AN ABSTRACT OF THE THESIS OF

Padma R. Baheti for the degree of Master of Science

in Electrical and Computer Engineering presented on October 27, 1975

Title: REGTRAN AND MICRO, THE IMPLEMENTATION OF TWO DIGITAL SYSTEM SIMULATORS.

Redacted for privacy

Abstract approved:

W. R. Adrion

REGTRAN (REGister TRANsfer), a hardware description language and SYSSIM (SYStems SIMulation), a simulator for the system described by REGTRAN were originally developed by Edward Pett, Jr [33] at the University of Texas at Austin. These two programs are successfully used to simulate many digital systems of complex nature. It is the primary purpose of this paper to describe the implementation of these two programs for KRONOS at Oregon State University. In this process, various problems, due to the differences in the operating systems, were faced. These problems are discussed in detail and the changes that were made are included. REGTRAN and SYSSIM are not capable of simulating microinstructions or a microprocessor. For this reason, another simulator MICRO is developed. Detailed description of MICRO and its use as a general purpose simulator as well as a microprogram simulator are given

in detail. As an example, partial emulation of DEC PDP-8 is shown. Suggestions for the improvement of MICRO, REGTRAN and SYSSIM are mentioned.

REGTRAN and MICRO, The Implementation of Two Digital System Simulators

by

Padma R. Baheti

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

June 1976

APPROVED:

Redacted for privacy

Assistant Professor of Electrical and Computer Engineering

in charge of major

Redacted for privacy

Head of Department of Electrical and Computer Engineering

Redacted for privacy

Dean of Graduate School

Date thesis is presented 27th October '75

Typed by Clover Redfern for Padma R. Baheti

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

REGTRAN AND MICRO, THE IMPLEMENTATION OF TWO DIGITAL SYSTEM SIMULATORS

I. INTRODUCTION

In computer science curricula, the hardware and software engineering appear as almost completely separate and independent disciplines because of conventional teaching methods. As a consequence of this, the student will have a poor understanding of hardware and software interaction. If a "user-reorganizable" laboratory computer is provided to the student, then he will be able to generate wiring diagrams and fabricate the machine. Not many universities, not even the few affluent ones in these days of shortage, can afford to build machines for teaching and experimenting purposes. By using a hardware description language with a simulator, it is possible for the students to learn design techniques. They could analyze and evaluate some performance aspects as well as proceed with software development in the pre-prototype stages of design.

An attempt is made to implement a hardware description language REGTRAN and a simulator for that language SYSSIM to KRONOS operating system. REGTRAN and SYSSIM are used for teaching a course in digital system design at the University of Texas at Austin. The same usage is anticipated at Oregon State University. Using REGRAN and SYSSIM, a wide variety of synchronous digital

systems can be successfully simulated. The simulation of micro-instructions or a microprogrammable computer can not be done by REGTRAN and SYSSIM. For this purpose another simulator MICRO is created. MICRO can be used as a microprogram simulator as well as a general purpose computer simulator at machine language level. Using REGTRAN, SYSSIM and MICRO, the student technicians can learn better design techniques.

1.1. Hardware Description Languages

Hardware description languages are of considerable importance in the development of advanced design automation techniques for digital machines. They offer the designer a convenient means for expressing his ideas in a form which can be easily read and understood by another designer.

A digital system can be described at several levels.

1. The highest level is the algorithmic level which specifies only the algorithm to be used for solving design problems.
2. The PMS (Processor, Memory, Switch) level.
3. The instruction level describes the instructions of a computer.
4. The register transfer or microinstruction level describes operations among registers.
5. The logic level expresses network in terms of gates and flip-flops.

6. The lowest level is the circuit level which implements gates and flip-flops by circuit elements such as transistors, registers etc.

By using Hardware Description Language along with a hardware simulator a much wider set of design alternatives can be quickly explored prior to the actual bread boarding of a prototype unit. It is much easier to change a line of code than it is to modify a breadboard circuit and its associated drawings and documents. The former one even saves the cost and time also in modifying the system. The main advantage of an HDL is the self documenting feature. The beneficiaries of this simulation approach would be the architectural designers in industry and the students in universities. They could analyze and evaluate some performance aspects as well as proceed with software development in the pre-prototype stages of design.

1.2. Requirements of Description Languages and Review of Different Languages Available

Languages for describing digital systems must provide sufficient information about the system behavior and about the system structure to show how it might be constructed. A system description language should allow easy and precise description of digital system behavior.

A number of high level languages have been reported during recent years. While many languages can be used at several levels,

each language is especially convenient for certain levels. Register transfer languages were created to meet the need for computer hardware description at a stage higher than the conventional gate level design. As languages they should satisfy some requirements; be amenable to simulation on a computer; and in an integrated and total design automation, and above all serve as a means of documentation and communication for computer engineers.

Reed's register transfer language [1-3] has received wider distribution. It is easily learned, generally applicable and its statements associate directly with hardware. But it is limited in application because it lacks means for describing complex, iterative networks, no provisions exist for partitioning the system and the small vocabulary of the language necessitates the use of many symbols. Determining the sequence of events which are to take place in a system from the Reed's language description may not be an easy task.

Schor's register transfer language is [4] with more options such as notations for indirect addressing, decoding, addition, subtraction and complementation and for identifying subregisters. It deals with flip-flops, gates, delays fan-in and fan-out requirements. It is a universal language, but is also a low-level language which requires many statements for a complex system.

Iverson's APL [5-7] (A Programming Language) has adequate high level operators for operating on arrays of data. The language is

broad in scope, having been developed for and applied effectively in such diverse areas as microprogramming, switching theory, operational research and so many more. The programs are sequences of statements. A mjaor disadvantage is its lack of means for describing parallel activities. It has no direct way of declaring dimensions and requires that its data be homogeneous. It has only a few primitive concepts and their use is consistent. However, its compact encoding of operators require the development of some reading skills.

Hill and Peterson's language AHPL [8] provides a complete description of sequential network or digital system. Only those APL operators which satisfy the constraints imposed by available hardware are included in AHPL. AHPL description will be translated directly into a wiring list or fan-in-list. The AHPL description itself then becomes the principle vehicle for communications about the network. This is beneficial in itself in that the control sequence, conventionally displays timing information, not at all evident in the schematic diagram.

Yohan Chu's language CDL [9-13] is a non procedural language, meaning that it attached no significance to the lexecographical ordering of statements, describing the operator of the system. Statements are associated with some sort of "label" defining the condition for execution. Sequencing is performed by modifying the control

variables used in the labels. A program written in CDL may be translated with little trouble into the language REGTRAN described in this report.

LOGAL by Lund, J. [14] is a modified register transfer language providing some additional properties not found in the original RTL such as partitioning information, new operations etc.

LOTIS by Schlaeppi, H.P. [15] describes the machine in two parts, declaration and procedures. Procedures may be written in turn in either sequences or functions. The behavior elements of LOTIS are register-to-register transfers. The main features of a hardware oriented notation have been presented, which is believed to be suitable for formally describing the relevant properties of the logical structures of digital machines and their internal timing and sequence.

APDL (Algorithmic Processor Description Language) [16-22] is structured as that of ALGOL; i.e., a set of blocks, each with its own declarations and statements. Blocks can be nested to any depth, providing a simple scheme to organize a description in a hierarchical fashion. It has additional (as compared to ALGOL) features to handle timing and register variables. It requires a large number of reserved key words and uses too many types of registers and arrays. It provides a high degree of flexibility in the organization and partition of the description to reflect the machine organization at several levels

of details and timing. Parnas [16] lays the ground for such a language. The actual structure of the language is well defined in [17], [18], and [19]. Reference [20] contains nice facilities in addition to those of APDL for describing a system as a network of subsystems.

ISP (Instruction Set Processor) [23-27] describes the primitives at the programming level of design. It can handle concurrency and sequencing of activities and provides an adequate set of data and control operators. Descriptions follow the block structure of ALGOL and can be named and used as independent processes or as parts of larger units. These descriptions are by a fixed format composed of declarations and actions. Declarations include memory, data types, data operations and instruction formats. Actions consist of an interpreter for fetching, coding, execution and specific instructions from a set.

LALSD (Language for Automated Logic and System Design) [28, 29] views a digital system in two parts: The structure and the control. The structure part performs the logical and arithmetic operations. The control part commands the behavior of the system. This separation makes it possible for the control part to be implemented either as hardware, firmware software, or any combination. Using techniques of this language, a complete system can be integrated from independently designed subsystems. LALSD is suitable for time sharing, interactive environment. It allows parallel

operations and can be used in designing synchronous, asynchronous, or mixed systems.

DDL (A Digital System Design Language) [30-32] by Duley and Dietmaya is easily understood as it is written in boolean-type equations, yet it covers a wide spectrum of digital system design. DDL's conciseness facilitates expressing, analyzing, modifying, and in general, dealing with large digital systems in an organized manner. The language is mnemonic and fundamental in concept to facilitate design and enhance readability. The organization of a document can parallel the block structure of the anticipated hardward. The language permits a specification of sufficient precision so that a hardware realization of the system's logic can be obtained using programmable algorithms.

REGTRAN (REGister TRANsfer) [33] by Edward Peet, Jr., is a modification of DDL. The major modifications were to the method of writing equations, as REGTRAN is designed to be used either on the CDC 6600 computer or by a teletype interfacing with this computer. Therefore, a symbol set had to be chosen compatible to both the teletype and the card reading system of the 6600. This encluded many of DDL's symbols, but substitutes were obtained when they seemed necessary. The DDL's delay declaration was not included in REGTRAN. The syntax of the <ST> statement was modified to permit easier and more versatile specification of conditional register

transfers. Basically, any synchronous digital system which can be described in DDL can be described in REGTRAN. However, asynchronous systems are difficult to describe in REGTRAN because of the unit delay restriction. A detailed version of this language is given in Chapter II.

1.3. Educational Values

A comparison between conventional hardware and software engineering shows structural versus procedural philosophies. That is why in computer science curricula these two disciplines appear as almost completely separate and independent. The consequences of this methodological segregation are well known: bad cooperation between hardware and software engineering, waste of time in education when introducing data processing principles, bad common understanding of hardware and software interaction.

Not many universities--not even the few affluent ones in these days of shortage--can afford to build machines for teaching and experimenting. A complex structure such as a multiprocessor may be described using HDL and yet simple enough that it can be done by one student in one quarter. This is the only possible alternative to teaching by qualitative argumentation. There should be simple, unequivocal simulators along with HDL.

A "user-reorganizable" laboratory computer has been designed at the University of Arizona in terms of AHPL. Student technicians were able to generate wiring diagrams and fabricate the machine directly from the AHPL description. This design language has been the principal means of communication between members of the design team as changes have been made in and features added to the system. The language description of the machine has greatly eased the documentation problem. There has been no pressure to produce detailed English language discussion of every aspect of the machine. As student technicians leave and are replaced by new students, the AHPL description will ensure a minimum of discontinuity in the ability to diagnose and repair failures in the machine.

APL is used in the ALERT system developed in IBM to be used as a front end for their design automation system.

REGTRAN and SYSSIM have been used teaching a course in digital system design at the University of Texas at Austin and continued use for this purpose is anticipated. Both of these programs provide a unit delay simulation, they are most useful for simulating clocked synchronous systems for which gate and wiring delays can be neglected. Simulation of asynchronous systems with gate and wiring delay would be awkward because each delay would have to be simulated by multiple cascaded delays. The REGTRAN language is fairly easy to learn for anyone who is familiar with FORTRAN or a similar

high level language. The REGTRAN and SYSSIM programs have been successfully used to simulate a wide variety of synchronous digital systems. Some examples are twelve-bit parallel Adder-Subtractor, Sequence Detector, State Table Simulator, Serial Multiplier, General-Purpose Computer and an Associative Processor.

Both of these programs are modified to run under the Kronos System at the Oregon State University. These programs are hoped to be useful in teaching digital system design.

The REGTRAN and SYSSIM programs have been successfully used to simulate very complex systems but they fail in simulating a MICRO PROCESSOR as the microinstruction formats can vary from a single encoded micro-operation field and an address field to a format where each field corresponds to a single control gate. In order to overcome this problem MICRO, a simulator for a simple micro-programmable computer, is developed which will simulate the micro instructions. More detailed version about MICRO is discussed in Chapter III. Using this simulator any microprogrammable machine can be simulated without trouble. This program is easy to use and is very useful tool: simulation of PDP-8 is given as an example. Both REGTRAN with SYSSIM and MICRO are hoped to be useful in teaching digital computer design and microprogramming.

II. IMPLEMENTATION OF REGTRAN AND SYSSIM

2.1. Introduction

REGTRAN [33] was originally developed by Charles E. Peet, Jr. at the University of Texas at Austin and was to run on CDC-6400-6600 TAURUS time-sharing system in an interactive mode. It is used in the preliminary design process to specify a system in terms of register transfers and logic equations. The user specifies system structure, using register transfers associated with each state of the system. A special simulation program SYSSIM, is used to simulate and debug the system by supplying inputs described by the user and printing results.

The REGTRAN program reads a source deck written in the REGTRAN design language and generates a Fortran source file which is then compiled by the FORTRAN compiler.

The block diagram of the simulated system is shown in Figure 2.1 in which the description is done by REGTRAN and then it is simulated under the control of SYSSIM.

Each major subdivision of the digital system to be simulated is converted into a separate FORTRAN subroutine, which is used to simulate the networks.

SYSSIM is an interactive FORTRAN program which obtains information from the user on the method of simulation. Then the

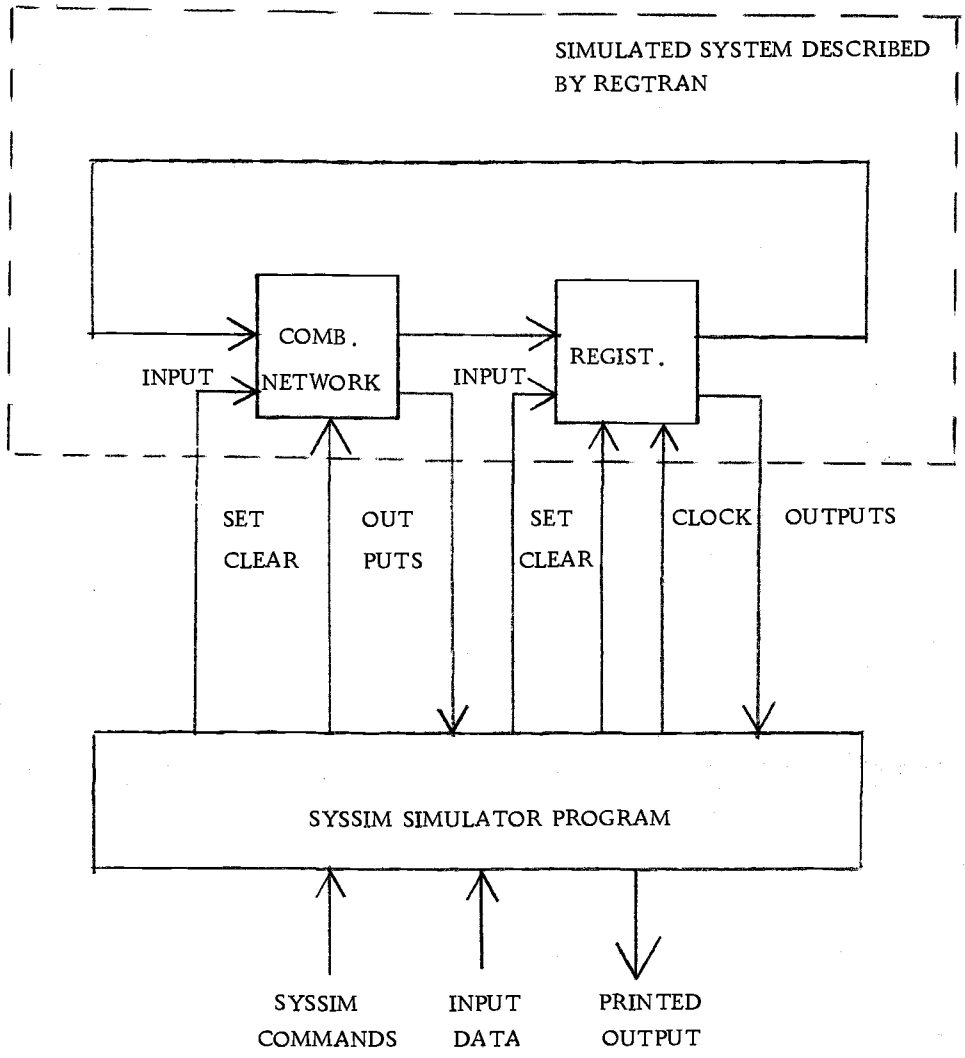


Figure 2.1. Block diagram of simulated system.

Fortran subroutines were called at the proper times to execute the simulation.

Both these programs (REGTRAN, SYSSIM) are provided with a large number of different errors in both description and simulation of the digital system. REGTRAN can print 64 unique error messages and SYSSIM contains 36 error messages. Although error and diagnostic messages are provided for the reasons given, REGTRAN and SYSSIM can not determine if the design objectives are met. This part is left to the operator.

Examples of the types of systems which can be described and simulated are: a combinational networks with a counter to cycle through all input states, sequential networks, arithmetic units (fixed or floating point) and special and general purpose computers.

In Figure 2.1 the combinatorial network responds immediately when an input terminal is changed. The logic equations for the combinatorial network and the conditions under which register transfers can occur are specified in REGTRAN. External set and clear signals, external inputs to the system, and dock pulses are supplied by SYSSIM when the simulation is running.

More detailed versions of the syntax and semantics of the description language REGTRAN and of the simulation language SYSSIM can be found in Reference [33]. Here in this chapter, a brief survey of conventions, statements, SYSSIM commands, memory

operations and a few other details will be discussed.

2.2. Description of REGTRAN

2.2.1. Registers, Terminals, Clocks and Memories

All registers, terminals, clocks and memories must be given alphanumonic names which may be up to seven characters in length and start with a letter. Automata must also be named with an alpha-numeric identifier of three or less characters.

A register is defined as a group of bits which are updated according to a register transfer equation on specified clock pulse. A terminal is defined as a group of bits which are updated continuously according to a boolean equation.

2.2.2. Numbers Representation

Three types of numbers may be used in REGTRAN--decimal, octal, and binary. The default type is decimal, that is, any number not followed by a letter is decimal. B, L are the letters that represent octal and binary.

2.2.3. Expressions and Statements

Expressions are composed of operations performed on memories, registars, terminals, or numbers. Table 2.2.1 gives the

different logical, arithmetic and relational or concatenation. The operations 12-15 are bit-by-bit logical operations. Numbers 6-11 are relational operations which result in all "0's" for false and all "1's" for true. Numbers 2-5 are arithmetic operations which use the CDC 6600 fixed point hardware. The operations are listed in the order of their hierarchy.

Table 2.2.1. REGTRAN operations.

1. Concatenation	$
2. Multiplication	*
3. Division	/
4. Addition	+
5. Subtraction	-
6. Equal	.EQ.
7. Not equal	.NF.
8. Greater than	.GT.
9. Less than	.LT.
10. Greater than or equal	.GE.
11. Less than or equal	.LE.
12. Not	.N. OR .NOT
13. And	.A. OR .AND
14. Or	.O. OR .OR
15. Exclusive-or	.E. OR .EOR.
If the symbol '#' is placed in 1st column then	
Exclusive-or	/
Or	+
And	*
Not	-

Expressions are evaluated as follows: each term of the expression is placed right justified in the CDC 6600 sixty bit word with zero fill. The operations are then performed in the order indicated, placing each result right jusitified in the sixty bit word and may

be truncated so that it will be with-in-58 bits in length.

Table 2.2.2 summarizes the types of statements used in REGTRAN. <SY>, <RE>, <TE>, <ME> and <TI> statements are used to declare system, register, terminal, memory and clock names. The equation in the <BO> statements are used to define the combinational part of the digital system. <OP> together with <PA> and <BO> may be used to define a combinational subnetwork as an operator. Sequential subnetworks (automata) are defined by using <AU> statements. Each automation may have a state sequencing register and one or two conditional registers defined in the <SS> and <CO> statements. The next state and the register transfers associated with each state are specified in an <ST> statement. IF clauses are used to specify the conditions under which the register transfers occur.

2.2.4. Predefined Operators

REGTRAN contains three predefined operators--MOD, JKFF and DECODER. MOD (A, B) is a 48 bit function with two inputs, A and B, and is equal to the remainder when A is divided by B.

