SIMULATION AND APL DESCRIPTION

OF THE PDP 11/40

By

ANAND VARDHAN PANDIT

Bachelor of Engineering

Osmania University

Hyderabad, India

1973

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
July, 1975

SIMULATION AND APL DESCRIPTION
OF THE PDP 11/40

Thesis Approved:

Donald D Fisher
Thesis Adviser

Donald W. Grace

D. E. Hedrick

N N Durham
Dean of the Graduate College

PREFACE

This paper describes the implementation of an assembler-simulator for the PDP 11/40 computer. It is concerned with methods used to implement an assembler, to generate code which is interpretively executed by a simulator. Program-controlled input/output as well as device-initiated input, has been implemented. The assembler-simulator programs are written in PL/1, and are implemented on the IBM 360/65.

I would like to express my gratitude to my major adviser, Dr. Donald D. Fisher, for his advice and guidance during this project. Also, appreciation is expressed to my other committee members, Dr. Donald W. Grace and Dr. George E. Hedrick, for their suggestions and assistance in the preparation of this paper. Thanks are also due to Mrs. Debbie Williams for typing the paper.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Symbol	Dimension	Function
ADC		Address computation defined operation
EXEC		Instruction execution defined operation
IOIG		I/O interrupt generator system program
MAC		Memory Access defined operation
PROC		Processor unit system program
C	16	Device control and status register
C_8		'Done' bit
C_9		Interrupt enable bit
D		Decoding matrices
F	5	Instruction fields
I	16	Instruction register
I_0		Word/byte instruction bit
M	2^{16},8	Main memory
N	85,10	Navigation matrix
R	8,16	General registers
R_6		Processor stack pointer
R_7		Program counter
U	56	Unibus lines
U_{0-15}		Data lines
U_{16-33}		Address lines
$U_{34,35}$		Control lines

Symbol	Dimension	Function
U_{40-43}		Bus request lines BR 7:4
a	2	effective address
a_1		first address
a_2		second address
b		local variable
e	5	program exceptions/traps
e_0		odd addressing
e_1		reserved instructions
e_2		time out
e_3		trap instructions
e_4		trace trap
g		local variable
h	2	interrupt holder
h_0		exceptions
h_1		I/O interrupt
i,j,k		local variables
m		addressing mode
n	10	navigation vector
n_0		instruction class
n_1		entry line in EXEC
$n_{2,3,4}$		branch control in EXEC
p	16	Processor Status word
$p_{0,1}$		current operation mode
$p_{2,3}$		previous operation mode
$p_{8,9,10}$		Processor priority

Symbol	Dimension	Function
p_{11}		Trap (T) bit
$p_{12,13,14,15}$		Condition code
q	5	memory access queue
r	5	memory access request
s	2	selection vector
u.v.w		local vectors

CHAPTER I

INTRODUCTION

The electronic computer affords a very powerful tool for system simulation. It is not by chance, therefore, that the significant increase in system simulations has almost paralleled the growth of electronic computers. The systems that are simulated can be business/ economic systems, social systems, environmental systems or even other computer systems. One of the many reasons for simulating systems on a digital computer is the rapidity with which results are obtained. Another reason is the provision it gives to consider the problem to any level of detail.

The reasons for simulating a system can be many fold (9). Among them are the facility of studying a dynamic system in either real time, compressed or expanded time, the ability to study a complicated system by breaking it into component subsystems, the provision it gives to experiment with the system being simulated without actually building a prototype.

Simulation of computer systems can be done either at the "macro" level or at the "micro" level (8). At the macro level, the effects of processing complete jobs are simulated, and each transaction may represent a total job. This level of simulation may be used to study the effects of an increase in the workload of the system, or the quality of service measured in terms of the turn-around time, quality of service under a projected workload, etc.

Micro level simulation involves extremely fine level of detail. The effect of each individual machine language instruction is simulated. At this level of simulation, a unit of real time requires many units of simulated time. Therefore, micro level simulation can be expensive, both in terms of programming and running times, and requires a detailed understanding of the system.

During development of software packages for minicomputers, the debugging stage performance may be severely limited by the computer memory size, input-output facilities, or by the lack of translators with diagnostic capabilities. It therefore becomes desirable to simulate the minicomputer on a large host computer, in a higher level language, to get the software packages at least past the debugging stage. Simulation in a higher level language provides ease with which data structures can be manipulated. Even if the simulator does not mimic the simulated machine in its entirety, it may be set up to simulate a sizable subset of the assembler package. Such a simulation has to be done at the micro level.

In this report a large subset of the PDP 11/40 assembly language has been implemented on an IBM 360/65 host computer in PL/1. The implementation also includes simulation of program controlled input/output from peripheral devices like the teletype and papertape reader/punch. Device-initiated interrupt simulation, using the computer interrupt structure is also incorporated in the implementation.

Chapter II describes the PDP 11/40 computer. Most of the description is based on the Digital Equipment Corporation system manuals of the PDP 11/40 (10, 11, 12). The architecture, instruction formats, processor operation, interrupt structure, and input-output are covered.

The simulator itself is made up of an assembler and an interpreter. The two pass assembler is described in Chapter III. The scanner, to pick up symbols, symbol table construction during pass I and generation of object code in pass II are covered. Assembly time error detection is also discussed. A good description of the various methods adopted for searching/sorting during table processing can be found in Hellerman (2) and Wegner (14). The factors determining the choice of the method are presented in Gear (1).

The object code generated by the assembler unit forms the input to the interpreter. The object code is loaded into memory before execution can begin. Chapter IV contains a description of instruction fetch and execution, execution-time error checking and debugging facilities, input-output, output formats, etc. The program setup for simulating device-initiated interrupts is also described.

A formal description of the PDP 11/40 is presented in Chapter V. The description is in APL (4) and models the description of the IBM S/360 by Falkoff, Iverson and Sussenguth (5). Programs for processor operation, interrupt handling, address calculation, instruction execution have been described. A word of caution has to be given at this point. Arrays (registers, instruction words, etc.) as handled in the PDP 11/40 system manuals, have the least significant bit position numbered 0, and the most significant bit position numbered 15, as shown in Figure 1.

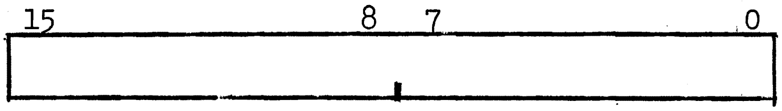


Figure 1. A PDP 11 Word as Used in the System Manuals

Since the simulator has been modeled on the descriptions in the manuals, words have been treated as shown in Figure 1. However, in the APL description in Chapter V, "words" are treated as shown in Figure 2, to be consistent with the language terminology, with the most significant bit numbered 0, and the least significant bit numbered 15.

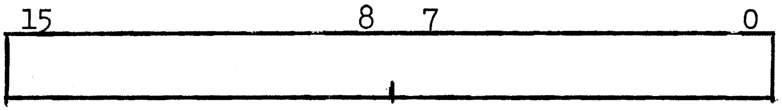


Figure 2. A Word as Treated in the APL Description (Chapter V)

Chapter VI is a Users Manual and describes the deck setup and options for using the assembler-simulator. The assembler output format, error messages and codes, are also discussed. A summary and conclusions are presented in Chapter VII. The program flowchart is given in Appendix A. Appendix B consists of a description of a sample run and the output. The machine operation code symbol table is given in Appendix C.

CHAPTER II

PDP 11/40 ARCHITECTURE

The PDP 11/40 is a 16-bit general purpose, parallel logic computer using two's complement arithmetic. The processor can address directly 32K 16-bit words or 64K 8-bit bytes. All communications among system components are performed on a single high-speed bus, the Unibus. The processor contains 16 hardware registers, eight of which are programmable. The eight nonprogrammable registers are used for storage of a variety of functions including intermediate addresses, source-destination data, console operation data, and the stack pointer for the Memory Management option. The eight programmable general purpose registers R0-R7 can be used as accumulators, pointers to memory locations, or full word index registers, but their most important function is to hold operand and result addresses. Two of these registers R6 and R7 are used as processor stack pointer and program counter, respectively. This means that the contents of R6 and R7 are changed automatically by various instructions and, hence, cannot be used as general purpose registers.

System Organization

The whole computer is organized around a single bus called the Unibus. The processor, memory and all peripheral devices share the same high speed bus (Figure 3). Because of the bus concept, all peripherals are compatible, and device-to-device transfers can be accomplished at a fast rate. The Unibus enables the processor to view peripheral devices as active memory locations and treat peripheral device addresses exactly like (nonrelocatable) memory addresses, in the basic system address space. The processor uses the same set of signals to communicate with memory as with peripheral devices. Memory locations, processor registers, device status and data registers are each assigned a unique address. All instructions that can be applied to date in core can be applied equally well to peripheral device registers, enabling peripheral devices to be manipulated as flexibly as memory.

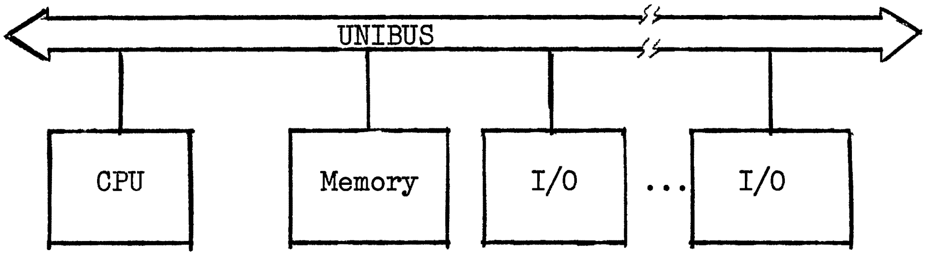


Figure 3. Basic System Organization

Unibus Operation

Communication between system components is over the 56 lines of the Unibus, 51 of which are bidirectional and 5 unidirectional. A bidirectional line permits signal flow in both directions. The five unidirectional bus grant (BG) lines are for priority bus control signals (10, pp. 179). The function of the 56 lines is as follows:

(1) 16 bidirectional data lines which carry all data transfers.

(2) 18 address lines. The same addressing scheme is used for programmed I/O, programmed processor/memory transfers, direct memory access (DMA).

(3) 22 control-logic and parity check lines.

Figure 4 shows the processor, memory and a peripheral device connected to the Unibus. The peripheral device as has been interfaced for programmed instructions, direct memory access and interrupt.

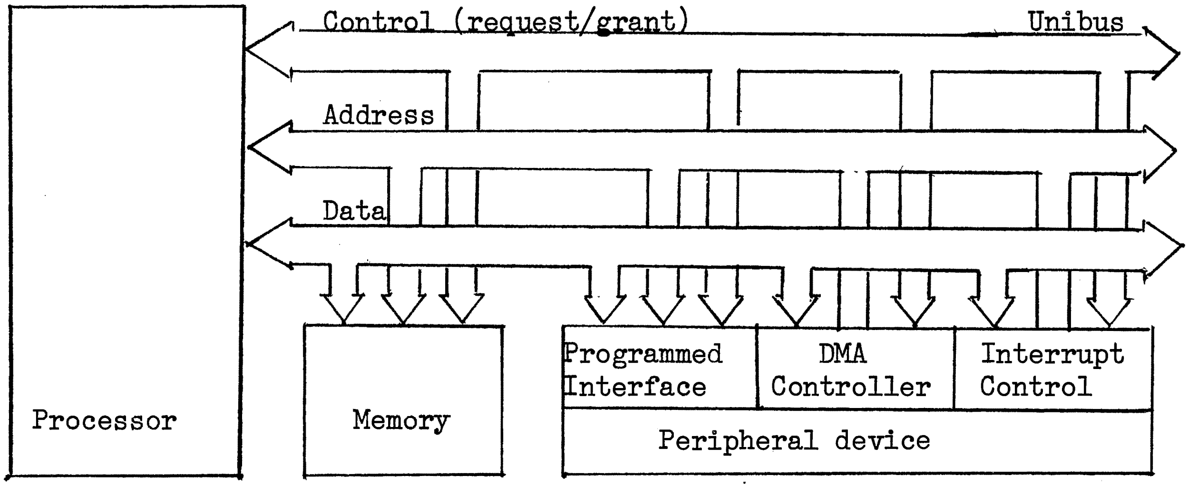


Figure 4. The Unibus System (6)

Master/Slave Operation

All bus activity is asynchronous and depends on interlocking of controlled signals. During transfer between two devices, the device controlling the bus is termed the "master," and the other device the "slave." Master-slave relationships are dynamic. Memory is always a slave. The nature of interlocked communication requires that for each control signal issued by the master, the slave issue a response to complete the transfer.

Full 16-bit word or 8-bit byte information can be transferred on the bus between master and slave. Bus operations can be classified into data operations and control operations. The DATI, DATIP data operations transfer data into the master, while the DATO, DATOB data operations transfer data out of the master (10, pp. 182). The bus request (BR) and nonprocessor request (NPR) control signals are used by devices to gain control of the bus. Bus control obtained under a BR is for an interrupt whereas control obtained under an NPR is for a direct memory access (DMA). A device can perform a DMA after acquiring bus control via a BR. Transfer of bus control from one device to another is made by a priority arbitration logic.

Memory Organization

PDP 11 memory can be addressed either as 16-bit words or 8-bit bytes. Words always start at even numbered memory locations. A PDP 11 "word" is divided into a high byte and a low byte as shown in Figure 5.

High Byte	Low Byte

Figure 5. The PDP 11 Word

Low bytes are stored at even numbered memory locations and high bytes at odd numbered locations. Memory addresses 0-255 are reserved for the system (interrupt vectors, trap vectors, etc.) and the top 4K words are reserved for general purpose registers, peripheral device registers, etc. The user, therefore, has 28K of the 32K directly addressable memory to program.

Processor Status Word (PS)

The processor status word, Figure 6, contains information about the status of the machine. The status can be described by the processor priority, current and previous operation modes and condition code.

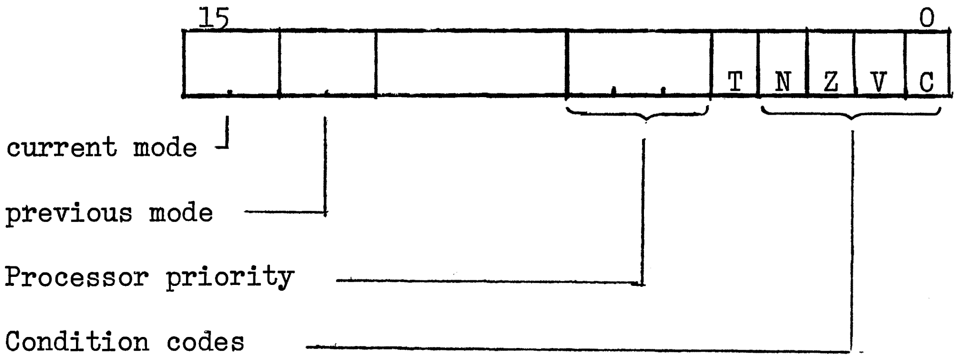


Figure 6. Processor Status Word

The two modes of operation (11, pp. 2-4) Kernel and User modes are available under the Memory Management option.

The processor can operate at any one of the 8 priority levels 0-7. The current priority is maintained in bits 7-5 of the PS.

The 4-bit condition code is set by any of a number of instructions, including many arithmetic instructions. The condition code is set depending on the result of the instruction. Conditions setting the bits are given in Table I.

TABLE I

CONDITION CODE BITS OF THE PROCESSOR STATUS WORD

PS bit	Bit name	Condition setting the bit
3	N	Result is negative
2	Z	Result is zero
1	V	Arithmetic overflow
0	C	Carry from the most significant bit

The trace trap bit T can be set or cleared under program control. When set, a processor trap will occur through location 14 on completion of instruction execution and a new processor status word will be loaded. The trace trap is a system debugging aid and is transparent to the programmer.

Processor Stack

To allow a programmer to make efficient use of frequently accessed data a processor stack is maintained in memory. Register 6 serves as a pointer to the top of the stack. The stack can be maintained anywhere in memory by initializing register 6 in the program. A typical processor stack is built with addresses decreasing from bottom to top as shown in Figure 7.

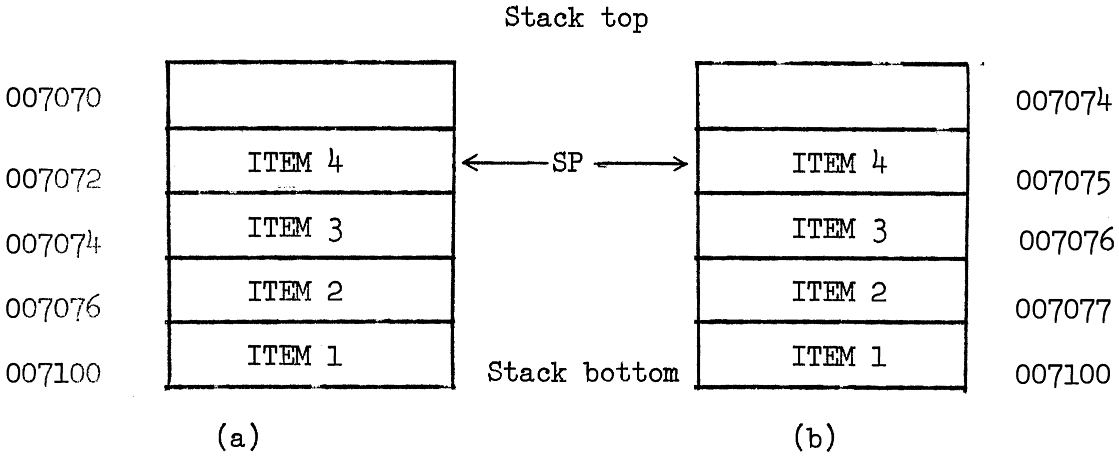


Figure 7. Processor Stacks (a) Word Stack (b) Byte Stack

Under the Memory Management option, the PDP 11 has two stacks called the Kernel and User stacks. When the processor is operating under the Kernel mode, it uses the Kernal stack and when operating under User mode, the User stack. The stack overflow boundary is at location 256. The Kernel stack boundary is a variable and is set through a stack limit register. Once the Kernel stack exceeds its boundary, the processor traps to location 4 after the current instruction is executed.

The stack permits save and restore of the program counter and status word in conjunction with subroutine calls and interrupts. This feature allows reentrant codes and nesting of subroutines. Items can be added or removed from the stack by using the autodecrement and autoincrement addressing modes with register 6.

Interrupt Structure

Since all components use the same Unibus, a certain amount of contention arises when more than one device requests to become bus master. A multilevel automatic priority structure is imposed to overcome this problem.

The Unibus contains 13 lines classified as priority transfer lines. Five of these are the bus request (BR) lines BR (7:4) and NPR and five are the corresponding bus grant (BG) lines BG (7:4) and NPG which the processor uses to respond to a request.

The priority arbitration logic assigns highest priority to NPR direct memory access data transfers. These requests are honored by the processor between bus cycles of an instruction execution. BR7 is the next highest priority and BR4 the lowest. These requests are honored by the processor between instructions. The priority is hardwired into each device except the processor. For example, the teletype and papertape reader/punch have a preassigned priority of BR4.

The processor priority can be set under program control to any one of the levels. This inhibits granting of bus requests on the same or lower levels and provides an effective masking technique. The priority interrupt structure is shown in Figure 8.

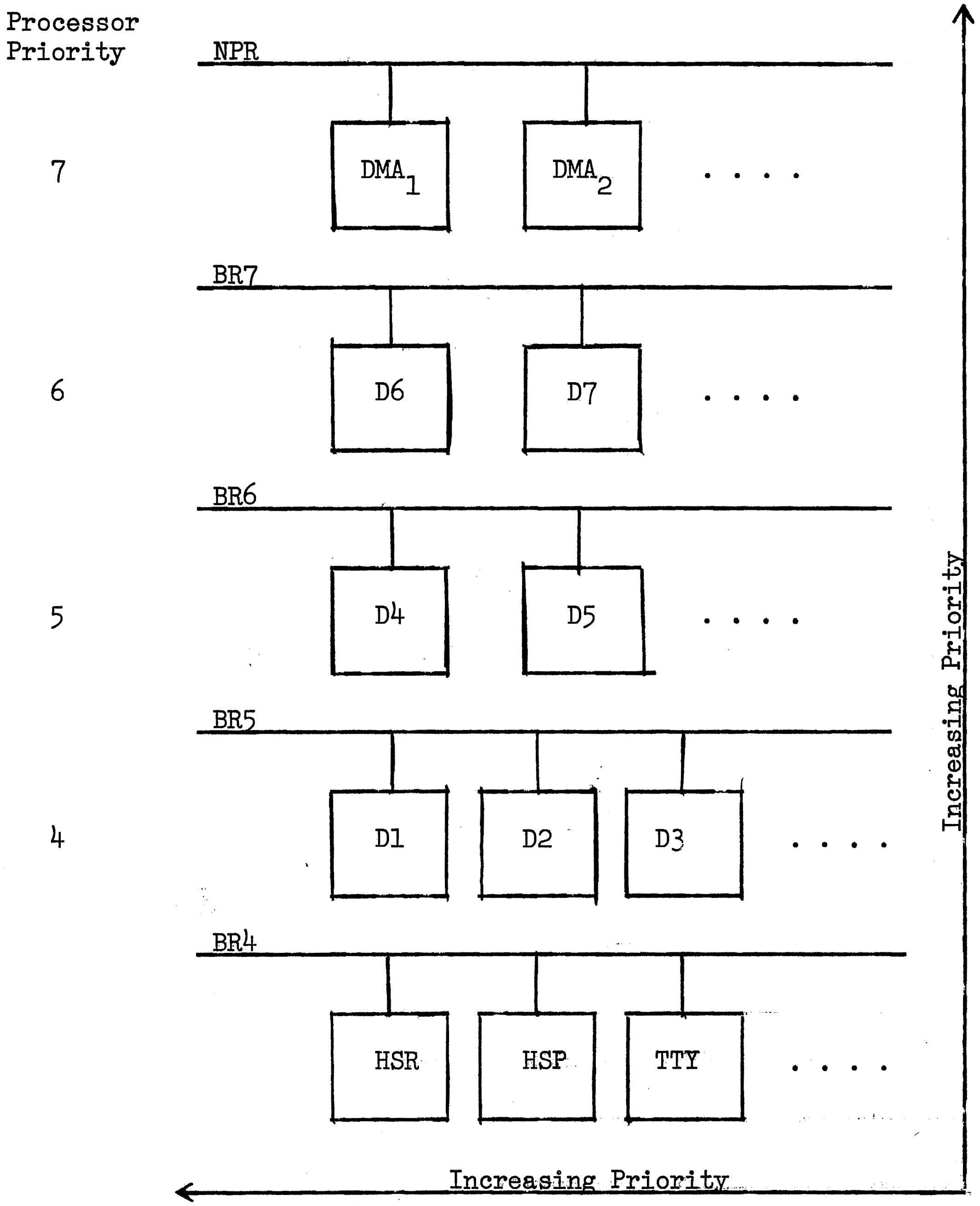


Figure 8. Priority Interrupt Structure (11)

Any number of devices can be chain-wired on each level, the device nearer the processor having a higher priority than a device farther away.

Each device on a particular priority level passes a grant signal to the next device on the line unless it has requested bus control; in this case the requesting device blocks the signal from the following devices and assumes bus control. A device may cause interrupt operation to occur any time it gains bus control on one of the BR lines.

Interrupt Service

Each device has a unique interrupt vector address in memory. These addresses are transmitted over the bus address lines. Two consecutive words in memory, the starting address of the service routine and the new PS are stored at the interrupt vector address. This unique identification eliminates the necessity of device polling. The operations required to service an interrupt can be described in APL as shown in Figure 9.

The operations involve pushing the Program Status word on the stack, lines 0,1 (Figure 9a), followed by pushing the program counter, lines 2, 3. The new program counter, which is the address of the service routine and the new Program Status word are loaded from the interrupt vector address, lines 4,5. Upon completion of service, a return from interrupt automatically restores the program counter and the old PS.

$R^6 \leftarrow (16)\top(\bot R^6)-2$	0
$M^{\bot R^6}, M^{1+\bot R^6} \leftarrow \omega^8/p,\ (\alpha^8/p)$	1
$R^6 \leftarrow (16)\top(\bot R^6)-2$	2
$M^{\bot R^6}, M^{1+\bot R^6} \leftarrow \omega^8/R^7,\ (\alpha^8/R^7)$	3
$R^7 \leftarrow M^{1+a}, M^a$	4
$p \leftarrow M^{3+a},\ M^{2+a}$	5

(a)

Legend

a	Interrupt vector address
M	Memory
	M^i is byte i
p	Program Status word
R^6	Register 6, the stack pointer
R^7	Register 7, the Program counter
	$\nu R^6 \equiv \nu R^7 \equiv \nu p \equiv 16$

(b)

Figure 9. Functional Description of Interrupt Control

Addressing and Instruction Set

Much of the power of the machine is derived from its wide ranging addressing capabilities. Addressing can be done either at the word level or the byte level and is performed through general registers which can be used interchangeably as accumulators, index registers or pointers to memory locations.

