Instruction Set

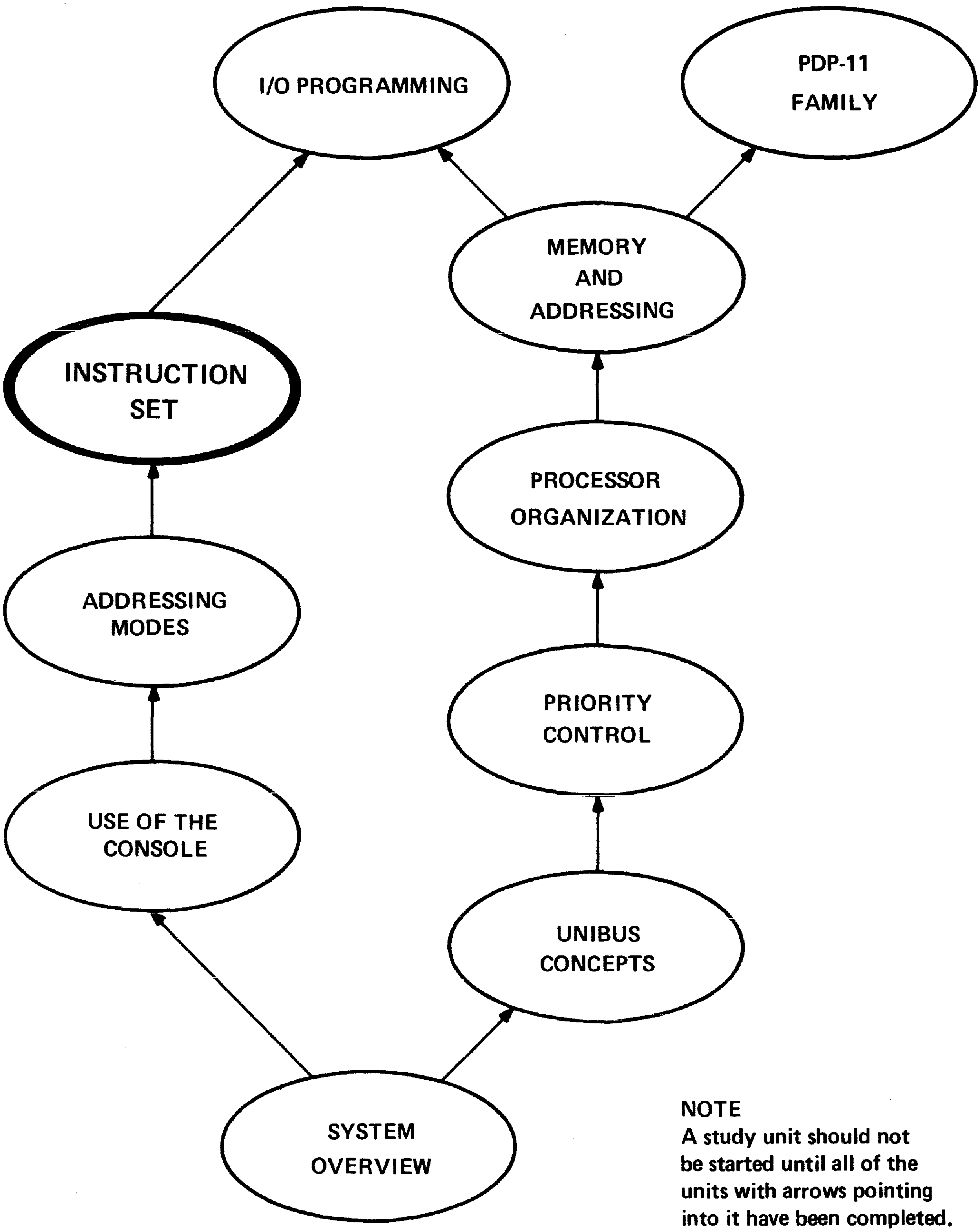
student workbook introduction to the pdp11

digital equipment corporation · maynard. massachusetts

lst Printing, March 1974
2nd Printing (Rev), November 1974
3rd Printing (Rev), April 1977

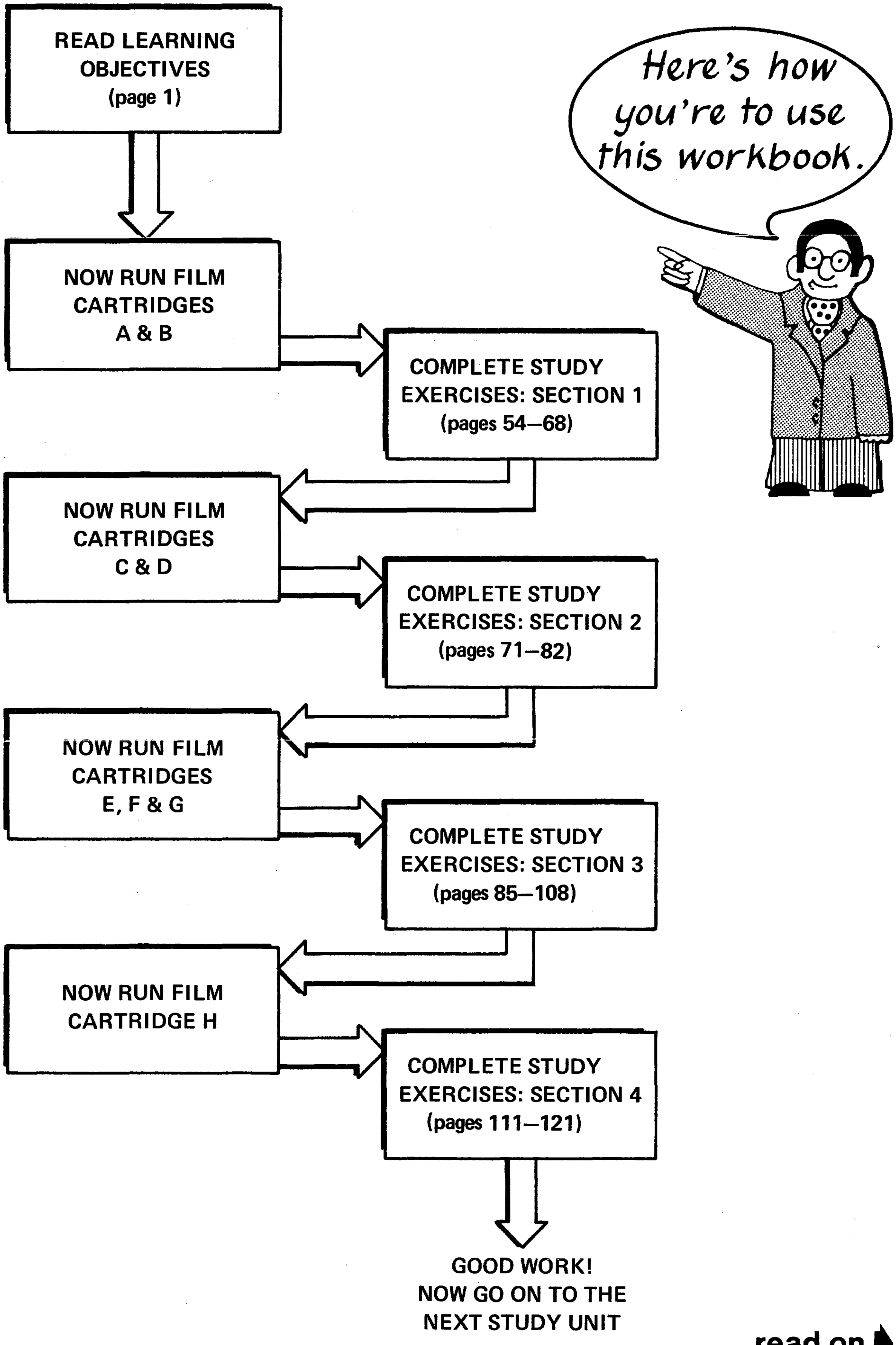
Printed in U.S.A.

