A MICROPROGRAMMED SIMULATION SYSTEM FOR GENERAL PURPOSE REGISTER AND FIXED PURPOSE REGISTER MINICOMPUTERS

By

GLENN RAY THOMPSON II

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1974

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the Degree of
MASTER OF SCIENCE
May, 1976

Thesis
1976
T471m
cop. 2

OKLAHOMA
STATE UNIVERSITY
LIBRARY

AUG 26 1976

A MICROPROGRAMMED SIMULATION SYSTEM FOR GENERAL PURPOSE REGISTER AND FIXED PURPOSE REGISTER MINICOMPUTERS

Thesis Approved:

Donald D Fisher
Thesis Adviser

D. E. Hedrick

James R. VanDoren

A N Durham
Dean of the Graduate College

947669

PREFACE

This thesis is a description of a microprogrammed simulation system for general purpose register and fixed purpose register minicomputers. Such systems aid in the efficiency in which assembler programs are developed for certain classes of minicomputers. The description is designed to instruct the reader in microprogramming techniques and how these techniques might be implemented.

The author wishes to express his appreciation to his major advisor, Dr. Donald D. Fisher, for his guidance and assistance throughout this study. Appreciation is also expressed to other committee members, Dr. George E. Hedrick and Dr. James R. Van Doren, for their invaluable assistance in the preparation of the thesis.

Finally, special gratitude is expressed to my wife, Vicki, for her understanding and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

This thesis is a presentation of a method for developing a microprogrammed simulation system for minicomputers chosen under two classifications (14): general purpose register systems and fixed purpose register systems. Of the classes of minicomputers, these two represent the largest number of minicomputers available on the market today. They are also more closely related than any other two classes of minicomputers.

The first classification consists of minicomputers with a General Purpose Register (GPR) structure. The hardware registers of this type of machine serve many functions. Two functions the registers serve are (1) index registers and (2) general purpose accumulators. In one section of an assembler program, a register may be used as an index register to obtain the effective address of an operand, which is to be used in computations of some form; and in that same section of the program the register may be used to contain the operand itself. Examples of machines with this hardware register configuration are the MODCOMP II and III (15) and the INTERDATA 7/16 (10). A subset of the instructions for the INTERDATA 7/16 has been defined for simulation purposes and is contained in Appendix C along with a sample assembler program run.

Minicomputers in the second classification have a Fixed Purpose

Register (FPR) structure. There are several different types of hardware registers used in such systems. Accumulators are designed to hold intermediate results of computations and serve a significant role in the Arithmetic Logic Unit (ALU). Index registers are used to determine the effective address of some operands and serve as an important tool in the use of data arrays as a primary data structure at the machine level. Extension registers, usually of one bit, serve as overflow or carry indicators for the accumulators. Depending upon the particular system, there may be many more registers that serve various purposes. Examples of machines with this hardware register configuration are the HEWLETT-PACKARD 2114A, 2115A, and 2116B (1), the VARIAN 520/i (20) and the INTERDATA Model 1 (9). A subset of the set of instructions for the HEWLETT-PACKARD 2114A has been defined for simulation purposes and is contained in Appendix C along with a sample assembler program run.

Simulators can be written for a large number of minicomputers in high level languages, such as FORTRAN or PL/I, on large host computers. This facilitates the incorporation of more sophisticated diagnostics into the simulation system, thereby decreasing the amount of debugging time required for any particular assembler language program. One reason this sophistication is usually not built into most minicomputer translation systems is the relatively small amount of main core storage available to the system. With this restriction, the producers of systems software must keep the size of the translators down to a minimum.

Minicomputers are playing an increasingly more important role in the computing industry. This is one reason for the development of

more generalized simulation systems. Minicomputers are used for such purposes as laboratory machines to monitor experiments, interface devices in computer networks and interface devices between large computer systems and peripheral devices. In some cases they are even used for automobile care and maintenance. Their versatility is mainly due to the fact that most minicomputers are bus oriented machines. A data bus allows information from external devices or internal registers to be transferred to logic units and back again along a single data path. This helps to lower hardware costs and make the system more flexible. It is not unusual for as many as 256 external devices to be connected to a single data bus.

An introduction to the subject of microprogramming is given in Chapter II. Background information and an illustration of a small microprogrammed system are discussed. An explanation of some of the differences between fixed instruction computers and microprogrammed computers is also given.

The simulation system itself is composed of a cross assembler and an interpreter. The two pass assembler is described in Chapter III. The scanner for detecting labels and operation codes, pass I symbol table construction and object (machine) code generation in pass II are discussed. Assembly time error detection and diagnostics are also discussed.

The object code generated by the assembler is input to the interpreter. The interpreter is actually a microprogrammable pseudo-machine for which microprograms are designed to simulate instruction execution for a desired system. Pass II of the assembler loads the simulated memory with the generated object code and it is from this simulated

memory that the interpreter obtains the machine instructions. Chapter IV contains a description of instruction fetch and execution, execution-time error detection, and input/output formats. Microinstruction formats and the development of microprograms for the simulated instructions are also discussed. It is not the purpose of this thesis to treat the subjects of input/output and interrupt servicing in great detail. Therefore, these subjects are only briefly discussed in terms of the microinstructions required to perform their basic functions.

The simulation process can be thought of as a sequence of state transitions. Figure 1 and Table I illustrate the state transitions involved in the simulation process.

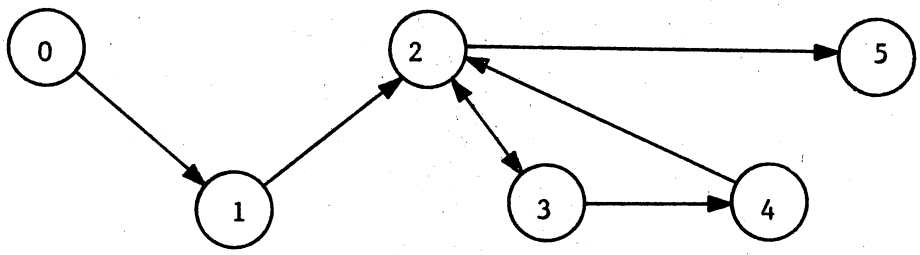


Figure 1. State Transitions of the Simulation Process.

Chapter V is a Users Manual and describes the deck setup and options for using the assembler-interpreter. The assembler output format and error messages are also discussed. A summary and further study is presented in Chapter VI. Appendix A contains a list of symbols used in

the description of the simulation system. A logic block diagram of the system is given in Appendix B and Appendix C contains sample assembler program runs and the output. Appendix D is a systems programmers guide to modifications of the simulation system.

TABLE I

DESCRIPTION OF STATE TRANSITIONS
FOR FIGURE 1

State	Description
0 (Start state)	Hardware register definition
1	Instruction set definition
2	Job control recognition
3	Assembly of source program
4	Object program interpretation
5 (Final state)	Termination of job stream

The basis for the method of construction of the simulator presented in this thesis was developed by Hill and Peterson (7) in their description of a microprogrammed Small Instructional Computer (SIC). This computer has a fixed purpose register structure with two accumulators, two index registers and one extension register. The instruction set is sufficiently large to illustrate most points about a microprogrammed system. The design of the microinstruction set was influenced by Husson (8) in his description of the microprogramming techniques for large scale computers.

CHAPTER II

MICROPROGRAMMING

Introduction

The term microprogramming was first coined by Professor M. V. Wilkes of the Cambridge University Mathematical Laboratory in 1951 (21, 22). In his thesis he stated that one can envision the control portion of a computer as effecting a number of register-to-register transfers of information in order to carry out the execution of a single machine instruction. Each of these steps can itself be thought of as the execution of an instruction for some machine (whose existence is unknown to the programmer). The steps used to effect a single instruction in the user machine can be thought of as constituting a program, usually called a microprogram. Microprograms can also be used for other necessary operations which are in some sense invisible to the programer, for example, fetching the next instruction or computing effective addresses.

There are at least two approaches to microprogrammed control, and they differ significantly from one another. One, called "vertical or sequential microprogramming" (18), relies on the more traditional concept of programming in which an instruction contains an operation code, secondary modifiers and one or more address fields. In this case three addressable storage areas are associated with the host machine: (1) main store, (2) control store and (3) local store.

Local store consists of registers and can be conceived of as being a general purpose storage area for use by the microprograms.

The other approach, called "horizontal microprogramming" (18), uses the microinstructions as control words whose individual bits are used to select specific data paths within the machine. In this case there are no addresses other than those implicitly specified by the bits of the control words. Although this scheme is less general than vertical microprogramming, it is possible to make more efficient use of the hardware of machines organized in this way. A modified version of the horizontal microprogramming scheme is used in the interpreter which is described in detail in Chapter IV.

In either case, the microinstructions generally are held in a control store, usually faster than main store, for which there must be some form of accessing mechanism. The majority of such systems use a non-destructive read-only store (ROS or RCM) for reasons of speed and economy. The read-only nature of these devices also insures the integrity of the simulated machine.

Alternatively, a few machines are now equipped with writable control stores implemented in core arrays. Such core arrays generally are built so that they have the properties of fast reading and slow writing (16). Fast reading is necessary for performance and slow writing can help in cost reduction.

Fixed Instruction Computers

In order to contrast some of the aspects of fixed instruction computers and microprogrammed computers, a description of some of the major characteristics of fixed instruction computers follows.

The hardware of fixed instruction computers generally is divided into four units (Figure 2): (1) an input/output unit, (2) memory (or storage), (3) an arithmetic-logic unit, and (4) a control unit. In general, the input/output unit consists of more than one physical device but input and output are often considered together.

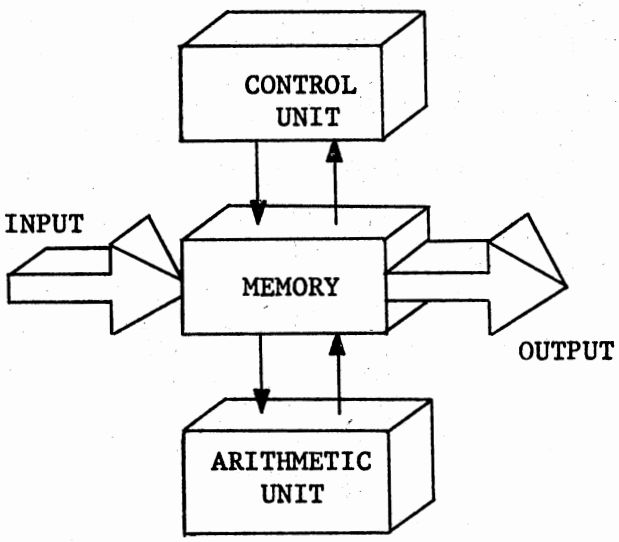


Figure 2. Simplified Block Diagram for Fixed Instruction Stored Program General Purpose Computers [From Microprogramming Handbook (14)]

Memory is considered in its conventional way, a series of memory cells and registers used to store the instructions of a program while in execution. Addresses are associated with each memory cell and with all registers. The control unit of the computer can refer to a storage

register by its number (or address). This is called the address property of the storage register (6). Storage registers have two other properties: (1) non-destructive read-out and (2) destructive write-in. Non-destructive read-out refers to the property of accessing the contents of a register without destroying it. Destructive write-in refers to the property of destroying the previous contents of a register when information is stored into the register.

All arithmetic is performed in the arithmetic-logic unit. There is at least one accumulator or general purpose register in the ALU of any computer. The structure of the machine is such that the contents of the accumulator or general purpose register can be tested to determine the characteristics of the value. Usually, this simply means to determine if the value is less than, equal to, or greater than zero. This is the basis for all logical and conditional operations that may be performed by the system.

The control unit coordinates the interaction of all the other units. When a program is in execution, the control unit retrieves instructions from memory in the proper order and also initiates the execution of each instruction. In the simplified case, the control unit has two special registers: (1) the instruction register (IR) and (2) the program counter (PC) or program status word. The PC contains the address of the next instruction to be executed. The IR contains the machine instruction currently being executed. Generally, the ALU and the control unit are together called the Central Processing Unit (CPU).

Microprogrammed Computers

Microprogrammed computers are very similar to fixed instruction

computers, but there is one significant difference: the control memory (Figure 3). Control memory is divided into sections. Each section contains the microinstructions that perform the operations of a particular machine instruction. Each time a machine instruction is fetched, it goes through a decoding process that breaks the instruction up into its appropriate fields and then control is transferred to the microprogram associated with the particular instruction. In the most general sense, microprograms can have some degree of modularity with the use of branching instructions and microprogram subroutines.

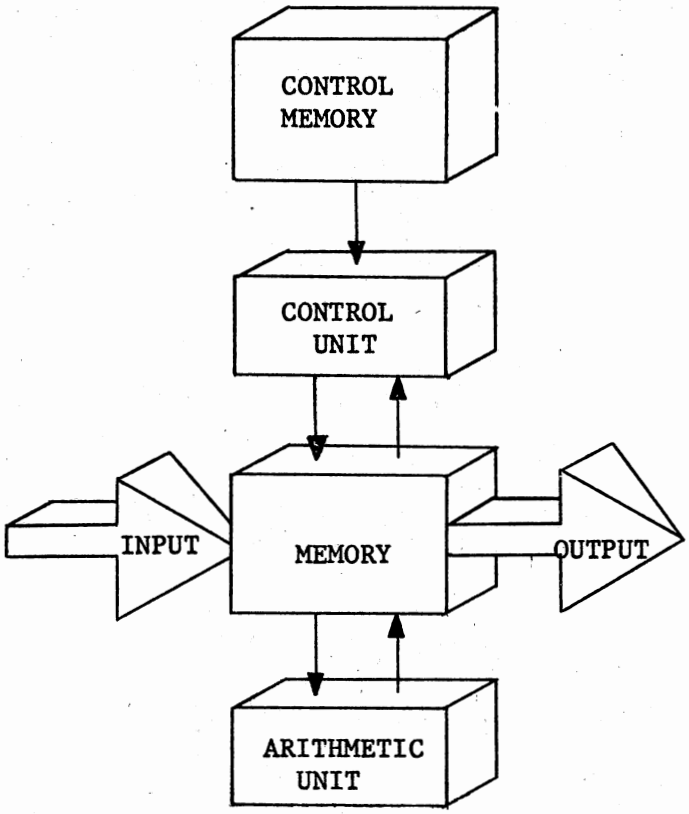


Figure 3. Simplified Block Diagram for Microprogrammed Computers [From Microprogramming Handbook (13)]

Microprogramming represents a systemic approach to control. Figure 4 depicts a ficticious vertical microprogrammed machine and its control functions. The characteristics of this machine can be summarized as: stored program, word organized, single address (18).

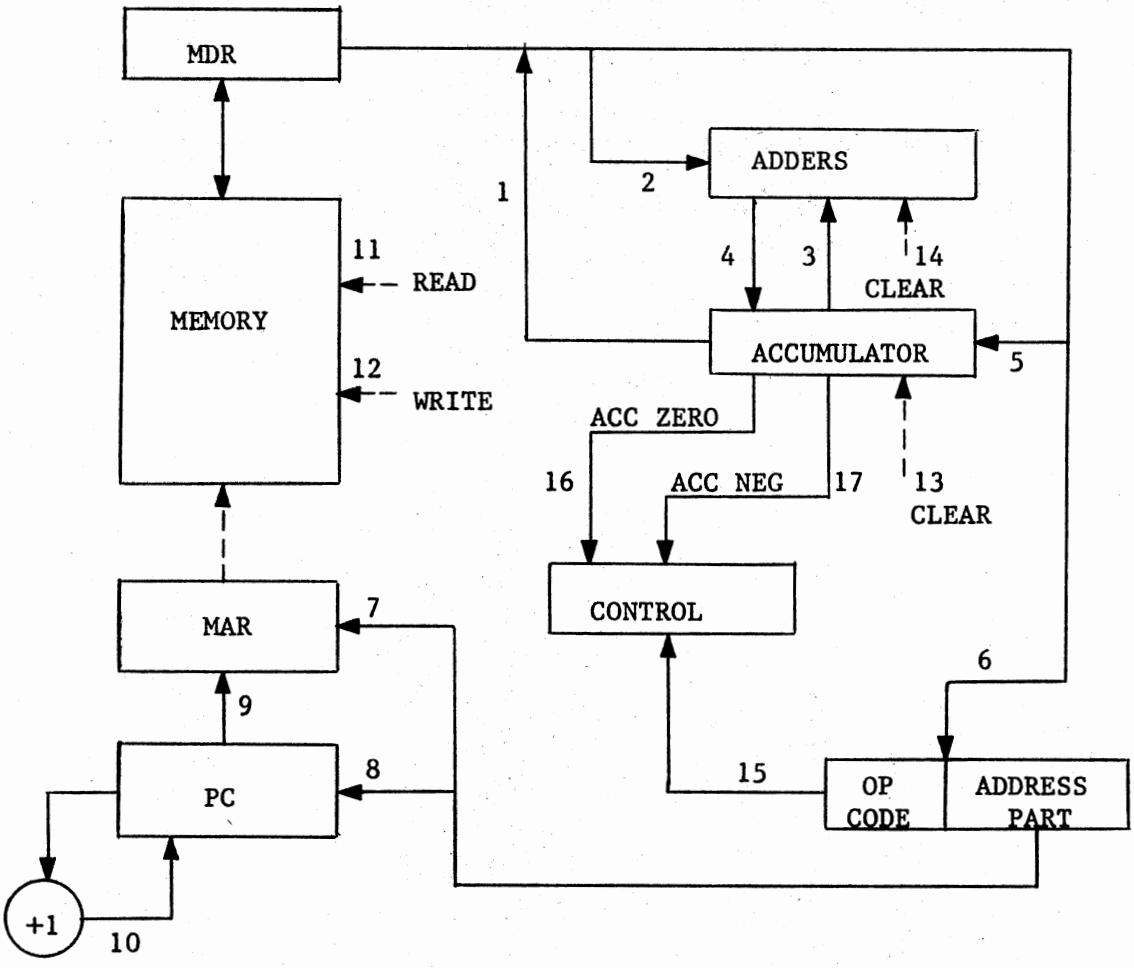


Figure 4. A Simple Fixed Purpose Register Machine
[From Rosin (18)]

The register and storage structure is not unlike many modern fixed purpose register computers. But, in general, many modern computers have a series of accumulators, multiple address instructions, and a multipurpose Program Status Word (PSW). The machine in Figure 4 does not use the bus concept of data transfer, for ease of illustration.

The execution or interpretation of an instruction can be considered as a series of register-to-register transfers. A few other primitive functions have been added, such as clearing a register and initiating main memory reads and writes. Table II summarizes the valid operations of the machine in terms of these primitives (18).