The five different instruction formats are as follows:

(1) Single operand

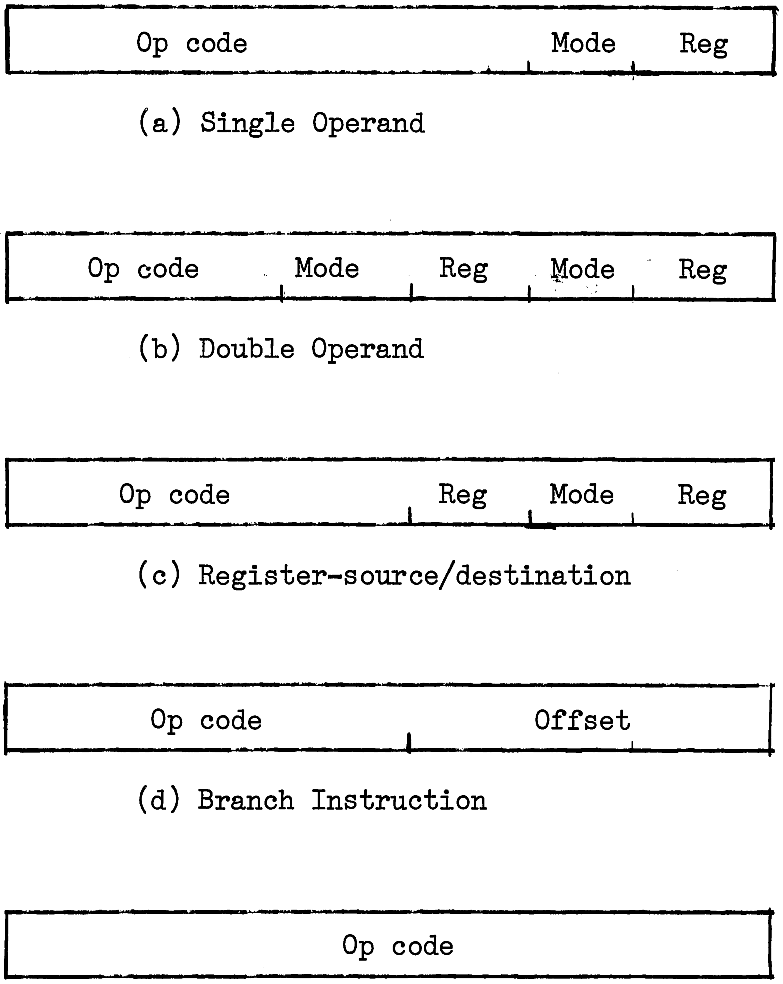
(2) Double operand

(3) Register-source/destination

(4) Branch instruction

(5) Operate instruction

These five formats are shown in Figure 10.



(a) Single Operand

(b) Double Operand

(c) Register-source/destination

(d) Branch Instruction

(e) Operate Instruction

Figure 10. Instruction Formats

Each operand in 1), 2) and 3) is specified by a general purpose register and the mode for using the register. The operand in Branch instructions are specified by an 8-bit word offset; the resetting of the program counter can be functionally represented in APL as shown in Figure 11.

$$\rightarrow \boxed{R^7 \leftarrow (16)\top(\perp R^7) + 2 \times ((\perp\omega^7/I) - I_8 \times 2^7) \qquad 0} \rightarrow$$

(a)

Legend	
I	Instruction register
R^7	Program counter
	$\nu I \underline{\underline{=}} \nu R^7 \underline{\underline{=}} 16$

(b)

Figure 11. Program Counter Modification in Branch Instruction

The offset in a branch instruction is the number of words from the current contents of the PC. The offset, given by the last 7 bits of the instruction register, is treated as a two's complement number. Since the PC expresses a byte address, the offset is multiplied by 2, to express bytes, before it is added to the PC.

The Operate instructions do not require an operand and execution proceeds immediately after instruction fetch.

Any of eight modes of addressing can be used to specify an operand. These are as follows:

(1) Register mode - Mode 0: Register specified contains the operand. Assembler syntax : Rn.

(2) Register deferred mode - Mode 1: Register specified contains the operand address. Assembler syntax: @ Rn or (Rn).

(3) Autoincrement mode - Mode 2: Register 5 used as a pointer and then incremented: Assembler syntax: (Rn)+
If register specified is R7, the mode is "immediate" and the operand follows the instruction. Assembler syntax: #n.

(4) Auto increment deferred - Mode 3: Register is used as a pointer to word containing operand address and then incremented. Assembler syntax: @(Rn)+. If register specified is R7, the mode is "absolute," and the absolute address follows the instruction. Assembler syntax: @#A.

(5) Auto decrement - Mode 4: Register is decremented and then used as pointer to operand. Assembler syntax: -(Rn).

(6) Auto decrement deferred - Mode 5: Register is decremented (always by 2 even for byte instructions) and then used as pointer to operand address. Assembler syntax: @-(Rn).

(7) Indexed - Mode 6: Word following the instruction is added to the register contents to give operand address. Assembler syntax: X(Rn). If register specified is R7, the mode is "relative" and the relative address follows the instruction: Assembler syntax: A.

(8) Index deferred - Mode 7: Word following instruction added to register contents gives address of the address of the operand. Assembler syntax: @x(Rn). If the register specified is R7, the mode is "relative deferred." Assembler syntax: @A. In all variations of auto increment and auto decrement modes the

register contents are incremented/decremented by 2 for word instructions and by 1 for byte instructions.

Some of the instructions can address both bytes and words. For byte instructions the leftmost bit of the instruction is 1. An APL description of the instruction set is given in Chapter V.

Input/Output and Peripherals

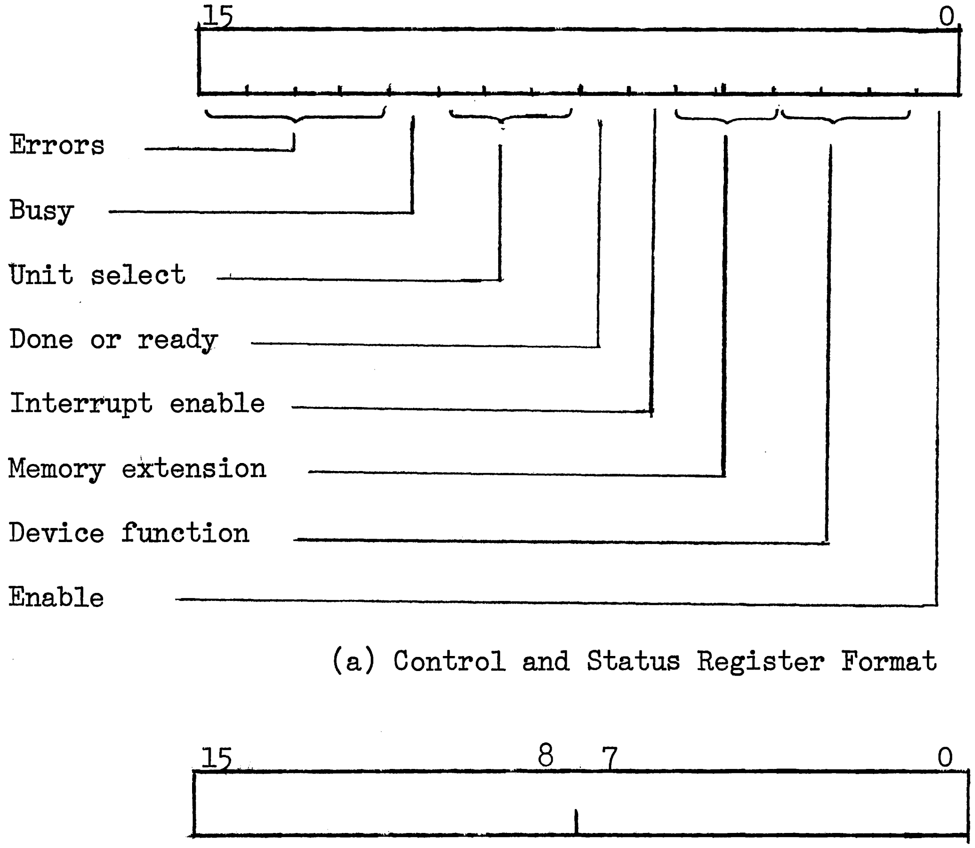
The Unibus permits a unified addressing structure in which control, status and data registers for peripheral devices are directly addressed as memory locations. The use of all memory reference instructions on device registers greatly increases the flexibility of input/output programming.

All peripheral devices are specified by two types of registers. These are 1) control and status registers, and 2) data registers and are shown in Figure 12.

Each device has one or more control and status registers that contain all the information necessary to communicate with that device. Many devices require less than sixteen status bits, and some others more than sixteen and, therefore, require additional registers.

The number and type of data registers associated with a device is a function of the device. Papertape reader and punch use single 8-bit data buffer registers, whereas a disk uses 16-bit data buffer registers.

PDP 11 Input/output devices include teleprinters, line printers, teletypes, card readers, alphanumeric displays. Storage devices range from small reel magnetic tape units to mass storage tape and moving or fixed head disk units.



(a) Control and Status Register Format

(b) Data Register Format

Figure 12. Peripheral Device Registers

CHAPTER III

THE ASSEMBLER

The first step in the simulation is the conversion of the program source code into machine executable object code. This Chapter contains a discussion of the assembly procedure, code generation, error detection and processing and loading the generated code into memory for execution.

To translate the source assembly language, the assembler must (1) replace each mnemonic instruction with its equivalent binary code, and (2) replace each symbolic address with its numerical address. One way of doing the former is by keeping a list of all mnemonic instructions in a table and consulting it to find the binary code, once a mnemonic is read. The latter problem can be approached in a similar manner by having a table of symbols and their addresses.

The assembly process can be subdivided into the following two phases:

(1) Scanning of the symbolic input and transforming symbolic names into corresponding codes.

(2) Assembling the codes for the mnemonics and addresses.

The two phases usually require two scans of the source code. The first scan determines which location is to be assigned to each symbol and on the second scan the assembler produces the binary object code.

Each phase is described in the following paragraph along with the method used for its implementation.

Scanner

During each scan of the source code labels, identifiers, numbers, operators, delimiters and assembler directives need to be picked up for the assembly. This function is performed by a scanner. The scanner, generally, is programmed as a subroutine which is called upon by a higher level routine to perform the scanning.

In this report, a finite state automation (FSA) approach is used for the scanner. Hopcroft and Ullman (3) define a finite automaton M over an alphabet Σ as "a system (K,Σ,δ,q_0,F), where K is a finite, nonempty set of states; Σ is a finite input alphabet; δ is a mapping of $K \times \Sigma$ into K; q_0 in K is the initial state and $F \subseteq K$ is the set of final states." The interpretation of $\delta(q,a) = p$ for q and p in K and a in is that the FSA goes from state q to state p if the input symbol scanned is a. This transition can be represented graphically as in Figure 13.

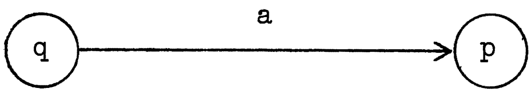


Figure 13. State Transition in a FSA

An input symbol y is said to be accepted or recognized by an FSA if $\delta(q_0,y) = p$ for some p in F.

In this report the FSA is set up to recognize identifiers, labels, numbers, operators, assembler directives and delimiters, which form the vocabulary of the PDP 11/40 assembler. As such, a final state is associated with each of these classes of symbols. The alphabet of the FSA is the character set of the PDP 11/40 assembler. The FSA as set up consists of 21 states, 0 through 20, with state 0 being the initial state. States 1, 2, (4,5), (8,10), 11, (12,13,14,15,16,17,18) and 20 are the final states and correspond to the classes of identifiers, labels, numbers, literals, directives, operators, and delimiters respectively. Each character in the alphabet causes a transition from one state to another. All possible transitions are represented by a state transition matrix DELTA (Table II). The rows correspond to the 21 states and the columns (0-21) to characters of the alphabet. Each entry DELTA (I,J) = N in the table represents a transition from state I to State N under input J. For transitions which are not permitted N = -1. After such a transition the FSA goes into state -1. Once the FSA transits to state -1, the scanning is terminated and the symbol which has been recognized is returned to the routine calling the scanning routine.

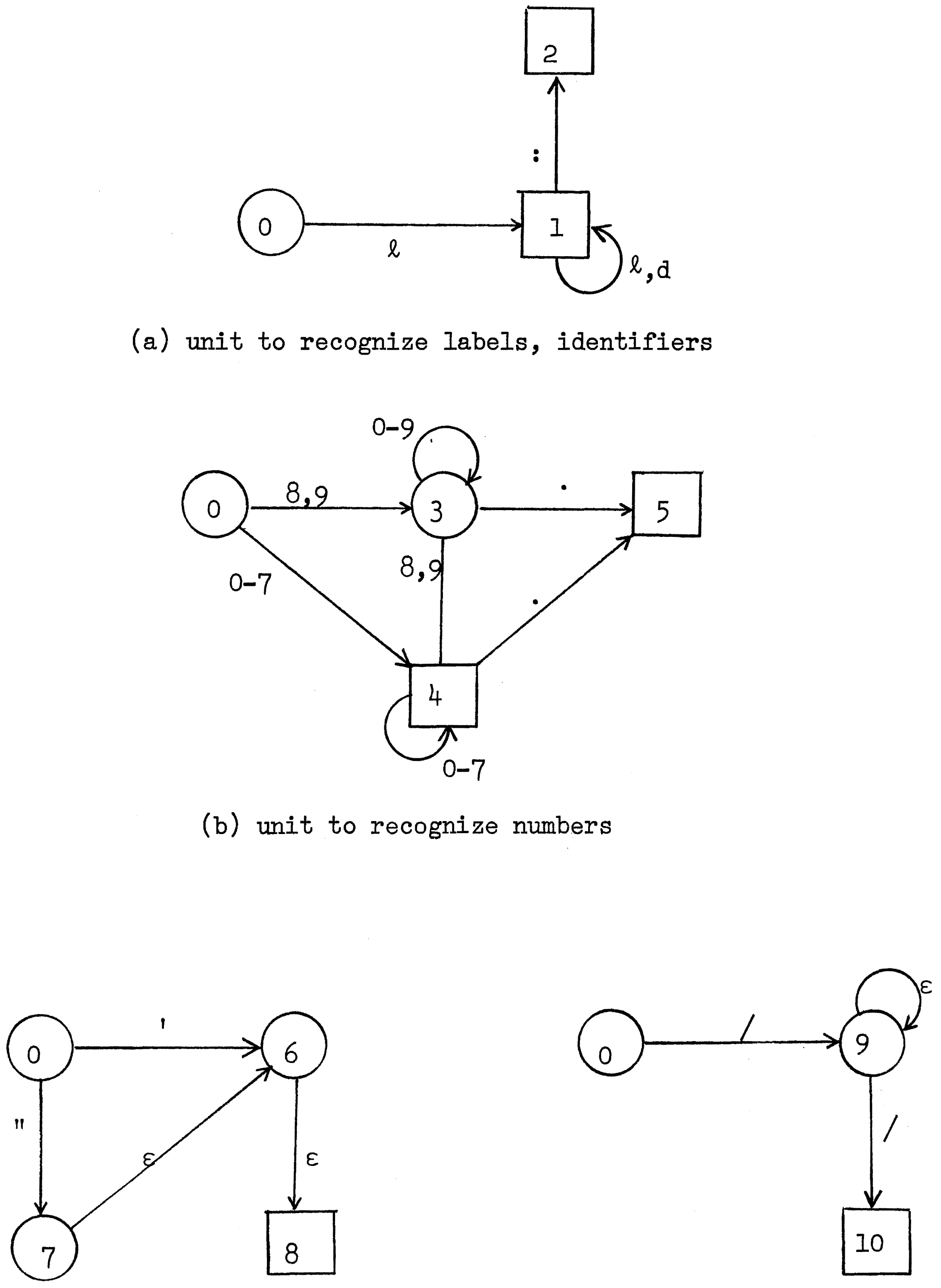
A graphical representation of the FSA is shown in Figure 14, where each component unit recognizes a particular class of symbols. The final states are shown as squares. Once a symbol is recognized, the scanner routine returns the symbol, the symbol class and the state information of the FSA.

TABLE II

THE STATE TRANSITION MATRIX

	VOCABULARY																					
	OTHER		'	"	/	+	&	|	=	#	(	%	)	@	-	,	;	:	.	A-Z,$	0-7	8,9
STATES																						
0	-1	0	6	7	9	12	12	12	12	12	12	12	13	14	15	20	20	-1	11	1	4	3
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2	-1	1	1	1
2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	5	3	3	3
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	5	3	4	3
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
7	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	9	9	9	9	10	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	11	-1	-1
12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
13	-1	-1	-1	-1	-1	16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
14	-1	-1	-1	-1	-1	-1	-1	-1	-1	18	17	-1	-1	-1	19	-1	-1	-1	-1	-1	-1	-1
15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
18	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
19	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
20	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

NOTE –   REPRESENTS A BLANK
-1 REPRESENTS AN INVALID TRANSITION

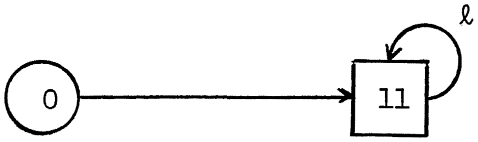


(a) unit to recognize labels, identifiers

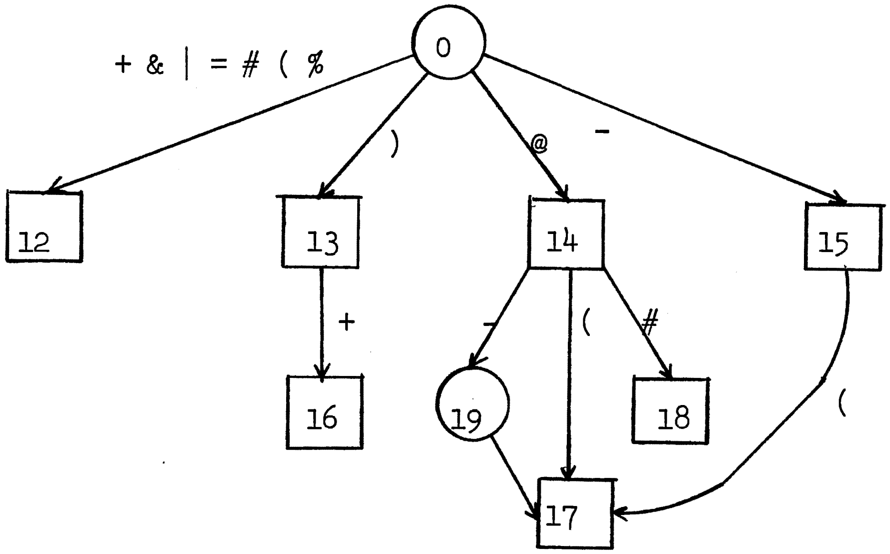
(b) unit to recognize numbers

(c) units to recognize literals

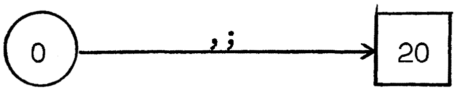
Figure 14. Finite State Automaton to Recognize Syntactical Categories



(d) unit to recognize assembler directives



(e) unit to recognize operators



(f) unit to recognize delimiters

Legend: alphabet Σ = b'"/+&|=#(%)@-,;:.$ABC...ZØ123...9

ℓ = $,A,B,C...Z

ε = any character in Σ

d = Ø,1,2,...9

Figure 14. (Continued)

TABLE III

SYNTACTICAL CATEGORIES AND ASSOCIATED STATES

State	Symbols Recognized
-1	error state
1	identifiers
2	labels
4	octal numbers
5	decimal numbers
8,10	literals
11	directives
12-18	operators
20	delimiters

Examples of the symbol recognition process are described below.

Examples

Numbers in the PDP 11/40 Assembler can either be octal or decimal. Decimal numbers are terminated by the decimal point. 123, 147 are octal numbers whereas 123., 147. are decimal numbers. Consider the number 123 as the input to the FSA. Figure 14b, the unit to recognize numbers gives the transitions under different inputs. Each character in the input causes a state transition. The initial state of the FSA is state 0. Under the first input character, "1," the FSA changes its state to 4. The FSA remains in this state for the input characters "2" and "3." Since the input has been exhausted and the FSA is in one of the final states, the number is a valid number. Similarly for the decimal number 123., the FSA will be in state 5 and the number will be recognized. Invalid octal numbers like 184 or 987, which contain 8 and 9 will cause the FSA to transit to intermediate state 3 and, therefore, will not be recognized. The character A in the invalid number 12A3. will cause a transition to state -1 and, therefore, cause an error.

Table Construction and Processing

The assembler basically has to deal with two types of symbols:

(1) Operation-code symbols

(2) Address symbols.

The binary code corresponding to operation-codes is specified by an operation-code symbol table. The binary code for any mnemonic can then be determined by a table look-up. The format of each node in the operation code symbol table used in the implementation is shown in Figure 15. The table itself is given in Appendix C.

Mnemonic	Number of operands	Link	Operation code	Mnemonic type
6 bytes	1 byte	2 bytes	4 bytes	1 byte

Figure 15. Node Format in Operation Code Table

Address symbols have their codes assigned to them by the assembler. Addresses may be data addresses, assigned according to the storage allocation scheme used for date, or instruction addresses, assigned by determining the address of the instruction having the symbol as its label. An address symbol table is constructed during pass I of the assembly. The format for a node in the address symbol table is shown in Figure 16.

Symbol	Link	Symbol value	Type
6 bytes	2 bytes	2 bytes	1 bit

Figure 16. Node Format in Address-Symbol Table

The scheme adopted for symbol table construction and lookup uses a hashing function. The hashing function partitions the symbols into 16 pseudo-randomly determined classes. The hashing function used is

$$\text{HASH ADDR} \leftarrow 2^4 \mid 5 \downarrow \text{S}.$$

S, the sum of the characters in the symbol taken two at a time from the leftmost position, is shifted 5 bits to the right. The result is divided by 16 and the remainder of the division, a number between 0 and 15, gives the hash address.

The address generated places the symbol into one of 16 buckets

0-15. Synonym generation is handled by placing the symbol into a node in an auxiliary table and linking the node to the bucket to which the symbol is hashed. Each table is, therefore, essentially 16 linked lists. Symbol table lookup involves hashing the symbol to obtain the bucket, and a serial search of the linked list of that bucket.

Pass I

The prime task of the first pass is to construct the address symbol table. A location counter is maintained in both the passes in order to determine the final location of a line of code. After each instruction is translated, the location counter is incremented by the length of the instruction.

The first step is to read a new line of the source for translation. Since the second pass needs to reread the input file, a copy of the input is produced on an auxiliary storage device. Once a source line is read, the assembler extracts various fields from it to form labels, mnemonics, and addresses.

Label Field

The symbol found in the label field is placed in the address symbol table along with its address value. Checking for multiple definitions of a label involves a table lookup to see if the label is already present in the table.

Mnemonic Field

An instruction mnemonic is recognized by the scanner as any other identifier. To see if the mnemonic is valid, an operation code table lookup is performed. If the mnemonic is not present in the table, the error is noted. If present, the address field is scanned.

Assembler directives, recognized by the scanner, are treated as a different class of symbols, distinct from identifiers. During pass I the address field of assembler directives is scanned. The .END directive terminates pass I.

Address Field

This field differs from instruction to instruction and may contain a number of subfields separated by commas. Each subfield may contain an expression involving names, numbers, and/or arithmetic operators.

Since the length of a PDP 11/40 instruction is determined by the "mode" of addressing and not by the mnemonic, the address field is scanned to determine the amount by which the location counter is to be incremented. The mode of addressing can be any one of eight different modes (Chapter II). Each operand, addressed by the indexed, relative, immediate or absolute modes requires an extra word. Instructions can, therefore, depending on the mode of addressing, require one, two or at most three words. During pass I the address field is scanned to determine the mode of addressing and hence the instruction length.

Evaluating Expressions in Address Fields

PDP 11/40 assembler allows only the + and - arithmetic operators in operand expressions. Parentheses are not permitted. Expression evaluation can therefore proceed from left to right. Evaluation may involve conversion of numbers or characters to binary.

Pass II

The purpose of this pass is to translate the source language into binary by using the symbol table constructed in Pass I to convert the addresses, and the operation-code table to convert the mnemonics. A line of code is read in and many of the steps of Pass I repeated. The label field, which is handled completely in Pass I is ignored. The mnemonic field is examined and its binary code fetched by a table lookup. Expressions in the address field are converted to binary. Pass II also is terminated by the .END directive.