course map



NOTE
A study unit should not be started until all of the units with arrows pointing into it have been completed.

read on



READ LEARNING OBJECTIVES (page 1)

NOW RUN FILM CARTRIDGES A & B

COMPLETE STUDY EXERCISES: SECTION 1 (pages 54–68)

NOW RUN FILM CARTRIDGES C & D

COMPLETE STUDY EXERCISES: SECTION 2 (pages 71–82)

NOW RUN FILM CARTRIDGES E, F & G

COMPLETE STUDY EXERCISES: SECTION 3 (pages 85–108)

NOW RUN FILM CARTRIDGE H

COMPLETE STUDY EXERCISES: SECTION 4 (pages 111–121)

GOOD WORK!
NOW GO ON TO THE
NEXT STUDY UNIT

read on ▶

objectives

After completing this study unit you should be able to . . . .

★ Recognize and use the three main instruction groups in the PDP-11 basic instruction set:

- Single-Operand Instructions
- Double-Operand Instructions
- Program Control Instructions

★ Use the appropriate addressing mode with any of the single-operand or double-operand instructions.

★ Select the most suitable branch instruction for the program conditions being tested and calculate the proper offset value for the branch instruction.

★ Understand and use basic programming techniques such as "loops" and "tallys."

★ Write PDP-11 programs by using the following steps:

- Analyze the problem by constructing a flow chart of the job to be performed.
- Solve the problem by implementing the flow chart with appropriate instructions.
- Refine the resultant program by eliminating and/or combining instructions.

★ Describe terms such as "offset," "condition codes," "tally," "loop," "pointer," "counter," "branch," and "initialize."

★ Write a program or any individual instruction in both the assembler syntax (mnemonic code) and in machine language (octal code).

★ Use the PDP-11 instruction card for selecting and coding instructions.

read on ➧

additional resources

- **PDP-11/04/05/10/35/40/45 Processor Handbook**

  If necessary, read Chapter 1, Paragraph 1.6, for a brief explanation of octal and binary number systems.

- **PDP-11/04/05/10/35/40/45 Processor Handbook**

  Read or review Chapter 4, Instruction Set. Also read Chapter 5 to familiarize yourself with various PDP-11 programming techniques.

- **PDP-11 Paper Tape Software Handbook**

  Read Chapter 2, Writing Assembly Language Programs.

read on ➧

review: basic information

Topic	*Key Points*	*Visual Ref.*
basic concepts	★ The PDP-11 instruction set is divided into two parts: ● *Basic* instructions, which are implemented in all PDP-11 processors. ● *Special* instructions, which are available only with larger processors, such as the PDP-11/45 and PDP-11/70.	3, 4
instruction groups	★ Most PDP-11 instructions fall into one of three main categories: ● *Single-Operand* – one part of the word specifies the job; the second part provides information for locating the operand. ● *Double-Operand* – the first part of the word specifies the job; the remaining two parts provide information for locating two operands. ● *Program Control* – the first part of the word specifies some action to be taken; the second part indicates where the action is to take place in the program.	5–7

SINGLE OPERAND INSTRUCTION

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>09</td><td>08</td><td>07</td><td>06</td><td>05</td><td>04</td><td>03</td><td>02</td><td>01</td><td>00</td></tr>
<tr><td colspan="10">OP CODE</td><td colspan="6">DST</td></tr>
</table>

DOUBLE OPERAND INSTRUCTION

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>09</td><td>08</td><td>07</td><td>06</td><td>05</td><td>04</td><td>03</td><td>02</td><td>01</td><td>00</td></tr>
<tr><td colspan="4">OP CODE</td><td colspan="6">Src</td><td colspan="6">DST</td></tr>
</table>

PROGRAM CONTROL INSTRUCTION

<table>
<tr><td>15</td><td>14</td><td>13</td><td>1 2</td><td>11</td><td>10</td><td>09</td><td>08</td><td>07</td><td>06</td><td>05</td><td>04</td><td>03</td><td>02</td><td>01</td><td>00</td></tr>
<tr><td colspan="8">OP CODE</td><td colspan="8">OFFSET</td></tr>
</table>

MI-0499

read on ➧

single-operand

Topic	*Key Points*	*Visual Ref.*
format	★ The format of all single-operand instructions is as follows: ● Bit 15 – indicates word or byte operation. ● Bits 14–6 – indicate the "op" code which specifies the job or operation to be performed. ● Bits 5–0 – consist of a 3-bit addressing mode field and a 3-bit general register field. These two fields are referred to as the "destination field." ● When using a word operation (bit 15 is a 0), the normal instruction mnemonic is used, such as CLR. When using a byte operation (bit 15 is a 1), then B is added to the mnemonic, such as CLRB.	8–16
using instructions	★ Whenever we use a series of instructions sequentially, it is called "straight line programming." Note that each instruction is executed only once.	17, 18
	★ A more efficient method of using instructions is available. This is called a program "loop." Note that each instruction in the loop is executed many times.	
program loops	★ When programming, it is often desirable to repeat a number of instructions. A program "loop" is an efficient method of handling repetitive operations.	19–20
	★ Program loops can be formed by using "branch" instructions. ● A branch instruction tests a condition to see if it has been met.	

(continued on next page)

read on ➧

Topic	*Key Points*	*Visual Ref.*
program loops (Cont.)	● If the condition is met, the program branches to a new location. This new location can be one of the starting instructions. Thus, the program "loops" back and repeats a group of instructions. ● If the condition is not met, the program continues with the next instruction.	
using a "tally"	★ Eventually, we must exit the "loop" and continue with the balance of the program. In order to know when to exit, we use a "tally." ★ A "tally" is used to keep count of how many times we "loop" in order to tell us when to stop looping and continue with the program. ★ A tally keeps count as follows: ● The number of items or operations to be counted is preset. ● The count is then decremented (or incremented) each time the program loop is executed. ● At the end of each program loop, the count is checked to determine if the program is to exit from the loop.	21–26
making a tally	★ One way of constructing a tally is as follows: ● A count is loaded into one of the GPRs. This count specifies the number of times a series of instructions must be repeated. ● Each time these instructions are executed, the count is decremented by one. ● A check for "done" is performed by using a BRANCH IF NOT ZERO instruction.	

(continued on next page)

read on ➧

Topic | *Key Points* | *Visual Ref.*

making a tally (Cont.)

- If the GPR is *not* zero, the program loops around and executes the same instructions again.

- Once the number in the GPR reaches zero, the branch instruction is no longer effective, and the program executes the next sequential instruction. In other words, the program exits from the loop.

using a tally

★ The following short program uses a "tally" to control a "loop," which clears out a specific block of memory. Note that the program has been set up to clear 30_8 byte locations beginning at memory address 600.