JKFF (Q, J, K) is a one bit function with three inputs Q, J, and K, and is defined as

JKFF = J.AND..NOT.Q.OR.Q.AND..NOT.K.

This function may be used to simulate a clocked J-K-flip-flop when

Table 2.2.2. Summary of REGTRAN statements.

Statement	Purpose	Remarks
<SY> name:	defines REGTRAN system name	must be the first statement of the program
<RE> name(a), name(a:b), name	defines register names and number of bits	
<TE> name(a), name(a:b), name	defines terminal names and number of bits	
<ME> name(a, b)	defines memory names, number of bits, and number of words	must precede all <OP> and <AU> statements
<TI> name(a), name(a, b)	defines clock names, period, and offset from time zero	
<BO>x = expression, y(a:b) = expression, Z(a) = expression	contains Boolean equations defining terminals	
<OP> name:	defines an operational subsystem	must precede all <BO> statements (except those under other <OP> statements)
<PA> name, name(a), name(a:b)	defines the dummy input parameters for each operational subsystem	must be the statement immediately following the operational subsystem name
<AU> name:	defines an automaton	must contain one <ST> statement
<SS> name or <SS> name (i)	defines the state sequencing register (SSR) and number of bits (i)	optional in the <AU> statement
<CO> name or <CO> name, name	defines the conditional registers	optional in the <AU> statement
<AS> name = number name = number	assigns numeric values to alphanumeric state names	optional in the <AU> statement
<ST> (see Table 2.2.3)	defines the register transfers	must be the last statement of the <AU> statement
<LO> name = list, name(i) = list, name(i:j) = list	loads a memory or a series of memory locations	if used, must immediately precede <EN>
<EN>	ends the REGTRAN program	

the J and K inputs are specified. If J1 and K1 have been defined in a <BO> statement, the register transfer statement

$$Q1 \longleftarrow JKFF\ (Q1, J1, K1)$$

will set Q1 to the proper next state.

DECODER (M, N) is a variable length function with two inputs, M and N, and is defined by the equation

$$DECODER = 2**(N-M-1)$$

where ** denotes exponentiation.

The DECODER operator decodes a register, or terminal M into a N-bit terminal. For example, if T is an -8 bit terminal and U is a 3-bit register, then

$$T = DECODER\ (U, 8)$$

defines a 3-bit to 8-bit decoder. If U = i, then bit i of T is set to 1. For example if U = 010 then $T = 2^{8-2-1} = 00100000$ and if U = 111, T = 00000001.

2.2.5. Register and Terminal Declaration

They are declared by the following statements:

<RE> $name_1, name_2, \ldots, name_m$.

<TE> $name_1, name_2, \ldots, name_m$.

If the name is simple alphanumeric name, it is only one bit. Multiple-bit registers and terminal may have from 1 to 58 bits. As many names as desired may follow <RE> and <TE>; they must be separated by commas.

2.2.6. Memory Declaration

Memories are declared in the following manner:

<ME> name <W,S> where W is the number of words in the memory ($1 \leq W \leq 8192$) and S is the number of bits in each word ($1 \leq S \leq 58$). Upt to 5 memories may be declared with each name being separated by commas. A memory may be loaded during running of the simulation using a SYSSIM SET command, or it may be loaded in advance by using the REGTRAN <LO> command. The later is particularly useful for a read-only memory where the memory contents is really part of the system description and is not normally changed during running of the simulation. If <LO> statement is used, it must be the last statement in the REGTRAN system description immediately preceding the <EN> statement.

2.2.7. Timing Elements (Clocks)

The periodic clocks which are to control the automata are declared in the following manner:

$$<TI>name_1(p_1, o_1), name_2(p_2, o_2),$$

where $name_i$ is an alphanumeric name as for registers. p_i is the period of the clock and is expressed as a positive integer. o_i is the offset of the clock from time "0"; it is a positive integer less than p_i but may be omitted, in which case the default value is zero.

2.2.8. Boolean Equations

The <BO> statement is a series of equations which defines each terminal as a function of registers, memory words, numbers and other terminals.

$$<BO>\ name_1 = expression_1, name_2 = expression_2, \ldots$$

2.2.9. Operations on Registers and Terminals of Different Lengths

When an operation occurs which would fill a register or terminal with more bits than were reserved for it, the leftmost excess bits are lost. This applies to both boolean statements and register transfers.

2.2.10. Operational Subsystems

The "OPERATOR" represents a group of gates with inputs and an output. It is defined once and then maybe used any number of timbes by specifying only its name and its inputs.

2.2.11. Automata

Each automation has associated with it a clock, a state-sequencing register, a optional set of conditional registers and internal registers and terminals. The operation is defined in terms of register transfers.

A maximum of 20 automata may be declared in a REGTRAN program, but none need to declared. Every automata must have one <ST> statement and it must be the last statement of the automata declaration. The state sequencing register (SSR) controls the state of the automation and is limited to a maximum of eight bits, thereby limiting the maximum number of possible states to 256. It is advisable to declare the SSR of the minimum length necessary to save compiler space.

The syntax of the <ST> statement is given in Table 2.2.3.

A conditional register is one which is tested at various times to determine if certain register transfers should take place. It is used in the conditional IF clauses.

2.2.12. Register Transfer Statements

The register transfer equation describes the new contents to be entered into the register when it is clocked. Unless the register is clocked, its contents will not change. The register transfers are

Table 2.2.3. Syntax of the <ST> statement.

REGTRAN Coding	Remarks
<ST> list of unconditional register transfers	see Note 1 below
[IF clause] list of register transfers	repeat this line as required
[conditional] (list of register transfers	repeat these lines as required
[IF clause] list of register transfers)	repeat this line as required
;	see Note 4 below
state list:	see Note 2 below
list of unconditional register transfers	
[IF clause] list of register transfers	repeat this line as required
[conditional] (list of register transfers	repeat these lines as required
[IF clause] list of register transfers)	repeat this line as required
;	see Note 4 below
state list: etc.	repeat above for each state list

Notes:

1. Register transfers specified before the first state list take place independently of the state of the automaton.
2. A state list consists of one or more state numbers or state names separated by commas (e.g., 2, 3, 8, S4, S5). The state list must be followed by a colon.
3. Parentheses are required enclosing the register transfers which follow a "conditional" as shown above. In addition, any list of register transfers may be enclosed in parentheses.
4. The lists of register transfers associated with each state list must be terminated with ";". (The final ";" at the end of the <ST> statement may be omitted.)
5. Register transfers in a list must be separated by commas.
6. Format is free field and the end of line is not significant.

specified in the <ST> statement. It contains a list of states with each state name followed by the register transfers to occur during that state. All register transfers in a list of register transfers must be separated by commas.

The register transfer statements are of the form

(1) Register name <— expression

(2) Sf(r, n)

where Sf is the shift function, r is the name of the register to be shifted and n is the number of places to be shifted.

2.2.13. IF Clauses

In conditional IF clause, during each state of the SSR, the contents of the conditional registers may be tested and if they are equal to some specified values, the specified register transfer takes place; otherwise they will not. The conditional IF clause is composed of numbers, either single or sequence for each conditional register.

Regular IF clauses may be included to test any register, terminal or memory elements. The register transfers are executed depending on the value of the expression is 0 or not zero.

REGTRAN is a free-field language; that is, each card is a continuation of the previous card. Columns are not distinguished but only the first 72 columns may be used. If an "*" appears in column 1 of any card, that card will be listed and ignored.

2.3. Description of SYSSIM

SYSSIM is a language used to control the simulation of a system described by REGTRAN. A series of SYSSIM commands are executed in sequence to control input and output to the simulation and to set conditions for starting and stopping the simulation. The system described by REGTRAN is normally idle with no clock being supplied to it. All registers, terminals, and memories are initially set to zero. The CLEAR, SET, READ, and INPUT commands are used to set registers, memories or terminals to specified values. These values are held until changed by the REGTRAN system or by another SYSSIM command. The PRINT, DISPLAY and HEADER commands are used to specify the desired simulator output. The RUN, STEP and ASTEP commands are used to supply clock pulses to the simulation. After every clock pulse, all affected registers are updated simultaneously, and then all terminals are updated using the new registers value. The PRINT or DISPLAY commands are then executed if required. At each time step during the simulation, a reserved register labeled TIME is automatically incremented by 1.

SYSSIM uses a free field input format with the restriction that each command must be placed on a separate line. To indicate that a command is continued on the next line, "#" is placed in any column.

REGTRAN reads an input deck composed of source input and generates a FORTRAN source program to describe the system to be simulated. Upon the successful compilation of the source input by REGTRAN, the CDC 6600 FORTRAN compiler is called to compile the FORTRAN just generated. This FORTRAN contains only subroutines and functions which are called from SYSSIM. The subprograms generated are grouped into three functional groups:

1. Boolean Equation Evaluation--This set of subroutines calculates and updates the values for all the terminals in the system.
2. Register Transfers--This set of subroutines calculates the values to which the registers will be set upon the application of the next clock pulse.
3. Memory Manipulation--SYSSIM does not have direct access to the memory words because it has no previous knowledge of the size of the memories. A set of subprograms is therefore provided to allow memory manipulation. The memory manipulation subroutines are called by SYSSIM when the SET, CLEAR, and PRINT commands which reference memory are used.

Table 2.3.1 consists of a list of the subprograms generated by RETRAN and their functions.

Table 2.3.1. FORTRAN subprograms generated by REGTRAN.

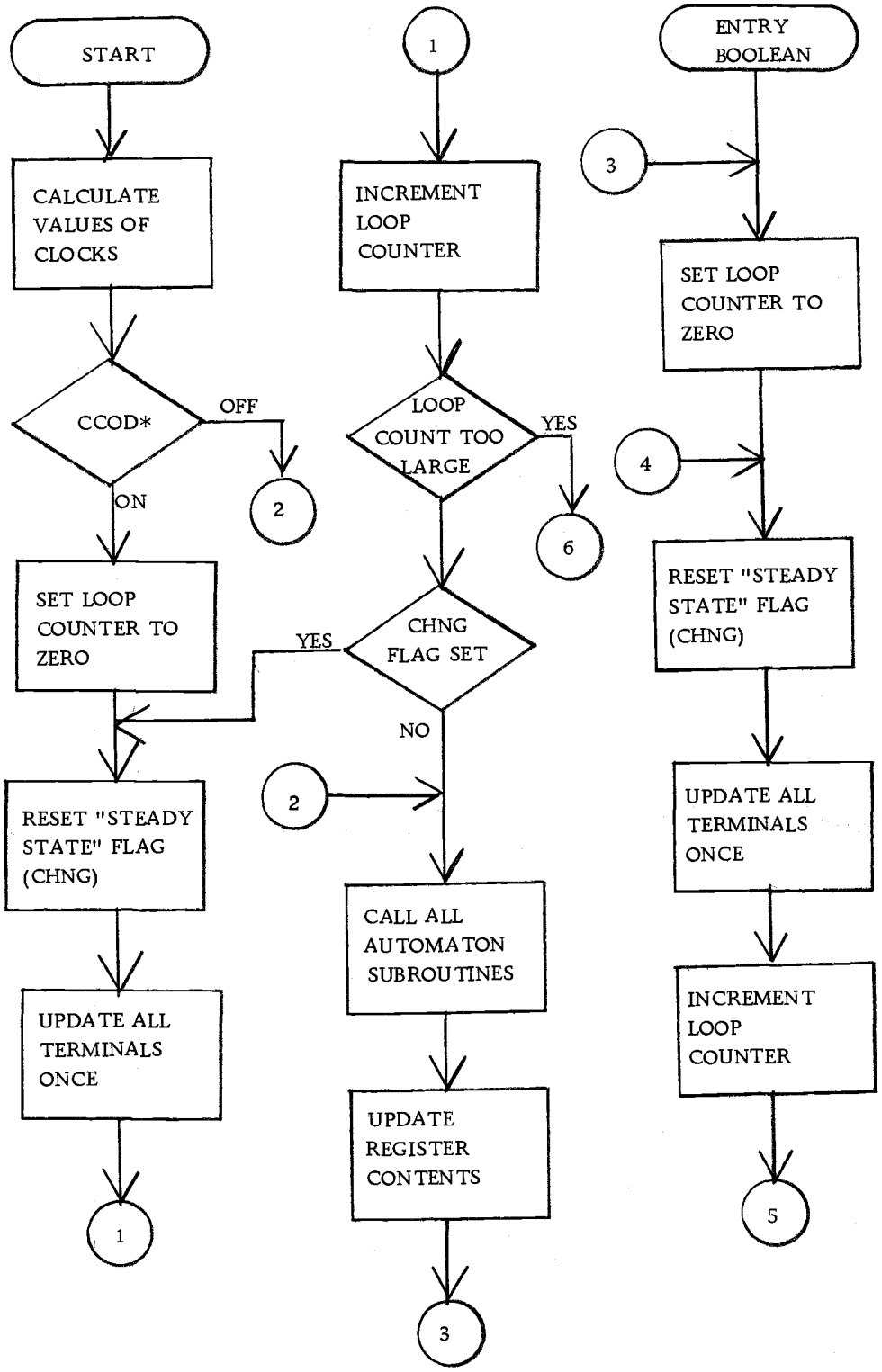
Subprogram Name	Subprogram Type	Purpose
COMBOOL	SUBROUTINE	Stores all boolean equations declared in common (<BO> statements before all <AU> statements)
Named after automaton	SUBROUTINE	Stores all register transfer equations for the automaton after which it was named
Named after automaton except followed by "ENT"	entry point of above subroutine	Stores all boolean equations declared in the automaton after which it was named
CONTROL	Subroutine	Called from SYSSIM simulator once each time step to update all registers and terminals. Uses the above three types of subprograms to accomplish this
BOOLEAN	entry point of CONTROL	Called from SYSSIM to update all terminal contents
REGSET	SUBROUTINE	Contains DATA statements to pass register and terminal names and the number of bits in each through COMMON to SYSSIM
MDATA	SUBROUTINE	Contains DATA statements to pass information concerning memories, including data entered in an <LO> statement
MSET	SUBROUTINE	Called from SYSSIM to set one word, in memory. Input parameters specify which memory, which word, and the data to be entered in that word.
MPRT	FUNCTION	Used to return to SYSSIM the contents of one memory word. Input parameters specify which memory and which word.

The controller, CONTROL is called from SYSSIM once per time unit. It is the responsibility of this subroutine to calculate the states of all clocks declared in the REGTRAN source input as well as the system clock, CLK.

However, it is the responsibility of each automation subroutine to determine, when called by this controlling subroutine, whether its controlling clock is in the proper state. If not, no calculations are performed. Upon calculating clock values, various subprograms are called by CONTROL to perform register transfer calculations and terminal updates. CONTROL also contains an entry point, BOOLEAN, which will only update the terminals without affecting the registers.

The flow chart for the CONTROL is given in Figure 2.2.

The conversion of the REGTRAN source program in Fortran subprograms is done by the use of an interpreter, a set of Floyd-Evans productions, a "Last-in-First out" stack, a lexical scanner, an error printing subroutine and an executive subroutine. The interpreter checks the syntax of the input program and controls the other elements listed. The lexical scanner is used to scan the symbols from the input source and convert them into recognizable elements such as number, alphanumeric identifiers, symbols, and reserved words. Each time the scanner is called, it places the new element into the top of the "last-in-first-out" stack.



* CCOD is the flag which indicates that a SET or CLEAR command has been executed since the previous simulation.

Figure 2.2. Flow chart for CONTROL.

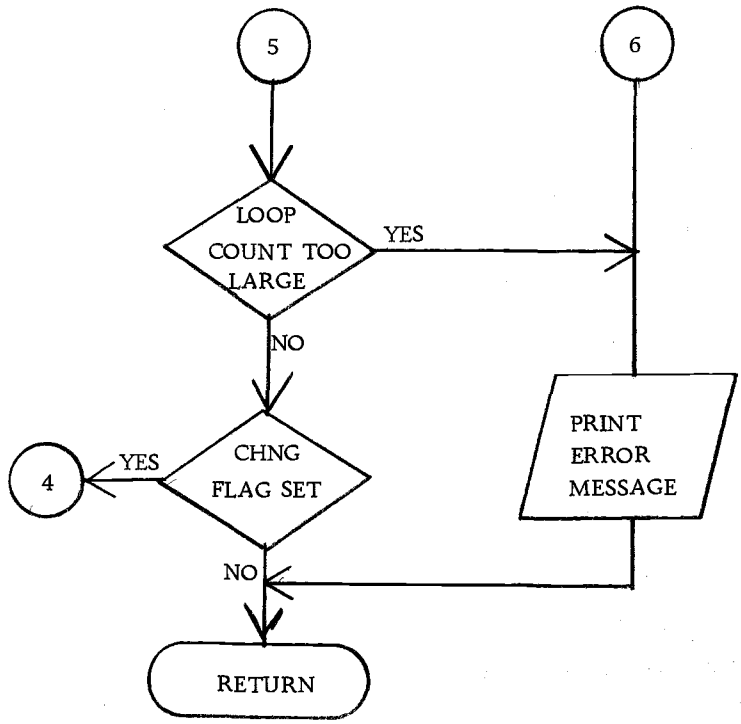


Figure 2.2. Continued.

The interpreter uses the stack along with the Floyd-Evans productions to determine if the input syntax is correct and to initiate any action necessary as a result of the input. The executive routine is used to generate the FORTRAN subprograms which will be linked with SYSSIM. All bookkeeping is also performed by the subroutine. The flow chart for REGTRAN is given in Figure 2.3.

REGTRAN contains TRACE feature, which is used to trace the progress of the interpreter. It is turned on by placing the symbol "#" in column 1, 2, and 3 of an input card. Auxilary trace is turned on by "#" in column 1, 2 and 4. The main and auxilary trace features can be turned on simultaneously by placing the symbol "#" in columns 1, 2, 3 and 4 and can be turned off by "#" in columns 1, 2 and 5.

The main program, SYSSIM acts as a command decoder, its flow chart is given in Figure 2.5 and the sequence of operations for the SYSSIM simulator are shown in Figure 2.4.

2.4. Implementation of REGTRAN and SYSSIM for KRONOS

REGTRAN and SYSSIM were written to run under the TAURUS operating system at the University of Texas at Austin. These two programs are implemented for KRONOS system at Oregon State University at Corvallis, Oregon. During this process, a lot of problems were faced and they will be discussed in detail in this section.

