PDP-11 SIMULATION

by

JOHN L. YAFFE

a thesis

presented to the Faculty of Graduate Studies

at the University of Manitoba in partial

fulfilment of the degree Master of Science

November 1971

ABSTRACT

A PDP-11 Assembler and Loader is written in IBM System/360 Assembler Language. When linked to a PDP-11 machine-code interpreter, this effectively becomes a simulation of Digital Equipment Corporation's PDP-11 computer. As such, its purpose is to be a teaching aid for computer science students.

ACKNOWLEDGEMENTS

This document along with the actual PDP-11 Simulator is being presented as a master's thesis to the Department of Computer Science at the University of Manitoba.

I would like to thank my thesis supervisor Dr. Carol Abraham for his encouragement and guidance throughout this project, and for his helpful suggestions and advice on many technical problems. Also, I gratefully acknowledge the criticisms offered by the two referees Dr. J. C. Muzio and Prof. R. B. Pinkney, both of the University of Manitoba.

John L. Yaffe

November, 1971

CONTENTS

INTRODUCTION

At most universities today, computer science is becoming a major field of study. Aside from the mere programming aspects, students of computer science are taught the structure and operation of computers, the principles that underlie their design, and important applications of computers to society.

To a lesser extent are students actually exposed to various computer hardwares and architectures. There are obvious constraints on these educational objectives. Curriculums depend heavily upon the computer services available at the particular institution. Due to the large capital investment involved, universities rarely possess more than one large-scale computer system. University computing centres seek financial support from the business community and local government. The interests of the faculty members determine areas of specialization, and thus affect research grants. Rapid technical growth tends to make equipment obsolete within a few years. In the end, it becomes an administrative decision as to what particular computer system is installed.

It is not a desirable practice to restrict Computer Science students to one particular computer, immaterial of the manufacturer. It is more appropriate for Computer Science departments to provide an environment where students can be exposed to a number of different computer hardwares, each one representing different characteristic concepts of computer hardware design. In this way, a student is not limited or influenced by one specific hardware;

rather, he is trained to make comparisons and evaluations of various designs, and thus is better equipped to make decisions which may be part of his future responsibility.

Simulation is a means of providing extra computing facilities which could not otherwise be afforded. Since purchase of additional machines for strictly educational purposes is economically out of the question, universities have successfully simulated both real and hypothetical computers. For example, SPECTRE from the University of Waterloo and PRISM from Massachusetts Institute of Technology are hypothetical computers. Real machines currently being simulated at the University of Manitoba are the UNIVAC-1108, the CDC-6600, on-line SPECTRE, and now the PDP-11.

The PDP-11 illustrates the advanced state of the art of computing today. Although classes as a 'mini-computer', it has several powerful features not available on many larger machines. Like the Burroughs 5000/6000/7000 series, the PDP-11 has hardware stack processing which allows automatic subroutine nesting and interrupt handling, and dynamic list structures for program data. The PDP-11 has a wide range of addressing capabilities -- list sequential addressing, full address indexing, stack addressing, and direct addressing of all core memory -- which lend a unique generality to its instruction repertoire. Instructions have a variable length format, depending upon which of the eight possible addressing modes is specified. Any memory location can act as an accumulator, thus eliminating needless 'load' and 'store' instructions. It also includes a full set of instructions for character manipulation. Further, there is no all-powerfull operating system

which controls the computer's supervisory functions. Thus, through the simulator, a programmer may create his own servicing routines to handle hardware interrupts, to control input and output, and to program peripherals. In other words, students will be able to develop operating systems on the simulated machine.

Consequently, the PDP-11 simulator will become a valuable teaching aid. Students will be exposed to new hardware and software features which provide an interesting contrast with the familiar concepts of non-stack computers. The PDP-11 simulator may also be used by programmers who want to produce real PDP-11 programs to be run later on a real PDP-11 computer. All the debugging can be done on the simulator, thus speeding up program development.

The following documentation is directed to readers who have some understanding of computers and computer software. The Table of Contents provides a general outline of each major section.

The PDP-11 Assembler User's Guide is written for programmers who are unfamiliar with the PDP-11 computer. It contains a general discussion of the hardware structure, detailed descriptions of the instruction set and programming techniques, and an explanation of the assembly process. Examples and program listings are presented. Several useful Appendices are also included. With its Table of Contents, the User's Guide is a useful reference text.

The PDP-11 Assembler Logic Manual describes how the actual simulation is designed, and in particular, how the Assembler itself is organized. Certain maintenance problems are discussed,

and several suggestions are made for modifying or creating assembler features.

The conclusion discusses the role of the simulated PDP-11 system, and points out some improvements which could be made.

PDP-11 ASSEMBLER USER'S GUIDE

TABLE OF CONTENTS

SECTION A GENERAL INFORMATION

THE PDP-11 ASSEMBLER PROGRAM

Computer programs may be expressed in machine language, using numeric codes directly interpreted by the computer, or in symbolic language, using letters, numbers, and symbols meaningful to a programmer. A symbolic language, however, must be translated into machine language before the computer can execute the program. This is the function of an assembler.

PAL-11R (Program Assembly Language for the PDP-11, Relocatable version) is the symbolic language designed by Digital Equipment Corporation for the PDP-11 computer.[1] The PDP-11 Assembler, then, translates PAL-11 source statements into PDP-11 machine code.

However, PAL-11R is not a conventional assembly language. Due to the hardware features of the PDP-11 computer, the assembly process is not a simple line-by-line translation of source statements. Processing involves the detection and identification of addressing modes, the generation of index words, the assignment of storage locations to instructions, index words and program data, the performance of auxiliary functions requested by the programmer, and the loading of the machine code into main storage. Further, the assembler furnishes a printed listing of the source statements and the machine code, with additional information such as symbol tables, error diagnostics, and assembly parameters.

SIMULATED PDP-11 SYSTEM

The processing of any given PAL-11R program involves three phases occurring at distinct times in the following sequence:

1. Assembly: At assembly time, a PAL-11R source program is read and translated into PDP-11 machine language by the Assembler.

2. Loading: At load time, the machine language instructions are placed into the PDP-11 core memory.

3. Execution: At execution time, the PDP-11 Interpreter automatically identifies and carries out the machine instructions. A small supervisory program called a monitor initiates these procedures, thus forming a system able to batch-process PAL-11R source programs.

This software constitutes a simulator of a PDP-11 system. That is, although the University of Manitoba does not possess an actual PDP-11 computer, by means of this software, the IBM System/360 Model 65 appears in structure, in capability, and in operation to be a PDP-11. (Note that any PDP-11 program may become part of the operating system under this simulation.)

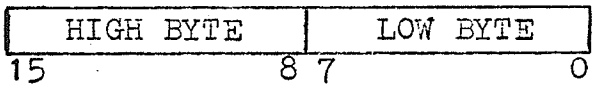
Interested readers are directed to the Assembler Logic Manual for specific techniques used in the simulation. Throughout this User's Guide, no distinctions will be made between the simulation and the actual PDP-11 computer.

HARDWARE FEATURES[2]

CORE MEMORY

A memory location is an 8-bit information unit called a byte.

The normal processing unit, called a word, is 16 bits long, and consists of two consecutive bytes.



PDP-11 WORD

Byte locations in core memory are numbered consecutively using octal notation starting with 000000. A word in storage is aligned on an even byte boundary, and is addressed by its low-order byte. The PDP-11 processor can directly access up to 32,768 words or 65,536 bytes. The maximum core memory size is 32K words, but any individual user may request the system to consider less core for running his job. (This is indicated on the $JOB card as discussed in Section E.)

GENERAL REGISTERS

The PDP-11 contains eight 16-bit registers, usually referred to as R0, R1, R2, ... R7. Each register may be used as an arithmetic accumulator, as a pointer to a memory location, or as an index register. The seventh register, R7, is used by the processor as the program counter (PC) register. The PC contains the address of the next instruction to be executed. Register R6 is known as the processor stack pointer (SP), and is used automatically in PDP-11 processor stack operations.

STACK PROCESSING

A stack is a dynamically increasing/decreasing sequential list of data which is maintained by a stack pointer (a register)

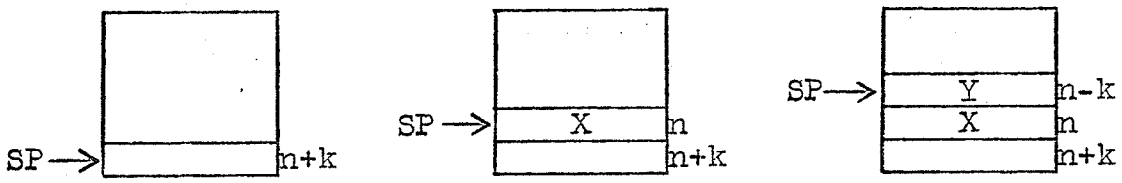
which at any time points to the beginning of the list. Such a stack is often called a 'push-down' stack or a Last-In-First-Out (LIFO) list. The following terminology is used:

1. The 'top of the stack' is the beginning of the list.

2. A 'stack pointer' always contains the address which is the current top of the stack. The stack is controlled by manipulating this pointer. The pointer is located in a register -- the processor stack pointer is R6 -- although a user may select any register as a stack pointer for a user-defined stack.

3. The 'processor stack' is used by the system in conjunction with subroutine calls and interrupts. The processor stack may also be used by a user. The user and the processor will take control of the stack at different times, thus avoiding any possible conflict.

4. To 'push-down' the stack means to enter data at the top of the stack.



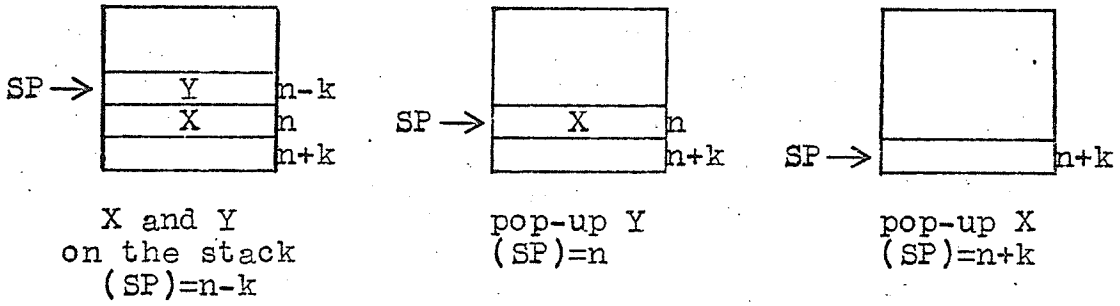
an empty stack
(SP)=n+k
k=1 for byte
k=2 for word

push-down X
(SP)=n

push-down Y
(SP)=n-k

A 'push-down' involves stepping the stack pointer to the next lower memory word (or byte), and physically entering a data word (byte) at that address. This address becomes the new top of the stack.

5. To 'pop-up' the stack means to remove an entry from the top of the stack.



X and Y on the stack (SP)=n-k

pop-up Y (SP)=n

pop-up X (SP)=n+k

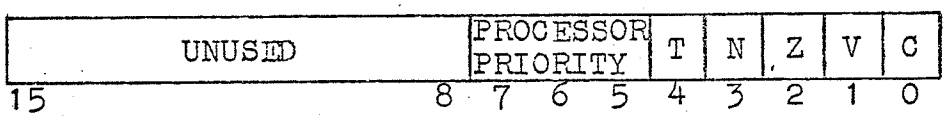
A 'pop-up' involves recovering the data pointed to by the stack pointer and increasing the stack pointer to the next higher word or byte. This address becomes the new top of the stack. Data is not physically erased; only the contents of the stack pointer are changed.

SUBROUTINE NESTING

Subroutine nesting to any depth is easily accomplished by using the stack mechanism. A user-defined stack may be generated as an argument list. The special instructions JSR (Jump to Sub-Routine) and RTS (ReTurn from Subroutine) effectively reserve and restore registers for use as stack pointers both for passing arguments and for determining the subroutine return address. These instructions are explained in Section D.

CENTRAL PROCESSOR STATUS REGISTER

The central processor status register, denoted PS, indicates the status of a program before the current instruction is executed. The PS is a reserved word in core memory with the following format:



Central Processor Status Register

Processor Priority: The current priority of the processor can be set by a programmer to any one of eight levels. This priority, indicated by bits 5, 6 and 7 of the PS, is used by the hardware interrupt system in determining whether external device interrupts gain control of the processor.

T-Bit: This is the trace bit which is useful for program debugging. If the T-bit is set, a program interrupt occurs after the execution of the current instruction using the interrupt vector at location 14 (octal). Normally an installation service routine will print out useful information about the program's status, although the user is free to write his own trace-handling routine.

Condition Codes: These four bits provide information about the result of the previous operation. The bits are set after the execution of every instruction as tabulated in Appendix E, where each bit indicates the following:

Z: set if the result was zero; cleared otherwise

N: set if the result was negative; cleared otherwise

C: set if the operation resulted in a carry from the most significant bit; cleared otherwise

V: set if the operation resulted in an arithmetic overflow; cleared otherwise

INTERRUPT HANDLING

The PDP-11 makes a logical distinction between machine instruction errors, input and output requests, installation service routines, and user-defined trap routines. They are all classed as interrupts, and they all use the same interrupt mechanism. But each cause of interrupt is associated with a fixed memory location (called a trap vector) thus eliminating the need to determine by software what was the cause of the interrupt. The interrupt system permits the processor to shift execution from any given routine to another one, and later return to the interrupted routine exactly as it left it.

A program interrupt may be caused in any of the following ways:

1. External Device Interrupt: Each peripheral device is assigned a priority level for interrupting the processor. When an external device needs the processor, a hardware interrupt occurs and control is given to the servicing routine of that device. (References to external devices have not yet been implemented in the Interpreter; consequently, no priority interrupts exist in this simulation.)

2. Machine Instruction Errors: Whenever an instruction error is detected, an interrupt is automatically generated to terminate the user program and print an octal dump of core memory.

3. User-Invoked Interrupt: A user may code an interrupt directly into his program in much the same manner as a subroutine call. The 'trap' instructions EMT (EMulator Trap) and TRAP are

used for this purpose.

A trap vector (or interrupt vector) comprises two consecutive words of memory. The first word contains the starting address of an interrupt-handling routine; the second contains a new processor status word. Each type of interrupt is associated with its own particular interrupt vector, as outlined in Appendix H. Locations 000000_8 to 000400_8 of core memory are reserved for these interrupt vectors.

The interrupt operation may be summarized as follows:

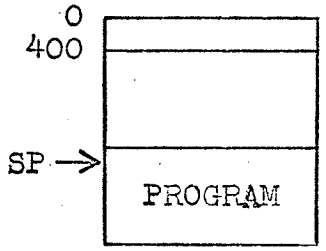
1. The contents of the program counter (PC) and the processor status register (PS) are pushed onto the processor stack.

2. A new PC and PS are loaded from the appropriate location of the interrupt vector, thereby sending control to an interrupt-handling routine, with the processor set to a new priority level.

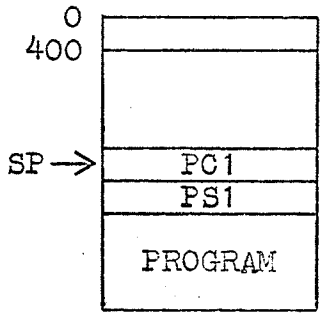
3. The interrupt-handling routine is terminated by the instruction RTI (ReTurn from Interrupt) which pops the top two words off the processor stack back into the PC and PS, returning control to the interrupted program.

An interrupt-handling routine may itself be interrupted, and this nesting of interrupts may go on to any level, limited only by the core available for the processor stack. Further, a service routine may use the processor stack for temporary data or dynamic lists provided the return mechanism is not destroyed. Figure 1 illustrates how the processor stack operates during nested interrupts. Note that the stack pointer (SP) is automatically adjusted.

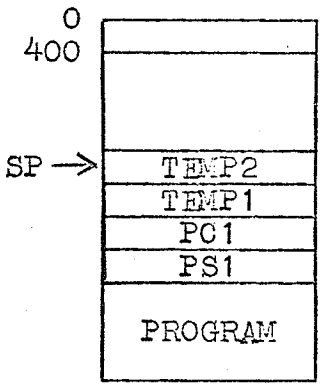
FIGURE 1 INTERRUPT OPERATIONS



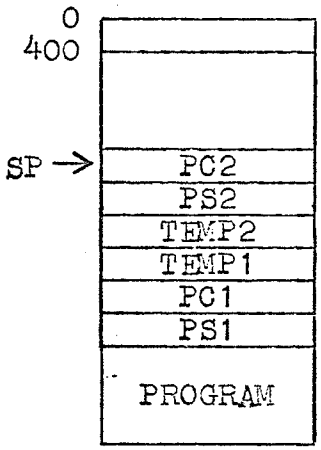
1. Routine 1 executes
(SP)=n



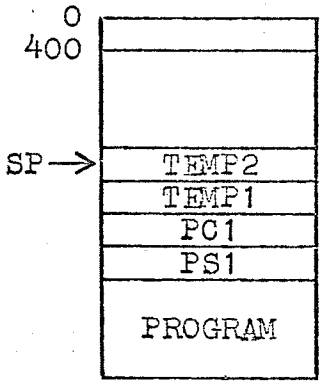
2. Interrupt Routine 1 by Routine 2
(SP)=n+4



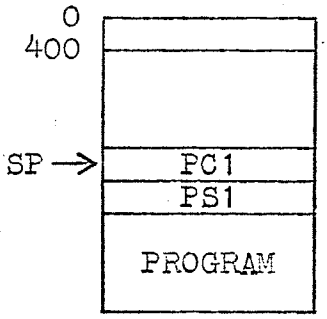
3. Routine 2 generates temporary storage
(SP)=n+8



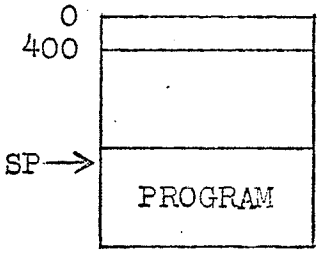
4. Interrupt Routine 2 by Routine 3
(SP)=n+12



5. Routine 3 terminates, return to Routine 2
(SP)=n+8



6. Release temporary storage
(SP)=n+4



7. Return to Routine 1
(SP)=n

ASSEMBLER FEATURES

PAL-11R ASSEMBLER LANGUAGE

PAL-11R Assembler Language provides a collection of mnemonic symbols composed of:

1. Instruction mnemonics which correspond to the PDP-11 machine language commands used at execution time;
2. Assembler directives which represent auxiliary functions to be performed by the Assembler at assembly time.

PROGRAM SECTIONING AND LINKING

The Assembler provides facilities for organizing a program into one or more parts called control sections. Control sections are assembled independently and the Assembler maintains a separate location counter and symbol table for each section.

Symbols defined in one control section may be referenced in another control section by using the .GLOBL directive. (See Section D.) Thus, a program written with different control sections may share data and transfer control among sections.

RELOCATABILITY

Program relocation is the loading of an object program into storage locations other than those originally assumed by the user. A program section is classed as relocatable or absolute respectively depending on whether it does or does not undergo program relocation. A user may specify an absolute section by means of an .ASECT directive, or a relocatable section by a .CSECT

directive. (See Section D: Program Sectioning Directives.)

Control sections are relocatable unless explicitly defined as absolute. Relocation is automatic, and in general invisible to the user. Actually, however, the Assembler will add the base address (hereafter relocatable zero) of the appropriate relocatable section to any address constants appearing in the program.

PROGRAM LOADING

In this version of the PDP-11 Assembler, the load phase is incorporated into the assembly phase. This restricts some of the capabilities of the PDP-11 simulation in that object modules cannot be loaded from auxiliary storage and relocated for execution. Such problems are discussed more fully in the Assembler Logic Manual.

All relocatable control sections are loaded contiguously at the highest locations of core memory. Any absolute sections are loaded at the locations specified by the user.

INPUT AND OUTPUT

The PDP-11 instruction set has no explicit input or output instructions. In an actual PDP-11 system, each peripheral device is assigned unique memory addresses for what are termed device status registers, control registers, and data registers. Under certain conditions, I/O hardware devices will interrupt the processor and direct its control to an appropriate input/output service routine. There is no need for device polling.

This hardware approach for I/O creates no additional problems

for the Assembler. However, at the present (1971) stage of development of the simulation, the Interpreter does not have the code to cope with peripheral devices.

In order to free the user from the details of programming peripheral devices, and to postpone the interpretation of such instructions, a set of service routines were made available in the form of extended assembly language instructions called monitor requests. Among these are the input and output macros:

1. READC - Read Character
2. READO - Read Octal
3. PRINTC - Print Character
4. PRINTO - Print Octal

ERROR MESSAGES

When a source program is assembled, it is analyzed for errors in the use of PAL-11R language. Detected errors are flagged, and a summary of the errors appears at the end of the program listing.

SECTION B PAL-11R LANGUAGE STRUCTURE

CHARACTER SET

A PAL-11R symbolic program is composed of instruction mnemonics, symbols, numbers, and separating characters (delimiters) using the following ASCII* characters:

1. the letters A through Z;
2. the digits 0 through 9;
3. the characters . and $;
4. the separating or terminating characters

   : = % # @ ( ) , ; " ' + - & ! and blank.

STATEMENTS

A statement may be composed of up to four fields which are identified by their order of appearance or by special terminating characters as explained below and summarized in Appendix B. These four fields are:

LABEL OPERATOR OPERAND COMMENT

where the LABEL and COMMENT fields are optional, and the OPERAND field depends upon the OPERATOR being used.

A symbolic program is submitted in the form of punched cards. Each source statement must be contained in columns 1 through 72 of a card, with one statement per card, and no continuations allowed. The statement fields are not associated with fixed locations on the data card.

* ASCII stands for American Standard Code for Information Interchange.

LABEL

A label is a symbolic name for a particular location within a program. The label field is optional. If a label is present, it always occurs first in a statement, and must be terminated by a colon (:). A label is a user-defined symbol which is assigned the current value of the location counter. (The location counter contains the address of the memory location where the machine code for the next instruction will be stored.) This value will be absolute or relocatable according to whether that program section is absolute or relocatable.

For example, if the current location counter is (relocatable) 40_8, the statement

```
ABC:        ADD    A,B
```

will assign the value (relocatable) 40_8 to the label ABC, so that all references to ABC will become references to location (relocatable) 40_8.

More than one label may appear within the same label field; and each label will be assigned the same value.

```
XYZ:    DD$:    A76:        ADD    A,B
```

In the above, the same value will be assigned to each of the labels XYZ, DD$, A76.

If the same label appears on any other statement within the same program section, the error M (Multiple definition of a label) will be generated.

OPERATOR

An operator is an instruction mnemonic or an assembler direc-

tive, as tabulated in Appendices E and F. An instruction mnemonic specifies what action will be performed at execution time. An assembler directive specifies a certain action to be performed during assembly time.

An operator may be preceded by one or more labels, and followed by one or more operands and/or a comment. An operator is terminated by a blank or any of the following characters:

@ (% + - & ! " ' , ;

The use of the above characters will be explained in subsequent sections.

OPERAND

Operand entries identify data to be acted upon by the operator. Operands may be symbols, expressions, or numbers. Depending on the type of instruction, one, two, or no operands may be written in the operand field. When more than one operand appears within a statement, each is separated from the next by a comma. An operand may be preceded by an operator and/or a label, and followed by a comment. Operands may represent storage locations, general registers, immediate data, or constant values.