27–45

```
                (R0)  =  600
                (R1)  =   30

       LOOP:    CLRB (R0)+
                DEC R1
                BRANCH IF NOT ZERO TO LOOP
                HALT
```

program analysis

★ The following is an analysis of this program:

- The CLRB (R0)+ instruction clears the contents of the location specified by R0. In other words, R0 contains the *address* of the operand.

- Because the auto-increment addressing mode is used, the pointer (R0) automatically moves to the next memory location after execution of the CLRB instruction.

- Register R1 indicates the number of locations to be cleared and is, therefore, a counter. Counting is performed by the DEC R1 instruction. In other words, each time a location is cleared, it is counted by decrementing R1.

(continued on next page)

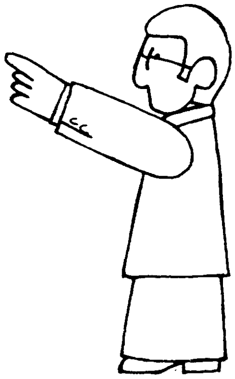
read on ➧

Topic	*Key Points*	*Visual Ref.*
program analysis (Cont.)	● The BRANCH IF NOT ZERO instruction checks for done. If the counter is *not* zero, the program branches back to start to clear another location. If the counter *is* zero, indicating done, then the program executes the next instruction, HALT.	
modifying the program	★ The above program can be modified to clear full words instead of bytes by making only two modifications. ● Changing the first instruction to the *word* instruction, CLR (R0)+. ● Loading a word, rather than a byte, count into R1. In other words, loading R1 with octal 14 rather than 30.	46–48
positive and negative numbers	★ In a PDP-11 system negative numbers are written in 2's complement notation. ● If we are working with *signed* numbers, bit 15 (the MSB) indicates whether the number is positive or negative. ● A zero in bit position 15 denotes a positive number; a one denotes a negative number. **NOTE** A PDP-11 word can be treated as an *unsigned* number in which all 16 bits represent the number; or the word can be treated as a *signed* number, in which the MSB is read as a sign (0 for positive, 1 for negative). The *same* word can be interpreted as either a signed or unsigned number.	50–52

read on ➧

single-operand

Topic	*Key Points*	*Visual Ref.*
forming the negative of a number	★ Forming the negative of a number is accomplished by complementing and incrementing the number. ● When complementing a number, all 0's are changed to 1's and all 1's to 0's. This can be done by the COMplement instruction. ● Adding one to the number can be done with the INCrement instruction.	53–57
COM & INC (or NEG)	★ Changing a positive number to a negative can be done by either: ● Using a COM and INC instruction. ● Using the NEGate instruction. ★ Negative numbers can be converted to positive numbers in the same manner.	
problem-solving	★ The single-operand instructions can be combined with "tally" and/or "loop" techniques to solve a variety of programming problems.	59



read on ➧

Topic | *Key Points* | *Visual Ref.*

an example — 60–71

★ Assume that the problem is to clear out *all* memory locations by using as few instructions as possible.

★ The following 2-word program does just that. It is loaded into the *last* two memory locations.

```
        (R1)  =  0

LOOP:   CLR (R1)+
        BRANCH LOOP (-2)
```

program analysis — 60–71

★ Here is how it works:

- Register R1 is used as a pointer. The CLR instruction clears out the first location indicated by the pointer. In this case, location 0.
- The auto-increment addressing mode then causes the pointer to point to the next *word* location.
- The BRANCH LOOP instruction simply returns the program to the CLR instruction.
- These two instructions cause the program to keep looping as it moves down through memory, clearing locations sequentially.
- When the pointer reaches the location of the CLR instruction, the CLR clears out the instruction itself.
- The BRANCH LOOP causes a branch back to the location formerly holding the CLR but which is now empty. Because all 0's is the op code for a HALT, the program stops.
- The last location, which holds the BRANCH LOOP, can be cleared manually from the console.

read on ➧

Topic | *Key Points* | *Visual Ref.*

multiplication and division — 73–78

★ Binary multiplication by two is accomplished by shifting one place to the left.

★ Binary division by two is accomplished by shifting one place to the right.

★ A shift left (multiply by two) is performed by the ASL instruction.

★ A shift right (divide by two) is performed by the ASR instruction.

an example — 79, 80

★ A simple program to multiply by a power of two can be constructed by loading a register with the number to be multiplied and loading a second register with the exponent of the power of two. For example:

```
          (R1) = 16            number to be
                               multiplied
          (R2) =  5             2^5

START:    ASL R1               multiply by a
                               power of 2
          DEC R2               keep count of
                               multiplications
          BR IF NOT 0 START    If not done, go back
                               and do it again.

          HALT                 Otherwise, stop.
```

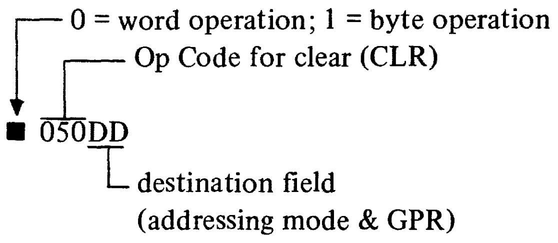
rotates — 81–86

★ Rotates differ from shifts in that the bit shifted out of the word is not lost but is rotated around to enter the other side of the word. In effect, a rotate forms a circular buffer.

★ During all rotates, the C-bit of the PSW is used as a link between bit 15 and bit 0 of the data *word.* If a *byte* is rotated, the C-bit forms a buffer between the MSB and LSB of the selected byte.

read on ➧

Topic	*Key Points*	*Visual Ref.*
ROL **ROR**	★ There are two rotate instructions: ROL (rotate left) and ROR (rotate right). As in other single-operand instructions, they can operate on bytes. The byte mnemonics are ROLB and RORB.	
swapping bytes (SWAB)	★ The swap byte or SWAB instruction reverses the high and low bytes of the selected word.	89, 90
testing (TST)	★ The TST (test) instruction tests the operand to determine if it is either a negative number or a zero. Depending on the result, the TST instruction sets the N bit or the Z bit.	91
instruction card	★ All single-operand instructions are listed on the PDP-11 instruction card. ● A black square on the card indicates that the associated bit may be a one or a zero, enabling the programmer to select a word or a byte operation. ● The next three octal digits indicate the "op" code for that instruction. ● The two D's indicate the two portions of the destination field . . . addressing mode and selected register. The DD indicates the programmer may use any mode and any register he desires. ● The 3-letter mnemonic is used by the assembler. It is in the form: CLR(B) which means that CLR is the form for a word instruction and CLRB is the form for a byte instruction.	93–97



double-operand

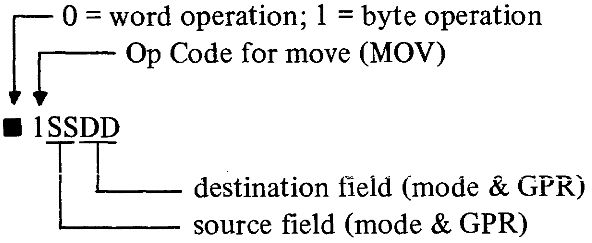
Topic — *Key Points* — *Visual Ref.*

double operand instructions

101–106

★ The format of a double-operand instruction is similar to that of a single-operand instruction except that it has *two* fields for locating operands.