Assembler Directives

Assembler directives fall into four classes. Directives for the PDP 11/40 are listed in Table IV.

TABLE IV

PDP 11/40 ASSEMBLER DIRECTIVES

Class	Directive	Action
Data loading	.WORD .BYTE	Load data in decimal or octal
Location counter control	.=	Set location counter
Table entry	=	Enter name with given definition in table
Character conversion	.ASCII	Convert character string to ASCII

For the data loading directives it is necessary to determine how many words of storage will be occupied by the data in the directive so that the location counter may be incremented during Pass I. For .WORD and .BYTE, this requires the address field be scanned to determine the number of data words provided, by counting commas. Character conversion-loading requires a count of the characters in the string, to increment the location counter.

Location counter control directive sets the location counter to a specific value. This requires evaluation of the expression on the right of the relational operator, and assigns the value to the counter. Table entry directives also require expression evaluation.

The symbol name is entered into the table along with the expression value.

During Pass II the address-field expression is converted to binary according to the rules of the directive. Character conversion can be done by a lookup on a table of characters and associated ASCII codes.

Error Detection and Handling

Error detection is done in both the passes. Invalid symbols are caught by the scanner. Other syntactical errors are caught during operand-field expression evaluation. Error handling is done by placing an error code into an error table along with the statement number. Every type of error has an error code associated with it. The type can be determined from the error detecting mechanism built into the program. As an example, if an invalid mnemonic is picked up in a statement, the error code associated with the error, code 9, along with the statement number, are stored in an error table. The error code is printed in the assembly listing, immediately after the statement causing the error. A list of error codes and associated diagnostic messages is given in Chapter IV.

Assembler Output

In addition to code generation the assembler usually produces a printed output. The output consists of the object code in octal, statement number and a listing of the source statement. Errors, if any, are indicated on the output by an error code, immediately following the statement in error. The diagnostic message for an error can be obtained by looking up the code in Table XI, in Chapter VI. Immediately following the source listing, symbols and associated values (in octal) are printed.

Loader

The binary code generated during assembly is saved on a secondary device to be used by the loader. Once the second pass is terminated, the generated code is loaded into memory for execution. Loading is done only if the assembly is error free. The scheme for loading uses a nonrelocatable loader. Loading begins from location 128 unless the location counter is set to a high value. The first 128 locations, locations (0-127), are reserved for the system. Once loading is complete, the program is ready for execution.

CHAPTER IV

THE SIMULATOR

The binary code generated by the assembler is loaded into memory for execution. This Chapter contains a description of instruction fetch and execution, program-controlled input/output, device-initiated interrupts, execution time error detection and debugging aids.

Instruction Fetch and Execution

During the loading process the address of the first instruction is placed in the program counter. During instruction fetch, the word in memory at the location given by the contents of the program counter (PC) are placed into the instruction register (IR). The contents of the PC are incremented by 2, to point to the next instruction. The order in which instructions are fetched and executed is determined by the statements of the program.

Having fetched the instruction from memory into the IR, it is necessary to decode the instruction in order to determine its type. The leftmost bit of the IR, specifies if the instruction is to operate on word or byte operands. The remaining 15 bits of the instruction are divided into five 3-bit fields F0-F4. Instruction decoding is done on the basis of these five fields.

For the purpose of instruction decoding and execution, the instruction set is divided into Single operand instructions, Double operand instructions which include Register-source/destination and Extended Instruction Set (EIS) instructions, Branch instructions and Operate instructions. Once the type is determined by the five fields, instruction execution begins.

The first step during execution is the operand fetch. Each operand in the single and double operand instructions is specified by a register and a mode for using the register. The mode gives the type of addressing used to fetch the operand. For single operand instructions the operand is fetched into the memory buffer register (MBR). For double operand instructions the source and destination operands are fetched into the memory buffer register-source operand (MBR_S) and MBR.

The second step in the execution phase is to operate on the operands in the MBR and MBR_S and nodify them as called for by the particular instruction being executed. Thus, instruction ADD, adds the contents of the two registers, whereas CLR clears MBR. The result of the operation is stored at the address specified in the instruction. For single operand instructions this address is the same as the address of the operand, and for double operand instructions this address is the address of the destination operand. Condition code bits are set/cleared if called for by the instruction.

For Branch instructions the operand address is specified as the offset from the current contents of the PC. This offset is added to/subtracted from the contents of the PC to affect the branch. Branch instructions do not modify the condition code.

Operate instructions do not require operand fetch. Some operate instructions change the condition code.

A detailed APL description of instruction fetch/execution is given in Chapter V.

Program-Controlled Input/Output

Peripheral device registers are treated by the Unibus as non-relocatable memory addresses. Therefore, operations on these registers, such as transferring information into or out of them, or manipulating data within them, are performed by normal memory reference instructions.

All devices are specified by a pair of registers. These are (1) a control and status register that contains all information necessary to communicate with the device, and (2) a data buffer register which temporarily holds data to be transferred into or out of the memory.

Input/output from teletype, papertape reader and papertape punch has been simulated in this report. All program controlled I/O is done by using the interrupt system of the computer. An interrupt request is made to the processor when information is ready to be input/output. Priorities permitting the processor accepts the request. Control passes to the appropriate interrupt service routine. When I/O is complete, the processor regains control and execution of the interrupted program is resumed.

I/O devices, which have been simulated, are all on the lowest priority level--bus request BR4. Among the devices on this level, highest priority is given to the papertape reader, followed by the punch and teletype. Interrupt requests are honored by the processor if its operating priority level is less than 4. Once a request is honored for a device on this level, the processor priority is raised to 4, thus, prohibiting any other device on this level to interrupt. After the interrupt has been serviced, the processor priority is lowered to its previous value. The raising/lowering of processor priority, which provides an efficient interrupt mask, is done by loading a new Processor Status word each time the processor is interrupted. The location from which the PS is loaded is unique to the device interrupting. Each device has a unique two-word interrupt vector address. The second word contains the address of the interrupt service routine. When an interrupt request is honored, the old program counter and PS are pushed onto the processor stack and the service routine address and the new PS are loaded.

For the devices simulated in this report, the addresses in memory of the device interrupt vector, control and status register and the device data register are summarized in Table V.

TABLE V

DEVICE REGISTER AND INTERRUPT VECTOR ADDRESSES (OCTAL)

Device	2-Word Interrupt Vector Address	Control & Status Register		Data Register	
		Actual Address	Address in Simulator	Actual Address	address in Simulator
Teletype					
a) Keyboard/reader	60	177560	134	177562	136
b) Printer/punch	64	177564	140	177566	142
Papertape Reader	70	177550	124	177552	126
Papertape Punch	74	177554	130	177556	132

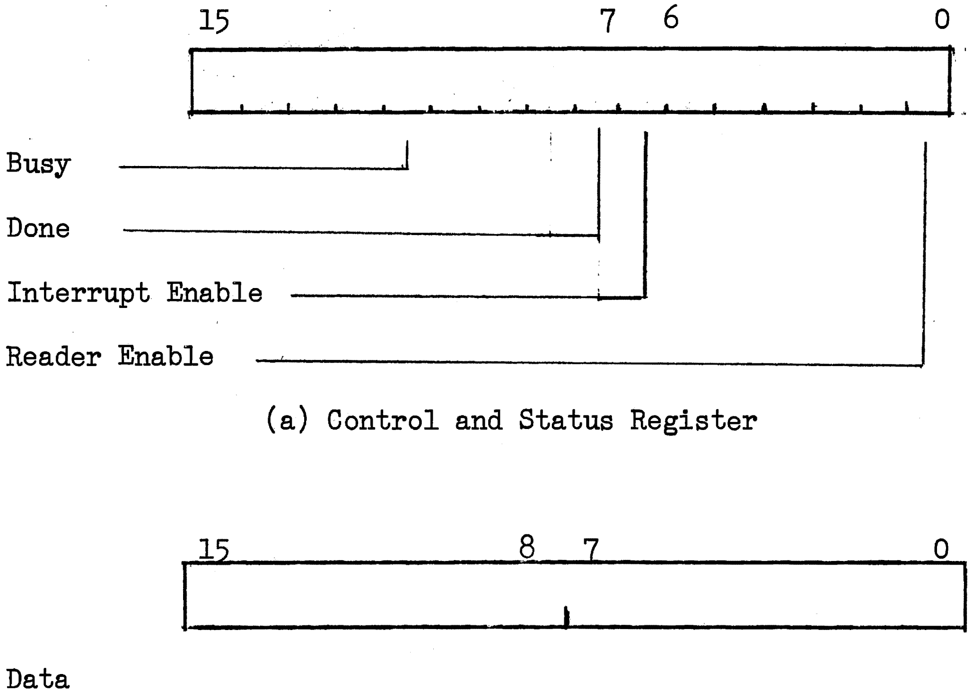
For the purpose of simulation the actual control register and data register addresses were converted to smaller addresses, so that simulation could be done even with a part of the 32K-word addressable memory. The top 128 words of the memory are reserved for the system and contain the interrupt vector addresses and addresses of the device control registers and device data registers.

I/O from the four devices simulated can be broken down into (1) Input from teletype keyboard/reader and papertape reader and (2) Output to teletype printer/punch and papertape punch.

Teletype Keyboard/Reader and Papertape Reader

Input from these two devices is similar in most respects. The

Control and Status register and data buffer register for these devices are shown in Figure 17.



(a) Control and Status Register

(b) Data Buffer Register

Figure 17. Reader Device Registers

TABLE VI

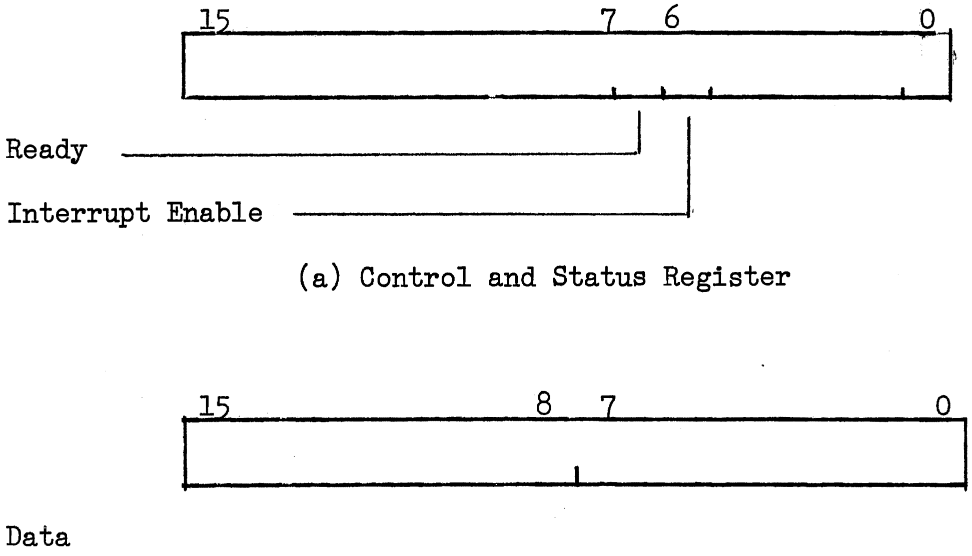
FUNCTION OF READER CONTROL
AND STATUS REGISTER BITS

Bit(s)	Name	Function
11	Busy	Set during reception of information bits
7	Done	Set when character available in buffer; cleared when reader enable is set or data buffer referenced; causes interrupt when interrupt enable is set.
6	Interrupt Enable	Enables interrupt
0	Reader Enable	Enables reader (not keyboard) to read read one character

Input can be initiated by setting the interrupt and reader enable bits in the status register. Setting the reader enable bit causes a character to be read into the buffer. When the character is available, the done bit is set, which causes an interrupt. The interrupt sequence is initiated and control passes to the service routine. When the buffer is referenced in the service routine, to transfer data from it into some location in memory, the done bit is cleared.

Teletype Printer/Punch and Papertape Punch

The manner in which output is handled by these devices is similar. The control and status registers and the data register are shown in Figure 18.



(a) Control and Status Register

(b) Data Buffer Register

Figure 18. Print/Punch Device Registers

The function of the bits of the control and Status register is summarized in Table VII.

TABLE VII

FUNCTION OF PUNCH CONTROL
AND STATUS REGISTER BITS

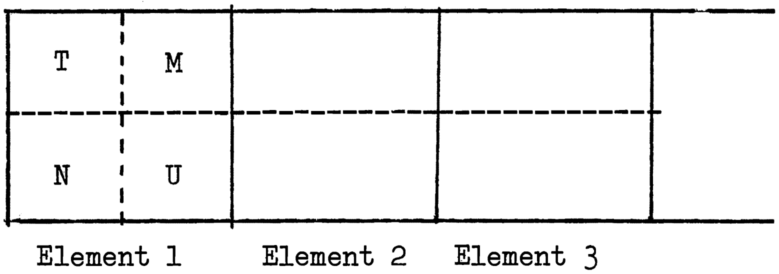
Bit	Name	Function
7	Ready	Punch available; cleared when buffer loaded; set when punching complete.
6	Interrupt Enable	Enables "Ready" to cause interrupt

During simulation the ready bit of the status register is set as part of the initialization process so that the printer/punch is available. To start output, the interrupt enable bit is set. This causes an interrupt and the interrupt sequence is initiated resulting in a branch to the service routine. When the buffer is loaded by the service routine, the ready bit is cleared and punching initiated. When punching is complete, the ready bit is set again.

Deviee-Initiated Inputs

In contrast to program controlled I/O, device-initiated interrupts are treated as nonprocessor request (NPR) type interrupts and, therefore, given the highest priority. NPR requests are honored by the Unibus between bus cycles and are generally for direct memory accesses. This is generally done by a cycle-steal process. Since cycle stealing in no way disturbs the program sequence, there is no need to save register contents and other information as with program interrupts. As simulated in this report, device-initiated interrupts are used only to input data into the computer from the interrupting devices. Device-initiated interrupts and non-processor requests are used synonymously.

The setup to simulate device-initiated inputs included a queue. Each element of the queue represents one NPR. The elements of the queue are initialized before the simulation begins. Each element consists of the interrupt time in cycles, the location in memory where the data is to be input, the number of characters to be input, and the unit number of the interrupting device for purpose of identification, as shown in Figure 19. The elements in the queue are arranged in increasing order of their interrupt times. A cycle counter is maintained as well as a next time to interrupt (NTTI). NTTI is the minimum of all interrupt times in the queue elements. Since the queue elements are arranged in incrpasing order of their interrupt times, NTTI is the interrupt time of the elements beginning from the first and proceeding to the rear of the queue.



T: time to interrupt
M: memory address where data is to be input
N: number of characters to be input
U: unit identification number

Figure 19. Queue for Device-Initiated Interrupt Simulation

When the cycle counter, which is incremented by one after each cycle, equals NTTI, the signals from the device interrupting, are input via a cycle-steal. For this simulation study the external signals are supplied from a file called EXTIN. After one NPR is honored, NTTI is set to the interrupt time of the next element in the queue. Values of T, M, N. U for each element in the queue are user supplied. The format and deck setup are given in Chapter VI.

Error Detection

Execution time errors are caught by the simulator and appropriate messages output. These errors may be due to addressing a word on a byte boundary, overflow/underflow, usage of registers/ modes not permitted in some instructions, etc. Most of the execution time errors are terminal errors and execution is suspended in those cases.

Debugging Aids

To facilitate debugging of programs, debugging aids have been provided in the simulator. Apart from the assembly time and execution time error detection, these aids help detect nonsyntactical execution time errors. Post-execution register and memory dumps are provided. An instruction trace facility is also provided. The instructions SET and CLT, set and clear the trace at (T) in the Processor Status word. When set, the instruction which is executed is traced. Tracing includes a dump of the general purpose registers, program counter, processor stack pointer, pseudo registers and the processor Status word. Setting and clearing the T-bit can be done under program control and provides a powerful debugging tool.

CHAPTER V

APL DESCRIPTION OF PDP 11/40

An APL description of the PDP 11/40 is presented in this Chapter. The computer system is described as seen by a programmer, and the description is independent of any particular hardware implementation. Iverson (4) gives a complete definition of the notation used. The description is based on the PDP 11/40 System Manual (11) and the Processor Handbook (10), and consists of a set of programs and tables.

The programs are either system programs or defined operations. All system programs operate concurrently and continuously, with one line active in each program. The defined operation program operates only when invoked by another program. In the description presented, PROC and IOIG are system programs, whereas ADC, EXEC, and MAC are defined operations. The description covers only those aspects of the system operation, which have been implemented in the assembler-simulator, and therefore, does not describe the PDP 11/40 completely.

The Processor

The PROC program, Figure 20, describes the sequencing and execution of instructions and the servicing of interrupts. The program segments, their functions and the state of the processor during each function are summarized in Table VIII.

PROC system program

```
      =   $1 : ip\ell$                                                            0
          $e \leftarrow \bar{\epsilon}(6)$                                         1
          $MAC^1\ (\perp R^7, 2, f; I)$                                            2
          $R^7 \leftarrow (16)\top 2 + \perp R^7$                                  3
      =   $1 : \vee/e$                                                             4
          $F, i \leftarrow (\perp\omega^3/\alpha^4/I), (\perp\omega^3/\alpha^7/I), (\perp\omega^3/\alpha^{10}/I),$
              $(\perp\alpha^3/\omega^6/I), (\perp\omega^3/I), 0$                   5
          $i \leftarrow (((F_0=0) \wedge (F_1 \geq 5)) \times 0) + (((F_0 \neq 0) \wedge (F_0 \neq 7)) \times 1)$
                $+ ((F_0=7) \times 2) + (((0 = \vee/\bar{\alpha}^1(7)/I) \wedge (F_2 \overset{\vee}{=} 1,3)) \times 4)$
                $+ ((0 = \vee/\bar{\alpha}^1(13)/I) \times 5) + (((0 = \vee/\bar{\alpha}^1(7)/I) \wedge (F_2=2)$
                $\times 6) + (((F_0=0) \wedge (F_1 \leq 3)) \times 3)$                6
          $S_{0,1} \leftarrow ((F_1, F_2), (I_0, F_0), (F_0, F_1), (F_1, /3; F_2 > 3; 4/),$
                $(F_1, F_2), (F_3, F_4), (/F_3; F_3=0; 6/, /F_4;$
                $F_3=0; 0/))_i$                                                    7
          $j \leftarrow {}^{i}D^{S_0}_{S_1}$                                        8
          $i \leftarrow j_{I_0}$                                                    8a
          $n \leftarrow N^i$                                                        9
          $\rightarrow (11, 13, 14, 16, 17)_{n_0}$                                  10
DOP       $ADC(F_1; F_2; I_0; a_1)$                                                 11
      =   $1 : \vee/e$                                                             12
          $a_1 \leftarrow F_2$                                                      13
```

Figure 20. The Processor System Program

```
SOP      $ADC(F_3;F_4;I_0;a_2)$                                                14
      =  $1 : \vee/e$                                                          15
B        $a_1 \leftarrow (\perp R^7) + 2\times((\perp\omega^7/I) - I_8\times 2^7)$   16
         EXEC                                                                  17
         $e_4 \leftarrow p_{11}$                                               18
         $0 : \vee/e$                                                          19
         $h_0 \leftarrow 1$                                                    20
         $0 : \vee/h$                                                          21  =
         $MAC^1(\perp R^6,4, s;p, R^7)$                                        22
         $MAC^1((4,8,4,28,12, \omega^{18}/\alpha^{34}/U)_{((e,h_1)/\iota^0)_0},$
             $4,f;R^7,p)$                                                      23
         $h_{(h/\iota^0)_0} \leftarrow 0$                                      24
```

Figure 20. (Continued)

TABLE VIII

"PROC" PROGRAM SEGMENTS

Lines	Function	Major State
1- 4	Instruction fetch	FETCH
5- 9	Instruction interpretation	
10-16	Effective address computation	SOURCE DESTINATION
17-18	Instruction execution	EXECUTE
19-24	Trap interrupt service	SERVICE

The processor can be described in terms of five major states. In the FETCH major state the instruction is fetched from memory. SOURCE and DESTINATION states obtain the source and destination operands, respectively. In the EXECUTE state the machine performs the action specified by the instruction, and the in SERVICE state, interrupts and traps are handled. In every major state the machine performs several minor operations, and a minor state is associated with each operation. For example, the FETCH major state consists of the minor operations: (1) retrieve the instruction from memory; (2) update the program counter; (3) load the instruction register; and (4) decode the instruction.

Instruction Fetch

The first step in program execution is to fetch the instruction from memory. In order to prepare for instruction fetch, the exceptions vector is initialized to zero (line 1). The 2-byte instruction is fetched from memory at the address given by the program counter, and placed in the instruction register (line 2). The program counter is incremented by 2 (line 3) and in case of any exceptions during instruction fetch, control branches to line 19. Exceptions during fetch may be due to errors in addressing.

Instruction Interpretation

To determine the operation specified by the instruction, the instruction is decoded next. The instruction is divided into five fields, specified by the five components of the vector F (line 5). Instruction interpretation is done on the basis of these fields. The instructions are divided into five different classes, and i takes the value of the class of the current instruction (line 6). Table IX summarizes the five classes.

TABLE IX

INSTRUCTION CLASSES

Class	i
S : Single operand	0,4
D : Double operand	1
R : Register-source/destination	2
B : Branch	3
O : Operate	5,6

The components of the selection vector, S, take on values of the fields depending on i (line 7). Lines 8,9 interpret the instruction by selecting a row N^i from the navigation matrix N, (Table X), to specify the vector n used in subsequent control of the instruction execution. The row of N selected, is determined by an element of a particular decoding matrix D, Figure 24, specified by the instruction class i, and the selection vector S. For example, if the instruction is 020314, the five fields F_0-F_4 have the values 2,0,3,1,4 respectively. Therefore, at line 6, i is assigned the value 1. Consequently, S_0, S_1 take on the values of I_0 and F_0 at line 7. The lower diagonal entry (I_0=0) in the second column (S_1=2) of the zeroth row (I_0=0) in the ^1D decoding matrix gives the instruction, CMP. The entry in the element of the matrix, 41 in this case, gives the row in the navigation matrix N.

Effective-address Computation

Address computation is done by the defined operation ADC. Computation depends on the instruction class. For double operand instructions, the address of the source operand (line 11) and the address of the destination operand (line 14) need to be calculated. For single operand instructions, only the address of the destination (line 14) is required. For the Register-Source/destination class of instructions, the register used (line 13) and the destination operand address (line 14) need to be computed. In the branch instruction, address calculation is done in line 16. Operate instructions do not need an operand and are executed immediately after instruction decode. Any exceptions during address computation abort execution (line 15).

Instruction Execution

Execution is done by the EXEC defined operation. The entry point in EXEC for any instruction depends on the component n_1, This is indicated informally by giving the instruction mnemonics on the left hand margin of the EXEC routine. Execution also may involve setting of the condition code. If the trap bit is set after execution, the exception is entered in e (line 18).

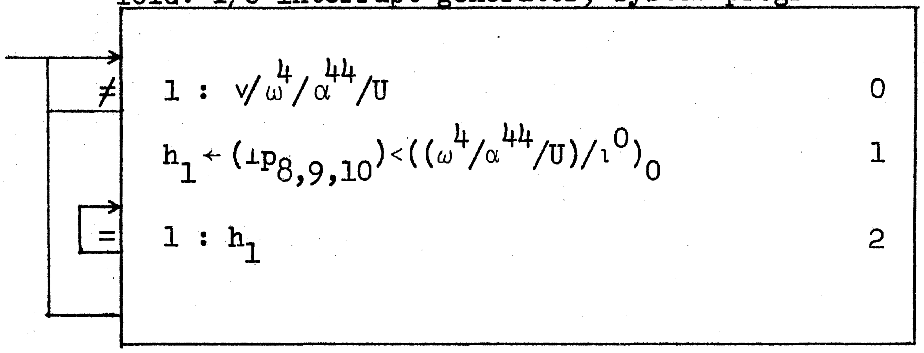
Interrupt Service

Servicing of exceptions is given priority over I/O interrupt service. In case of any exception the bit (0 for exceptions, 1 for I/O interrupt) in the interrupt holder h is set (line 20). The interrupt service sequence is initiated, if at least one interrupt is pending (line 21). The sequence consists of pushing the processor status word (PS) and the program counter (PC) onto the processor stack (line 22), and loading the new PS and PC from the interrupt vector address (line 23). The interrupt vector address is selected from one of the six fixed locations in memory. The interrupt vector address of the peripheral device, is obtained from the address lines of the Unibus, when the processor accepts the request. The element of h, which caused the interrupt is reset.

Input/Output Interrupts

The IOIG system program, Figure 21, determines presence of interrupt requests by peripheral devices, and sets the bit in the interrupt holder, h, accordingly, line 1. The dwell at line 0 checks for interrupts on the Unibus bus request line BR(7:4). When an interrupt request is detected, the processor priority is compared against the request level, line 1. If the processor priority is less than the request level, the bit in the interrupt holder is set. This prohibits further interrupts until a new program counter and a processor status word is loaded and the interrupt holder bit reset (PROC lines 22-24).