COMMENT

Comments do not affect the assembly or the execution of a program; however, they are useful as documentation for the program listing. A comment must begin with a semi-colon (;). The comment field is optional and may contain any characters. It may be preceded by none, any or all of the other three fields.

The following are examples of comments:

```
LABEL:    MOV  X,Y      ; THIS IS A COMMENT
; THIS IS A COMMENT CARD
```

SYMBOLS

A symbol is a character or combination of characters used to represent storage locations or arbitrary integers. Symbols, by their use as labels and operands, provide a convenient way to name and reference program data. There are two types of symbols, each with its own symbol table:

1. permanent symbols
2. user-defined symbols

PERMANENT SYMBOLS

Permanent symbols consist of the instruction mnemonics and assembler directives which represent the instruction capabilities of PAL-11R. These symbols reside in a permanent part of the Assembler called the Permanent Symbol Table, and need not be defined by the programmer before being used in an Assembler source program.

USER-DEFINED SYMBOLS

User-defined symbols are created by the programmer to be used as labels and operands. These symbols are entered by the Assembler into the User Symbol Table as they are encountered during the first pass of the assembly. A string of characters is a legal user-defined symbol only if the following rules apply:

1. The first character in a symbol must not be a digit.

2. No blanks or separating characters may be included.

3. Each symbol must be unique within the first six characters. Symbols of more than six characters will be accepted; the seventh and subsequent characters will be checked for legality, but otherwise ignored by the Assembler.

A user-defined symbol may duplicate a permanent symbol without confusion:

1. When a symbol is detected in the operator field, it is assigned its corresponding machine operation code as tabulated in the Permanent Symbol Table. If no such instruction (symbol) exists, the .WORD directive is assumed (see Section D: Assembler Directives) and the symbol is considered as an operand.

2. If a symbol is detected in the operand field, it is associated with its user-defined value, if any, as found in the User Symbol Table. Failing this, the symbol is assumed permanent and is assigned an absolute value corresponding to its machine operation code. If a symbol is found to be neither user-defined nor permanent, it is assigned the value (relocatable) zero, and is flagged as undefined.

GLOBAL SYMBOLS

Global symbols are user-defined symbols which also appear in the .GLOBL assembler directive. (See Section D.)

NUMBERS

Numbers are self-defining terms which provide a means of specifying values without using symbolic names. A number is

classified as absolute since its value does not change during relocation. A number is transformed during the first pass of the assembly into its 16-bit binary equivalent. If that number requires more than 16 bits, it is truncated on the left, that is, its high order bits are ignored, and flagged with the error T (Truncation error.) Each number is calculated as soon as it is encountered, and no symbol table entry is associated with it. The Assembler recognizes two different types (number base) of numbers, octal and decimal.

OCTAL NUMBER

An octal number consists of the digits 0 through 7 only. Each octal digit is assembled as a 3-bit binary code:

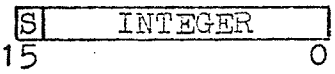
0 - 000	4 - 100
1 - 001	5 - 101
2 - 010	6 - 110
3 - 011	7 - 111

DECIMAL NUMBER

A decimal number is written as a signed or unsigned sequence of decimal digits followed by a decimal point (.). A number containing the digits 8 or 9 but not terminated by a decimal point is still interpreted as decimal, but the error message N (Number error) is generated.

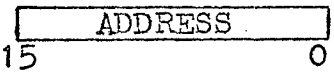
DATA FORMAT

1. All numbers are treated as word quantities including a sign bit and 15 binary integer bits.



Positive numbers are stored in true binary form with a sign bit of 0. Negative numbers are stored in two's complement form with a sign bit of 1. The two's complement of a binary number is obtained by inverting each bit of the binary representation and adding one to it. In byte operations, a full word value is calculated, but truncated to the low-order byte.

2. All addresses are assumed to be positive integers and are stored as 16-bit true binary numbers with no sign bit.



3. Logical and character data is stored as unstructured bytes.

DIRECT ASSIGNMENT

A direct assignment statement defines a symbol by assigning to it the value and relocatability attributes of an expression in the operand field.

SYMBOL= EXPRESSION

where the following rules apply:

1. An equal sign (=) must terminate the symbol name.

2. A direct assignment statement may be preceded by a label and/or followed by a comment.

3. Only one symbol may be defined by any one direct assignment statement.

4. Only one level of forward referencing is allowed. An

An example of two levels of forward referencing is:

```
X=Y
Y=Z
Z=5
```

At the end of Pass 1, X and Y are undefined, although Z is defined as 5. Consequently, in Pass 2, 'X=Y' cannot be evaluated. This causes the error message U (Undefined symbol.)

A symbol may be redefined by another direct assignment statement. However, a symbol may not be defined both by direct assignment and as a label. Such action will be flagged as a D-error (Doubly defined symbol.)

It should be understood that direct assignment statements are non-executable instructions. The symbols are defined at assembly time only. No machine instructions are generated for execution time.

REGISTER SYMBOLS

A register symbol is a symbolic name for a register and is defined by direct assignment. The eight general registers of the PDP-11 are identified by the numbers 0 to 7. Thus, the defining expression for a register symbol must be absolute and in the range 0 to 7. In addition, at least one term in the expression must either be preceded by a % sign, or be a previously defined register symbol.

```
R0=%0        ; DEFINE R0 AS REGISTER 0
R1=R0 + 1    ; DEFINE R1 AS REGISTER 1
R5= 3 + %2   ; DEFINE R5 AS REGISTER 5
```

The percent sign, %, indicates a reference to a register. In fact, the % may appear in any expression in any instruction

where a register symbol is required.

```
CLR  %4        ;CLEAR REGISTER 4
CLR  4         ;CLEAR MEMORY LOCATION 4
```

A register symbol must be defined before it is referenced. Otherwise, the Assembler may interpret the statement in a way not intended by the programmer.

ASSEMBLY LOCATION COUNTER

The location counter is used by the Assembler during assembly of a program to assign storage addresses to program statements. It is the Assembler's equivalent to the program counter at execution time. As each instruction is assembled, the location counter is incremented by the length of the assembled item. Thus, it always points to the next available storage location. Any label that is encountered, then, is assigned the value of the location counter before this incrementing occurs. In this way, a label is seen as a symbolic address whose numerical equivalent (the value of the location counter) is the address of the first byte of the machine instruction being assembled.

The period (.) is the permanent symbol for the location counter and may be used in any expression in PAL-11R. For example, storage locations may be reserved in a program by advancing the location counter.

```
.= . + 20.        ;RESERVE 20 BYTES OF MEMORY
MOV   .,R5        ;LOAD THE MOV INSTRUCTION INTO R5
```

The location counter has a mode associated with it: it is absolute if it appears in an absolute program section (see Program Section-

ing Directives); otherwise, it is relocatable.

EXPRESSIONS

An expression is composed of a single term, or an arithmetic or logical combination of terms. A term may be a permanent symbol, a user-defined symbol, a number, or the location counter. An expression is evaluated term by term from left to right and reduced to a single word quantity by the Assembler. Parentheses are not allowed within an expression.

ARITHMETIC AND LOGICAL OPERATORS

The arithmetic operators are:

+	addition or a positive number
-	subtraction or a negative number

The logical operators are:

&	logical AND
!	logical inclusive OR

AND	OR
0 & 0 = 0	0 ! 0 = 0
0 & 1 = 0	0 ! 1 = 1
1 & 0 = 0	1 ! 0 = 1
1 & 1 = 1	1 ! 1 = 1

A missing term or expression is interpreted as a zero. A missing operator is interpreted as a plus. The error code Q (Questionable syntax) is generated for a missing operator.

```
X + - 100          ;MISSING OPERAND
```

is evaluated as X plus 0 minus 100_8.

ASCII CONVERSION

'	ASCII byte
"	ASCII word

1. The apostrophe (') assigns the 7-bit ASCII value (Appendix A) of the character following it.

```
'A      ;EVALUATED AS 101₈
```

2. The quotation mark (") forms a word quantity from the two characters following it as shown below:

   a. The low byte is the ASCII value of the first character.

   b. The high order byte is the ASCII value of the second character.

   c. Any additional characters are ignored.

```
"BC     ;EVALUATED AS 041502₈
```

where	high byte	low byte
	01000011	01000010
	C	B

MODE OF EXPRESSIONS

A term is either absolute, relocatable in the current program section, or relocatable in another program section. Note that there are no external symbols since previously assembled programs can not be loaded into core from auxiliary storage. Numbers, permanent symbols, and generated data are treated as absolute terms.

Similarly, expressions are absolute or relocatable according to the following rules:

Absolute:

1. absolute term preceded optionally by a plus or minus sign
2. relocatable expression minus a relocatable term belonging to the same program section

3. any combination of absolute terms

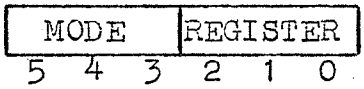
Relocatable:

1. a relocatable term
2. a relocatable expression plus or minus an absolute expression
3. an absolute expression plus a relocatable expression

Relocatable terms from different program sections may not appear in the same expression. Also, logical operations involving two relocatable terms are illegal. These errors are flagged by the message A (Addressing error.)

SECTION C ADDRESSING MODES

PDP-11 machine instruction words contain a six-bit address field divided into two 3-bit subfields which specify the general register and the mode of calculating the operand address.



Address Field

The register subfield identifies which of the eight general registers is to be used in the address calculation. The mode subfield indicates how this register is to be used.

The following conventions are used throughout this section:

a. E represents any expression.

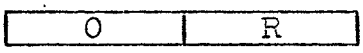
b. R represents a register expression.

c. ER represents a register expression or an absolute expression in the range 0 to 7.

d. A is a six-bit address field as described above.

e. Examples are provided using the clear instruction CLR which zeroes out the operand location. (operation code 005000_8)

REGISTER MODE

Address Field: 

Format: R

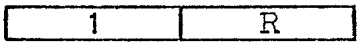
Description: The register contains the operand. The PDP-11 general registers are located in 'fast' memory,

hence operations involving registers as operands have a definite speed advantage.

Example:

```
000001      R1= %1      ;DEFINE REGISTER 1 AS R1
005001      CLR R1      ;CLEAR REGISTER 1
```

DEFERRED REGISTER MODE

Address Field: 

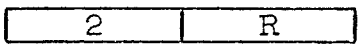
Format: @R or (ER)

Description: The register contains the address of the operand. The separating character '@' indicates to the Assembler that the following expression is a pointer to an operand address. In this case, it is the programmer's responsibility to ensure that the register involved actually will contain the required address at execution time.

Example:

```
                CLR @R1       ; CLEAR THE WORD AT THE
                  or          ; ADDRESS CONTAINED IN
005011          CLR (R1)      ; REGISTER 1.
                  or
                CLR (1)
```

AUTOINCREMENT MODE

Address Field: 

Format: (ER)+

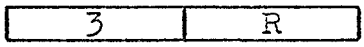
Description: The contents of the register are incremented immediately after being used as the address of the operand. Autoincrement addressing provides automatic increasing of a pointer through a sequential list or table of operands, and therefore it facilitates the hardware stack pro-

cessing. For both increment modes, the registers will normally be incremented by two, which is the implied length of the operand. However, for byte manipulation (see Section D) the increment will be one. Registers 6 and 7 are incremented or decremented always by two.

Example:

```
        005021      CLR (R1)+       ;CLEAR WORDS AT THE ADDRESSES
        005024      CLR (R1+3)+     ;INDICATED BY THE CONTENTS OF
                                    ;REGISTERS 1 AND 4 AND INCREMENT
                                    ;THESE REGISTERS BY 2
```

DEFERRED AUTOINCREMENT MODE

Address Field: 

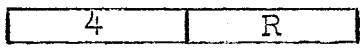
Format: @(ER)+

Description: The register contains a pointer to the address of the operand. The contents of the register are incremented after being used. This mode is most useful in subroutines where arguments are typically passed in the form of address constants.

Example:

```
        005032      CLR @(2)+       ;REGISTER 2 POINTS TO A MEMORY
                      or            ;LOCATION WHICH CONTAINS THE
                    CLR @(R2)+      ;ADDRESS OF THE WORD TO BE
                                    ;CLEARED
```

AUTODECREMENT MODE

Address Field: 

Format: -(ER)

Description: The contents of the register are first decreased by two (for byte operations, they are decreased by one); then the register contents are used as

the address of the operand. This mode is used to push data onto a stack.

Example:

```
005041      CLR -(R1)       ;DECREMENT CONTENTS OF REGISTERS
005043      CLR -(R1+2)     ;1, 3 AND 4 BY TWO BEFORE USING
005044      CLR -(4)        ;THEM AS ADDRESSES OF WORDS TO
                            ;CLEAR
```

DEFERRED AUTODECREMENT MODE

Address Field: | 5 | R |

Format: @-(ER)

Description: The contents of the register are decremented before being used as a pointer to the address of the operand.

Example:

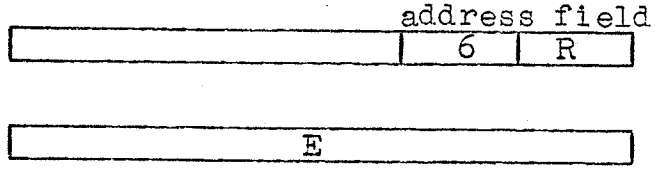
```
005052      CLR @-(2)       ;DECREASE REGISTER 2 BY TWO
                            ;BEFORE USE AS A POINTER TO
                            ;A WORD TO BE CLEARED
```

INDEX MODE

Address Field: | 6 | R |

Format: E(ER)

Description: The operand address is calculated as the sum of the value E plus the contents of the register ER. The value of the expression E is calculated by the Assembler and stored as an index word in the instruction stream at the next location.



The value E is called the base, and the contents of register ER are called the index. At execution time, the base is fixed, and the index may vary under program control. Any register (0 to 7) may be used as an index register. This mode permits random access of data in tables or stacks.

Example: Suppose X is location 126_8.

```
005061      CLR  X-4(R1)      ;CLEAR THE WORD AT ADDRESS
000122                        ;X-4 PLUS THE CONTENTS OF
                              ;REGISTER 1
```

DEFERRED INDEX MODE

Address Field:

7	R

Format: @E(ER)

Description: A pointer to the address of the operand is calculated as the sum of the expression E and the contents of the register ER. The Assembler generates an index word containing the value E as above. This mode can access data from stacks of address constants.

Example: Suppose R2 contains 600_8 and location 600_8 contains 714_8.

```
005072      CLR  @24(R2)      ;LOCATION 740₈ IS CLEARED
```

The program counter (PC) may be used with any of the above addressing modes. There are four special formats associated with the PC. The double operand instruction MOV (which moves the first operand to the second operand location, operation code 010000_8) will be used in the examples.

IMMEDIATE MODE

Address Field: | 2 | 7 |

Format: #E

Description: The operand itself is stored as an index word and is accessed by autoincrement addressing through the program counter. At execution time, whenever an instruction is fetched, the PC points to the word following that instruction. In this case, the word following the instruction is the operand.

INSTRUCTION
IMMEDIATE DATA

When the operand is fetched, the PC is again incremented by two, and will point to the next instruction. Even in byte instructions, a full word is assembled for immediate operands so that instructions are always fetched from even byte locations.

Example:

```
012702      MOV  #120,R2      ;LOAD 120(8) INTO R2
000120
```

ABSOLUTE MODE

Address Field: | 3 | 7 |

Format: @#E

Description: This is deferred autoincrement using the PC. The word following the instruction is used as the address of the operand. As in immediate mode, the Assembler stores the value of the expression as an index word in the instruction stream.

Example: Suppose A is stored in location 412_8.

```
013704      MOV  @#A,R4      ;LOAD A INTO R4
```

RELATIVE MODE

Address Field:

6	7

Format: E

Description: This is index mode using the PC. An index word is generated containing the displacement between the operand address and the program counter.

INSTRUCTION
E-.-2

But at execution time, after the index word is fetched, the PC contains the address of the word following the index word. Thus the displacement is calculated by the Assembler as:

E - . - 2

This is called relative mode since the address is calculated relative to the current PC.

Example: Suppose .=100 (octal), A is location 120_8, and B is location 124_8.

```
100:  016767      MOV   A,B      ;MOVE LOCATION 120(8)
102:  000014                     ;TO LOCATION 124(8)
104:  000016
```

DEFERRED RELATIVE MODE

Address Field:

7	7

Format: @E

Description: This is deferred index mode using the PC.

The Assembler calculates and stores an index word as in relative mode. Location E is a pointer to the operand address.

Example: Suppose .=36_8 and BCD is location 64_8.

```
36:  005077      CLR  @BCD    ;CLEAR THE WORD WHOSE ADDRESS
40:  000022                   ;IS IN LOCATION BCD
```

ADDRESSING SUMMARY[3]

The following modes do not increase the instruction length:

FORMAT	MODE	NAME
R	0r	register
@R or (ER)	1r	deferred register
(ER)+	2r	autoincrement
@(ER)+	3r	deferred autoincrement
-(ER)	4r	autodecrement
@-(ER)	5r	deferred autodecrement

The following modes add one word to the instruction length:

FORMAT	MODE	NAME
E(ER)	6r	index
@E(ER)	7r	deferred index
#E	27	immediate
@#E	37	absolute
E	67	relative
@E	77	deferred relative

SECTION D PAL-11R LANGUAGE STATEMENTS

INSTRUCTION MNEMONICS

SYMBOLIC FORMATS

The set of machine instructions for the PDP-11 computer are expressed by symbolic (mnemonic) instructions. Symbolic instructions encountered during assembly are translated into executable machine commands. The Assembler groups these instruction mnemonics into seven classes according to their symbolic format.

The following notation is used in this section:

OP represents a PAL-11R instruction mnemonic.

R is a register expression.

E is an expression.

ER is a register expression or an absolute expression in the range 0 to 7.

A is any operand specifying an address mode as described in the preceding section and summarized in Appendix C.

Listed below are the instruction classes and symbolic formats:

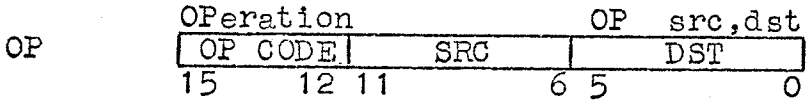
Instruction Class	Operand Field
double operand	OP A,A
single operand	OP A
operate	OP
branch	OP E where $-128 \leq \frac{E-.-2}{2} \leq 127$
subroutine call	JSR ER,A
subroutine return	RTS ER
trap	OP or OP E where $0 \leq E \leq 377_8$

The symbolic instruction formats are closely related to the machine instruction formats of the PDP-11 as shown in Appendix D.

In the following sections, each instruction will be discussed in terms of its symbolic mnemonic, its English equivalent, its machine code, and its operation.[4] In most cases, examples will also be included.

DOUBLE OPERAND INSTRUCTIONS

Double operand instructions are represented as follows:



where src - the source operand of the mnemonic

dst - the destination operand of the mnemonic

SRC - the source operand address field of the machine code

DST - the destination operand address field of the machine code

Instructions of this class include:

1. Arithmetic operations:

MOV(B)	MOVe (Byte)
CMP(B)	CoMPare (Byte)
ADD	ADD
SUB	SUBtract

2. Boolean operations:

BIC(B)	BIt Clear (Byte)
BIS(B)	BIt Set (Byte)
BIT(B)	BIt Test (Byte)

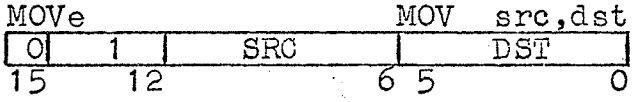
ARITHMETIC OPERATIONS

The following instructions perform fixed point binary arithmetic on their operands, which may be addresses, constants,

or immediate data. A fixed-point number or integer consists of a sign bit and a 15-bit binary integer field.

Negative numbers are stored in two's complement form. For byte-operations in register mode, only the low order byte of the specified register is used.

MOV



Description: The source operand is placed in the destination location. The previous contents of the destination are lost. The contents of the source are not affected.

Condition Codes:
- Z - set if the source operand is zero, cleared otherwise
- N - set if the source operand is negative, cleared otherwise
- C - not affected
- V - cleared

Examples: The MOV instruction typifies the capabilities of all double operand instructions by its generality. Depending on the addressing modes chosen, MOV may be used to load or store a register, push or pop a stack, and transfer data register-to register or memory-to-memory.

```
MOV   B,R1      ;LOAD REGISTER 1 WITH THE CONTENTS OF B
MOV   R1,C      ;STORE REGISTER 1 IN LOCATION C

MOV   #10,R2    ;LOAD IMMEDIATE DATA INTO REGISTER 2
MOV   #123,X    ; OR INTO A MEMORY LOCATION
```

```
MOV  B,-(SP)     ; PUSH B ONTO THE STACK
MOV  (SP)+,C     ; POP C OFF THE STACK

MOV  R2,R3       ; LOAD REGISTER 2 INTO REGISTER 3
MOV  X,Y         ; MOVE CONTENTS OF LOCATION X INTO
                 ;  LOCATION B
```

MOVB — MOVe Byte — MOVB src,dst

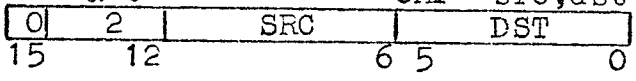
Description: MOVB operates on bytes exactly as MOV operates on words. However, with a destination in register mode, MOVB moves the source byte into the low order byte (bits 7-0) of the indicated register and extends the sign bit (bit 7) through the high order byte (bits 15-8). This is known as sign extension.

Condition Codes: set on the byte result as in MOV

Example:

```
MOVB  #7,R1      ;LOAD REGISTER 1 WITH 7
```

CMP — CoMPare — CMP src,dst



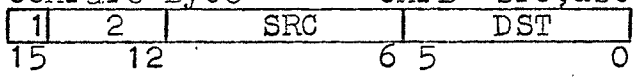
Description: The source operand is compared with the destination operand and the result determines the condition code. Neither operand is changed. Internally, the destination is subtracted from the source, and the result is compared to zero.

Condition Codes:
- Z - set if the opernads are equal, cleared otherwise
- N - set if the source operand is lower than the destination operand, cleared otherwise
- c - set if there was a carry, cleared otherwise
- V - set if there was arithmetic overflow, cleared otherwise

Examples:

```
CMP   R0,R1      ;COMPARE REGISTER TO REGISTER
CMP   #100,R1    ;COMPARE IMMEDIATE TO REGISTER
CMP   B,C        ;COMPARE MEMORY TO MEMORY
CMP   R1,B       ;COMPARE REGISTER TO MEMORY
```

CMPB — CoMPare Byte — CMPB src,dst



Description: Same as CMP

Condition Codes: Set on the byte result as CMP

ADD

Description: The source operand is added to the destination operand and the result is stored at the destination address. The original contents of the destination are lost. The contents of the source are not affected.