- One field is called the "source" field and the other is called the "destination" field.
- Each field is further divided into "addressing mode" and "selected register."
- Each field is completely independent. The mode and register used by one field may be completely different than the mode and register used by another field.


0 = word operation; 1 = byte operation
Op Code for move (MOV)
■ 1SSDD
destination field (mode & GPR)
source field (mode & GPR)

format

★ The double-operand format is as follows:

- Bit 15 – indicates word or byte operation *except* when used with op code 6. Then it indicates an ADD or SUBtract instruction.
- Bits 14–12 – indicate the "op" code which specifies the job or operation to be done.
- Bits 11–6 – consist of a 3-bit addressing mode field and a 3-bit general register field. These two fields are referred to as the SOURCE field.
- Bits 5–0 – consist of a 3-bit addressing mode field and a 3-bit general register field. These two fields are referred to as the DESTINATION field.

read on ➧

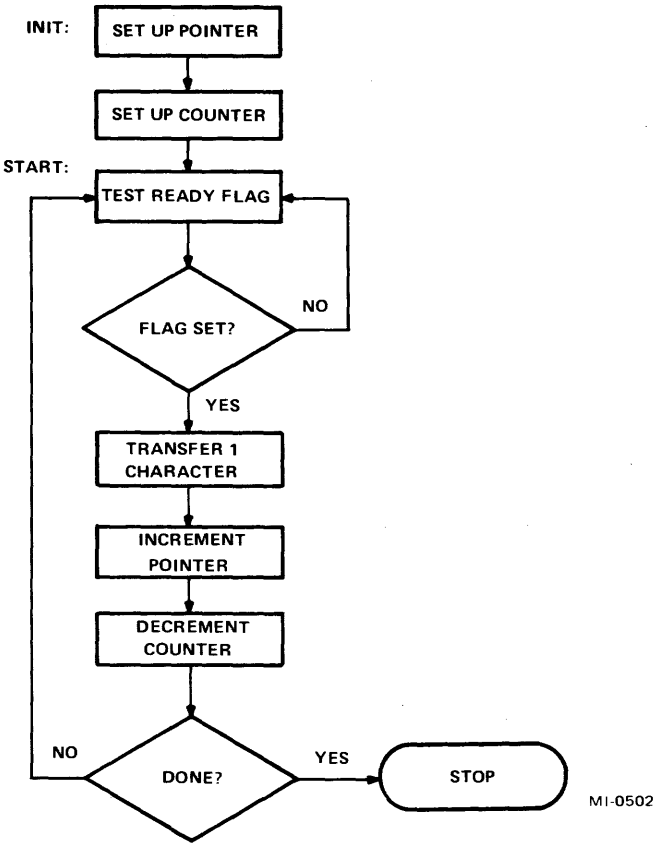
double-operand

Topic	*Key Points*	*Visual Ref.*
byte instructions	★ Byte instructions are possible by setting bit 15. Thus, when bit 15 is 0, the move instruction mnemonic would be MOV; when bit 15 is set, the mnemonic would be MOVB. Remember that op code 6 is different. There are no byte operations for ADD and SUB.	107–109
MOV instruction	★ The MOV (move) instruction moves data from the location specified by the *source* field to the location specified by the *destination* field.	111
CMP instruction	★ The CMP (compare) instruction compares the two operands (source and destination) and tells which one is larger or if they are both equal by setting the appropriate condition code bits in the processor status word.	112
ADD instruction	★ The ADD instruction adds the source operand to the destination operand. The result is stored in the destination.	113
SUB instruction	★ The SUB (subtract) instruction subtracts the source *from* the destination and stores the result in the destination.	113
instruction execution	★ Except for CMP, the *results* of *all* of the above double-operand instructions are always stored in the destination location. We will discuss other double-operand instructions later.	114
	★ A double-operand instruction can transfer data between any *two* Unibus devices. ● For example, data can be MOVed from a GPR to an I/O device using a MOV R0, (R1) instruction. ● Or the contents of one memory location can be ADDed to the contents of another memory location using a ADD (R0), (R1) instruction.	117–121

read on ▶

Topic	*Key Points*	*Visual Ref.*
problem-solving	★ The double-operand instructions can be used with the techniques covered so far to solve a variety of program problems.	122
the problem	★ Assume that the problem is to have a portion of the payroll program printed out for review by the supervisor. It is known that 76 characters (bytes) are to be printed and the block starts at address 600.	123, 124
the program	★ The following program solves the problem:	125–144

```
INIT:     MOV #600, R0
          MOV #76, R1
START:    BITB #200, @#CSR
          BPL START
          MOVB (R0)+, @#777 566
          DEC R1
          BNE START
          HALT
```



read on

Topic *Key Points* *Visual Ref.*

program analysis

★ Here is how the program functions:

125–145

- MOV is the instruction normally used to set up the initial conditions. Here, the first MOV places the starting address (600) into R0 which will be used as a *pointer.* The second MOV sets up R1 as a *counter* by loading the desired number of locations (76) to be printed.

- The BITB instruction tests the *ready flag* in the printer status register (CSR). If the ready flag is set, (i.e., bit position 7=1), the BITB instruction sets the N condition code bit.

- The BPL (branch if plus) instruction keeps the CPU in a wait loop until the ready flag is set (for additional information, see topic "testing I/O device for ready").

- The MOVB instruction moves a byte of data to the printer for printing. The data comes from the location specified by R0. The pointer R0 is then incremented to point to the next sequential location. The *address* of the printer's buffer register is 777 566.

- The counter (R1) is then decremented to indicate one byte has been transferred.

- The program then checks for done with the BNE instruction. If the counter has not reached zero, indicating more transfers must take place, then the BNE causes a branch back to START and the program continues.

- When the counter (R1) reaches zero, indicating all data has been transferred, the branch does not occur and the program executes the next instruction, HALT.

read on ▶

Topic	*Key Points*	*Visual Ref.*
testing I/O device for "ready"	★ I/O devices are not always available to the program. When a device is available, it sets a "ready" flag.	141b
	★ The "ready" flag is bit 7 of the I/O status register (CSR). This bit can be tested by the program.	141c
	★ One method of testing for ready is to use the BITB (bit test) instruction.	141d–141g
	● The BITB instruction performs a logical AND between selected bits in a known and unknown byte.	
	● For instance, testing bit 3 could be done by loading a 1 in bit 3 of the known byte. The BITB instruction would AND the two bits (bit 3 in both the known and unknown bytes). If they were both set ($1 \wedge 1 = 1$), the result would be 1, clearing the Z bit. If the unknown bit were 0, however, then $1 \wedge 0 = 0$, and the Z bit would be set.	
	● In testing the ready flag, the low byte of the status register is used. Therefore, bit 7 is interpreted as the sign bit. We set bit 7 in the known byte (source) and follow it with all zeros because we don't care about any other bits.	
	● The BITB instruction ANDs the known byte (source) with the unknown byte (destination or status register). If the ready flag is set (bit 7 set), BITB produces a one which sets the N condition code because bit 7 denotes a *negative* number.	
	● While waiting for a device to be ready, a loop must be made. In other words, test for ready and if the device is not ready, branch back and test again.	