IOIG: I/O interrupt generator, system program



```
≠  $1 : \vee/\omega^{4}/\alpha^{44}/U$                                   0
   $h_1 \leftarrow (\perp p_{8,9,10}) < ((\omega^{4}/\alpha^{44}/U)/\iota^{0})_0$   1
=  $1 : h_1$                                                          2
```

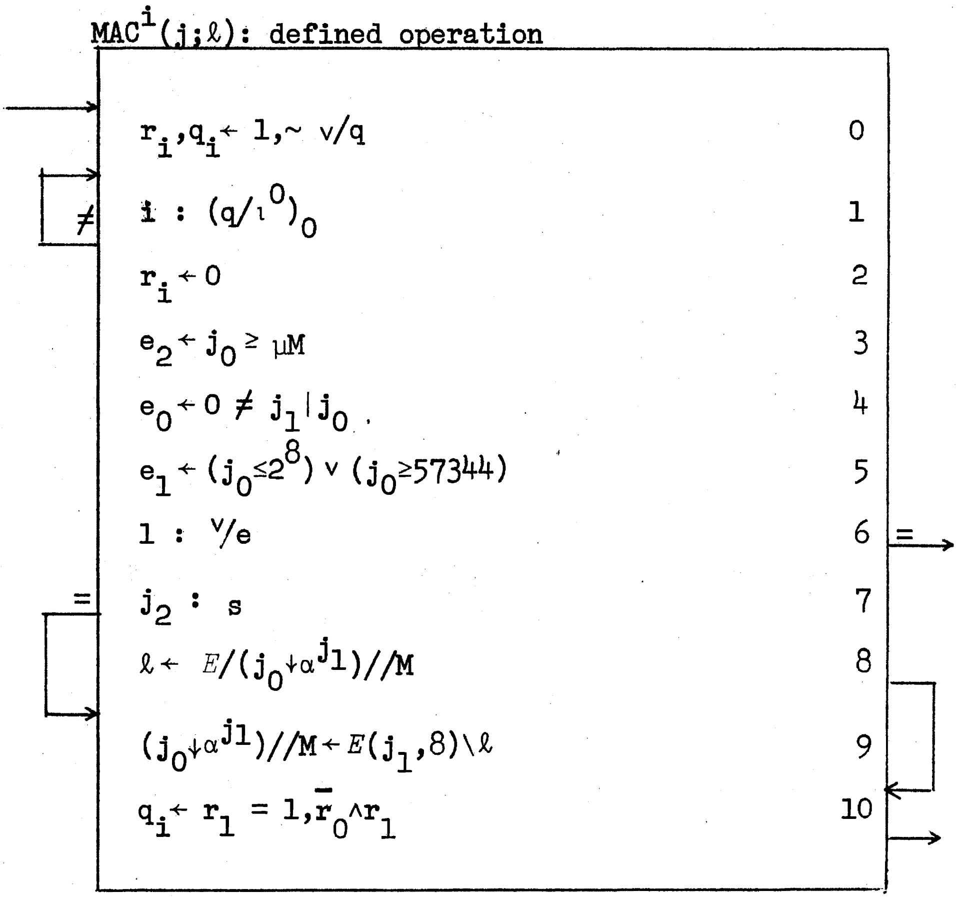
Figure 21. Input/Output Interrupt Generator

Memory Access Program

The MAC operation, Figure 22, fetches or stores a specified number of bytes from the memory at a given address. The general form of the operation is $MAC^i(j;\ell)$, where i specified the device requesting access; j is a three component vector specifying the address in memory (j_0), number of bytes to be accessed (j_1) and type of operation (store: j_2=s; fetch: j_2=f), respectively; ℓ specifies the vector into/ from which the accessed data is to be stored/fetched.

All data transfer operations are carried on the 56 lines of the Unibus. The addresses, j_0, are communicated over the 18 address lines, the data, contents of ℓ, are placed on the 16 data lines and the type of operation is determined by the signal on the two Unibus control lines. Since memory is always a slave, a store operation, j_2=s, transfers data from master to slave and corresponds to the DATO operation. Conversely, a fetch, j_2=f, requests data from a slave and corresponds to the DATI operation. A description of these operations and Unibus transactions is given in the Peripherals and Interfacing Handbook (10).

Access to memory can be for a nonprocessor request (NPR) by one of the peripheral devices (i=0) or by the CPU (i=1). The request for service is entered in the bus request vector, r, and in the queue if it is empty (line 0). The queue discipline is on a priority basis with the NPR having greater priority than the CPU. The program dwells at line 1 until i is recognized as the first nonzero entry in the queue Requests that are not entered at line 0 are entered in



```
$MAC^{i}(j;\ell)$: defined operation

$r_i, q_i \leftarrow 1, \sim \vee/q$                              0
$i : (q/\iota^0)_0$                                   1
$r_i \leftarrow 0$                                         2
$e_2 \leftarrow j_0 \geq \mu M$                                3
$e_0 \leftarrow 0 \neq j_1 | j_0$ .                             4
$e_1 \leftarrow (j_0 \leq 2^8) \vee (j_0 \geq 57344)$                     5
$1 : {}^{\vee}/e$                                       6
$j_2 : s$                                          7
$\ell \leftarrow E/(j_0 \downarrow \alpha^{j_1} 1)//M$                      8
$(j_0 \downarrow \alpha^{j_1} 1)//M \leftarrow E(j_1, 8)\backslash \ell$                9
$q_i \leftarrow r_1 = 1, \bar{r}_0 \wedge r_1$                          10
```

Figure 22. Memory Access Operation

line 10. After the request has been honored, the entry in the request vector is blanked out (line 2).

Any form of exceptions are noted (line 3,4,5), and entered in the exceptions vector, e. Time out errors (line 3), odd addressing errors (line 4), errors due to addressing reserved memory locations (line 5) are noted. If no exceptions occur, a fetch (line 8) or store (line 9) is performed.

Address Computation

The ADC operation, Figure 23, is used for effective address calculation of the operands. The general form for ADC is (m;r;b;a) where m is the mode of addressing (one of the possible eight), r is the register used for addressing, b gives the type of instruction byte (b=1) or word (b=0) and 'a' is the address returned by the operation.

There are basically four types of addressing. These are register, auto-increment, auto-decrement and indexed. Each type can be direct or deferred. In the direct mode the register used in the addressing contains the operand. In the deferred mode, the contents of the register contain the address of the operand. When the program counter is used as the register in addressing, variations of auto-increment, auto-increment-deferred, index and index-deferred are called immediate, absolute, relative and relative-deferred, respectively.

```
                ADC(m;g;b;a) : defined operation

                $\rightarrow(1,2,3,3,8,8,13,13)_m$                       0
REG             $a \leftarrow g$                                         1  $\rightarrow$
REG-DEF         $a \leftarrow \perp R^g$                                 2  $\rightarrow$
AUTO-INC
AUTO-INC-DEF    $a \leftarrow \perp R^g$                                 3
                $R^g \leftarrow (16)\top(2,1)_{(g\leq5)\wedge b} + \perp R^g$   4
                $m : 2$                                                  5  $\xrightarrow{=}$ AUTO-
                                                                            INC
                $MAC^1(a,2,f;u)$                                         6
                $a \leftarrow \perp u$                                   7  $\rightarrow$ AUTO-
                                                                            INC-
                                                                            DEF
AUTO-DEC
AUTO-DEC-DEF    $R^g \leftarrow (16)\top(\perp R^g)-(2,1)_b$             8
                $a \leftarrow \perp R^g$                                 9
                $m : 4$                                                  10 $\xrightarrow{=}$ AUTO-
                                                                            DEC
                $MAC^1(a,2,f;u)$                                         11
                $a \leftarrow \perp u$                                   12 $\rightarrow$ AUTO-
                                                                            DEC-
                                                                            DEF
INDX,
INDX-DEF        $MAC^1(\perp R^7,2,f,;u)$                                13
                $R^7 \leftarrow (16)\top 2 + \perp R^7$                  14
                $a \leftarrow (\perp u) + \perp R^g$                     15
                $m : 6$                                                  16 $\xrightarrow{=}$ INDX
                $MAC^1(a,2,f;u)$                                         17
                $a \leftarrow \perp u$                                   18 $\rightarrow$ INDX-
                                                                            DEF
```

Figure 23. Address Computation Operation

0	1	2	3	4	5	6	7	
CLRB 38 37 CLR	COMB 44 43 COM	INCB 51 50 INC	DECB 46 45 DEC	NEGB 60 59 NEG	ADCB 2 1 ADC	SBCB 71 70 SBC	TSTB 83 82 TST	5
RORB 66 65 ROR	ROLB 64 63 ROL	ASRB 9 8 ASR	ASLB 7 6 ASL	55 MARK			80 SXT	6

0_D

(a) Single Operand Instructions

0	1	2	3	4	5	6	7	
	56 MOV	41 CMP	21 BIT	17 BIC	19 BIS	3 ADD		0
	MOVB 57	CMPB 42	BITB 32	BICB 18	BISB 20	SUB 78		1

1_D

(b) Double Operand Instructions

0	1	2	3	4	5	6	7	
	58 MUL	47 DIV	4 ASH	5 ASHC	85 XOR		77 SOB	7

2_D

(c) Register-Source/Destination Instructions

Figure 24. Instruction Decoding Matrices

	3	4
0	BPL 29	BMI 27 31 BR
1	BHI 15 28 BNE	BLOS 25 12 BEQ
2	BVC 32 13 BGE	BVS 33 26 BLT
3	BCC 10 14 BGT	BCS 11 23 BLE
4	EMT 48 54 JSR	TRAP 81 54 JSR

3_D

i = 3

(d) Branch Instructions

	0	1	2
0	53 JMP		79 SWAB

4_D

i = 5

(e) Single Operand Instructions

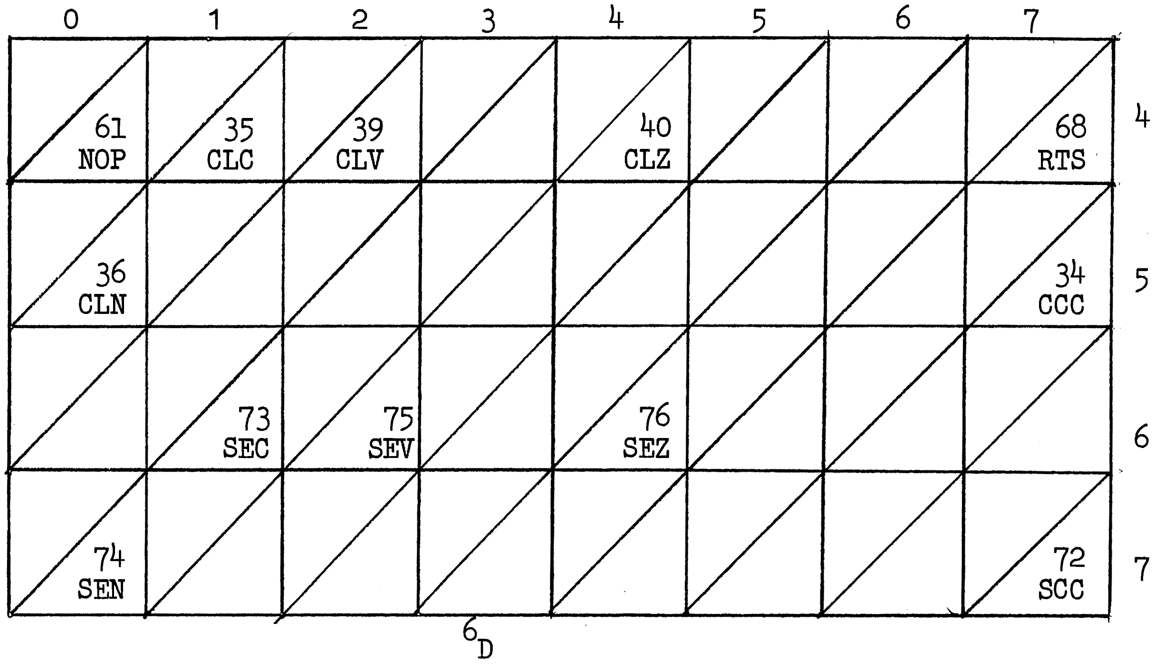
	0	1	2	3	4	5	6	7
0	49 HALT	84 WAIT	67 RTI	30 BPT	52 IOT	62 RESET	69 RTT	

5_D

i = 6

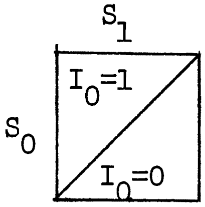
(f) Operate Instructions

Figure 24. (Continued)



	0	1	2	3	4	5	6	7
4	61 NOP	35 CLC	39 CLV		40 CLZ			68 RTS
5	36 CLN							34 CCC
6		73 SEC	75 SEV		76 SEZ			
7	74 SEN							72 SCC

(g) Operate Instructions



LEGEND

Figure 24. (Continued)

In the register mode (line 1) the register number is returned, and in the register deferred mode the contents of the register give the operand address (line 2). In auto-increment the contents of the register are incremented by (2,1), depending on the type of instruction (b), after using its contents as the address of the operand. In auto-increment-deferred the register contents give the address of the address of the operand (lines 6,7). In auto-decrement and auto-decrement-deferred the register contents are first decremented and then used as the address (line 9) or address of the address (line 11, 12) of the operand. In indexed and index-deferred the contents of the specified register (r) are added (line 15) to the contents of the word in memory after the instruction (line 13,14) to give the address (line 15) or the address of the address (line 17,18) of the operand.

Instruction Execution

The entry point in the EXEC, Figure 25, routine for an instruction is determined by n_2 (line a_0). Execution also involves setting up of the condition codes (if necessary) after the execution.

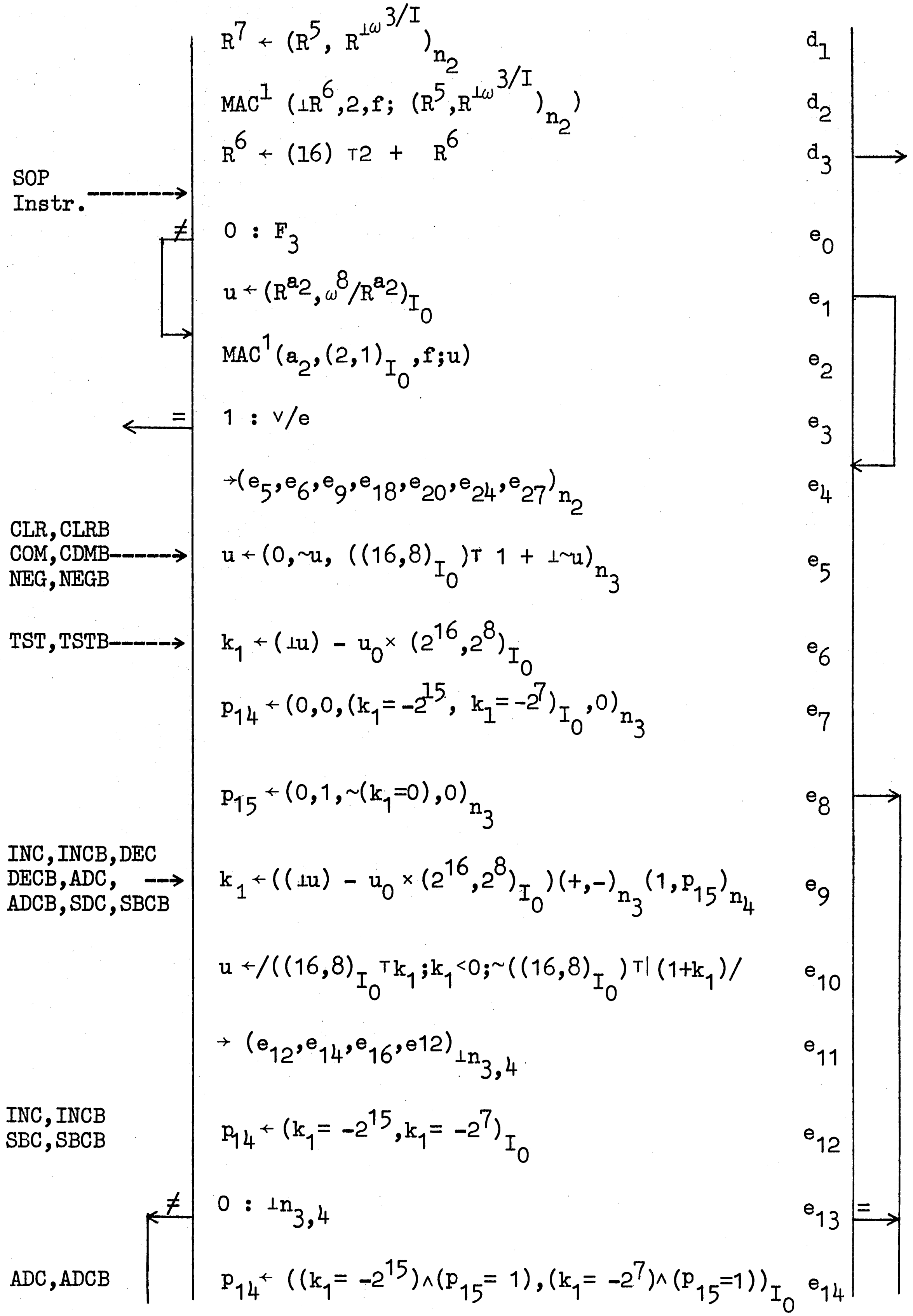
Lines a_1-a_8

The TRAP, BPT, EMT, IOT instruction enter at a_1. In each case the program status word and the Program counter get stacked (lines a_1, a_2) and the new PSW and PC are fetched from preassigned positions in memory (lines a_4,a_5). In RTI, RTT, the PC and PSW which were stacked are loaded back (lines a_6,a_7) and the stack pointer (R_6) incremented

EXEC: definer operation

	$\rightarrow n_1$	a_0	
TRAP,BPT EMT,IOT →	$MAC^1(\perp R^6,2,s;p)$	a_1	
	$MAC^1((\perp R^6)-2,2,s;R^7)$	a_2	
	$R^6 \leftarrow (16)\top(\perp R^6)-4$	a_3	
	$MAC^1((12,16,24,28)_{n_2},2,f;R^7)$	a_4	
	$MAC^1((14,18,26,30)_{n_2},2,f;p)$	a_5	→
RTI,RTT →	$MAC^1(\perp R^6,2,f;R^7)$	a_6	
	$MAC^1((2 + \perp R^6),2,f;p)$	a_7	
	$R^6 \leftarrow (16)\top 4 + \perp R^6$	a_8	→
CLC,CLN,CLV CLZ,SEC,SEN → SEV,SEZ	$p_{n_2} \leftarrow (0,1)_{n_2}$	b_0	→
SCC,CCC →	$p_{12,13,14,15} \leftarrow (\bar{\epsilon}(4),\epsilon(4))_{n_3}$	b_1	→
BR,JMP →	$R^7 \leftarrow (16)\top(a_1,a_2)_{n_2}$	c_0	→
BPL,BCC/BHIS BNE,BVC →	$R^7 \leftarrow /R^7;\sim p_{(12,13,14,15)_{n_2}};(16)\top a_1/$	c_1	→
BEQ,BMI,BVS BCS/BLO →	$R^7 \leftarrow /R^7;\ p_{(12,13,14,15)_{n_2}};(16)\top a_1/$	c_2	→
BLT,BGE →	$R^7 \leftarrow /(16)\top a_1;\sim p_{12}(=,\neq)_{n_2}p_{14};R^7/$	c_3	→
BET,BLE →	$R^7 \leftarrow /(16)\top a_1;p_{13} \vee\sim (p_{12}(=,\neq)_{n_2}p_{14});R^7/$	c_4	→
BHI,BLOS →	$R^7 \leftarrow /R^7;(p_{13}\vee p_{15})=(0,1)_{n_2};(16)\top a_1/$	c_5	→
MARK →	$R^6 \leftarrow (16)\top(\perp R^6) + 2\times\perp\omega^6/I$	d_0	

Figure 25. EXEC Routine


SOP Instr.
≠
=
CLR, CLRB COM, CDMB NEG, NEGB
TST, TSTB
INC, INCB, DEC DECB, ADC, ADCB, SDC, SBCB
INC, INCB SBC, SBCB
≠
=
ADC, ADCB

$R^7 \leftarrow (R^5, R^{\perp\omega^{3}/I})_{n_2}$ d_1

$MAC^1 (\perp R^6, 2, f; (R^5, R^{\perp\omega^{3}/I})_{n_2})$ d_2

$R^6 \leftarrow (16) \top 2 + R^6$ d_3

$0 : F_3$ e_0

$u \leftarrow (R^{a2}, \omega^8/R^{a2})_{I_0}$ e_1

$MAC^1(a_2, (2,1)_{I_0}, f; u)$ e_2

$1 : \vee/e$ e_3

$\rightarrow(e_5, e_6, e_9, e_{18}, e_{20}, e_{24}, e_{27})_{n_2}$ e_4

$u \leftarrow (0, \sim u, ((16,8)_{I_0}) \top 1 + \perp\sim u)_{n_3}$ e_5

$k_1 \leftarrow (\perp u) - u_0 \times (2^{16}, 2^8)_{I_0}$ e_6

$p_{14} \leftarrow (0, 0, (k_1 = -2^{15}, k_1 = -2^7)_{I_0}, 0)_{n_3}$ e_7

$p_{15} \leftarrow (0, 1, \sim(k_1 = 0), 0)_{n_3}$ e_8

$k_1 \leftarrow ((\perp u) - u_0 \times (2^{16}, 2^8)_{I_0})(+,-)_{n_3}(1, p_{15})_{n_4}$ e_9

$u \leftarrow /((16,8)_{I_0} \top k_1; k_1 < 0; \sim((16,8)_{I_0}) \top | (1 + k_1)/$ e_{10}

$\rightarrow (e_{12}, e_{14}, e_{16}, e12)_{\perp n_{3,4}}$ e_{11}

$p_{14} \leftarrow (k_1 = -2^{15}, k_1 = -2^7)_{I_0}$ e_{12}

$0 : \perp n_{3,4}$ e_{13}

$p_{14} \leftarrow ((k_1 = -2^{15}) \wedge (p_{15} = 1), (k_1 = -2^7) \wedge (p_{15} = 1))_{I_0}$ e_{14}

Figure 25. (Continued)