Condition Codes:
- Z - set if the result is zero, cleared otherwise
- N - set if the result is negative, cleared otherwise
- C - set if there was a carry from the most significant bit of the result, cleared otherwise
- V - set if there was arithmetic overflow, cleared otherwise

Examples:

```
ADD   X,R1       ;ADD X TO REGISTER 1
ADD   R2,Y       ;ADD REGISTER 2 TO LOCATION Y
ADD   R3,R4      ;ADD REGISTER 3 TO REGISTER 4
```

Arithmetic operations can be performed directly in memory locations, thereby saving needless loading and storing of accumulators.

```
ADD   A,B        ;ADD LOCATION A TO LOCATION B
```

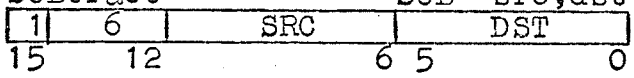
Immediate addition may be used either in registers or in memory whenever a constant is required.

```
ADD  #25.,R1    ;ADD 25 TO REGISTER 1
ADD  #10.,C     ;ADD 10 TO LOCATION C
```

Addition may be useful in processing stacks.

```
ADD  (SP)+,(SP)  ;REPLACE THE TOP TWO ELEMENTS
                 ;OF THE STACK BY THEIR SUM
```

SUB SUBtract SUB src,dst



Description: The source operand is subtracted from the destination operand and the result is stored at the destination address. The original contents of the destination are lost. The contents of the source are not affected.

Condition Codes:
- Z - set if the result is zero, cleared otherwise
- N - set if the result is negative, cleared otherwise
- C - cleared if there was a carry in the result, set otherwise
- V - set if there was arithmetic overflow, cleared otherwise

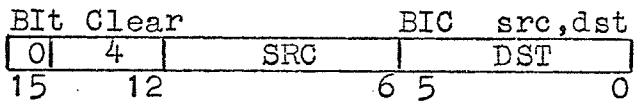
Examples:

```
SUB  @R1,@R2    ;SUBTRACT THE WORD WHOSE ADDRESS
                ;IS IN REGISTER 1 FROM THE WORD
                ;WHOSE ADDRESS IS IN REGISTER 0
SUB  (SP)+,(SP) ;REPLACE THE TOP TWO ENTRIES ON
                ;THE STACK BY THEIR DIFFERENCE
```

BOOLEAN OPERATORS

The following instructions perform operations on data at the bit level. The source operand is used as a 16-bit or 8-bit mask when used in the word or byte instruction respectively. The same mask may be used to set, clear or test the state of particular bits in a word (byte).

BIC



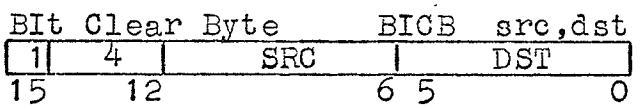
Description: The BIC instruction clears each bit in the destination that corresponds to a set bit in the source. The original contents of the destination are lost. The contents of the source are not affected.

Condition Codes:
- Z - set if the result is zero, cleared otherwise
- N - set if the high-order bit of the result is 1, cleared otherwise
- C - not affected
- V - cleared

Examples: Suppose the word X contains 177777_8.

```
BIC  #123456,X    ; X BECOMES 054321 (8)
BIC  X,X          ; X IS REPLACED BY (8) ZEROS
```

BICB



Description: Same as BIC

Condition Codes: Set on the byte result as in BIC

BIS

Description: The BIS instruction sets each bit in the destination that corresponds to a bit set in the source. The original contents of the destination are lost. The source is not affected. This is the boolean 'OR' operation.

Condition Codes: Z - set if the result is zero, cleared otherwise
N - set if the high-order bit of the result is set, cleared otherwise
C - not affected
V - cleared

Example: BIS is used to set particular bits to one. Suppose the word X contains 000102_8.

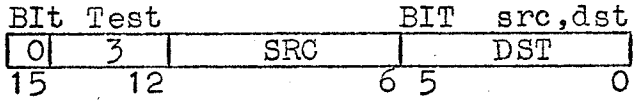
```
MASK= 100001
BIS  #MASK,X      ; X BECOMES 100103(8)
```

BISB

Description: Same as BIS

Condition Codes: Set on the byte result as in BIS

BIT



Description: The state of the destination operand bits as selected by the mask (source operand) determines the condition code. A mask bit of one indicates that the corresponding destination bit is to be tested. When a mask bit is zero, that destination bit is ignored. Neither the source nor the destination operand is changed.

Condition Codes: Z - set if the result is zero, cleared otherwise
N - set if the high-order bit of the result is set, cleared otherwise
C - not affected
V - cleared

Example: BIT checks whether specific bits in a destination word are set.

```
        BIT  #177400,R1    ; Z-BIT SET ONLY IF R1 HAS
                           ; A HIGH BYTE OF ZEROS
        BIT  #100001,B     ; B IS AN EVEN POSITIVE
                           ; INTEGER IF Z-BIT IS SET
```

BITB

Description: Same as BIT

Condition Codes: Set on the byte result as in BIT

Example: Suppose storage location 4000_8 contains 373_8 and register 5 contains 3772_8.

```
        BITB  #303,6(R5)
```

where the operand is 373_8 or 11111011_2
and the test mask is 303_8 or 11000011_2
with the result $11----11_2$

Z	N	C	V
0	1	-	0

SINGLE OPERAND INSTRUCTIONS

Single operand instructions are represented as follows:

OP

Instructions of this class include:

1. General operations:

CLR(B)	CLeaR (Byte)
INC(B)	INCrement (Byte)
DEC(B)	DECrement (Byte)
NEG(B)	NEGate (Byte)
COM(B)	COMplement (Byte)
TST(B)	TeST (Byte)

2. Multiple precision operations:

ADC(B)	ADd Carry (Byte)
SBC(B)	SuBtract Carry (Byte)

3. Rotates: ROR(B) ROtate Right (Byte)
ROL(B) ROtate Left (Byte)
SWAB SWAp Bytes

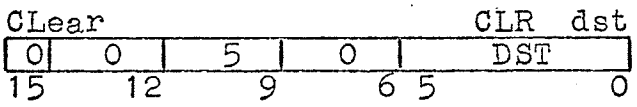
4. Shifts: ASR(B) Arithmetic Shift Right (Byte)
ASL(B) Arithmetic Shift Left (Byte)

5. Jump Instruction: JMP JuMP

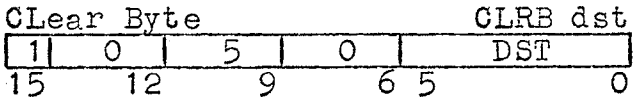
GENERAL OPERATIONS

General operations may perform their arithmetic calculations on either a word or a byte operand. Henceforth, the corresponding word and byte mnemonic will be presented together. However, a word instruction requires a word operand, and in deferred modes must specify an even-byte word address. A byte instruction uses a byte operand, and any address (even/odd) is suitable.

CLR



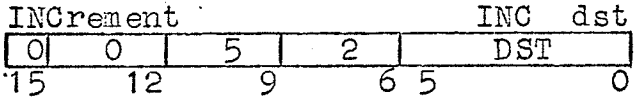
CLRB



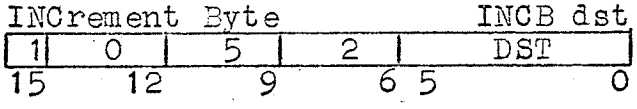
Description: A word (byte) of zeros is inserted at the operand address. The previous contents of the operand are lost.

Condition Codes: Z - set
N - cleared
C - cleared
V - cleared

INC



INCB



Description: The word (byte) at the destination address is incremented by one. For INCB, the carry from a byte does not affect any other byte. Thus, in register mode, only the low-order byte of the register is incremented.

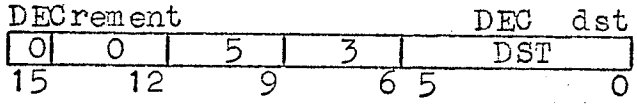
Condition Codes:
- Z - set if the result is zero, cleared otherwise
- N - set if the result is negative, cleared otherwise
- C - not affected
- V - set if the operand was 077777_8, cleared otherwise

Example: An instruction of the form

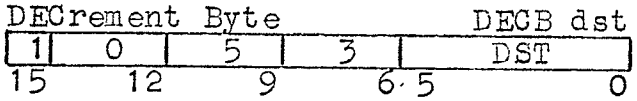
```
INC  TABLE(R4)
```

may be used to generate an array of sums (TABLE) where entries to be incremented are selected by the index register R4.

DEC



DECB

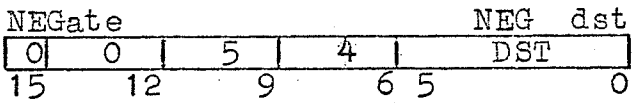


Description: The word (byte) at the destination address is decremented by one. For DECB in register mode, only the low-order byte of the register is decremented, but if necessary, the bits 15-8 may be changed to represent the sign extention of the result in bits 7-0.

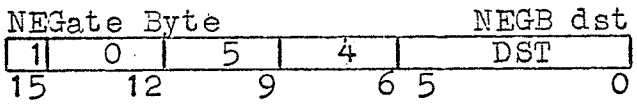
Condition Codes:
- Z - set if the result is zero, cleared otherwise
- N - set if the result is negative, cleared otherwise
- C - not affected
- V - set if the operand was 100000_8, cleared otherwise

Example: INC and DEC are commonly used to control program looping. See the examples under BRANCH instructions.

NEG



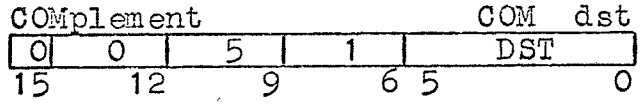
NEGB



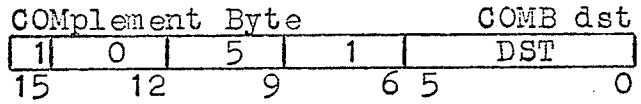
Description: The two's complement of the destination word (byte) replaces the operand. For NEG, the value 100000_8 is replaced by itself since there is no positive counterpart for the most negative number.

Condition Codes:
- Z - set if the result is zero, cleared otherwise
- N - set if the result is negative, cleared otherwise
- C - cleared if the result is zero, set otherwise
- V - set if the result is 100000_8, cleared otherwise

COM



COMB

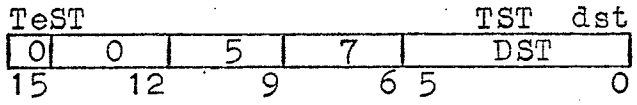


Description: COM(B) replaces the word (byte) contents of the destination address by its logical complement. That is, each bit equal to 0 is set, and each bit equal to 1 is cleared.

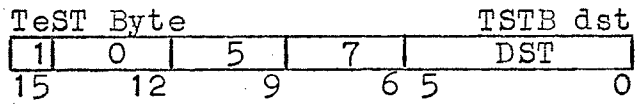
Condition Codes:
- Z - set if the result is zero, cleared otherwise

N - set if the most significant bit of the result is set, cleared otherwise
C - set
V - cleared

TST



TSTB



Description: The condition codes are set according to the contents of the destination address.

Condition Codes:
Z - set if the result is zero, cleared otherwise
N - set if the result is negative, cleared otherwise
C - cleared
V - cleared

Example: The TST instruction is equivalent to

```
CMP  dst,#0
```

It may be used to set up a three-way branch by testing the result of previous calculations, or comparing elements in an array to zero.

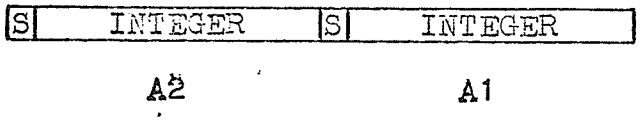
Suppose the array TABLE is stored in location 144_8.

```
012702      MOV  #TABLE,R2    ;GET THE ARRAY ADDRESS
000144
005722      TST  (R2)+        ;COMPARE AN ARRAY ENTRY TO
                              ;ZERO AND RESET R2 TO THE
                              ;NEXT ENTRY
```

MULTIPLE PRECISION OPERATIONS

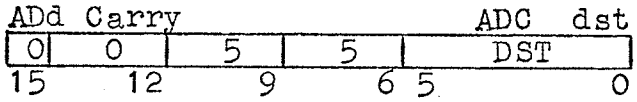
Often it is necessary to do arithmetic on operands considered as multiple words. Suppose A2 and A1 are assigned to consecutive PDP-11 word locations. These two words may be considered logically as a double precision integer with two sign bits and 30 binary

integer bits as follows:

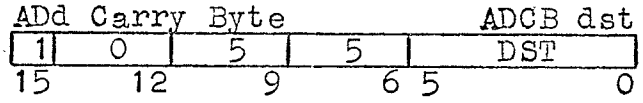


where A1 is the low-order word, and A2 is the high-order word. Although there are no explicit instructions for double precision arithmetic (as in the IBM/360), PDP-11 facilitates such operations by means of the following instructions:

ADC



ADCB



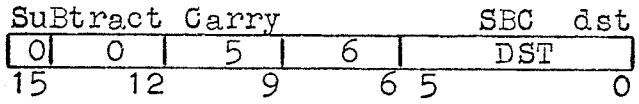
Description: The contents of the C-bit in the processor status register is added to the destination. In this way, the carry from an addition may be recovered in a subsequent addition.

Condition Codes:
- Z - set if the result is zero, cleared otherwise
- N - set if the result is negative, cleared otherwise
- C - set if the operand was 177777_8 and (C) was 1, cleared otherwise
- V - set if the operand was 077777_8 and (C) was 1, cleared otherwise

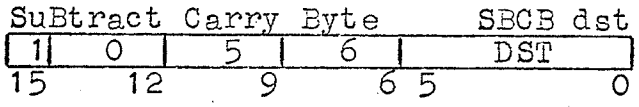
Example: Double precision addition may be accomplished by the following sequence of instructions where A1, A2 and B1, B2 are consecutive words as described above:

```
ADD   A1,B1    ;ADD LOW ORDER WORDS
ADC   B2       ;ADD CARRY INTO HIGH ORDER
ADD   A2,B2    ;ADD HIGH ORDER WORDS
```

SBC



SBCB



Description: The contents of the C-bit in the central processor status register are subtracted from the destination. Thus, the carry from a subtraction may be recovered for a multiple precision result.

Condition Codes:
- Z - set if the result is zero, cleared otherwise
- N - set if the result is negative, cleared otherwise
- C - cleared if the result is zero and (C) is 1, set otherwise
- V - set if the result is 100000_8, cleared otherwise

Examples: Double precision subtraction may be done as follows:

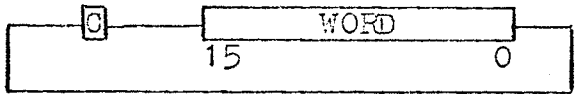
```
SUB  A1,B1   ;SUBTRACT LOW ORDER PARTS
SBC  B2      ;SUBTRACT CARRY FROM HIGH ORDER
SUB  A2,B2   ;SUBTRACT HIGH ORDER PARTS
```

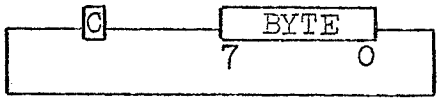
Double precision negation is accomplished by:

```
NEG  A1      ;NEGATE LOW ORDER WORD
SBC  A2      ;ADJUST FOR CARRY
NEG  A2      ;NEGATE HIGH ORDER WORD
```

ROTATES

Rotate operations are useful for examining and testing the bit structure of a word or byte. The C-bit of the processor status register is appended to the destination operand by circular bit-shifting.





ROR

ROtate Right ROR dst

0	0	6	0	DST
15	12	9	6 5	0

RORB

ROtate Right Byte RORB dst

1	0	6	0	DST
15	12	9	6 5	0

Description: All bits of the destination are rotated right one place. Bit 0 is loaded into the C-bit of the processor status register, and the previous contents of the C-bit are loaded into bit 15 (bit 7) of the destination word (byte).

Condition Codes:
- Z - set if all bits of the result are zero, cleared otherwise
- N - set if the high-order bit of the result is 1, cleared otherwise
- C - loaded with the low-order bit of the destination
- V - set if either the new N-bit or C-bit is 1, but not both (viz 'Exclusive OR' of N and C), cleared otherwise

ROL

ROtate Left ROL dst

0	0	6	1	DST
15	12	9	6 5	0

ROLB

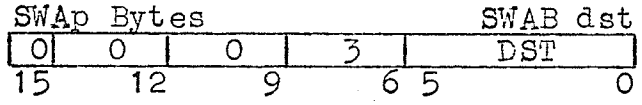
ROtate Left Byte ROLB dst

1	0	6	1	DST
15	12	9	6 5	0

Description: All bits of the destination are rotated left one place. Bit 15 (bit 7) is loaded into the C-bit of the processor status register, and the previous contents of the C-bit are loaded into bit 0 of the destination.

Condition Codes:
- Z - set if all bits of the result are zero, cleared otherwise
- N - set if the high-order bit of the result is 1, cleared otherwise
- C - loaded with the high-order bit of the destination
- V - set as the Exclusive OR of N and C

SWAB



Description: The low-order byte and the high-order byte of the destination word are interchanged. Note that SWAB is a word instruction, so the destination must be a word (even) address.

Condition Codes:
- Z - set if the low-order byte of the result is zero, cleared otherwise
- N - set if bit 7 of the result is 1 (viz the high-order bit of the low-order byte is 1), cleared otherwise
- C - cleared
- V - cleared

Example: Suppose location A contains 012503_8.

```
        SWAB  A       ;SWAP BYTES AT LOCATION A
                      ;RESULT IS 041425(8)
```

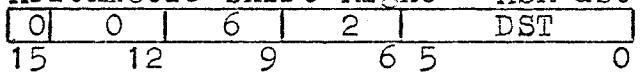
012503_8 = 0001 010 101 000 011_2

(swap) = 0100 001 100 010 $101_2 = 041425_8$

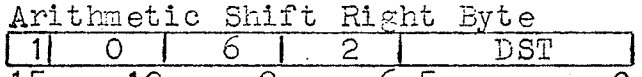
SHIFTS

Shift instructions may be used to multiply or divide any register or memory location by a factor of two.

ASR



ASRB



Description: All bits of the destination are shifted right one place. The sign bit remains unchanged. The C-bit is loaded from bit 0 of the destination.

Condition Codes:
- Z - set if the result is zero, cleared otherwise
- N - set if the high-order bit of the result is 1, cleared otherwise
- C - loaded with the low-order bit of the destination
- V - set as the Exclusive OR of N and C

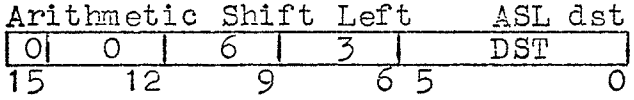
Examples: A right shift is equivalent to division by two with rounding downward.

```
012701      MOV  #15.,R1     ;LOAD R1 WITH 15.
000017                       ; INTEGER DIVISION
106301      ASRB  R1         ; RESULTS IN 7.
```

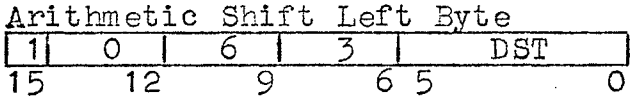
Double precision right shifts may be accomplished by the following:

```
            ASR  A2      ;LOW ORDER OF A2 INTO C-BIT
            ROR  A1      ;C-BIT INTO HIGH ORDER OF A1
```

ASL — Arithmetic Shift Left — ASL dst



ASLB — Arithmetic Shift Left Byte



Description: All bits of the destination are shifted left one place. Bit 0 is loaded with a zero. The C-bit in the processor status register is loaded with the most significant bit of the destination.

Condition Codes:
- Z - set if the result is zero, cleared otherwise
- N - set if the high-order bit of the result is 1, cleared otherwise
- C - loaded with the high-order bit of the destination
- V - set as the Exclusive OR of N and C

Examples: A left shift is equivalent to multiplication by two, but arithmetic overflow may affect the result.

```
012705      MOV  #16710.,R5    ; 16710 DECIMAL IS
040506                         ; ) 040506 OCTAL
006305      ASL  R5            ; MULTIPLY BY 2
```

The result is 101214 octal or -32116 decimal due to arithmetic overflow.

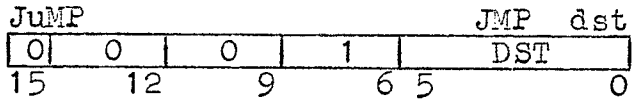
Double precision left shifts are programmed as follows:

```
ASL  A1     ;HIGH ORDER BIT OF A1 INTO C-BIT
ROL  A2     ; C-BIT INTO LOW BIT OF A2
```

JUMP INSTRUCTION

The Jump instruction transfers processor control to any word in memory using any of the PDP-11 addressing modes except register mode. Register mode is illegal because control cannot be sent to a register. Unlike the general BRANCH instructions, JMP may have a variable-length format.

JMP



Description: Control is transferred to the destination address. Since all instructions must be aligned on a word boundary, the destination address must specify an even-byte location. A 'boundary error' results when the processor attempts to fetch an instruction from an odd address.

Condition Codes: not affected

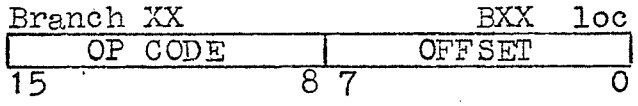
Example: Using the deferred index mode, control may be sent to a location chosen from a table of addresses.

```
JMP   @TABLE(R0)
```

Here the register R0 is used as an index register into the array TABLE whose entries must be legal (even) program addresses.

BRANCH INSTRUCTIONS

Branch instructions are one word in length with the following machine format:



where BXX is a branch instruction mnemonic

loc is a symbolic branch address located up to 127 words before or 128 words after the branch instruction

offset is an 8-bit signed displacement of the branch address relative to the PC

An instruction word is always fetched by the processor from the memory address contained in the PC. Whenever a word is fetched, the PC is automatically incremented by two to point to the next available word. Branch instructions can provide a change in this normal sequential operation of the processor by loading the branch address into the PC.

The offset is calculated automatically by the Assembler as a signed two's complement displacement to be multiplied by two and added to the PC. But the PC points to the word following the branch instruction, consequently

$$\text{offset} = (E-PC)/2 = (E-.-2)/2$$

where E is the actual branch address.

The branch address must be within a limited range and does not use any of the PDP-11 addressing modes. Under this restriction, all branch addresses are calculated easily and efficiently at execution time in the following way:

1. The sign of the offset is extended through bits 8 to 15 to form a full word value.

2. This value is multiplied by two to yield the number of bytes in the displacement.

3. This result is added to the PC to form the ultimate branch address.

Branch instructions are classified as follows:

1. Unconditional branch: BR BRanch

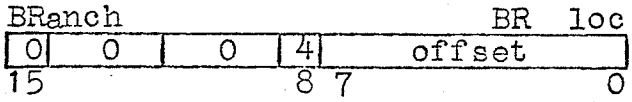
2. Conditional branches:

Simple:	BEQ	Branch on EQual
	BNE	Branch on Not Equal
	BPL	Branch on PLus
	BMI	Branch on MInus
	BCS	Branch on Carry Set
	BCC	Branch on Carry Clear
	BVS	Branch on oVerflow Set
	BVC	Branch on oVerflow Clear
Signed:	BLT	Branch on Less Than
	BGE	Branch on Greater or Equal
	BLE	Branch on Less or Equal
	BGT	Branch on Greater Than
Unsigned:	BHI	Branch on HIgher
	BLOS	Branch on LOwer or Same
	BHIS	Branch on HIgher or Same
	BLO	Branch on LOwer

UNCONDITIONAL BRANCH

The unconditional branch loads the branch address into the PC as described. There is no effect on the condition codes. Control is sent to the branch address.

BR



CONDITIONAL BRANCHES

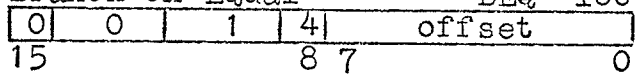
Conditional branches are used for decision-making. Whether a branch is successful or unsuccessful depends on the result of operations preceding the branch instruction as reflected by the condition codes. In either case, the condition codes are inspected but remain unchanged.