(continued on next page)

read on ➧

Topic	*Key Points*	*Visual Ref.*
testing I/O device for "ready" (Cont.)	● The ready loop is composed of two instructions: BITB #200, @#CSR and BPL START (branch to START if N=0). **NOTE** CSR represents the status register in the printer. Bit 7 in this status register is the ready flag.	141d – 141g
moving data from an I/O to memory	★ Assume now that our problem is to move data *from* the I/O device (source) to memory (destination). ● There is a danger that continual storing of data into memory could cause data to be moved into an area of memory containing the payroll program, thereby wiping it out.	155–157
preventing erasure of the payroll program	★ One method of preventing the payroll program from being destroyed is to set a limit at which point the data transfer program either stops or takes appropriate action. ● The limit can be set up by specifying the address of the last location where data can be safely stored. ● The destination pointer is continually moving toward this limit as data is loaded into memory. ● A CMP (compare) instruction is used to continually compare the address pointer with the limit address to determine if the limit is reached.	158–166

(continued on next page)

read on ➧

Topic	*Key Points*	*Visual Ref.*
preventing erasure of the payroll program (Cont.)	● When the CMP instruction indicates that the *limit* has been reached, some action must be taken to prevent destruction of the payroll program. One method would be to HALT the program. However, this means that the balance of the payroll data would not be loaded into memory.	
	● A better method would be to relocate our payroll program to make room for additional payroll data.	
relocating a program	★ We want to move the entire payroll program from its present location to a new area of memory. We know the starting and final address of the payroll program (1102 and 1776) and the starting address (3000) of the new memory area.	167, 168
	★ These steps are used to relocate the program:	169–182
	● We initialize by using two MOV instructions to set up source and destination pointers. The source pointer points to the first payroll program address; the destination pointer points to the first new memory address. MOV #1102, R2 MOV #3000, R3	
	● We then use a MOV (R2)+, (R3)+ instruction to transfer one word of the program to the new memory area as specified by the pointers. Both pointers are autoincremented so they move to the next locations.	
	● A loop is constructed so that the pointers keep moving down through memory as the payroll program is transferred, instruction by instruction, to the new area of memory.	

(continued on next page)

read on ➧

Topic *Key Points* *Visual Ref.*

relocating a program (Cont.)

- The payroll program pointer (R2) is constantly compared with a limit, which is the last address (1776) of the program. If the pointer is lower or the same as the limit, a BLOS START instruction causes the CPU to branch to START and move another word. The CPU will remain in the program loop until R2 is autoincremented to 2000 (this will ensure that the last word is moved out of location 1776).

NOTE

The program that we moved must be written in Position Independent Code (PIC); otherwise, it will not run once it has been moved to another area of memory. For a brief description of PIC, refer to Chapter 5 of your PDP-11 Processor Handbook. Also, refer to the workbook on "Addressing Modes."

program

★ The following program will transfer the payroll program from its present location to a new area of memory:

181

```
label     instruction         comments

INIT:     MOV #1102, R2       ; set up
          MOV #3000, R3       ; pointers.
START:    MOV (R2)+, (R3)+    ; move one word
LIMIT:    CMP R2, #1776       ; if not done
          BLOS START          ; go to start.
          HALT                ; if done stop
```

instruction summary

★ Double-operand instructions are powerful because they deal with two operands . . . source and destination.

185–189

- The MOV instruction is particularly effective in the INITIALIZATION portion for setting up pointers, counters and limit values.
- The CMP instruction is useful in comparing pointers with limit values to determine when to exit from a program loop.

read on ➧

Topic	*Key Points*	*Visual Ref.*
instruction summary (Cont.)	● The ADD and SUB instructions have the power of dealing with two numbers simultaneously. ● The BIT instruction permits comparison of a known bit pattern and an unknown bit pattern, such as we did when testing for a ready flag.	
the three BIT instructions	★ There are *three* related *bit* instructions: ● BIT (bit test) – which tests an unknown bit to determine if the bit is a 0 or a 1. BIT performs this test by performing a logical AND function. ● BIS (bit test) – which performs an inclusive OR function. In other words, for any bit *set* in the source operand, the corresponding bit is set in the destination operand. ● BIC (bit clear) – which clears specific bits in the destination. When BIC is used, each bit *set* in the source is *cleared* in the destination.	190–192
masking	★ The BIC instruction can "mask" portions of a data word. Only the bits specified in the source are "masked" out.	193

For example:

DST = 012 345
SRC = 177 770

Then the instruction:

BIC SRC, DST

will "mask out" the first five octal digits, leaving the value 000 005 in the DST.

Doing it digit by digit:

$$\begin{array}{lll} \text{DST} & = & 0\ 001\ 010\ 011\ 100\ 101 \\ \text{SRC} & = & \underbrace{1\ 111\ 111\ 111\ 111}_{\text{Mask}}\ 000 \\ \text{RESULT} & = & 0\ 000\ 000\ 000\ 000\ 101 \end{array}$$

condition codes

Topic	*Key Points*	*Visual Ref.*
condition codes	★ There are four condition code bits: ● N – indicating a negative condition ● Z – indicating a zero condition ● V – indicating an overflow condition ● C – indicating a carry condition ★ These four bits are part of the processor status word (PSW). ★ The result of *any* single-operand or double-operand instruction affects one or more of the four condition code bits. ★ A new set of condition codes is created after execution of *each* instruction. ★ The CPU may be asked to check the condition codes after execution of an instruction. ★ The condition codes are used by the various branch instructions to determine whether or not the program is to branch.	197–204
Z-bit	★ Whenever the CPU sees that the result of an instruction is *zero*, it *sets* the Z bit. If the result if *not zero,* then it *clears* the Z bit. ★ There are a number of ways of obtaining a zero result. For example: ● Adding two numbers equal in magnitude but different in sign. ● Comparing two numbers of equal value. ● Using the CLR instruction.	205–208

read on ➧

Topic	*Key Points*	*Visual Ref.*
N-bit	★ In this case, the CPU only looks at the most significant bit (MSB) of the result. • If the MSB is a 1, indicating a negative value, the CPU *sets* the N-bit. • If the MSB is a 0, indicating a positive value, then the CPU *clears* the N-bit.	209–211
C-bit	★ The CPU *sets* the C-bit when the result of an instruction has caused a *carry out* of the most significant bit of the result. ★ When the instruction results in a carry out of the most significant bit of the result, the *carry itself* is usually moved into the C-bit. Otherwise, the C-bit is cleared. ★ A carry of 1 *sets* the C-bit while a carry of 0 *clears* the C-bit. However, there are exceptions. For example: • SUB and CMP *set* the C-bit when there is *no* carry. • INC and DEC do not affect the C-bit. ★ During rotate instructions (ROL and ROR), the C-bit forms a "buffer" between the most significant bit and the least significant bit of the word.	212–216
V-bit	★ The V-bit is set to indicate that an overflow condition exists. An overflow condition occurs when an arithmetic operation produces a result that "spills over" into the MSB (sign position). There are two methods the hardware uses to check for an overflow condition. ★ One way is for the CPU to test for a *change of sign.* • When using single-operand instructions, such as INC, DEC, or NEG, a change of sign indicates an overflow condition.	217–233