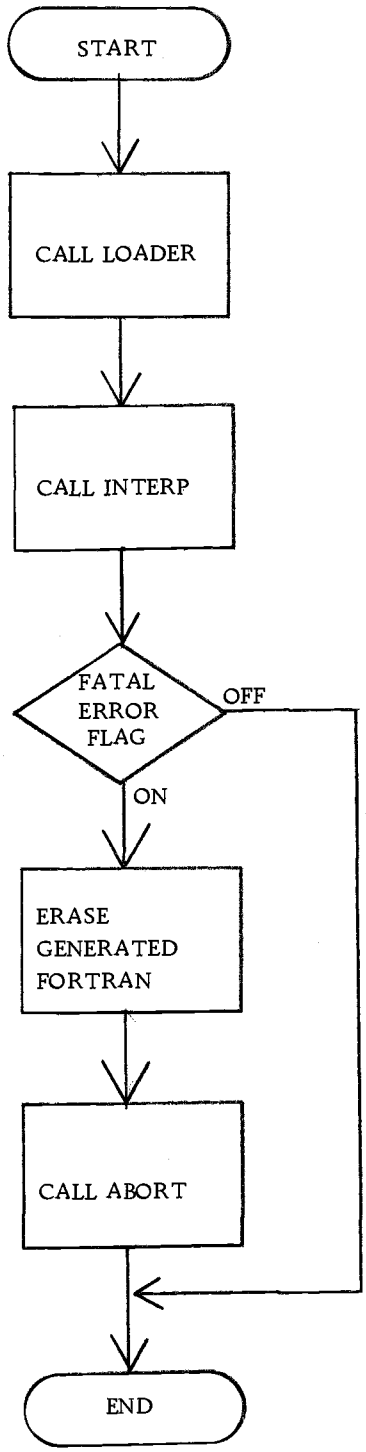
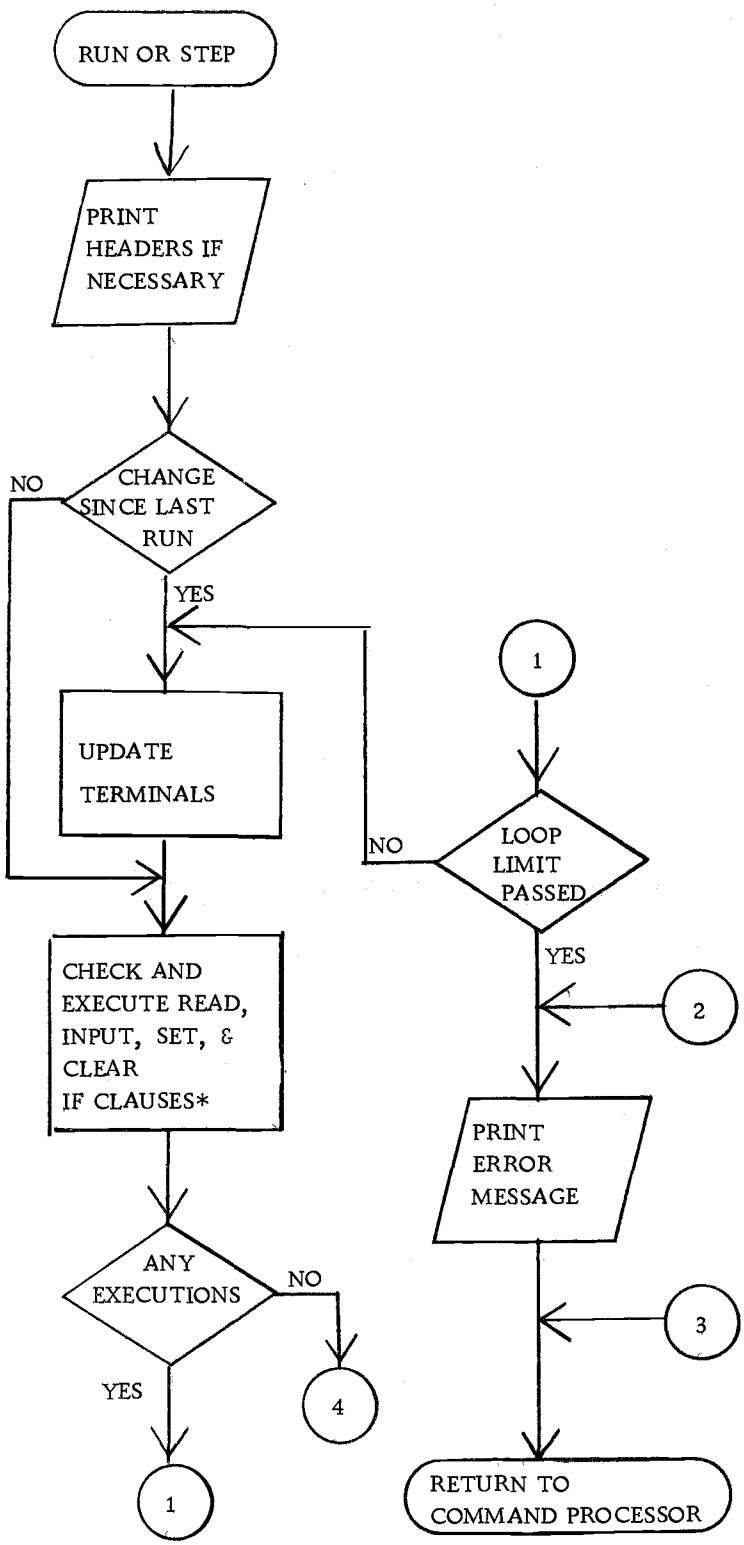
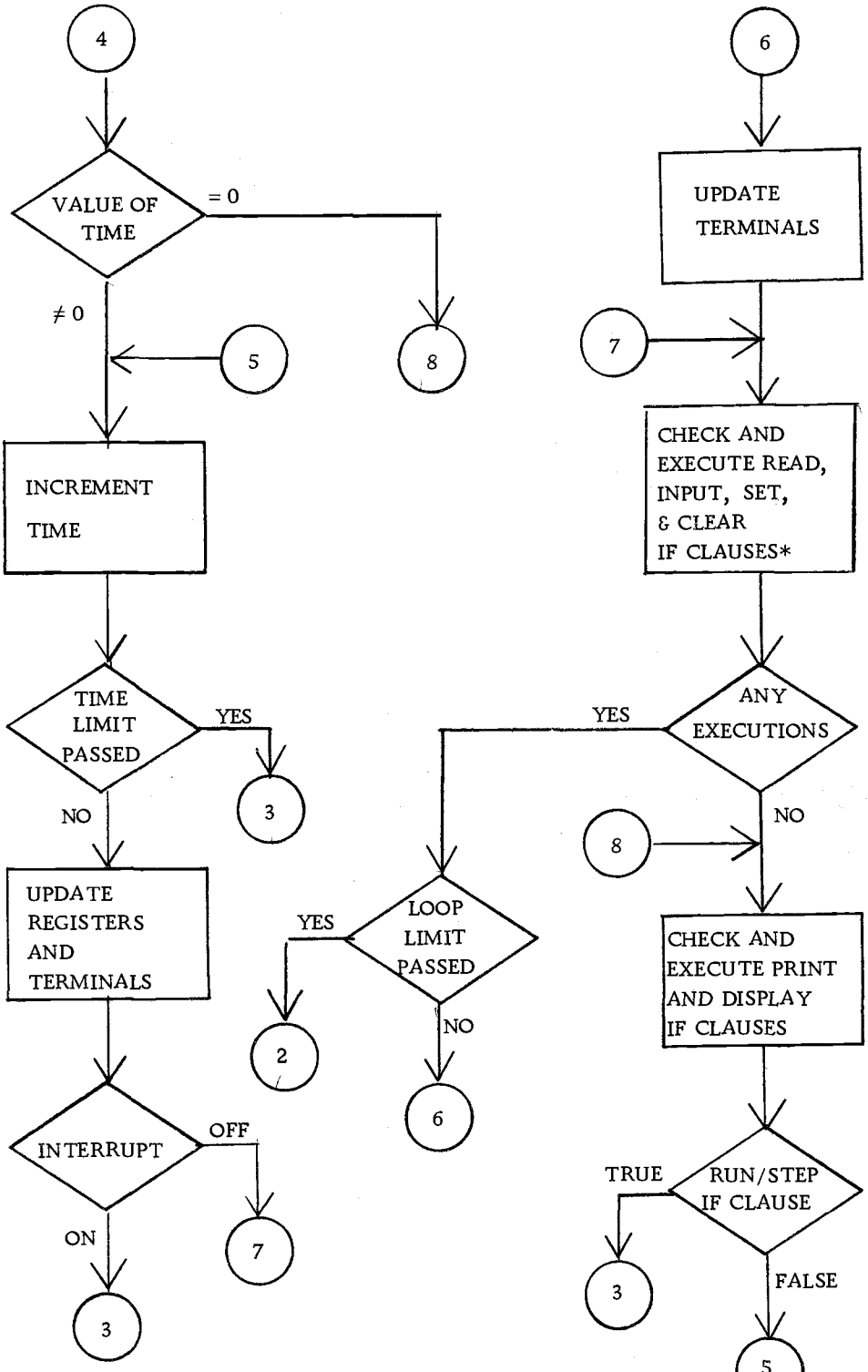


Figure 2.3 Flow chart for REGTRAN.



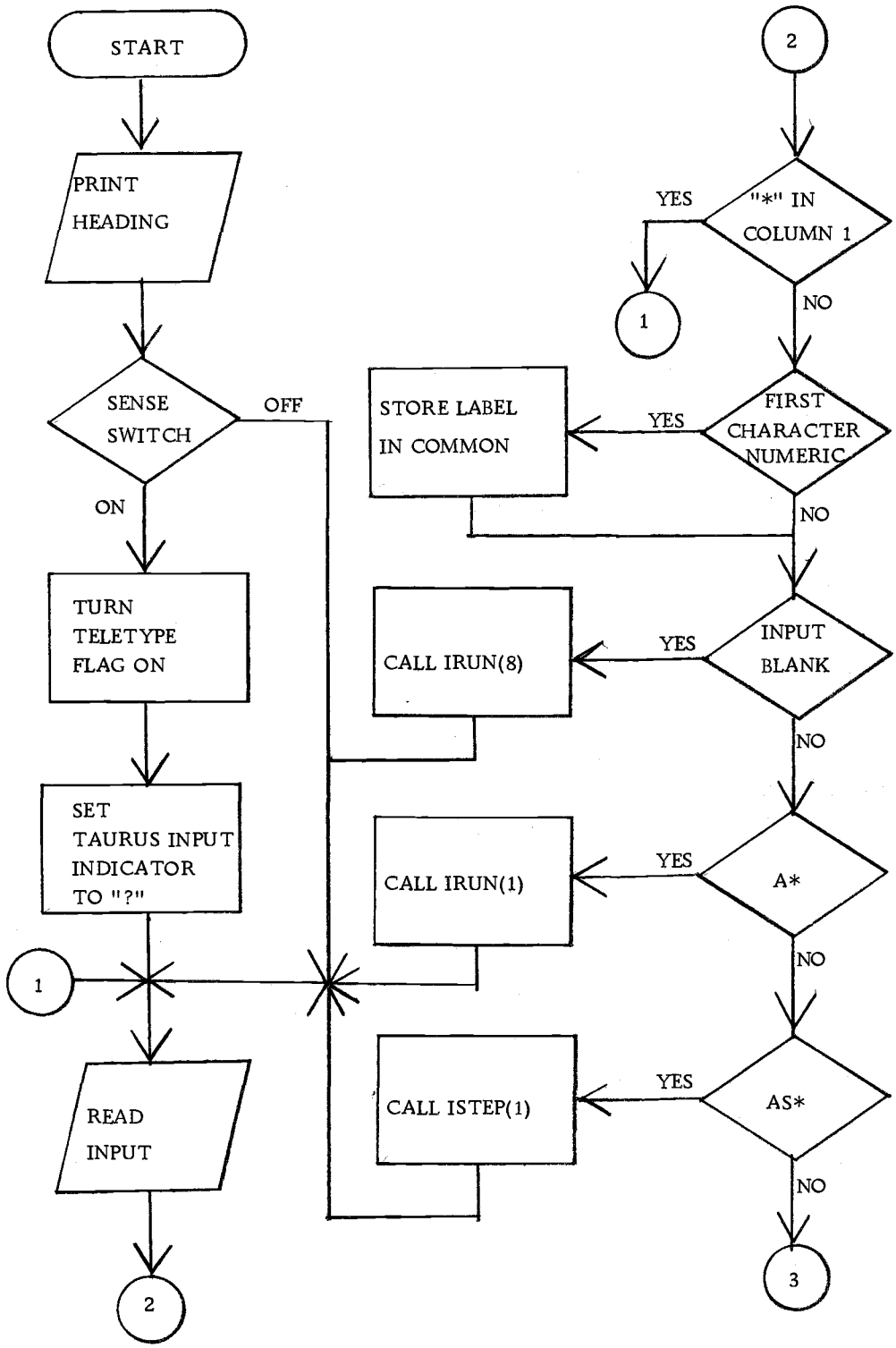
* Each of these IF Clause commands can be executed only once per RUN or STEP command.

Figure 2.4. Sequence of operations for the SYSSIM simulator.



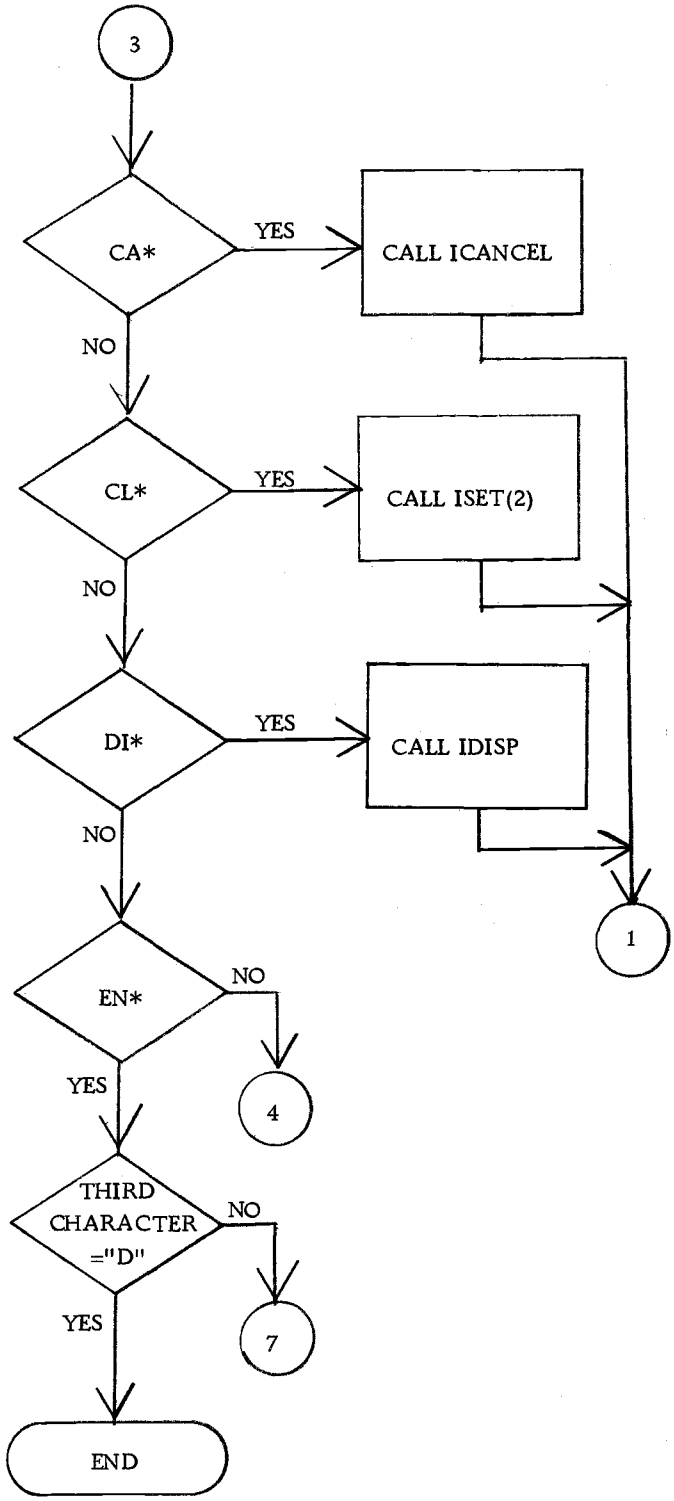
* Each of these IF Clause commands may be executed only once per time step.

Figure 2.4. Continued.



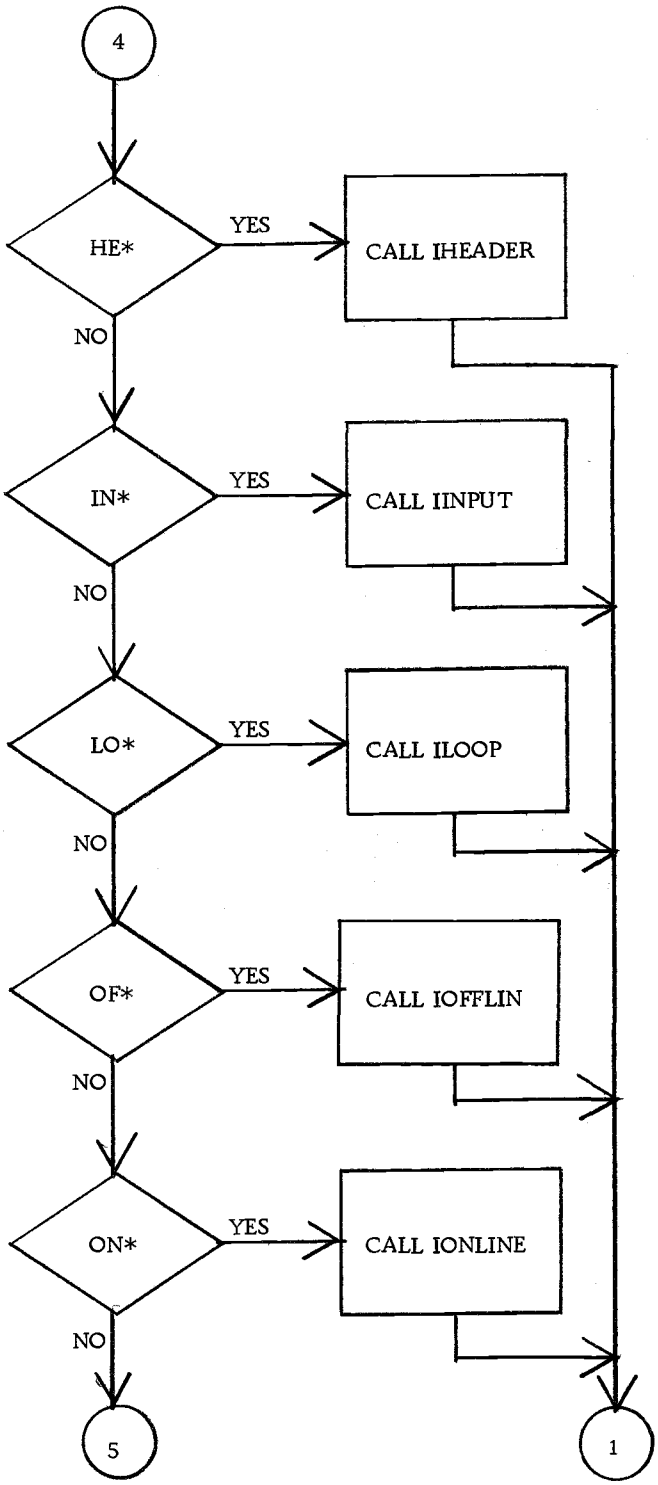
* These characters refer to the first alphanumeric characters in the input record.

Figure 2.5. Flow chart for SYSSIM.



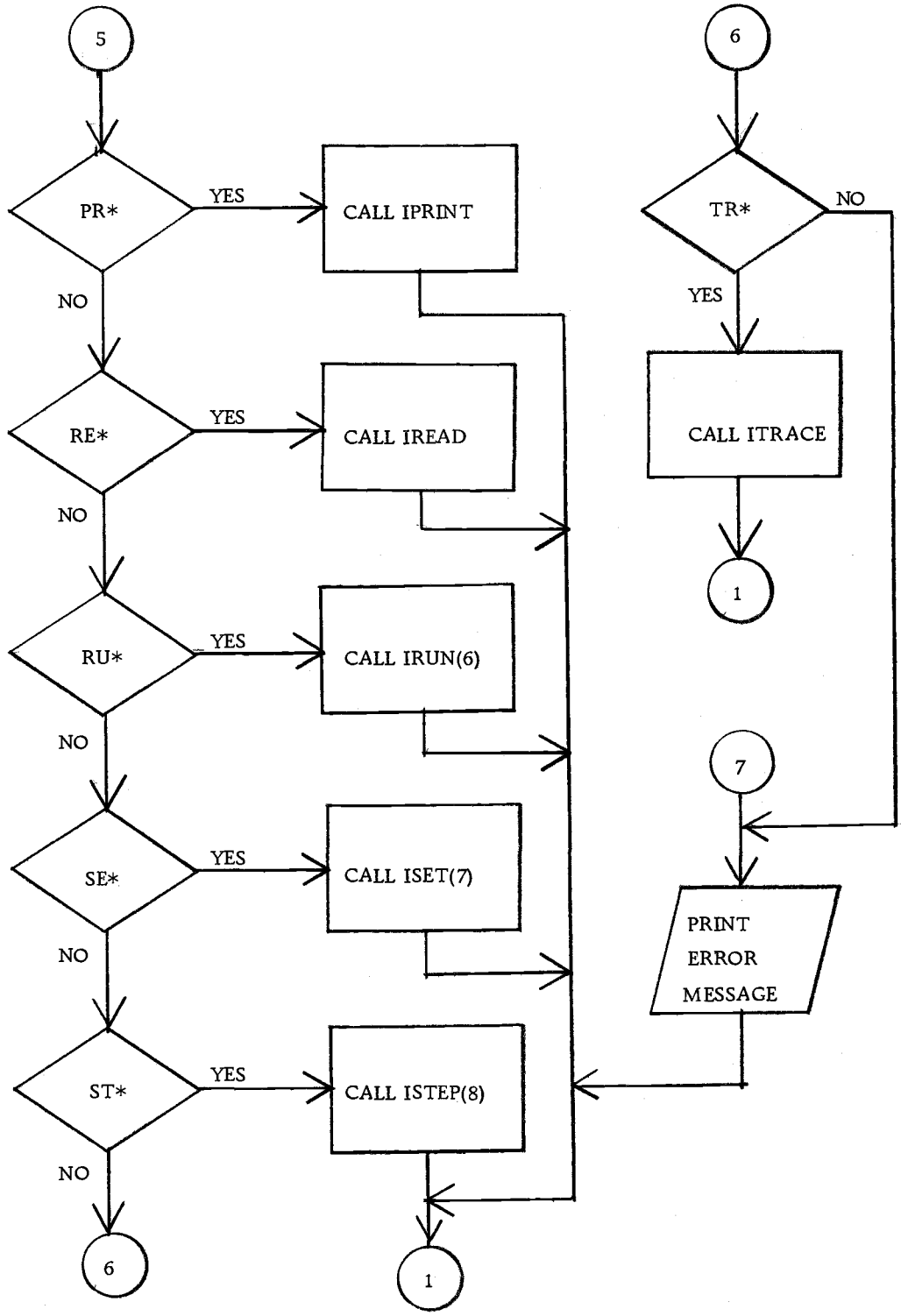
* These characters refer to the first two alphanumeric characters in the input record.