The following instructions are grouped in pairs according to which condition code bits will initiate the branch. These instructions are mutually exclusive in that if one branch is successful, the other branch must be unsuccessful. In each case, the mnemonic is self-explanatory.

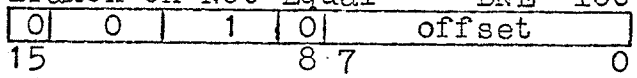
SIMPLE CONDITIONALS

With each condition code bit are associated two simple conditional branches as follows:

BEQ — Branch on EQual — BEQ loc



BNE — Branch on Not Equal — BNE loc



Description: The value of the Z-bit in the processor status register determines whether the branch is taken. If Z is 1, BEQ is successful; if Z is 0, BNE is successful.

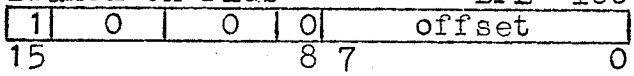
Examples: To test for equality after a comparison:

```
CMP   X,Y       ; COMPARE X AND Y
BEQ   SAME      ; BRANCH IF THEY ARE EQUAL
```

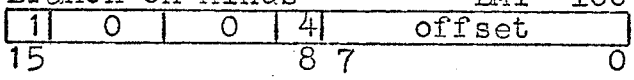
Branches may be set up to control program looping:

```
            ;ZERO OUT AN ARRAY OF 50 ELEMENTS
ARRAY:      .= .+100.       ;RESERVE SPACE FOR 50 WORDS
            MOV  #-50.,R1   ;INITIALIZE A COUNTER
            MOV  #ARRAY,R2  ;GET ARRAY ADDRESS IN R2
LOOP:       CLR  (R2)+      ;ZERO AN ARRAY ELEMENT
            INC  R1         ;DONE?
            BNE  LOOP       ;NO, CONTINUE IF NOT ZERO
```

BPL — Branch on PLus — BPL loc



BMI — Branch on MInus — BMI loc



Description: The value of the N-bit determines whether the branch is taken. If N is 0, BPL is successful; if N is 1, BMI is successful.

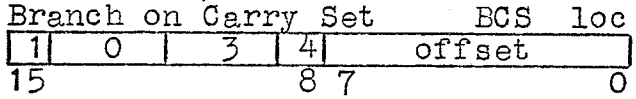
Examples: To test the sign of an arithmetic result:

```
            SUB   A,B      ;SUBTRACT A FROM B
            BMI   NEG      ;BRANCH IF NEGATIVE
```

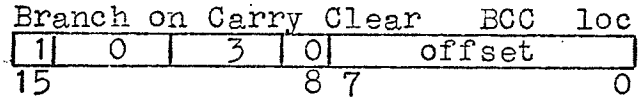
To control iterations:

```
            MOV  #20.,NCOUNT     ;SET THE ITERATION COUNTER
LOOP:
              .  .  .  .

            DEC  NCOUNT          ;DECREASE THE COUNTER
            BPL  LOOP            ;REPEAT IF POSITIVE
```

BCS — Branch on Carry Set — BCS loc



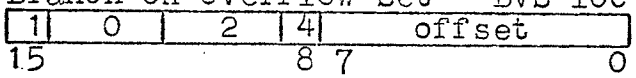
BCC — Branch on Carry Clear — BCC loc



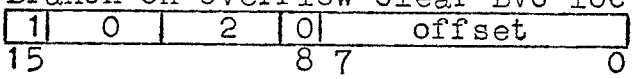
Description: The value of the C-bit determines whether the

branch is taken. If C is 1, BCS is successful; if C is 0, BCC is successful.

BVS — Branch on oVerflow Set BVS loc



BVC — Branch on oVerflow Clear BVC loc



Description: The value of the V-bit determines whether the branch is taken. If V=1, BVS is successful; if V=0, BVC is successful.

Example: To normalize an integer with 1 as its most significant bit:

```
NORM:   ASL  X       ;SHIFT LEFT INSERTING A LOW-ORDER 0
        BEQ  ZERO    ;STOP IF RESULT IS ZERO
        BVC  NORM    ;CONTINUE IF NO SIGN CHANGE
        ROR  X       ;RESTORE THE SIGN
        . . . .
ZERO:
```

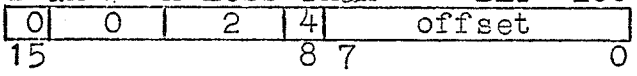
SIGNED (ARITHMETIC) CONDITIONALS

Particular combinations of the condition code may be inspected by signed conditional branches. The results of operations are tested where the value is treated as a signed two's complementary integer. The hierarchy of values for signed integers is as follows:

positive	077777
	077776
	....
	....
	000001
zero	000000
	177777
	....
	....
	100001
negative	100000

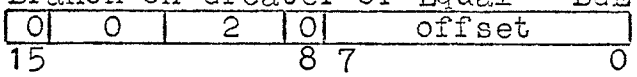
Branch on Less Than BLT loc

BLT



Branch on Greater or Equal BGE loc

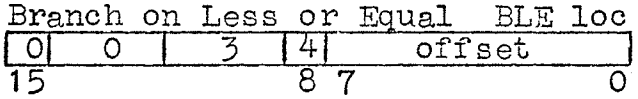
BGE



Description: The value of N 'eXclusive OR' V determines which branch is taken. If N 'XOR' V is 1, BLT is successful; if N 'XOR' V is 0, BGE is successful.

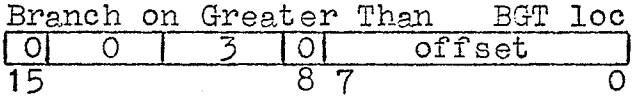
Branch on Less or Equal BLE loc

BLE



Branch on Greater Than BGT loc

BGT



Description: The value of Z OR (N 'eXclusive OR' V) determines which branch is taken. If Z OR (N 'XOR' V) is 1, BLE is successful; if Z OR (N 'XOR' V) is 0, BGT is successful.

Example: For checking the result of a comparison:

```
        CMP   A,B       ;COMPARE A AND B
        BGT   HIGH      ;BRANCH IF A IS GREATER THAN B
```

UNSIGNED (LOGICAL) CONDITIONALS

Results treated as unsigned logical values may be tested using unsigned conditional branches. The hierarchy of logical values is:

highest	177777
	177776
	....
	....
	000002
	000001
lowest	000000

```
                Branch on HIgher                    BHI  loc
BHI      | 1 | 0 | 1 | 0 |          offset              |
         15                  8 7                       0

                Branch on LOwer or Same             BLOS loc
BLOS     | 1 | 0 | 1 | 4 |          offset              |
         15                  8 7                       0
```

Description: The carry bit and the zero bit determine whether the branch is taken. If C and Z are both 0, BHI is successful; if either C or Z is 1, BLOS is successful.

```
                Branch on HIgher or Same            BHIS loc
BHIS     | 1 | 0 | 3 | 0 |          offset              |
         15                  8 7                       0

                Branch on LOw                       BLO  loc
BLO      | 1 | 0 | 3 | 4 |          offset              |
         15                  8 7                       0
```

Description: The value of the C-bit determines the branch. BHIS is equivalent to BCC; BLO is equivalent to BCS.

Example: For sorting of character data:

```
        CMPB  @(R1)+,R2     ;COMPARE TWO CHARACTERS
        BHIS  NOTLOW        ;BRANCH IF TEST BYTE NOT LOWER
```

OPERATE INSTRUCTIONS

Operate instructions perform specific functions for the PDP-11 hardware. They do not require any operands.

```
                OPeration
OP       |            OP-CODE                          |
         15                                            0
```

Instructions of this class include the following:

1. Condition Code operators:

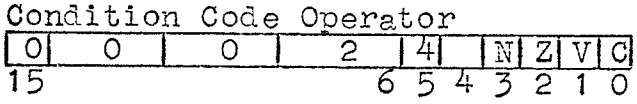
CCC	Clear Condition Codes
CLC	CLear Carry bit
CLN	CLear Negative bit
CLV	CLear oVerflow bit
CLZ	CLear Zero bit
CNZ	Clear Negative and Zero bits
NOP	No OPeration
SCC	Set Condition Codes
SEC	SEt Carry bit
SEN	SEt Negative bit
SEV	SEt oVerflow bit
SEZ	SEt Zero bit

2. Control operators:

RTI	ReTurn from Interrupt
HALT	HALT
WAIT	WAit for InTerrupt
RESET	RESET
IOT	Input/Output Trap

CONDITION CODE OPERATORS

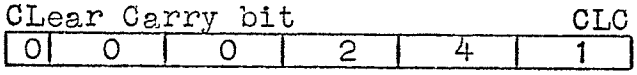
Condition code operators are used to set or clear various bits in the condition code. All instructions have the following format:



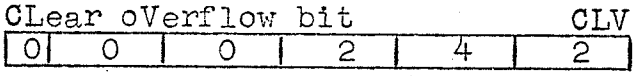
where bits 0-3 of the condition code are set or cleared according to the set/clear bit -- bit 4 -- of the instruction.

The following instructions clear the condition code bits as specified by the mnemonic:

CLC



CLV



CLZ — CLear Zero bit — CLZ

0	0	0	2	4	4

CLN — CLear Negative bit — CLN

0	0	0	2	5	0

CNZ — Clear Negative and Zero bits — CNZ

0	0	0	2	5	4

CCC — Clear Condition Codes — CCC

0	0	0	2	5	7

The following instructions set the condition code bits:

SEC — SEt Carry bit — SEC

0	0	0	2	6	1

SEV — SEt oVerflow bit — SEV

0	0	0	2	6	2

SEZ — SEt Zero bit — SEZ

0	0	0	2	6	4

SEN — SEt Negative bit — SEN

0	0	0	2	7	0

SCC — Set Condition Codes — SCC

0	0	0	2	7	7

If none of the bits 0-3 in the instruction are set, no operation will result.

NOP — No OPeration — NOP

0	0	0	2	4	0

Although mnemonics do not exist, new instructions may be created at execution time to affect different combinations of bits. Suppose the following coding is assembled in memory at address 630_8:

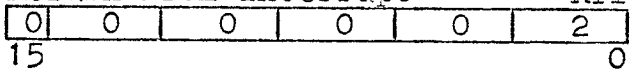
```
630   152767          BISB  #31,CCODE+1    ;MODIFY THE NOP
632   000031                               ; INSTRUCTION
634   000051

                      .  .  .  .  .

706   000240  CCODE:  NOP                  ;NOP TO BE MODIFIED TO
                                           ;FORM A NEW INSTRUCTION
```

At execution time:

```
630:   152767
632:   000031
634:   000051
        ....

706:   000271      ;VIZ SET NEGATIVE AND CARRY
                   ;       BITS
```

CONTROL OPERATORS

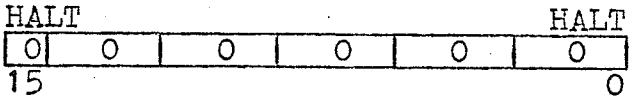
RTI — ReTurn from Interrupt — RTI



Description: The PC and PS are popped from the processor stack, and the SP is adjusted accordingly. RTI is used to exit from an interrupt or a user service routine using the stack mechanism as described in Section A.

Condition Codes: loaded from the processor stack

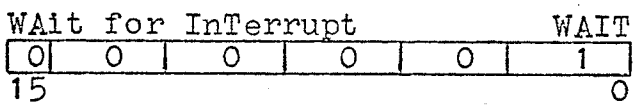
HALT — HALT — HALT



Description: All processing stops. The PC contains the address of the next instruction to be executed. If the HALT instruction is encountered, a user will receive an octal dump of his program, including information about the machine status when the HALT was detected. (See the .EXIT command under Monitor Requests.)

The following instructions are not interpreted by this simulation since they involve recovering control of the communications 'bus' from external devices. They are presented here for the sake of completeness.

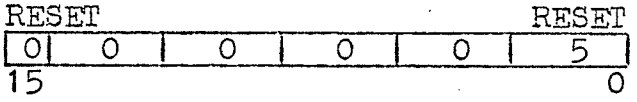
WAIT



Description: The processor goes into a 'wait' state, that is, it stops fetching instructions from memory. The PC points to the instruction following the WAIT. When an external device interrupt occurs, the PC and PS are pushed onto the processor stack. The ensuing RTI instruction will end the 'wait' and resume processing at the next instruction.

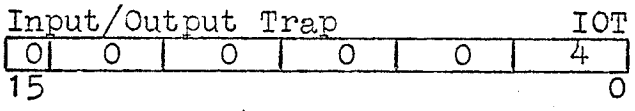
Condition Codes: not affected

RESET



Description: All external devices are reset by sending a clearing signal through the 'bus'. Condition codes are not affected.

IOT



Description: The PC and PS are pushed onto the processor stack. An input/output routine is given control using the interrupt vector at location 20_8 (cf Trap Instructions.) A system-defined

input/output package would provide real-time interaction with external devices, but is not included in this simulation. (See Monitor Requests.)

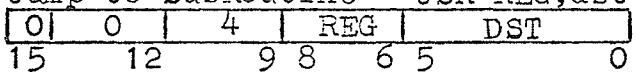
SUBROUTINES

A subroutine is a sequence of instructions designed to perform some specific task. These instructions are assembled and stored in memory only once, but may be executed any number of times by using the JSR (Jump to SubRoutine) and RTS (ReTurn from Subroutine) instructions. For example, to invoke a subroutine named SUBR, the following coding might appear:

```
        JSR  R5,SUBR      ;LINK TO THE SUBROUTINE
         . . . . .
SUBR:    . . . . .        ;SUBROUTINE ENTRY POINT
        RTS  R5           ;RETURN TO INSTRUCTION FOLLOWING
                          ; JSR
```

Subroutine handling in the PDP-11 uses the stack mechanism to dynamically allocate storage for linkage registers. Linkage registers are automatically saved and restored. Consequently, subroutines may be nested (viz. invoke other subroutines), recursive (viz invoke themselves), or have multiple entry points even if using the same linkage register without special programming considerations.

JSR — Jump to SubRoutine — JSR REG,dst



where REG is the linkage register

Description: The PC already contains the return address, namely the address of the word following the JSR instruction. The linkage register is pushed onto the processor stack and is replaced by the PC. Now the linkage register contains the return address. Then the PC is loaded with the destination address, thereby sending control to that location.

Condition Codes: not affected

Examples: A subroutine call may transfer arguments through the general registers. For example:

```
        MOV  X,R1       ;ARGUMENT IN REGISTER 1
        JSR  R4,SIN     ;LINK TO SIN SUBROUTINE
```

Care must be taken that the return address is not lost by destroying the linkage register.

Typically, arguments are passed to subroutines as word data located immediately following the JSR instruction. (The .WORD directive defines a word of memory equal to the value of its expression.) The subroutine may access these arguments by autoincrement or indexed addressing using the linkage register. These addressing modes may be deferred if the arguments are addresses rather than the operands themselves. For example:

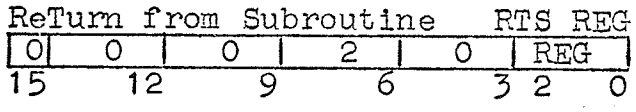
```
        JSR  R5,SORT      ;CALLING SEQUENCE FOR SORT
        .WORD  ARRAY      ;ADDRESS OF ARRAY TO BE SORTED
        .WORD  SIZE       ;SIZE OF THE ARRAY
         . . . . .

SORT:   MOV  @(R5)+,R1    ;GET ARRAY ADDRESS
        MOV  (R5)+,R2     ;GET ITS SIZE
         . . . . .

        RTS  R5           ;RETURN
```

Two routines may swap program control and then resume

operation where they left off. Such routines are called 'co-routines'. The PC is exchanged with the top element of the stack:

```
        JSR  PC,(SP)        ;LINK TO CO-ROUTINE
```

RTS



Description: The contents of the linkage register (the return address) are loaded into the PC. The top element is popped off the processor stack into the linkage register, thus restoring it to its original value.

Condition Codes: not affected

Example: A subroutine may need to save all the registers on the stack, then do its processing, and then restore the registers before returning.

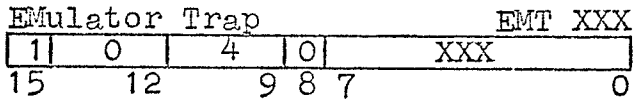
```
        JSR  R5,SUBR        ;CALLING SEQUENCE

SUBR:   MOV  R4,-(SP)       ;R5 PUSHED BY THE JSR
        MOV  R3,-(SP)       ;R5 AT THE BOTTOM FOLLOWED
        MOV  R2,-(SP)       ;BY R4,R3,R2,R1, AND R0 IS
        MOV  R1,-(SP)       ;AT THE TOP
        MOV  R0,-(SP)
         . . . . .
         . . . . .          ;PROCESSING FOR SUBROUTINE
         . . . . .

        MOV  (SP)+,R0       ;RESTORE REGISTERS IN REVERSE
        MOV  (SP)+,R1       ;ORDER TO HOW THEY WERE SAVED
        MOV  (SP)+.R2
        MOV  (SP)+,R3
        MOV  (SP)+,R4       ;R5 IS RESTORED BY THE RTS
        RTS  R5             ;RETURN
```

TRAP INSTRUCTIONS

Trap instructions are programmed interrupts used as subroutine calls to user or installation defined routines. As in all interrupts, the current PC and PS are pushed onto the stack, and the new PC and PS are loaded from an appropriate interrupt (trap) vector. In addition, the low order byte of the instruction may be used to transmit information to the trap-handling routine.

With the expansion of this PDP-11 simulation, these trap instructions might become requests to an operating system for some user services such as input or output, debugging aids, or system library functions.[6] (See Monitor Requests.)

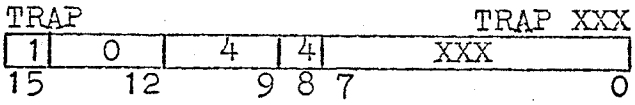
EMT



Description: An interrupt occurs using the trap vector at location 30_8. The low-order byte of the EMT instruction, bits 0-7, contains information for the emulating routine -- a total of 256 different codes, 0 to 255. The new PC is taken from the word at location 30_8, the new PS from location 32_8.

Condition Codes: loaded from the trap vector

TRAP



Description: An interrupt occurs using the trap vector at location 34_8. Otherwise, TRAP and EMT are identical.

Condition Codes: loaded from the trap vector

MONITOR REQUESTS

In order to implement a batch-processing PDP-11 facility in the absence of a full-scale operating system monitor, certain management and user services had to be provided. This was accomplished within the PDP-11 hardware environment by imbedding these services into the interrupt system as extentions to PAL-11R Assembly Language. Extended mnemonics were developed for the Assembler so that these monitor requests could be assembled into a user program as special trap instructions. At execution time, the processor stack and an interrupt vector are invoked exactly as for any regular trap instruction. In fact, a programmer may code the monitor request by its PAL-11R equivalent instead of the extended mnemonic and obtain the same results.

The basic support services available as monitor requests are as follows:

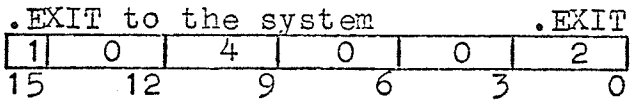
.EXIT	EXIT to the system
.DUMP	memory DUMP

Input/Output Macros:	PRINTC	PRINT Character
	PRINTO	PRINT Octal
	READC	READ Character
	READO	READ Octal

MUL	MULtiply
DIV	DIVide

.EXIT

.EXIT to the system .EXIT

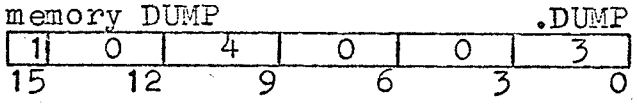


Equivalent: EMT 2

Description: .EXIT must be the last executable statement in a

user program. It terminates processing of that job and returns control to the 'system'. This enables all parameters to be re-initialized for the next job in the batch. The HALT instruction, on the other hand, is interpreted as an illegal instruction, and will bring about a memory dump in order to return to the system.

.DUMP



Equivalent: EMT 3

Description: .DUMP prints on the output listing an absolute copy in octal notation of all core allocated to that program. This .DUMP will terminate all processing of that job, and exit to the system.

INPUT/OUTPUT MACROS

A macro instruction is a source statement. The Assembler generates a sequence of PAL-11R assembler language statements for each occurrence of a macro. These generated statements are then processed like any other assembler language statement. Each time a given macro appears, it is replaced by the same sequence of instructions.

The use of these macros simplifies the coding of programs by standardizing all requests for input and output. However, a programmer may code the macro expansion statements by himself, instead of calling the macro by its mnemonic name.

The format for these I/O macros is the same:

```
MACRO    SOURCE,LENGTH
```

where SOURCE is the address of the data to be printed or read;

LENGTH is the number of words (octal mode) or bytes (character mode) to be printed from or read into that source.

There are two modes of data transfer: octal and character. In octal mode, a word source is required. Each word is treated as a 6-digit octal number. In character mode, any byte address may be specified. Each byte is treated as a binary 8-bit ASCII character as tabulated in Appendix A.

Macros differ only in their EMT codes. The macro expansion is as follows:

```
                MACRO   SOURCE,LENGTH

+               MOV     #LENGTH,-(SP)     ;PUSH LENGTH ON STACK
+               MOV     #SOURCE,-(SP)     ;PUSH ADDRESS ON STACK
+               EMT     N                 ;EMT CALL FOR I/O
```

where the plus sign (+) in column 1 of the output listing indicates a macro expansion. Note that the macro requires 5 words (10 bytes) of memory. None of the various addressing modes may be used for either argument. Programming errors in monitor request macros are discovered after the macro expansions have been created.

PRINTC PRINT Character

Expansion:

```
                PRINTC  SOURCE,LENGTH
+               MOV     #LENGTH,-(SP)
+               MOV     #SOURCE,-(SP)
+               EMT     4
```

Description: The number of characters specified are printed onto the output listing. Up to 80 characters may be printed on any one line, (viz., $1 \leq LENGTH \leq 80$.) If no ASCII code exists for the data, blanks are inserted.

PRINTO PRINT Octal

```
Expansion:          PRINTO   SOURCE,LENGTH
          +         MOV   #LENGTH,-(SP)
          +         MOV   #SOURCE,-(SP)
          +         EMT   0
```

Description: The number of words specified are printed on the output listing as 6-digit octal numbers, with up to eight words per line, (viz., $1 \leq LENGTH \leq 8$.)

READC READ Character

```
Expansion:          READC   SOURCE,LENGTH
          +         MOV   #LENGTH,-(SP)
          +         MOV   #SOURCE,-(SP)
          +         EMT   5
```

Description: The number of characters specified are read from a data card and stored into successive bytes at the source address. All 80 columns of a data card are read -- one character per column -- until the required number of characters are read. The parameter LENGTH must be between 1 and 80.

READO READ Octal

```
Expansion:          READO   SOURCE,LENGTH
          +         MOV   #LENGTH,-(SP)
          +         MOV   #SOURCE,-(SP)
          +         EMT   1
```

Description: The number of words specified are read from a data card and stored into successive words starting at the source address. Each word is 6 octal digits long and is taken from the data card as a 6-column field. Up to eight words may appear contiguously on any one data card.