(continued on next page)

read on ➧

Topic	*Key Points*	*Visual Ref.*
V-bit (Cont.)	● When using double-operand instructions, such as ADD, SUB, or CMP, in which both the source and destination have *like* signs, a change of sign in the result indicates an overflow condition. ★ Another method used by the CPU is to test the N-bit and C-bit when dealing with *shift* and *rotate* instructions. ● If the N-bit is set, an overflow exists. ● If the C-bit is set, an overflow exists. ● If *both* the N and C bits are set, there is *no* overflow condition.	217–233
multiple codes	★ More than one condition code can be set by a particular instruction. For example, both a carry and overflow condition may exist after instruction execution.	234
instruction card	★ The instruction card indicates which codes are affected by each specific instruction. ● – Indicates that the corresponding condition code bit is *never affected.* ● 0 Indicates that the corresponding condition code bit is *always cleared.* ● 1 Indicates that the corresponding condition code bit is *always set.* ● * Indicates that the corresponding condition code bit is either *set* or *cleared* depending on the results of the instruction.	235–237

read on ➧

condition code operators

Topic — *Key Points* — *Visual Ref.*

changing the condition codes

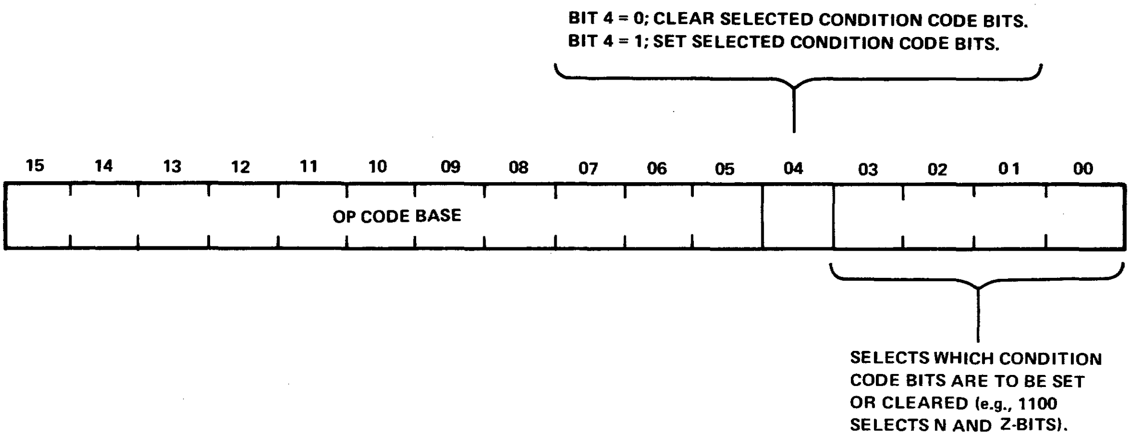
★ There are two ways of changing the condition code bits in the processor status word (PSW). — 238

- The result of a single-operand or double-operand instruction will *automatically* change appropriate condition code bits.
- A separate set of condition code instructions, or operators, allows the program to control the states of the condition code bits.

instruction format

★ The format of the condition code operators is as follows: — 239–242

- Bits 15–5 the "op" code base = 000 240
- Bit 4 the "operator" which indicates the job to be done. If set, any selected bit is set; if clear, any selected bit is cleared.
- Bits 3–0 the "select" field. Each of these bits corresponds to one of the four condition code bits. When one of these bits is set, then the corresponding condition code bit is set or cleared depending on the state of the "operator" (bit 4).


BIT 4 = 0; CLEAR SELECTED CONDITION CODE BITS.
BIT 4 = 1; SET SELECTED CONDITION CODE BITS.
15
14
13
12
11
10
09
08
07
06
05
04
03
02
01
00
OP CODE BASE
SELECTS WHICH CONDITION CODE BITS ARE TO BE SET OR CLEARED (e.g., 1100 SELECTS N AND Z-BITS).

MI-0501

read on ➧

condition code operators

Topic — *Key Points* — *Visual Ref.*

instruction mnemonics — 243–248

★ Mnemonics have been assigned for certain states of the "operator" and "select field."

● CLN	–	clear N bit
● CLZ	–	clear Z bit
● CLV	–	clear V bit
● CLC	–	clear C bit
● CCC	–	clear all four bits
● SEN	–	set N bit
● SEZ	–	set Z bit
● SEV	–	set V bit
● SEC	–	set C bit
● SCC	–	set all four bits

other combinations — 249

★ In addition to the instructions that have mnemonics assigned, there are other instructions available. You must remember two things:

- It is possible to operate on any combination of bits . . . such as two or three bits. For example, an instruction could set the Z and C bits.
- When using these combinations, no mnemonics exist, so it is necessary to use the octal op code for the instruction.

read on ➧

condition code operators

Topic	*Key Points*	*Visual Ref.*
no operation	★ If no bit has been selected for use, then it doesn't matter whether or not the "operator" specifies a set or clear operation. Nothing will happen. Therefore: ● 000 240 is a NOP (no operation) instruction. ● 000 260 is a NOP.	250
summary	★ There are four *set* instructions to set any one of the four condition code bits. ★ There are four *clear* instructions to clear any one of the four condition code bits. ★ There is one instruction to clear *all* bits. ★ There is one instruction to set *all* bits. ★ There are a number of instructions, without mnemonics, for setting or clearing various combinations of bits. ★ There are two NOP instructions.	251

read on ➧

branch

Topic	*Key Points*	*Visual Ref.*
branch instructions	★ Any branch instruction orders the CPU to test a condition and take action based on the results of the test. ● The conditions tested are the states (set or cleared) of the four condition code bits in the PSW . . . . . N, Z, V, and C. ● There are 16 conditional branch instructions. They are "conditional" because they branch only if the specified condition is met. ● There is one unconditional branch (BR). This instruction does not test a condition. It causes a branch whenever used.	255–257
format	★ The high byte of the instruction is an op code specifying the condition to be tested. ★ The low byte of the instruction is the "offset" value that specifies *where* the CPU is to branch for its next instruction (if the condition is satisfied).	258–261
typical branch	★ A typical branch instruction is BNE . . . . Branch if Not Equal to zero. ● The BNE instruction tests the condition of the Z bit in the PSW. ● If the Z-bit is *clear*, indicating "not equal to zero", then a branch occurs. ● If the Z-bit is *set,* indicating "equal to zero", then a branch *does not occur.* The program simply executes the next sequential instruction.	262, 263