Figure 2.5. Continued.



* These characters refer to the first two alphanumeric characters in the input record.

Figure 2.5. Continued.



* These characters refer to the first two alphanumeric characters in the input record.

Figure 2.5. Continued.

TAURUS is the operating system which is written by the University of Texas computer center staff to run on a CDC 6400 computer. KRONOS is the operating system supplied by the CDC corporation to run on CYBER-73. CYBER-73 is a new version of the CDC-6400 with some hardware differences. TAURUS is basically derived from the CDC SCOPE operating system, as was KRONOS, with some features added and some features being removed. ASCII code differences exist between KRONOS and TAURUS. The library routines are the same with some name differences. For example LSHIFT in TAURUS performs the same shift operation as SHIFT in KRONOS. Both the operating systems use SCOPE loader. The loader at TAURUS operating system has the added feature of loading multiple files. KRONOS has two types of loaders, LINK and LOAD. Neither of these can load multiple files. In order to achieve this, the files have to be made library files by using LIBGEN command. The control language is different for KRONOS and TAURUS. For example "LABEL" is not allowed in KRONOS.

TAURUS uses RUN compiler and KRONOS uses FTN compiler. In the RUN compiler, logical and non logical mixed operations are allowed. In the FTN compiler, these types of mixed operations are identified as "FATAL ERROR" and the successful compilation of the program is stopped. RUN compiler allows DATA statement anywhere, before an entry point of the program, but the FTN compiler imposes a

condition that the DATA statement has to be only after the declarative statements and before the entry point of the program, otherwise it will be identified as FATAL ERROR, which interferes with the successful compilation of the program.

The two problems encountered are the differences between the two operating systems and the insufficient documentation available about REGTRAN and SYSSIM. KRONOS as well as these two programs were new and some time was spent in studying the programs as well as learning how to use KRONOS.

REGTRAN uses a Floyd-Evans production table, for the purpose of stack manipulation in the process of compiling equivalent FORTRAN from the REGTRAN statements. REGTRAN uses various ASCII code comparisons. SYSSIM also does the same thing. In both these programs such comparisons are changed to the equivalent ASCII of KRONOS system. All the library routine calls of TAURUS are changed to the equivalent library calls of KRONOS. Some of the TAURUS calls were also changed.

The problem of mixed operation error is taken care of by reframing the description of the problem in REGTRAN language such that mixed operations will not be generated in the corresponding FORTRAN generated by REGTRAN. This required the REGTRAN syntax to be somewhat more restrictive.

The problem of the DATA statement was taken care of by writing another program RTEST. This program scans through the FORTRAN generated by REGTRAN and removes the DATA statement that occur between the header of the program and the declarative statements and inserts them after the declarative statements and before the executable statements. Flow chart for RTEST is given in Figure 2.7.

REGTRAN and SYSSIM can be run ONLINE or OFFLINE. Subroutines IONLINE and IOFFLINE in SYSSIM are used to execute the ONLINE and OFFLINE commands respectively. IONLINE sets the teletype flag to make it ONLINE.

In order to run this on KRONOS, different methods were tried. First, the binary object files of SYSSIM and the binary files of the compiles FORTRAN file generated by REGTRAN were loaded. Unfortunately, the optimization feature of the KRONOS FTN compiler, seems to drop off subroutines which do not contain any active or executable statements from the FORTRAN produced by REGTRAN. When SYSSIM refers to these dropped subroutines, it will not be able to find them and the load error "unsatisfied external references" stops the execution of the program.

The second method tried, consists of joining the FORTRAN generated by REGTRAN at the end of the FORTRAN source program of SYSSIM by using "MERGE" command in EDIT mode. Then the final file which is created, is compiled using FTN compiler. This method

seems to successfully execute the program, but it is very expensive and inefficient. Every time a new problem will be tried, the SYSSIM needs to be joined with the FORTRAN of the problem and then compiled. So the total time needed to compile and execute will be an average of 20 to 25 CP seconds for a moderately simple problem. This method is obviously inefficient and expensive.

The third method is using a KRONOS control language macro call. The name of the macro is REGRUN. A listing of the REGRUN is given in Figure 2.6. This consists of a set of system commands, that call the REGTRAN and SYSSIM programs binary files along with the other necessary files such as the production table. From the generation and compiling of Fortran to execution of the compiled binary file with the SYSSIM commands, all can be done just by using REGRUN. This macro can be called both ONLINE and OFFLINE also and depending upon the setting of the starting point, this REGRUN will start execution online or offline. For example: To run REGTRAN and SYSSIM is shown in the following sections.

```
SET (R1=0)
3, GET, TAPE2/UN=80035.
GET, BREG/UN=80035.
OFFSW, 2.
REWIND, RFOR, TAPE2.
IF (R1=1) GO TO, 2.
1, RFL, 100000.
BREG, RIN, ROUT, RFOR, TAPE2.
REWIND, RFOR, RB.
FTN, I=RFOR, L=RL, B=RB.
GET, BSYS/UN=80035.
LIBGEN, F=RB, P=ULIB, N=ULIB.
REWIND, BSYS, ULIB.
LOAD, BSYS, ULIB.
EXECUTE.
GOTO, 5.
2, REWIND, RIN, RFOR, TAPE2.
ONSW, 2.
GOTO, 1.
4TTY, SET (R1=1)
GOTO, 3.
5, OFFSW, 2.
```

Figure 2.6. Listing of REGRUN

2.4.1. ONLINE

With the account number and password assigned, the user logs on the KRONOS and the system responds by "READY" appearing on the terminal. The user has to type

/GET, REGRUN, UN=__*__, PW=__*__

*User number and password for REGRUN file, allows user to obtain a read-only copy.

The system responds with

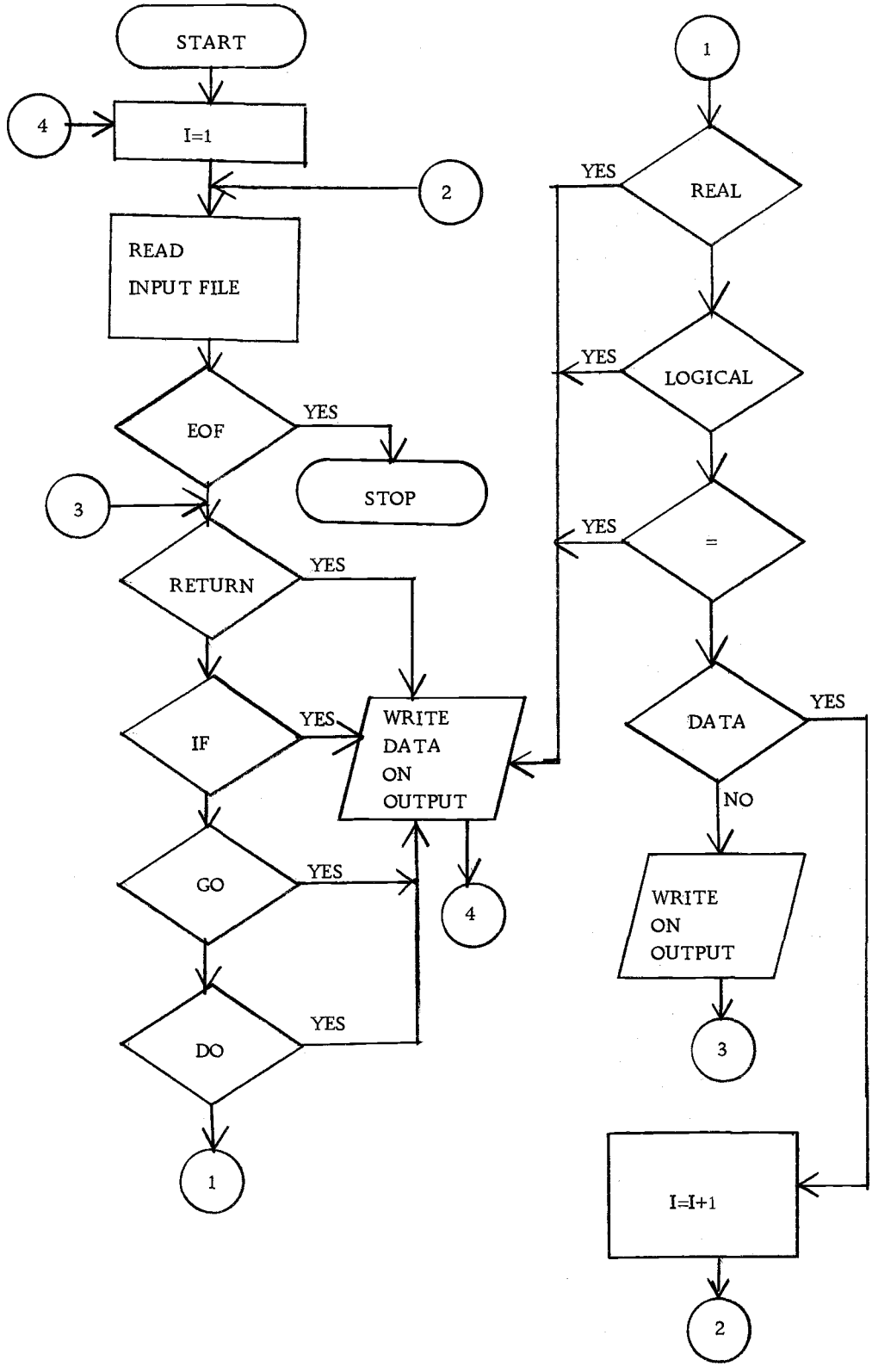


Figure 2.7. Flow chart for RTEST.

/CALL, REGRUN, C, S=4TTY, RENAME, RIN=TEST,

ROUT=OUTPUT

where

C - clears all pointer and renames

S - starting point, which starts at lable no. 4 in REGRUN

RIN - name used in REGRUN for input file

Test - name of the input file

ROUT - REGTRAN output which consists of test and error messages given by REGTRAN, if there were any.

After the successful execution of this macro REGRUN, the system responds with header printout as "SYSSIM SIMULATION PROGRAM" and with a "?" mark and waits for SYSSIM commands. After issuing the right SYSSIM commands, the answeres will be printed.

RIN=INPUT (allows REGTRAN program to be entered on line).

2.4.2. OFFLINE

JOBIDEN, CM55000, T100

ACCOUNT, [USER NUMBER], [USER PASSWORD]

GET, REGRUN/UN=80035.

CALL, REGRUN, C, RENAME, RIN=INPUT, ROUT=OUTPUT.

$^{7}_{8}_{9}$

REGTRAN INPUT.

7_{8_9}

SYSSIM COMMANDS which should have OFFLINE as first command

$6_{7_{8_9}}$

Using macro call to run the program, requires 2 to 3 seconds of CP time. Since SYSSIM does not need to be compiled for every example individually. Hence this method seems to be more efficient and inexpensive.

The REGTRAN and SYSSIM programs are totally dependent on the Floyd-Evans production table. As a consequence of the ordered set of rules for the creation of FORTRAN from REGTRAN language, even a minute mistake in the creation of source file causes syntax error and wipes off all the FORTRAN thus generated. Because of the ASCII differences, the same example problem that ran on TAURUS gave trouble on KRONOS. TRACE feature in both of these programs helped to understand and analyze this problem. If these two programs were rewritten that would have been easier than going through someone elses logic with insufficient documentation.

In the following paragraphs, an example will be shown completely.

2.4.3. Examples

Twelve-Bit Parallel Adder-Subtractor. This example demonstrates the simulation of a combinational network. No automation is defined, and therefore no registers are declared. This has two twelve-bit inputs, A and X, which are added or subtracted to form the twelve bit sum, S. Another input K is set to 0 or 1 to subtract A from X. A one-bit-output, OVERFLO, is used to indicate overflow. A complete listing consisting of the description of parallel adder-subtractor in REGTRAN, the FORTRAN generated and the SYSSIM commands along with the TRACE features demonstrations and the results are shown in Appendix A.

Sequence Detector. Another example shown is the sequence detector. This used J-K flip-flops (A, B and C). It has one input, X, and one output, Z, which is 1 whenever the input sequence 101 or 0110 is detected. A complete listing of the example is also shown in Appendix A. The simulator output shows the simulation results for 14-bit input sequences.

III. IMPLEMENTATION OF MICRO

3.1. Introduction

Microprogramming was first introduced by M. W. Wilkes to overcome the difficulty of designing the control unit of a complex computer system as a sequential network. Microprogramming can be defined as

> . . . a technique for designing and implementing the control function of a data processing system as a sequence of control signals, to interpret fixed or dynamically changeable data processing functions. These control signals, organized on a word basis and stored in a fixed or dynamically changeable control memory, represent the states of the signals which control the flow of information between the executing functions and the orderly transition between these signal states.

In the past decade, microprogramming has changed from a machine implementation process for large computing devices to a widespread design practice covering the full spectrum of machines as measured by their size, performance and cost. The flexibility and ease of change of microprogrammed machines allows the designer to delay commitment to a particular control process until much later in the design process than had previously been the case. This same flexibility has significantly reduced the design time and installation cost of engineering changes. Readily implemented control sequences give the designer an opportunity to consider a much richer repetoire of instructions and commands directed toward efficient solutions to

the problems for which computation is being done. The solutions can be less heavily dependent on the exact hardware on which processing is to take place.

While the initial applications of microprogramming were a fairly straightforward replacement of "random" control logic with a control storage element, an immediate expansion has occurred into the areas of better hardware diagnosis tools. Simultaneously, the emulation of predecessor machines on newer technology emerged as an important practice--one which has grown today into a major microprogramming application.

A microprogram consists of a collection of microinstruction. These fields correspond to one or more micro-operations. Micro-operations are the fundamental operations of the hardware and include the following:

1. Activating a data path
2. Initiating an arithmetic operation
3. Initiating a data transfer
4. Testing a condition

Microinstructions and opperations are intimately tied to the system timing. Micro instruction formats can vary from a single encoded micro-operation field and an address field to a format where each field corresponds to a single control gate. The former format is essentially similar to decoding a machine instruction. Many

microinstructions would be required to implement a given operation with only a few encoded micro-operation fields. This type of format is often called vertical microprogramming in reference to the length of the microprograms. The cycle time would be short since only a few micro-operations can be performed with a single microinstruction. In the other extreme, all possible micro-operations can be specified with a single microinstruction. This is called horizontal microprogramming due to the width of the microinstructions. Here, the major cycle time is much longer since each microinstruction must be sequenced with minor cycles to avoid conflicts. Wilkes scheme is an example of horizontal microprogramming. Most implementation of microprogramming lie somewhere in between the extremes. Mutually exclusive micro operations are grouped in one encoded micro-operation field.

Microprogramming can be implemented with a decoding tree and plugboards or diode arrays as in Wilkes scheme, however, semiconductor Read Only Memeories are in present use.

In this chapter we will be considering a computer system which is shown in Figure 3.1. The simulator for this system "MICRO"; is written in FORTRAN. The control section uses ROM. The current contents of the instruction register (IR) will be decoded by decoder, these are used by the address logic to calculate the next micro instruction address. The address is stored into the ROM address

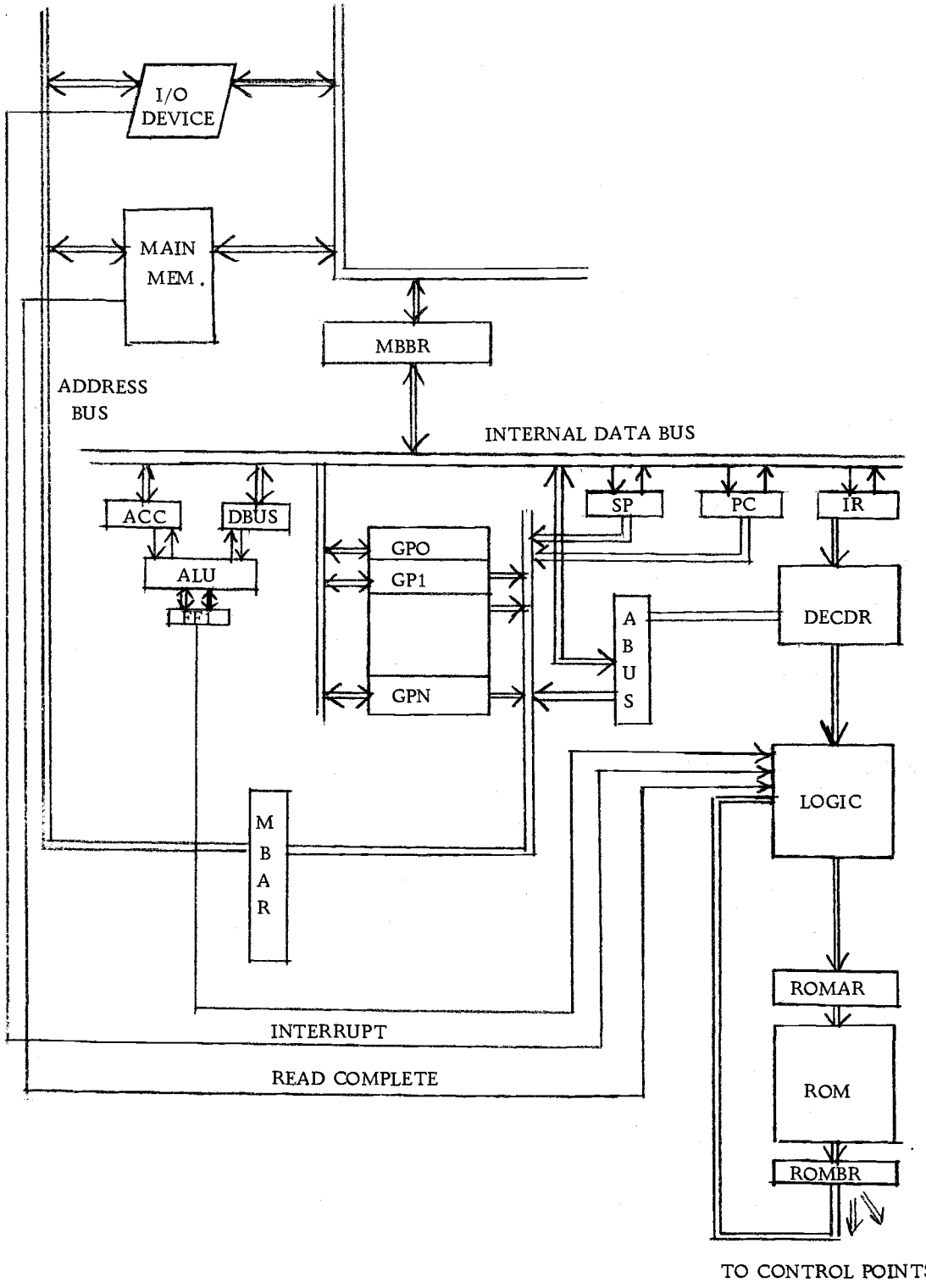


Figure 3.1. Computer system under consideration.

register (ROMAR) and the next micro instruction is read into the ROM buffer (ROMBR). The micro operation fields are decoded and sent to the control points of the system.

3.2. Description of MICRO

MICRO (a simulator for a simple microprogrammable computer) is a FORTRAN program, written to run under the system KRONOS at Oregon State University.

MICRO can be used as a simulator at two levels. (1) As micro instruction formats can be a single encoded micro-operation field and address field, the micro instructions can be direct. The encoded micro instructions can be used to simulate a microprogram. MICRO can be a simulator for microprogram. (2) MICRO simulates any microprogrammable processor, using ROM control logic and DECODER logic. This consists of three major sections; main memory, ALU and control storage. The dimensions of the main memory can be specified by the user according to his need and requirement of the machine he is trying to simulate. But the number of bits in a main memory word can not exceed more than 60 bits as KRONOS will allow only 60 bits. It has six associated registers, memory address register (MBAR), a memory buffer register (MBBR), a program counter (PC), an instruction register (IR), a stack pointer (SP) and an address bus register (ABUS).