	WORD1	WORD2	WORD3	WORD4	WORD5	WORD6	WORD7	WORD8
column:	1	7	13	19	25	31	37	43

MUL AND DIV

The PDP-11 has no hardware instructions for multiplication or division. These operations may be carried out by a series of shifts, additions and rotates. The monitor requests MUL (MULtiply) and DIV (DIVide) were included to free the user from this restriction by providing system-defined routines available as TRAP instructions.

Arguments are passed to the multiplication and division routines in the following way:

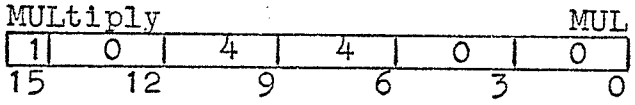
1. Register 0 must contain the address of the first operand, (the multiplicand or the dividend.)

2. Register 1 must contain the address of the second operand, (the multiplier or the divisor.)

3. The results are stored as two words at the first operand location.

The programmer must ensure that the proper addresses have been placed into the registers. No operands may be specified in the actual MUL or DIV request.

MUL



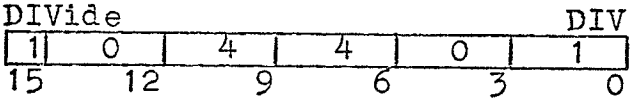
Equivalent: TRAP 0

Description: The address of the multiplicand is in Register 0. The address of the multiplier is in Register 1. A double-word product is calculated and stored at the multiplicand address, the high-order word first, the low-order second. The original values of the multiplicand and the following word are lost.

Example:

```
MOV  #MCAND,R0      ;MULTIPLICAND ADDRESS IN R0
MOV  #MPLIER,R1     ;MULTIPLIER ADDRESS IN R1
MUL                 ;TRAP TO MULTIPLY ROUTINE
```

DIV



Equivalent: TRAP 1

Description: The address of the dividend is in Register 0. The address of the divisor is in Register 1. After the division, a word remainder and a word quotient are stored in successive words at the dividend address; the original values of the dividend and the following word are lost.

Example:

```
MOV  #DIVEND,R0     ;DIVIDEND ADDRESS IN R0
MOV  #DIVOR,R1      ;DIVISOR ADDRESS IN R1
DIV                 ;TRAP TO DIVIDE ROUTINE
```

ASSEMBLER DIRECTIVES

Assembler directives are requests to the Assembler to perform certain operations at assembly time.[5] These directives may generate data, cause storage areas to be reserved for working space, or alter the location counter. Assembler directives form the operator field of a statement; hence only one directive may appear in any one statement. The directive may be preceded by a label or followed by a comment. The number of operands (if any) varies from directive to directive.

The assembler directives, summarized in Appendix F, are presented as follows:

.END END of program

Data generating directives:

.WORD	WORD generator
.BYTE	BYTE generator
.RAD50	RADix 50
.ASCII	ASCII characters
.LIMIT	program core LIMITs
.EVEN	EVEN the location counter
.TITLE	module TITLE

Program sectioning directives:

.ASECT	Absolute SECTion
.CSECT	reloCatable SECTion
.GLOBL	GLOBaL symbol

Conditional assembly directives:

.IFDF	IF DeFined
.IFNDF	IF Not DeFined
.IFZ	IF Zero
.IFNZ	IF Not Zero
.IFL	IF Less than zero
.IFLE	IF Less or Equal to zero
.IFG	IF Greater than zero
.IFGE	IF Greater or Equal to zero
.ENDC	END of Conditional

.END .END E

Description: The .END directive indicates the physical end of a source program. It must be the last statement of any job. The .END directive is followed by one operand which specifies the starting address for execution of that program. This address is passed to the Interpreter at execution time.

DATA GENERATING DIRECTIVES

.WORD .WORD E1,E2,E3, ...

Description: The .WORD directive generates successive words of data equal to the values of its operands. There may be one or more operands separated by commas. Each operand generates one data word. An operand may be any legal expression. Values exceeding 16-bits are flagged and truncated from the left to word quantities. In addition, in any statement where no legal instruction is specified, or there is a leading arithmetic or logical operator, the .WORD directive is assumed by default.

Examples:

```
        Suppose .=2160 (octal) and X=15. (0017 octal)
2160:   104567    LABEL:    .WORD  104567,X,LABEL+10
2162:   000017
2164:   002170

        Missing operands are stored as zeros:
000000          .WORD  ,25,
000025
000000

        Instruction mnemonics may appear as word data:
100000          .WORD  MOV,INC      ;MOV=100000 OCTAL
005200                              ;INC=005200 OCTAL

        .WORD is default for missing instructions:
000000      VALUE:     0,5,10
000005
000010
```

.BYTE .BYTE E1,E2,E3, ...

Description: The .BYTE directive generates bytes of data equal to the values of its operands. Each operand generates one byte. Values exceeding 8-bits are flagged and truncated to a byte capacity.

Example:

```
                    VAL= 32.            ;040 OCTAL
000                 .BYTE  0,VAL, ,1
040
000
001
```

.RAD50 .RAD50 /CCC/

Description: The RAD50 directive generates the RADix 50 representation of up to three ASCII characters within the delimiters. The general form of the directive is:

.RAD50 /CCC/

where the slash (or any ASCII character except = or :) is the delimiter, and CCC represents the characters to be converted, chosen among A to Z, 0 to 9, $, . and blank. Radix 50 notation enables three characters to be packed into one 16-bit word (called a triad) as follows:

1. Each character is translated into a radix 50 code as tabulated below:

Character	Radix 50 Code (octal)
blank	0
A-Z	1-32
$	33
.	34
0-9	36-47

2. If there are fewer than 3 characters, they are considered to be left-justified and trailing blanks are inserted. Characters

beyond the third place are ignored.

3. The radix 50 triad for C1,C2,C3 is formed from the above codes as follows:

TRIAD = ((C1 * 50) - C2) * 50 - C3)

Example: All symbol names used by the Assembler are stored as two packed triads in radix 50 notation.

```
.RAD50  ?ADC?       ;GENERATES 003343 OCTAL
                    ; ? IS THE DELIMITER
```

.ASCII

Description: The .ASCII directive generates strings of ASCII characters as tabulated in Appendix A. Each character fills one byte of memory. The general form is as follows:

.ASCII / ..... text ..... /

where the text is any string of characters, and the delimiter may be any ASCII character (except = or :) that is not used in the text.

Example:

```
040         .ASCII   / WHAT?/
127
110
101
124
077
```

.TITLE .TITLE symbol

Description: The .TITLE directive is used to name the object module. Otherwise, by default, the name '.MAIN' is used.

The following directives do not require arguments, and any present are ignored.

.LIMIT memory LIMITs

Description: The .LIMIT directive generates two words of data indicating the absolute memory locations of the machine code as relocated for execution. The first word is the address of the first byte of code; the second is the address of the byte after the last byte of code. These addresses are always even since all programs are allocated core in word quantities.

.EVEN

Description: The .EVEN directive ensures that the assembly location counter is even by adding 1 if it is odd.

PROGRAM SECTIONING DIRECTIVES

The Assembler allows eight program sections: an absolute section declared by .ASECT, an unnamed relocatable section declared by .CSECT, and six named relocatable sections declared by .CSECT SYMBOL where SYMBOL is any legal symbol name.

The Assembler maintains separate location counters for each section. Consequently, sections may be interrupted and later resume where they left off, so that instructions not coded contiguously may still be assembled into contiguous memory locations. Any labels appearing on the .ASECT or .CSECT directive are assigned the value of the location counter before that directive takes effect.

.ASECT Absolute SECTion

Description: .ASECT declares the beginning or the resumption

of an absolute section. The first appearance of an .ASECT assumes the location counter is absolute zero. Subsequent appearances load the location counter with the address of the next available location for that section. All labels in an absolute section are absolute. The .ASECT directive remains in effect until another program sectioning directive is issued. Absolute sections are loaded into core at the locations specified by the programmer. For example:

```
.ASECT              ;START AN ABSOLUTE SECTION
.= 500              ; AT MEMORY ADDRESS 500₈
```

```
.CSECT              reloCatable SECTions
.CSECT SYMBOL
```

Description: .CSECT identifies the beginning or the resumption of a relocatable section. If a symbol names the .CSECT directive, that symbol is established as the name of the section; otherwise the section is considered to be unnamed. The Assembler automatically begins assembly with an unnamed relocatable section unless instructed otherwise. The first appearance of the .CSECT directive assumes the location counter is relocatable zero. All statements following the .CSECT are assembled as part of that section until a directive to the contrary is encountered. Further .CSECTs resume assembly where that section left off. All labels in a relocatable section are relocatable. For execution, all relocatable sections are loaded contiguously into the highest available core locations. This leaves maximum unused core available for the processor stack.

Each section is assembled independently into contiguous areas of core, and symbols defined in one section have no relationship to symbols defined in another unless specifically equated by a .GLOBL directive.

.GLOBL .GLOBL SYM1,SYM2,SYM3, ...

Description: The .GLOBL directive allows various program sections to communicate with each other by declaring symbols as global. A global symbol may be an entry point -- that is, defined in the current program section -- or an external symbol -- that is, defined in another program section. A symbol is not global unless it appears in a .GLOBL directive. Thus, a symbol does not become global by appearing in a direct assignment statement with a global symbol.

CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly directives allow the programmer to include or delete a sequence of instructions from the assembly process depending upon certain conditions.

Directive	Condition
.IFZ E	if E equals zero
.IFNZ E	if E is not zero
.IFL E	if E is less than zero
.IFLE E	if E is less or equal to zero
.IFG E	if E is greater than zero
.IFGE E	if E is greater or equal to zero

Directive	Condition
.IFDF EL	if EL is defined
.IFNDF EL	if EL is not defined

where E represents any expression

EL is a logical expression of symbolic names and logical operators & or !

Expressions are evaluated from left to right. If the condition is met, all statements up to the matching .ENDC (END of Conditional) are assembled. Otherwise, these statements are ignored. Conditionals may be nested to a depth of 127. Syntax errors (flagged 'Q') will result from missing or extra .ENDC directives. Labels may appear on conditional statements, but are ignored if the condition is not met.

Example:

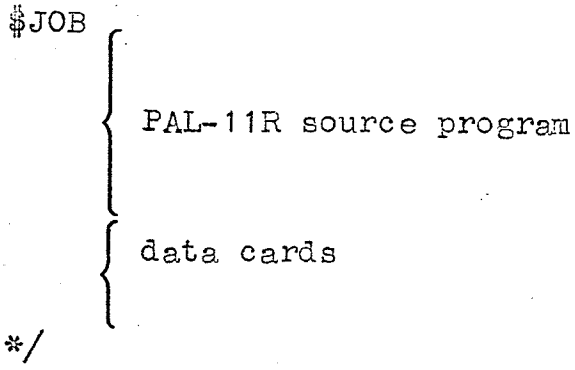
```
        .IFDF   X!Y&Z       ;ASSEMBLE IF (EITHER X OR Y IS
                            ;DEFINED) AND (Z IS DEFINED)
```

SECTION E OPERATING PROCEDURE

The following sections explain how to submit a program to be executed by the PDP-11 simulator.

CONTROL CARDS

Control cards are used to delimit a job as shown. Each control card must begin in column 1.



ASSEMBLER OPTIONS

If no options are coded, a job will be assembled, loaded, and executed using 2K words of memory, and allowing 5 seconds of execution time.

These default options may be overridden by specifying the following parameters on the $JOB card:

1. Execution Option: (EXEC=0)

Execution of a job can be omitted by specifying EXEC=0. In this case, the job is assembled and an assembly listing is produced. Otherwise, execution will be assumed.

2. Time Option: (TIM=nn)

An execution time of up to 35 seconds may be requested. If a user program has not completed within this time, a .DUMP is initiated. Default is 5 seconds.

3. Maximum Core Option: (MAX=nn)

The PDP-11 may address up to 32K words of memory. The user may obtain any size of core from 2K to 32K for the execution of his program. Default is 2K words.

Job options must be placed on the $JOB card starting in column 16. The entire option keyword or only the first letter may be coded. Each keyword must be followed by an equal sign (=) followed by one or two digits. Options may be specified in any order, but must be separated by commas, with no blanks allowed; for example:

```
$JOB            MAX=16,TIM=10,EXEC=1
Assemble, load and execute using 16K words
of memory and 10 seconds execution time.
```

Comments or user identification may be included on the $JOB card by leaving one blank after the option field.

STACK ADDRESSABILITY

In order to use the processor stack -- and therefore any trap or I/O instructions -- a programmer must first initialize the stack pointer, register 6. Since a program is loaded into the highest memory locations, all core above the physical beginning of the machine code is unused. Thus the stack pointer may

be set by writing the first instructions as follows:

```
            SP=%6                   ;DEFINE SP AS REGISTER 6
START:      MOV  #START,SP          ;SET THE STACK POINTER
              .  .  .  .  .
              .  .  .  .  .
              .  .  .  .  .

            .END  START
```

THE PROGRAM LISTING

A program listing contains the following information:

1. ASSEMBLY PARAMETERS

The first page of output serves to separate the various jobs in the batch. In addition, it includes a copy of the $JOB card, a summary of the assembler options in effect, and the absolute memory address of the start of the machine code. For example:

```
********** PDP-11 ASSEMBLER OPTIONS **********
            ASSEMBLE,LOAD,EXECUTE
            MAXIMUM EXECUTION TIME 5 SECONDS
            MAXIMUM CORE  2K WORDS
            ORIGIN AT 007746
```

2. SOURCE PROGRAM

A side-by-side copy of the source statements and the machine code equivalent is printed out with 55 lines per page, and sub-titles explaining the various entries. (See the Sample Programs.) The listing appears in the following format:

```
E  LLLLLL  000000A  NNNN  SSSSSSSSSS...          ...S
           000000
           000000
```

(a) E represents an error flag as tabulated in Appendix G.

(b) The L-field represents a 6-digit octal address of the memory location in which the machine code is assumed to exist. In most cases, this is an 'apparent' address since all relocatable sections are loaded into the highest available memory locations. The Assembler parameter ORIGIN specifies the absolute location of 'relocatable zero'.

(c) The O-field represents the machine code in octal notation. Any index words (a maximum of two) generated by the Assembler are listed directly under the machine instruction. No address locations precede these index words since the address order is sequential. For a direct assignment statement, the value of the defining expression is printed in the object code field although it is not actually part of the generated machine instructions. For a .BYTE directive and an .ASCII directive, the object code field is 3 octal digits.

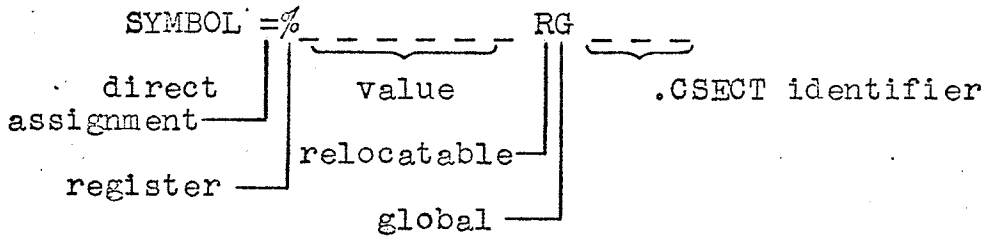
(d) The 'A' is a relocation flag -- represented on the listing by an apostrophe (') -- which indicates that the second or third word of the machine code is a relocated address constant. Whenever an absolute memory reference is made, the Assembler ensures that the corresponding machine code specifies the relocated absolute address. However, since this value is meaningless to the programmer, the unrelocated symbol table value is printed on the listing.

(e) Each source statement is assigned a statement number by the Assembler. The N-field contains this statement number as a 4-digit decimal integer.

(f) The S-field represents the 80-character source statement.

3. SYMBOL TABLES

The User Symbol Table is printed out in alphabetical order with three symbols per line in the following format:[7]



The identifying characters indicate the attributes of the symbol -- a label, direct assignment, register symbol, relocatable, absolute, or global. If a symbol is undefined, six asterisks replace its value. The .CSECT identifier is left blank for symbols defined in an .ASECT or an unnamed .CSECT since they can be identified by the absence or presence of the 'R' respectively. Symbols belonging to named .CSECT's have the ID's 002 through 007, where the n'th named .CSECT is assigned the ID n+1.

Immediately following the symbol table listing is a summary of the .CSECT names with their corresponding origin, length, and ID numbers.

.CSECT ORIGIN LENGTH ID

4. ERROR DIAGNOSTICS

As noted, an error flag appears on the line where that error occurred. More than one error may be detected in any one source statement, but only one error flag is printed on that statement. However, at the end of the listing is tabu-

lated a complete summary of all the errors in the program. This summary appears in logical order by statement number and includes an explanatory diagnostic message for each error. For example:

```
3 STATEMENTS FLAGGED IN THIS ASSEMBLY

STMT  TYPE  MESSAGE

 17     R   REGISTER-TYPE ERROR -- EXPRESSION NOT IN THE
                                RANGE 0 TO 7
 21     U   UNDEFINED SYMBOL -- ASSIGNED THE VALUE ZERO
 28     Q   QUESTIONABLE SYNTAX -- ILLEGAL SEQUENCE OF
                                OPERATORS OR SCAN INCOMPLETE
```

In addition, certain conditions may terminate the assembly of a user program. If this happens, no program listing is created, and one of the following error messages is printed:

```
***** SYSTEM ERROR *****  SCRATCH AREA BUFFER HAS OVERFLOWED
JOB CANNOT BE PROCESSED

***** SYSTEM ERROR *****  SYMBOL TABLE OVERFLOW
```

If the end-of-program delimiter */ is encountered before an .END directive is read, an .END card is generated and assembly continues. However, if a new $JOB card is encountered, the old job is abandoned with the message:

```
*** MISSING CONTROL CARD *** JOB ABANDONED
```

SAMPLE PROGRAMS

The following are program listings for three sample problems using the PDP-11 simulator:

EXAMPLE 1: Program to compute the factorial of an integer

```
 LOC     CODE     STMT    SOURCE STATEMENT

         000000      1          R0=%0        ;DEFINE REGISTER SYMBOLS
         000001      2          R1=%1
         000006      3          SP=%6
                     4  ;
000000   012706'      5  START: MOV  #START,SP     ;SET STACK POINTER
         000000
                     6          READO  NUMBER,1    ;READ A NUMBER
000004   012746       7  +       MOV  #1,-(6)
         000001
000010   012746'      8  +       MOV  #NUMBER,-(6)
         000074
000014   104005       9  +       EMT  5             ;TRAP FOR I/O REQUEST
000016   005767      10          TST  NUMBER
         000052
000022   003414      11          BLE  OUT           ;EXIT IF NEGATIVE
000024   012700'     12  FACT:   MOV  #RESULT,R0    ;MULTIPLICAND ADDRESS
         000070
000030   012701'     13          MOV  #NUMBER,R1    ;MULTIPLIER ADDRESS
         000074
000034   104400      14          MUL                ;MULTIPLY
000036   102413      15          BVS  OUT1          ;EXIT FOR OVERFLOW
000040   016767      16          MOV  RESULT+2,RESULT
         000026
         000022
000046   005367      17          DEC  NUMBER        ;DECREMENT INTEGER
         000022
000052   001364      18          BNE  FACT          ;CONTINUE IF NON-ZERO
                     19  OUT:    PRINTO  RESULT,1   ;PRINT THE FACTORIAL
000054   012746      20  +       MOV  #1,-(6)
         000001
000060   012746'     21  +       MOV  #RESULT,-(6)
         000070
000064   104000      22  +       EMT  0             ;TRAP FOR I/O REQUEST
000066   104002      23  OUT1:   .EXIT              ;EXIT
000070   000001      24  RESULT: 1,1
         000001
000074   000000      25  NUMBER: 0
000000               26          .END START
```

SYMBOL TABLE

```
FACT     000024R     NUMBER   000074R     OUT      000054R
OUT1     000066R     RESULT   000070R     R0     =%000000
R1     =%000001      SP     =%000006      START    000000R

CSECT       ORIGIN      LENGTH      ID
.MAIN       000000      000076      001

NO STATEMENTS FLAGGED IN THIS ASSEMBLY
```

EXAMPLE 2: Program to generate a histogram

```
LOC     CODE     STMT    SOURCE STATEMENT

                    1  ;FIND THE FREQUENCY OF OCCURENCE OF GIVEN VALUES
                    2  ;       OTABLE:     OUTPUT TABLE
                    3  ;       ITABLE:     INPUT TABLE
                    4  ;
        000000      5           R0=%0
        000001      6           R1=%1
        000002      7           R2=%2
        000004      8           R4=%4
        000006      9           SP=%6
000000  012706'    10  START:   MOV  #START,SP    ;SET THE STACK POINTER
        000000
000004  012700'    11  HIST:    MOV  #OTABLE,R0   ;OUTPUT TABLE ADDRESS
        000062
000010  012701     12           MOV  #-100.,R1    ;100 ENTRIES
        177634
000014  005020     13  CLOOP:   CLR  (R0)+        ;ZERO NEXT ENTRY
000016  005201     14           INC  R1           ;CHECK IF DONE
000020  001375     15           BNE  CLOOP        ;IF NOT, GO BACK
000022  012700'    16           MOV  #ITABLE,R0   ;SET INPUT POINTER
        000372
000026  012701     17           MOV  #-1000.,R1   ;LENGTH OF INPUT
        176030
000032  012702     18           MOV  #100.,R2     ;MAXIMUM INPUT VALUE
        000144
000036  012004     19  HLOOP:   MOV  (R0)+,R4     ;GET AN INPUT VALUE
000040  003405     20           BLE  NOCNT        ;IGNORE ZERO OR LESS
000042  020402     21           CMP  R4,R2        ;COMPARE TO UPPER LIMIT
000044  002403     22           BGT  NOCNT        ;IGNORE IF GREATER
000046  006304     23           ASL  R4           ; 2 BYTES PER ENTRY
000050  005264'    24           INC  OTABLE(R4)   ;INCREMENT PROPER ENTRY
        000062
000054  005201     25  NOCNT:   INC  R1           ;INPUT DONE?
000056  001367     26           BNE  HLOOP        ;NO, REPEAT
000060  104002     27           .EXIT             ;COMPLETE
        000372     28  OTABLE:    .=.+200.        ;RESERVE FOR OUTPUT
        002342     29  ITABLE:    .=.+1000.       ;RESERVE FOR INPUT
000000             30           .END  START
```

SYMBOL TABLE

```
CLOOP    000014R        HIST     000004R        HLOOP    000036R
ITABLE   000372R        NOCNT    000054R        OTABLE   000062R
R0     =%000000         R1     =%000001         R2     =%000002
R4     =%000004         SP     =%000006         START    000000R
.      = 002342R

CSECT       ORIGIN      LENGTH      ID
.MAIN       000000      002342      001
```

