yes

LINE ← pseudo-op on IF card

No

Return

Subroutine SET.

Perform the arithm. operation specified on the SET card

Replace proper actual argum. in $S$ by its new value

Return
Subroutine 30.

Search for label in MDT, from: MDT(i) to MDT(i+1)

Label found in MDT?

Yes

\[ s(3P+1) \leftarrow \text{Pointer in MDT line with found label} \]

Return

No

Call ERROR (undefined label)
Subroutine NUKER

ARG ← numerical value of WORD

Return

Subroutine ERROR

ERROR

Print MESS(IE)

Fatal error?

Yes

Print F.E. diagnostic

No

Return

STOP 1

- 113 -
APPENDIX C:

References to the DLS and MACRO programs.

A detailed description, a documented computer listing and a punched deck of the DLS simulator are available at the Electrical Engineering Department, together with additional information supplied by the writer.

A listing and a deck of the MACRO program are also available at the EE Department as well as at the Lehigh University Computer Center.
VITA

George Jotirios Maroudas was born in Athens, Greece on 20 April 1951. He graduated from Athens College, a high school in Psychico, Athens in 1970. He then attended Lehigh University at Bethlehem, Pa. and received his Bachelor of Science in Electrical Engineering in 1973. He continued his studies at Lehigh University where he received a Master of Science in Electrical Engineering in 1975. He is a member of HKN and TBP societies as well as of IEEE.