TABLE II

CONTROL LINES FOR FIGURE 4
[From Rosin (18)]

1	MDR ← ACC
2	ADDERS ← MDR
3	ADDERS ← ACC
4	ACC ← ADDERS
5	ACC ← MDR
6	IR ← MDR
7	MAR ← ADDRESS PART
8	PC ← ADDRESS PART
9	MAR ← PC
10	PC ← PC + 1
11	MS READ
12	MS WRITE
13	CLEAR ACC
14	CLEAR ADDERS
	BRANCH INSTRUCTION
15	CONTROL ← OPCODE
16	CONTROL ← AC = 0
17	CONTROL ← AC NEG
18	UNCONDITIONAL

For this particular machine the ACC is used strictly for computational purposes and for conditional (negative or zero) branching. The ADDERS work together with the ACC to perform the necessary arithmetic. The machine instructions consist of an operation code and an operand field and the IR is used in decoding the instructions. The MDR and MAR are used in conjunction to perform the memory access functions. The MDR contains the data word and the MAR contains the address of the data word. The PC is used as a pointer to the current instruction for instruction fetch and may only be incremented to point to the next memory word. The functions of the CONTROL unit, in this case, are few in number: fetch and decode the next instruction and handle any necessary branching.

To implement a simple ADD instruction (addition of the contents of the MDR to the contents of the ACC) this sequence of microinstructions may be specified.

14 CLEAR ADDERS

2 ADDERS ← ACC

3 ADDERS ← MDR

1 ACC ← ADDERS

Except for the fact that these four instructions contain no address parts, the sequence resembles a short conventional program.

As in any other automatic computer, microprograms require the capability to execute branches of control, both conditional and unconditional. In this simplified machine, conditional branches may be based on a zero or negative value in the ACC or the opcode itself. In the latter case, the value of the opcode determines the destination of the branch. For instructions 16, 17, and 18, a single address

part must be specified.

Figure 5 contains a set of microprograms which simulates a simple machine using the organization of Figure 4. Address modification is not used and, in this case, it is not possible to modify the micro-program store. IFETCH1 simply updates the PC to point to the next instruction. IFETCH2 loads the next machine instruction into the IR and transfers the control of the program to the control unit. Note that the PC must be initialized at some point before the execution of the first machine instruction. ADD fetches a data word from memory and adds the data word to the contents of ACC. The result is placed into the ACC. CLEAR ADD simply fetches a data word from memory and loads it into the ACC. STORE ACC stores the contents of the ACC into the memory word designated by the address part of the IR. TRA loads the address part of the instruction into the PC and branches to IFETCH2 (a conventional branch or jump). TRA IF ACC NEG tests the ACC for a negative value. If the value is negative, then a branch is made to TRA, otherwise the next sequential instruction is fetched. STORE ZERO uses the ACC to store a zero value in a memory data word.

Figure 5 illustrates how the instruction set of a simple machine may be represented by a set of microprograms. The same concepts apply to the instruction sets of more sophisticated machines. Chapter IV discusses a more general approach to the development of the instruction sets for GPR and FPR minicomputers than the approach discussed in this chapter.

```
10                              IFETCH1:  PC ← PC + 1
 9                              IFETCH2:  MAR ← PC
11                                        READ
 6                                        IR ← MDR
15                                        CONTROL ← OPCODE

 7                                  ADD:  MAR ← ADDRESS PART
11                                        READ
14                                        CLEAR ADDERS
 2                                        ADDERS ← ACC
 3                                        ADDERS ← MDR
 4                                        ACC ← ADDERS
18 (IFETCH1)                              GO TO IFETCH1

 7                             CLEAR ADD:  MAR ← ADDRESS PART
11                                        READ
 5                                        ACC ← MDR
18 (IFETCH1)                              GO TO IFETCH1

 7                             STORE ADD:  MAR ← ADDRESS PART
 1                                        MDR ← ACC
12                                        WRITE
18 (IFETCH1)                              GO TO IFETCH2

 8                                  TRA:  PC ← ADDRESS PART
18 (IFETCH2)                              GO TO IFETCH2

17                         TRA IF ACC NEG:  IF ACC < 0 GO TO TRA
18 (IFETCH1)                              GO TO IFETCH1

14                             STORE ZERO:  CLEAR ADDERS
 3                                        ADDERS ← ACC
13                                        CLEAR ACC
 1                                        MDR ← ACC
 7                                        MAR ← ADDRESS PART
12                                        WRITE
 4                                        ACC ← ADDERS
18 (IFETCH1)                              GO TO IFETCH1
```

Figure 5. A Simple Simulator
[From Rosin (18)]

CHAPTER III

THE CROSS ASSEMBLER

The first step in the simulation process is the conversion of assembler language source code into machine executable object code. This chapter contains a discussion of how this objective is achieved. Included is the assembly process, code generation, error detection and processing and loading the generated code into the simulated memory for execution.

To translate the source assembly program, the assembler must (1) replace each mnemonic op code with its equivalent binary code, and (2) replace each symbolic address with its corresponding location in memory. To achieve the former a table is kept with all the mnemonic op codes and corresponding binary code along with other vital information for the assembly process. Each time an assembler source statement is encountered, a table lookup is performed in order to determine the binary code equivalent. The op code table is generated in an earlier phase of the simulation process. This topic is discussed in Chapter V. In order to achieve (2) it is necessary to keep another table, called a symbol table. Each time a symbolic reference is made, a table look-up is performed in order to determine the corresponding absolute machine address.

Two scans of the source code are required to complete the assembly process for the cross assembler. It is possible to make only one

scan and achieve the same end, but the process is much more complicated and requires additional "housekeeping". The first scan determines which location in memory is to be assigned to each symbol and during the second scan the assembler produces the binary object code. Each phase of the assembly process is described in the following paragraphs along with the method used for its implementation.

Scanner

The scan phase serves two purposes: (1) allows the label identifier to be separated from the source statement so it may be placed into the symbol table, and (2) allows the op code to be separated from the source statement so that a search may be made of the op code table to find the equivalent binary code. The process of isolating the identifier can be handled in several ways. One way to accomplish this is by the development of a Finite State Automaton (FSA) to recognize the identifiers. Figure 6 represents a possible FSA for recognizing identifiers.

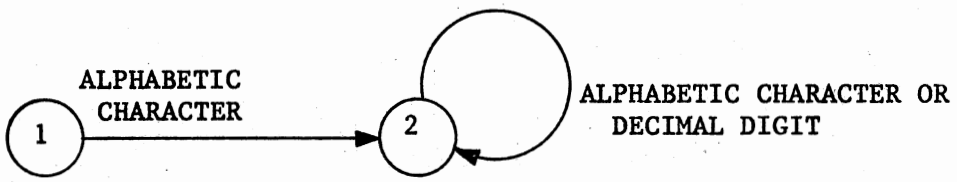


Figure 6. FSA for Recognizing Label Identifiers

State 1 is the "start" and a transition to the "final" state,

state 2, may only be made under an alphabetic character. A transition from state 2 to state 2 may be made under an alphabetic character or a decimal digit. As each transition is made, the character recognized is catenated onto the current symbol to form the complete identifier. Notice that a count must be kept of the length of the symbol so that the symbol will not become larger than the space available for the identifier in the symbol table.

Another method for isolating the identifier, which is similar to the first method, is the simple catenation of valid characters in the label field onto the current symbol. The catenation process is continued until a blank character or the end of the field is encountered. The latter method is the one used in the cross assembler described later in this chapter.

The scan phase for recognizing the mnemonic op codes can be constructed in exactly the same way as the phase for recognizing the label identifiers. The only difference being in the mnemonic op code table lookup performed after the field has been scanned.

Symbol Table Construction and Processing

The operation code symbol table is constructed prior to the assembly process so it will not be treated in this section. It is discussed in detail in Chapter V. The table discussed in this section is the identifier symbol table and its general structure is shown in Figure 7. Notice that the symbol table is actually composed of two tables: (1) the symbol definition table, and (2) the symbol reference table. Table (1) contains the identifier, its value, the statement number in which it was defined and a pointer to the reference

table (Figure 8a). Table (2) contains the statement numbers where the identifier is referenced. The reference for each identifier are linked by means of pointers to the subsequent references. The reference table is kept so that a cross-reference listing may be printed after a program has been assembled. Figure 8b shows the node structure for each entry in the symbol reference table.

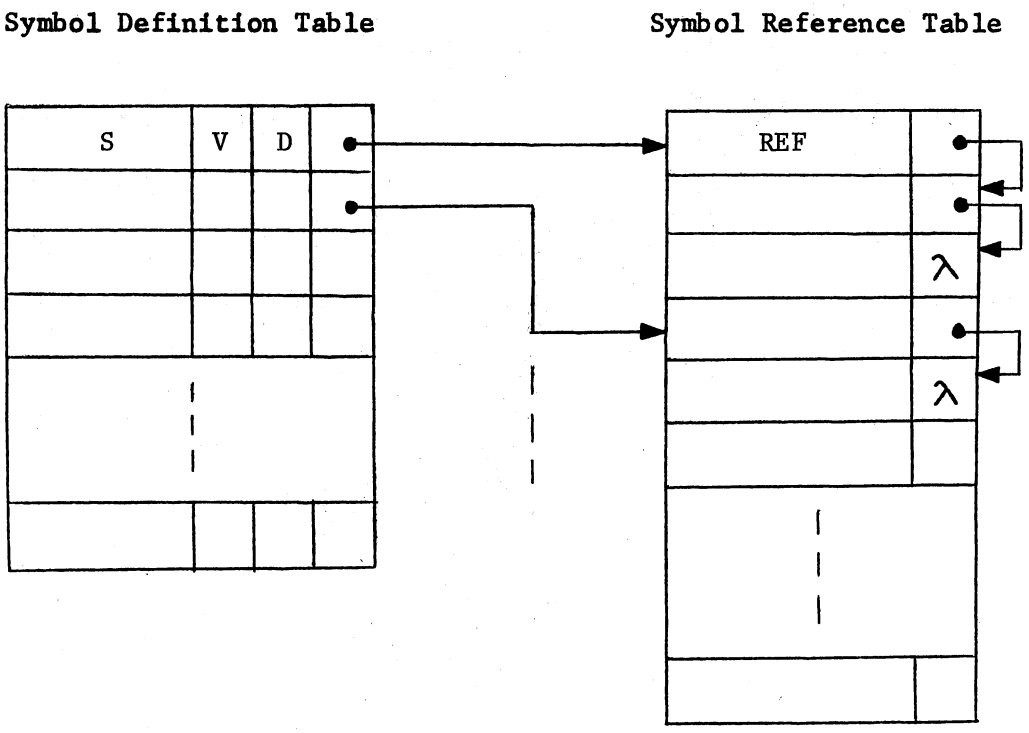


Figure 7. The Identifier Symbol Table

The symbol table is constructed in a sequential manner and table lookup also is done sequentially. This method may be somewhat slower than other methods, such as "hashing" techniques or binary searching, when the table becomes very large, but it is straight forward and does

not require function evaluations and/or pointer updating when a symbol is defined or referenced.

Identifier	Value	Statement declared	Pointer to the reference table

(a) node format for symbol definition table

Statement referenced	Pointer to next reference

(b) node format for symbol reference table

Figure 8. Node Formats for Symbol Table

Pass I

Pass I has two primary tasks: (1) to construct the identifier symbol table, and (2) to determine the binary code representation for the mnemonic op code and substitute it into the machine instruction. A location counter (or program counter) is kept in both pass I and pass II in order to determine the exact location in the simulated memory to place the instruction. After each instruction is interpreted, the program counter is incremented by the length of the instruction.

The first step is to read a new line of the source assembly program. Since pass II will need to reread the input source program, a copy of the input is produced on an auxiliary storage device.

Label Field

An identifier found in the label field is placed into the symbol

table along with the current value of the program counter. Before an identifier is placed into the table, a search is made to ensure that the symbol has not been previously defined.

Mnemonic Op Code Field

After a mnemonic op code has been recognized, an operation code table lookup is performed. If the mnemonic is not present, a record of the error is kept. If present, the binary code equivalent is placed into the machine instruction.

Operand Field

There are five instructions for which the operand field is parsed during pass I: (1) EQU, (2) ORG, (3) DC, (4) DS and (5) END. For these instructions it is possible to interpret the contents of the operand field the first time the instruction is encountered. Note that this places a restriction upon the use of the EQU, DC, and DS instructions. It must be possible to resolve all symbolic references at the time the instructions are first encountered. These instructions are provided by the simulation system and need not be defined by the user. The formats for these instructions are given in Chapter V.

Pass II

The purpose of pass II is to complete the machine instructions by using the symbol table constructed in pass I and to resolve all symbolic references. The operand field may be divided into as many as five separate fields, depending upon the type of machine being simulated. Commas are used as delimiters in the operand field. The

relative position of a value in the operand field dictates its position in the machine instruction. The assembler language operand field syntax for GPR systems is given in Table III and the operand field syntax for FPR systems is given in Table IV.

TABLE III

OPERAND FIELD SYNTAX FOR GENERAL PURPOSE REGISTER SYSTEMS

Type	Syntax
1	X1
2	D1
3	R1,R2
4	B1,D1
5	R1,K1
6	R1,D1
7	R1,X1
8	S1
9	R1,S1
10	R1,R2,S1
11	R1,X1,S1
12	R1,D1,S1
13	B1,D1,S1
14	R1,I1
15	R1,B1,D1
16	S1,S2
17	S1,I1
18	D1,S1
19	R1,D1,D2
20	R1,R2,D1
21	blank field

The instruction type dominates the valid combinations of operands in the operand field. The meaning of the symbols used in the operand field definitions of the two classes of machines is given in Appendix A.

Examples of the types of operands that can be specified for GPR machines, taken from Table III, follow. The syntax of the operand specifications closely resemble that of the simulated test machine, the INTERDATA 7/16.

```
 2     +12
 8     SYM-2400
11     0,15,*+48
13     12,-64,*
16     SYM1,SYM2+4
17     SYM,=A(*-28)
20     9,10,+8
```

Examples of the types of operands that can be specified for FPR machines, taken from Table IV, follow. The syntax of the operand specifications closely resemble that of the simulated test machine, the HEWLETT-PACKARD 2114A.

```
1      ALPHA+4
2      =X'00FF'
3      OPR1,OPR2,OPR3
5      12
```

TABLE IV

OPERAND FIELD SYNTAX FOR FIXED PURPOSE REGISTER SYSTEMS

Type	Syntax
1	S1
2	I1
3	operate function 1,...,operate function 5
4	blank field
5	K1

Grammars are constructed to aid in the translation of the operand field. Since PL/I supports recursive procedures, the grammars are coded directly into the PL/I source language. Figure 9 illustrates the grammar used to parse the operand fields of the assembler statements for GPR systems. Figure 10 illustrates the grammar used to parse the operand fields of non-operate instructions for FPR systems and the grammar used for operate instructions is given in Figure 11.

```
ARFLD:  DECL; DSTOR; RSPEC; ASPEC; ASPEC , ASPEC; ASPEC , ASPEC , ASPEC.
DECL:  B QDIGITS; F QDIGITS; X QDIGITS; A LADDR.
QDIGITS:  ' SDIGITS '.
SDIGITS:  + DIGITS; - DIGITS; DIGITS.
DIGITS:  DIGIT DIGITS; DIGIT.
DSTOR:  PDV DECL.
LADDR:  ( ASPEC ).
RSPEC:  SERPDV; ASPEC, = DECL; PDV; ASTARTH , ASPEC; ASTARTH.
SERPDV:  PDV ADITSPEC.
ADITSPEC:  COMPDV; COMASPEC.
COMPDV:  CTERMPDV COMASPEC; CTERMPDV.
CTERMPDV:  , PDV;  , ASTARTH.
COMASPEC:  , ASPEC;  , ASTARTH.
PDV:  DECINT.
DECINT:  INT DECINT; INT.
ASPEC: * ASTARTH; *; SYMADDR; PDV.
ASTARTH:  + PDV; - PDV.
SYMADDR:  SYMI ASTARTH; SYMI.
SYMI:  LET SYMJ; LET.
SYMJ:  LET SYMJ; INT SYMJ; LET; INT.
INT:  0; 1; 2; 3; 4; 5; 6; 7; 8; 9.
LET:  A; B; C; D; E; F; G; H; I; J; K; L; M; N; O; P; Q; R; S; T; U;
      V; W; X; Y; Z; &; $; %; #; @.
DIGIT:  0; 1; 2; 3; 4; 5; 6; 7; 8; 9; A; B; C; D; E; F.
```

Figure 9. Grammar for Parsing Operand Fields of General Purpose Register Systems

The grammars illustrated in Figures 9, 10 and 11 use certain conventions for distinguishing metasymbols, nonterminal symbols and

terminal symbols (5). The nonterminal symbols on the left hand side of a rule are always followed by the metasymbol, colon (:). The right hand side alternatives of the rules immediately follow the colon and are separated by a semi-colon (;). The last alternative of a rule is followed by a period (.). All symbols not appearing on the left hand side of a rule are terminal symbols and all other symbols which are not metasymbols are nonterminal symbols.

```
ARFLD:  DECL; DSTOR; ASPEC; = DECL.
DECL:  B QDIGITS; F QDIGITS; X QDIGITS; A LADDR.
QDIGITS: ' SDIGITS '.
SDIGITS: + DIGITS; - DIGITS; DIGITS.
DIGITS:  DIGIT DIGITS; DIGIT.
DSTOR:  PDV DECL.
LADDR:  ( ASPEC ).
PDV:  DECINT.
DECINT:  INT DECINT; INT.
ASPEC:  * ASTARTH; * ; SYMADDR; PDV.
ASTARTH:  + PDV; - PDV.
SYMADDR:  SYMI ASTARTH; SYMI.
SYMI:  LET SYMJ; LET.
SYMJ:  LET SYMJ; INT SYMJ; LET; INT.
INT:  0; 1; 2; 3; 4; 5; 6; 7; 8; 9.
LET:  A; B; C; D; E; F; G; H; I; J; K; L; M; N; O; P; Q; R; S; T; U;
      V; W; X; Y; Z; &; $; %; #; @.
DIGIT:  0; 1; 2; 3; 4; 5; 6; 7; 8; 9; A; B; C; D; E; F.
```

Figure 10. Grammar for Parsing Non-Operate Operand Fields of Fixed Purpose Register Systems.

Grammars give a systematic method for parsing the operand fields of the assembler statements. Using a grammar allows the operand field to be separated into sub-fields and the semantic meaning of each sub-field determined. For example, if an identifier is encountered, a sequential search through the symbol table can be made to find the

value of the identifier. Once the value has been determined, parsing of the field may continue in order to evaluate any modifiers of the symbol value.

```
OPRSPEC:  OPRLIST.
OPRLIST:  OPR , OPRLIST; OPR.
OPR:  ALPHA ACHAR; ALPHA.
ACHAR:  ALPHA ACHAR; NUM ACHAR; ALPHA; NUM.
ALPHA:  A; B; C; D; E; F; G; H; I; J; K; L; M; N; O; P; Q; R; S; T; U;
        V; W; X; Y; Z; &; $; %; #; @.
NUM:  0; 1; 2; 3; 4; 5; 6; 7; 8; 9.
```

Figure 11. Grammar for Parsing Register Operate Operand Fields of Fixed Purpose Register Systems

Error Detection

Error detection is accomplished in both passes of the assembler. Invalid symbols and mnemonics are detected by the scanner in pass I. Syntactical errors are detected during the evaluation of the expressions in the operand fields. The use of grammars aids in the error detection facility. The grammars define the form that must be followed in specifying operands. An error occurs any time the form is not followed. As each error is detected, the code for the error is placed into the identification record for the assembler statement. When the source listing is printed in pass II, the errors detected in each statement are printed immediately following the statement. A list of all assembly errors is given in Chapter V.