EXAMPLE 3: Program to simplify arithmetic expressions

```
LOC     CODE     STMT    SOURCE STATEMENT

                    1   ;PROGRAM TO SIMPLIFY ARITHMETIC EXPRESSIONS
                    2   ;    INPUT:  LEFT JUSTIFIED CHARACTER STRING
                    3   ;            NO BLANKS ALLOWED
                    4   ;            OPERANDS ARE INTEGERS
                    5   ;            OPERATORS INCLUDE +  -  *  /
                    6   ;
        000000      7           R0=%0
        000001      8           R1=%1
        000002      9           R2=%2
        000003     10           R3=%3
        000004     11           R4=%4
        000005     12           R5=%5
        000006     13           SP=%6
000000  012706'     14   BEGIN:  MOV  #BEGIN,SP    ;SET STACK POINTER
        000000
                   15   READ:   READC  INPUT,80.   ;READ A DATA CARD
000004  012746      16   +       MOV  #80.,-(6)
        000120
000010  012746'     17   +       MOV  #INPUT,-(6)
        000060
000014  104001      18   +       EMT  1             ;TRAP FOR I/O REQUEST
                   19           PRINTC  INPUT,80.
000016  012746      20   +       MOV  #80.,-(6)
        000120
000022  012746'     21   +       MOV  #INPUT,-(6)
        000060
000026  104004      22   +       EMT  4
000030  012703'     23           MOV  #INPUT,R3     ;DATA STARTING ADDRESS
        000060
000034  004567      24           JSR  R5,NSCAN      ;GET FIRST INTEGER
        000432
000040  010267      25           MOV  R2,OPND1      ;SAVE AS OPND1
        000204
000044  122713      26   LOOP:   CMPB  #040,@R3     ;BLANK NEXT?
        000040
000050  001453      27           BEQ  WRITE         ;YES, PRINT RESULT
000052  004567      28           JSR  R5,OPSCAN     ;NO, FIND OPERATOR
        000202
000056  000771      29           BR   LOOP          ;CONTINUE
                    30   ;
        000200      31   INPUT:  .=.+80.            ;RESERVE FOR INPUT
                    32   ;
                    33   ;CONVERT RESULT TO ASCII CHARACTERS AND PRINT
                    34   ;
000200  005002      35   WRITE:  CLR  R2            ;LENGTH OF RESULT
000202  005202      36   WRITE1: INC  R2            ;INCREMENT COUNTER
000204  012700'     37           MOV  #OPND1,R0     ;ADDRESS OF RESULT
        000250
```

```
  LOC     CODE    STMT     SOURCE STATEMENT

000210  012701'    38            MOV  #TEN,R1       ;ADDRESS OF DIVISOR
        000256
000214  104401     39            DIV                 ;DIVIDE BY 10
000216  052767     40            BIS  #060,TEMP      ;CONVERT REMAINDER
        000060
        000026
000224  116743     41            MOVB TEMP+1,-(R3) ;SAVE CHARACTER
        000023
000230  005767     42            TST  OPND1          ;ZERO QUOTIENT?
        000014
000234  001362     43            BNE  WRITE1         ;NO, GO BACK
000236  010246     44            MOV  R2,-(SP)       ;LENGTH ON STACK
000240  010346     45            MOV  R3,-(SP)       ;ADDRESS ON STACK
000242  104004     46            EMT  4              ;TRAP FOR PRINTC
000244  000657     47            BR   READ           ;RETURN
000246  000000     48  OPND2:    0                   ;SECOND OPERAND
000250  000000     49  OPND1:    0                   ;FIRST OPERAND
000252  000000     50  TEMP:     0,0                 ;TEMPORARY WORK AREA
000254  000000
000256  000012     51  TEN:      10.                 ;DECIMAL CONSTANT
                   52  ;
                   53  ;     SCAN FOR LEGAL OPERAND AND SIMPLIFY
                   54  ;
000260  012704     55  OPSCAN:   MOV  #3,R4          ;SET OPTAB BYTE INDEX
        000003
000264  121364'    56  SCAN1:   CMPB @R3,OPTAB(R4)  ;IDENTIFY OPERATOR?
        000316
000270  001057     57            BNE  SCAN2          ;NO, CONTINUE
000272  005203     58            INC  R3             ;INCREMENT SCAN INDEX
000274  004567     59            JSR  R5,NSCAN       ;GET SECOND OPERAND
        000172
000300  010267     60            MOV  R2,OPND2
        177742
000304  006304     61            ASL  R4             ;FORM WORD INDEX
000306  004574'    62            JSR  R5,@ADDTAB(R4)  ;GO TO SIMPLIFY
        000322
000312  102433     63            BVS  OVER           ;OVERFLOW?
000314  000205     64            RTS  R5             ;RETURN
                   65  ;
000316     057     66  OPTAB:    .ASCII '/*-+'       ;TABLE OF OPERATORS
000317     052
000320     055
000321     053
000322  000366'    67  ADDTAB:  .WORD  DIVIS,MULT,MINUS,PLUS
000324  000352'
000326  000342'
000330  000332'
                   68  ;
000332  066767     69  PLUS:     ADD  OPND2,OPND1   ;ADD THE OPERANDS
        177710
        177710
```

```
LOC     CODE    STMT    SOURCE STATEMENT

000340  000205    70            RTS   R5              ;RETURN
000342  166767    71    MINUS:  SUB   OPND2,OPND1 ;SUBTRACT THE OPERANDS
        177700
        177700
000350  000205    72            RTS   R5              ;RETURN
000352  012700'   73    MULT:   MOV   #OPND2,R0   ;MULTIPLICAND ADDRESS
        000246
000356  012701'   74            MOV   #OPND1,R1   ;MULTIPLIER ADDRESS
        000250
000362  104400    75            MUL                   ;MULTIPLY
000364  000205    76            RTS   R5
000366  012700'   77    DIVIS:  MOV   #OPND1,R0   ;DIVIDEND ADDRESS
        000250
000372  012701'   78            MOV   #OPND2,R1   ;DIVISOR ADDRESS
        000246
000376  104401    79            DIV                   ;DIVIDE
000400  000205    80            RTS   R5
                  81    ;
                  82    OVER:   PRINTC  ERR3,8.
000402  012746    83    +       MOV   #8.,-(6)
        000010
000406  012746'   84    +       MOV   #ERR3,-(6)
        000420
000412  104004    85    +       EMT   4               ;TRAP FOR I/O REQUEST
000414  000167    86            JMP   READ
        177364
000420     117    87    ERR3:   .ASCII  /OVERFLOW/
000421     126
000422     105
000423     122
000424     106
000425     114
000426     117
000427     127
                  88    ;
000430  105704    89    SCAN2:  TSTB -(R4)            ;DECREMENT AND TEST
000432  002314    90            BGE   SCAN1           ;CONTINUE?
                  91            PRINTC  ERR1,16.
000434  012746    92    +       MOV   #16.,-(6)
        000020
000440  012746'   93    +       MOV   #ERR1,-(6)
        000452
000444  104004    94    +       EMT   4               ;TRAP FOR I/O REQUEST
000446  000167    95            JMP   READ            ;GET NEXT CARD
        177332
000452     111    96    ERR1:   .ASCII  /ILLEGAL OPERATOR/
000453     114
000454     114
000455     105
000456     107
000457     101
```

```
  LOC     CODE    STMT     SOURCE STATEMENT

000460      114
000461      040
000462      117
000463      120
000464      105
000465      122
000466      101
000467      124
000470      117
000471      122
                     97  ;  TRANSFORM CHARACTER DATA INTO BINARY INTEGER
                     98  ;
000472  005067      99  NSCAN:  CLR   TEMP          ;CLEAR WORK AREA
        177554
000476  122713     100  NSCAN1: CMPB  #071,@R3      ;GREATER THAN 9?
        000071
000502  003020     101          BGT   OUT1          ;YES STOP
000504  122713     102          CMPB  #060,@R3      ;LESS THAN ZERO?
        000060
000510  002415     103          BLT   OUT1          ;YES STOP
000512  113302     104          MOVB  @(R3)+,R2     ;SAVE CHARACTER
000514  042702     105          BIC   #177760,R2    ;CONVERT TO BINARY
        177760
000520  012700'    106          MOV   #TEMP,R0      ;DIGIT ADDRESS
        000252
000524  012701'    107          MOV   #TEN,R1       ;MULTIPLIER
        000256
000530  104400     108          MUL                 ;MULTIPLY
000532  066702     109          ADD   TEMP+2,R2     ;CUMULATIVE SUM
        177516
000536  010267     110          MOV   R2,TEMP       ;ADJUST MULTIPLICAND
        177510
000542  000755     111          BR    NSCAN1        ;CONTINUE
000544  005767     112  OUT1:   TST   TEMP          ;INTEGER FOUND?
        177502
000550  001401     113          BEQ   OUT2          ;NO, ERROR
000552  000205     114          RTS   R5            ;YES, RETURN
                   115  OUT2:   PRINTC  ERR2,15.
000554  012746     116  +       MOV   #15.,-(6)
        000017
000560  012746'    117  +       MOV   #ERR2,-(6)
        000572
000564  104004     118  +       EMT   4             ;TRAP FOR I/O REQUEST
000566  000167     119          JMP   READ
        177212
000572      115    120  ERR2:   .ASCII  /MISSING OPERAND/
000573      111
000574      123
000575      123
000576      111
000577      116
```

```
  LOC     CODE    STMT     SOURCE STATEMENT

000600      107
000601      040
000602      117
000603      120
000604      105
000605      122
000606      101
000607      116
000610      104
000000              121          .END   BEGIN
```

SYMBOL TABLE

ADDTAB	000322R	BEGIN	000000R	DIVIS	000366R
ERR1	000452R	ERR2	000572R	ERR3	000420R
INPUT	000060R	LOOP	000044R	MINUS	000342R
MULT	000352R	NSCAN	000472R	NSCAN1	000476R
OPND1	000250R	OPND2	000246R	OPSCAN	000260R
OPTAB	000316R	OUT1	000544R	OUT2	000554R
OVER	000402R	PLUS	000332R	READ	000004R
R0	=%000000	R1	=%000001	R2	=%000002
R3	=%000003	R4	=%000004	R5	=%000005
SCAN1	000264R	SCAN2	000430R	SP	=%000006
TEMP	000252R	TEN	000256R	WRITE	000200R
WRITE1	000202R				

CSECT	ORIGIN	LENGTH	ID
.MAIN	000000	000612	001

NO STATEMENTS FLAGGED IN THIS ASSEMBLY

APPENDIX A

CHARACTER CODES

Character	Code	Character	Code	Character	Code
blank	040	A	101	a	141
!	041	B	102	b	142
"	042	C	103	c	143
#	043	D	104	d	144
$	044	E	105	e	145
%	045	F	106	f	146
&	046	G	107	g	147
'	047	H	110	h	150
(	050	I	111	i	151
)	051	J	112	j	152
*	052	K	113	k	153
+	053	L	114	l	154
,	054	M	115	m	155
-	055	N	116	n	156
.	056	O	117	o	157
/	057	P	120	p	160
0	060	Q	121	q	161
1	061	R	122	r	162
2	062	S	123	s	163
3	063	T	124	t	164
4	064	U	125	u	165
5	065	V	126	v	166

Character	Code	Character	Code	Character	Code
6	066	W	127	w	167
7	067	X	130	x	170
8	070	Y	131	y	171
9	071	Z	132	z	172
:	072				
;	073				
≤	074				
=	075				
≥	076				
?	077				
@	100				

APPENDIX B

PAL-11R SEPARATING OR TERMINATING CHARACTERS

Character	Function
:	label terminator
=	direct assignment delimiter
%	register term delineator
blank	item terminator, field terminator
#	immediate expression field indicator
@	deferred addressing indicator
(	initial register field indicator
)	terminal register field indicator
,	operand field separator
;	comment field delineator
+	arithmetic addition operator
-	arithmetic subtraction operator
&	logical AND operator
!	logical OR operator
"	double ASCII text indicator
'	single ASCII text indicator

APPENDIX C

ADDRESS MODE SYNTAX

Notation: r is an integer from 0 to 7.

R is a register expression.

E is an expression.

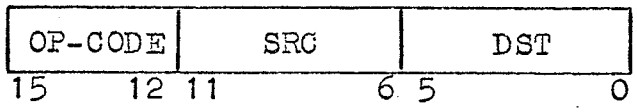
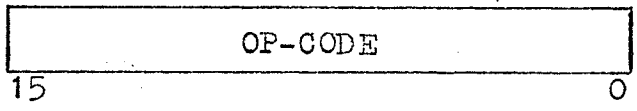
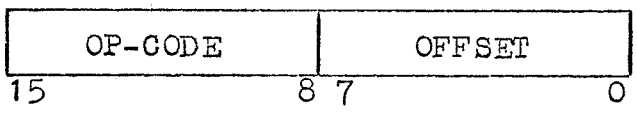
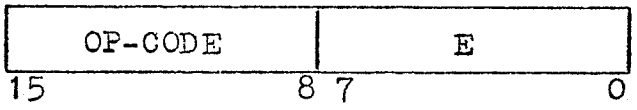
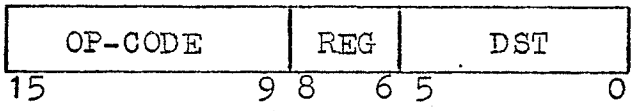
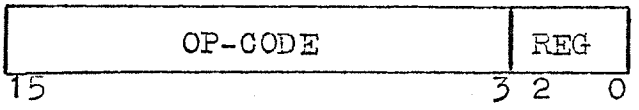
ER is an absolute expression in the range 0 to 7 or a register expression.

Octal Value	Mode Name	Syntax	Explanation
0r	register	R	Register R contains the operand.
1r	deferred register	@R or (R)	Register R contains the operand address.
2r	autoincrement	(ER)+	Register ER is incremented after use as the operand address
3r	deferred autoincrement	@(ER)+	Register ER is incremented after use as a pointer to the address of the operand.
4r	autodecrement	-(ER)	Register ER is decremented before use as the operand address.
5r	deferred autodecrement	@-(ER)	Register ER is decremented before use as a pointer to the address of the operand.
6r	index	E(ER)	E plus the contents of register ER is the operand address.
7r	deferred index	@E(ER)	E plus the contents of register ER is a pointer to the address of the operand.

Octal Value	Mode Name	Syntax	Explanation
27	immediate	#E	E is the operand.
37	absolute	@#E	E is the operand address.
67	relative	E	E is the address of the operand.
77	deferred relative	@E	E is a pointer to the address of the operand.

APPENDIX D

INSTRUCTION FORMATS

Instruction Class	Symbolic Format	Machine Format
double operand	OP A,A	
single operand	OP A	
operate	OP	
branch	OP E	
		$-128 \quad \frac{E-.-2}{2} \quad 127$
trap	OP E	
		$0 \quad E \quad 377_8$
subroutine call	JSR ER,A	
subroutine return	RTS ER	

APPENDIX E

INSTRUCTION MNEMONICS

This appendix contains an alphabetical list of all the machine instructions in PAL-11R.

Instruction	Mnemonic Op-Code	Machine Op-Code	Page	Condition Code Z	N	C	V
ADd Carry	ADC	0055DD	53	x	x	x	x
ADd Carry Byte	ADCB	1055DD	53	x	x	x	x
ADD	ADD	06SSDD	44	x	x	x	x
Arithmetic Shift Left	ASL	0063DD	57	x	x	x	x
Arithmetic Shift Left Byte	ASLB	1063DD	57	x	x	x	x
Arithmetic Shift Right	ASR	0062DD	56	x	x	x	x
Arithmetic Shift Right Byte	ASRB	1062DD	56	x	x	x	x
Branch on Carry Clear	BCC	1030XX	62	-	-	-	-
Branch on Carry Set	BCS	1034XX	62	-	-	-	-
Branch on EQual (zero)	BEQ	0014XX	61	-	-	-	-
Branch on Greater or Equal	BGE	0020XX	64	-	-	-	-
Branch on Greater Than	BGT	0030XX	64	-	-	-	-
Branch on HIgher	BHI	1010XX	65	-	-	-	-
Branch on HIgher or Same	BHIS	1030XX	65	-	-	-	-
BIt Clear	BIC	04SSDD	46	x	x	-	0
BIt Clear Byte	BICB	14SSDD	46	x	x	-	0
BIt Set	BIS	05SSDD	46	x	x	-	0
BIt Set Byte	BISB	15SSDD	47	x	x	-	0
BIt Test	BIT	03SSDD	47	x	x	-	0
BIt Test Byte	BITB	13SSDD	48	x	x	-	0
Branch on Less or Equal	BLE	0034XX	64	-	-	-	-
Branch on LOw	BLO	1034XX	65	-	-	-	-
Branch on LOw or Same	BLOS	1014XX	65	-	-	-	-

Instruction	Mnemonic Op-Code	Machine Op-Code	Page	Condition Code Z	N	C	V
Branch on Less Than	BLT	0024XX	64	-	-	-	-
Branch on MInus	BMI	1004XX	62	-	-	-	-
Branch on Not Equal	BNE	0010XX	61	-	-	-	-
Branch on PLus	BPL	1000XX	62	-	-	-	-
BRanch	BR	0004XX	60	-	-	-	-
Branch on oVerflow Clear	BVC	1020XX	63	-	-	-	-
Branch on oVerflow Set	BVS	1024XX	63	-	-	-	-
Clear Condition Codes	CCC	000257	67	0	0	0	0
CLear Carry bit	CLC	000241	66	-	-	0	-
CLear Negative bit	CLN	000250	67	-	0	-	-
CLeaR	CLR	0050DD	49	1	0	0	0
CLeaR Byte	CLRB	1050DD	49	1	0	0	0
CLear oVerflow bit	CLV	000242	66	-	-	-	0
CLear Zero bit	CLZ	000244	67	0	-	-	-
CoMPare	CMP	02SSDD	43	x	x	x	x
CoMPare Byte	CMPB	12SSDD	44	x	x	x	x
Clear Negative and Zero bits	CNZ	000254	67	0	0	-	-
COMplement	COM	0051DD	51	x	x	0	0
COMplement Byte	COMB	1051DD	51	x	x	0	0
DECrement	DEC	0053DD	52	x	x	-	x
DECrement Byte	DECB	1053DD	52	x	x	-	x
EMulator Trap	EMT	104000 to 104377	73	x	x	x	x
HALT	HALT	000000	68	-	-	-	-
INCrement	INC	0052DD	49	x	x	-	x
INCrement Byte	INCB	1052DD	49	x	x	-	x
Input/Output Trap	IOT	000004	69	x	x	x	x
JUMP	JUMP	0001DD	58	-	-	-	-

Instruction	Mnemonic Op-Code	Machine Op-Code	Page	Condition Code Z	N	C	V
Jump to SubRoutine	JSR	004RDD	70	-	-	-	-
MOVe	MOV	01SSDD	42	x	x	-	0
MOVe Byte	MOVB	11SSDD	43	x	x	-	0
NEGate	NEG	0054DD	51	x	x	x	x
NEGate Byte	NEGB	1054DD	51	x	x	x	x
No OPeration	NOP	000240	67	-	-	-	-
RESET	RESET	000005	69	-	-	-	-
ROtate Left	ROL	0061DD	55	x	x	x	x
ROtate Left Byte	ROLB	1061DD	55	x	x	x	x
ROtate Right	ROR	0060DD	55	x	x	x	x
ROtate Right Byte	RORB	1060DD	55	x	x	x	x
ReTurn from Interrupt	RTI	000002	68	x	x	x	x
ReTurn from Subroutine	RTS	00020R	72	-	-	-	-
SuBtract Carry	SBC	0056DD	54	x	x	x	x
SuBtract Carry Byte	SBCB	1056DD	54	x	x	x	x
Set Condition Codes	SCC	000277	67	1	1	1	1
SEt Carry bit	SEC	000261	67	-	-	1	-
SEt Negative bit	SEN	000270	67	-	1	-	-
SEt oVerflow bit	SEV	000262	67	-	-	-	1
SEt Zero bit	SEZ	000264	67	1	-	-	-
SUBtract	SUB	16SSDD	45	x	x	x	x
SWAp Bytes	SWAB	0003DD	56	x	x	0	0
TRAP	TRAP	104400 to 104777	73	x	x	x	x
TeST	TST	0057DD	52	x	x	0	0
TeST Byte	TSTB	1057DD	52	x	x	0	0
WAit for InTerrupt	WAIT	000001	69	-	-	-	-

APPENDIX F

ASSEMBLER DIRECTIVES

Directive	Operands	Operation
.ASCII	/..text../	generates 7-bit ASCII characters for text enclosed by delimiters
.ASECT	none	the start or continuation of an absolute program section
.BYTE	E,E,...	generates bytes of data equal to the values of the expressions
.CSECT	SYMBOL	the start or continuation of a relocatable program section (unnamed if no operand)
.END	E	indicates the physical end of a symbolic program and optionally specifies the start of execution
.ENDC	none	terminates the range of a conditional directive
.EVEN	none	forces the assembly location counter to be even by adding 1 if it is odd.
.GLOBL	S1,S2,...	specifies each name to be a global symbol
.IFDF	EL	assemble up to the matching .ENDC if the (logical) expression is defined
.IFG	E	assemble if E is greater than zero
.IFGE	E	assemble if E is greater than or equal to zero
.IFL	E	assemble if E is less than zero
.IFLE	E	assemble if E is less than or equal to zero
.IFNDF	EL	assemble if EL is not defined

Directive	Operands	Operation
.IFNZ	E	assemble if E is not zero
.IFZ	E	assemble if E is zero
.LIMIT	none	generates two words containing the low and high address limits of the relocatable sections
.RAD50	/CCC/	generates the radix-50 representation of up to three characters within the delimiters
.TITLE	SYMBOL	generates a name for the object module
.WORD	E,E,...	generates words of data equal to the values of the expressions

MONITOR REQUEST MACROS

Request	Macro Expansion	Function

PRINTC ADDR,LNGTH	MOV #LNGTH,-(6)
	MOV #ADDR,-(6)
	EMT 4
		print ASCII characters

PRINTO ADDR,LNGTH	MOV #LNGTH,-(6)
	MOV #ADDR,-(6)
	EMT 0
		print octal numbers

READC ADDR,LNGTH	MOV #LNGTH,-(6)
	MOV #ADDR,-(6)
	EMT 1
		read ASCII characters

READO ADDR,LNGTH	MOV #LNGTH,-(6)
	MOV #ADDR,-(6)
	EMT 5
		read octal numbers

EXTENDED MNEMONICS

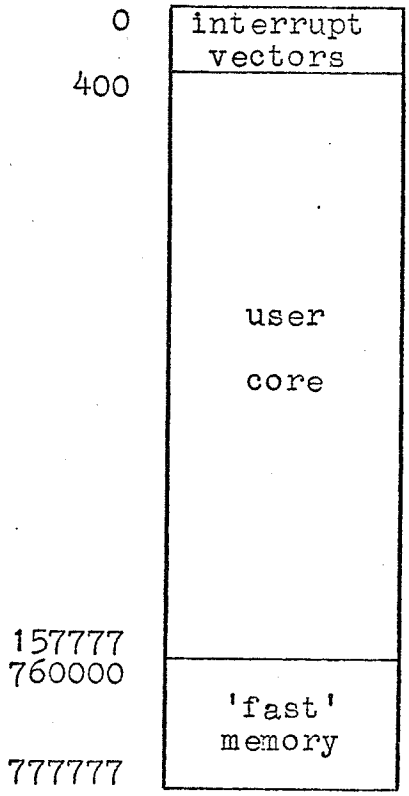
Instruction	Equivalent	Function
DIV	TRAP 1	R0 points to a dividend. R1 points to a divisor. After division, the quotient replaces the dividend.
MUL	TRAP 0	R0 points to a multiplicand. R1 points to a multiplier. After multiplication, the product replaces the multiplicand.
.DUMP	EMT 3	all of core is printed out in octal notation
.EXIT	EMT 2	all processing of the program is terminated, and control returns to the system.