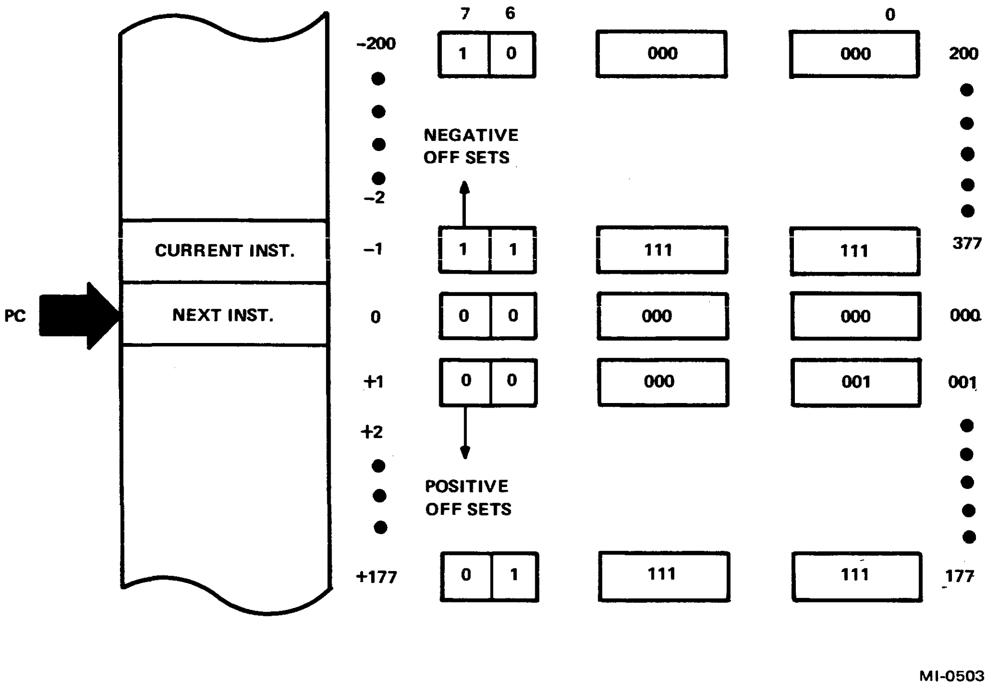
read on ➧

Topic	*Key Points*	*Visual Ref.*
using branch instructions	★ Assume that it is necessary to find the value "Y" located somewhere in a large table of values.	265
analyzing the problem	★ The problem of finding "Y" can be solved by the following steps: ● Compare an entry from the table with the known value "Y." ● Test the Z bit. If it is zero, there is no match so repeat the process. ● If the Z bit is set, it indicates a match so we know we have found the desired value "Y."	266
implementing the solution	★ A program can be implemented as follows: ● Set up a pointer to point to the first address in the table of values. This is done by MOV #ADRS, R0. ● Compare the entry with the known value "Y" and then move the pointer to the next location. This is done by a CMPB #Y, (R0)+ instruction. ● Test the result of the compare by branching if there is no match. This is done by BNE START. Thus, if we haven't found "Y," the program compares the next entry in the table. ● Once "Y" is found, the comparison results in setting the Z bit, the branch does not take place, and the program executes the next instruction. INIT: MOV #ADRS, R0 START: CMPB #Y, (R0)+ BNE START	267–273

read on ➧

Topic	*Key Points*	*Visual Ref.*
offset	★ Offset is another word for displacement. It represents the distance between the point we wish to branch to and the current address in the PC (the offset is always calculated *relative* to the updated PC). ★ The offset represents the distance in *words*.	275–277
direction	★ The offset can be used in either direction. ● A *positive* offset indicates the branch point is forward from the PC. ● A *negative* offset indicates the branch point is backward from the PC.	278
range	★ The offset range is limited. ● The maximum positive offset is 177 octal words. ● The maximum negative offset is 200 octal words.	279
location	★ The offset is contained in the low byte of a branch instruction. ● Bit 7 is the sign of the offset (0 = positive offset; 1 = negative offset). ● Bits 0–6 indicate the magnitude.	280

read on ➧



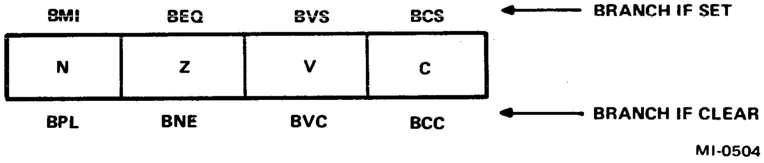
Topic	*Key Points*	*Visual Ref.*
using offsets	★ When dealing with positive offsets, simply place a 0 in the sign and the appropriate number of words in the magnitude.	281–291
	★ When dealing with negative offsets, write the number of words desired. Then complement the entire offset, including sign. Finally, increment this value to obtain the two's complement. This step is not necessary when using assembler syntax. For example, just write BNE TAG (TAG is a *label* that identifies the location you wish to branch to).	
	★ Two offsets that may cause problems are:	292
	● -1 – causes a branch back to the instruction causing the branch. Therefore, a continual loop is formed.	
	● 0 – indicates that there is no branch at all.	
effective address	★ The address of the branch destination is called the "effective address."	293

read on ➧

Topic	*Key Points*	*Visual Ref.*
calculating the effective address	★ The assembler takes the offset value, multiplies it by two, and adds the result to the program counter. The effective address can be found by either the programmer or the assembler by using the formula: **(offset X 2) + (updated PC) = effective address**	294–298
calculating the offset value	★ The first step is to calculate the *octal difference* between the PC and the effective address. ★ The second step is to divide this difference by two because the PC is always incremented by two. ★ Thus, the formula for finding the offset value is: $\frac{\textbf{effective address – updated program counter}}{2}$	299–302
some branch instructions	★ Testing the Z condition code bit: ● BEQ – tests for a zero condition which exists when the Z-bit is *set.* ● BNE – tests for a non-zero condition which exists when the Z-bit is *clear.*	304
	★ Testing the C condition code bit: ● BCS – branch if C-bit is set. ● BCC – branch if C-bit is clear.	305
	★ Testing the V condition code bit: ● BVS – branch if V-bit set. ● BVC – branch if V-bit clear.	306
	★ Testing the N condition code bit: ● BMI – branch if N-bit set, which indicates a negative or minus. ● BPL – branch if N-bit clear, which indicates a positive or plus.	307

read on ➧

Topic	*Key Points*	*Visual Ref.*
summary	★ There are eight separate branch instructions dedicated to testing the four condition code bits of the PSW.	308



Topic	*Key Points*	*Visual Ref.*
using branch instructions	★ Because branch instructions are some of the most versatile instructions available, we are going to show their use by constructing a rather large program.	309
the problem	★ A long series of numbers is stored in a random order in consecutive bytes in memory. To use these values more efficiently, it is desired to place all of the numbers in ascending order.	310
placing numbers in order	★ If, for example, the job was to sort three numbers into sequential order, the following steps would be necessary: ● Compare the first and second numbers. ● If the *second* number is *not* larger, switch the two numbers around. ● Compare the number now in the second position with the third number. ● If the third number is *not* larger than the second, switch the two numbers around. ● Compare the first and second numbers again. This is necessary because the previous step could have resulted in a new second number.	311–319

(continued on next page)

read on ➧

Topic	*Key Points*	*Visual Ref.*
placing numbers in order (Cont.)	• Again, if the second number is not larger, switch them around.	311–319
	★ At this point, the three numbers are now in proper order. Notice that any subsequent comparisons do not require any further switching of numbers.	
comparison of numbers	★ When comparing two numbers, we ask, "Is the second number larger?" • If the second number is larger, we can continue with the program. • If the second number is *not* larger, we must switch them around.	320
	★ The CMPB (compare byte) instruction will compare the two numbers. The first number is the SOURCE operand and the second number is the DESTINATION operand. There are three possible results: • Zero – indicating both numbers are equal. • Positive – indicating the source is larger. • Negative – indicating that the destination is larger.	321
	★ Because we want to know when the second number (destination) is larger, we are looking for a minus result so we use the branch instruction BMI – branch if minus. NOTE There are a number of branch instructions available in the PDP-11 instruction set and there might be a better choice for this particular job. However, we are not going to cover these additional branch instructions until later in this study unit.	322

(continued on next page)

read on ➧

Topic *Key Points* *Visual Ref.*

comparison of numbers (Cont.)