The size of all the six registers is not fixed and can be specified to the requirement of the user. ALU consists of an accumulator (ACC) and a temporary data bus register (DBUS) and a set of general purpose registers. The control section contains a "Read-Only-Memory" of 15 bit word length and the number of words is not fixed and can be specified by the user, along with an address (ROMAR) and buffer (ROMBR) registers. In this system the size of the accumulator, general purpose registers and other temporary registers are not fixed sizes, in order to provide this system with a flexibility, with which the user can easily set up the sizes, fitting to his requirements and need not worry of the complexity of his machine adjusting to this system. It provides ease and less problems in simulation of other microprogrammable machines on the simulator.

3.2.1. Microinstructions

The microinstructions are divided into two groups. Group (1) microinstructions manipulate the contents of the accumulator. These instructions can not be combined with the other instructions of group (1). Group (1) also does the input, output operations.

3.2.1.1. Group (1). The group (1) microinstruction format is shown in Figure 3.2 and the microinstructions are explained in the succeeding paragraphs.

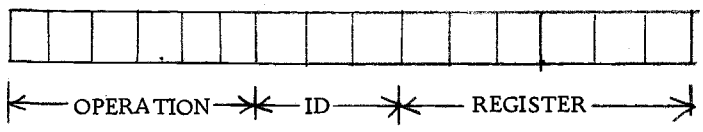


Figure 3.2. Group (1) format.

Operation Codes. The ROM word consists of five octal-digits, occupying a total of 15 bits. The first two digits denote the operation code.

1. No Operation (NO OP)

   Octal code: 00

   Operation: This command causes a 1-cycle delay in the program before the next sequential instruction is initiated. This command is used to add execution time to a program. The NOP also provides the programmer with a convenient means of removing an instruction.

2. Subtract From the Accumulator (SUB)

   Octal code: 01

   Operation: The contents of the temporary register are subtracted from the accumulators and the result is left in the accumulator and the original contents of

the accumulator are lost.

3. Subtract the Accumulator (SUB)

   Octal code: 02

   Operation: This operation is the same as the previous one except in this the contents of the accumulator are subtracted from the temporary register's contents.

4. Addition (ADD)

   Octal code: 03

   Operation: The contents of the temporary register are added to the contents of the accumulator. The result of this addition is held in the accumulator, the original contents of the accumulator are lost.

5. Complement Accumulator (CMA)

   Octal code: 04

   Operation: The contents of the ACC are changed to the two's complement of the current contents of the ACC.

6. Increment Accumulator (INCA)

   Octal code: 05

   Operation: The contents of the accumulator are incremented by one.

7. Increment Temporary (INCT)

   Octal code: 06

   Operation: The contents of the temporary register are incremented by one.

8. Increment Programcounter (INC PC)

   Octal code: 07

   Operation: The contents of the program counter are incremented by one.

9. AND

   Octal code: 11

   Operation: The AND operation is performed between the contents of the accumulator and the contents of the register denoted by the address field. This instruction is often called extract or mask, can be considered as a bit-by-bit multiplication.

10. OR

    Octal code: 12

    Operation: Same as above, except the operation performed will be OR.

11. EXCLUSIVE OR

    Octal code: 13

    Operation: The operation here will be exclusive OR.

12. COMPLEMENT

Octal code: 14

Operation: The contents of the address in the address field are complemented.

13. Right Shift (RS)

Octal code: 15

Operation: The contents of the accumulator will be shifted to the right one binary position and overflow flag is set if there occur any.

14. Left Shift (LS)

Octal code: 16

Operation: The contents of the accumulator are shifted to the left one binary position. Left overflow flag is set, if there occurs any.

15. READ (Memory Refference Instruction)

Octal code: 21

Operation: Information from main memory can be read in this operation.

16. WRITE (Memory Refference Instruction)

Octal code: 22

Operation: Information can be stored in the main memory using this operation.

17. Clear the Accumulator (CLA)

    Octal code: 31

    Operation: The contents of each bit of the AC is cleared (made equal to 0).

18. Change the Signbit of Accumulator

    Octal code: 32

    Operation: The sign of the accumulator will be changed.

19. Clear Right Overflow Flag

    Octal code: 33

    Operation: Clears the right overflow flag.

20. Clear Left Overflow Flag

    Octal code: 34

    Operation: Clears the left overflow flag.

The third digit in the ROM word is the identification digit which indicates whether the operation is right or left justified, whether the temporary location is a data bus or a address bus and whether the operation is on bus or off bus. Table 3.2.1 shows a summary of the action of W_3, the identification digit. The last two digits indicate the register number.

Table 3.2.1. Significance of identification digit.

W_3 ID Bits Position	Significance
000	On Bus, Data, Right Justified
001	On Bus, Data, Left Justified
010	On Bus, Addr, Right Justified
011	On Bus, Addr, Left Justified
100	Off Bus, Data, Right Justified
101	Off Bus, Data, Left Justified
110	Off Bus, Addr, Right Justified
111	Off Bus, Addr, Left Justified

In addition to the above 8 registers, there are general purpose registers, a maximum of 24, which will be denoted by W_4W_5 from 09 to 32. Table 3.2.2 shows the registers and their names which will be used in the MICRO.

Table 3.2.2. Register names in MICRO.

W_4W_5 In Octal	Register They Indicate	
00	Instruction	IR
01	Accumulator	ACC
02	Memory Buffer	MBBR
03	Memory Address	MBAR
04	Data Temp	DBUS
05	Address Temp	ABUS
06	Stack Pointer	SP
07	Program Counter	PC

One example of this type of instruction format is given below.

$W_1W_2W_3W_4W_5$
0 3 0 17

This can be translated as the addition of the contents of the general purpose register 7 with the contents of the accumulator and the temporary register used for this is data register and the transfer is right justified. In conclusion

[GP7] $\longrightarrow$ TEMP

[TEMP] + [ACC] $\longrightarrow$ ACC

3.2.1.2. Group (2) Microinstruction Format. The group (2) format is shown below in Figure 3.3.

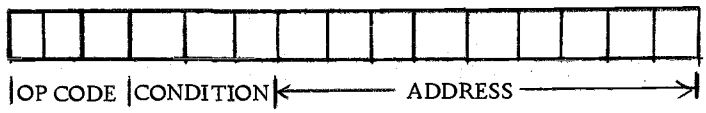


Figure 3.3. Group (2) format.

In this the first digit W_1 determines the kind of operation to be performed and W_2 determines the condition that needs to be satisfied in order for the operation to take place and $W_3W_4W_5$ constitute the address of the ROM location, which will be used by the operation.

The various conditions of W_2 are listed in Table 3.2.3.

Table 3.2.3. Conditions indicated by conditional digits W_2.

W_2 In Octal	Condition Indicated
0	Unconditional
1	Right overflow
2	Left overflow
3	Sign positive
4	Zero
5	Index
6	Indirect
7	Sign negative

The various operations are listed in the following paragraphs.

1. JUMP

   Octal code: 4

   Operation: Jump is the address indicated by the address field when the accumulator satisfies the condition indicated by the conditional bit.

2. SUBJUMP

   Octal code: 5

   Operation: SUBROUTINE JUMP, stores the present address of the operation and jumps to the address indicated by address field.

3. RETURN

   Octal code: 6

   Operation: Return to the address indicated by the address field.

4. JUMPT

Octal code: 7

Operation: Jump to the address indicated by the address field when the temporary register satisfies the condition indicated by the conditional bits.

One example of group (2) type of microinstruction is "43132" JUMP ON [ACC] > 0 to ROM LOCATION 132_8. Table 3.2.4 gives a summarized form of all the micro operations.

3.2.2. Description of Operation

The flow chart for MICRO is given in Figure 3.4. The control section consists of 15 bit ROM, a ROM buffer (ROMBR), address logic, and a ROM address register (ROMAR).

In the initializing process the size of the ROM and the contents of ROM, the decoder values, (more detailed version of the decoder and its importance will be discussed in the following paragraphs), the size and number of general purpose registers, and the sizes of instruction register, accumulator, main memory buffer, main memory address register, temporary data register, temporary address register, stack pointer of the ROM and program counter of ROM will be read in. Also, the size of main memory and the contents of main memory will be read in. Then the parameter for TIME, RUN,

Table 3.2.4. Summary of microinstructions.

Octal Code	Type	Description	Operation Called
W_1W_2			
00	1	NO OPERATION	EXEC (1)
01	1	SUB - [-T+ACC→ACC]	EXEC (2)
02	1	SUB - [T-ACC→ACC]	EXEC (3)
03	1	ADD - [T+ACC→ACC]	EXEC (4)
04	1	2'COMP [$\overline{\overline{ACC}}$→ACC]	EXEC (5)
05	1	INCA [ACC+1→ACC]	EXEC (6)
06	1	INCT [T+1→T]	EXEC (7)
07	1	INCPC [PC+1→PC]	EXEC (8)
11	1	AND [T.AND.ACC→ACC]	EXEC (9)
12	1	OR [T.OR.ACC→ACC]	EXEC (10)
13	1	XOR [T.EOR.ACC→ACC]	EXEC (11)
14	1	COMP [$\overline{ACC}$→ACC]	EXEC (12)
15	1	RS [RIGHT SHIFT ACC BY ONE BIT]	EXEC (13)
16	1	LS [LEFT SHIFT ACC ONE BIT]	EXEC (14)
21	1	READ	EXEC (18)
22	1	WRITE	EXEC (19)
31	1	CLA [0→ACC]	EXEC (26)
32	1	$\overline{SB}$→SB [SIGN OF ACC IS COMPLEMENTED]	EXEC (27)
33	1	0→ RO	EXEC (28)
34	1	0→LO	EXEC (29)
W_1			
4	2	JUMP	NONE
5	2	SUBJUMP	NONE
6	2	RETURN	NONE
7	2	TJUMP	NONE

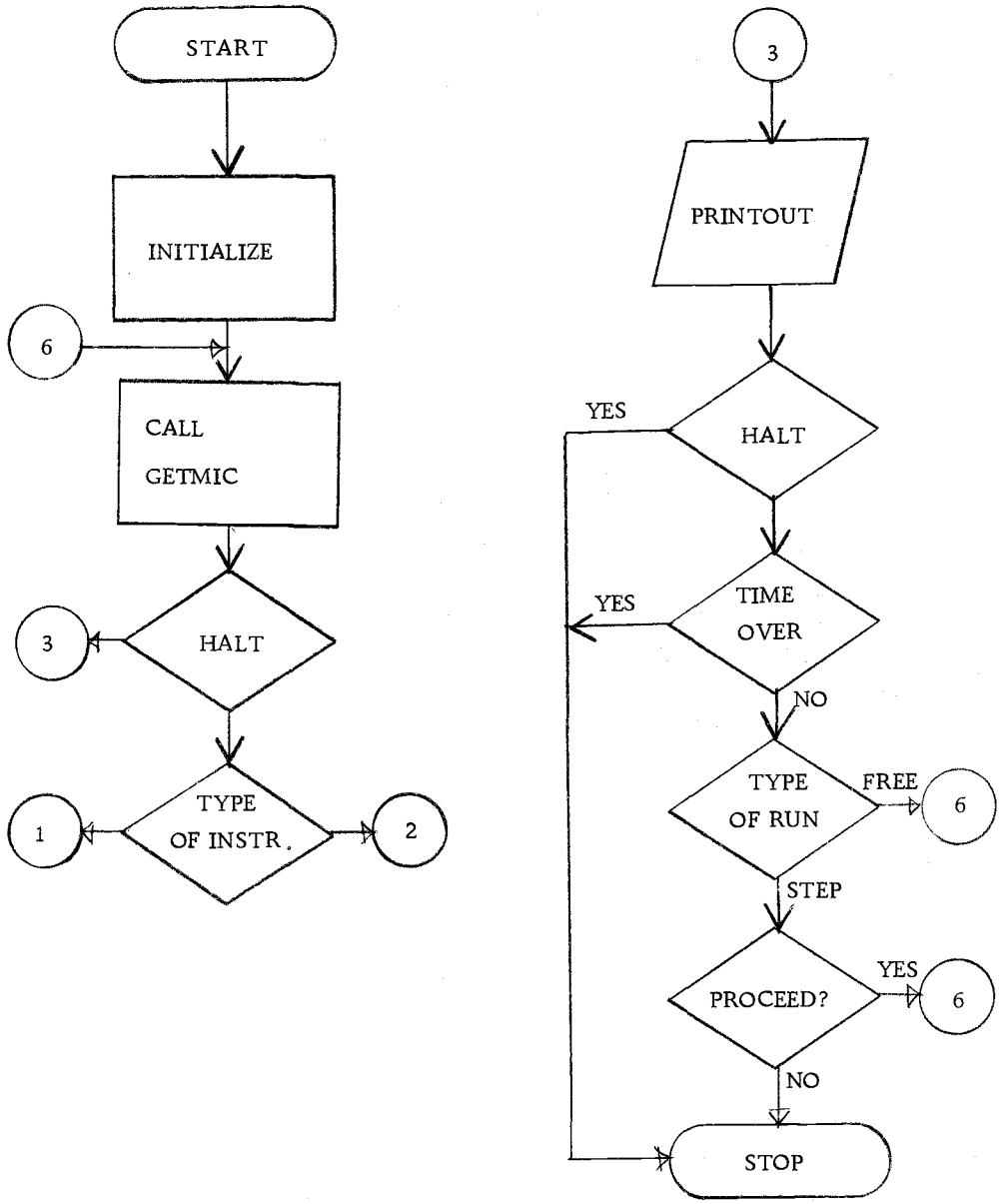


Figure 3.4. Flow chart for MICRO.

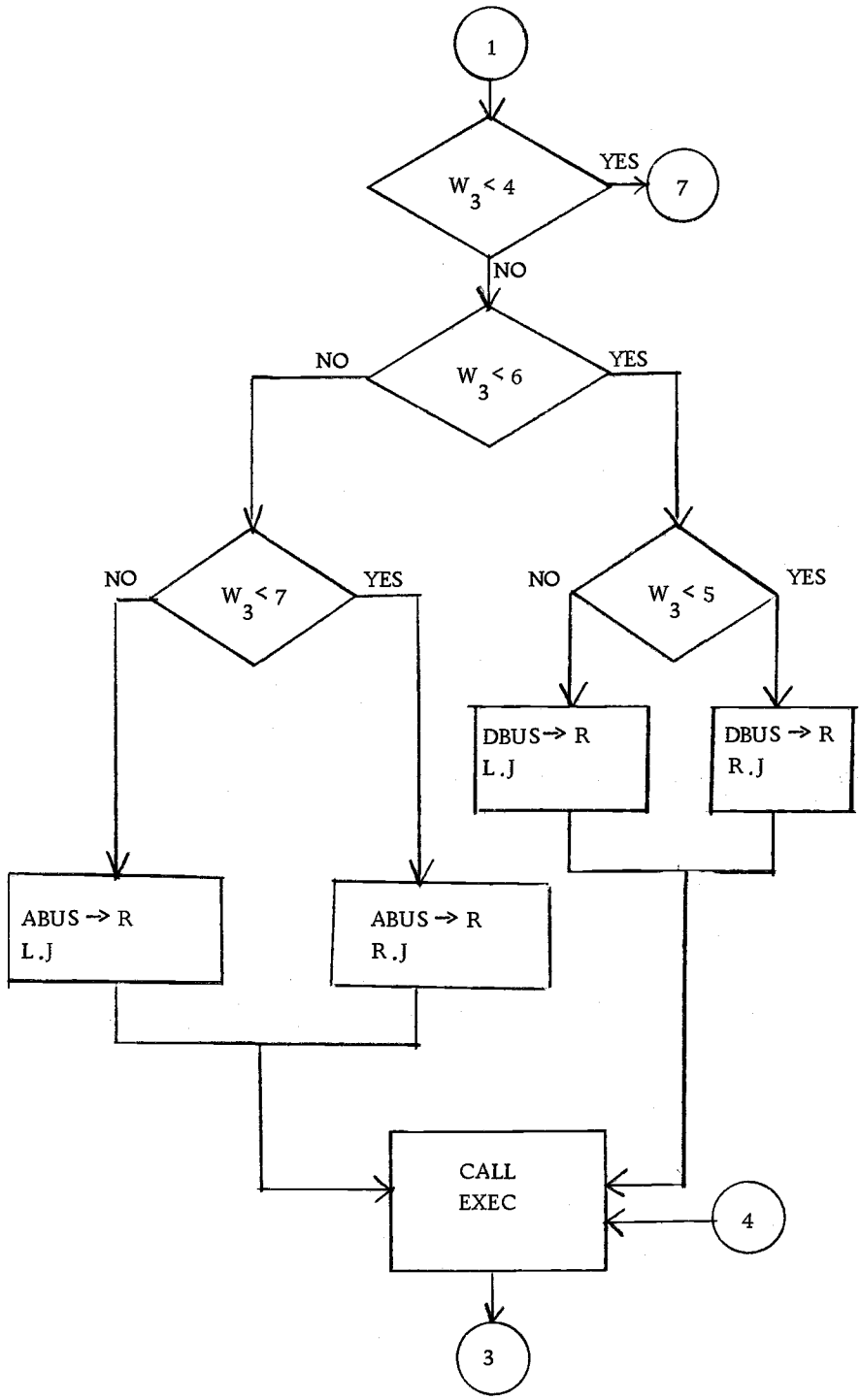


Figure 3.4. Continued.

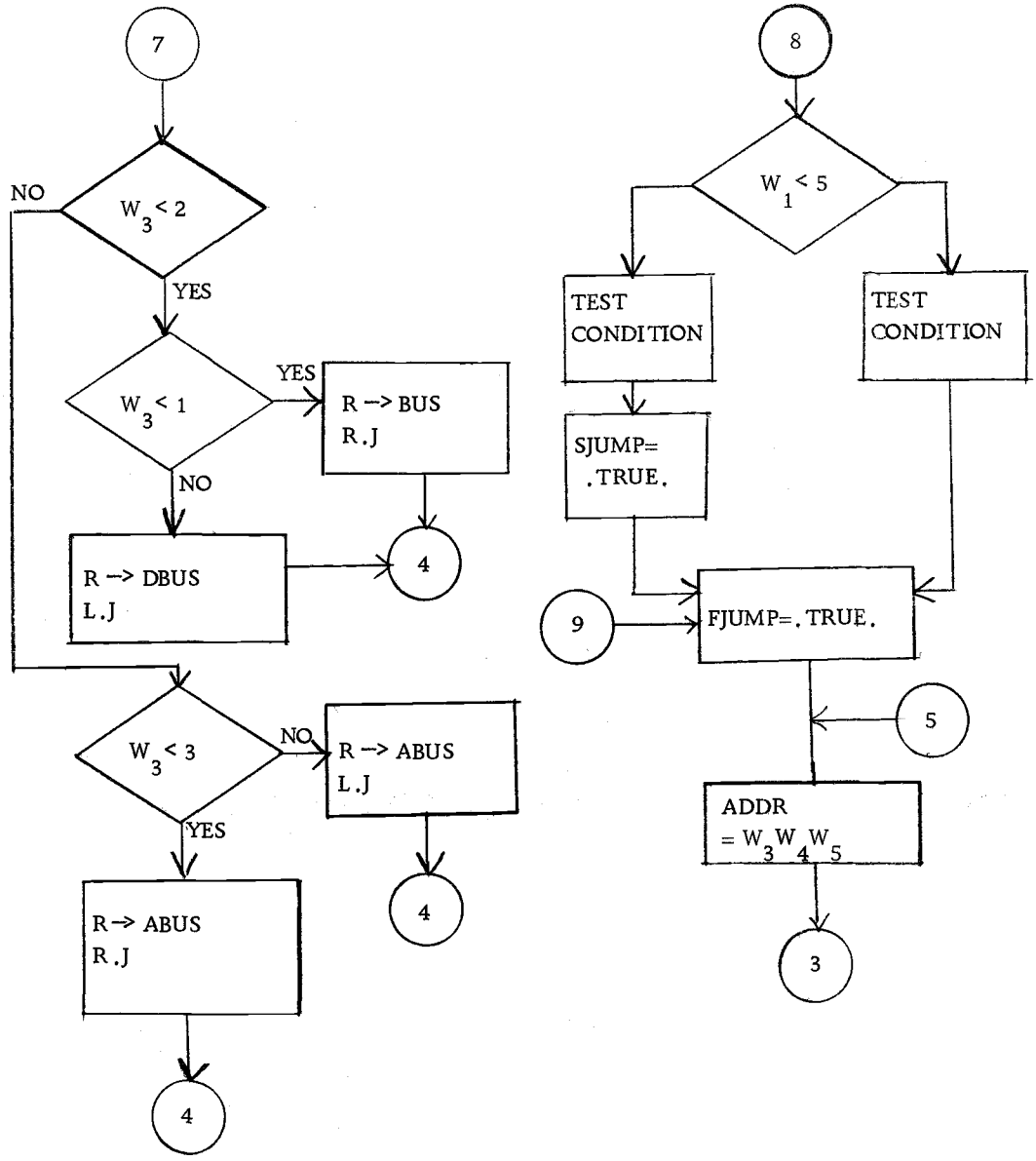


Figure 3.4. Continued.