APPENDIX G

ERROR MESSAGES

Error	Explanation
A	Addressing error. An address within the instruction is incorrect, or an illegal expression was formed.
B	Boundary error. Data or instructions are being assembled at an odd address in memory.
D	Doubly-defined symbol. A symbol is defined both as a label and by direct assignment.
I	Illegal character. An illegal character was encountered within a symbol name.
M	Multiple definition of a label. A symbol is defined as a label more than once in the same program section.
N	Number error. An illegal number was detected, or a decimal number was not terminated by a decimal point.
Q	Questionable syntax. This includes such errors as unmatched parentheses, too many arguments, illegal sequences of terminating characters, etc.
R	Register-type error. A register expression is out of range or used improperly.
T	Truncation error. More than the allowable number of bits are in a result, and it was truncated on the left.
U	Undefined symbol. An undefined symbol was encountered in an expression, and was assigned the value zero.

APPENDIX H STORAGE ADDRESS MAP



up to 32K words

4K peripheral storage: including general registers, program status register, and external device registers to be assigned at a later date

The interrupt vectors are assigned as follows:

Location	Function
4	instruction errors
10	reserved instructions
14	trace
20	IOT
24	power failure trap
30	EMT
34	TRAP

'Fast' memory assignment is as follows:

Location	Function
777700 - 777707	R0, R1, R2, ..., R7
777776 - 7777777	processor status register

PDP-11 ASSEMBLER LOGIC MANUAL

TABLE OF CONTENTS

INTRODUCTION

This manual describes the organization of the PDP-11 Assembler and its implementation in the PDP-11 System Simulation at the University of Manitoba. It is assumed that the reader understands the assembly process as discussed in the PDP-11 Assembler User's Guide and is also familiar with IBM System/360 Assembler Language. The program logic of the Assembler is presented here in increasing levels of detail including advice on maintenance and suggestions for future modification.

Section A describes the simulation environment and gives a general overview of the processing performed by the various routines in the PDP-11 Assembler. By studying this section, a system programmer may locate precisely the section of coding which should be modified to correct or introduce a particular feature. Section B describes many of the programming techniques used in designing the Assembler, and points out some of the constraints encountered. Section C describes how to change the Assembler parameters, and suggests some possible improvements.

A listing of the PDP-11 Assembler may be obtained by permission of Dr. Carol Abraham of the Department of Computer Science at the University of Manitoba.

SECTION A GENERAL STRUCTURE

SYSTEM SIMULATION

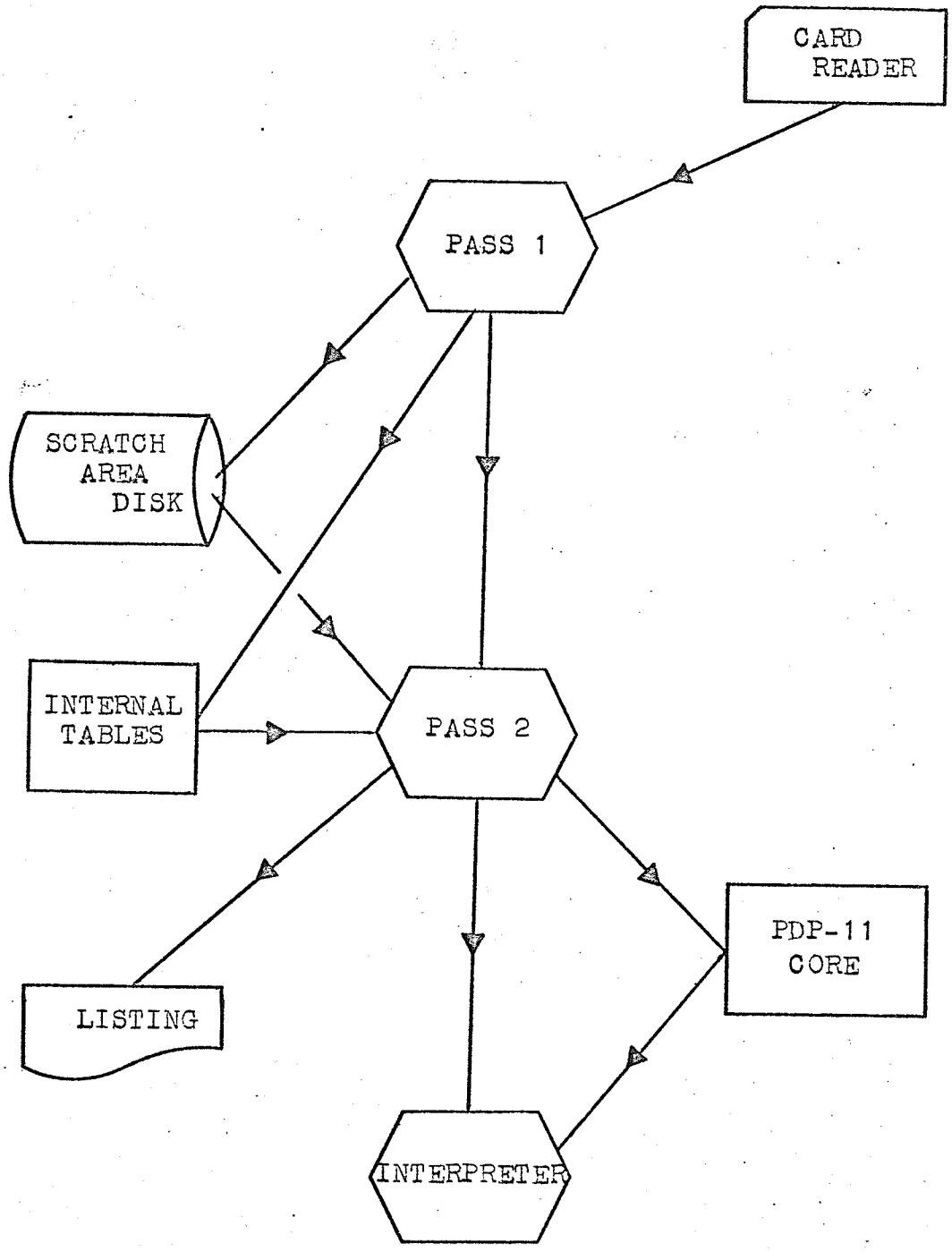
The simulated PDP-11 System is able to batch-process PAL-11R source programs, providing program listings and error diagnostics. The System is written in IBM/360 Assembler Language and consists of the following:

1. PDP-11 simulated memory (65K bytes)
2. a two-pass PDP-11 Assembler (13K bytes)
3. a PDP-11 machine-code Interpreter (8K bytes)
4. a scratch area on disk.

A user specifies the amount of PDP-11 core he wishes for his job. The Assembler assembles and loads the source program into that core. The Interpreter executes the program. The scratch area is used internally by the Assembler. There is no operating system as such, although a small part of the Assembler acts as a monitor by co-ordinating the assembly and interpretation phases for each job. This is a batch-processing system since one job is assembled, loaded, and executed before the next job may begin. Figure 1 illustrates the overall structure of the simulator.

The PDP-11 simulator normally executes in a 96K region (with disk space allocated dynamically as a scratch area) on an IBM/360 Model 65 under O.S. (IBM Operating System.) The entire package forms a relocatable object module on disk and may be invoked by the following JCL (Job Control Language):

FIGURE 1 SIMULATION STRUCTURE



```
//       JOB
//  EXEC ASMGLG,SIZE=96K
//LKED.SYSLIB DD DSN=YAF.A0299.CIA,UNIT=DISK,DISP=SHR,
//                VOL=SER=UM1405
//LKED.SYSIN  DD *
//GO.SYSUDUMP DD SYSOUT=A
//GO.STORE    DD DSN=JY,VOL=REF=ONE.MONTH,DISP=(NEW,DELETE),
//                SPACE=(TRK,(20,5))
//GO.PRINTOUT DD SYSOUT=A
//GO.READIN   DD *
                    PAL-11R Programs (batch)
/*
```

where STORE is the DCB (Data Control Block) for the scratch area on disk

PRINTOUT is the DCB for the assembly listing

READIN is the DCB for the card reader

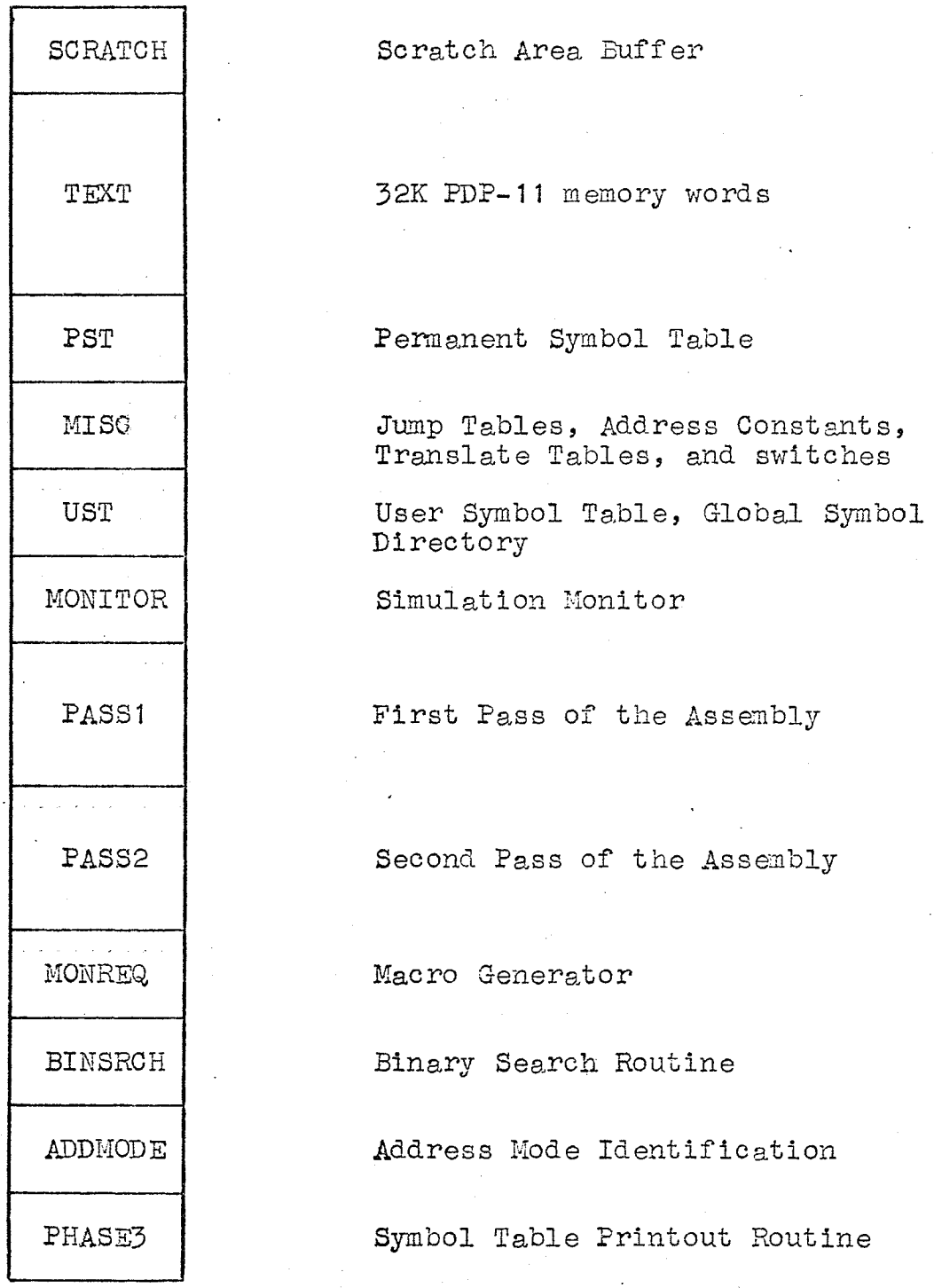
ASSEMBLER STRUCTURE

The Assembler program is organized as several distinct sections as shown in Figure 2. Some sections are data areas; some are single subroutines; and some are collections of related routines. They are summarized as follows:

SCRATCH AREA BUFFER

The scratch area buffer (SCRATCH) is used to store PAL-11R card images and a corresponding 'intermediate' text of numeric codes which are generated during Pass 1. These codes represent the instruction, the operands, and the addressing modes as encountered on that source card. The buffer has been arbitrarily

FIGURE 2 ASSEMBLER STRUCTURE



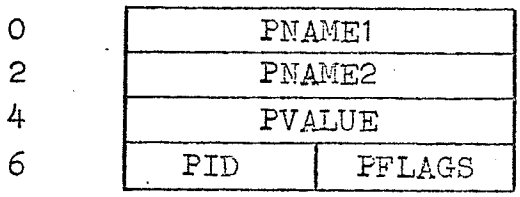
defined as 1280 (IBM/360) bytes in length. After twelve card images and their codes have been stored, the entire buffer is transferred to disk where it remains until re-examined during Pass 2. The scratch area buffer may be filled and saved on disk several times during the course of an assembly. By lumping data into 1280-byte sections in this manner, a scratch area is generated on disk with reduced I/O activity.

PDP-11 CORE

Up to 32K PDP-11 words of 16-bits each may be requested by a user job. The maximum area (called TEXT) is set aside as a permanent part of the Assembler. The PDP-11 machine code is loaded into this area during Pass 2. The Interpreter executes a job from this simulated core.

PERMANENT SYMBOL TABLE

The Permanent Symbol Table (PST) is an alphabetically ordered list containing the instruction mnemonics, assembler directives, and monitor requests available for the PDP-11 simulation. The format of each entry is as follows:[8]



where PNAME1, PNAME2 are two radix-50 packed triads representing a mnemonic;

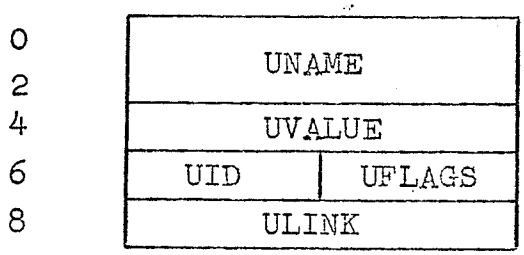
PVALUE is a machine operation code (for instruction mnemonics) or a displacement into an address table (for assembler directives);

PID is the section identification which is zero for all permanent symbols;

PFLAGS represents the instruction class of that entry.

USER SYMBOL TABLE

The User Symbol Table (UST) contains all the symbols used in a program. A symbol is entered as soon as it is encountered during Pass 1. The UST is link-listed into alphabetical order, with each symbol appearing only once. Each entry is ten bytes long with the following format:



where UNAME is two packed triads in radix-50 notation representing the symbol name;

UVALUE is the value associated with that symbol;

UID is the section identification assigned as follows:

	.ASECT	000
unnamed	.CSECT	001
named	.CSECT	002 - 007

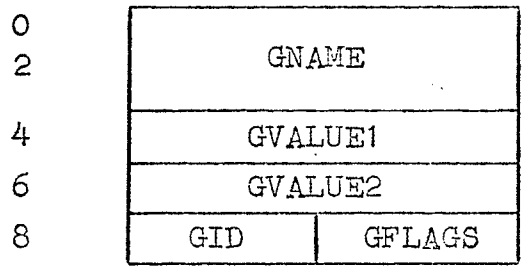
UFLAGS indicate the attributes of that symbol,

viz. relocatable, absolute, undefined, register symbol, etc.

ULINK is a pointer to the next higher symbol in alphabetical order.

GLOBAL SYMBOL DIRECTORY

The Global Symbol Directory (GSD) is perhaps a misnomer because it contains not all global symbols, but only those appearing in a .CSECT or .TITLE directive. The GSD is used by the Assembler to store information about the various program sections defined in a job. Each entry is ten bytes long with the following format:



where GNAME is two radix-50 packed triads representing the section name;

GVALUE1 is the entry address;

GVALUE2 is the section length;

GID is the section identification;

GFLAGS is the section attribute (relocatable or absolute.)

MONITOR

The Monitor is a subprogram which permits batch-processing by coordinating the Assembler and the Interpreter. The monitor performs the following functions:

1. Initializes the batch by opening files and determining the date for the job header page;

2. Scans for control cards;

3. Determines and initializes the Assembler options, including clearing the memory requested for that job;

4. Prints the header page;

5. Invokes the Assembler;

6. Calculates and prints the assembly time;

7. Invokes the Interpreter;

8. Calculates and prints the execution time;

9. Repeats steps 2 - 8;

10. Terminates the batch and closes all files.

PASS 1

During Pass 1, labels and their relative addresses are entered into the User Symbol Table. Also, symbols defined by direct assignment are evaluated and placed in the UST. All instructions are identified and their addressing modes are determined so that space may be reserved for any index words. An intermediate text of special code numbers is generated and stored in a scratch area on disk for use in Pass 2. In addition, many syntax errors are detected. After encountering the .END statement, an absolute address (ORIGIN) is calculated specifying exactly where the machine instructions will be loaded into PDP memory during Pass 2.

PASS 2

The intermediate text in the scratch area is examined and PDP-11 machine code is generated. All expressions are simplified, and all index words are calculated. A program listing is created line-by-line including card images from the disk, and any error messages detected in either pass. As a statement is so processed, its machine code is loaded into memory using the ORIGIN calculated in Pass 1.

PHASE3

The User Symbol Table, the Global Symbol Directory, and a summary of all error messages is printed out in the format described in the PDP-11 Assembler User's Guide. This ends the assembly phase, and control returns to the Monitor.

MACRO GENERATOR

The subroutine MONREQ is invoked during Pass 1 to generate PAL-11R source statements for a monitor request macro. The subroutine creates PAL-11R text in a special macro buffer (MACLIB) and effectively overrides the card reader so that the macro expansion created may be read from the buffer. The arguments from the monitor request are scanned and inserted as IBM hexadecimal characters into the buffer, without error checking. In this way, MONREQ acts as a pre-processor for monitor request macros. The generated statements are then processed normally by Pass 1.

BINARY SEARCH

The Binary Search Routine (BINSRCH) is used to detect a valid instruction and identify its type. During Pass 1, the radix-50 representation of a mnemonic is calculated, and passed to the BINSRCH routine. This value is compared to the mid-point of the Permanent Symbol Table, and half of the table is eliminated with successive searches until that entry is either found, or shown not to exist. In this way, any one of 105 possible instructions is detected within seven comparisons.

ADDRESS MODE IDENTIFICATION

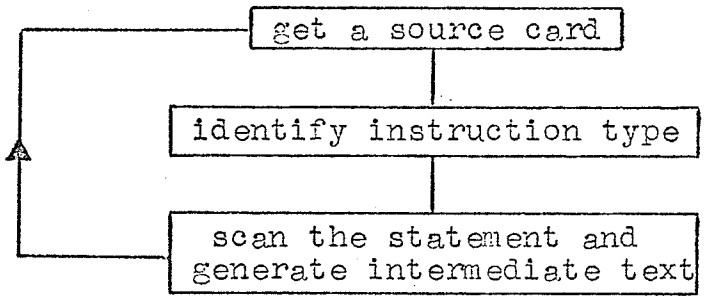
For each operand in single- and double-operand instructions, the address mode must be identified during Pass 1. As an operand is scanned, certain logical 'switches' are set to indicate the syntax and attributes of that operand. These switches are examined by the routine ADDMODE, and using the syntax formats described in Chapter 3 of the PDP-11 Assembler User's Guide, the appropriate address mode is generated into the scratch area. No error checking is done here, since illegal syntax is detected before ADDMODE is invoked.

PDP-11 ASSEMBLY

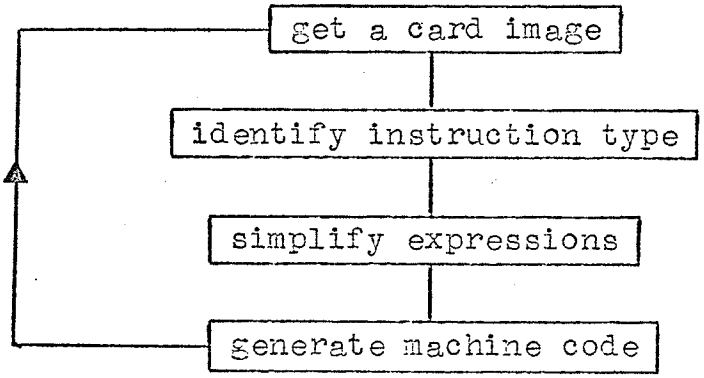
The PDP-11 Assembler is called a 'two-pass' assembler because it examines the contents of each source card twice as it assembles the program. In this case, the second pass is somewhat simplified by utilizing the intermediate text created by Pass 1, rather than re-scanning the source.

The basic steps in the assembly phase may be summarized as follows:

PASS 1:



PASS 2:

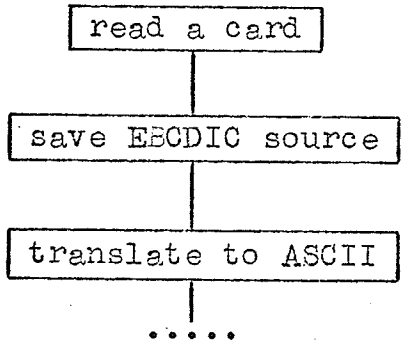


In order to understand the details of the Assembler logic, it is convenient to examine some of the programming techniques used.

TRANSLATE TABLE

A PDP-11 user expects data as 7-bit ASCII characters. However, with the IBM equipment at University of Manitoba, input to the computer is as EBCDIC characters.* Consequently, the Assembler includes a translation table (HEX2OCT) for converting all input into ASCII characters. In order to cut down simulation overhead, the EBCDIC source is first saved in the scratch area so that the program-listing may be created without re-translating the card images. Thus:

PASS 1:



Effectively, then, all scanning and processing by the Assembler uses ASCII characters.

JUMP TABLES

A jump table is a list whose entries represent addresses, where the position of an entry within the jump table indicates the purpose of that address. Jump tables can provide automatic decision-making by providing branch addresses without a series of comparisons and condition-checking. In the PDP-11 Assembler, jump tables control branching for instruction initialization, delimiter-handling, and assembler directive processing.

*It is possible to specify 8-bit ASCII input for the IBM/360, but this too needs to be transformed into 7-bit characters.

Each element in a jump table is a halfword (IBM) displacement from a pre-defined memory location (BASE) to a specific routine. This is a two-byte-per-entry saving over the normal full word address constant. Halfword S-constants could have been used instead, but displacements seemed more flexible. An S-constant requires address modification and reserved space in the instruction stream. Displacements, on the other hand, allow indexed addressing, while the tables may be stored as a distinct CSECT. In order to minimize the base register allocation, this space consideration was an important factor.

Altogether, there are seven jump tables as follows:

PASS 1:	INSTYP1	identify instruction type
	DIRTAB	identify assembler directives
	ADDRTAB	scan single- and double-operand instructions (with addressing modes)
	ADD4DIR	scan assembler directives and other instructions without addressing modes
PASS 2:	INSTYP2	identify instruction type
	DIRTAB2	identify assembler directives
	OPTAB	simplify and evaluate expressions

SCANNING

PAL-11R Assembler Language has a well-defined syntax structure which relies on special terminating and separating characters. Thus scanning in Pass 1 is accomplished by means of a TRanslate and Test (TRT) instruction on the ASCII source. The delimiters are assigned the following values from the translate

table TESTTAB:

Character	Byte Code (hexadecimal)
+	02
-	04
&	06
!	08
,	0A
(	0C
)	0E
;	10
=	12
:	14
%	16
blank	18
@	1A
#	1C
'	1E
"	20

These hexadecimal codes represent a displacement into a jump table. Successive delimiters point to successive halfwords in the table.