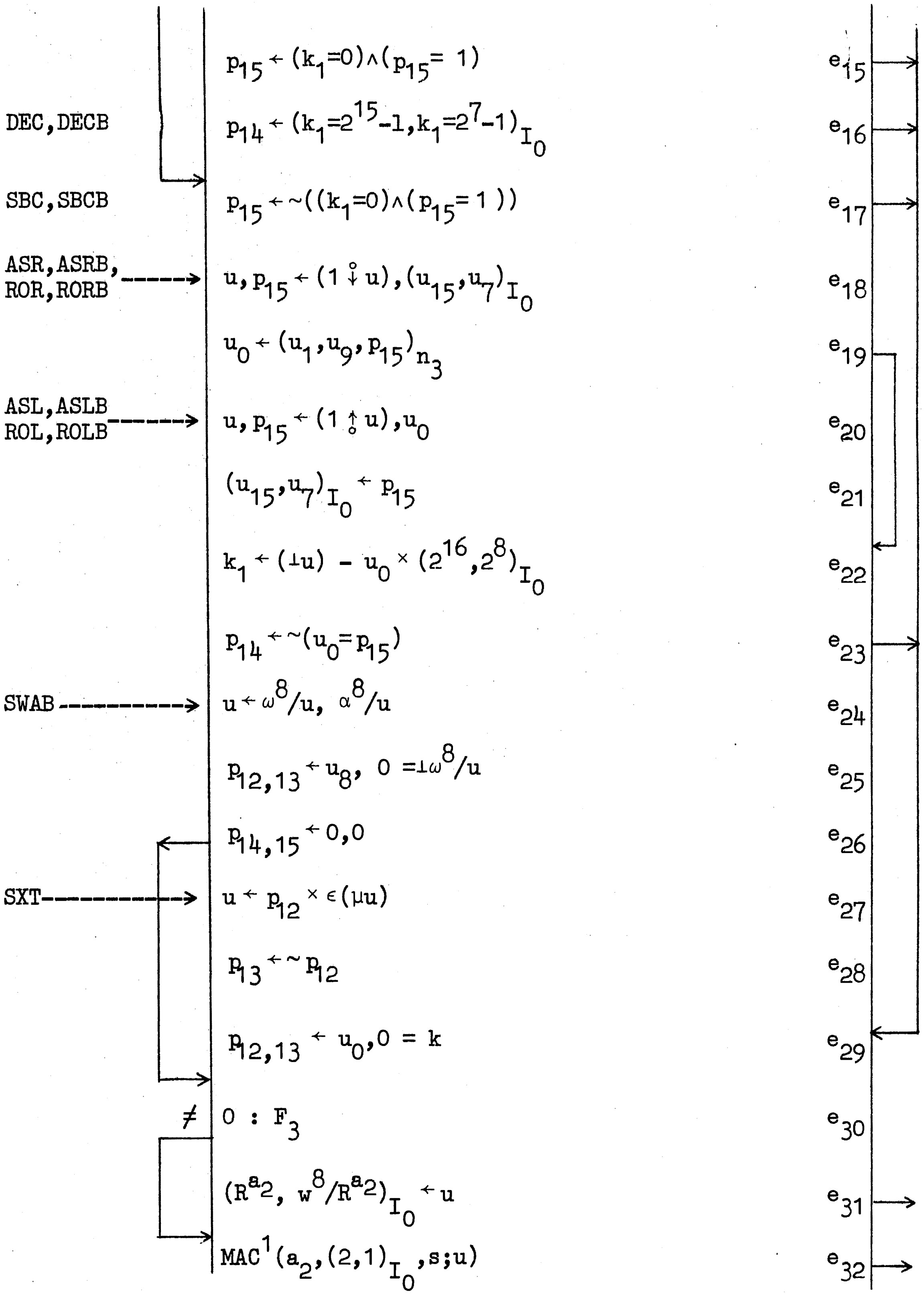
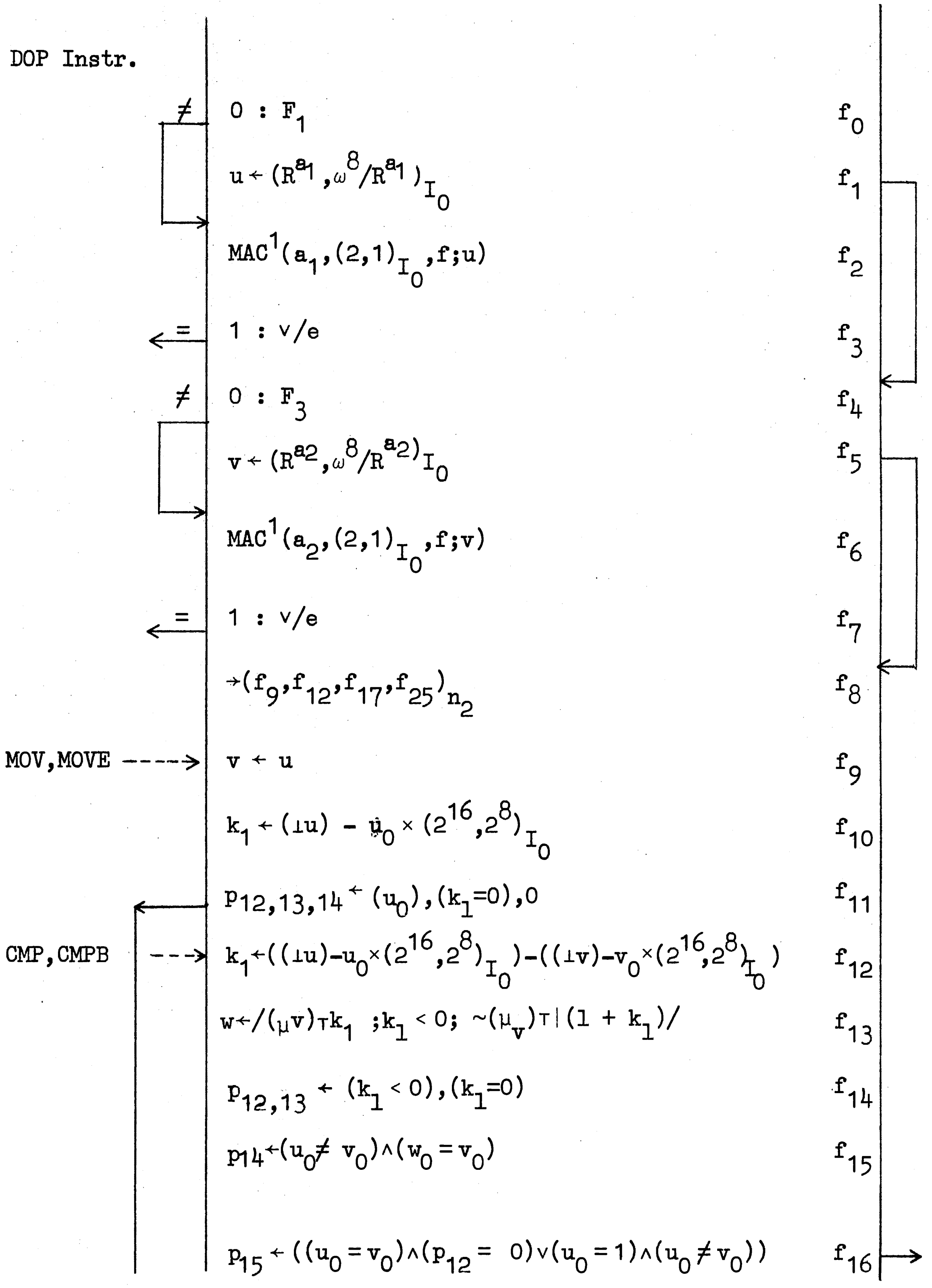

$p_{15} \leftarrow (k_1=0)\wedge(p_{15}=1)$ e_{15}
DEC,DECB $p_{14} \leftarrow (k_1=2^{15}-1, k_1=2^7-1)_{I_0}$ e_{16}
SBC,SBCB $p_{15} \leftarrow \sim((k_1=0)\wedge(p_{15}=1))$ e_{17}
ASR,ASRB, ROR,RORB $u, p_{15} \leftarrow (1 \overset{\circ}{\downarrow} u), (u_{15}, u_7)_{I_0}$ e_{18}
$u_0 \leftarrow (u_1, u_9, p_{15})_{n_3}$ e_{19}
ASL,ASLB ROL,ROLB $u, p_{15} \leftarrow (1 \underset{\circ}{\uparrow} u), u_0$ e_{20}
$(u_{15}, u_7)_{I_0} \leftarrow p_{15}$ e_{21}
$k_1 \leftarrow (\perp u) - u_0 \times (2^{16}, 2^8)_{I_0}$ e_{22}
$p_{14} \leftarrow \sim(u_0 = p_{15})$ e_{23}
SWAB $u \leftarrow \omega^8/u,\ \alpha^8/u$ e_{24}
$p_{12,13} \leftarrow u_8,\ 0 = \perp\omega^8/u$ e_{25}
$p_{14,15} \leftarrow 0,0$ e_{26}
SXT $u \leftarrow p_{12} \times \epsilon(\mu u)$ e_{27}
$p_{13} \leftarrow \sim p_{12}$ e_{28}
$p_{12,13} \leftarrow u_0, 0 = k$ e_{29}
$\neq$ $0 : F_3$ e_{30}
$(R^{a_2}, w^8/R^{a_2})_{I_0} \leftarrow u$ e_{31}
$MAC^1(a_2, (2,1)_{I_0}, s; u)$ e_{32}

Figure 25. (Continued)



```
DOP Instr.
  ≠   $0 : F_1$                                                   $f_0$
      $u \leftarrow (R^{a_1}, \omega^8/R^{a_1})_{I_0}$                $f_1$
      $MAC^1(a_1, (2,1)_{I_0}, f; u)$                                $f_2$
  =   $1 : \vee/e$                                                $f_3$
  ≠   $0 : F_3$                                                   $f_4$
      $v \leftarrow (R^{a_2}, \omega^8/R^{a_2})_{I_0}$                $f_5$
      $MAC^1(a_2, (2,1)_{I_0}, f; v)$                                $f_6$
  =   $1 : \vee/e$                                                $f_7$
      $\rightarrow (f_9, f_{12}, f_{17}, f_{25})_{n_2}$               $f_8$
MOV,MOVE ---->  $v \leftarrow u$                                  $f_9$
      $k_1 \leftarrow (\perp u) - u_0 \times (2^{16}, 2^8)_{I_0}$     $f_{10}$
      $p_{12,13,14} \leftarrow (u_0), (k_1 = 0), 0$                  $f_{11}$
CMP,CMPB --->  $k_1 \leftarrow ((\perp u) - u_0 \times (2^{16}, 2^8)_{I_0}) - ((\perp v) - v_0 \times (2^{16}, 2^8)_{I_0})$   $f_{12}$
      $w \leftarrow /(\mu v) \top k_1 \; ; k_1 < 0; \; \sim(\mu_v) \top |(1 + k_1)/$   $f_{13}$
      $p_{12,13} \leftarrow (k_1 < 0), (k_1 = 0)$                    $f_{14}$
      $p_{14} \leftarrow (u_0 \neq v_0) \wedge (w_0 = v_0)$           $f_{15}$
      $p_{15} \leftarrow ((u_0 = v_0) \wedge (p_{12} = 0) \vee (u_0 = 1) \wedge (u_0 \neq v_0))$   $f_{16}$
```

Figure 25. (Continued)

ADD, SUB	$k_1 \leftarrow ((\perp v)-v_0\times(2^{16},2^8)_{I_0})(+,-)_{n_3}((\perp u)-u_0\times(2^{16},2^8)_{I_0})$	f_{17}
	$w \leftarrow /(\mu v)\top k_1;k_1<0;\sim(\mu v)\top\vert(1+k_1)/$	f_{17a}
	$\rightarrow(f_{19},f_{21})_{n_3}$	f_{18}
ADD	$p_{14}\leftarrow(u_0 = v_0)\wedge(u_0 \neq w_0)$	f_{19}
	$p_{15}\leftarrow(u_0 = v_0)\wedge(w_0= 1)\vee(w_0=0)\wedge(u_0\neq v_0)$	f_{20}
	$p_{14}\leftarrow(u_0 \neq v_0)\wedge(u_0 = w_0)$	f_{21}
	$p_{15}\leftarrow((u_0=v_0)\wedge(w_0=0)\vee(v_0=1)\wedge(u_0=v_0))$	f_{22}
	$v \leftarrow w$	f_{23}
	$k_1 \leftarrow (\perp v) - v_0 \times (2^{16},2^8)_{I_0}$	f_{24}
BIS,BIC BIT -->	$(w,v,v)_{n_3} \leftarrow ((u\wedge v), (\sim u\wedge v), (u\vee v))_{n_3}$	f_{25}
	$k_1 \leftarrow (\perp(w,v,v)_{n_3}) - (w_0,v_0,v_0)_{n_3}\times(2^{16},2^8)_{I_0}$	f_{26}
	$p_{14}\leftarrow 0$	f_{27}
	$p_{12,13}\leftarrow v_0,k_1 = 0$	f_{28}
≠	$0 : F_3$	f_{29}
DOP Instr. Store result	$(R^{a1},\omega^8/R^{a2})_{I_0\wedge F_0\neq 6} \leftarrow v$	f_{30}
	$MAC^1(a_2,(2,1)_{I_0\wedge F_0\neq 6},s;v)$	f_{31}

Figure 25. (Continued)

R-D Instr: MUL,DIV,ASH,ASHC,XOR

		$\rightarrow(g_1,g_2,g_3,g_1,g_4)_{n_2}$	g_0	
		$c \leftarrow 2\|a_1$	g_1	
	≠	$1 : e_0 \leftarrow 0 \neq 2\|a_1$	g_2	≡
		$c \leftarrow 1$	g_3	
		$0 : F_3$	g_4	≠
		$v \leftarrow R^{a2}$	g_5	
		$MAC^1(a_2,2,f;v)$	g_6	
		$\rightarrow(g_8,g_8,g_{25},g_{25},g_{39})_{n_2}$	g_7	
MUL,DIV		$k_2 \leftarrow (\perp v) - v_0 \times 2^{16}$	g_8	
		$\rightarrow(g_{10},g_{11})_{n_2}$	g_9	
		$k_1 \leftarrow (\perp R^{a1}) - R_0^{a1} \times 2^{16}$	g_{10}	
		$k_1 \leftarrow (\perp R^{a1},R^{a1+1}) - R_0^{a1} \times 2^{32}$	g_{11}	
		$k_3 \leftarrow k_1(\times,\div)_{n_2}\ k_2$	g_{12}	
		$\rightarrow(g_{14},g_{19})_{n_2}$	g_{13}	
		$\rightarrow(g_{15},g_{16})_c$	g_{14}	
MUL		$R^{a1},R^{a1+1} \leftarrow /(32)\top k_3;k_3<0;\sim(32)\top\|(1+k_3)/$	g_{15}	
		$R^{a1} \leftarrow /(16)\top 2^{16}\|k_3;(2^{16}\|k_3)<0;\sim(16)\top\|(1+2^{16}\|k_3)/$	g_{16}	
		$p_{14,15} \leftarrow 0,(k_3<-2^{15})\vee(k_3\geq 2^{15} - 1)$	g_{17}	

Figure 25. (Continued)

MUL	$p_{12,13} \leftarrow (k_3 < 0),(k_3 = 0)$	g_{18}	→
DIV ← =	$1 : p_{14} \leftarrow (\lvert k_1) > (\lvert k_2)$	g_{19}	
	$R^{a_1} \leftarrow /(16)\top\lfloor k_3;\sim(k_3<0);\sim(16)\top \mid (1+\lceil k_3)/$	g_{20}	
	$p_{12,13} \leftarrow k_3<0,\ k_3 = 0$	g_{21}	
	$k_3 \leftarrow (\lvert k_2)\mid(\lvert k_1)$	g_{22}	
	$R^{a_1+1} \leftarrow /(16)\top k_3;\ k_1<0;\sim(16)\top\mid(1+k_3)/$	g_{23}	
DIV	$p_{14,15} \leftarrow k_2 = 0$	g_{24}	→
ASH,ASHC	$0 : k_1 \leftarrow (\bot\omega^6/v) - v_{10} \times 2^6$	g_{25}	$\xrightarrow{=}$
	$(g_{27}, g_{28})_{n_3}$	g_{26}	
	$u \leftarrow R^{a_1}$	g_{27}	
	$u \leftarrow ((R^{a_1}, R^{a_1+1}), R^{a_1})_c$	g_{28}	
	$\rightarrow (g_{34}, g_{30})_{k_1 < 0}$	g_{29}	
right	$k_1 \leftarrow \lvert k_1$	g_{30}	
	$p_{14,15} \leftarrow 0, u_{(\mu u)-k_1}$	g_{31}	
	$\bar{\alpha}^{1}\tilde{/}u \leftarrow k_1 \overset{\circ}{\downarrow} \bar{\alpha}^{1}/u$	g_{32}	
	$\alpha^{k_1+1}/u \leftarrow u_0 \times \epsilon(k_1+1)$	g_{33}	→
	$p_{15}, k_2 \leftarrow u_{(k_1-1)}, u_0$	g_{34}	
left	$u \leftarrow k_1 \underset{\circ}{\uparrow} u$	g_{35}	
	$p_{14} \leftarrow k_2 \neq u_0$	g_{36}	

Figure 25. (Continued)

		$p_{12,13} \leftarrow u_0,\ (\perp u) = 0$	g_{37}	
ASH,ASHC		$((R^{a_1}, R^{a_1+1}),\ R^{a_1})_{n_3 \wedge c} \leftarrow u$	g_{38}	
XOR		$u \leftarrow R^{a_1}$	g_{39}	
		$v \leftarrow u \neq v$	g_{40}	
		$p_{12,13,14} \leftarrow (v_0), ((\perp v) = 0), 0$	g_{41}	
	≠	$0 : F_3$	g_{42}	
		$R^{a_2} \leftarrow v$	g_{43}	
XOR		$MAC^1(a_2, 2, s; v)$	g_{44}	
SOB,JSR		$\rightarrow (h_1, h_4)_{n_2}$	h_0	
SOB		$k_1 \leftarrow ((\perp R^{a_1}) - R_0^{a_1} \times 2^{16}) - 1$	h_1	
		$R^{a_1} \leftarrow /(16)\top k_1;\ k_1 < 0; \sim (16)\top \vert (1+k_1)/$	h_{1a}	
		$0 : k_1$	h_2	=
		$R^7 \leftarrow (16)\top(\perp R^7) - 2 \times \perp \omega^6 / I$	h_3	
JSR	≠	$0 : F_3$	h_4	
		$v \leftarrow R^{a2}$	h_5	
		$MAC^1(a_2, 2, f; v)$	h_6	

Figure 25. (Continued)

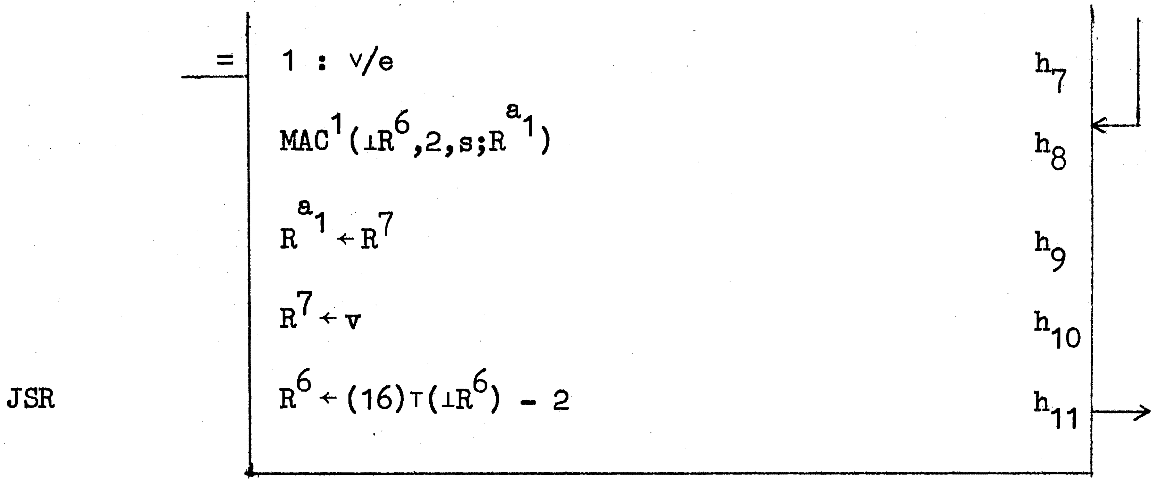


Figure 25. (Continued)

(line a_8). None of these instructions affect the condition codes.

Lines b_0-b_1

Instructions which clear and set the condition codes are executed here. Condition codes are cleared or set depending on n_3 (line b_0). For SCC and CCC all four condition code bits are set or cleared depending on n_3.

Lines c_0-c_5

Branch Instructions get executed here. The address to which control transfers (PROC 15) is placed in the PC (R^7) depending on the condition codes.

Lines d_0-d_3

For the MARK instruction the stack pointer is reinitialized by adding the value of the last six bits of I (line d_0). From then on execution of MARK and RTS is identical. The contents of the register specified are placed in the PC and the word on the top of the stack is placed in the register (line d_2). The stack pointer is set to point to the top element.

Lines e_0-e_{32}

This is the entry point for single operand instructions. The destination operand is fetched from the register specified in the instruction if the mode is zero (line e_1) or from memory (line e_2). Execution is aborted in case of exceptions (line e_3). The destination is cleared, complemented or negated (line e_5) and two of the four CC bits set for CLR, COM, NEG instructions. The result is stored back at the destination address (lines e_{30}- e_{32}).

In shift and rotate instructions the destination is shifted/ rotated right (lines e_{18} -e_{19}) or left (lines e_{20} - e_{21}). In SWAB the bytes of the destination are sqapped (line e_{24}).

Lines f_0-f_{21}

The double operand instructions are executed here. The two operands are fetched from the registers specified if the mode is zero (lines f_1,f_5) or from memory (lines f_2,f_6). Execution proceeds if there are no exceptions (lines f_3,f_7). After the execution the result is stored back at the destination address (lines f_{29}-f_{31}).

Lines g_0-g_{44}

The Register-Destination type instructions have an entry point at g_0. For MUL and ASHC the type of register used (even or odd) is determined (line g_1). If an odd register is used in the DIV instruction, the execution is aborted and the specification noted (line g_2). The destination operand v is fetched from a register (if mode is zero (line g_5)) or from memory (line g_6). In MUL and DIV the destination and source operands are treated as two's complement numbers (lines g_8,g_{10},g_{11}). In the DIV instruction the contents of the pair of even and odd registers is treated as a two's complement number (line g_{11}). If an even register is used in MUL, the entire product is stored in a pair of registers (line g_{15}). Else only the last 16 bits are stored back in the odd register specified (line g_{16}).

In DIV the results are not stored in case of an overflow (line g_{19}). The quotient is stored in the even register (line g_{20}) and the remainder in the odd register (line g_{23}).

In the arithmetic shift operations the last 6 bits of the destination operand are treated as a signed two's complement number giving the amount of shift (line g_{25}). A positive number indicates left shift (lines g_{35-37}) and a negative number indicates right shift (lines g_{30-34}). The source operand is fetched from the register specified in the instruction g_{27}. In ASHC the source operand is treated as the contents of a single register or a pair of registers depending on whether the register used is odd or even (line g_{28}). In right shift the sign bit is extended (lines 32,33). The results are stored back at the destination (line g_{38}).

In XOR the source and destination operands are exclusive and the result stored back at the destination address (lines g_{42-49}).

Lines h_0-h_{11}

Two of the other R-D instructions SOB and JSR have an entry at h_0. In SOB the contents of the register are decremented by 1 and if the result is not zero, then PC is decremented by the amount given by two times the value of the last six bits of the instruction, thus affecting a branch.

In JSR the destination operand is fetched from a register or memory (lines h_5,h_6) and execution proceeds if no exception has been noted (line h_7). The contents of the register are stacked (line h_8, h_{11}); the return address is stored in the register (line h_9) and the subroutine address stored in the PC (line h_{10}).