Assembler Output

In addition to code generation, the assembler normally produces a printed output. The output consists of the location of each instruction (in hex), the object code (in hex), the statement number and a listing of each source statement. Errors, if any, are indicated immediately following the statement in error. Following the source listing is the symbol table and cross-reference dictionary which contains the symbols, their associated values (in hex) and a list of the statements in which the symbol is referenced. An indication as to the number of assembly errors is also given.

Loader

The machine instructions generated throughout the assembly process are loaded into the simulated memory for execution. For a GPR system, loading begins at memory location zero. But for FPR systems, loading begins at the first location in page one of the simulated memory. In this context, a page refers to a physical memory page used by most fixed purpose register minicomputers in address/page mapping schemes. Page zero is used by the assembler for patching direct and indirect addresses that cannot be accessed in the current page. The user is not permitted to access any memory location in page zero, other than location zero. This helps to keep all indirect references intact. Once loading is completed the program is ready for execution.

CHAPTER IV

THE INTERPRETER

The binary object code generated by the assembler is loaded into the simulated memory for execution. The machine instructions must be interpreted to carry out the processes of the simulated machine. This chapter describes how the machine instructions are fetched from memory and decoded, how the microinstructions are used in the decoding process, error detection and debugging aids.

Instruction Fetch and Execution

Execution begins with the first instruction of the machine language program. The address of this instruction, along with other vital information is known to the interpreter through the use of "global" variables. Once the address is determined in the assembly phase of the simulation process, the global variable is assigned the address of the first executable instruction. The information contained in the global variable is passed to the PC and primes the instruction fetch cycle. The complete instruction fetch and execution cycle, as shown by Hedrick (6), is illustrated in Figure 12. This is an overview of the subject matter discussed in the remaining portion of the chapter. The actual implementation of the fetch-execute process is much more complicated than is shown. The PC and IR are the most important registers used in the fetch-execute cycle. Once the

address of the next instruction is determined, the instruction is extracted from memory and loaded into the IR.

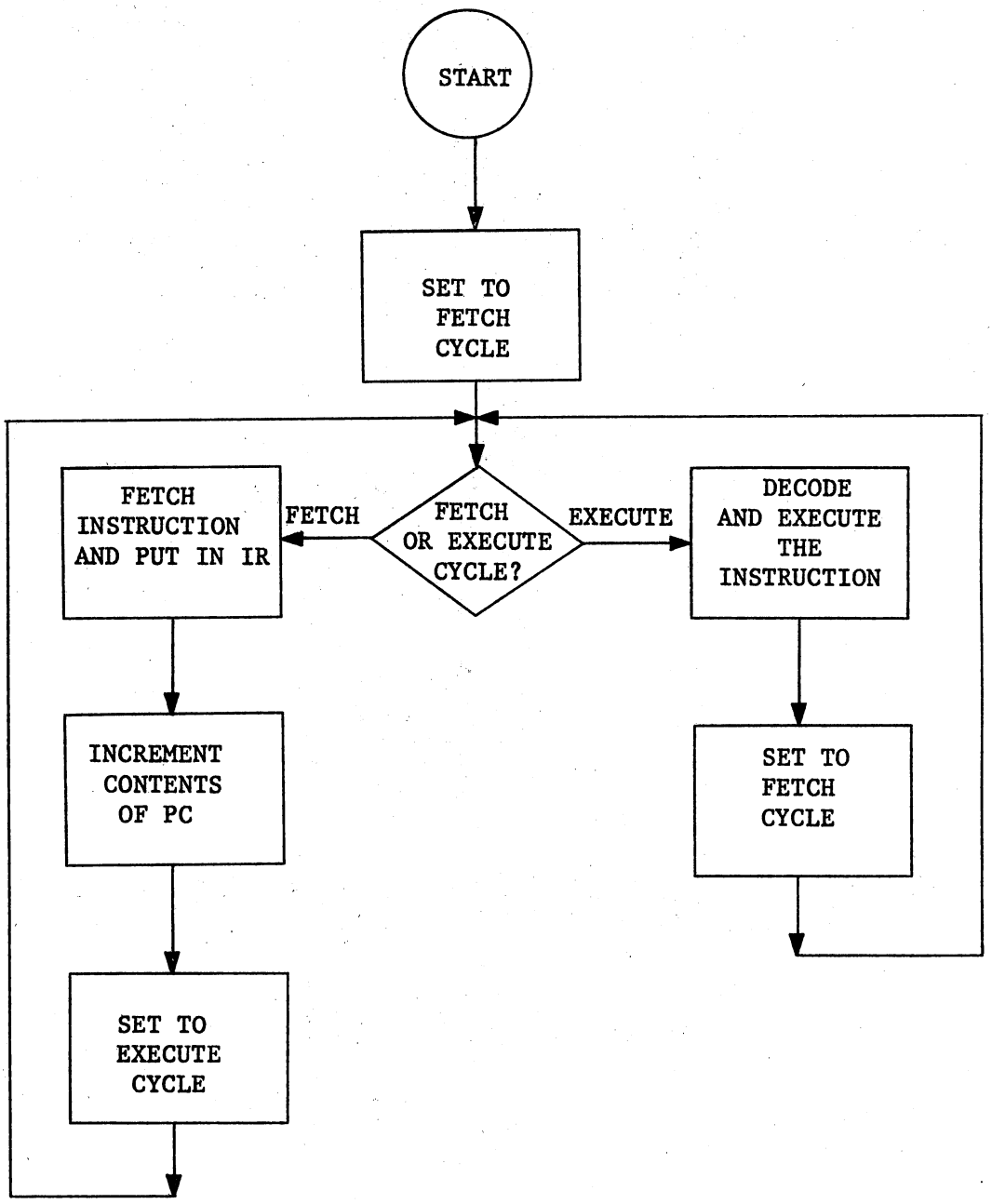


Figure 12. Instruction Fetch and Execution Cycle

At this point the PC is updated and the instruction is subjected to the decoding process. The decoding process consists of determining the registers and/or addresses specified in the instruction, performing a table lookup of the binary operation code and executing the corresponding microprogram. The machine instruction formats for GPR and FPR systems are displayed in Figures 13 and 14, respectively. (See Appendix A for the symbols used in the two figures.) Notice that the operation code must appear in the same field of every instruction. The interpreter must know where the operation code resides within the machine instruction even before the instruction format is known. This and other related information is part of the initial machine definition and is discussed in Chapter V.

The implementation of a simulation system for the INTERDATA 7/16 required the use of eight of the instruction formats displayed in Figure 13: 1, 3, 5, 8, 9, 11, 14, and 21. Fifty-four of the instructions for the INTERDATA 7/16 have been simulated and a number of them tested in a sample program (See Appendix C).

The implementation of the HEWLETT-PACKARD 2114A required the use of three of the instruction formats of Figure 14: 1, 3, and 4. Appendix C displays the program used to test the simulation system for the HEWLETT-PACKARD 2114A. Machine instruction format 3, Figure 14, is used to specify the register operate instructions. Operate instructions manipulate the contents of internal hardware registers and exercise certain control functions without referencing the random access simulated memory.

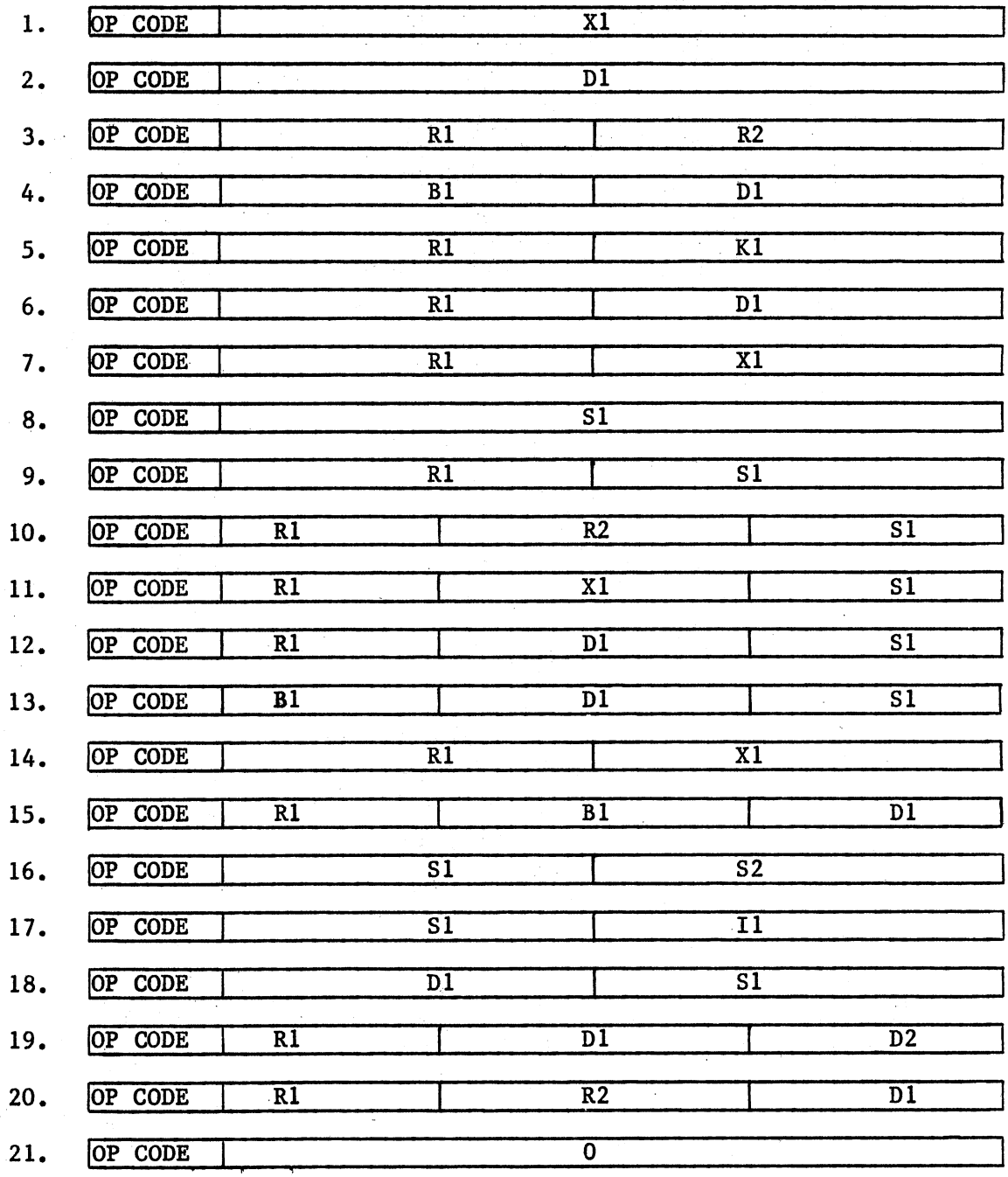


Figure 13. Machine Instruction Formats for General Purpose Register Systems

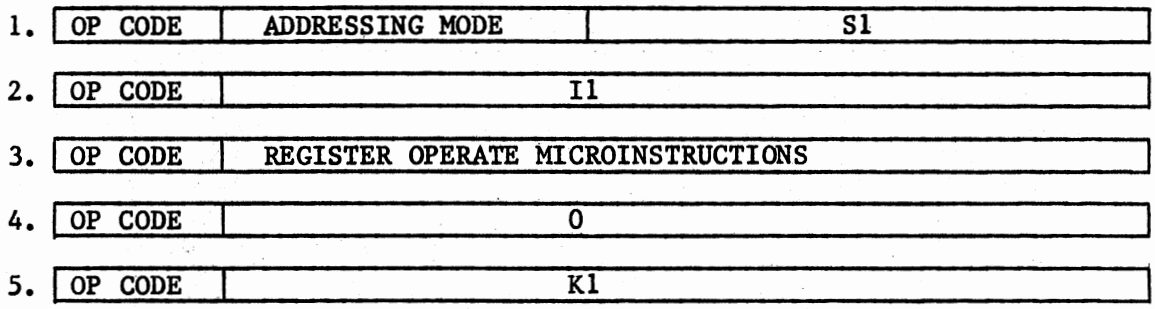


Figure 14. Machine Instruction Formats for Fixed Purpose Register Systems

Machine Instruction Decoding

As part of the execution phase of the simulation system, the decoding process consists of three main operations: (1) performing a table lookup of the binary operation code, (2) determining the registers and memory locations involved in the execution of the instruction, and (3) executing the microprogram that corresponds to the instruction.

The first operation is relatively simple. The operation code is extracted from the instruction and placed into a temporary location. The op code table is then searched sequentially until the op code is found. Once the operation code is located, all information (instruction format, instruction length, number of operands, etc.) concerning the instruction becomes available.

The function of the instruction format is to show how the operand fields of the instruction are to be used. It specifies when an operand field of a machine instruction represents a register or when it represents an absolute memory address. For example, if instruction

format 1 (Figure 13) is used, field 2 of the instruction specifies an index register. The information contained in the operand fields of the machine instructions is used in the execution of the microprogram. This explains the function of operation (2).

The third operation, executing the microprogram, is the most complex of the three operations. Each eleven digit microinstruction is placed into the Micro Instruction Register (MIR) (see Figure 15) and is subjected to a decoding process of its own. The MIR is a decimal register and the sequence of decimal digits specify the micro-operations to be performed. Figure 16 displays the register and bus configuration for valid micro-operation specifications for the microprogrammable pseudo-machine. The user need only concern himself with the microinstructions necessary to accomplish the simulation of a desired instruction set. Instruction fetch and program counter update are taken care of automatically by the interpreter. The micro-operations are defined in terms of data paths to and from input/output buses and specific hardware registers. In some cases, data paths are provided to work areas for the microprograms and also to areas containing the operand fields of machine instructions.

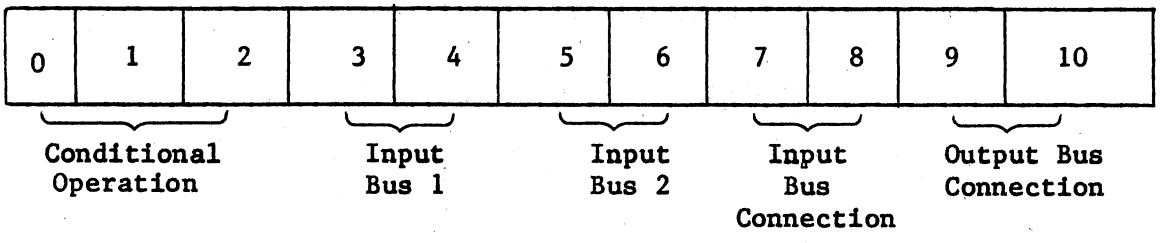


Figure 15. The Microinstructions for the Microprogrammable Pseudo-machine

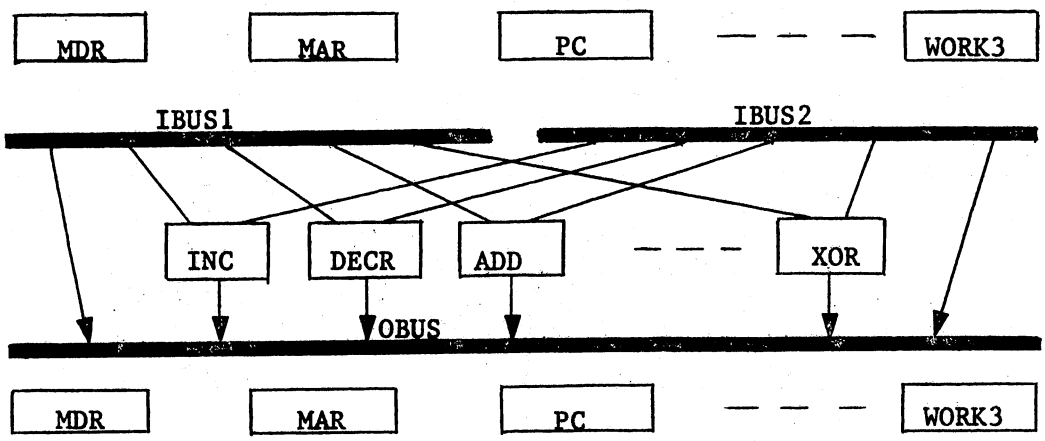


Figure 16. Register and Bus Configuration for the Microprogrammable Pseudo-machine

Positions 0-2 of the MIR are used to specify conditional operations. The contents of work area registers may be tested and program control determined from the results. Table V shows how the conditional operations may be specified.

If position 0 of the MIR is set to 1, positions 1 and 2 are decoded and the test operation performed. All other positions of the microinstruction are ignored. If the result of the test is "true", the remaining microinstructions are decoded and executed. If the result is "false", the remaining microinstructions are not executed and control is transferred to the next sequential machine instruction.

If position 0 of the MIR is set to 0, positions 3-10 are decoded as are the remaining microinstructions.

The entries in Table V are used to specify conditional operations for both GPR and FPR systems.

TABLE V

CONDITIONAL STATEMENTS FOR GENERAL AND FIXED PURPOSE REGISTER SYSTEMS

MIR 1 2	CONDITION
0 0	Filler
0 1	Compare T1 and T2
0 2	Compare T1 and T2 immediate
0 3	T1 = T2
0 4	T1 ≠ T2
0 5	T1 < T2
0 6	T1 > T2
0 7	T1 < = T2
0 8	T1 > = T2
0 9	T1 < 0
1 0	T1 > 0
1 1	T1 = 0
1 2	T1 = 1
1 3	T1 = -1
1 4	T1 even
1 5	T1 odd
1 6	T1 all 1's
1 7	T2 < 0
1 8	T2 > 0
1 9	T2 = 0
2 0	T2 = 1
2 1	T2 = -1
2 2	T2 even
2 3	T2 odd
2 4	T2 all 1'2

Table VI illustrates the input bus scheme for the microinstructions designed for GPR systems. Position 3 and 4 of the MIR specify the register or work area data path for input bus 1 and positions 5 and 6 specify the same for input bus 2.

Table VII shows that positions 3 and 4 of the MIR also serve to specify special I/O, debug and operate functions for GPR systems. When an interrupt is signalled by the user (26 in positions 3 and 4

of the MIR), positions 5-10 of the MIR are used to specify the type of interrupt that has occurred.

TABLE VI

INPUT BUS SCHEME FOR GENERAL PURPOSE REGISTER SYSTEMS

MIR 3 4	IBUS1 INPUTS	MIR 5 6	IBUS2 INPUTS
0 0	Filler	0 0	Filler
0 1	MDR	0 1	MDR
0 2	MAR	0 2	MAR
0 3	PC	0 3	PC
0 4	$\underline{e}$	0 4	e
0 5	$\overline{e}$	0 5	$\overline{e}$
0 6	R1	0 6	R1
0 7	R2	0 7	R2
0 8	B1 (explicit)	0 8	B1 (explicit)
0 9	B (implicit)	0 9	B (implicit)
1 0	D1	1 0	D1
1 1	D2	1 1	D2
1 2	S1	1 2	S1
1 3	S2	1 3	S2
1 4	X1	1 4	X1
1 5	I1	1 5	I1
1 6	WORK1	1 6	WORK1
1 7	WORK2	1 7	WORK2
1 8	WORK3	1 8	WORK3

Table VIII consists of the valid input bus connections. Some connections involve both input buses some involve only one. A list the functions used is given in Appendix A. Position 7 and 8 of the MIR specify the input bus connections for both GPR and FPR systems.