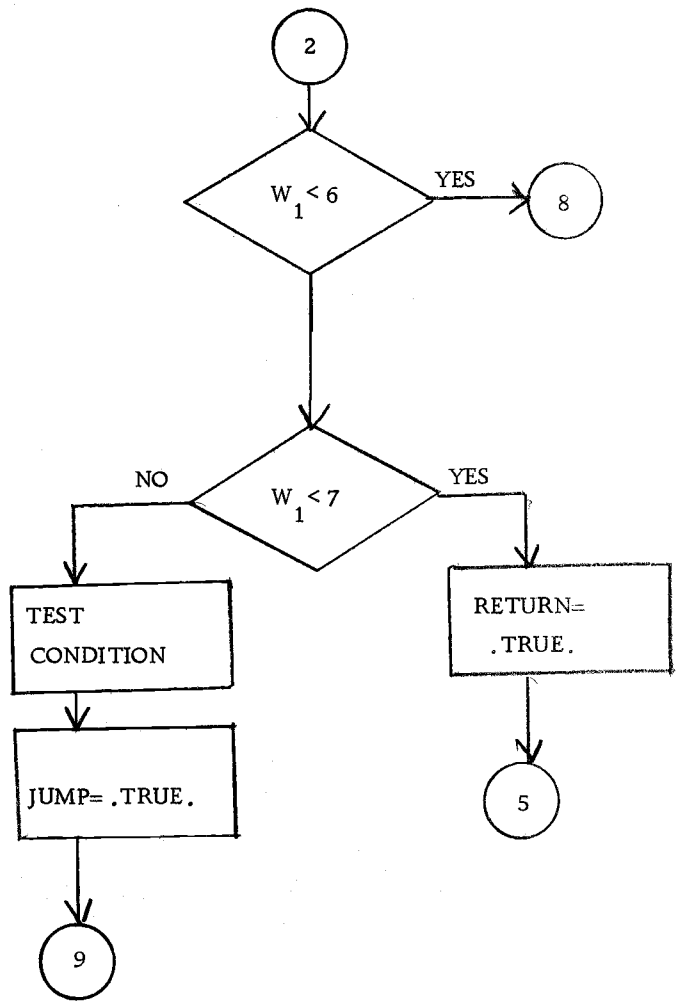


Figure 3.4. Continued.

PRINT will be read in and corresponding flags are set. Then SUBROUTINE GETMIC will be called to fetch the instruction.

The flow chart for GETMIC is given in Figure 3.5. In this subroutine, the IFETCH flag is checked to see if the instruction register already has the main memory instruction loaded in it, in which case the subroutine DECODER will be called to calculate the address of ROM instruction and will be placed in ROMAR. If IFETCH is not set, then the fact whether it might be one of the group (2) instructions is verified. In case of JUMP instruction, the address of the microinstruction to where the jump operation need to take place will be placed in ROMAR. In case of SUBJUMP instruction, the contents of the program counter will be saved and the address of the ROM location to where the SUBJUMP need to take place will be loaded into ROMAR. In case of RETURN instruction, the contents of STACK which are previously saved by SUBJUMP will be loaded into the ROMAR.

Once ROMAR contains the address of the ROM instructions to be performed next, the instruction will be read into ROMBR and it will be decoded into five octal digits. The control returns back to MICRO with these five octal digits.

Hence, once an instruction from ROM is fetched in, the left most two digits are used to determine whether the instruction is HALT or GROUP (1) or GROUP (2). In case of HALT the program prints out

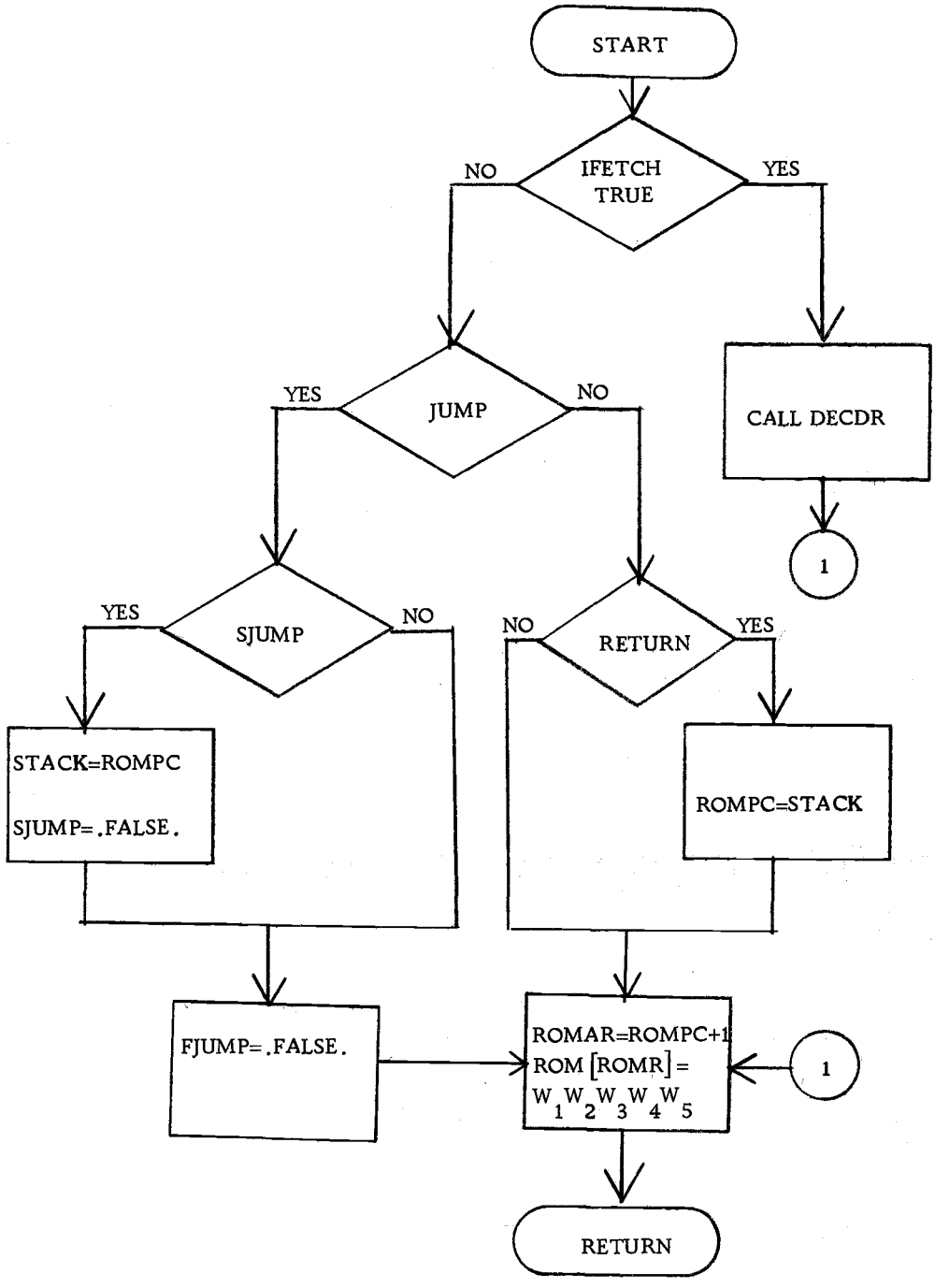
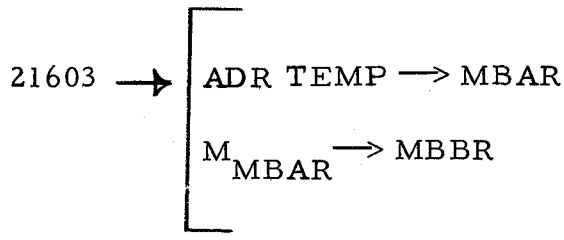


Figure 3.5. Flow chart for GETMIC.

the results depending upon the type of printout desired by the user (more details about this will be discussed later) and then stops. In group (1) microinstruction, the left most two digits indicate the type of operation to be performed and the third digit indicates the right or left justification, the type of temporary register to be used and direction of flow of information. The right most two digits constitute the address of the register to be used to perform the operation. In this type of microinstruction set, all operations are performed through temporary registers. For example, in this particular micro-instruction "21603" information is first transferred from temporary address register to memory address register and the transfer is right justified, then the contents of the address stored in memory address register are loaded in to the main memory buffer by the execute sub-routine.

In summary



All the operations in Group (1) microinstructions are performed by the EXECUTE SUBROUTINE.

In this subroutine the first two digits are used to calculate the operation number and it will be executed and the control returns back to micro. After performing group (1) operation, the results are

printed according to the wish of the user and the next microinstruction will be fetched in by the subroutine GETMIC if it is a FREE run. If it is a STEP run, the control waits for the command by the user to proceed further.

In case of Group (2) microinstructions, the first digit from the left denotes the operation to be performed and the second from the left denotes the condition to be satisfied in order for the operation to take place and the right most three digits constitute the address that will be used in performing the operation. Once group (2) operation is performed, the results are printed according to the instructions of the user and the control returns back to GETMIC subroutine if the instruction is a FREE RUN or waits for the proceed command if it is otherwise.

3.2.3. Decoder

Unlike the other kinds of microinstruction simulators, this MICRO does not assume the instruction formation of the emulated machine. The user is at liberty to use any kind of format he wishes to. In the initializing routine which takes place in the beginning of the program, the values which denote the number of bits from left, that constitute the operation code in the instruction format DN, the offset parameters D1 and D2, with which the programmer wishes to locate his microinstruction in ROM, the index, indirect and page bits and

the size of the address field IAD will be read in.

The flow chart for DECODER SUBROUTINE is given in Figure 3.6. The function of this subroutine is to calculate the address of the microinstruction in ROM from the main memory instruction which is in the INSTRUCTION register IR, using the parameters DN, D1, D2 and with the operation code of the instruction in IR. The address field of the IR is used along with index, indirect and page bits to calculate the address in the main memory where the instructions store or retrieve data, and this address is stored in the temporary address register.

3.2.4. Free Run and Single Step Run

MICRO can be run either in FREE RUN or in SINGLE STEPRUN. In case of FREE RUN, the program will be executing the instructions from the main memory using the ROM microinstructions until it comes acorss one of the following three conditions.

1. When it finds a HALT in the ROM's microinstructions which is indicated by the left most two digits of the microinstruction equal to 77.
2. When the clock, which is set in the beginning of the program to certain value, exceeds that value.
3. When it finds a HALT instruction in the main memory and executes that using ROM microinstructions, in which case

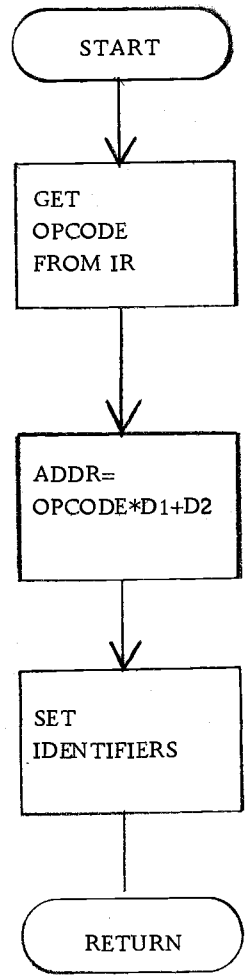


Figure 3.6. Flow chart for DECODER.

the microinstruction HALT will be met.

In case of SINGLE STEP RUN, the program will execute every microinstruction from ROM and will wait for the proceed command from the user, which is done by typing 'P'. If it does not get the proceed "P" command, the program stops execution. The other times the program comes to a stop is when the clock time limit expires or when it sees a HALT in ROM microinstructions. SINGLE STE RUN is very useful, when the user wants to examine the actions performed by each microinstruction on line before he does nest one, which is generally done while debugging. The FREERUN is useful when he is not running on line and when he is running his data and the program in the input form of card deck or when he wants the program to run for certain clock pulses.

To set the run of the program for either FREE or SINGLE STEP the following procedure is followed. In the initializing routine after reading in ROM size and contents, decoder values, size and number of general purpose registers, sizes of other main registers and size and contents of main memory, the following parameters will be requested by the program.

TX (sets clock value) — The number of clock pulses, the program needs to run. The default value of the clock is 50.

F1 (sets the run flag) — By setting this equal to zero, the program will have FREE run. Any thing more than zero the program will have SINGLE STEP run. If no value given, the system will have FREE RUN.

R1 (sets the print flag) — By setting this equal to zero, the values of the registers will be printed for every ROM's micro instruction. By setting this to any other value than zero, the values of the registers will be printed for every main memory's instruction. If no value is given the print out will be for every ROM microinstruction.

The clock is updated after every ROM's instruction.

If each ROM's instruction execution is considered as one minor cycle and each main memory's instruction execution is considered as one major cycle, then the system's clock is updated for every minor cycle. Every major cycle consists of two parts, EXECUTION and FETCH. FETCH consists of four minor cycles and EXECUTION consists of more than one minor cycle. So for every machine's instruction, the system goes through a maximum of one major and minimum of six minor cycles. If the clock is set to 40, the program goes through 40 minor cycles and if PRINT FLAG is set TRUE by setting R1=0, then the print out will be for every minor cycle otherwise it will be for every major cycle. Minor cycles consist of two pulses and they are shown below for Group (1).

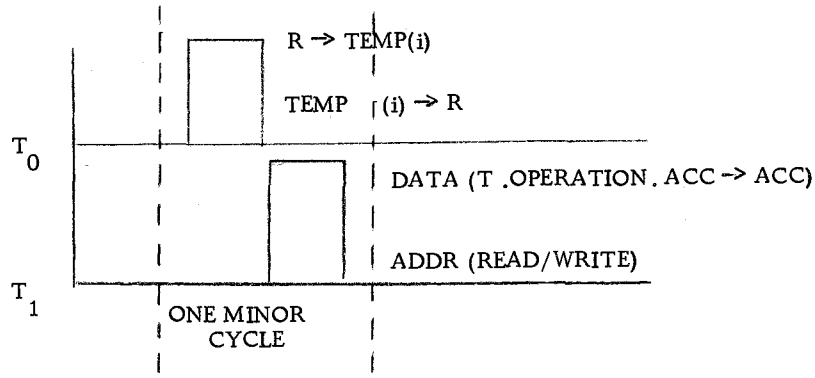


Figure 3.7. Time pulses for group (1) microinstructions.

Every minor cycle consists of three pulses for group (2). They are shown below in Figure 3.8.

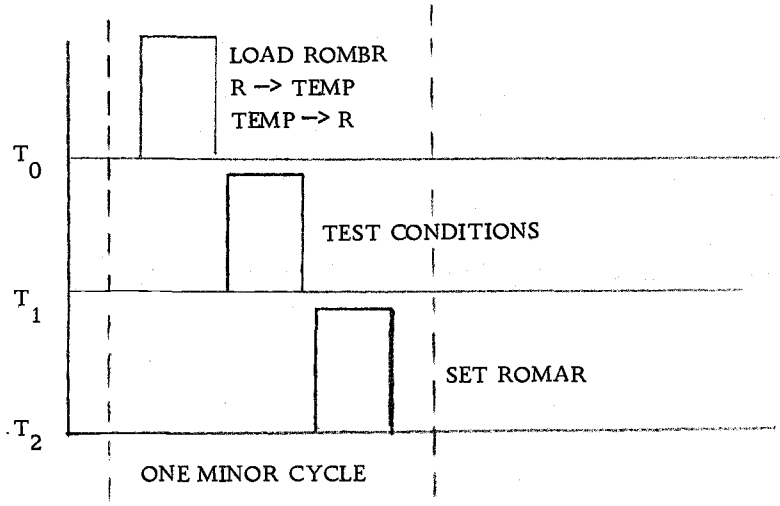


Figure 3.8. Time pulses for group (2) microinstructions.

3.2.5. Trace Feature

The MICRO is provided with a trace feature which allows for easy troubleshooting of the program. This feature produces output of considerable amount and should be used only if necessary and should be used with the knowledge of what that printout means. This prints

series of information as it processes every ROM instruction regarding the HALT test, the register number that will be used for the transfer of data, the kind of justification, the type of execution to be performed etc. A copy of the trace printout is given in Appendix B.

3.2.6. How to Use MICRO

MICRO can be used online or offline.

3.2.6.1. Online Operation. Every user will be provided with a user number, password. Using these he can log on the KRONOS system, the system will respond by the message "READY".

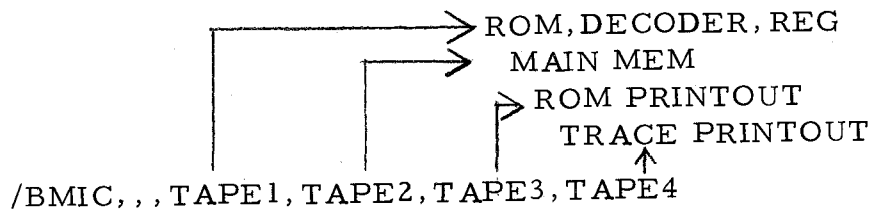
He can use the EDIT and edit his two input files, one consisting of all the information regarding ROM, all registers and decoder values and the other containing of main memory information. Let us say the first file is on TAPE1, and second file is on TAPE2. Using the GET or OLD command he had to call the BMIC (Binary coded MICRO) and then the following commands will be given.

/BATCH,50000

The system responds with

/RFL$50000

The user types



Here the output will be the terminal itself.

Then the system prints out the main memory contents in both octal and decimal and comes back with a question mark for the parameters TIME, TYPE OF RUN, TYPE OF PRINTOUT.

The user types

? 40,8,0

The time is a four integer variable so it needs to have four characters and hence the blanks are included, the type of run is STEP run and the printout is for every minor cycle. So the program will wait for letter 'P' after every ROM execution and printout. If a carriage return is given instead of 'P' the program terminates. Here the clock value is 40.

3.2.6.2. Offline Operation. The JOB DECK set up will be the following.

JOBIDEN,CM55000,T100.

ACCONT,[USER NUMBER],[USER PASSWORD].

GET,BMIC.

BMIC,,TAPE5,,,,,.

7
8
9

[TAPE1]

[TAPE2]

[TX,F1,R1]

$6_7 8_9$

3.3. Example--Emulation of PDP-8

The level of microprogramming used in this MICRO is the same as that of the "user microprogrammable" minicomputers and micro processors. Actually the only distinction between this level microprograms and machine language lies in the fact that the instructions reside in a separate control memory (ROM) and can not be loaded with the data as one usually does with machine instruction.

As an example, a partial emulation of DEC-PDP-8 is shown. A general block diagram of simplified PDP-8 is given in Figure 3.9.

The PDP has a 3 bit op code field. Six of the 8 possible op codes correspond to the memory reference which store or retrieve data from the core memory such as: Logical AND (AND), two's complement Addition (TAD), Increment the Memory Location and Skip on Zero (ISZ), Deposit and Clear Accumulator (DCA), Subroutine Jump (JMS) and Unconditional Jump (JMP). The remaining two op codes correspond to an I/O instruction and two "microinstruction".