TABLE X

THE NAVIGATION MATRIX

				Octal Code	
$n_0n_1n_2n_3n_4$	Index	Mnemonic	Name	$I_0F_0F_1F_2F_3F_4$	Type
2 e_0 2 0 1	1	ADC	Add carry	0 0 5 5 - -	S
2 e_0 2 0 1	2	ADCB	Add carry (byte)	1 0 5 5 - -	S
0 f_0 2 0	3	ADD	Add	0 6 - - - -	D
1 g_0 2 0	4	ASH	Arith. shift	0 7 2 - - -	R
1 g_0 3 1	5	ASHC	Arith. shift combined	0 7 3 - - -	R
2 e_0 4	6	ASL	Arith. shift left	0 0 6 3 - -	S
2 e_0 4	7	ASLB	Arith. shift left (byte)	1 0 6 3 - -	S
2 e_0 3 0	8	ASR	Arith. shift right	0 0 6 2 - -	S
2 e_0 3 1	9	ASRB	Arith. shift right (byte)	1 0 6 2 - -	S
3 c_1 3	10	BCC	Branch if carry clear	1 0 3 - - -	B
3 c_2 3	11	BCS	Branch if carry set	1 0 3 - - -	B
3 c_2 1	12	BEQ	Branch if equal (to zero)	0 0 1 - - -	B
3 c_3 0	13	BGE	Branch if ≥0	0 0 2 - - -	B
3 c_4 0	14	BGT	Branch if >0	0 0 3 - - -	B
3 c_5 0	15	BHI	Branch if higher	1 0 1 - - -	B
3 c_1 3	16	BHIS	Branch if higher or same	1 0 3 - - -	B
0 f_0 3 1	17	BIC	Bit clear	0 4 - - - -	D
0 f_0 3 1	18	BICB	Bit clear (byte)	1 4 - - - -	D
0 f_0 3 2	19	BIS	Bit set	0 5 - - - -	D

TABLE X (Continued)

$n_0n_1n_2n_3n_4$	Index	Mnemonic	Name	Octal Code $I_0F_0F_1F_2F_3F_4$	Type
0 f_0 3 2	20	BISB	Bit set (byte)	1 5 - - - -	D
0 f_0 3 0	21	BIT	Bit test	0 3 - - - -	D
0 f_0 3 0	22	BITB	Bit test (byte)	1 3 - - - -	D
3 c_4 1	23	BLE	Branch if ≤0	0 0 3 - - -	B
3 c_2 3	24	BLO	Branch if lower	1 0 3 - - -	B
3 c_5 1	25	BLOS	Branch if lower or same	1 0 1 - - -	B
3 c_3 1	26	BLT	Branch if <0	0 0 2 - - -	B
3 c_2 0	27	BMI	Branch if minus	1 0 0 - - -	B
3 c_2 1	28	BNE	Branch if not equal (to zero)	0 0 1 - - -	B
3 c_1 0	29	BPL	Branch if plus	1 0 0 - - -	B
4 a_1 0	30	BPT	Break point trap	0 0 0 0 0 3	0
3 c_0 0	31	BE	Branch	0 0 0 - - -	B
3 c_1 2	32	BVC	Branch if overflow clear	1 0 2 - - -	B
3 c_2 2	33	BVS	Branch if overflow set	1 0 2 - - -	B
4 b_1 0	34	CCC	Clear condition codes	0 0 0 2 5 7	0
4 b_0 15 0	35	CLC	Clear C	0 0 0 2 4 1	0
4 b_0 12 0	36	CLN	Clear N	0 0 0 2 5 0	0
2 b_0 0 0	37	CLR	Clear	0 0 5 0 - -	S
2 e_0 0 0	38	CLRB	Clear (byte)	1 0 5 0 - -	S
3 b_0 14 0	39	CLV	Clear V	0 0 0 2 4 2	0

TABLE X (Continued)

$n_0n_1n_2n_3n_4$	Index	Mnemonic	Name	Octal Code $I_0F_0F_1F_2F_3F_4$	Type
3 b_0 1 3 0=	40	CLZ	Clear Z	0 0 0 2 4 4	0
0 f_0 1	41	CMP	Compare	0 2 - - - -	D
0 f_0 1	42	CMPB	Compare (byte)	1 2 - - - -	D
2 e_0 0 1	43	COM	Complement	0 0 5 1 - -	S
2 e_0 0 1	44	COMB	Complement (byte)	1 0 5 1 - -	S
2 e_0 2 1 0	45	DEC	Decrement	0 0 5 3 - -	S
2 e_0 2 1 0	46	DECB	Decrement (byte)	1 0 5 3 - -	S
1 g_0 1	47	DIV	Divide	0 7 1 - - -	R
4 a_1 2	48	EMT	Emulator trap	1 0 4 0 0 0 to 1 0 4 3 7 7	0
4	49	HALT	Halt	0 0 0 0 0 0	0
1 e_0 2 0 0	50	INC	Increment	0 0 5 2 - -	S
2 e_0 2 0 0	51	INCB	Increment (byte)	1 0 5 2 - -	S
4 a_1 1	52	IOT	I/O trap	0 0 0 0 0 4	0
2 c_0 1	53	JMP	Jump	0 0 0 1 - -	S
1 h_0 1	54	JSR	Jump to subroutine	0 0 4 - - -	R
4 d_0 0	55	MARK	Mark	0 0 6 4 - -	S
0 f_0 0	56	MOV	Move	0 1 - - - -	D
0 f_0 0	57	MOVB	Move (byte)	1 1 - - - -	D
1 g_0 0	58	MUL	Multiply	0 7 0 - - -	R
2 e_0 0 2	59	NEG	Negate	0 0 5 4 - -	S
2 e_0 0 2	60	NEGB	Negate (byte)	1 0 5 4 - -	S
4	61	NOP	No-op.	0 0 0 2 4 0	0

TABLE X (Continued)

$n_0n_1n_2n_3n_4$	Index	Mnemonic	Name	Octal Code $I_0F_0F_1F_2F_3F_4$	Type
4	62	RESET	Reset	0 0 0 0 0 5	0
2 e_0 4	63	ROL	Rotate left	0 0 6 1 - -	S
2 e_0 4	64	ROLB	Rotate left (byte)	1 0 6 1 - -	S
2 e_0 3 2	65	ROR	Rotate right	0 0 6 0 - -	S
2 e_0 3 2	66	RORB	Rotate right (byte)	1 0 6 0 - -	S
4 a_6	67	RTI	Return from interrupt	0 0 0 0 0 2	0
4 d_1 1	68	RTS	Return from subroutine	0 0 0 2 0 -	S
4 a_6	69	RTT	Return from interrupt	0 0 0 0 0 6	0
2 e_0 2 1 1	70	SBC	Subtract carry	0 0 5 6 - -	S
2 e_0 2 1 1	71	SBCB	Subtract carry (byte)	1 0 5 6 - -	S
4 b_1 1	72	SCC	Set condition codes	0 0 0 2 7 7	0
4 b_0 15 1	73	SEC	Set C	0 0 0 2 6 1	0
4 b_0 12 1	74	SEN	Set N	0 0 0 2 7 0	0
4 b_0 14 1	75	SEV	Set V	0 0 0 2 6 2	0
4 b_0 13 1	76	SEZ	Set Z	0 0 0 2 6 4	0
1 h_0 0	77	SOB	Subtract 1 and branch if ≠ 0	0 7 7 - - -	R
0 f_0 2 1	78	SUB	Subtract	1 6 - - - -	D
2 e_0 e	79	SWAB	Swap bytes	0 0 0 3 - -	S
2 e_0 6	80	SXT	Sign extend	0 0 6 7 - -	S

TABLE X (Continued)

$n_0n_1n_2n_3n_4$	Index	Mnemonic	Name	Octal Code $I_0F_0F_1F_2F_3F_4$	Type
$4\ a_1\ 3$	81	TRAP	Trap	1 0 4 4 0 0 to 1 0 4 7 7 7'	0
$2\ e_0\ 1\ 3$	82	TST	Test	0 0 5 7 - -	S
$2\ e_0\ 1\ 3$	83	TSTB	Test byte	1 0 5 7 - -	S
4	84	WAIT	Wait	0 0 0 0 0 1	0
$1\ g_0\ 4$	85	XOR	Exclusive OR	0 7 4 - - -	R

CHAPTER VI

USERS MANUAL

This manual is a reference for a programmer writing assembler language programs for the PDP 11/40 computer, using the assembler simulator described in this report. The assembler accepts a large subset of the standard assembler language and the execution time interpreter simulates almost the complete instruction set, with error checking, diagnostics and completion dump.

The first part of the manual describes the assembly language commands permitted by the assembler-simulator and essentially notes the differences from the standard assembler language. The PDP-11 Paper Tape Software Handbook (11), the PDP 11/40 Processor Handbook (10), and the Peripherals and Interfacing Handbook (9), completely describe the standard assembly language, addressing, input, output, etc., which this manual closely follows.

The second part describes input/output and debugging facilities available at execution time.

The third part describes the control cards, JCL and deck setup required to assemble/execute programs.

The fourth section describes the output from the assembler-interpreter, including the listing, format of the dump, error messages during assembly and execution.

The Assembly Language

This section describes the subset of the standard PDP 11/40 assembly languages accepted by the assembler. Only those features which the assembler omits or treats differently are described. With some exceptions, any program which assembles and executes correctly under this simulator should do so using the standard software. The assembler produces a listing of the source program, error messages, if any, and the location in memory into which the object deck is loaded.

Most of the section subheadings in this manual are taken from the Paper Tape Software Handbook (11).

Character Set

The Standard PDP 11/40 Character set is accepted except the characters for carriage return, tab, space, line feed and form feed.

Statements

Each statement of the program must be on a single card and within columns 1-40. Statement segments beyond column 40 are ignored, and may give assembly time errors. A statement may consist of label, opcode, operand, and comment fields. The label and comment fields are optional and operand(s) may or may not be required depending on the operator. A free format of the fields is acceptable.

A label is a symbol terminated by a colon. No embedded blanks between the symbol and the colon are permitted. Multiple labels per statement are acceptable. An example of a statement with 3 labels is given below:

```
A:AB:LABEL: STATEMENT;
```

The opcode field contains an instruction mnemonic or an assembler directive. The opcode field is terminated by a semicolon or by two or more blanks or by any of the special characters. Examples, of opcode fields terminated by a semicolon and a special character are

```
HALT;
MOV#RDINT,R2;
```

where the mnemonics are HALT and MOV.

The operand field may contain one or more subfields, separated by commas. Embedded blanks are not permitted in this field. Operand subfield may contain symbols, expressions or numbers. The operand field is terminated by a semicolon or two or more blanks. The following assembler statement is an example.

```
.WORD A,MN+2,-4,'I-3;
```

The operand field in the example statement consists of four subfields.

Comments may follow operand (5) and may extend up to column 80 of the card.

Symbols

All labels and symbols used in the operand field have to be defined. Labels can be defined by using the label symbol in the label field. Other symbols may be defined by direct assignments. A symbol may be defined only once except in cases of direct assignments. Periods are not permitted in symbols.

Direct Assignment

Direct assignment statements assign values to symbols. A symbol may be defined/redefined by a direct assignment. The '=' operator must not be preceded or followed by one or more blanks. Forward referencing in direct assignments, such as

```
x=y
y=2
```

is not acceptable.

Register Symbols

All variations of register symbols of the standard language are accepted except the following. Register symbols can take values between 0 and 7, the registers accessible to a programmer. They cannot be assigned values of their memory locations as can be done in the standard assembly language.

Expressions

Only arithmetic operators + and - are acceptable in expressions. Logical operators & and | are not supported. Expressions also may contain symbols, numbers, ASCII data, or location counter references. Expression evaluation is done from left to right. Parentheses and embedded blanks are not permitted.

Location Counter

The location counter may be referenced in expressions. It also can be set to a value by direct assignment. Setting the counter to less than 128 results in an error as the first 128 words are reserved for the system.

Machine Instructions

A large subset of the machine instructions is supported. The instructions EMT, TRAP, BPT, IOT, RTT, WAIT and RESET, however, are not supported. The extended instruction set (EIS) instructions MUL, DIV, ASH, ASHC are supported. All instructions are assembled on word boundaries.

Assembler Directives

The .EVEN, .END, .WORD, .BYTE and .ASCII directives are supported. The .END directive signifies end of the source program and must have an operand (a single symbol) which matches with the label on the first executable statement of the program and signifies the program entry point. Absence of either the operand or the label may cause assembly time errors. A .END directive is mandatory for every program.

Operands of the .WORD and .BYTE directives can be symbols, expressions or numbers, and are separated by commas. Embedded blanks are not permitted.

Addressing

All 12 variations of the 8 different modes of addressing are accepted by the assembler. A detailed description and the assembler formats of these variations is given in Chapter II.

Stack Operation

The processor stack is used to save data temporarily which might otherwise be altered. The stack is a series of memory locations, pointed to by a stack pointer (normally register 6). As such, the stack pointer has to be initialized at the beginning of each program, if the program makes use of the stack.

A description of various programming techniques is given in Chapter 9 of the PDP 11 Paper Tape Software Programming Handbook (11).

Suggestions for Improving Assembly Time

Assembly time efficiency can be improved if the following suggestions are adopted in the program statements.

(1) Semicolons are used immediately after the last operand subfield if one is present, or after the instruction mnemonic if no operand is required, to terminate a statement.

(2) Decimal numbers are used, wherever possible, instead of octal numbers. This eliminates the conversion from octal to decimal.

Input/Output and Debugging

Input/output from the papertape reader/punch and teletype has been simulated using the system interrupt structure. A detailed description of I/O performed in this manner is given in the Peripherals and Interfacing Handbook (9).

I/O is initiated by setting the device enable bit in the device control and status register. When the data is available in the device buffer, an interrupt request is made. If the request is honored, control branches to a service routine. All service routines are user supplied. The address of the device service routine, has to be loaded by the programmer, into the device interrupt vector, as part of program initialization. The example statements

```
MOV  #RDINT,@#48.
MOV  #PUNINT,@#52.
```

illustrate this. The two statements move the addresses of the read and punch routines RDINT and PUNINT, to memory locations 48 and 52, which are the interrupt vector addresses of the teletype reader and punch, respectively. This information is used when an interrupt request is made. To effect a branch to the service routine, the program counter is loaded from the contents of the interrupt vector address.

Numerous examples of service routines are available in the Peripherals and Interfacing Handbook (11).

Debugging

A pair of debugging instructions SET and CLT is provided. These instructions are not supported by the standard assembly language. Instruction SET, sets the trace bit in the Processor Status word which causes a dump of all general purpose registers, pseudo registers and the Processor Status word, in octal. The dump is printed for each instruction executed after SET, until a CLT clears the trace bit.

A post-execution memory dump is provided in octal which also includes a dump of all registers. In case of execution time errors, the execution is terminated after a register and memory dump is printed.

Simulation of Device-Initiated Interrupts

The simulator also supports simulation of up to a maximum of two device initiated interrupts, concurrently with program execution. The time at which the processor gets interrupted by a device is supplied by the programmer. Once the processor is interrupted, the programmer supplied data is input into user-defined memory locations. If input consists of more than 1 character they are placed in consecutive memory locations, beginning with the location defined by the programmer.

A description of the deck setup is given in a later section.

Control Cards and JCL

Each program should begin with a JOB card. The format for the JOB card is shown in Figure 26.

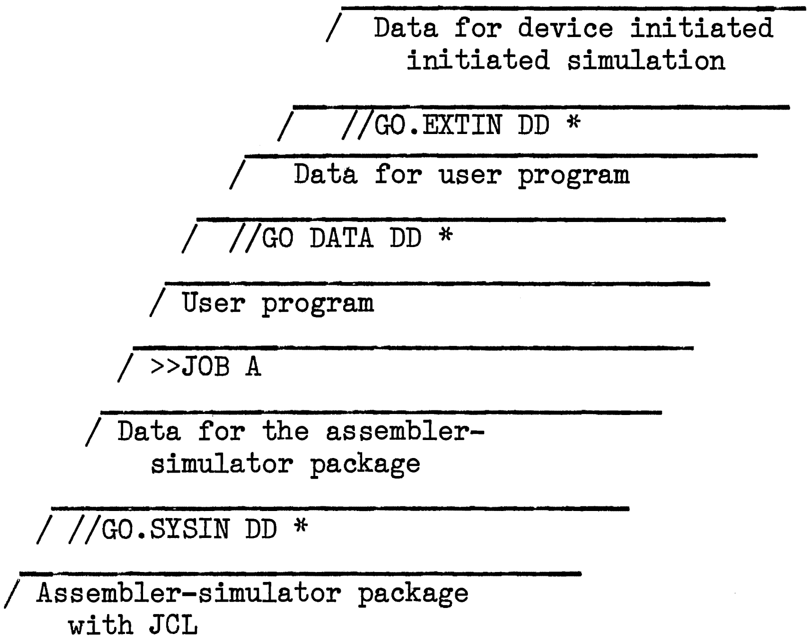
```
123456789
>>JOB
```

Figure 26. The JOB Card

Column 7 must contain either the character 'A' or the character 'E.' Under option 'A,' the user program is assembled but not executed. The assembler listing and the symbol table are printed. Under option 'E,' the user program is executed if it is error free.

Deck Setup and Output

The deck setup to use the assembler-interpreter is shown in Figure 27.



Data for device initiated initiated simulation

//GO.EXTIN DD *

Data for user program

//GO DATA DD *

User program

\>>JOB A

Data for the assembler-simulator package

//GO.SYSIN DD *

Assembler-simulator package with JCL

Figure 27. Deck Setup

The setup consists of the assembler-simulator package which operates on one or more user programs. Each user program begins with a >>JOB Card. Data for each user program is contained in the file DATA, on separate cards. Data for device-initiated interrupts is contained in file EXTIN. More than one job can be assembled and executed in one compilation of the assembler-simulator. The deck setup is as shown in Figure 28. Data for the jobs is given in file DATA.

```
>>JOB E
    JOB1 . . .
       .
       .
       .
         .END JOB1
>>JOB A
    JOB2 . . .
       .
       .
       .
         .END JOB2
```

Figure 28. Program Setup

Data for Device-Initiated Interrupt Simulation

At most two device-initiated interrupts can be simulated. Simulation is done only if option 'E' is specified on the job card. The interrupt gueue is generated at program execution time. Data for the queue, which includes interrupt time, number of characters to be input (device-initiated interrupts, as set up in the program, simulate direct memory accesses to input data from the interrupting devices), and the memory locations where the characters should be stored, is supplied on a header card in the file EXTIN. The format of the header card is as follows.

(1) Columns 1-5, 18-22, contain the times to interrupt, in cycles, one for each interrupt. The interrupt times should be right justified in the fields and should be integer numbers greater than zero. The time of the first interrupt, in columns 1-5, should be smaller than the time of the second interrupt, columns 18-22; the queue is processed in a strictly sequential manner beginning with the first element and proceeding to the rear. When the cycle counter of the processor equals the interrupt time, the processor gets interrupted, and the number of characters supplied is input at the memory locations specified. If device initiated interrupt simulation is not required, the interrupt times should be set to negative numbers.

(2) Columns 7-11, 24-28, contain the address of the memory location, in decimal, where the characters input are to be stored, one for each interrupt. If the number of characters is greater than one, the characters are stored in consecutive locations starting with

the location specified. The location specified should be a number greater than 128, right justified.

(3) Columns 13-15, 30-32, contain the number of characters to be input, one for each interrupt. The number should be greater than zero, right justified.

(4) Column 17, 34 contain the unit number of the interrupting device, one for each interrupt. The unit number is solely for the purpose of identification, and assignment of unit numbers to devices is arbitrary. The unit number appears in the message in the simulator output when the processor is interrupted.

Following the header card are the data cards which contain the sets of characters to be input, one set for each interrupt. Each set begins on a new card.

The assembly listing produced by the assembler consists of the source statement, a statement number, the assembler code in octal and the location in memory where the instruction is loaded. Error messages are printed after each statement causing the messages. The error messages consist of a two-digit error code in the format

***ERROR #NN

where NN is the error code. The following section lists the codes and messages issued by the assembler. The program will not be loaded into memory for execution even if a single error is detected.

Immediately following the assembler listing, the sorted symbol table is printed along with the values associated with the symbols. The values are printed in octal.

Output from the interpreter consists of the register dumps if SET and CLT are used in the program, plus a post execution memory dump. The format of the dump consists of 16 words printed per line with the memory location printed at the left. If device interrupted simulation is done, the output also consists of messages indicating the processor interrupt time, the unit number of the interruptive device and a list of characters which are input into the memory.

The assembler error codes and messages are given in Table XI. Execution-time error messages are printed as soon as an error is detected. Most execution time errors cause termination of program execution.

TABLE XI

ERROR CODES AND MESSAGES

Code	Message
0	Invalid sequence of operators The sequence of operators in an operand expression is illegal. A few senquences which may give this error are: @-, anything other than a blank or ',' following) or)+, # or @# preceding a register expression, -(exp)+, @#(or #(.
1	Invalid symbol The construct of the symbol does not conform to the rules.
2	Invalid label The construct of the label does not conform to the rules.
3	Soubly defined label Either a label is used more than once or the first six characters of the label are identical to the first six characters of another label.
4	Unidentifiable symbol Symbol in a statement is neither a label nor a machine opcode.
5	Undecodable statement Symbol in a statement cannot be identified with any of the four fields of the statement.
6	Invalid symbol in expression Symbol in an expression is not an arithmetic operator, number or a valid symbol.
7	Relational operator missing The '=' operator is missing in a direct assignment. Statement is also flagged if

TABLE XI (Continued)

Code	Message
	one or more blanks precede or follow the '=' operator.
8	Illegal assembler directive Assembler directive used is not supported. Statement is also flagged if the label on the statement, if one is used, does not conform to the rules of label construct.
9	Invalid mnemonic Instruction mnemonic used in the statement is not supported. Statement is also flagged if the label on the statement, if one is used, does not conform to the rules of label construct.
10	Missing operand in expression A number or symbol is missing in an expression with the result that two operators are consecutive.
11	Invalid ASCII conversion in expression ASCII conversion symbol used is not supported. Only the ' conversion symbol is supported.
12	Invalid decimal number The period terminating a decimal number is missing.
13	Missing operator Arithmetic operator in an expression is missing.
14	Invalid operator Arithmetic operator in an expression is other than + or -.
15	More than one '%' in register expression A register expression contains more than one register definition symbol '%.' only one is permissible.

TABLE XI (Continued)

Code	Message
16	Undefined symbol or label in expression Operand expression contains a symbol or label which has not been defined.
17	'%' used on a register symbol The register definition symbol '%' is used on a symbol which has already been defined as a register symbol.
18	Register expression evaluates to a value greater than 7 or less than 0. Only registers 0-7 are programmer accessible.
19	Unmatched parenthesis Either a left or a right parenthesis is missing in the operand field.
20	Extra operands Operand field contains operands which are in excess of the number required by the instruction mnemonic. Statement is also flagged if an unpermitted sequence of operators is used.
21	Nested parenthesis Nested parenthesis encountered in an expression. Use of parenthesis in expressions is illegal.
22	Expression in the first operand subfield of a Register Destination type instruction evaluates to a value greater than 7 or less than 0. Statement is also flagged if an unpermitted sequence of operators is used (as in code 0).
23	Offset in a branch instruction evaluates to a value greater than 127 or less than -128.
24	Absolute, Immediate or Indexed mode used for operand of a Branch instruction.
25	No .END Statement in the program.

TABLE XI (Continued)

Code	Message
26	Operand of an .END directive is an invalid symbol or is an undefined label. The operand of the .END directive should match the label on the first executable statement in the program.
27	Register used in an RTS expression evaluates to a value greater than 7 or less than 0
28	Register expression used in the indexed field of an operand. Only ordinary expressions are permissible (i.e. expressions with '%').
29	Second operand of a SOB instruction evaluates to a value less than -63 words, or is positive and may therefore cause branch in a forward direction.
30	Expression in the second operand subfield of the SOB or MARK instruction is a register expression. Only an ordinary expression is permissible.
31	Operand of MARK instruction is negative or greater than 63.
32	Register symbol used in an operand field is undefined.
33	Attempt to set the location counter to a value less than 128. The first 128 words are reserved for the system.

CHAPTER VII

SUMMARY AND CONCLUSIONS

Using the methods outlined in this paper, an assembler-simulator package for the PDP 11/40 has been implemented. Two options have been provided for using the package. The option can be specified on the JOB card. Under option 'A' the user-program is assembled only, and the source-program listing, the symbol table and diagnostic messages, if any, are printed. Under option 'E' the user program is executed after assembly.

The package has also performed simulated input/output from teletype and papertape reader/punch. Software packages, that need to be developed for the PDP 11/40, can be tested using the package. Extensive assembly and execution time error checking is performed. Lucid diagnostic messages are printed in the assembler listing. Special debugging instructions have been added to the instruction set, to help in execution time error detection. A register and memory dump is printed at the termination of program execution.

Device-initiated interrupt simulation, is also performed concurrent with program execution. Processor operation is interrupted at user-defined interrupt times, for a direct memory access, to input signals into memory.

A formal description of the computer in APL gives in detail the processor operation, instruction interpretation and execution, interrupt servicing, etc. Beside providing a concise description of the complex operations, APL permits sufficient detail to describe operation at the hardware level.

With some exceptions any program which assembles and executes under this package should do so using standard system software.

BIBLIOGRAPHY

(1) Gear, William C. Computer Organization and Programming. New York: McGraw-Hill, 1969.

(2) Hellerman, H. Digital Computer System Principles. New York McGraw-Hill, 1967.

(3) Hopcroft, J. E. and J. D. Ullman. Formal Languages and Their Relation to Automata. Reading, Massachusetts: Addison-Wesley, 1969, pp. 26-27.

(4) Iverson, Kenneth E. A Programming Language. New York: John Wiley, 1962.

(5) Iverson, Kenneth E. et al. Formal Description of System /360. New York: IBM Systems Journal, Vol. 3, No. 3, 1964.

(6) KD11 - A Processor Manual DEC-11-HKDAA-A-D. Maynard, Massachusetts: Digital Equipment Corporation, 1973.

(7) Korn, Granino A. Minicomputers for Engineers and Scientists. New York: McGraw-Hill, 1973.

(8) Maisel, H. and G. Gnugnoli. Simulation of Discrete Stochastic Systems. Chicago, Illinois: Science Research Associates, 1972.

(9) Naylor, Thomas H. et al. Computer Simulation Techniques. New York: John Wiley, 1966.

(10) PDP 11 Peripherals and Interfacing Handbook. Maynard, Massachusetts: Digital Equipment Corporation, 1971.

(11) PDP 11/40 Processor Handbook. Maynard, Massachusetts: Digital Equipment Corporation, 1972.

(12) PDP 11/40 Paper Tape Software Programming Handbook. Maynard, Massachusetts: Digital Equipment Corporation, 1973.