TABLE VII

SPECIAL I/O, DEBUG AND OPERATE FUNCTIONS FOR GENERAL PURPOSE REGISTER SYSTEMS

MIR 3 4	FUNCTION
1 9	Input data block (list directed)
2 0	Output data block (list directed)
2 1	Memory dump
2 2	Register trace "on"
2 3	Register trace "off"
2 4	Halt
2 5	No operation
2 6	Signal interrupt

TABLE VIII

INPUT BUS CONNECTIONS FOR GENERAL AND FIXED PURPOSE REGISTER SYSTEMS

MIR 7 8	CONNECTION
0 0	Filler
0 1	$\underline{\text{IBUS1}}$
0 2	$\overline{\text{IBUS1}}$
0 3	$\underline{\text{IBUS2}}$
0 4	$\overline{\text{IBUS2}}$
0 5	INC (IBUS1)
0 6	INC (IBUS2)
0 7	DECR (IBUS1)
0 8	DECR (IBUS2)
0 9	ADD (IBUS1, IBUS2)
1 0	SUB (IBUS1, IBUS2)
1 1	MUL (IBUS1, IBUS2)
1 2	DIV (IBUS1, IBUS2)
1 3	MOD (IBUS1, IBUS2)
1 4	ABS (IBUS1)
1 5	ABS (IBUS2)
1 6	NEG (IBUS1)
1 7	NEG (IBUS2)
1 8	AND (IBUS1, IBUS2)
1 9	OR (IBUS1, IBUS2)
2 0	XOR (IBUS1, IBUS2)
2 1	MDR ← M(MDR)
2 2	M(MAR) ← MDR

Table IX contains the valid output bus connections for GPR systems. Once the input bus connection has been made, the data is transferred to the output bus and from there to an appropriate register. Positions 9 and 10 of the MIR delegate to what register or work area the output bus connection is made.

TABLE IX

OUTPUT BUS CONNECTIONS FOR GENERAL PURPOSE REGISTER SYSTEMS

MIR 9	MIR 10	CONNECTION
0	0	Filler
0	1	MDR ← OBUS
0	2	MAR ← OBUS
0	3	PC ← OBUS
0	4	R1 ← OBUS
0	5	R2 ← OBUS
0	6	T1 ← OBUS
0	7	T2 ← OBUS
0	8	WORK1 ← OBUS
0	9	WORK2 ← OBUS
1	0	WORK3 ← OBUS
1	1	B1 ← OBUS
1	2	B ← OBUS
1	3	X1 ← OBUS
1	4	(R1 + 1) ← OBUS
1	5	(R2 + 1) ← OBUS
1	6	R1 ← K1 ↑ OBUS
1	7	R1 ← K1 ↓ OBUS
1	8	R1 ← K1 ↕ OBUS
1	9	R1 ← K1 ⇳ OBUS

The input bus scheme for FPR systems is shown in Table X. Notice that some of the same registers are used for FPR systems as GPR systems. Such operations as branching and data transfers are common to

both systems. Positions 3 and 4 of the MIR specify the register or work area data path for input bus 1 and positions 5 and 6 specify the same for input bus 2.

TABLE X

INPUT BUS SCHEME FOR FIXED PURPOSE REGISTER SYSTEMS

MIR 3 4	IBUS1 INPUTS	MIR 5 6	IBUS2 INPUTS
0 0	Filler	0 0	Filler
0 1	MDR	0 1	MDR
0 2	MAR	0 2	MAR
0 3	PC	0 3	PC
0 4	e	0 4	e
0 5	$\overline{e}$	0 5	$\overline{e}$
0 6	1	0 6	1
0 7	ACC A	0 7	ACC A
0 8	ACC B	0 8	ACC B
0 9	XR1	0 9	XR1
1 0	XR2	1 0	XR2
1 1	I1	1 1	I1
1 2	S1	1 2	S1
1 3	L	1 3	L
1 4	WORK1	1 4	WORK1
1 5	WORK2	1 5	WORK2
1 6	WORK3	1 6	WORK3

As with the GPR systems, Table XI shows that positions 3 and 4 of the MIR also serve to specify special I/O, debug and operate functions for FPR systems. When an interrupt is signalled by the user (24 in positions 3 and 4 of the MIR), positions 5-10 of the MIR are used to specify the type of interrupt that has occurred.

Table XII contains the valid output bus connections for FPR sys-

tems. Once the input bus connection has been made, the data is transferred to the output bus and from there to the appropriate register. Positions 9 and 10 of the MIR delegate to what register or work area the output bus connection is made.

TABLE XI

SPECIAL I/O DEBUG AND OPERATE FUNCTIONS FOR FIXED PURPOSE REGISTER SYSTEMS

MIR 3 4	FUNCTION
1 7	Input data block (list directed)
1 8	Output data block (list directed)
1 9	Memory dump
2 0	Register trace "on"
2 1	Register trace "off"
2 2	Halt
2 3	No operation
2 4	Signal interrupt

Table XIII illustrates the microinstructions necessary to add the the contents of two general purpose registers (of a GPR system) and branch if the contents of the destination register is zero. For this operation, instruction format 10 (Figure 13) must be specified. Table XIV illustrates the microinstructions necessary to load the contents of index register 1 into the accumulator (of an FPR system) and branch if the contents of the accumulator is negative. Instruction format 1 (Figure 14) must be specified for this operation.

TABLE XII

OUTPUT BUS CONNECTIONS FIXED PURPOSE FOR REGISTER SYSTEMS

MIR 9	MIR 10	CONNECTION
0	0	Filler
0	1	MDR ← OBUS
0	2	MAR ← OBUS
0	3	PC ← OBUS
0	4	XR1 ← OBUS
0	5	XR2 ← OBUS
0	6	T1 ← OBUS
0	7	T2 ← OBUS
0	8	WORK1 ← OBUS
0	9	WORK2 ← OBUS
1	0	WORK3 ← OBUS
1	1	ACC A ← OBUS
1	2	ACC B ← OBUS
1	3	L ← OBUS
1	4	K1 ← OBUS
1	5	ACC A ← K1 ↑ OBUS
1	6	ACC A ← K1 ↓ OBUS
1	7	ACC A ← K1 ↟ OBUS
1	8	ACC A ← K1 ↡ OBUS
1	9	ACC B ← K1 ↑ OBUS
2	0	ACC B ← K1 ↓ OBUS
2	1	ACC B ← K1 ↟ OBUS
2	2	ACC B ← K1 ↡ OBUS

TABLE XIII

IMPLEMENTATION OF ADD REGISTER TO REGISTER AND BRANCH ON ZERO

INSTRUCTION NUMBER	MIR	DESCRIPTION
1.	00006070905	R1 ← R1 + R2
2.	00007000106	T1 ← R1
3.	11100000000	If T1 = 0 then go to next; else decode next machine instruction
4.	00012000103	next: PC ← S1

TABLE XIV

IMPLEMENTATION OF LOAD ACCUMULATOR FROM INDEX REGISTER AND BRANCH ON NEGATIVE

INSTRUCTION NUMBER	MIR	DESCRIPTION
1.	00009000111	ACC A ← XR1
2.	00007000106	T1 ← ACC A
3.	10900000000	If T1 < 0 then go to next; else decode next machine instruction
4.	00012000103	next: PC ← S1

Error Detection

Errors occurring during execution time are detected by the interpreter and appropriate messages are printed. Errors such as addressing and operation exceptions are detected during normal execution. In all cases the occurence of an execution time error causes termination of the program. This is due to the fact that there are no error correction capabilities built into the simulation system. The cause of the error must be determined and corrected and the program resubmitted for execution.

Debugging Aids

Instructions used as tools for debugging may be defined with the micro-operations provided. Two such instructions are the instruction trace and memory dump operations. The instruction trace may be turned "on" and "off" as desired and causes the contents of all hardware registers to be printed (in decimal) after the execution of any subse-

quent instructions. The dump instruction provides a means of determining the contents of memory within a certain region. Pre-execution and post-execution memory dumps are provided without specification.

Other Simulation Systems

Other types of hardware simulation languages have been developed. Two of these are APL (the interactive terminal languate) (3) and PMSL (12). Both APL and PMSL require strict knowledge of the simulated computer system but treat the simulation process on different levels.

APL makes use of a special character set, which includes both an upper and lower case alphabet, to achieve parallelism with AHPL (A Hardware Programming Language), the hardware "description" language. At the APL level of simulation, register transfer, memory access, I/O buffering, etc. can be specified in detail not unlike that of the simulated system. Thus, detailed knowledge of the interaction of registers, memory and peripherals is necessary.

PMSL, as designed by Knudsen (12), is a conversational facility for the creation, modification, storage, retrieval and analysis of descriptions of computer hardware at the top system level, where work is performed on processors, memories, controllers, channels and peripherals. PMSL provides a powerful tool to the design engineers of computer systems. Performance, cost and device utilization can be monitored to give the designer a look at critical aspects of new system designs. The language is based on the PMS (Processors, Memories and Switches) notations in Bell and Newell (2).

CHAPTER V

USERS MANUAL

This chapter describes how to use the simulation system. The machine description phase, assembly language statements and options, deck setup and output, and control cards are discussed. At certain points, the current restrictions of the simulation system are also discussed.

Initial Machine Description

The first phase of the simulation process is the machine description. The user is required to supply necessary information about the simulated machine in the form of input cards. The details of the card formats for the input information are given in Tables XV and XVI. Information about the hardware of the machine such as the types of registers used, the size of the registers, and the memory word size must be specified. Other details about the instruction set and formats must also be specified. If at any time a specification is requested that does not apply to the particular machine, a negative or zero value should be placed in the corresponding field of the input card. The input formats for the machine definition must be followed exactly to ensure correct results in later phases of simulation. Note that all character and bit values must be left justified in the appropriate fields and all numeric values must be right justified in the

appropriate fields.

TABLE XV

INPUT CARD FORMATS FOR MACHINE DESCRIPTION OF GENERAL PURPOSE REGISTER SYSTEMS

CARD SET	CARD COLUMNS	FIELD TYPE	DESCRIPTION
1	1	Bit	Specify 1 if output of machine description is desired Specify 0 for no output
	3-5	Char.	Machine type, GPR
	7-11	Numeric	Memory word size
	13	Bit	Specify 1 if output of assembly debug statements is desired Specify 0 for no output (Should normally be set to 0)
	15-19	Numeric	Number of operations to be defined
	21-25	Numeric	Bit length of the binary op code
	27-56	Char.	Machine name
2	1-5	Numeric	Number of general purpose registers (max. of 16)
	7-11	Numeric	GPR size
	13-17	Numeric	MDR size
	19-23	Numeric	MAR size
	25-29	Numeric	PC size
	31-35	Numeric	Number of words in the IR (max. of 2)
	37-41	Numeric	Total IR size
	43-47	Numeric	Implicit base register
	49-53	Numeric	Number of instruction formats to be used
	55-59	Numeric	Number of words read in a READ instruction
	61-65	Numeric	Number of words written in a WRITE instruction
3	1-5	Numeric	Instruction format number
	7-11	Numeric	Number of words in the instruction
	13-17	Numeric	Field 1 starting position
	19-23	Numeric	Field 1 ending position
	25-29	Numeric	Field 2 starting position
	31-35	Numeric	Field 2 ending position
	37-41	Numeric	Field 3 starting position
	43-47	Numeric	Field 3 ending position
	49-53	Numeric	Field 4 starting position
	55-59	Numeric	Field 4 ending position (Repeat above 10 fields for each instruction format used)

TABLE XV (CONTINUED)

CARD SET	CARD COLUMNS	FIELD TYPE	DESCRIPTION
4	1-...	Bit	Binary operation code (max. length of 32)
	34-38	Char.	Mnemonic operation code
	40-44	Numeric	Number of machine cycles required
	46-50	Numeric	Instruction format number
	52-56	Numeric	Number of microinstructions that define the machine instruction (max of 20).
	1-11	Numeric	Microinstruction
	13-23	Numeric	Microinstruction
	25-35	Numeric	Microinstruction
	37-47	Numeric	Microinstruction
	59-59	Numeric	Microinstruction
	61-71	Numeric	Microinstruction
			(Repeat above 6 fields until micro-definition is complete (max. of 20) and repeat above 11 fields for each machine instruction)

TABLE XVI

INPUT CARD FORMATS FOR MACHINE DESCRIPTION OF FIXED PURPOSE REGISTER SYSTEMS

CARD SET	CARD COLUMNS	FIELD TYPE	DESCRIPTION
1	1	Bit	Specify 1 if output of machine description is desired Specify 0 for no output
	3-5	Char.	Machine type, FPR
	7-11	Numeric	Memory word size
	13	Bit	Specify 1 if output of assembly debug statements is desired Specify 0 for no output (Should normally be set to 0)
	15-19	Numeric	Number of non-operate operation codes
	21-25	Numeric	Number of operate operation codes
	27-31	Numeric	Length of binary operation codes
	33-37	Numeric	Number of bits in the mode specification

TABLE XVI (CONTINUED)

CARD SET	CARD COLUMNS	FIELD TYPE	DESCRIPTION
	39-43	Numeric	Number of words in a memory page
	45-74	Char.	Machine name
2	1-5	Numeric	Number of index registers (1 or 2)
	7-11	Numeric	Index register size
	13-17	Numeric	MDR size
	19-23	Numeric	MAR size
	25-29	Numeric	PC size
	31-35	Numeric	Number of words in the IR (max. of 2)
	37-41	Numeric	Total IR size
	43-47	Numeric	Accumulator A size
	49-53	Numeric	Accumulator B size
	55-59	Numeric	Number of instruction formats to be used
	61-65	Numeric	Number of words read in a READ instruction
	67-71	Numeric	Number of words written in a WRITE instruction
3	1-5	Numeric	Instruction format number
	7-11	Numeric	Number of words in the instruction
	13-17	Numeric	Field 1 starting position
	19-23	Numeric	Field 1 ending position
	25-29	Numeric	Field 2 starting position
	31-35	Numeric	Field 2 ending position
	37-41	Numeric	Field 3 starting position
	43-47	Numeric	Field 3 ending position
4			(Input is the same as shown in Table XV, card set 4, for GPR machines)
5	1-5	Char.	Op code field mnemonic for operate functions
	7-...	Bit	Binary op code for operate functions (max. length of 32)
6	1-5	Numeric	Number of positions in the instruction that are set
	7-...	Numeric	Bit positions in the instruction that are set for the operate function (Must be separated by one blank column using a two column field for the position numbers)
7	1-5	Char.	Operand field mnemonic for the operate function
	7-11	Numeric	Number of machine cycles required
	13-17	Numeric	Number of microinstructions that define the machine instruction
8			(Input for the microinstructions is the same as shown in Table XV, card set 4, for GPR machines)

TABLE XVI (CONTINUED)

CARD SET	CARD COLUMNS	FIELD TYPE	DESCRIPTION
9	1-...	Bit	Binary code for each mode, separated by one blank column

The Cross Assembly Language

This section describes the format of the statements for the cross assembler. The format is not unlike those used by many assemblers, but it does have its unique characteristics.

Symbols

All symbols and identifiers must begin with an alphabetic character. The remaining characters may be alphabetic or numeric. The alphabet in this case consists of all letters in the Roman alphabet plus the special characters $, @, #, &, and %. Label identifiers must begin in column 1 of an input card and may be no longer than eight characters. The mnemonic op code symbols must begin in column 10 and may be no longer than five characters.

Addressing Specifications

For fixed purpose register systems, column 16 of the assembler statement is used as an address specification field. This field is used to specify direct or indirect addressing, or index registers. For indirect addressing an "I" is used. To specify the use of index register 1, a "1" or "A" is used. To specify the use of index register

2, a "2" or "B" is used. Direct addressing is specified by leaving the address specification field blank. The addressing modes used by the simulation system for FPR machines are described in Table XVII.

TABLE XVII

ADDRESSING MODES FOR FIXED PURPOSE REGISTER SYSTEMS

TYPE	MODE
1	Direct to current sector
2	Direct through zero sector
3	Indirect through current sector
4	Indirect through zero sector
5	Indexed, register 1
6	Indexed, register 2

Operands

The number of operands in the operand field may vary depending on the type of machine being simulated. For GPR systems, the maximum number of operands is three. For FPR systems, the maximum number of operands is five (for register operate functions). In either case the operands are separated by commas and there can be no imbedded blanks in the field. Also, each assembler statement must be contained in its entirety on one input card. Assembler statements may not be continued across card boundaries.

Comments may follow the operand field if a blank column separates the two fields. A comment can also be specified by an asterisk (*) in

column one. In this case the entire card is treated as a comment. An identification or sequence value may appear in columns 73-80. A summary of the assembler statement field boundaries appears in Tables XVIII and XIX.

TABLE XVIII

FIELD BOUNDARIES FOR ASSEMBLER LANGUAGE STATEMENTS OF GENERAL PURPOSE REGISTER SYSTEMS

CARD COLUMNS	DESCRIPTION
1 - 8	Label field
10 - 14	Mnemonic op code
20 - 39	Operand field
41 - 72	Comments
73 - 80	Identification sequence

TABLE XIX

FIELD BOUNDARIES FOR ASSEMBLER LANGUAGE STATEMENTS OF FIXED PURPOSE REGISTER SYSTEMS

CARD COLUMNS	DESCRIPTION
1 - 8	Label field
10 - 14	Mnemonic op code
16	Addressing specification
20 - 39	Operand field
41 - 72	Comments
73 - 80	Identification sequence

Pseudo Operations

There are five pseudo operation codes that the user need not define: (1) EQU, (2) ORG, (3) DC, (4) DS, and (5) END. The EQU pseudo op assigns the absolute or resolved value of the symbolic address in the operand field to the identifier in the label field. Note that any symbolic references in the operand field must be defined before the occurrence of the EQU statement. The ORG pseudo op reinitializes the PC to the positive decimal value in the operand field. Table XX illustrates the valid operand specifications for the DC and DS instructions as well as for other instructions. The END pseudo op designates the end of the assembly process and may have a label identifier in the operand field.

TABLE XX

ASSEMBLER LANGUAGE OPERAND SPECIFICATIONS

	OPERAND DESCRIPTION	
DC	B' positive or negative binary value'	
	F' positive or negative decimal value'	
	X' positive or negative hexadecimal value'	
	A (symbolic or absolute address)	
DS	positive decimal value	B'positive or negative binary value'
		F'positive or negative decimal value'
		X'positive or negative hexadecimal value'
		A(symbolic or absolute address)
	positive decimal value	

TABLE XX (CONTINUED)

OPERAND DESCRIPTION	
IMMEDIATE SPECIFICATION	Same operand specification as for DC instruction preceded by an equal sign (=)
REGISTER	Positive decimal value (register number) or symbol equated to register number
SYMBOLIC ADDRESS	Symbol Symbol + positive decimal value Symbol - positive decimal value * (reference to current PC value) * + positive decimal value * - positive decimal value Positive decimal value (absolute address)

Assembly Error Messages

The occurrence of any assembly time errors causes execution of the object program to be suspended. To aid in the elimination of assembly errors, descriptive error messages are printed immediately following the statement in error. A listing of these error messages is given in Table XXI.