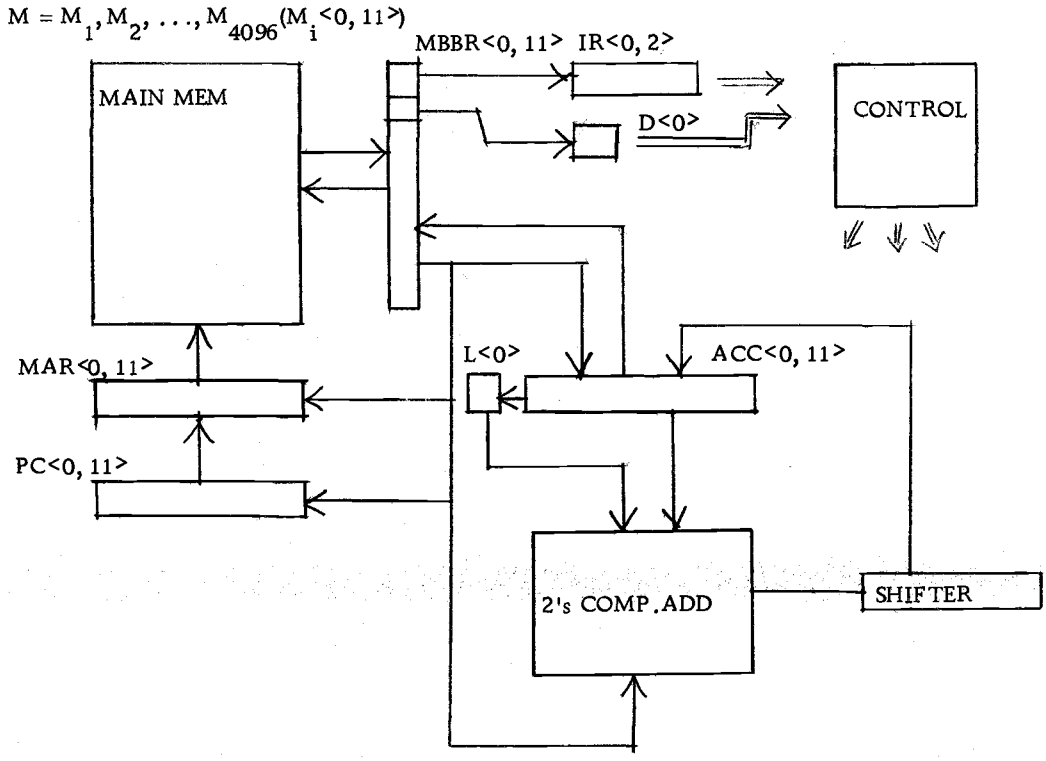
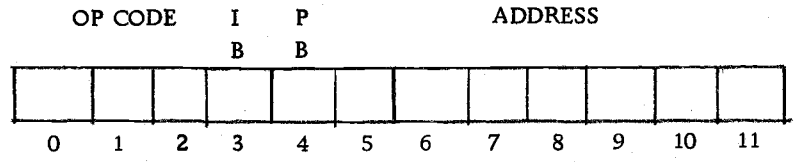


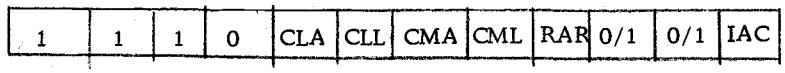
Figure 3.9. DEC-PDP-8 systems block diagram.

The PDP-8 basic processor is a simple address, fixed word length, parallel transfer computer using 12-bit, 2's complement arithmetic. This consists of 3 major cycles: Fetch, Defer and Execute. Each major cycle is divided into 3 minor cycles. The defer cycle handles one level of indirect addressing. The "microinstructions" consist of instructions which can be executed in one minor cycle (e.g., complement accumulator, etc.) and hence can be overlapped within one execution major cycle.

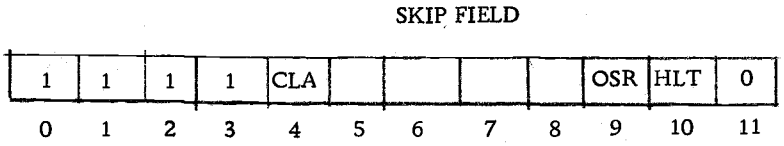
The PDP-8 in Figure 3.9 has 4096 words of memory. The instruction format is shown in Figure 3.10.



(a) Memory reference



(b) Type (1) MICRO instruction



(c) Type (2) MICRO instruction

Figure 3.10. PDP-8 instruction format.

As an example we will microprogram a simulator for the PDP-8. Only a partial simulation will be shown as an example. In this case we have to set up the DECODER parameters as

DN=3 The number of bits that represent the op code.

D1 =5
D2 =5 The offset parameters

IAD=9 The number of bits that represent the instruction register.

The index, indirect and page bits are set to zero values in order to simplify the simulation process but otherwise they can be used too. That operations that we emulate are listed in Table 3.3.1. The octal ROM locations for example for TAD is = 1 x D1 + D2 = 10. At 10^{th} ROM location the execution of TAD will start.

Table 3.3.1. Operations emulated.

IR	Operation
000	AND
001	TAD
002	ISZ
003	DCA
004	JMS
005	JMP
006	μ-INS
007	HALT

The sizes of the necessary registers such as ACC etc are set to 12 bit each and they are denoted as follows in Table 3.3.2.

Table 3.3.2. Notation of registers.

Register Number	Type of Register	Size	Name
00	Instruction	12	IR
01	Accumulator	12	ACC
02	Main memory buffer	12	MBBR
03	Main memory address	12	MBAR
04	Data temp	12	DBUS
05	Address temp	12	ABUS
06	Stack pointer	12	SP
07	Program counter	12	PC

Table 3.3.3 shows different operations of PDP-8 that are emulated using MICRO and the contents of various ROM locations that process the operations. Only partial emulation is shown here. The details omitted include the setting of page, index, and indirect bits. At present they were set to zero.

Only the memory reference and housekeeping instructions were shown, the rest of augumented instructions are omitted for the present example. The input/output instructions that allow the program to converse with peripherals, i.e., external communication, is omitted.

Using the ROM shown in Table 3.3.3, a small addition of two signed numbers is done and the output with trace feature is attached in Appendix B.

Table 3.3.3. ROM for [DPD-8].

ROM Location	ROM in Octal	Register Transfer of Emulation	Operation of PDP-8
1	0 7 2 0 7	PC $\longrightarrow$ ABUS, PC+1 $\longrightarrow$ PC	FETCH
2	2 1 6 0 3	ABUS $\longrightarrow$ MBAR $M_{MBAR} \longrightarrow$ MBBR	
3	0 0 0 0 2	MBBR $\longrightarrow$ DBUS	
4	0 0 4 0 0	DBUS $\longrightarrow$ IR	
5	2 1 6 0 3	ABUS $\longrightarrow$ MBAR $M_{MBAR} \longrightarrow$ MBBR	AND
6	1 1 0 0 2	MBBR $\longrightarrow$ DBUS DBUS.AND.ACC $\longrightarrow$ ACC	
7	4 0 0 0 0	JUMP TO FETCH	
10	2 1 6 0 3	ABUS $\longrightarrow$ MBAR $M_{MBAR} \longrightarrow$ MBBR	TAD
11	0 3 0 0 2	MBBR $\longrightarrow$ DBUS DBUS+ACC $\longrightarrow$ ACC	
12	4 0 0 0 0	JUMP TO FETCH	
13	0 0 0 0 0		
14	0 0 0 0 0		
15	2 1 6 0 3	ABUS $\longrightarrow$ MBAR $M_{MBAR} \longrightarrow$ MBBR	ISZ
16	0 6 0 0 2	MBBR $\longrightarrow$ DBUS, DBUS+1 $\longrightarrow$ DBUS	
17	2 2 4 0 2	DBUS $\longrightarrow$ MBBR MBBR $\longrightarrow M_{MBAR}$	
18	7 4 0 4 3	IF DBUS=0, JUMP TO 35 location in ROM	
19	4 0 0 0 0	JUMP TO FETCH	

Table 3.3.3. Continued.

ROM Location	ROM in Octal	Register Transfer of Emulation	Operation of PDP-8
20	3 1 0 0 1	ACC ⟶ DBUS, 0 ⟶ ACC	DCA
21	0 0 4 0 2	DBUS ⟶ MBBR	
22	2 2 6 0 3	ABUS ⟶ MBAR MBBR ⟶ $M_{[MBAR]}$	
23	4 0 0 0 0	JUMP TO FETCH	
24	0 0 0 0 0		
25	0 0 0 0 7	PC ⟶ DBUS	JMS
26	0 0 4 0 2	DBUS ⟶ MBBR	
27	2 2 6 0 3	ABUS ⟶ MBAR MBBR ⟶ $M_{[MBAR]}$	
28	0 6 0 0 5	ABUS ⟶ DBUS DBUS+1 ⟶ DBUS	
29	4 0 0 4 5	JUMP TO 45_8=37th location in ROM	
30	0 0 0 0 5	ABUS ⟶ DBUS	JMP
31	0 0 4 0 7	DBUS ⟶ PC	
32	4 0 0 0 0	JUMP TO FETCH	
33	0 0 0 0 0		
34	0 0 0 0 0		
35	0 7 0 0 0	PC+1 ⟶ PC	Rest of the operation of ISZ
36	4 0 0 0 0	JUMP TO FETCH	
37	0 0 4 0 7	DBUS ⟶ PC	Rest of the operation of JMS
38	4 0 0 0 0	JUMP TO FETCH	
39	0 0 0 0 0		
40	7 7 0 0 0	HALT	HALT

3.4. Conclusion

Using MICRO any general purpose computer can be simulated. Also any microprogram can be simulated. The user can easily study the performance of a number of systems. As he becomes more familiar with better usage of this, he can change his system with no more trouble than editing his data file.

If MICRO were to be rewritten the following changes are suggested.

1. At present the interactive feature of MICRO is poor. There has to be some statements included in the printout to make the user aware of what is expected to be entered in, if the user is ONLINE. For example, in the very beginning of the program, the data needs to be entered is "ROM SIZE", which should be printed by the terminal. But right now this is not included and it will be nice if it is added. In essence, more information needs to be included in the output.
2. External communication--MICRO does not contain any means to specify which peripheral the program wants. It will be better if there is any means along with the I/O interupt incorporated along with it.
3. MICRO is currently capable of only single address instruction with modifiers.

4. MICRO's output prints every register in a fixed format, which is a little awkward. It will show better if the leading zeros are suppressed.

IV. CONCLUSION

The REGTRAN language is fairly easy to learn for anyone who is familiar with FORTRAN or a similar high level language. The interactive capability of SYSSIM provides an efficient way for the digital system designer to debug his design before working out the detailed logic design. These two programs can be used in teaching a course in digital system design. A student of logic design, can debug his network with little effort by simulating it using these two programs. To learn design techniques, digital systems ranging in complexity, may be described and simulated. As the student becomes more familiar with better techniques, he can change his system with no more trouble than editing his data file.

The present version limits the number of bits per word to 58, the numbered memory words to 8192 per memory, and the total number of registers and terminal names to 500. Conversion to run on a computer other than the CDC 6600, requires some changes in the production tables symbols and the ASCII comparison of the symbols in the REGTRAN and SYSSIM programs. Also other several changes have to be made because the programs make extensive use of the 60-bit word length. Transfer to a computer with a shorter word length would reduce the permissible length of registers, terminals and memory words Alphanumeric characters are also packed ten per

word, so conversion to another computer would involve a change in many FORMAT statements. Many out put routines require the use of variable FORMATS, so another method would be needed, such as multiple FORMAT statements, if variable FORMAT was not available.

If the programs were to be rewritten, several changes would probably be made, which were given at the end of Chapter II. REGTRAN language is easy to learn but the generation of FORTRAN by REGTRAN is such a complex process, that it will not allow not even a single mistake in the creation of the source file in REGTRAN language. A single error in the source file can completely wipe off the FORTRAN thus generated, which causes termination of the whole process of execution. So in creating the source file the user has to be very observant, not to make a minute error. Also, they have to frame their problem and describe in a way so that mixed operation logical and nonlogical statements will not be generated. As described in Chapter II, this problem happens due to the change of compiler from RUN compiler to FTN compiler. REGTRAN and SYSSIM are functional at present but can be made more efficient, if more flow charting and explanation of some routines were available.

MICRO--a simulator for a simple microprogrammable computer is very useful as REGTRAN and SYSSIM can not simulate any micro-instruction, thus this is achieved by MICRO. The microinstructions, which are simulated by MICRO are similar to machine language

instructions. This level microprogramming is typical of that used in "user microprogrammable" minicomputer and microprocessors. Actually the only distinction between this level microprogram and machine language lies in the fact that the instructions reside in a separate control memory (ROM) and can not be loaded with the data as one usually does with machine instructions. To demonstrate the features of MICRO, a partial simulation of DEC PDP-8 is shown as an example. Unlike some of the other simulators, that existed so far for the simulation of microprogrammable computers, the MICRO will not assume the FORMAT specification of the machine, which it will be simulating. This provides the user with a flexibility of assigning the FORMAT specification fitting to his own problem. The size of ROM, main memory, all the eight system registers, the general purpose registers are all left to have flexible dimension for the same reason. Other options in creating the main memory include features such that an input format of one card can control the rest of the main memory. A star '*' in the input field will clear the rest of the memory locations.

The method to use the MICRO is very easy and is flexible in its application. Adding a new register, arithmetic or a test condition etc. can be done without any difficulty and can be done by adding one more routine to EXEC subroutine and very little changes in the main program are needed.

A tradeoff exists between the various levels of microprogramming. The higher the level, the easier it is for the microprogrammer. The decode and timing logic becomes more complex as the language used moves from direct to encoded to high level.

If MICRO were to be rewritten then the changes that would be made include (1) better interactive feature which allows the user to change his files, whenever he wises. (2) A better print out which gives more information than at present. (3) The input/output interrupt feature incorporated with user specified devices.

In conclusion MICRO can simulate successfully the specified machine but it can be made more efficient.

BIBLIOGRAPHY

1. I. S. Reed, Symbolic Design Techniques Applied to Generalized Computer, M.I.T. Lincoln Lab., Lexington, Mass., TR No. 141, Jan. 3, 1957.

2. I. S. Reed, T.C. Bartee and I.L. Lebow, Theory and Design of Digital Systems, The McGraw-Hill Book Co., Inc. N.Y., N.Y. 1962.

3. I. S. Reed, Symbolic Synthesis of Digital Computers, Proc. ACM, Sept. 1952. pp. 90-94.

4. H. Schorr, Gomputer Aided Digital System Design and Analysis Using a Register Transfer Language, IEEE Transactions on Electronic Computers, Vol. EC-13, Dec. 1964, pp. 730-737.

5. K.E. Iverson, A Programming Language, Proc. of 1962 SJCC.

6. K.E. Iverson, A Common Language for Hardware, Software and Applications, Proc. of 1962 FJCC.

7. K.E. Iverson, Programming Notations in System Design, IBM Systems Journal, June 1963.

8. F.J. Hill and G.R. Peterson, Digital Systems: Hardware Organization and Design, Wiley, N.Y. 1973.

9. Y. Chu, Computer Organization and Microprogramming, Prentice-Hall, Englewood-Cliffs, N.Y. 1972.

10. Y. Chu, Design Automation by the Computer Design Language, Technical Report 69-86, Computer Science Center, University of Maryland, March 1968.

11. Y. Chu, A Higher-Order Language for Describing Micropro-grammed Computers, Technical report 68-78, Computer Science Center, University of Maryland, Sept. 1968.

12. Y. Chu, Structure of CDL Programs, Technical Note 74-58, Department of Computer Science, University of Maryland, May, 1974.

13. Y. Chu, An ALGOL Like Computer Design Language, Communications of ACM, Oct. 1965, pp. 607-615.

14. J Lund, LOGAL-Logic Algorithmic Language, Univac Tech. Memo A00317, March 5, 1973, Reseville, Minnesota.

15. H.P. Schleppi, A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS), IEEE Trans. on Electronic Computers. Vol. Ec-13, Aug. 64. pp. 439-448.

16. D.L. Parnas, System Function Description ALGOL, Ph.D. Thesis, Carnegie-Mellon, Univ. Pittsburgh, Pennsylvania, Feb. 1965, Dept. of Elec. Eng.

17. J.A. Darringer, The Description, Simulation and Automatic Implementation of Digital Computer Processors, Ph.D. Thesis, Carnegie-Mellon Univ., May 1969.

18. J.A. Darringer, A Language for the Description of Digital Computer Processors. Proc. of the Design, Automation Workshop, 1968. pp. 15-1 to 15-8.

19. J.A. Darringer and D.L. Parnas, More on Simulation Languages and Design Methodology for Computer Systems. Proc. Spring Joint Computer Conference, 1969.

20. J.A. Darringer and D.L. Parnas, SODAS and a Methodology of Systems Design, Proc. FJCC, 1967.

21. J.A. Wilber, A Language for Describing Digital Computers. MS Thesis, Dept. of CS, Univ. of Illinois, Urbana, Feb. 1966.

22. A Giese, HARGOL - A Hardware Oriented Algol Language, Copenhagen, Denmark, A/S Regenecentralen, Feb. 1969.

23. C.G. Bell and Newell, Computer Structures: Readings and Examples, McGraw-Hill, 1971.

24. C.G. Bell and Newell, Register Transfer Design: Computers and Digital Systems Using the PDP-16, Digital Press, 1972.

25. M.B. Barbacci and D.P. Siewiorek, Automated Exploration of the Design Space for Register Transfer Systems. Proc. of the 1st Annual Symposium on Computer Architecture, Gainesville, Florida, December 1973.

26. C.G. Bell and A. Newell, The PMS and ISP Descriptive Systems for Computer Structures. Proc. of 1970 SJCC.

27. M.J. Knudsen, PML - An Interactive Language for System-Level Description and Analysis of Computer Structures. Ph.D. Thesis. Dept. of CS. Carnegie-Mellon Univ.

28. M.B. Bany and S.Y.H. Su, A Digital System Modeling and Design Language. Proc. of the 8th Annual Design Automation Workshop, 1971.

29. S.Y.H. Su, A Language for Automated Logic and System Design. Workshop on Computer Descriptive Language. Rutgers Univ., New Brunswick, N.J. Sept. 6-7, 1973.

30. D.L. Dietmayer and R.L. Arndt, DDLSIM - A Digital Design Language Simulator. Proc. of NEC, Vol. 26, Dec. 1970. pp. 116-118.

31. J.R. Duley, DDL - A Digital System Design Language, Ph.D. dissertation, University of Wisconsin, Madison, 1967.

32. J.R. Duley and D.L. Dietmeyer, A Digital System Design Language, IEEE Trans. on Computers. Vol. C-17, Sept. 1968, pp. 850-861.

33. C. E. Peet, Jr., A Register Transfer Simulator for Digital System Design. M.S. Thesis at the University of Texas at Austin, 1973.