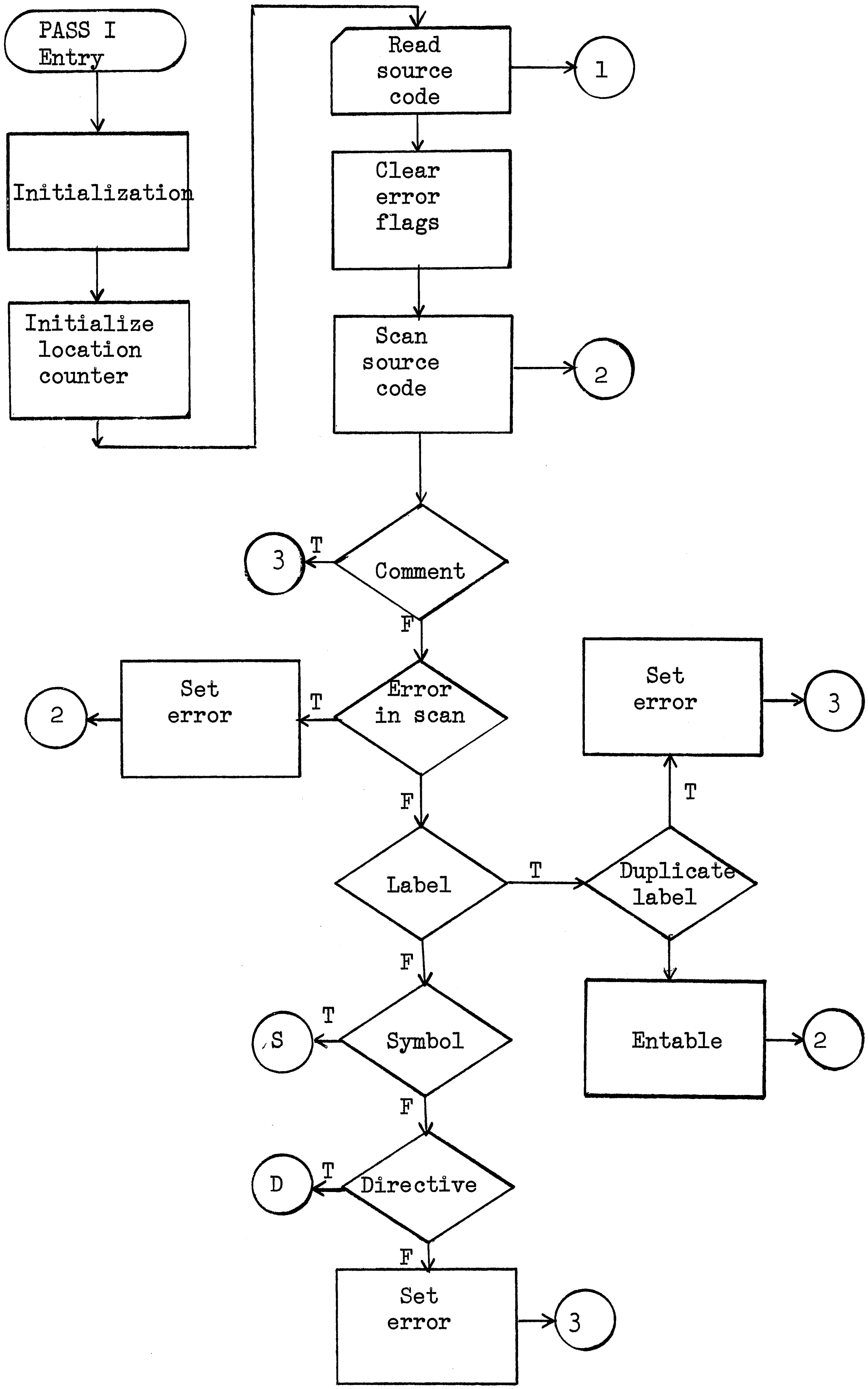
(13) PDP 11/40 System Manual DEC-11-H40SA-A-D. Maynard, Massachusetts: Digital Equipment Corpoation, 1973.

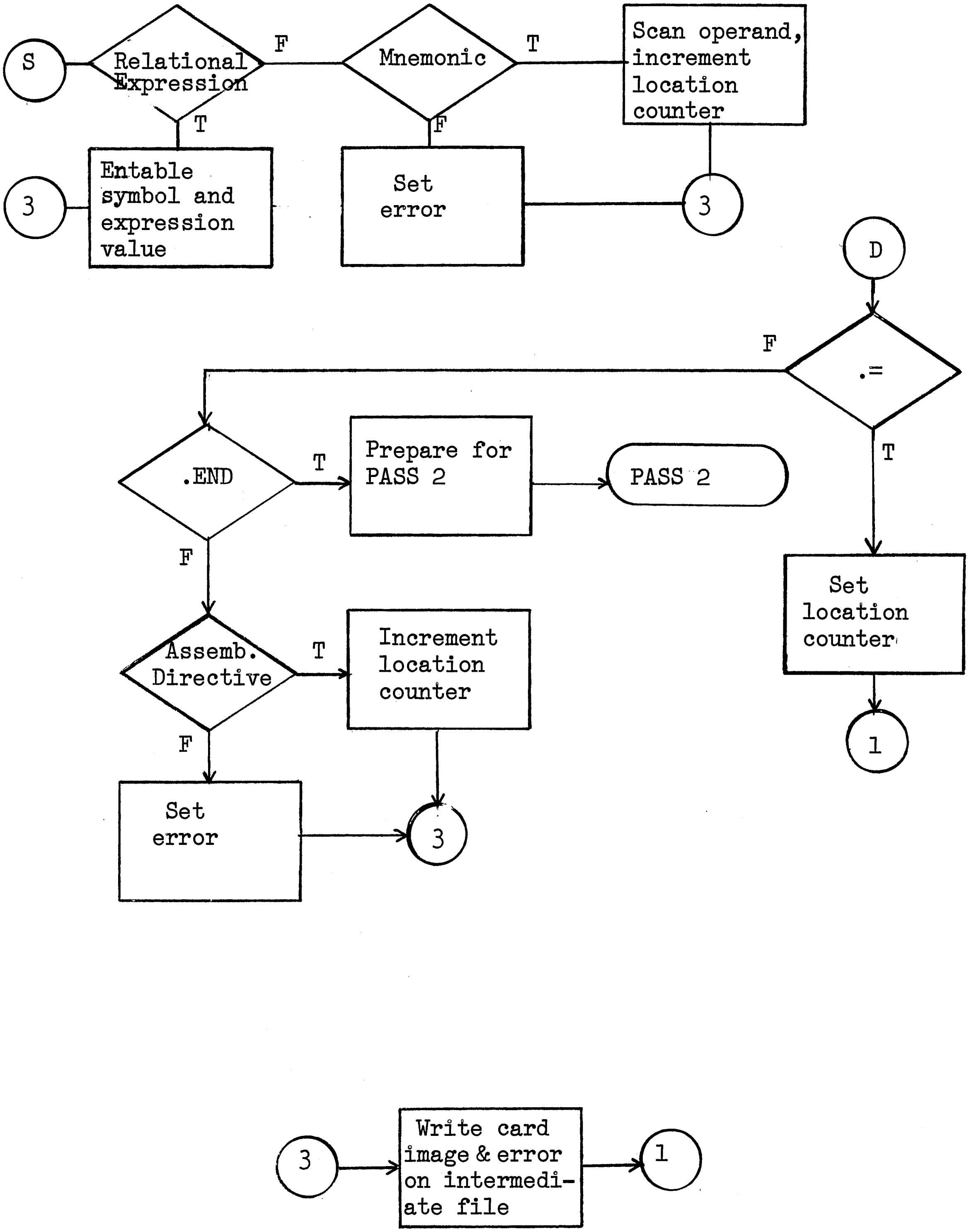
(14) Soucek, Branko, Minicomputers in Data Processing and Simulation. New York: Wiley-Interscience, 1972.

(15) Wegner, Peter. Programming Languages, Information Structures and Machine Organization. New York: McGraw-Hill, 1968.

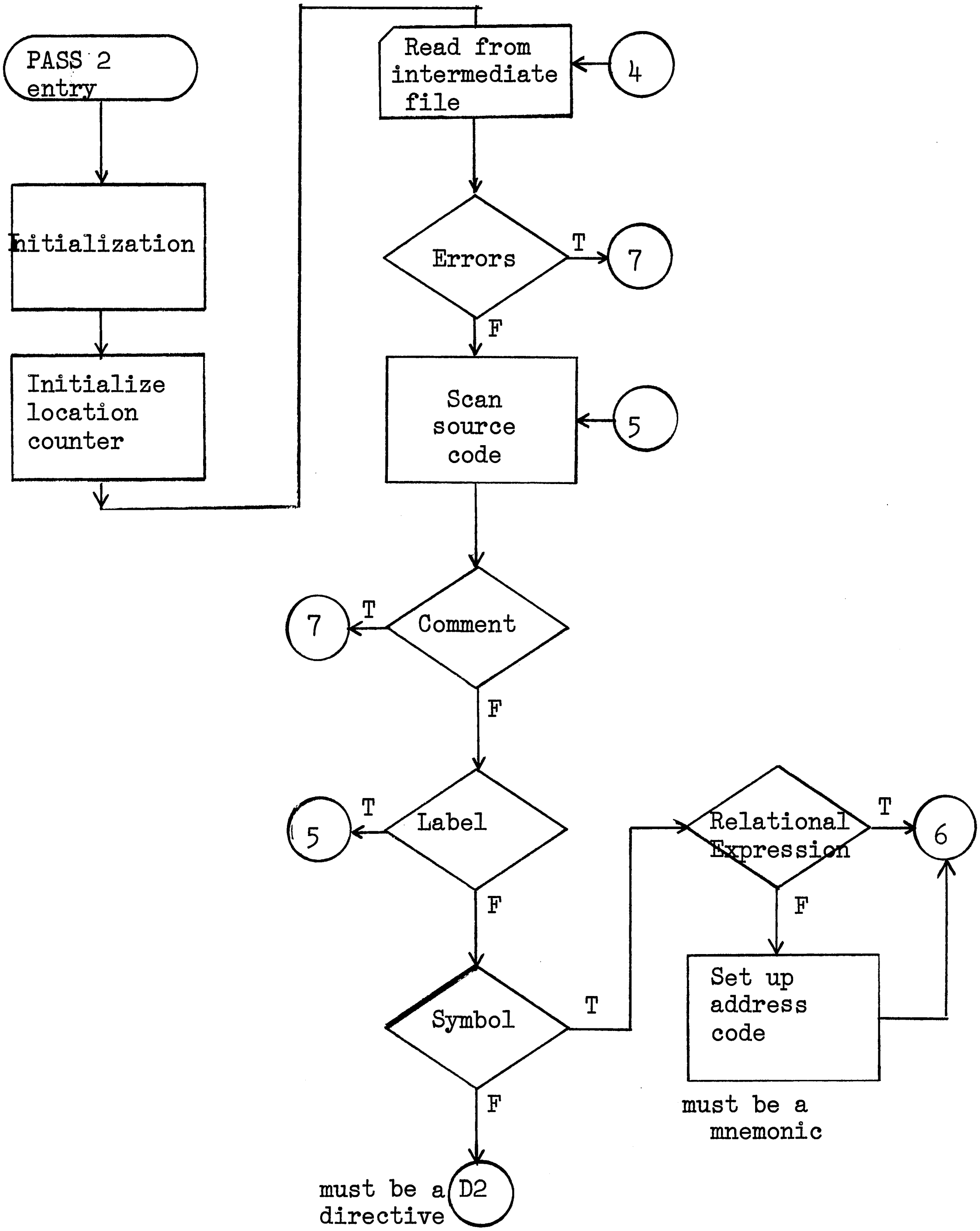
APPENDIX A

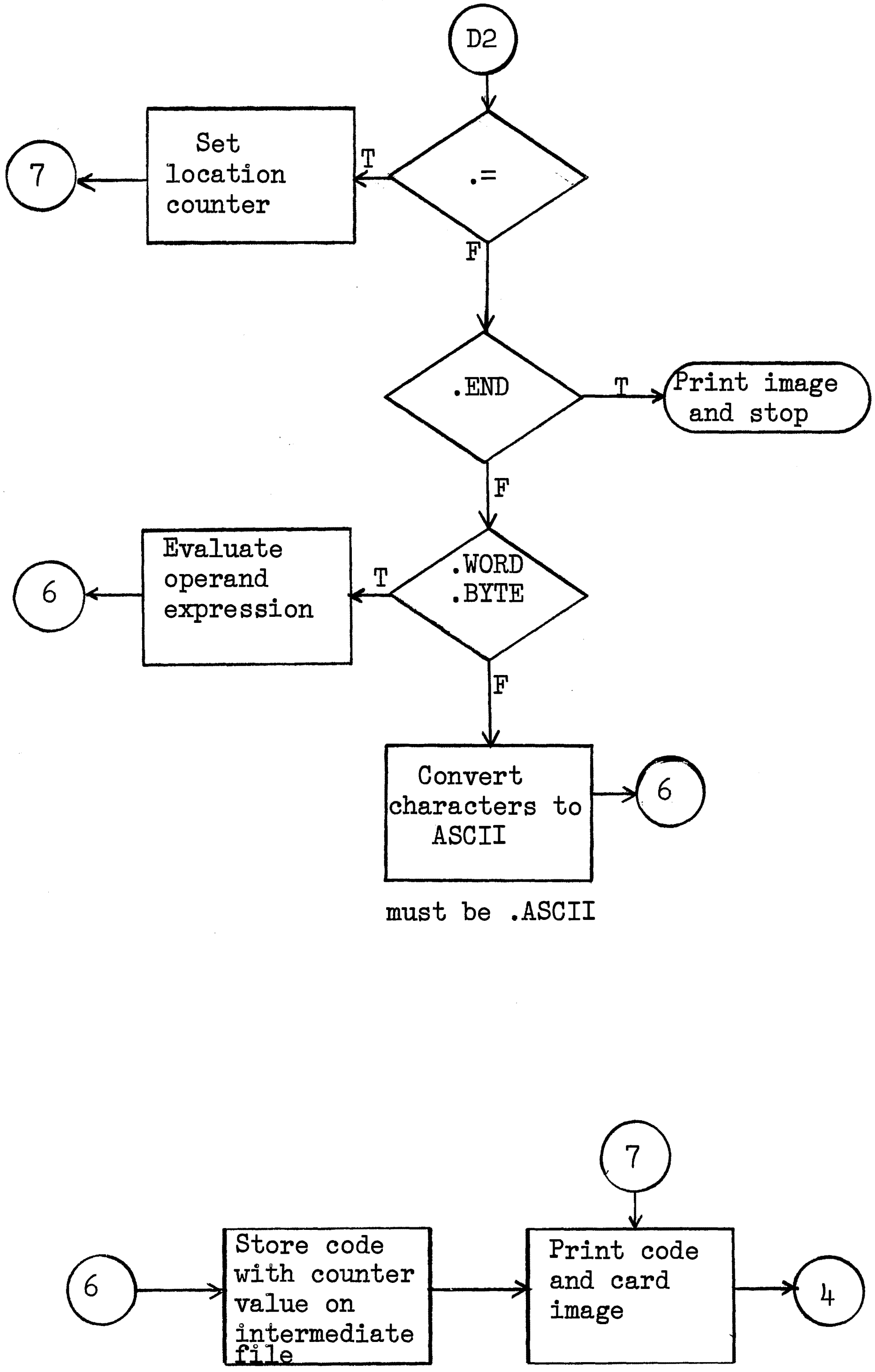
PROGRAM FLOWCHART


PASS I Entry
Initialization
Initialize location counter
Read source code
1
Clear error flags
Scan source code
2
Comment
T
3
F
Error in scan
T
Set error
2
F
Label
T
Duplicate label
T
Set error
3
Entable
2
F
Symbol
T
S
F
Directive
T
D
F
Set error
3

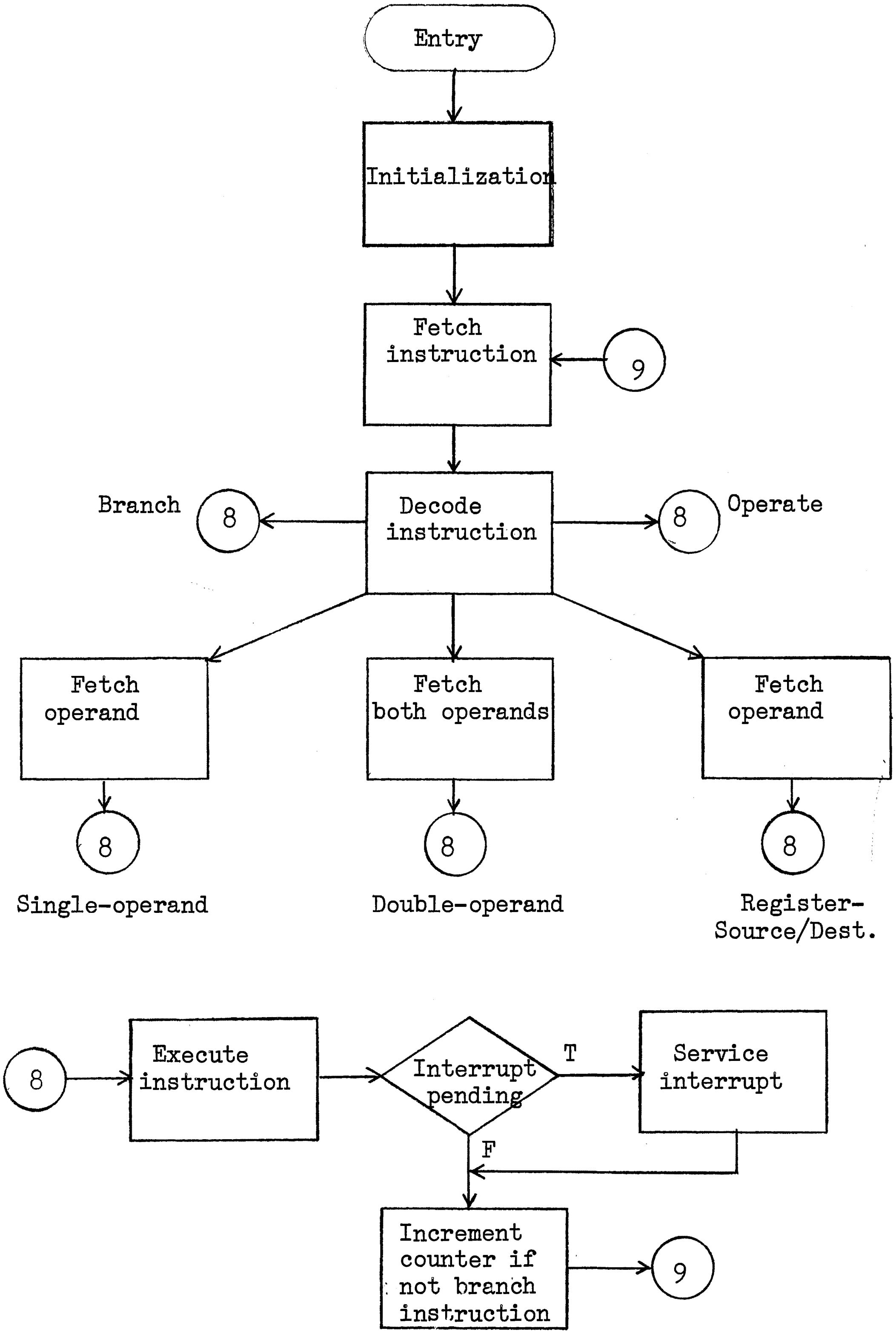

S
Relational Expression
F
T
Entable symbol and expression value
3
Mnemonic
T
F
Scan operand, increment location counter
Set error
3
D
.=
F
T
Set location counter
1
.END
T
Prepare for PASS 2
PASS 2
F
Assemb. Directive
T
Increment location counter
F
Set error
3
3
Write card image & error on intermediate file
1

PASS 1 of the Assembler

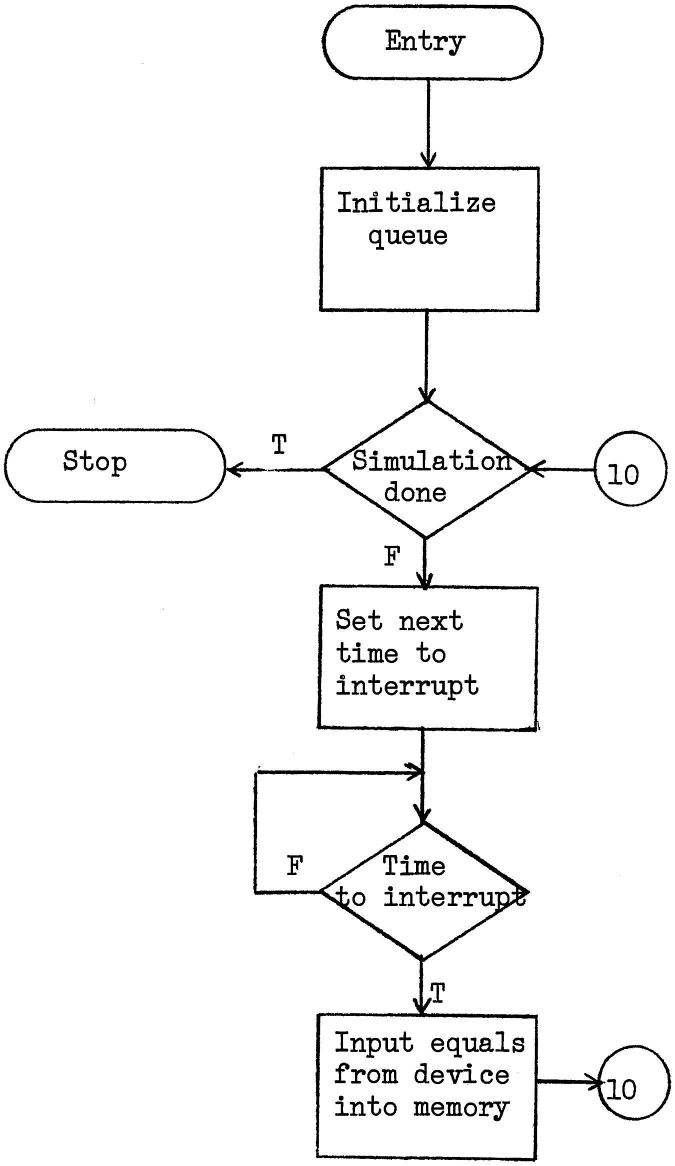

PASS 2 entry
Initialization
Initialize location counter
Read from intermediate file
4
Errors
T
7
F
Scan source code
5
Comment
T
7
F
Label
T
5
F
Symbol
T
F
Relational Expression
T
6
F
Set up address code
must be a mnemonic
must be a directive
D2


D2
.=
T
Set location counter
7
F
.END
T
Print image and stop
F
.WORD .BYTE
T
Evaluate operand expression
6
F
Convert characters to ASCII
6
must be .ASCII
7
6
Store code with counter value on intermediate file
Print code and card image
4

PASS 2 of the Assembler



The Simulator



Simulation of Device-Initiated Interrupts

APPENDIX B

SAMPLE PROGRAM OUTPUT

APPENDIX B

SAMPLE PROGRAM OUTPUT

A description of two test cases for the assembler-simulator is presented. The first program, 'SAMPLE,' has been tested with option 'E' on the JOB card. The program is, consequently, assembled and executed. The second program, 'TEST,' has been tested under option 'A,' and if, therefore, assembled only.

The output of the first program, consists of the source-program listing followed by the symbol table. The symbol table contains symbols, followed by their values, printed in octal. The program computes the sum of a series, converts the sum to ASCII, and prints the result using the teletype printer. Concurrent device-initiated interrupt servicing is also involved. The execution time output immediately follows the symbol table.

Two device initiated interrupts have been serviced during execution. The first involved input of 14 characters and the second, an input of 10 characters into memory. The messages identify the interrupting device and contain the characters which are input. The register dump, which can be used as an execution time debugging aid, consists of register and pseudo-register contents, printed in octal. The processor status word is printed in memory. The sum of the series, computed by the program, is printed next. A register and memory dump terminates program execution.

The second program assembler listing and symbol table follow the output of the first program. No execution was necessary in this case.