Control Cards and JCL

Each program begins with a >> JOB card. Other control cards may be specified in order to restrict the execution of the object module.

The >>TIME card specifies the maximum number of machine cycles allowed for execution before termination of the program. The >> REGION card specifies the size of the memory region (in words) in which the object program is to be loaded. Column 12 of the >> EXEC card must contain the character 'E' or 'A'. With option 'E', the user program is assembled and if no assembly errors are detected, the object module is loaded and executed. If option 'A' is specified, the user program is assembled but not executed. A sample program setup is given in Figure 17.

TABLE XXI

ASSEMBLY ERROR MESSAGES

NO.	MESSAGE
1.	Invalid character in label field
2.	Imbedded blanks in label field
3.	Label doesn't begin with an alphabetic character
4.	Previously defined identifier in label field
5.	Invalid or missing data type on DS statement
6.	Invalid or missing data type on DC statement
7.	Invalid or missing operation code
8.	Invalid addressing option specified
9.	Negative address specified
10.	Invalid operand specified
11.	Operand missing
12.	Undefined operand specified
13.	One or more undefined operate functions in operand field
14.	Maximum number of operate functions has been exceeded
15.	Missing END card, one has been generated for assembly
16.	Label appears on ORG or END statement
17.	Storage allocation on DS statement exceeds region size
18.	Operand on ORG statement causes the destruction of previously assembled program segment
19.	Operand on ORG statement extends beyond region size
20.	Operand and instruction type do not match
21.	Region size too small to assemble the following portion of the program
22.	Write-protect violated, operand specifies page zero address

```
>>JOB      name1
>>TIME=550
>>REGION=200
>>EXEC ASM E
>>ASM. PROG
        User program
>>ASM. DATA
        User program data

>>
>>JOB      name2
    .
    .
    .
>>
    .
    .
    .
```

Figure 17. Sample Program Setup

Deck Setup

The deck setup consists of the simulation system, which operates on one or more user programs, the machine description, the user programs, and a file used by the assembler when generating intermediate object code. The deck setup to use the implemented simulation system is shown in Figure 18.

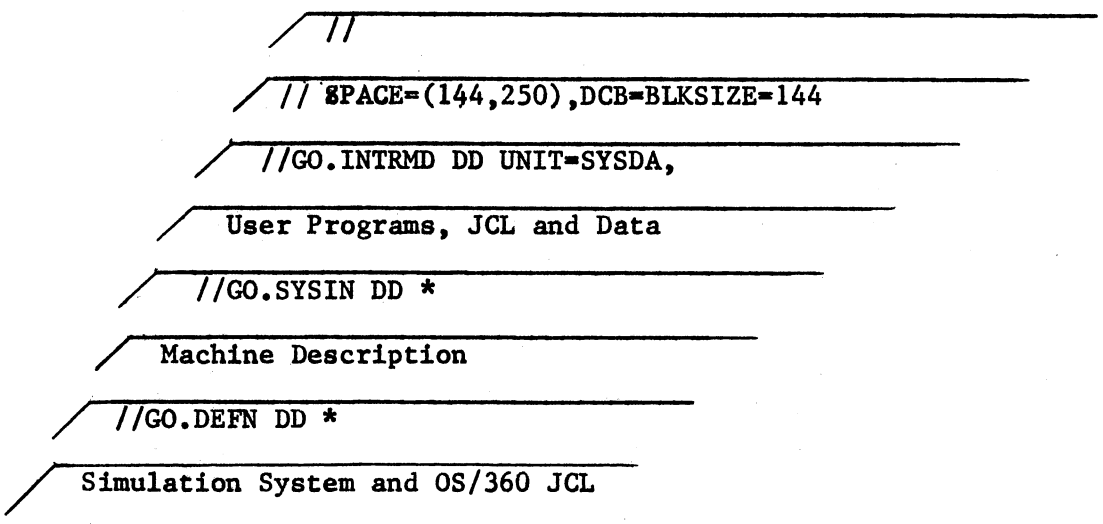


Figure 18. Deck Setup

CHAPTER VI

SUMMARY AND FURTHER STUDY

Using the methods discussed in this thesis, a microprogrammed simulation system has been implemented in PL/I on the IBM System 360/65. The system supports the simulation of general purpose and fixed purpose register minicomputers.

The most significant input to the system is in the form of cross assembly language programs. There are two options available to the user when assembling a program. Using the JCL and control statements of the simulation system, the user may specify assembly of the program and execution of the load module, or only assembly of the program. Other control statements are available to restrict the execution of the load module.

The simulation system makes possible the simulation of a large percentage of the instructions for general and fixed purpose register minicomputers. Specification of the input/output instructions contains the least flexibility. Input from the card reader (list directed) and output to the line printer (list directed) is the only type of I/O supported. More extensive work in this area would make possible the simulation of the input/output instructions and the interrupt structure for the types of minicomputers discussed. Also, the microinstruction set could be extended to support "special" instructions unique to specific machines.

Two types of special instructions that are common on most modern minicomputers are circular list processing instructions and hardware stack instructions. Usually, hardware pointers are involved in both types of instructions. For list processing, pointers to the top and bottom of the list must be kept so that overlapping of the elements in the list can be detected. For stack processing, pointers to the top and bottom of the stack are also kept. But, in this case, data items are inserted and deleted only from the top of the stack; only one pointer is usually updated. This allows overflow and underflow conditions to be detected.

In order for the stack and list processing instructions to be implemented using microinstructions, either specific hardware registers or locations in memory must be used for the necessary pointers. Once these registers or locations are determined, the microinstructions for updating the pointers can easily be constructed. Testing for overflow and underflow of the stack and overlapping of the list can be performed with the micro-operations currently implemented.

The cross assembler is an important tool for testing the simulated machine instructions. But it is necessary for the user to translate the assembler programs written for the simulated machine to the cross assembly language for testing. The simulation system could be further generalized to allow a description of the assembler language syntax for the simulated machine to be input. This would make possible the direct assembly of the programs written for the simulated machine.

A SELECTED BIBLIOGRAPHY

(1) A Pocket Guide to Hewlett-Packard Computers. Palo Alto, California: Hewlett-Packard Corporation.

(2) Bell, C. G. and A. Newell. Computer Structures: Readings and Examples. New York: McGraw-Hill, 1971.

(3) Bingham, H. W. "Use of APL in Microprogrammable Machine Modeling." Special Interest Group on Programming Languages. 6, 9 (October, 1971), 105-109.

(4) Fuller, S. H., V. R. Lesser, C. G. Bell and C. Kaman. Microprogramming and Its Relationship to Emulation and Technology. Pittsburg, Pennsylvania: Carnegie-Melon University, 1974.

(5) Gries, D. Compiler Construction for Digital Computers. New York: John Wiley and Sons, 1971.

(6) Hedrick, G. E. An Introduction to Basic Concepts in Computer Science: An Approach Based on Behavioral Objectives. Stillwater, Oklahoma: Oklahoma State University, 1972.

(7) Hill, F. J. and G. R. Peterson. Digital Systems: Hardware Organization and Design. New York: John Wiley and Sons, 1973.

(8) Husson, S. S. Microprogramming Principles and Practices. New York: Prentice-Hall, 1970.

(9) Interdata Model 1. Palo Alto, California: Interdata Corporation, 1970.

(10) Interdata Reference Manual. Oceanport, New Jersey: Interdata Corporation, 1974.

(11) Jones, L. H. "Survey of Current Work in Microprogramming." Computer, 8, 8 (August, 1975), 33-38.

(12) Knudsen, J. J.. PMSL, An Interactive Language for System-Level Description and Analysis of Computer Structures. Pittsburg, Pennsylvania: Carnegie-Melon University, 1974.

(13) Mallach, E. G. "Emulator Architecture." Computer, 8, (August, 1975), 24-32.

(14) Microprogramming Handbook. Santa Ana, California: Microdata Corporation, 1971.

(15) Modular Computer Systems Assembler Reference Manual. Fort Lauderdale, Florida: Modular Computer Systems Corporation, 1973.

(16) Opler, A. "Fourth Generation Software." Datamation, 13, 1 (1967), 22-24.

(17) Pandit, A. V. "Simulation and APL Description of the PDP 11/40." (Unpublished Master of Science thesis, Oklahoma State University, Stillwater, Oklahoma, 1975.)

(18) Rosin, R. F. "Contemporary Concepts of Microprogramming and Emulation." Computing Surveys, 1, 4 (December, 1969), 197-212.

(19) Siewiorek, D. P. and M. R. Barbacci. Some Observations on Modular Design Technology and the Use of Microprogramming. Pittsburg, Pennsylvania: Carnegie-Melon University, 1974.

(20) Varian 520/i. Irvine, California: Varian Data Machines, 1969.

(21) Wilkes, M. V. "The Growth of Interest in Microprogramming-A Literature Survey." Computing Surveys, 1, 3 (September, 1969), 139-145.

(22) Wilkes, M. V. "The Best Way to Design an Automatic Calculating Machine." Manchester, England: Manchester University Computer Inaugural Conference, 1951, 16-21.

APPENDICES

APPENDIX A

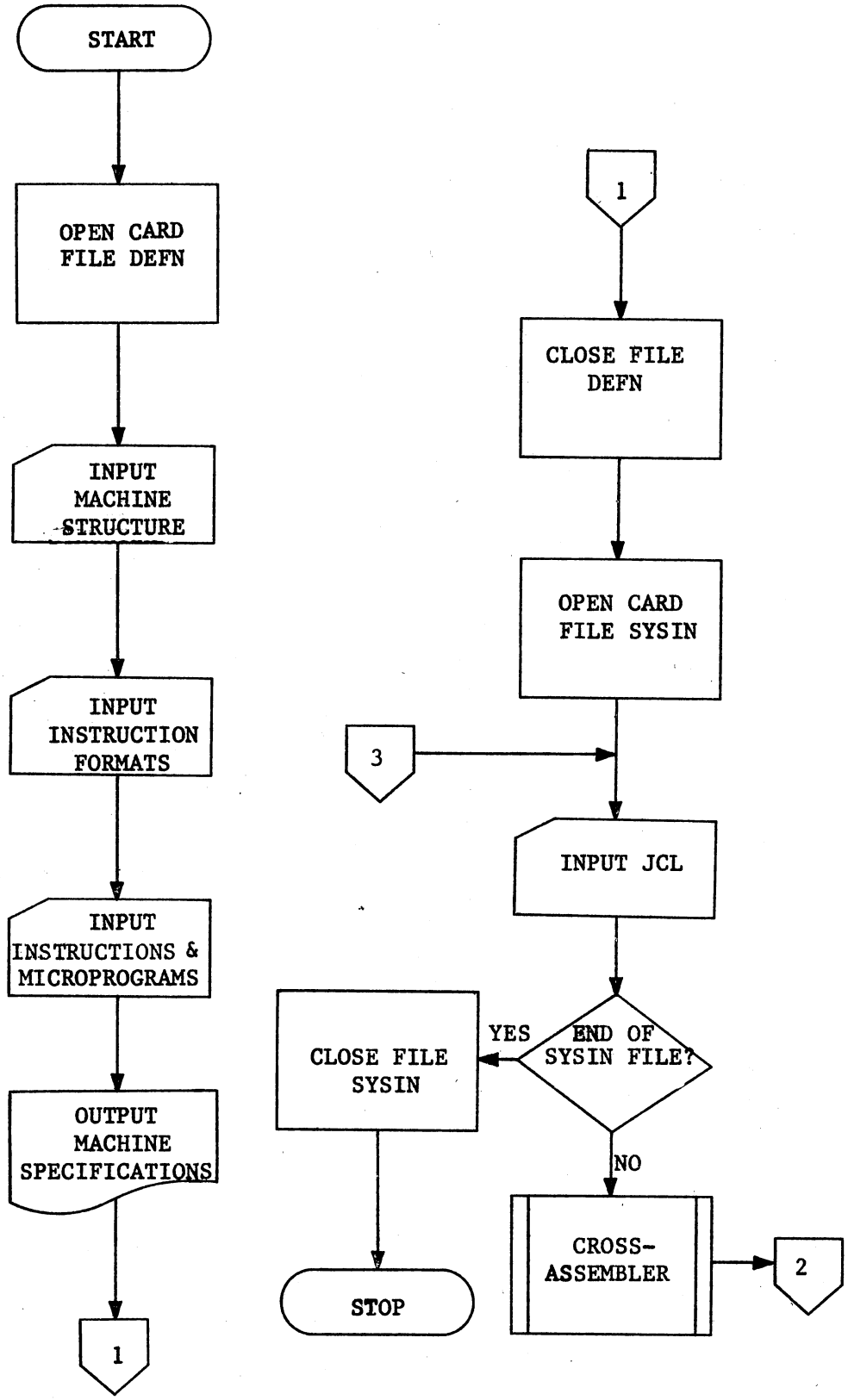
LIST OF SYMBOLS

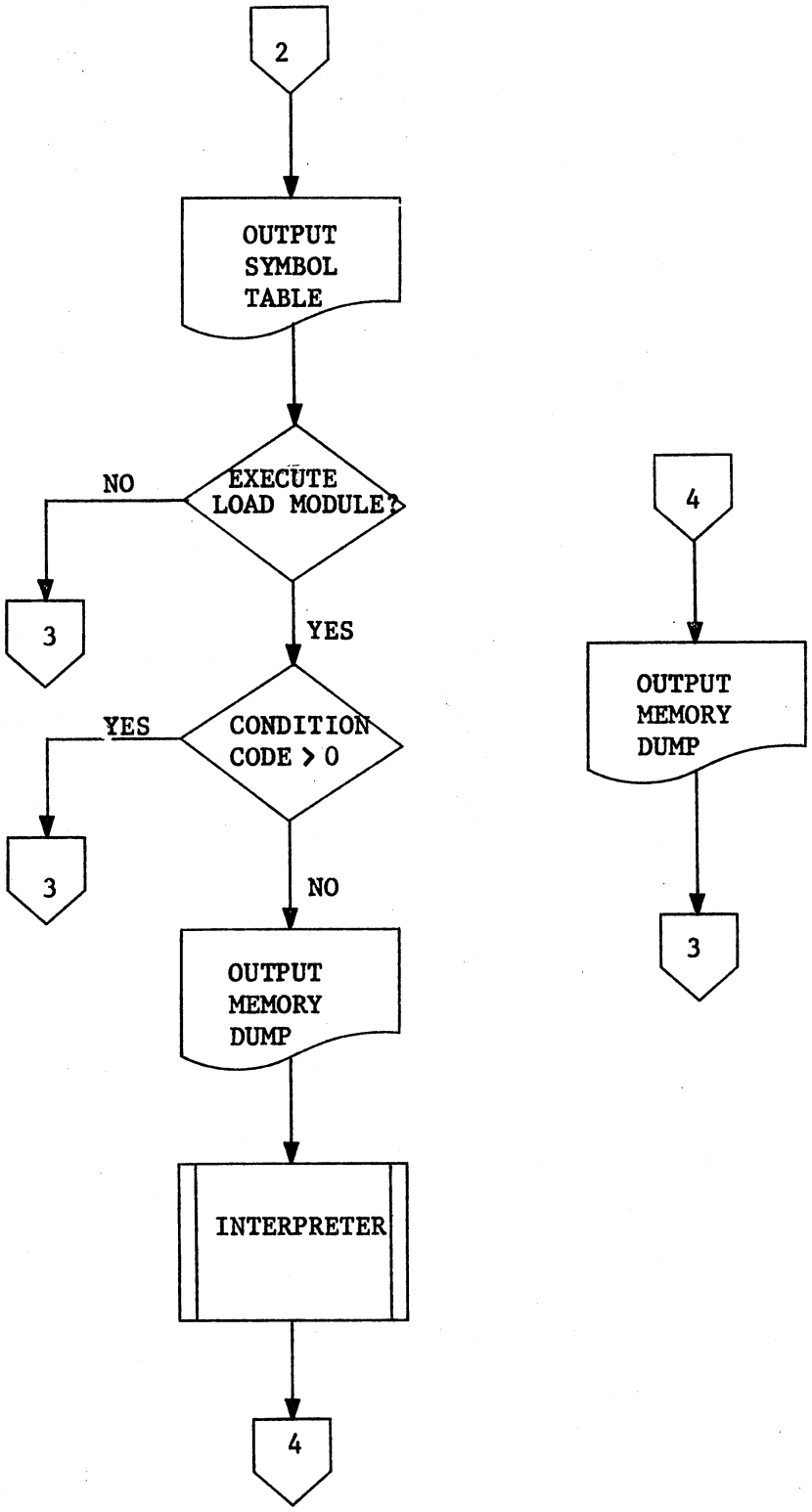
SYMBOL	MEANING
ABS	absolute value function
ACC A	accumulator A
ACC B	accumulator B
ADD	integer addition of two operands
ALU	arithmetic logic unit
AND	logical "AND" of two operands
B	implicit base register (0-15)
B1	explicit base register (0-15)
CPU	central processing unit
DECR	decrement value by one
DIV	integer division of two operands
D1, D2	positive or negative displacements
e	a binary value of all 1's
FILLER	no micro-operation specified
FPR	fixed purpose register
GPR	general purpose register
IBUS1	input bus 1
IBUS2	input bus 2
INC	increment value by one
IR	instruction register
I1	"immediate" operand
K1	shift or rotation count
L	link bit for accumulators
M	simulated memory
MAR	memory address register
MDR	memory data (buffer) register
MIR	microinstruction register
MOD	residue modulo
MUL	integer multiplication of two operands
NEG	negation
OBUS	output bus
OR	logical "OR" of two operands
PC	program counter
PSW	program status word
ROM	read-only memory
ROS	read-only store
R1, R2	general purpose registers
SIC	small instructional computer
SUB	integer subtraction of two operands
S1, S2	resolved symbolic references
T1, T2	registers used for comparisons and testing
WORK1	microprogram work area register 1
WORK2	microprogram work area register 2
WORK3	microprogram work area register 3
XOR	logical "EXCLUSIVE OR" of two operands
XR1, XR2	index register 1, index register 2
X1	index register (0-15)
λ	null pointer
$\leftarrow$	assignment