APPENDICES

APPENDIX A

```
* PARALLEL ADDER-SUBTRACTER
      <TE> X(12),Y(12),C(13),S(12),A(12),K(12),OVERFLO
 *****<SY> MISSING
 *****SYSTEM NAME MISSING
      <BO> Y= -A*K + A*(-K),
      C(12) = K(11),
        C(0:11) =X.A.Y .O.X.A.C(1:12) .O.Y.A.C(1:12),
S = X.E.Y.E.C(1:12),
      OVERFLO = X(0)*Y(0)*-S(0) + -X(0)*-Y(0)*S(0)
<EN>

TIME TO COMPILE REGTRAN PROGRAM =  1.160 SECONDS
```

```
      SUBROUTINE CONTROL
      WRITE(7,1)
    1 FORMAT(5X,47H*****SYSTEM <UNKNO CONTAINED A FATAL ERROR*****)
      RETURN
      END
```

```
* PARALLEL ADDER-SUBTRACTER
<SY> PAS
      <TE> X(12),Y(12),C(13),S(12),A(12),K(12),OVERFLO
      <BO> Y= -A*K + A*(-K),
      C(12) = K(11),
        C(0:11) =X.A.Y .O.X.A.C(1:12) .O.Y.A.C(1:12),
S = X.E.Y.E.C(1:12),
      OVERFLO = X(0)*Y(0)*-S(0) + -X(0)*-Y(0)*S(0)
<EN>

TIME TO COMPILE REGTRAN PROGRAM =  1.184 SECONDS
```

```
                    SYSTEM SIMULATION PROGRAM


≥ONLINE
≥1:PRINT(TSTEP(0,0)) DEC(S),OVERFLO
  *****LAST NUMBER IN TSTEP IS ZERO
≥1:PRINT(TSTEP(0,1)) DEC(S),OVERFLO
≥2:READ(TSTEP(0,1))X,A,K
≥3:RUN(TIME.EQ.4)
READ  X       A       K
 17,28,0
 S       =   45  OVERFLO=0
READ  X       A       K
 1,3,4095
 S       =4094  OVERFLO=0
READ  X       A       K
 2048,1,4095
  *****NUMBER OF LOOPS EXCEEDS.......   10
 S       =2047  OVERFLO=1
≥LOOP 50
≥RUN
READ  X       A       K
 2048,1,4095
 S       =2047  OVERFLO=1
READ  X       A       K
 4094,4094,0
 S       =4092  OVERFLO=0
≥TRACE,REG
  *****SYNTAX ERROR
≥TRACE REG

    N NAMES(N)        CONTENTS
    1 TIME      00000000000000000004
    6 X         00000000000000007775
    7 Y         00000000000000007776
    8 C         00000000000000007774
    9 S         00000000000000007774
   10 A         00000000000000007776
   11 K         00000000000000000000
   12 OVERFLO   00000000000000000000
 ≥TRACE CN
 ≥RUN(TIME.EQ.6)
 *--TYPE=1,PHRASE=TIME
 *--TYPE=4,PHRASE=.EQ.
 *--TYPE=*,PHRASE=          0
 *--TYPE=2,PHRASE=               5
 *--TYPE=3,PHRASE=)
 ...LABEL =   3
 READ  X       A       K
  TRACE OFF
 *--TYPE=1,PHRASE=TRACEOFF
  *****TRY AGAIN
  1,3,0
 *--TYPE=2,PHRASE=               1
 *--TYPE=3,PHRASE=,
 *--TYPE=2,PHRASE=               3
 *--TYPE=3,PHRASE=,
 *--TYPE=2,PHRASE=               0
 *--TYPE=3,PHRASE=
  S       =   4  OVERFLO=0
 ≥TRACE OFF
```

```
      SUBROUTINE COMBOOL
      INTEGER JKFF
      INTEGER DECODER
      INTEGER RP,RT
      LOGICAL CHNG,SSRERR,L
      COMMON/REG/NAMES( 500),NB( 500),RP( 500),RT( 500),NON,CHNG,SSRERR,
     $CCOD
      COMMON/TEMP/J(200),L(200)
      RT(    6)  =RP(    7)  .AND.DECODE(000000000001B
      IF(RP(    6)  .NE.RT(    6)  ) CHNG=.TRUE.
      RP(    6)  =RT(    6)
      RETURN
      END

      SUBROUTINE CONTROL
      LOGICAL CHNG,SSRERR,COMERR,CCOD
      INTEGER RP,RT
      COMMON/REG/NAMES( 500),NB( 500),RP( 500),RT( 500),NON,CHNG,SSRERR,
     $CCOD
      COMMON/MAIN/KARD(80),NCR,KP,COMERR,TELE,KHAR,PHRASE,TYPE,TR,MLOOPS
      RP(    2)=MOD(RP(1)-1,    1)
      RT(    2)=RP(    2)
      IF (.NOT.CCOD) GO TO 7
      CCOD=.FALSE.
      NLOOP=0
    6 CHNG=.FALSE.
      CALL COMBOOL
      NLOOP=NLOOP+1
      IF (NLOOP.GT.MLOOPS) GO TO 20
      IF (CHNG) GO TO 6
    7 CONTINUE
      DO 10 I=1,NON
   10 RP(I)=RT(I)
      ENTRY BOOLEAN
      CCOD=.FALSE.
      NLOOP=0
   15 CHNG=.FALSE.
      CALL COMBOOL
      NLOOP=NLOOP+1
      IF (NLOOP.GT.MLOOPS) GO TO 20
      IF (CHNG) GO TO 15
      RETURN
   20 CALL ERROR(36,3,MLOOPS)
      RETURN
      END

      SUBROUTINE REGSET
      LOGICAL SSRERR
      COMMON/REG/NAMES( 500),NB( 500),RP( 500),RT( 500),NON,CHNG,SSRERR,
     $CCOD
      COMMON/MEM/MEMNA(6),MEMNB(6),MEMSIZ(6),NOM
      DATA NOM/    0/
      DATA NON/    7/
      DATA NAMES(    1),NB(    1)/10HTIME      ,    21/
      DATA NAMES(    2),NB(    2)/10H          ,     1/
      DATA NAMES(    3),NB(    3)/10H.         ,    48/
      DATA NAMES(    4),NB(    4)/10H          ,     1/
      DATA NAMES(    5),NB(    5)/10H          ,    48/
      DATA NAMES(    6),NB(    6)/10HX         ,     1/
      DATA NAMES(    7),NB(    7)/10HY         ,     1/
      RETURN
      END
```

```
      FUNCTION MPRT(MEM,NWORD)
      COMMON/MEM/MEMNA(6),MEMNB(6),MEMSIZ(6),NCM
      COMMON/MAIN/KARD(80),NCR,KP,COMERR,TELE,KHAR,PHRASE,TYPE,TR,MLOOPS
      LOGICAL TELE
   10 MPRT=0
      RETURN
      END
```

```
      SUBROUTINE MSET(MEM,NWORD,MDAT)
      COMMON/MEM/MEMNA(6),MEMNB(6),MEMSIZ(6),NCM
   10 RETURN
      END
```

```
      SUBROUTINE MDATA
      RETURN
      END
```

SYSTEM SIMULATION PROGRAM

```
≥CNLINE
≥1:INPUT(TSIER(0,1)) (X=0,0,1,1,0,1,1,1,1,0,0,1,0,1)
≥2:DISPLAY X,A,B,C,Z
≥3:RUN(TIME.EQ.15)
```

```
1- X A B C Z

1- 0 0 0 0 0
1- 0 0 1 1 0
1- 1 0 1 1 0
1- 1 1 1 1 0
1- 0 1 0 0 1
1- 1 0 1 0 1
1- 1 1 1 1 0
1- 1 1 0 0 0
1- 1 1 1 0 0
1- 0 1 1 0 0
1- 0 0 1 0 0
1- 1 0 1 1 0
1- 0 1 1 1 0
1- 1 0 1 0 1
1- 1 1 1 1 0
1- 1 1 0 0 0
≥END
```

```
<SY> SEQDDT
<TE> Z1,Z2,X,NX,KB,JC,JC1,Z
<RE> A,B,C
<BO>
↓ NX=-X,
↓ Z1=-(-X*A*-B),
↓  Z2= -(X*-A*B*-C),
↓ Z=-(Z1*Z2),
↓ KB=-(-X+-A+-C),
↓ JC1=-(B+-X),
↓ JC=-(A+JC1)
<AU> ASC
<ST> A=JKFF(A,X,NX),
     B=JKFF(B,1,KB),
     C=JKFF(C,JC,A)
<EN>

TIME TO COMPILE REGTRAN PROGRAM =  1.346 SECONDS
```

```
      SUBROUTINE COMBOOL
      INTEGER JKFF
      INTEGER DECODER
      INTEGER RP,RT
      LOGICAL CHNG,SSRERR,L
      COMMON/REG/NAMES( 500),NO( 500),RP( 500),RT( 500),KON,CHNG,SSRERR,
     CCCD
      COMMON/TEMP/J(200),L(200)
      J(    1)=.NOT.RP(    8)   .AND.0000000000000000000001B
      RT(    9)   =J(      1)  .AND.0000000000000000000001B
      IF(RP(    9)   .NE.RT(    9)   ) CHNG=.TRUE.
      RP(    9)   =RT(    9)
      J(    1)=.NOT.RP(    8)   .AND.0000000000000000000001B
      J(    1)=J(      1)  .AND.      RP(   14)
      J(    2)=.NOT.RP(   15)   .AND.0000000000000000000001B
      J(    1)=J(      1)  .AND.      J(      2)
      J(    1)=.NOT.J(      1)  .AND.0000000000000000000001B
      RT(    6)   =J(      1)  .AND.0000000000000000000001B
      IF(RP(    6)   .NE.RT(    6)   ) CHNG=.TRUE.
      RP(    6)   =RT(    6)
      J(    1)=.NOT.RP(   14)   .AND.0000000000000000000001B
      J(    1)=RP(    8)   .AND.      J(      1)
      J(    1)=J(      1)  .AND.      RP(   15)
      J(    2)=.NOT.RP(   16)   .AND.0000000000000000000001B
      J(    1)=J(      1)  .AND.      J(      2)
      J(    1)=.NOT.J(      1)  .AND.0000000000000000000001B
      RT(    7)   =J(      1)  .AND.0000000000000000000001B
      IF(RP(    7)   .NE.RT(    7)   ) CHNG=.TRUE.
      RP(    7)   =RT(    7)
      J(    1)=RP(    6)   .AND.      RP(    7)
      J(    1)=.NOT.J(      1)  .AND.0000000000000000000001B
      RT(   13)   =J(      1)  .AND.0000000000000000000001B
      IF(RP(   13)   .NE.RT(   13)   ) CHNG=.TRUE.
      RP(   13)   =RT(   13)
      J(    1)=.NOT.RP(    8)   .AND.0000000000000000000001B
      J(    2)=.NOT.RP(   14)   .AND.0000000000000000000001B
      J(    1)=J(      1)  .OR.       J(      2)
      J(    2)=.NOT.RP(   16)   .AND.0000000000000000000001B
      J(    1)=J(      1)  .OR.       J(      2)
      J(    1)=.NOT.J(      1)  .AND.0000000000000000000001B
```

```
      RT(  10)  =J(      1) .AND.00000000000000000018
      IF(RP(  10)  .NE.RT(  10)  ) CHNG=.TRUE.
      RP(  10)  =RT(  10)
      J(   1)=.NOT.RP(   8)  .AND.00000000000000000018
      J(   1)=RP(  15)  .OR.      J(      1)
      J(   1)=.NOT.J(      1) .AND.00000000000000000018
      RT(  12)  =J(      1) .AND.00000000000000000018
      IF(RP(  12)  .NE.RT(  12)  ) CHNG=.TRUE.
      RP(  12)  =RT(  12)
      J(   1)=RP(  14)  .OR.      RP(  12)
      J(   1)=.NOT.J(      1) .AND.00000000000000000018
      RT(  11)  =J(      1) .AND.00000000000000000018
      IF(RP(  11)  .NE.RT(  11)  ) CHNG=.TRUE.
      RP(  11)  =RT(  11)
      RETURN
      END
      SUBROUTINE ABC
      INTEGER JKFF
      INTEGER DECODER
      INTEGER RP,RT
      LOGICAL CHNG,SSRERR,L
      COMMON/REG/NAMES( 500),NB( 500),RP( 500),RT( 500),NON,CHNG,SSRERR,
     $CCOD
      COMMON/TEMP/J(200),L(200)
      IF(RP(   2).NE.                0)RETURN
      GO TO 257
      ENTRY ABCENT
      RETURN
  257 CONTINUE
      J(   1)=JKFF    (RP(  14)  ,RP(   8)  ,RP(   9)
     $)
      RT(  14)  =J(      1) .AND.00000000000000000018
      J(   1)=00000000000000000018
      J(   1)=JKFF    (RP(  15)  ,J(      1) ,RP(  10)
     $)
      RT(  15)  =J(      1) .AND.00000000000000000018
      J(   1)=JKFF    (RP(  16)  ,RP(  11)  ,RP(  14)
     $)
      RT(  16)  =J(      1) .AND.00000000000000000018
  299 RETURN
    1 RETURN
    2 RETURN
  298 CALL ERROR(33,2,3HABC)
      SSRERR=.TRUE.
  297 RETURN
      END
      SUBROUTINE REGSET
      LOGICAL SSRERR
      COMMON/REG/NAMES( 500),NB( 500),RP( 500),RT( 500),NON,CHNG,SSRERR,
     $CCOD
      COMMON/MEM/MEMNA(6),MEMNB(6),MEMSIZ(6),NCM
      DATA NCM/   0/
      DATA NON/  17/
      DATA NAMES(   1),NB(   1)/10HTIME      ,   21/
      DATA NAMES(   2),NB(   2)/10H          ,    1/
      DATA NAMES(   3),NB(   3)/10H          ,   48/
      DATA NAMES(   4),NB(   4)/10H          ,    1/
      DATA NAMES(   5),NB(   5)/10H          ,   48/
      DATA NAMES(   6),NB(   6)/10HZ1        ,    1/
      DATA NAMES(   7),NB(   7)/10HZ2        ,    1/
      DATA NAMES(   8),NB(   8)/10HX         ,    1/
      DATA NAMES(   9),NB(   9)/10HNX        ,    1/
      DATA NAMES(  10),NB(  10)/10HKB        ,    1/
```

```
      DATA NAMES(  11),NB(  11)/10HJC         ,    1/
      DATA NAMES(  12),NB(  12)/10HJC1        ,    1/
      DATA NAMES(  13),NB(  13)/10HZ          ,    1/
      DATA NAMES(  14),NB(  14)/10HA          ,    1/
      DATA NAMES(  15),NB(  15)/10HB          ,    1/
      DATA NAMES(  16),NB(  16)/10HC          ,    1/
      DATA NAMES(  17),NB(  17)/10HABCSSR     ,    1/
      RETURN
      END
      SUBROUTINE CONTROL
      LOGICAL CHNG,SSRERR,COMERR,CCOD
      INTEGER RP,RT
      COMMON/REG/NAMES( 500),NB( 500),RP( 500),RT( 500),NON,CHNG,SSRERR,
     $CCOD
      COMMON/MAIN/KARD(80),NCR,KP,COMERR,TELE,KHAR,PHRASE,TYPE,TR,MLOOPS
      RP(   2)=MOD(RP(1)-1,   1)
      RT(   2)=RP(   2)
      IF (.NOT.CCOD) GO TO 7
      CCOD=.FALSE.
      NLOOP=0
    6 CHNG=.FALSE.
      CALL COMBOOL
      CALL ABCENT
      NLOOP=NLOOP+1
      IF (NLOOP.GT.MLOOPS) GO TO 20
      IF (CHNG) GO TO 6
    7 CONTINUE
      CALL ABC
      DO 10 I=1,NON
   10 RP(I)=RT(I)
      ENTRY BOOLEAN
      CCOD=.FALSE.
      NLOOP=0
   15 CHNG=.FALSE.
      CALL COMBOOL
      CALL ABCENT
      NLOOP=NLOOP+1
      IF (NLOOP.GT.MLOOPS) GO TO 20
      IF (CHNG) GO TO 15
      RETURN
   20 CALL ERROR(36,3,MLOOPS)
      RETURN
      END
      SUBROUTINE MSET(MEM,NWORD,MDAT)
      COMMON/MEM/MEMNA(6),MEMNB(6),MEMSIZ(6),NOM
   10 RETURN
      END
      FUNCTION MPRT(MEM,NWORD)
      COMMON/MEM/MEMNA(6),MEMNB(6),MEMSIZ(6),NCM
      COMMON/MAIN/KARD(80),NCR,KP,COMERR,TELE,KHAR,PHRASE,TYPE,TR,MLOOPS
      LOGICAL TELE
   10 MPRT=0
      RETURN
      END
      SUBROUTINE MDATA
      RETURN
      END
```

APPENDIX B

LINE NO.	ROM WORD
1	07207
2	21603
3	00002
4	00400
5	00000
6	21603
7	11002
8	40000
9	00000
10	21603
11	03002
12	40000
13	00000
14	00000
15	21603
16	06002
17	22402
18	74043
19	40000
20	31001
21	00402
22	22603
23	40000
24	00000
25	00007
26	00402
27	22603
28	06005
29	40045
30	00005
31	00407
32	40000
33	00000
34	00000
35	07000
36	40000
37	00407
38	40000
39	00000
40	77000

```
MEMORY SIZE IMPLIES   48IT MBAR
  I      MEM(I),OCTAL          MEM(I),DECIMAL
    1 000000000000001013                  523
    2 000000000000001014                  524
    3 000000000000003015                 1549
    4 000000000000007009                 3584
    5 000000000000000000                    0
    6 000000000000000000                    0
    7 000000000000000000                    0
    8 000000000000000000                    0
    9 000000000000000000                    0
   10 000000000000000000                    0
   11 000000000000000000                    0
   12 000000000000000012                   10
   13 777777777777777773                   -4
   14 000000000000000000                    0
   15 000000000000000000                    0
   16 000000000000000000                    0
 ROM BUFFER CONTAINS= 0 7 2 0 7
 ROM BUFFER CONTAINS= 2 1 6 0 3
 ROM BUFFER CONTAINS= 0 0 0 0 2
 ROM BUFFER CONTAINS= 0 0 4 0 0
AT TIME T=      4
        ACC                   DBUS                  MBR
000000000000000000000 000000000000001013 000000000000001013OCTAL
                    0               523                523DECIMAL
         SP=  00000000000000000000
          PC                   IR                  MBAR
000000000000000000001 000000000000001013 000000000000000000000
```

```
MEMORY SIZE IMPLIES   48BIT MBAR
  I      MEM(I),OCTAL          MEM(I),DECIMAL
  1 000000000000001013                   523
  2 000000000000001014                   524
  3 000000000000003915                  1549
  4 000000000000007000                  3584
  5 000000000000000000                     0
  6 000000000000000000                     0
  7 000000000000000000                     0
  8 000000000000000000                     0
  9 000000000000000000                     0
 10 000000000000000000                     0
 11 000000000000000000                     0
 12 000000000000000012                    10
 13 777777777777777773                    -4
 14 000000000000000000                     0
 15 000000000000000000                     0
 16 000000000000000000                     0
ROMAR=   1 ROMPC=   1 ROM STACK=   0
ROM BUFFER CONTAINS= 0 7 2 0 7
AT TIME T=      1
        ACC                    DBUS                   MBBR
000000000000000003000 000000000000000000000 000000000000000000000CTAL
                    0                      0                      0DECIMAL
         SP=   000000000000000000000
          PC                     IR                    MBAR
000000000000000000001 000000000000000000000 000000000000000000000
ROMAR=   2 ROMPC=   2 ROM STACK=   0
ROM BUFFER CONTAINS= 2 1 6 0 3
AT TIME T=      2
        ACC                    DBUS                   MBBR
000000000000000000000 000000000000000000000 000000000000000010130CTAL
                    0                      0                    523DECIMAL
         SP=   000000000000000000000
          PC                     IR                    MBAR
000000000000000000001 000000000000000000000 000000000000000000000
ROMAR=   3 ROMPC=   3 ROM STACK=   0
ROM BUFFER CONTAINS= 0 0 0 0 2
AT TIME T=      3
        ACC                    DBUS                   MBBR
000000000000000000000 000001000000000001013 000000000000000010130CTAL
                    0                    523                    523DECIMAL
         SP=   000000000000000000000
          PC                     IR                    MBAR
000000000000000000001 000000000000000000000 000000000000000000000
ROMAR=   4 ROMPC=   4 ROM STACK=   0
ROM BUFFER CONTAINS= 0 0 4 0 0
AT TIME T=      4
        ACC                    DBUS                   MBBR
000000000000000000000 000000000000000001013 000000000000000010130CTAL
                    0                    523                    523DECIMAL
         SP=   000000000000000000000
          PC                     IR                    MBAR
000000000000000000001 000000000000000001013 000000000000000000000
```

```
          0          7          2          0          7
 CROSSED HALT TEST
 307
 RADDR=    8ABOUT TO CALL TRANS
  REACHED TRANS
 R(I) BEFORE TRANSFER=00000000000000000000
 R(J) BEFORE FINAL MASK 00000000000000000000
 FINISHED R.J   BACK TO MICRO
R(J)= 00000000000000000000
  8  GOING TO EXEC
  8
          2          1          6          0          3
 CROSSED HALT TEST
 703
 RADDR=    4ABOUT TO CALL TRANS
  REACHED TRANS
 R(I) BEFORE TRANSFER=00000000000000000000
 R(J) BEFORE FINAL MASK 00000000000000000000
 FINISHED R.J   BACK TO MICRO
R(J)= 00000000000000000000
 18  GOING TO EXEC
 18
 NB  IMMRY(N3) 00000000000000001013R(33) 00000000000000001013
  REACHED TRANS
 R(I) BEFORE TRANSFER=00000000000000001013
 R(J) BEFORE FINAL MASK 00000000000000001013
 FINISHED R.J   BACK TO MICRO
R(J)= 00000000000000001013
          0          0          0          0          2
 CROSSED HALT TEST
 102
 RADDR=    3ABOUT TO CALL TRANS
  REACHED TRANS
 R(I) BEFORE TRANSFER=00000000000000001013
 R(J) BEFORE FINAL MASK 00000000000000001013
 FINISHED R.J   BACK TO MICRO
R(J)= 00000000000000001013
  1  GOING TO EXEC
  1
          0          0          4          0          0
 CROSSED HALT TEST
 500
 RADDR=    1ABOUT TO CALL TRANS
  REACHED TRANS
 R(I) BEFORE TRANSFER=00000000000000001013
 R(J) BEFORE FINAL MASK 00000000000000001013
 FINISHED R.J   BACK TO MICRO
R(J)= 00000000000000001013
  1  GOING TO EXEC
  1
```