The deck setup for the test cases is given.

```
                  111111111122222222223333333333444444
Column   123456789012345678901234567890123456789012345 . . .

         // JOB
           (IBM 360 JCL)
              .
              .
              .
           CALL TO ASSEMBLER-SIMULATOR
              .
              .
              .
         //GO.SYSIN DD *
              .
              .
              .
              DATA/TABLES FOR ASSEMBLER-SIMULATOR
              .
              .
              .
         >>JOB E
              .
              .
              .
              PROGRAM 'SAMPLE'
              .
              .
              .
         >>JOB A
              .
              .
              .
              PROGRAM 'TEST'
         /*
         //GO.DATA DD *
              :
             Data for programs
         /*
         //GO.EXTIN DD *
              :
             DATA for Device initiated interrupt simulation
         /*
         //.
```

```
LOC      CODE             STMT SOURCE STMT                                          PAGE         1
                                                                                    DATE 06/12/75

                             1 ;
                             2 ;
                             3 ;       THIS SAMPLE PROGRAM EVALUATES THE SUM OF THE SERIES
                             4 ;       SUM = 2 * SUM1 + 4 * SUM2 WHERE
                             5 ;         SUM1 = X(1) + X(3) + ............. + X(2N+1) AND
                             6 ;         SUM2 = X(2) + X(4) + ............. + X(2N).
                             7 ;       THE SUM IS THEN CONVERTED TO ASCII BY THE ROUTINE $ICO, AND
                             8 ;       PRINTED BY USING THE TELETYPE PRINTER.
                             9 ;
       000000               10           R0=%0;
       000001               11           R1=%1;
       000002               12           R2=%2;
       000003               13           R3=%3;
       000004               14           R4=%4;
       000005               15           R5=%5;
       000006               16           SP=%6;
       000007               17           PC=%7;
       177564               18           TPS=177564;
       177566               19           TPB=177566;
000200 012706               20 SAMPLE:   MOV    #1000.,SP;     INITIALIZE STACK POINTER
       001750
000204 012737               21           MOV    #PRTRTN,@#52.;  ADDRESS OF PRINT ROUTINE TO VECTOR ADDRESS
       000310
       000064
000212 012702               22           MOV    #TABLE,R2;     ADDRESS OF TABLE IN R2
       000636
000216 012703               23           MOV    #3.,R3;        COUNT IN R3
       000003
000222 012204               24           MOV    (R2)+,R4;      SUM IN R4
000224 006304               25           ASL    R4;
000226 012205               26 LOOP:     MOV    (R2)+,R5;      GET NEXT NUMBER
000230 006305               27           ASL    R5;            NUMBER * 2
000232 062205               28           ADD    (R2)+,R5;      ADD NEXT TERM
000234 006305               29           ASL    R5;            NUMBER * 2
000236 060504               30           ADD    R5,R4;         SUM IN REGISTER 4
000240 077306               31           SOB    R3,LOOP;       REPEAT IF MORE
000242 000265               32           SET
000244 010467               33           MOV    R4,SUM;
       000362
000250 000245               34           CLT
000252 012746               35           MOV    #ASCSUM,-(SP);        PREPARE TO CONVERT TO ASCII
       000722
000256 012746               36           MOV    #4.,-(SP);     FIELD LENGTH ON STACK
       000004
000262 010446               37           MOV    R4,-(SP);      VALUE TO BE CONVERTED ON STACK
000264 004767               38           JSR    PC,$ICO;
       000046
000270 016700               39           MOV    MILEN,R0;
       000410
000274 012702               40           MOV    #MSG,R2;       ADDRESS OF MESSAGE AREA
       000706
000300 012737               41           MOV    #64.,@#TPS;    ENABLE PRINTER
       000100
       177564
000306 000000               42           HALT
                            43 ;
```

```
LOC     CODE             STMT SOURCE STMT                                                     PAGE      2

                           44 ;       THE PRINTER SERVICE ROUTINE
                           45 ;
000310 112237              46 PRTRTN:  MOVB  (R2)+,@#TPB;  CHARACTER FROM AREA TO BUFFER
       177566
000314 005300              47          DEC   R0;           DECREMENT COUNT
000316 001401              48          BEQ   PUNOVR;
000320 000002              49          RTI;
000322 005037              50 PUNOVR:  CLR   @#TPS;
       177564
000326 000002              51          RTI;
                           52 ;
                           53 ;
                           54 ;       ROUTINE $ICO TO CONVERT INTEGER TO ASCII
                           55 ;       ROUTINE $OCO TO CONVERT OCTAL TO ASCII
                           56 ;       CALLING SEQUENCE :      PUSH FIELD START LOCATION ON STACK
                           57 ;                               PUSH FIELD LENGTH ON STACK
                           58 ;                               PUSH VALUE ON STACK
                           59 ;                               JSR   PC,$ICO(OR $OCO)
                           60 ;       ERROR WILL RETURN WITH C BIT SET ON
                           61 ;   NOTE - R0, R1, R2, R3 ARE DESTROYED
                           62 ;
                           63 ;
000330 012700              64 $OCO:    MOV   #OCT$25-REL$25,R0;      POINT TO OCTAL TABLE
       000172
000334 000402              65          BR    GO$25;
000336 012700              66 $ICO:    MOV   #DEC$25-REL$25,R0;      POINT TO DECIMAL TABLE
       000160
000342 010446              67 GO$25:   MOV   R4,-(SP);
000344 016603              68          MOV   8.(SP),R3;    GET FIELD START
       000010
000350 016602              69          MOV   6.(SP),R2;    GET FIELD LENGTH
       000006
000354 002003              70          BGE   LPS$25;       JUMP OF NOT NEG
000356 005002              71          CLR   R2;
000360 005066              72          CLR   6.(SP);
       000006
000364 016604              73 LPS$25:  MOV   4.(SP),R4;    GET VALUE TO BE CONVERTED
       000004
000370 012746              74          MOV   #' ,-(SP);    CLEAR SIGN
       000040
000374 020027              75          CMP   R0,#OCT$25-REL$25;      CHECK IF DOING OCTAL
       000172
000400 001405              76          BEQ   POS$25;       YES, GIVE MAGNITUDE RESULT
000402 005704              77          TST   R4;
000404 002003              78          BGE   POS$25;       JUMP IF +
000406 005404              79          NEG   R4;           GET ABSOLUTE VALUE
000410 012716              80          MOV   #'-,@SP;      SAVE -
       000055
000414 005046              81 POS$25:  CLR   -(SP);        SET FENCE
000416 062700              82          ADD   #2.,R0;
       000002
000422 060700              83          ADD   PC,R0;
                           84 REL$25:
000424 005710              85 TST$25:  TST   @R0;
000426 001420              86          BEQ   MOV$25;       JUMP IF ALL POWERS DONE
000430 005001              87          CLR   R1;
```

```
000432 011005           88 SUB$25:  MOV    @R0,R5;
000434 005405           89          NEG    R5;
000436 060504           90          ADD    R5,R4;          SEE IF CURRENT POWER WILL GO AGAIN
000440 002402           91          BLT    BAC$25;
000442 005201           92          INC    R1;             BUMP DIGIT
000444 000772           93          BR     SUB$25;
000446 062004           94 BAC$25:  ADD    (R0)+,R4;       TOO MUCH, BACK UP
000450 005701           95          TST    R1;
000452 001002           96          BNE    NZE$25;         JUMP IF DIGIT NOT 0
000454 005716           97          TST    @SP;
000456 001762           98          BEQ    TST$25;         JUMP OF NO NON-ZERO DIGITS YET
000460 062701           99 NZE$25:  ADD    #48.,R1;        CONVERT TO ASCII
       000060
000464 010146          100          MOV    R1,-(SP);
000466 000756          101          BR     TST$25;
000470 060203          102 MOV$25:  ADD    R2,R3;          POINT TO FIELD END
000472 062704          103          ADD    #48.,R4;        CONVERT LEAST SIGNIFICANT DIGIT
       000060
000476 110443          104          MOVB   R4,-(R3);
000500 005302          105 DCR$25:  DEC    R2;
000502 003410          106          BLE    FUL$25;         JUMP IF COUNT EXHAUSTED
000504 112643          107          MOVB   (SP)+,-(R3);    MOVE DIGIT
000506 001374          108          BNE    DCR$25;         JUMP IF NOT FENCE
000510 112613          109          MOVB   (SP)+,@R3;      MOVE OUT THE SIGN
000512 005302          110 FIL$25:  DEC    R2;
000514 001410          111          EEQ    CNE$25;         JUMP IF FIELD FILLED
000516 112743          112          MOVB   #' ,-(R3);      MOVE IN LEADING BLANKS
       000040
000522 000773          113          BR     FIL$25;
000524 005726          114 FUL$25:  TST    (SP)+;
000526 001011          115          BNE    ERR$25;         NUMBER TOO BIG FOR FIELD
000530 022726          116          CMP    #' ,(SP)+;
       000040
000534 001011          117          BNE    STS$25-4.;
000536 012604          118 DNE$25:  MOV    (SP)+,R4;
000540 012666          119          MOV    (SP)+,4.(SP);     MOVE RETURN UP
       000004
000544 005726          120          TST    (SP)+;          FLUSH VALUE
000546 006126          121          ROL    (SP)+;          FLUSH FLAG AND SET C BIT ON IF ERROR
000550 000207          122          RTS    PC;
000552 005726          123 ERR$25:  TST    (SP)+;
000554 001376          124          BNE    ERR$25;
000556 005726          125          TST    (SP)+;          FLUSH SIGN
000560 016603          126          MOV    8.(SP),R3;
       000010
000564 112723          127 STS$25:  MOVB   #'*,(R3)+;      FILL FIELD WITH *
       000052
000570 005366          128          DEC    6.(SP);
       000006
000574 003373          129          BGT    STS$25;         JUMP OF MORE TO DO
000576 005166          130          COM    6.(SP);
       000006
000602 000755          131          BR     CNE$25;
000604 023420          132 DEC$25:  .WORD 10000.,1000.,100.,10.,0;
000606 001750
000610 000144
```

```
LOC     CODE            STMT SOURCE STMT                                                    PAGE      4


000612 000012
000614 000000
                         133 OCT$25:
000616 100000            134        .WORD 100000,10000,1000,100,10,0;
000620 010000
000622 001000
000624 000100
000626 000010
000630 000000
                         135 ;
                         136 ;
                         137 ;
                         138 ;       STORAGE AREA
                         139 ;
000632 000000            140 SUM:      .WORD 0;
000634 000000            141 N:        .WORD 0;
000636 000005            142 TABLE:    .WORD 5.,115.,23.,-46.,13.;
000640 000163
000642 000027
000644 177722
000646 000015
000650 000020            143           .WORD 16.,97.,-48.,60.,-54.;
000652 000141
000654 177720
000656 000074
000660 177712
000662 000101            144           .WORD 65.,44.,-49.,56.,13.;
000664 000054
000666 177717
000670 000070
000672 000015
000674 000002            145           .WORD 2.,19.,15.,21.;
000676 000023
000700 000017
000702 000025
000704 000022            146 MILEN:    .WORD 18.;
000706    124            147 MSG:      .ASCII  /THE SUM IS= /
000707    110
000710    105
000711    040
000712    123
000713    125
000714    115
000715    040
000716    111
000717    123
000720    075
000721    040
000722 000000            148 ASCSUM:   .WORD 0,0,0;
000724 000000
000726 000000
       000200            149           .END  SAMPLE;
```

```
                                                SYMBOL TABLE

$ICO    000336   $OCO    000330   ASCSUM 000722   BAC$25 000446   DCR$25 000500   DEC$25 000604   DNE$25 000536   ERR$25 000552
FIL$25 000512   FUL$25 000524   GO$25   000342   LOOP    000226   LPS$25 000364   MOV$25 000470   MSG     000706   MILEN   000704
N       000634   NZE$25 000460   OCT$25 000616   PC      000007   POS$25 000414   PRTRTN 000310   PUNOVR 000322   REL$25 000424
R0      000000   R1      000001   R2      000002   R3      000003   R4      000004   R5      000005   SAMPLE 000200   SP      000006
STS$25 000564   SUB$25 000432   SUM     000632   TABLE   000636   TPB     177566   TPS     177564   TST$25 000424
```

```
EXECUTION FOLLOWS



         PROCESSOR INTERRUPTED BY UNIT5
         (DEVICE INITIATED INTERRUPT)

1  9  8  6  5  2  7  4  3  0  1  2  4  5

INTERRUPT SERVICED: NORMAL PROCESSING RESUMES



REGISTER DUMP AT LOCATION  000244

    IR = 000265     MAR = 000242     MBR = 000265     PS = 0000000000010000

    R0 = 000000     R1 = 000000     R2 = 000654     R3 = 000000     R4 = 001150     R5 = 000402     R6 = 001750     R7 = 000244


REGISTER DUMP AT LOCATION  000250

    IR = 010467     MAR = 000000     MBR = 001150     PS = 0000000000010000

    R0 = 000000     R1 = 000000     R2 = 000654     R3 = 000000     R4 = 001150     R5 = 000402     R6 = 001750     R7 = 000250


         PROCESSOR INTERRUPTED BY UNIT7
         (DEVICE INITIATED INTERRUPT)

A  (  I  E  F  #  $  %  <  /

INTERRUPT SERVICED: NORMAL PROCESSING RESUMES

THE SUM IS=  616



REGISTER DUMP AT LOCATION  000306

    IR = 000000     MAR = 000306     MBR = 000000   ' PS = 0000000000000000

    R0 = 000000     R1 = 000061     R2 = 000730     R3 = 000722     R4 = 001150     R5 = 177766     R6 = 001750     R7 = 000306


000200    012706 001750 012737 000310 000064 012702 000636 012703 000003 012204 006304 012205 006305 062205 006305 060504
000240    077306 000265 010467 000362 000245 012746 000722 012746 000004 010446 004767 000046 016700 000410 012702 000706
```

```
000300    012737 000100 177564 000000 112237 177566 005300 001401 000002 005037 177564 000002 012700 000172 000402 012700
000340    000160 010446 016603 000010 016602 000006 002003 005002 005066 000006 016604 000004 012746 000040 020027 000172
000400    001405 005704 002003 005404 012716 000055 005046 062700 000002 060700 005710 001420 005001 011005 005405 060504
000440    002402 005201 000772 062004 005701 001002 005716 001762 062701 000060 010146 000756 060203 062704 000060 110443
000500    005302 003410 112643 001374 112613 005302 001410 112743 000040 000773 005726 001011 022726 000040 001011 012604
000540    012666 000004 005726 006126 000207 005726 001376 005726 016603 000010 112723 000052 005366 000006 003373 005166
000600    000006 000755 023420 001750 000144 000012 000000 100000 010000 001000 000100 000010 000000 001150 000000 000005
000640    000163 000027 177722 000015 000020 000141 177720 000074 177712 000101 000054 177717 000070 000015 000002 000023
000700    000017 000025 000022 044124 020105 052523 020115 051511 020075 033040 033061 000000 017604 000026 016114 000066
```

```
LOC     CODE            STMT SOURCE STMT                                                PAGE        1
                                                                                        DATE 06/12/75

                           1 ;
                           2 ;
                           3 ;
                           4 ;       ROUTINE $ICI TO CONVERT ASCII CHARACTERS TO INTEGERS
                           5 ;       ROUTINE $OCI TC CONVERT ASCII TO OCTAL
                           6 ;       THE CALLING SEQUENCE :  PUSH CHARACTER FIELD START ON STACK
                           7 ;                               PUSH CHARACTER FIELD LENGTH ON STACK
                           8 ;                               JSR   PC,$ICI(OR $OCI)
                           9 ;       RETURNS WITH INTEGER RESULT ON TOP OF STACK
                          10 ;       ROUTINE TAKEN FROM FPMP- 11 USERS MANUAL, PP 106-108
                          11 ;                           DIGITAL EQUIPMENT CORPORATION.
                          12 ;
                          13 ;
                          14 ;
       000000             15           R0=%0;
       000001             16           R1=%1;
       000002             17           R2=%2;
       000003             18           R3=%3;
       000004             19           R4=%4;
       000005             20           R5=%5;
       000006             21           SP=%6;
       000007             22           PC=%7;
000200 012706             23 TEST:     MOV    #1020.,SP;     INITIALIZE STACK POINTER
       001774
000204 012746             24 $OCI:     MOV    #55.,-(SP);    SET OCTAL FLAGS
       000067
000210 000402             25           BR     GO$24;
000212 012746             26 $ICI:     MOV    #471,-(SP);              SET DECIMAL FLAGS
       000471
000216 010146             27 GO$24:    MOV    R1,-(SP);                SAVE R1
000220 016601             28           MOV    8.(SP),R1;               GET STRING START
       000010
000224 066666             29           ADD    6(SP),8.(SP);            GET END + 1
       000006
       000010
000232 016666             30           MOV    4(SP),6(SP);             FIDDLE RETURN POINTER
       000004
       000006
000240 010066             31           MOV    R0,4(SP);                SAVE R0
       000004
000244 010246             32           MOV    R2,-(SP);                SAVE R2
000246 005046             33           CLR    -(SP);                   CLEAR SIGN
000250 005000             34           CLR    R0;                      CLEAR WORKSPACE
000252 000265             35           SET
000254 112102             36 STT$24:   MOVB   (R1)+,R2;                GET NEXT CHARACTER
000256 042702             37           BIC    #177600,R2;
       177600
000262 120227             38           CMPB   R2,#' ;
       000040
000266 000245             39           CLT
000270 001004             40           BNE    SGS$24;                  JUMP IF NOT BLANK
000272 020166             41           CMP    R1,12.(SP);
       000014
000276 002766             42           BLT    STT$24;                  JUMP IF MORE TO SCAN
000300 000456             43           BR     SGN$24;                  DONE
000302 105766             44 SGS$24:   TSTB   7.(SP);                  IF OCTAL CONVERSION
```

```
LOC     CODE           STMT SOURCE STMT                                              PAGE      2

        000007
000306 001002          45            BNE    SN1$24;                DO NOT PERMOT SIGNS
000310 005216          46            INC    @SP;                   OCTAL - FAKE SIGN BIT
000312 000420          47            BR     NCK$24;                GO TO PROCESS DIGIT
000314 120227          48 SN1$24:    CMPB   R2,#'+;
        000053
000320 001443          49            BEQ    FLD$24;                JUMP IF +
000322 120227          50            CMPB   R2,#'-;
        000055
000326 001012          51            BNE    NCK$24;                JUMP IF NOT -
000330 005216          52            INC    @SP;                   SET SIGN -
000332 000436          53            BR     FLD$24;
000334 112102          54 NXT$24:    MOVB   (R1)+,R2;     GET NEXT CHARACTER
000336 042702          55            BIC    #177600,R2;
        177600
000342 120227          56            CMPB   R2,#' ;
        000040
000346 001002          57            BNE    NCK$24;       JUMP IF NOT BLANK
000350 112702          58            MOVB   #48.,R2;      BLANK = ZERO
        000060
000354 120227          59 NCK$24:    CMPB   R2,#'0;
        000060
000360 002444          60            BLT    ERR$24;       JUMP IF TOO SMALL
000362 120266          61            CMPB   R2,6.(SP);
        000006
000366 003041          62            BGT    ERR$24;       JUMP IF TOO BIG
000370 162702          63            SUB    #48.,R2;      MAKE NUMERIC
        000060
000374 105766          64            TSTB   7.(SP);       OCTAL OR BINARY
        000007
000400 001441          65            BEQ    OCL$24;
000402 000265          66            SET
000404 006300          67            ASL    R0;           R0 = BASE * R0 + R2
000406 102431          68            BVS    ERR$24;
000410 160002          69            SUB    R0,R2;
000412 006300          70            ASL    R0;
000414 102426          71            BVS    ERR$24;
000416 006300          72            ASL    R0;
000420 102424          73            BVS    ERR$24;
000422 160200          74            SUB    R2,R0;
000424 000245          75            CLT
000426 102421          76            BVS    ERR$24;
000430 020166          77 FLD$24:    CMP    R1,12.(SP);
        000014
000434 002737          78            BLT    NXT$24;       JUMP IF MORE TO SCAN
000436 006026          79 SGN$24:    ROR    (SP)+;        TEST SIGN
000440 000265          80            SET
000442 103403          81            BCS    DNE$24;       JUMP IF -
000444 005400          82            NEG    R0;           MAKE +
000446 102412          83            BVS    NGM$24;       JUMP IF -NEG MAX
000450 000241          84            CLC;        SET SUCCESS FLAG
000452 012602          85 DNE$24:    MOV    (SP)+,R2;     RESTORE R2
000454 012601          86            MOV    (SP)+,R1;     RESTORE R1
000456 006126          87            ROL    (SP)+;        FLUSH FLAG AND SET C BIT IF ERROR
000460 010066          88            MOV    R0,4.(SP);    RETURN RESULT
        000004
```

```
LOC     CODE            STMT SOURCE STMT                                                  PAGE        3


000464 000245            89           CLT
000466 012600            90           MOV   (SP)+,R0;
000470 000207            91           RTS   PC;
000472 005726            92 ERR$24:   TST   (SP)+;          FLUSH SIGN
000474 005000            93 NGM$24:   CLR   R0;
000476 005166            94           COM   4,(SP);         SET ERROR FLAG
       000004
000502 000763            95           BR    ONE$24;
                         96 ;
000504 006100            97 OCL$24:   ROL   R0;             SHIFT 3 BITS LEFT,
000506 103771            98           BCS   ERR$24;         CHECKING AS YOU GO
000510 006100            99           ROL   R0;
000512 103767           100           BCS   ERR$24;
000514 006100           101           ROL   R0;
000516 103765           102           BCS   ERR$24;
000520 060200           103           ADD   R2,R0;          ADD IN THE DIGIT
000522 000742           104           BR    FLD$24;
                        105 ;
                        106 ;
       000200           107           .END  TEST;
```

SYMBOL TABLE

```
$ICI    000212  $OCI    000204  DNE$24  000452  ERR$24  000472  FLD$24  000430  GO$24   000216  NCK$24  000354  NGM$24  000474
NXT$24  000334  OCL$24  000504  PC      000007  R0      000000  R1      000001  R2      000002  R3      000003  R4      000004
R5      000005  SGN$24  000436  SGS$24  000302  SN1$24  000314  SP      000006  STT$24  000254  TEST    000200
```

APPENDIX C

MACHINE OPCODE TABLE

		NUM.OF			
S.NO	MNEMONIC	OPERANDS	LINK	OPCODE	TYPE
0	MOV	2	16	01----	D
1	MUL	2	2	070---	R
2	RTS	1	3	0002--	S
3	ASH	2	7	072---	R
4	MOVB	2	38	11----	D
5	ASHC	2	44	073---	R
6	BR	1	0	0004--	B
7	BVC	1	10	1020--	B
8	CLR	1	48	0050--	S
9	ASL	1	68	0063--	S
10	SXT	1	0	0067--	S
11	TSTB	1	15	1057--	S
12	CLRB	1	75	1050--	S
13	ASLB	1	82	1063--	S
14	RESET	0	0	000005	O
15	TRAP	1	0	001044	O
16	DEC	1	17	0053--	S
17	BNE	1	18	0010--	B
18	COM	1	19	0051--	S
19	ADD	2	84	06----	D
20	BLE	1	21	0034--	B
21	BGT	1	22	0030--	B
22	BLT	1	23	0024--	B
23	BMI	1	24	1004--	B
24	BPL	1	25	1000--	B
25	BGE	1	26	0020--	B
26	BCC	1	27	1030--	B
27	DIV	2	28	071---	R
28	BIT	2	29	03----	D
29	BIC	2	30	04----	D
30	BHI	1	31	1010--	B
31	SBC	1	32	0056--	S
32	BPT	0	33	000003	O
33	SEV	0	34	000262	O
34	SEC	0	35	000261	O
35	SEN	0	36	000270	O
36	SEZ	0	37	000264	O
37	SCC	0	0	000277	O
38	DECB	1	39	1053--	S
39	COMB	1	40	1051--	S
40	MARK	1	41	0064--	S
41	BITB	2	42	13----	D
42	BICB	2	43	14----	D
43	SBCB	1	0	1056--	S

S.NO	MNEMONIC	NUM.OF OPERANDS	LINK	OPCODE	TYPE
44	BHIS	1	45	1030--	B
45	SWAB	1	46	0003--	S
46	HALT	0	47	000000	O
47	WAIT	0	0	000001	O
48	INC	1	49	0052--	S
49	BEQ	1	50	0014--	B
50	ROL	1	51	0061--	S
51	CMP	2	52	02----	D
52	JMP	1	53	0001--	S
53	NEG	1	54	0054--	S
54	ROR	1	55	0060--	S
55	SOB	2	56	077---	R
56	XOR	2	85	074---	R
57	BCS	1	58	1034--	B
58	BLO	1	59	1034--	B
59	ADC	1	60	0055--	S
60	BIS	2	61	05----	D
61	EMT	0	62	001040	O
62	IOT	0	63	000004	O
63	CLC	0	64	000241	O
64	CLV	0	65	000242	O
65	CLN	0	66	000250	O
66	CLZ	0	67	000244	O
67	CCC	0	0	000257	O
68	JSR	2	69	004---	R
69	RTI	0	70	000002	O
70	SUB	2	71	16----	D
71	ASR	1	72	0062--	S
72	BVS	1	73	1024--	B
73	TST	1	74	0057--	S
74	RTT	0	0	000006	O
75	INCB	1	76	1052--	S
76	CMPB	2	77	12----	D
77	ROLB	1	78	1061--	S
78	RORB	1	79	1060--	S
79	NEGB	1	80	1054--	S
80	ADCB	1	81	1055--	S
81	BISB	2	0	15----	D
82	ASRB	1	83	1062--	S
83	BLOS	1	11	1014--	B
84	SET	0	20	000265	O
85	CLT	0	86	000245	O
86	NOP	0	57	000240	O

VITA

Anand Vardhan Pandit

Candidate for the Degree of

Master of Science

Thesis: SIMULATION AND APL DESCRIPTION OF THE PDP 11/40

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Hyderabad, India, June 11, 1951, the son of Mr. and Mrs. S. V. Pandit.

Education: Graduated from St. Paul's High School, Hyderabad, in May, 1966; received Bachelor of Engineering degree in Electronics and Communication Engineering from Osmania University in 1973; completed requirements for Master of Science degree at Oklahoma State University in July, 1975.