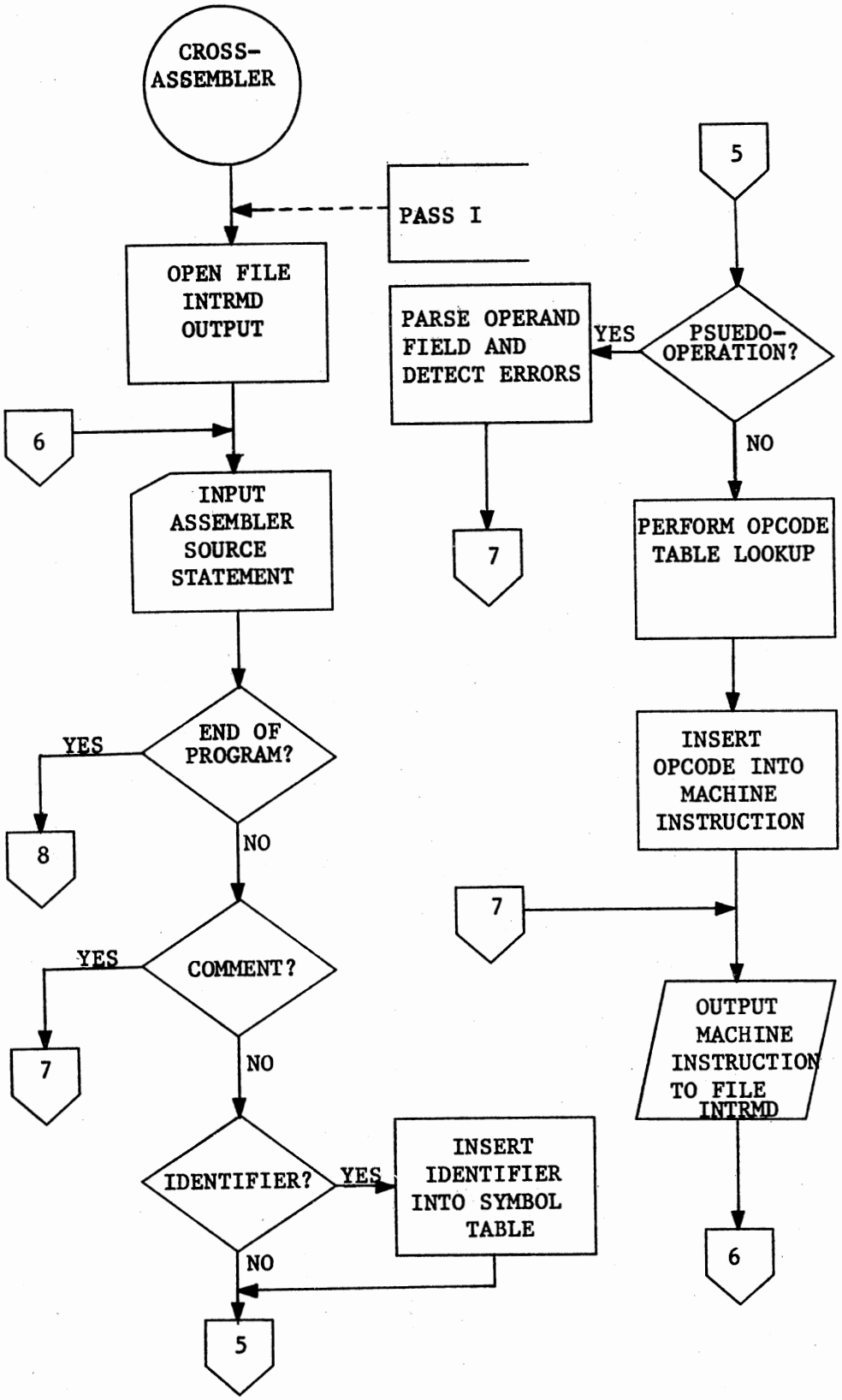
SYMBOL	MEANING
↑	left rotation
↓	right rotation
↟	left shift
↡	right shift

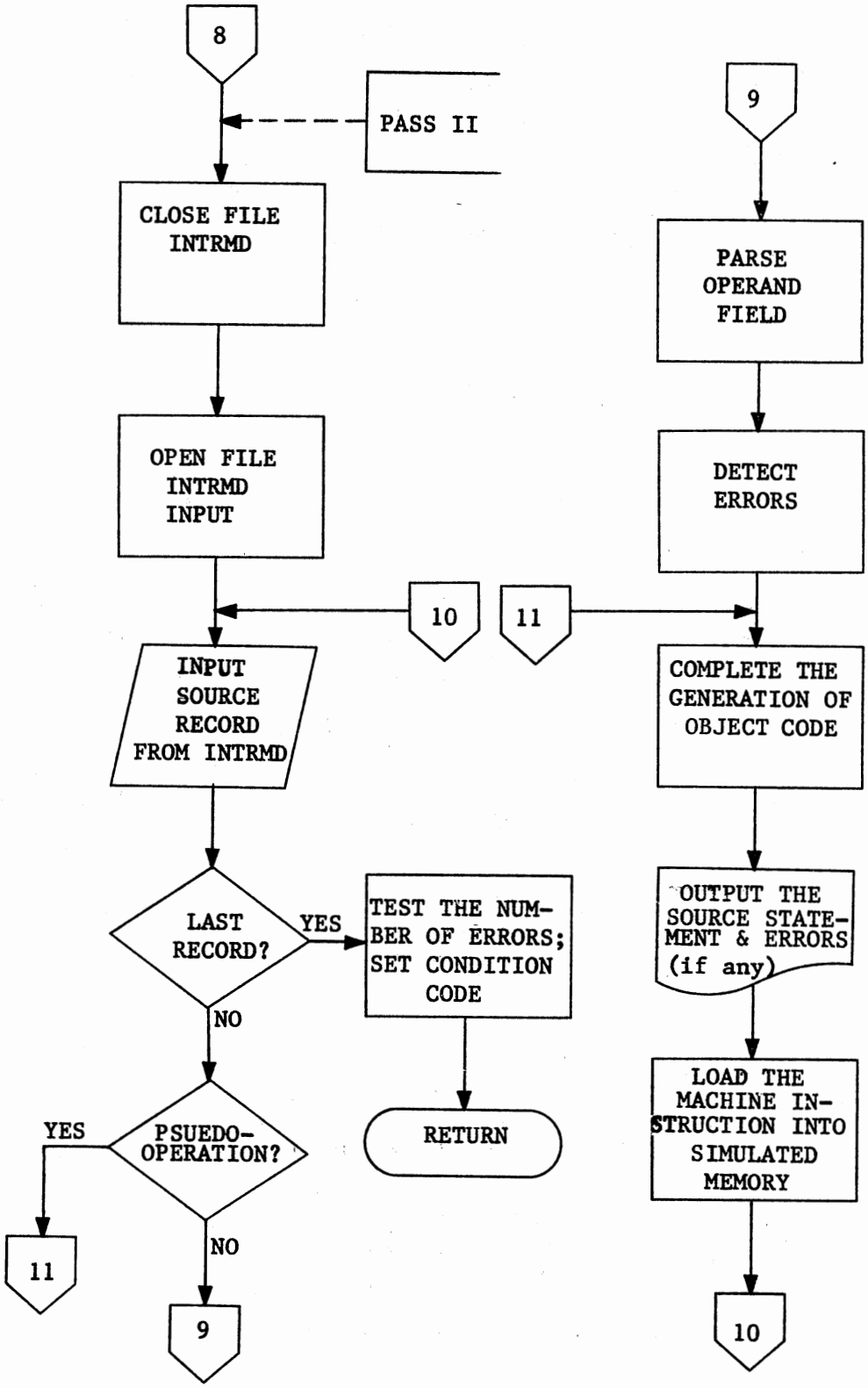
APPENDIX B

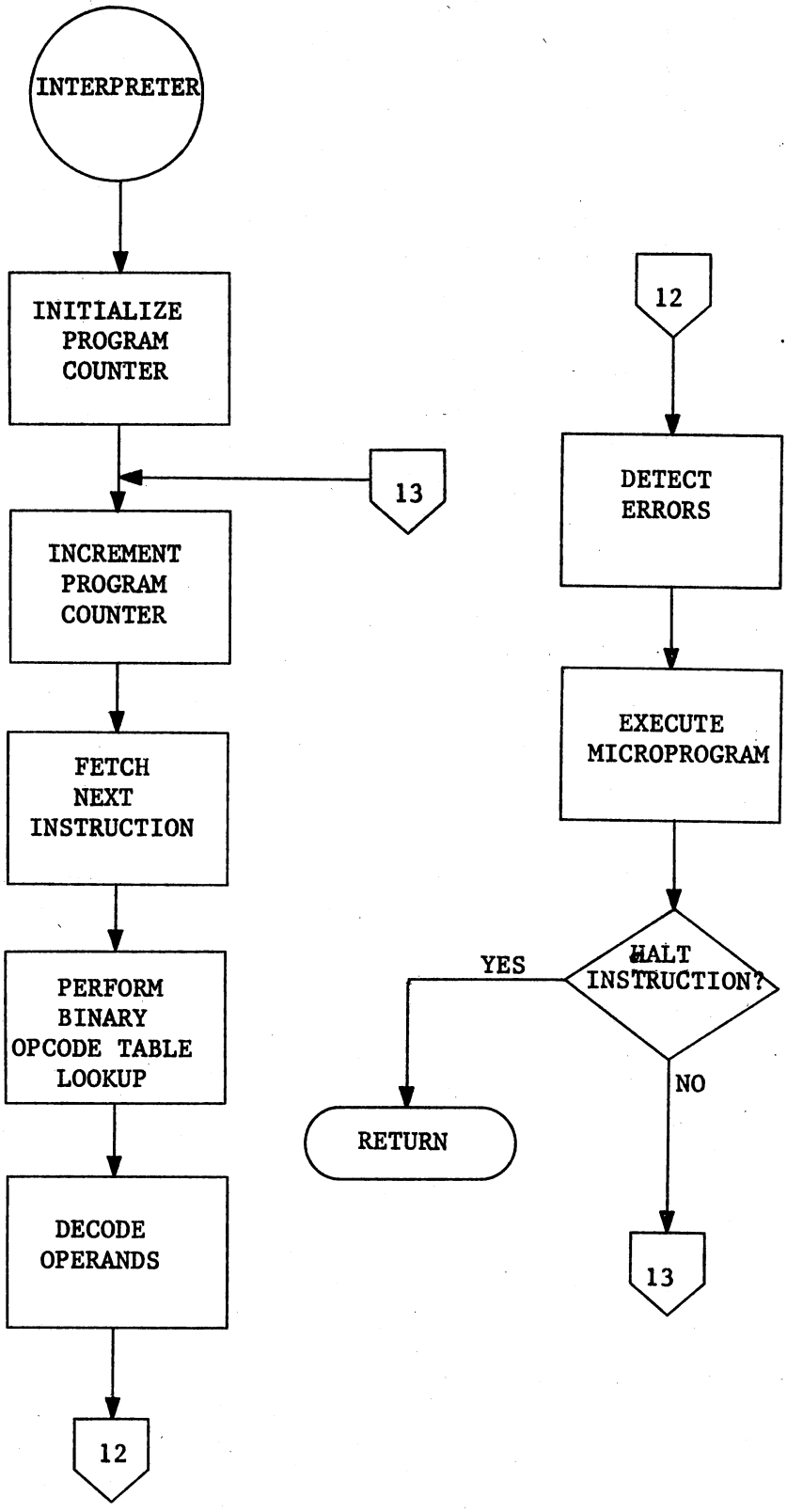
LOGIC BLOCK DIAGRAM


START
OPEN CARD FILE DEFN
INPUT MACHINE STRUCTURE
INPUT INSTRUCTION FORMATS
INPUT INSTRUCTIONS & MICROPROGRAMS
OUTPUT MACHINE SPECIFICATIONS
1
1
CLOSE FILE DEFN
OPEN CARD FILE SYSIN
3
INPUT JCL
END OF SYSIN FILE?
YES
CLOSE FILE SYSIN
NO
CROSS-ASSEMBLER
2
STOP


2
OUTPUT SYMBOL TABLE
EXECUTE LOAD MODULE?
NO
3
YES
CONDITION CODE > 0
YES
3
NO
OUTPUT MEMORY DUMP
INTERPRETER
4
4
OUTPUT MEMORY DUMP
3


CROSS-ASSEMBLER
PASS I
OPEN FILE INTRMD OUTPUT
6
INPUT ASSEMBLER SOURCE STATEMENT
END OF PROGRAM?
YES
8
NO
COMMENT?
YES
7
NO
IDENTIFIER?
YES
INSERT IDENTIFIER INTO SYMBOL TABLE
NO
5
7
PARSE OPERAND FIELD AND DETECT ERRORS
5
PSUEDO-OPERATION?
YES
NO
PERFORM OPCODE TABLE LOOKUP
INSERT OPCODE INTO MACHINE INSTRUCTION
7
OUTPUT MACHINE INSTRUCTION TO FILE INTRMD
6


8
PASS II
CLOSE FILE INTRMD
OPEN FILE INTRMD INPUT
10
INPUT SOURCE RECORD FROM INTRMD
LAST RECORD?
YES
TEST THE NUMBER OF ERRORS; SET CONDITION CODE
RETURN
NO
PSUEDO-OPERATION?
YES
11
NO
9
9
PARSE OPERAND FIELD
DETECT ERRORS
11
COMPLETE THE GENERATION OF OBJECT CODE
OUTPUT THE SOURCE STATEMENT & ERRORS (if any)
LOAD THE MACHINE INSTRUCTION INTO SIMULATED MEMORY
10


INTERPRETER
INITIALIZE PROGRAM COUNTER
INCREMENT PROGRAM COUNTER
FETCH NEXT INSTRUCTION
PERFORM BINARY OPCODE TABLE LOOKUP
DECODE OPERANDS
12
13
12
DETECT ERRORS
EXECUTE MICROPROGRAM
HALT INSTRUCTION?
YES
NO
RETURN
13

APPENDIX C

SAMPLE RUNS AND OUTPUT

M A C H I N E   D E F I N I T I O N

* *

GENERAL PURPOSE REGISTER (GPR) SYSTEM   INTERDATA 7/16

NUMBER OF GPR'S   16

GPR SIZE   16

MDR SIZE   16

MAR SIZE   16

PC SIZE   32

NUMBER OF IR'S   2

IR SIZE   32

IMPLICIT BASE REGISTER   0

NUMBER OF INSTRUCTION FORMATS USED   8

NUMBER OF OP CODES   54

LENGTH OF OP CODES   8

NUMBER OF WORDS READ IN A READ INSTR   1

NUMBER OF WORDS WRITTEN IN A WRITE INSTR   1

WORD SIZE   16

INSTRUCTION FIELD BOUNDARIES

1	0 TO 7, 8 TO 11
3.	0 TO 7, 8 TO 11, 12 TO 15
5.	0 TO 7, 8 TO 11, 12 TO 15
8	0 TO 7, 8 TO 31
9.	0 TO 7, 8 TO 11, 12 TO 31
11.	0 TO 7, 8 TO 11, 12 TO 15, 16 TO 31
14.	0 TO 7, 8 TO 11, 12 TO 31
21.	0 TO 7

OP CODES AND THEIR MICRO-DEFINITIONS

1. MNEMONIC AH
   BIT PATTERN 01001010
   MACHINE CYCLES 1
   INSTRUCTION TYPE 11
   MICRO-DEFINITION 00014120902 00000002100 00001060904

2. MNEMONIC AHI
   BIT PATTERN 11001010
   MACHINE CYCLES 1
   INSTRUCTION TYPE 14
   MICRO-DEFINITION 00006150904

3. MNEMONIC AHM
   BIT PATTERN 01100001
   MACHINE CYCLES 2
   INSTRUCTION TYPE 11
   MICRO-DEFINITION 00014120902 00000002100 00001060901 00000002200

4. MNEMONIC AHR
   BIT PATTERN 00001010
   MACHINE CYCLES 1
   INSTRUCTION TYPE 3
   MICRO-DEFINITION 00006070904

5. MNEMONIC AIS
   BIT PATTERN 00100110
   MACHINE CYCLES 1
   INSTRUCTION TYPE 9
   MICRO-DEFINITION 00012060904

6. MNEMONIC B
   BIT PATTERN 00110000
   MACHINE CYCLES 1
   INSTRUCTION TYPE 8
   MICRO-DEFINITION 00012000103

7. MNEMONIC BAL
   BIT PATTERN 01000001
   MACHINE CYCLES 1
   INSTRUCTION TYPE 11
   MICRO-DEFINITION 00003000108 00012140903 00016000104

8. MNEMONIC BALR
   BIT PATTERN 00000001
   MACHINE CYCLES 1
   INSTRUCTION TYPE 3
   MICRO-DEFINITION 00003000103 00007000103 00016000104