Several jump tables exist which specify different branch addresses for the same delimiters depending on the pass, or which type of instruction is being processed. For example, in branch instructions, the delimiter # is illegal and will generate a 'Q' flag (Questionable syntax error.) In single operand instructions, however, the # indicates immediate addressing. Similarly, Q-errors are generated whenever address mode syntax is encountered illegally.

The scanning uses the result of the TRT instruction (in Register 2) as follows:

```
BRTAB    LH    R14,ADDRTAB(R2)        GET DISPLACEMENT
         B     B@SE(R14)
```

As soon as the instruction type is identified, the appropriate jump table is inserted into the instruction to complete the scan. Jump tables are 'swapped' by address modification using S-constants:

```
         MVC   BRTAB+2(2),ADD2      SWAP JUMP TABLES

ADD1     DC    S(ADDRTAB)
ADD2     DC    S(ADD4DIR)
```

INSTRUCTION TYPE IDENTIFICATION

In a similar way, jump tables are used to identify the instruction type. Here the displacement is taken from the Permanent Symbol Table as the PFLAG for that mnemonic. They are summarized as follows:

Instruction Type	Byte Code (hexadecimal)
operate	00
assembler directive	02
single operand	04
monitor request	06
double operand	08
reserved for comment	0A
RTS	0C
reserved for error	0E
branch	10
reserved for direct assignment	12
JSR	14
unused	16
trap	18

ASSEMBLER DIRECTIVES

Since each of the Assembler directives performs a unique function, another jump table is used to locate the branch address for the appropriate directive-handling subroutine. This displacement is also taken from the PST as the PVALUE

entry as follows:

Directive	Displacement
.WORD	00
.TITLE	02
.RAD50	04
.LIMIT	06
.GLOBL	08
.EVEN	0A
.END	0C
.ENDC	0E
.CSECT	10
.BYTE	12
.ASECT	14
.ASCII	16
.IFZ	18
.IFNZ	1A
.IFNDF	1C
.IFLE	1E
.IFL	20
.IFGE	22
.IFG	24
.IFDF	26

This enables different subroutines to be called in Pass 1 and in Pass 2 for the same assembler directive.

USER SYMBOL TABLE

A symbol table is required for several reasons. It provides the information which enables the Assembler to replace operands by storage addresses. Simplifying expressions and error-checking involve examining a symbol's attributes as stored in the UST. Register symbols must be identified to help determine the addressing mode. Relocatable symbols must be detected in creating the machine code. In addition, throughout the assembly, a symbol is referred to by its symbol table location rather than the character string which makes up its name.

ATTRIBUTES

Both the UST and the GSD use the following format for the symbol attributes. These 'flags' each occupy one bit of the 8-bit field: (UFLAGS, GFLAGS)

FLAGS

=	%	G	R			.	U
7	6	5	4	3	2	1	0

Bit	Name	Function
=	ABSFLG	indicates direct assignment
%	REGFLG	indicates a register symbol
G	GLBFLG	indicates a global symbol
R	RELFLG	indicates a relocatable symbol
.	PCFLG	indicates the program counter (.)
U	UNDF	indicates an undefined symbol

These 'flags' are set or cleared as the conditions dictate whenever a symbol is first encountered, or appears as a label, or is defined by direct assignment.

LINKED-LIST

The UST is link-listed into alphabetical order. At the expense of increasing the size of each symbol table entry to include a link field, two major advantages resulted:

1. Searching time is reduced since only those entries whose name field is alphabetically lower than the 'test symbol' need be compared. As soon as a higher name is detected, searching ends.

2. The User Symbol Table can be alphabetically printed without

further sorting.

Other symbol table structures were considered but rejected for the following reasons:

1. A simple sequential list, although adequate for small programs, is inefficient for large programs.
2. Since variable names are encountered in a random order, the use of a tree structure will not necessarily form a 'balanced tree'[9] which will minimize search time. Further, a tree structure requires two link fields rather than the one used.
3. A binary ordered symbol table is wasteful of time since whenever an entry is made, all symbols located higher in the table must be moved to preserve the order.[10] Moreover, the expected symbol table size is too small to make a binary search feasible.

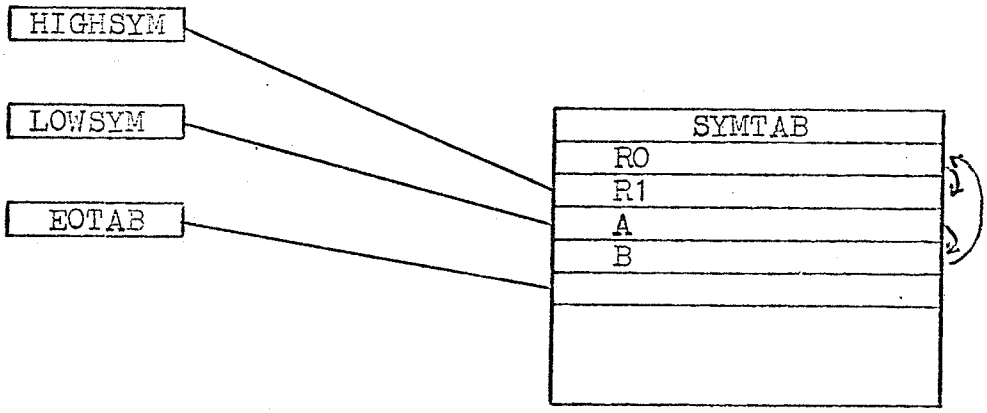
SEARCHING

The subroutine SEARCH, part of Pass 1, is used to place a symbol into the User Symbol Table and/or return its symbol table address in Register 9. This operation, described below, is depicted in Figure 3.

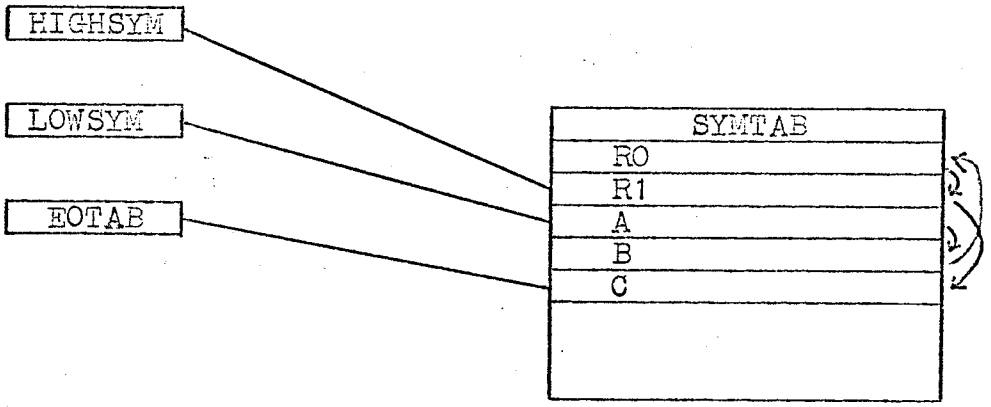
The link field (ULINK) in each symbol table entry is a halfword displacement from the beginning of the symbol table to the next higher entry. Associated with the UST are three pointers:

HIGHSYM	points to the highest entry
LOWSYM	points to the lowest entry
EOTAB	points to the next available location

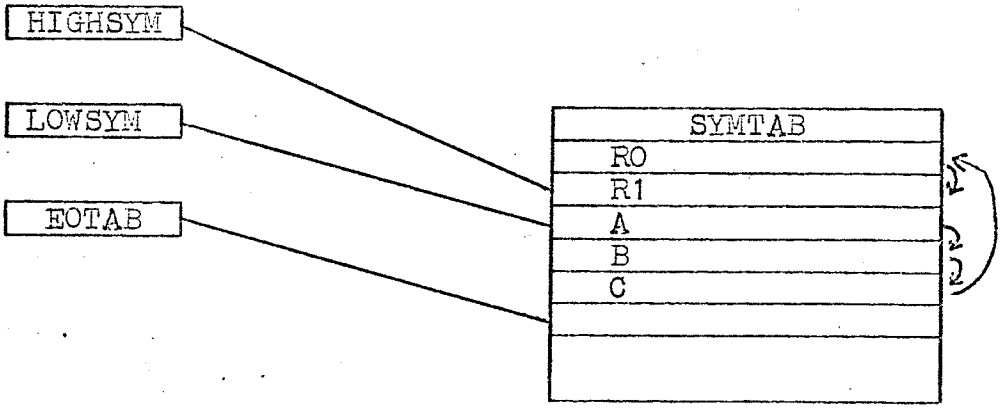
FIGURE 3 SEARCHING BY LINKED-LIST



1. Before search



2. Place new entry at the end of the table.



3. Link alphabetically and reset pointers.

A symbol is previously transformed into its radix-50 equivalent. The 'test symbol' is always entered at the physical end of the table (as provided by EOTAB) and linked from the current highest entry. Thus, during the ensuing ordered search -- which proceeds alphabetically by means of the links -- one of the following must occur:

1. The table is empty, in which case the given symbol is the first entry, and the pointers are initialized.

2. The test symbol is found before the end of the list. Then, that symbol already appears in alphabetical order, so no changes result.

3. The test symbol is found at the end of the list. That symbol is therefore a new symbol, and in fact the new highest symbol.

4. A value higher than the test symbol is detected. Then the test symbol is a new entry, and must be linked appropriately. It may be the new lowest entry.

SCRATCH AREA

The scratch area contains a summary of everything encountered on the source cards. Each original source card is represented by a card image followed by a sequence of code words which are in fact displacements (into jump tables or the UST) or other bytes of information. Pass 2, then, selects the correct jump table and uses these displacements to locate the branch address for the required processing. In this way, information detected during Pass 1 is recovered during Pass 2

with a minimum of overhead.

INSTRUCTION TYPES

The instruction type code in the scratch area is the same as described for Pass 1. It is used as a displacement into the jump table INSTYP2.

PASS 2 OPERATORS

Note that not all delimiters in a statement will appear as code words in the scratch area. Several characters are only needed in Pass 1 to describe the addressing mode (for example # and @ and %) or to generate words of absolute data (such as ' and "). Consequently, an abridged code of operators which represent displacements into the jump table OPTAB is as follows:

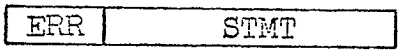
Operator	Name	Byte Code (hexadecimal)
EOF	end of file	00
+	plus	02
-	minus	04
&	AND	06
!	OR	08
,	comma	0A
(	left bracket	0C
)	right bracket	0E
ERR	error	10

ERROR MESSAGES

The error operator indicates that the next byte in the scratch area specifies which type of error was detected. This is the means by which all errors identified in Pass 1 are transmitted to Pass 2.

Error Message		Byte Code
A	Addressing error	00
B	Boundary error	01
D	Doubly defined symbol	02
I	Illegal character	03
M	Multiple label	04
N	Number error	05
Q	Questionable syntax	06
R	Register error	07
T	Truncation error	08
U	Undefined symbol	09

Whenever an error is encountered, the appropriate error flag is inserted onto the program listing, and a message code is stored in a special array ERRFILE as follows:



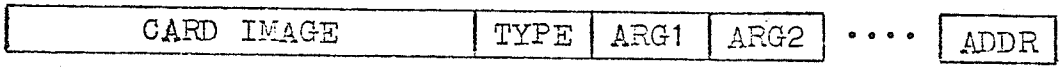
where ERR is a one-byte error code from above,

STMT is the statement number in a 3-byte packed decimal format.

This error table is printed out at the end of each job to provide a summary of all the errors.

SCRATCH AREA FORMAT

The scratch area has a variable length format since the number of code words depends upon the type of instruction, the number of operands, the number of terms in the operands, and the number of errors detected. The general format of the scratch area is as follows:



1. The card image is 80 bytes of EBCDIC characters.

2. The instruction type is a four-byte field:

TYPE

IC	-	OP-CODE

where IC is the instruction type code;

- is an unused byte to preserve halfword alignment;

OP-CODE is the machine operation code, or a displacement which identifies an assembler directive.

3. The arguments are each normally a four-byte field:

ARG

OP	SW	VALUE

where OP is a Pass 2 operator;

SW is a logical switch as follows:
00 - if the operand is a symbol
FF - if the operand is absolute data;

VALUE is a displacement into the UST, or an actual value of the operand depending on SW.

If the operator is the error operator, the field is two bytes as described previously.

4. The address field is present only in single- or double-operand instructions.

ADDR

OP	A	SW	-

where OP is either 00 (EOF) or 0A (comma)

A is the address mode for that operand;

SW indicates the nature of any index words
00 - no index words
0F - relative index word
FF - absolute index word

Where no addressing modes are involved, the operators 00 and 0A use two-byte fields where the second byte is unused. Double operand instructions have another set of arguments followed by an address field. Figure 4 illustrates three possible scratch area representations.

BRANCH MASKS

Throughout the Assembler, instruction modification is used to change the masks controlling various branch instructions of the form:

```
BC    0,DEST
```

Here the coded branch masks are dummy parameters and are reset periodically by certain subroutines according to the type of instruction being processed. Branches become successful or unsuccessful by inserting one of the following values into the second byte of the instruction.

Name	Value	Result	
NOP	00	BC	0,DEST
BRA	F0	BC	15,DEST

For example:

```
COMMA   BC    0,STXERR
        MVI   COMMA+1,BRA      DISABLE FURTHER COMMAS
```

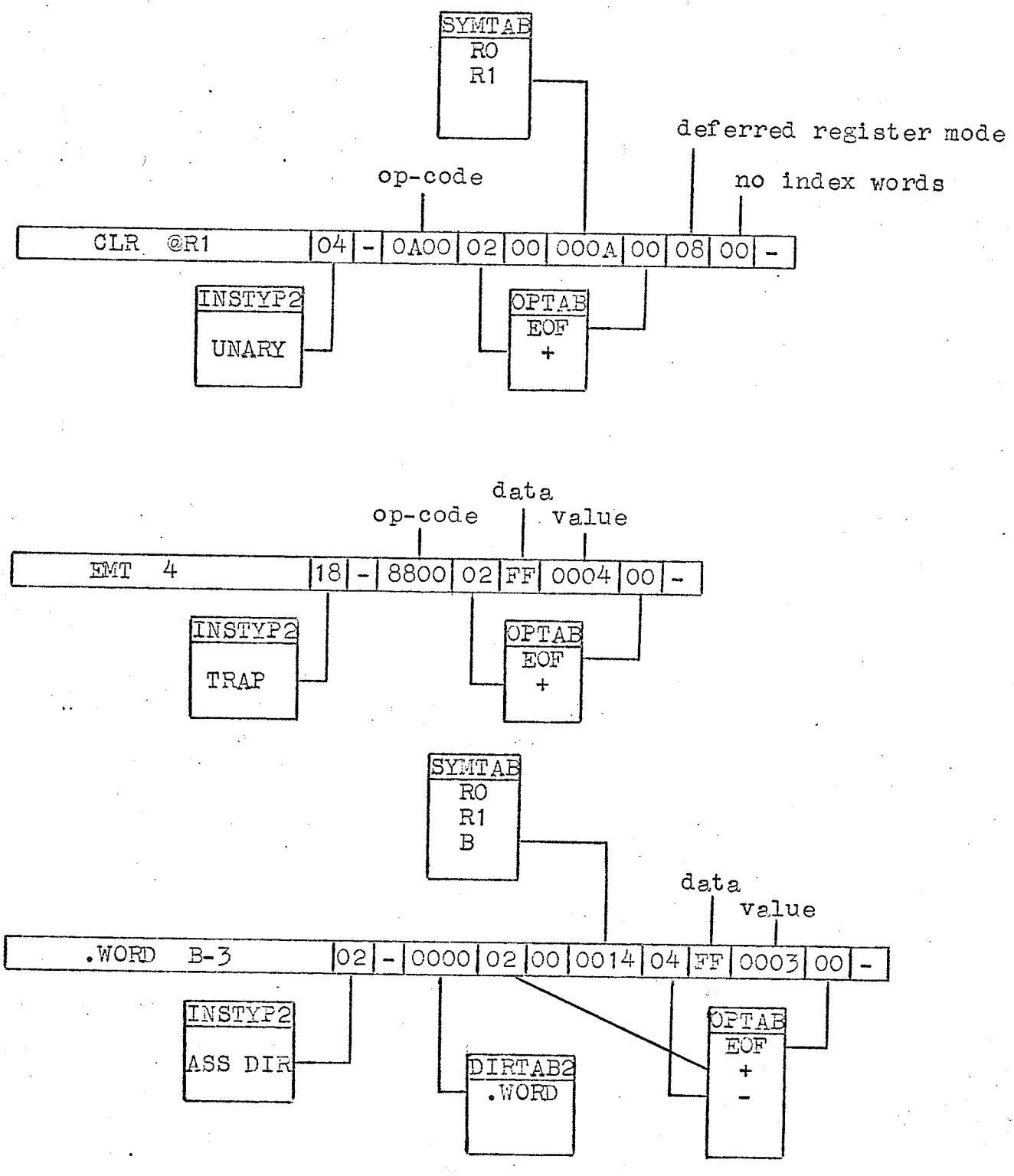
In single operand instructions, no commas are allowed; thus the above is initialized by

```
        MVI   COMMA+1,BRA      NO COMMAS ALLOWED
```

In double operand instructions, one comma is expected:

```
        MVI   COMMA+1,NOP      COMMA ALLOWED
```

FIGURE 4 SCRATCH AREA FORMATS



SECTION C MAINTENANCE

RESTRICTIONS

This system is still under a state of developement, and, needless to say, there are certain restrictions imposed:

1. Up to 256 user symbols allowed;
2. Up to 8 control sections;
3. Approximately 1200 cards per program maximum;
4. No external devices. Reading and printing must be done by the monitor request macros.

The first three restrictions may be altered as follows:

1. The User Symbol Table size is defined by the parameter SYMLGTH which appears in an EQU statement at the beginning of the Assembler. Any other desired size may be substituted here to enlarge the table without any other programming considerations.

2. Additional control sections may be allowed by defining attributes for named control sections in the GSD. For example:

```
DC    4H'0',X'0810'
```

would define a ninth CSECT with the ID number 008, and a relocatable attribute.

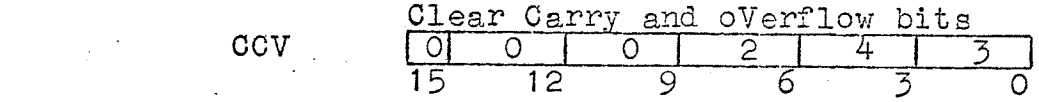
3. A program of over 1200 cards will overflow the scratch area on disk. More disk space may be obtained by altering the scratch area DD card.

4. No external devices were implemented in order to limit the scope of this thesis at the master's level.

NEW INSTRUCTION MNEMONICS

Additional instruction mnemonics may be added to the Permanent Symbol Table by inserting the appropriate hexadecimal data and preserving the table's alphabetical order. The format is described in Section A.

For example, to insert a new operate instruction CCV:



where the radix-50 equivalent of CCV is 011706_8 or 1306_{16}, insert the following between the instructions CCC and CLC:

```
DC     X'13C6000000A20000'
```

ADDITIONAL ERROR MESSAGES

The error messages given are somewhat general in nature. More specific diagnostics may be printed by increasing the number of error-code bytes (as described in Section B) and inserting a corresponding message into the array ERRNOTE. Each message is currently 60 characters long. For example, the Q-errors might be subdivided as follows:

Code	Error Message
0A	Q Syntax Error: Unmatched parenthesis
0B	Q Syntax Error: Unexpected # detected
etc.	

where the code indicates which message is to be printed; and the error message would be defined as part of the array ERRNOTE with the appropriate displacement.

MONITOR REQUESTS

Currently the four monitor request macros are processed identically by generating three PAL-11R instructions. If new monitor requests were invented (perhaps .OPEN and .CLOSE for files) a different type of macro expansion might be required. In fact, the entire macro feature could be enlarged to handle all the extended mnemonics. Monitor requests then could be handled in much the same way as assembler directives. That is:

1. The instruction type would still be identified by the jump table INSTYP1.

2. Specific monitor requests would be distinguished by defining a new jump table, say MONTAB, as follows:

Request	Displacement	Trap
PRINTO	00	EMT 0
READC	02	EMT 1
.EXIT	04	EMT 2
.DUMP	06	EMT 3
PRINTC	08	EMT 4
READO	0A	EMT 5
.OPEN	0C	EMT 6
.CLOSE	0E	EMT 7
MUL	10	TRAP 0
DIV	12	TRAP 1
SIN	14	TRAP 2
SQRT	16	TRAP 3

Thus, new management facilities and perhaps a system library could be included by simply enlarging the scope of the monitor requests. Naturally, new software would have to be added to the Assembler (and the Interpreter) to implement these new features.

OPERATING SYSTEM

The monitor requests attempt to perform the functions of an operating system in an artificial manner. There is no real operating system software in the PDP-11 core. Rather the onus is on the Interpreter to provide these features through IBM's O.S. However, this is still feasible, and can remain invisible to a user of the simulation.

Thus, such things as external devices, formatted I/O, and even asynchronous operations may be successfully simulated. The chief problem is in designing and interpreting such features. The Assembler must provide the Interpreter with whatever information is required. Thus, formats for control blocks, and system tables must be rigorously defined. Since the Assembler and the Interpreter were written more or less independently, a full-scale operating system was not implemented.

CONCLUSION

The PDP-11 Simulator is currently being used in conjunction with a graduate level course on computer hardwares at the University of Manitoba. Students are not only explained the architecture of the PDP-11, and the principles of stack-processing, but are encouraged to apply this knowledge by creating and testing programs on the PDP-11 Simulator. Thus students are allowed to develop sophisticated software for the PDP-11, and even to design their own operating systems. In this way, the simulator is a valuable teaching aid and, as a supplement to formal lectures, can provide a better insight into the capabilities of the PDP-11 computer.

Certainly, due to this exposure to the PDP-11 simulation, students may urge that additional features become available. It is expected that external devices and priority interrupts will be implemented in the near future. Other possibilities include formatted input and output, floating-point arithmetic, and a system library of common user subroutines.

Information on procedures concerning the use of this simulation, including any revisions which may be made in the future, may be obtained from Dr. Carol Abraham of the Department of Computer Science at the University of Manitoba.

REFERENCES

1. Digital Equipment Corporation, PAL-11R Assembler Programmer's Manual, (DEC-11-ASDA-D), 1971, Maynard, Massachusetts, Chapter 1, p. 1-1.

2. Digital Equipment Corporation, PDP-11 Handbook, 1969, Maynard, Massachusetts, Chapter 2, pp. 5-10.

3. PAL-11R Assembler Programmer's Manual, op. cit., Chapter 7, p. 7-6.

4. PDP-11 Handbook, op. cit., Chapter 4, pp. 17-43.

5. PAL-11R Assembler Programmer's Manual, op. cit., Chapter 8, pp. 8-1 - 8-10.

6. Digital Equipment Corporation, PDP-11 Disk Operating System, (DEC-11-SERA-D), 1971, Maynard, Massachusetts, Chapter 2, p. 2-1.

7. PAL-11R Assembler Programmer's Manual, op. cit., Chapter 9, p. 9-4.

8. PAL-11R Assembler Programmer's Manual, op. cit., Appendix C, pp. C-1 - C-10.

9. Lee, J. A. N., The Anatomy of a Compiler, Reinhold Publishing Corporation, 1967, New York, Chapter 4, pp. 86-97.

10. Barron, D. W., Assemblers and Loaders, MacDonald & Company, London, 1969, Chapter 2, pp. 15-17.