9. MNEMONIC BE
   BIT PATTERN 10110011
   MACHINE CYCLES 1
   INSTRUCTION TYPE 8
   MICRO-DEFINITION 10300000000 00012000103

```
10.  MNEMONIC  BL
     BIT PATTERN  00101000
     MACHINE CYCLES   1
     INSTRUCTION TYPE   8
     MICRO-DEFINITION    10500000000  00012000103

11.  MNEMONIC  BM
     BIT PATTERN  00100001
     MACHINE CYCLES   1
     INSTRUCTION TYPE   9
     MICRO-DEFINITION    00006000106  00900000000  00012000103

12   MNEMONIC  BNE
     BIT PATTERN  10100011
     MACHINE CYCLES   1
     INSTRUCTION TYPE   8
     MICRO-DEFINITION    10400000000  00012000103

13.  MNEMONIC  BNL
     BIT PATTERN  00111000
     MACHINE CYCLES   1
     INSTRUCTION TYPE   8
     MICRO-DEFINITION    10800000000  00012000103

14.  MNEMONIC  BNM
     BIT PATTERN  00110001
     MACHINE CYCLES   1
     INSTRUCTION TYPE   9
     MICRO-DEFINITION    00006000106  11000000000  11100000000  00012000103

15.  MNEMONIC  BNP
     BIT PATTERN  00110010
     MACHINE CYCLES   1
     INSTRUCTION TYPE   9
     MICRO-DEFINITION    00006000106  00005000107  10100000000  10700000000  00012000103

16   MNEMONIC  BNZ
     BIT PATTERN  00100011
     MACHINE CYCLES   1
     INSTRUCTION TYPE   9
     MICRO-DEFINITION    00006000106  00005000107  10100000000  10400000000  00012000103

17   MNEMONIC  BP
     BIT PATTERN  00100010
     MACHINE CYCLES   1
     INSTRUCTION TYPE   9
     MICRO-DEFINITION    00006000106  11000000000  00012000103

18.  MNEMONIC  BR
     BIT PATTERN  00110100
     MACHINE CYCLES   1
     INSTRUCTION TYPE   1
     MICRO-DEFINITION    00014000103
```

```
19   MNEMONIC  BZ
     BIT PATTERN  00110011
     MACHINE CYCLES   1
     INSTRUCTION TYPE   9
     MICRO-DEFINITION    00006000106  11100000000  00012000103 -

20.  MNEMONIC  CH
     BIT PATTERN  01001001
     MACHINE CYCLES   2
     INSTRUCTION TYPE  11
     MICRO-DEFINITION    00012140902  00000002100  00006000106  00001000107  10100000000

21.  MNEMONIC  CHR
     BIT PATTERN  00001001
     MACHINE CYCLES   1
     INSTRUCTION TYPE   3
     MICRO-DEFINITION    00006000106  00007000107  10100000000

22.  MNEMONIC  CHI
     BIT PATTERN  11001001
     MACHINE CYCLES   1
     INSTRUCTION TYPE  14
     MICRO-DEFINITION    00006000106  00015000107  10100000000

23.  MNEMONIC  CLH
     BIT PATTERN  01000101
     MACHINE CYCLES   2
     INSTRUCTION TYPE  11
     MICRO-DEFINITION    00012140902  00000002100  00006000106  00001000107  10100000000

24.  MNEMONIC  CLHR
     BIT PATTERN  00000101
     MACHINE CYCLES   1
     INSTRUCTION TYPE   3
     MICRO-DEFINITION    00006000106  00007000107  10100000000

25.  MNEMONIC  CLHI
     BIT PATTERN  11000101
     MACHINE CYCLES   1
     INSTRUCTION TYPE  14
     MICRO-DEFINITION    00006000106  00015000107  10100000000

26.  MNEMONIC  DUMP
     BIT PATTERN  00010100
     MACHINE CYCLES  10
     INSTRUCTION TYPE  21
     MICRO-DEFINITION    00021000000

27.  MNEMONIC  HALT
     BIT PATTERN  00010010
     MACHINE CYCLES   1
     INSTRUCTION TYPE  21
     MICRO-DEFINITION    00024000000
```

```
28.  MNEMONIC  LCS
     BIT PATTERN  00101001
     MACHINE CYCLES   1
     INSTRUCTION TYPE   9
     MICRO-DEFINITION    00012001604

29.  MNEMONIC  LH
     BIT PATTERN  01001000
     MACHINE CYCLES   1
     INSTRUCTION TYPE  11
     MICRO-DEFINITION    00012140902  00000002100  00001000104

30.  MNEMONIC  LHI
     BIT PATTERN  11001000
     MACHINE CYCLES   1
     INSTRUCTION TYPE  14
     MICRO-DEFINITION    00015000104

31.  MNEMONIC  LHR
     BIT PATTERN  00001000
     MACHINE CYCLES   1
     INSTRUCTION TYPE   3
     MICRO-DEFINITION    00007000104

32.  MNEMONIC  LIS
     BIT PATTERN  00100100
     MACHINE CYCLES   1
     INSTRUCTION TYPE   9
     MICRO-DEFINITION    00012000104

33.  MNEMONIC  NH
     BIT PATTERN  01000100
     MACHINE CYCLES   2
     INSTRUCTION TYPE  11
     MICRO-DEFINITION    00012140902  00000002100  00006011804

34.  MNEMONIC  NHI
     BIT PATTERN  11000100
     MACHINE CYCLES   1
     INSTRUCTION TYPE  14
     MICRO-DEFINITION    00006151804

35.  MNEMONIC  NHR
     BIT PATTERN  00000100
     MACHINE CYCLES   1
     INSTRUCTION TYPE   3
     MICRO-DEFINITION    00006071804

36   MNEMONIC  NOP
     BIT PATTERN  00010001
     MACHINE CYCLES   0
     INSTRUCTION TYPE  21
     MICRO-DEFINITION    00025000000
```

```
37.  MNEMONIC  OR
     BIT PATTERN  01000110
     MACHINE CYCLES    2
     INSTRUCTION TYPE  11
     MICRO-DEFINITION     00012140902  00000002100  00006011904

38   MNEMONIC  ORI
     BIT PATTERN  11000110
     MACHINE CYCLES    1
     INSTRUCTION TYPE  14
     MICRO-DEFINITION     00006151904

39.  MNEMONIC  ORR
     BIT PATTERN  00000110
     MACHINE CYCLES    1
     INSTRUCTION TYPE    3
     MICRO-DEFINITION     00006071904

40.  MNEMONIC  RH
     BIT PATTERN  11011001
     MACHINE CYCLES    3
     INSTRUCTION TYPE  11
     MICRO-DEFINITION     00012140902  00019000000

41.  MNEMONIC  SH
     BIT PATTERN  01001011
     MACHINE CYCLES    1
     INSTRUCTION TYPE  11
     MICRO-DEFINITION     00014120902  00000002100  00001061004

42.  MNEMONIC  SHI
     BIT PATTERN  11001011
     MACHINE CYCLES    1
     INSTRUCTION TYPE  14
     MICRO-DEFINITION     00006151004

43   MNEMONIC  SHR
     BIT PATTERN  00001011
     MACHINE CYCLES    1
     INSTRUCTION TYPE    3
     MICRO-DEFINITION     00006071004

44.  MNEMONIC  SIS
     BIT PATTERN  00100111
     MACHINE CYCLES    1
     INSTRUCTION TYPE    9
     MICRO-DEFINITION     00006121004

45.  MNEMONIC  SLLS
     BIT PATTERN  10010001
     MACHINE CYCLES    1
     INSTRUCTION TYPE    5
     MICRO-DEFINITION     00006000118
```

```
46.  MNEMONIC  SRLS
     BIT PATTERN  10010000
     MACHINE CYCLES   1
     INSTRUCTION TYPE   5
     MICRO-DEFINITION    00006000119

47.  MNEMONIC  STH
     BIT PATTERN  01000000
     MACHINE CYCLES   1
     INSTRUCTION TYPE  11
     MICRO-DEFINITION    00012140902  00006000101  00000002200

48.  MNEMONIC  THI
     BIT PATTERN  11000011
     MACHINE CYCLES   1
     INSTRUCTION TYPE  14
     MICRO-DEFINITION    00006000106  00015000107  10100000000

49.  MNEMONIC  TRCFF
     BIT PATTERN  00010101
     MACHINE CYCLES   1
     INSTRUCTION TYPE  21
     MICRO-DEFINITION    00023000000

50   MNEMONIC  TRCN
     BIT PATTERN  00010011
     MACHINE CYCLES   1
     INSTRUCTION TYPE  21
     MICRO-DEFINITION    00022000000

51.  MNEMONIC  WH
     BIT PATTERN  11011000
     MACHINE CYCLES   3
     INSTRUCTION TYPE  11
     MICRO-DEFINITION    00012140902  00020000000

52.  MNEMONIC  XR
     BIT PATTERN  01000111
     MACHINE CYCLES   2
     INSTRUCTION TYPE  11
     MICRO-DEFINITION    00012140902  00000002100  00006012004

53.  MNEMONIC  XRI
     BIT PATTERN  11000111
     MACHINE CYCLES   1
     INSTRUCTION TYPE  14
     MICRO-DEFINITION    00006152004

54   MNEMONIC  XRR
     BIT PATTERN  00000111
     MACHINE CYCLES   1
     INSTRUCTION TYPE   3
     MICRO-DEFINITION    00006072004
```

```
>>JOB TESTPGM1,(INTERDATA 7/16)
>>TIME=390
>>REGION=80K
>>EXEC ASM,E
>>ASM.PROG
>>ASM.DATA
>>
```

A S S E M B L E R

S O U R C E   L I S T I N G

```
ADDR      INSTRUCTION          STMT  SOURCE * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                                  1  *          THE FOLLOWING PROGRAM TESTS SOME OF THE INSTRUCTIONS
                                  2  *       IMPLEMENTED FOR THE INTERDATA 7/16 BY GENERATING THE FIRST
                                  3  *       N NUMBERS IN THE FIBONACCI SEQUENCE STARTING WITH ZERO.  THE
                                  4  *       VALUE, N, IS READ FROM AN INPUT CARD AND SEVERAL SEQUENCES
                                  5  *       MAY BE GENERATED BY SUPPLYING THE DESIRED VALUE FOR N.  THE
                                  6  *       PROGRAM IS TERMINATED WHEN A VALUE OF ZERO OR A NEGATIVE VALUE
                                  7  *       IS READ FOR N.
                                  8  *
00000000  1100                    9  TESTPGM1 NOP
                                 10  R0      EQU     0          *
                                 11  R1      EQU     1          *
                                 12  R2      EQU     2          *
                                 13  R3      EQU     3          *
                                 14  R4      EQU     4          *
                                 15  R5      EQU     5          *
                                 16  R6      EQU     6          *
                                 17  R7      EQU     7          *    NAME ALL REGISTERS
                                 18  R8      EQU     8          *    FOR CLARIFICATION
                                 19  R9      EQU     9          *
                                 20  R10     EQU     10         *
                                 21  R11     EQU     11         *
                                 22  R12     EQU     12         *
                                 23  R13     EQU     13         *
                                 24  R14     EQU     14         *
                                 25  R15     EQU     15         *
                                 26  *
                                 27  READN   EQU     *
```

```
00000001  C8F0  003E          28          LHI     R15,=A(N)       R15 CONTAINS ADDR OF N
00000003  C8E0  001F          29          LHI     R14,=X'001F'    R14 CONTAINS "DUMMY" DEVICE
00000005  D9EF  0000          30          RH      R14,R15,0       INPUT N
00000007  48DF  0000          31          LH      R13,R15,0       LOAD R13 WITH N
00000009  32D0  003D          32          BNP     R13,TESTEND     TEST FOR END-OF-FILE
0000000B  C8C0  002F          33          LHI     R12,=X'002F'    R12 CONTAINS "DUMMY" DEVICE
0000000D  D8CF  0000          34          WH      R12,R15,0       OUTPUT N
0000000F  C5D0  0001          35          CLHI    R13,=F'1'       TEST FOR N = 1
00000011  A300  0018          36          BNE     MORE            MORE THAN ONE NUMBER
00000013  27D0  0001          37          SIS     R13,1           SUBTRACT 1 FROM N
00000015  4CDF  0000          38          STH     R13,R15,0       STORE NEW N
00000017  D8CF  0000          39          WH      R12,R15,0       OUTPUT VALUE IN N
00000019  3000  0001          40          B       READN           BRANCH TO READ ANOTHER N
                              41  MORE    EQU     *
0000001B  C8B0  003F          42          LHI     R11,=A(ZERO)    R11 CONTAINS ADDR OF ZERO
0000001D  C8A0  0040          43          LHI     R10,=A(ONE)     R10 CONTAINS ADDR OF ONE
0000001F  489F  0000          44          LH      R9,R15,0        R9 CONTAINS N
00000021  487B  0000          45          LH      R7,R11,0        LOAD ZERO INTO R7
00000023  D8CB  0000          46          WH      R12,R11,0       WRITE ZERO
00000025  486A  0000          47          LH      R6,R10,0        LOAD ONE INTO R6
00000027  D8CA  0000          48          WH      R12,R10,0       WRITE ONE
00000029  2790  0002          49          SIS     R9,2            DECREMENT N BY 2
0000002B  C840  0041          50          LHI     R4,=A(SAVE)     R4 CONTAINS ADDR OF SAVE AREA
                              51  WRITE   EQU     *
0000002D  0855                52          SHR     R5,R5           CLEAR R5
0000002E  0A57                53          AHR     R5,R7           ADD R7 TO R5
0000002F  0A56                54          AHR     R5,R6           ADD R6 TO R5
00000030  4C54  0000          55          STH     R5,R4,0         STORE R5 IN SAVE AREA
00000032  D8C4  0000          56          WH      R12,R4,0        WRITE SAVE AREA
00000034  2790  0001          57          SIS     R9,1            DECREMENT R9 BY 1
```

```
00000036  3390  0001          58           BZ      R9,READN      TEST FOR LAST VALUE
00000038  0876                59           LHR     R7,R6         LOAD R7 FROM R6
00000039  4864  0000          60           LH      R6,R4,0       LOAD R6 FROM SAVE AREA
0000003B  3000  002D          61           B       WRITE         CONTINUE
                              62  TESTEND  EQU     *
0000003D  1200                63           HALT
0000003E  0001  0000          64  N        DS      1X'0000'
0000003F  0000                65  ZERO     DC      F'0'
00000040  0001                66  ONE      DC      F'1'
00000041  0001  0000          67  SAVE     DS      1X'0000'
                              68           END     TESTPGM1
```

SYMBOL TABLE & CROSS REFERENCE DICTIONARY

SYMBOL	VALUE	DEFN	REFERENCES						
MORE	0000001B	0041	0036						
N	0000003E	0064	0028						
ONE	00000040	0066	0043						
READN	00000001	0027	0040	0058					
R0	00000000	0010							
R1	00000001	0011							
R10	0000000A	0020	0043	0047	0048				
R11	00000009	0021	0042	0045	0046				
R12	0000000C	0022	0033	0034	0039	0046	0048	0056	
R13	0000000D	0023	0031	0032	0035	0037	0038		
R14	0000000E	0024	0029	0030					
R15	0000000F	0025	0028	003C	0031	0034	0038	0039	0044
R2	00000002	0012							
R3	00000003	0013							
R4	00000004	0014	0050	0055	0056	0060			
R5	00000005	0015	0052	0053	0054	0055			
R6	00000006	0016	0047	0054	0059	0060			
R7	00000007	0017	0045	0053	0059				
R8	00000008	0018							
R9	00000009	0019	0044	0049	0057	0058			
SAVE	00000041	0067	0050						
TESTEND	00000030	0062	0032						
TESTPGM1	00000000	0009	0068						
WRITE	0000002D	0051	0061						
ZERO	0000003F	0065	0042						

***** NUMBER OF ASSEMBLY ERRORS 0

PRE-EXECUTION MEMORY DUMP

ADDR							
000C0000	1100	C8F0	003E	C8E0	001F	D9EF	0000	48DF
00C00008	0000	3200	003D	C8C0	002F	D8CF	0000	C5D0
00C00010	0001	A300	0013	27D0	0001	40DF	0000	D8CF
00C00018	0000	3000	0001	C9B0	003F	C8A0	0040	489F
00C00020	0000	4878	0000	D8CB	0000	486A	0000	D8CA
00000028	0000	2790	0002	C840	0041	0855	0A57	0A56
00C00030	4054	0000	D8C4	0000	2790	0001	3390	0001
00C00038	C876	4884	0000	3000	002D	1200	0000	0000
00C00040	0001	00C0	00C0	0000	0000	0000	0000	0000
00C00048	0000	0000	0000	0000	0000	0000	00C0	0000

24
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657

POST-EXECUTION MEMORY DUMP

ADDR								
00000000	1100	C8F0	003E	C8E0	001F	D9EF	0000	48DF
00000008	0000	3200	0030	C8C0	002F	D8CF	0000	C5D0
00000010	0001	A300	001B	2700	0001	400F	0000	D8CF
00000018	0000	3000	0001	C880	003F	C8A0	0040	489F
00000020	0000	427B	0000	D8C8	0000	486A	0000	D8CA
00000028	C000	2790	0002	C840	0041	0855	0A57	0A56
00000030	4054	0000	D8C4	0000	2790	0001	3390	0001
00000038	0876	4864	0000	3000	002D	1200	FFFF	0000
00000040	0001	6FF1	0000	0000	0000	0000	0000	0000
00000048	0000	0000	0000	0000	0000	0000	0000	0000

***** MACHINE CYCLES 295

MACHINE DEFINITION

****

FIXED PURPOSE REGISTER (FPR) SYSTEM HEWLETT-PACKARD 2114A

NUMBER OF INDEX REGISTERS 0

INDEX REGISTER SIZE 0

MDR SIZE 16

MAR SIZE 16

PC SIZE 16

NUMBER OF INSTRUCTION REGISTERS 1

IR SIZE 16

ACCUMULATOR-A SIZE 16

ACCUMULATOR-B SIZE 16

NUMBER OF INSTRUCTION FORMATS USED 3

OPCODE LENGTH 4

NUMBER OF WORDS READ IN A READ INSTR 1

NUMBER OF WORDS WRITTEN IN A WRITE INSTR 1

NUMBER OF NON-OPERATE INSTRUCTIONS 15

NUMBER OF REGISTER OPERATE INSTRUCTIONS 35

WORD SIZE 16

INSTRUCTION FIELD BOUNDARIES

1. 0 TO 3, 4 TO 5, 6 TO 15

3. 0 TO 3, 4 TO 15

4. 0 TO 3

NON-OPERATE OP CODES AND THEIR MICRO-DEFINITIONS

1. MNEMONIC ADA
   BIT PATTERN 1000
   MACHINE CYCLES 4
   INSTRUCTION TYPE 1
   MICRO-DEFINITION 00012000102 00000002100 00007010911

2. MNEMONIC ADB
   BIT PATTERN 1001
   MACHINE CYCLES 4
   INSTRUCTION TYPE 1
   MICRO-DEFINITION 00012000102 00000002100 00008010912

3. MNEMONIC AND
   BIT PATTERN 0010
   MACHINE CYCLES 4
   INSTRUCTION TYPE 1
   MICRO-DEFINITION 00012000102 00000002100 00007011811

4. MNEMONIC CPA
   BIT PATTERN 1010
   MACHINE CYCLES 4
   INSTRUCTION TYPE 1
   MICRO-DEFINITION 00012000102 00000002100 00007000106 00001000107 10100000000 10400000000 00003000503

5. MNEMONIC CPB
   BIT PATTERN 1011
   MACHINE CYCLES 4
   INSTRUCTION TYPE 1
   MICRO-DEFINITION 00012000102 00000002100 00008000106 00001000107 10100000000 10400000000 00003000503

6. MNEMONIC ICR
   BIT PATTERN 0110
   MACHINE CYCLES 4
   INSTRUCTION TYPE 1
   MICRO-DEFINITION 00012000102 00000002100 00007011911

7. MNEMONIC ISZ
   BIT PATTERN 0111
   MACHINE CYCLES 4
   INSTRUCTION TYPE 1
   MICRO-DEFINITION 00012000102 00000002100 00001000501 00000022C0 00001000106 11100000000 00003000503

8. MNEMONIC JMP
   BIT PATTERN 0101
   MACHINE CYCLES 2
   INSTRUCTION TYPE 1
   MICRO-DEFINITION 00012000103

9. MNEMONIC JSB
   BIT PATTERN 0011
   MACHINE CYCLES 4
   INSTRUCTION TYPE 1
   MICRO-DEFINITION 00012000102 00003000101 00000002200 00002000503

```
10.  MNEMONIC  LDA
     BIT PATTERN  1100
     MACHINE CYCLES   4
     INSTRUCTION TYPE   1
     MICRO-DEFINITION    00012000102  00000002100  CCC01C00111

11.  MNEMONIC  LDB
     BIT PATTERN  1101
     MACHINE CYCLES   4
     INSTRUCTION TYPE   1
     MICRO-DEFINITION    00012000102  CCCCCC0210C  CCC01000112

12.  MNEMONIC  NOP
     BIT PATTERN  0001
     MACHINE CYCLES   2
     INSTRUCTION TYPE   4
     MICRO-DEFINITION    00000000000

13.  MNEMONIC  STA
     BIT PATTERN  1110
     MACHINE CYCLES   4
     INSTRUCTION TYPE   1
     MICRO-DEFINITION    00012000102  00007000101  00000002200

14.  MNEMONIC  STB
     BIT PATTERN  1111
     MACHINE CYCLES   4
     INSTRUCTION TYPE   1
     MICRO-DEFINITION    00012000102  00008000101  00000002200

15.  MNEMONIC  TRCN
     BIT PATTERN  0100
     MACHINE CYCLES   4
     INSTRUCTION TYPE   4
     MICRO-DEFINITION    00020000000
```

OPERATE OP CODES AND THEIR MICRO-DEFINITIONS

MNEMONIC OP CODE OPR
BIT PATTERN C000

1. MNEMONIC ALF
   BIT POSITIONS 6 7 8 9
   MACHINE CYCLES 2
   MICRO-DEFINITION 00006000114 00007000115 CC007000115 00007000115 00007000115

2. MNEMONIC ALS
   BIT POSITIONS 6
   MACHINE CYCLES 2
   MICRO-DEFINITION 00006000114 00007000117

3. MNEMONIC BLF
   BIT POSITIONS 4 6 7 8 9
   MACHINE CYCLES 2
   MICRO-DEFINITION 00006000114 00008000119 00008000119 00008000119 00008000119

4. MNEMONIC BLS
   BIT POSITIONS 4 6
   MACHINE CYCLES 2
   MICRO-DEFINITION 00006000114 00008000121

5. MNEMONIC CCA
   BIT POSITIONS 5 6 7
   MACHINE CYCLES 2
   MICRO-DEFINITION 00004000111

6. MNEMONIC CCB
   BIT POSITIONS 4 5 6 7
   MACHINE CYCLES 2
   MICRO-DEFINITION 00004000112

7. MNEMONIC CCE
   BIT POSITIONS 5 8 9
   MACHINE CYCLES 2
   MICRO-DEFINITION 00004000113

8. MNEMONIC CLA
   BIT POSITIONS 5 7
   MACHINE CYCLES 2
   MICRO-DEFINITION 00005000111

9. MNEMONIC CLB
   BIT POSITIONS 4 5 7
   MACHINE CYCLES 2
   MICRO-DEFINITION 00005000112

10. MNEMONIC CLE
    BIT POSITIONS 10
    MACHINE CYCLES 2
    MICRO-DEFINITION 00005000113

11. MNEMONIC CMA
    BIT POSITIONS 5 6
    MACHINE CYCLES 2
    MICRO-DEFINITION 00007001611

```
12.  MNEMONIC  CMB
     BIT POSITIONS   4   5   6
     MACHINE CYCLES   2
     MICRO-DEFINITION    00008001612

13.  MNEMONIC  CME
     BIT POSITIONS   5   8
     MACHINE CYCLES   2
     MICRO-DEFINITION    00013000213

14.  MNEMONIC  HLT
     BIT POSITIONS   5
     MACHINE CYCLES   2
     MICRO-DEFINITION    00022000000

15.  MNEMONIC  INA
     BIT POSITIONS   5  13
     MACHINE CYCLES   2
     MICRO-DEFINITION    00007000511

16.  MNEMONIC  INB
     BIT POSITIONS   4   5  13
     MACHINE CYCLES   2
     MICRO-DEFINITION    00008000512

17.  MNEMONIC  LIA
     BIT POSITIONS   5   7   9
     MACHINE CYCLES   4
     MICRO-DEFINITION    00005000102  00017000000  00000002100  00001000111

18.  MNEMONIC  LIB
     BIT POSITIONS   4   5   7   9
     MACHINE CYCLES   4
     MICRO-DEFINITION    00005000102  00017000000  00000002100  00001000112

19.  MNEMONIC  MIA
     BIT POSITIONS   5   7
     MACHINE CYCLES   4
     MICRO-DEFINITION    00005000102  00017000000  00000002100  00007011911

20.  MNEMONIC  MIB
     BIT POSITIONS   4   5   7
     MACHINE CYCLES   4
     MICRO-DEFINITION    00005000102  00017000000  00000002100  00008011912

21.  MNEMONIC  NOP
     BIT POSITIONS  15
     MACHINE CYCLES   2
     MICRO-DEFINITION    00000000000

22.  MNEMONIC  OTA
     BIT POSITIONS   5   7   8
     MACHINE CYCLES   4
     MICRO-DEFINITION    00005000102  00007000101  00000002200  00018000000
```

```
23.  MNEMONIC  OTB
     BIT POSITIONS    4   5   7   8
     MACHINE CYCLES   4
     MICRO-DEFINITION     00005000102  00008000101  00000002200  00018000000

24.  MNEMONIC  RAL
     BIT POSITIONS    6   8
     MACHINE CYCLES   2
     MICRO-DEFINITION     00006000114  00007000115

25.  MNEMONIC  RAR
     BIT POSITIONS    6   8   9
     MACHINE CYCLES   2
     MICRO-DEFINITION     00006000114  00007000116

26.  MNEMONIC  RBL
     BIT POSITIONS    4   6   8
     MACHINE CYCLES   2
     MICRO-DEFINITION     00006000114  00008000119

27.  MNEMONIC  RBR
     BIT POSITIONS    4   6   8   9
     MACHINE CYCLES   2
     MICRO-DEFINITION     00006000114  00008000120

28.  MNEMONIC  SEZ
     BIT POSITIONS    5  10
     MACHINE CYCLES   2
     MICRO-DEFINITION     00013000106  11100000000  00003000503

29.  MNEMONIC  SLA
     BIT POSITIONS   12
     MACHINE CYCLES   2
     MICRO-DEFINITION     00007000106  11400000000  00003000503

30.  MNEMONIC  SLB
     BIT POSITIONS    4  12
     MACHINE CYCLES   2
     MICRO-DEFINITION     00008000106  11400000000  00003000503

31.  MNEMONIC  SSA
     BIT POSITIONS    5  11
     MACHINE CYCLES   2
     MICRO-DEFINITION     00007000106  00005000107  10100000000  10800000000  00003000503

32.  MNEMONIC  SSB
     BIT POSITIONS    4   5  11
     MACHINE CYCLES   2
     MICRO-DEFINITION     00008000106  00005000107  10100000000  10800000000  00003000503

33.  MNEMONIC  SWP
     BIT POSITIONS    4   5
     MACHINE CYCLES   2
     MICRO-DEFINITION     00007000108  00008000111  00014000112
```

34. MNEMONIC SZA
BIT POSITIONS 5 14
MACHINE CYCLES 2
MICRO-DEFINITION 00007000106 1110C000CC0 C0003000503

35. MNEMONIC SZB
BIT POSITIONS 4 5 14
MACHINE CYCLES 2
MICRO-DEFINITION 0CCC8000106 11100000000 00003000503

ADDRESSING MODES

DIRECT TO CURRENT SECTOR 01

DIRECT THROUGH ZERO SECTOR CO

INDIRECT THROUGH CURRENT SECTOR 11

INDIRECT THROUGH ZERO SECTOR 10

INDEXED, REGISTER 1 00

INDEXED, REGISTER 2 00

```
>>JOB TESTPGM2,(HEWLETT-PACKARD 2114A)
>>TIME=1000
>>REGION=96
>>EXEC ASM E
>>ASM.PROG
>>ASM.DATA
>>
```

```
ADDR     INSTRUCTION        STMT  SOURCE * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                              1    *            THE FOLLOWING PROGRAM TESTS SOME OF THE INSTRUCTIONS
                              2    *       IMPLEMENTED FOR THE HEWLETT-PACKARD 2114A.  THE PROGRAM IS
                              3    *       DESIGNED TO PERFORM THE PRIMITIVE STACK OPERATIONS: (1) "POP"
                              4    *       THE ITEM FROM THE TOP OF THE STACK, AND (2) "PUSH" THE ITEM
                              5    *       ONTO THE TOP OF THE STACK.  WHEN STACK UNDERFLOW OR STACK
                              6    *       OVERFLOW IS DETECTED AN ERROR CODE IS PRINTED AND THE PROGRAM
                              7    *       TERMINATES.
                              8    *            STACK OPERATION COMMANDS ARE READ FROM INPUT CARDS.  A
                              9    *       POSITIVE VALUE FOLLOWED BY THE STACK ITEM, DESIGNATES A "PUSH"
                             10    *       OPERATION.  A NEGATIVE VALUE DESIGNATES A "POP" OPERATION.  A
                             11    *       ZERO VALUE DESIGNATES THE END OF THE STACK COMMANDS.
                             12    *            EACH STACK COMMAND IS PRINTED FOLLOWED BY THE CORRESPONDING
                             13    *       STACK ITEM.  THE COMPLETED STACK AND STACK TOP POINTER MAY BE
                             14    *       VIEWED IN THE POST-EXECUTION MEMORY DUMP.
                             15    *
00000020 1000                16 TESTPGM2 NOP
00000021 0000                17         CPR     CLA,CLB       CLEAR ACCUMULATORS A AND B
00000022 0001                18         LDA     LOWTEST       LOAD (LOWER BOUND ADDR) - 1 INTO A
00000023 0600                19         CPR     CMA           TWO'S COMPLEMENT OF A
00000024 E002                20         STA     LOWTEST       STORE RESULT
00000025 0003                21         LDA     HIGHTEST      LOAD (UPPER BOUND ADDR) + 1 INTO A
00000026 0600                22         CPR     CMA           TWO'S COMPLEMENT OF A
00000027 E004                23         STA     HIGHTEST      STORE RESULT
                             24 INPUT    EQU     *
00000028 0540                25         CPR     LIA           INPUT STACK COMMAND INTO A
00000029 0410                26         CPR     SSA           SKIP IF C(A) >= 0
0000002A 540E                27         JMP     POP           POP ITEM FROM TOP OF STACK
```

```
C000002B  0402          28            CPR     SZA         SKIP IF C(A) = 0
C000002C  541C          29            JMP     PUSH        PUSH ITEM CNTO TOP OF STACK
C000002D  C4C0          30            CPR     HLT         STOP RUN
                        31  POP       EQU     *           "POP" PROCEDURE
C000002E  0580          32            CPR     OTA         OUTPUT STACK COMMAND
C000002F  C005          33            LDA     STACKPTR    LOAD STACK POINTER INTO A
C0000030  8006          34            ADA     LOWTEST     ADD LOWER BOUND TEST VALUE TO A
C0000031  0402          35            CPR     SZA         SKIP IF C(A) = 0
C0000032  5416          36            JMP     DELETE      BRANCH TO DELETE ITEM FROM STACK
                        37  UNDRFLOW  EQU     *
C0000033  C007          38            LDA     ERRFLAG1    LOAD STACK UNDERFLOW FLAG INTO A
C0000034  0580          39            CPR     OTA         OUTPUT STACK UNDERFLOW ERROR
C0000035  0400          40            CPR     HLT         STOP RUN
                        41  DELETE    EQU     *
C0000036  C808          42            LDA   I STACKPTR    LOAD STACK TOP ITEM INTO A
C0000037  0580          43            CPR     OTA         OUTPUT STACK TOP ITEM
C0000038  C009          44            LDA     STACKPTR    LOAD STACK POINTER INTO A
C0000039  800A          45            ADA     NEGONE      DECREMENT STACK POINTER BY 1
C000003A  E008          46            STA     STACKPTR    STORE RESULT
C000003B  5408          47            JMP     INPUT       BRANCH TO READ ANOTHER COMMAND
                        48  PUSH      EQU     *           "PUSH" PROCEDURE
C000003C  0040          49            CPR     LIB         INPUT NEW STACK ITEM INTO B
C000003D  0580          50            CPR     OTA         OUTPUT STACK COMMAND
C000003E  0080          51            CPR     OTB         OUTPUT NEW STACK ITEM
C000003F  C00C          52            LDA     STACKPTR    LOAD STACK POINTER INTO A
C0000040  0404          53            CPR     INA         INCREMENT STACK POINTER BY 1
C0000041  E414          54            STA     STACKPTR    STORE RESULT
C0000042  8417          55            ADA     HIGHTEST    ADD UPPER BOUND TEST VALUE TO A
C0000043  0402          56            CPR     SZA         SKIP IF C(A) = 0
C0000044  5408          57            JMP     INSERT      BRANCH TO INSERT ITEM IN STACK
```

```
                                    58  OVERFLOW EQU       *
00000045   C419                     59           LDA       ERRFLAG2       LOAD STACK OVERFLOW FLAG INTO A
CCCCCC46   0580                     60           CPR       0TA            OUTPUT STACK OVERFLOW FLAG
C0CC0047   0400                     61           OPR       HLT            STOP RUN
                                    62  INSERT   EQU       *
CCC00048   FC14                     63           STB   I   STACKPTR       PUSH ITEM ONTO TOP OF STACK
C0000049   500D                     64           JMP       INPUT          BRANCH TC REAC ANCTHER COMMAND
                                    65  *
C000004A   000A  0000               66  STACK    DS        10X'0000'      STACK AREA
000C0054   CC49                     67  STACKPTR DC        A(STACK-1)     INITIALIZE STACK POINTER
C0000055   FFFF                     68  NEGONE   DC        F'-1'          DECREMENT VALUE FOR STACK POINTER
000C0056   0049                     69  LOWTEST  DC        A(STACK-1)     LOWER BOUND TEST VALUE FOR STACK PTR
CCCOCC57   0054                     70  HIGHTEST DC        A(STACK+10)    UPPER BOUND TEST VALUE FOR STACK PTR
C0000058   DD48                     71  ERRFLAG1 DC        F'-8888'       UNDERFLOW FLAG
C00C0059   D8F1                     72  ERRFLAG2 DC        F'-9999'       OVERFLOW FLAG
                                    73           END       TESTPGM2
```

SYMBOL TABLE & CROSS REFERENCE DICTIONARY

SYMBOL	VALUE	DEFN	REFERENCES						
DELETE	CCCC0036	0041	0036						
ERRFLAG1	00000058	0071	0038						
ERRFLAG2	CCCC0059	0072	0059						
HIGHTEST	C0000057	0070	0021	0023	0055				
INPUT	00000028	0024	0047	0064					
INSERT	CCCC0048	0062	0057						
LOWTEST	C0000056	0069	0018	0020	0034				
NEGONE	00000055	0068	0045						
OVERFLOW	C0000045	0058							
POP	0000002E	0031	0027						
PUSH	CCC0003C	0048	0029						
STACK	0000004A	0066	0067	0069	0070				
STACKPTR	00000054	0067	0033	0042	0044	0046	0052	0054	0063
TESTPGM2	CCCC0020	0016	0073						
UNDRFLOW	00000033	0037							

***** NUMBER OF ASSEMBLY ERRORS 0

PRE-EXECUTION MEMORY DUMP

ADDR								
CCCCCCC0	0000	0056	0056	0057	0057	0054	0056	0058
00000008	0054	0054	0055	0054	0054	0028	0000	0000
00000010	0000	0000	0000	0000	0000	0000	0000	0000
00000018	0000	0000	0000	0000	0000	0000	0000	0000
00000020	1000	0D00	C001	0600	E002	CC03	06C0	E004
00000028	0540	0410	540E	0402	541C	0400	0580	C005
0000003C	8006	0402	5416	C007	0580	0400	C808	0580
00000038	C009	800A	E00B	5408	0D40	0580	0D80	C00C
00000040	0404	E414	8417	0402	5408	C419	0580	0400
00000048	FC14	500D	000C	0000	0000	0000	0000	0000
00000050	0000	0000	0000	0000	0049	FFFF	0049	0054
00000058	DD48	D8F1	0000	0000	0000	0000	0000	0000

```
1
128
1
256
1
512
1
1024
1
2048
1
4096
1
8192
1
16384
-1
16384
-1
8192
-1
4096
-1
2048
```

POST-EXECUTION MEMORY DUMP

ADDR							
00000000	0000	0056	0056	0057	0057	0054	0056	0058
00000008	0054	0054	0055	0054	0054	0028	0000	0000
00000010	0000	0000	0000	0000	0000	0000	0000	0000
00000018	0000	0000	0000	0000	0000	0000	0000	0000
00000020	1000	0D00	C001	060C	EC02	CC03	060C	E004
00000028	0540	0410	540E	0402	541C	0400	0580	C005
00000030	8006	0402	5416	C007	0580	0400	0308	0580
00000038	C009	800A	E00B	5408	0D40	058C	0D80	C00C
00000040	0404	E414	8417	0402	5408	0419	0580	0400
00000048	FC14	500D	CC8C	0100	0200	0400	0800	1000
00000050	2000	4000	0000	000C	004D	FFFF	FFB7	FFAC
00000058	DD48	D8F1	0000	0000	0000	0000	0000	0000

***** MACHINE CYCLES 588

APPENDIX D

SYSTEMS PROGRAMMERS GUIDE

Additions to the simulation system have been made to aid in the maintenance of the program. Column 13 of the first input card in the machine definition phase (Chapter V) is used to signal a trace of the interpretation of all assembler statement operand fields. This feature aids in determining the validity of the operands in the generated machine code. All of the generated machine code can be examined on the assembler listing.

Features of the simulation system that may require modification are: (1) the maximum size of the simulated memory, (2) the maximum length of the symbol and reference tables, and (3) the default time and region parameters for execution.

The maximum size of a machine language program is 2000 words. This should be sufficient memory for the execution of most programs, but it may be increased for the execution of larger programs. To increase the size of the simulated memory, the upper bound on the memory array can be changed to the desired value. If it is necessary to change the maximum memory size to 2500, for example, the PL/I program statements

```
DECLARE
     MEM (0:2000) BIT (32),
     MEMDF (0:2000, 0:31) BIT (1) DEFINED MEM,
```

should be changed to

```
DECLARE
     MEM (0:2500) BIT (32),
     MEMDF (0:2500, 0:31) BIT (1) DEFINED MEM,
```

The maximum number of identifiers that can be specified in a single assembler program is 100. The maximum number of references to the

identifiers is 200. Both of these limits can be increased by changing the upper bounds on the two arrays. For example, if it is necessary to change the upper bound of the symbol table array to 150 and the upper bound of the reference table array to 300, the PL/I program statements

```
DECLARE
     1 SYMTBL EXTERNAL,
       2 DECTBL (101),
         .
         .
         .
       2 REFTBL (200),
         .
         .
         .
MAXSYM = 100;
MAXREF = 200;
```

should be changed to

```
DECLARE
     1 SYMTBL EXTERNAL,
       2 DECTBL (151),
         .
         .
         .
       2 REFTBL (300),
         .
         .
         .
MAXSYM = 150;
MAXREF = 300;
```

The default number of machine cycles for the execution of a program is 500. The default region size is 200 words. Both of these parameters are easily changed in the JCL for the user program (Chapter V).

The PL/I debug options of SUBSCRIPTRANGE, SIZE, and STRINGRANGE

have been enabled for the execution of the entire simulation system. These features aid in the initial debugging process of a program, but they tend to decrease the performance of a program because of the error checking that is performed. If desired, these debug options can be disabled, or enabled only for specific sections of the program, which increases the execution speed and decreases the size of the generated object program.

VITA

Glenn Ray Thompson

Candidate for the Degree of

Master of Science

Thesis: A MICROPROGRAMMED SIMULATION SYSTEM FOR GENERAL PURPOSE REGISTER AND FIXED PURPOSE REGISTER MINICOMPUTERS

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Fort Smith, Arkansas, July 16, 1952, the son of Mr. and Mrs. W. R. Thompson.

Education: Graduated from Mount Saint Mary's High School, Oklahoma City, Oklahoma, in May, 1970; received Bachelor of Science degree in Mathematics from Oklahoma State University in May, 1974; completed requirements for Master of Science degree at Oklahoma State University in May, 1976.

Professional Experience: Graduate teaching assistant, Oklahoma State University, Computing and Information Sciences Department, August, 1975, to May, 1976; systems analyst, Armco Steel Corporation, May, 1975, to August, 1975; graduate teaching assistant, Oklahoma State University, Mathematics Department, August, 1974, to May, 1975.