★ A flowchart analysis of this problem follows.

flowchart analysis:

325–336

NOTE

Refer to the flowchart on the facing page.

placing numbers in order

★ After the initial conditions are set up, the first two numbers are compared.

★ The pointers are then incremented so that they are pointing to the next two numbers in the list.

is 2nd number larger?

★ Next, the results of the comparison are tested to determine if the numbers need to be switched.

- If the second number is *larger,* no switch is necessary.
- If the second number is *not* larger, the two numbers must be switched around. A *flag* is then *set* to indicate there was a switch.

check limit

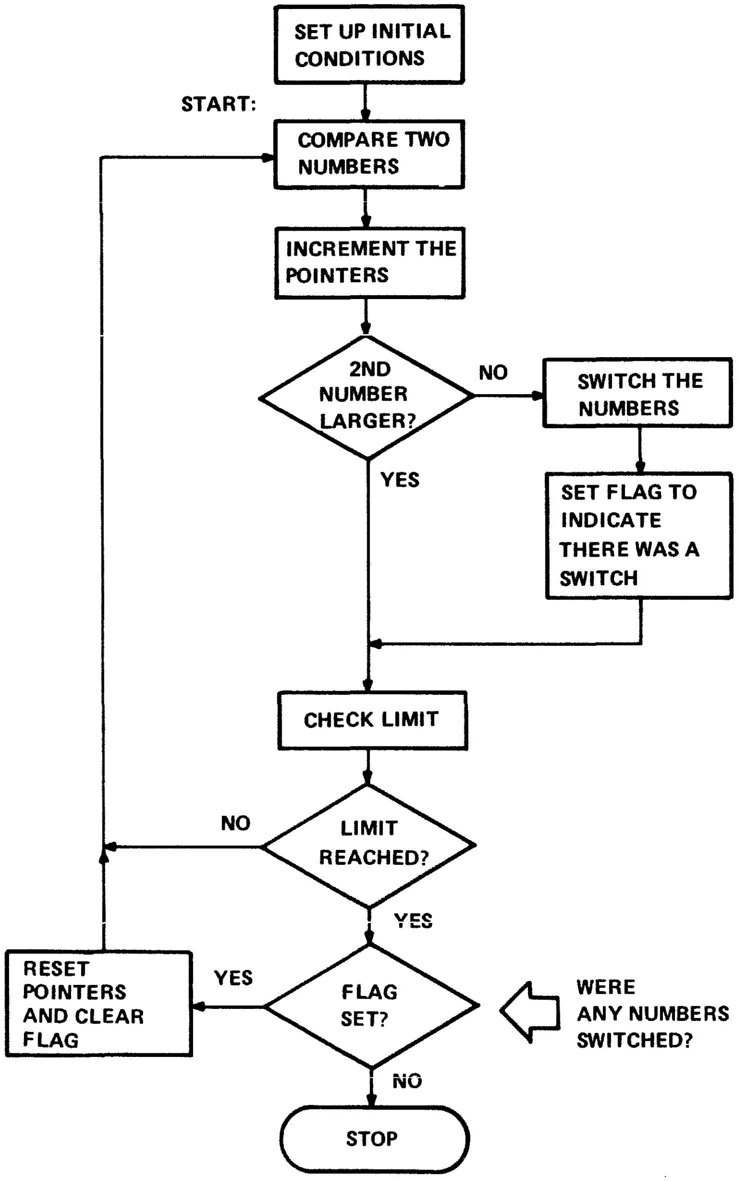
★ After two numbers have been compared and switched (if necessary), the limit is checked to determine if there are any more numbers in the list.

- If there are more numbers (i.e., if the limit is *not* reached), the CPU branches back to START and compares the next two numbers. The CPU remains in the loop until all numbers in the list have been compared.
- When the limit is reached, the CPU exits from the loop and goes to the next step to determine if the flag is set.

(continued on page 36)

read on ➧

FLOWCHART FOR SORTING A LIST OF NUMBERS INTO THEIR PROPER ORDER



MI-0500

read on ➧

Topic	*Key Points*	*Visual Ref.*
is flag set?	★ The flag tells us if we are done. ● If the flag is *set,* it indicates that two numbers were switched around. Any number switching could affect the order of other numbers. Therefore, it is necessary to *reset* the pointers to their initial values, *clear* the flag, and then repeat the entire process until there are no further switches. ● If the flag is *not* set, *all* of the numbers are now in their proper sequence. Therefore, we can stop our program.	
program divisions	★ The program to sort a series of numbers is divided into six major divisions: ● Initialization ● Comparing numbers ● Switching numbers, if necessary ● Checking the limit to see if all numbers have been handled ● Checking for done (is flag = 0?) to see if program has sorted all numbers ● Resetting pointers and recycling through entire program if not done (flag = 1).	336

read on ➧

Topic *Key Points* *Visual Ref.*

initialization

★ There are four initial conditions that must be taken care of. In the first three cases, the *immediate* addressing mode is used.

339, 340

- Set the limit – MOV #604, R0
- Set left pointer – MOV #600, R1
- Set right pointer – MOV #601, R2
- Reset flag to zero – CLR R4

comparing numbers

★ When comparing numbers, it is necessary to compare the numbers, increment both pointers, and then see if the second number is larger.

341–343

- Comparing numbers and incrementing pointers is done with the instruction, CMPB (R1)+, (R2)+.
- Checking to see if the second number is larger is done with BMI CHECK. If the second number is larger, the program branches to the next functional unit of checking the limit.
- However, if the numbers must be switched, the branch does not occur, and the program goes to the next sequential functional block which is: switching the numbers.

read on ➧

Topic	*Key Points*	*Visual Ref.*
switching numbers	★ In order to switch two numbers, the program performs the following steps. Notice that the first number is the *source* and the second number is the *destination.* ● Save the destination by moving it into a GPR for temporary storage. ● Move the source into the destination location. ● Move the former destination from the GPR into the source location. ● At this point, the numbers have been switched around.	344–348
pointers can cause trouble	★ After the first compare of the first and second numbers, the pointers are pointing to the second and third numbers. It is necessary to get the pointers back to the first and second numbers if a switch is needed. ● Since the left pointer (R1) is pointing to the second number at this time, the destination can be saved by using the instruction: MOVB (R1), R3. ● Now, by using auto-decrement addressing, both pointers can be moved back and the source moved into the destination. We use, MOVB -(R1), -(R2) to do this. ● The number held in the GPR is now moved into the source by the instruction, MOVB R3, (R1)+. This instruction restores the left pointer. ● The right pointer is then restored by an INC R2 instruction. ● A flag is then set to indicate a switch was made. Bit 0 in R4 is used as our flag. The instruction MOV #000 001, R4 sets the flag (bit 0) to a 1.	349–359

read on ➧

Topic	*Key Points*	*Visual Ref.*
checking the limit	★ The limit is checked to see if all numbers in the table have been handled by the program. ● Because the right pointer will reach the limit before the left pointer, it is used to check for the limit. ● The contents of the right pointer are compared with the limit by the instruction CMP R0, R2. ● The compare instruction is followed by a BNE START instruction. ● If the results of the compare are not equal, a branch is made back to the start of the program to handle the next set of numbers. ● However, if the compare indicates that the right pointer and the limit value are equal, we know that all numbers have been handled, so we go to the next program block to check for done.	360–370
checking for done	★ Whenever the flag is set (bit zero in R4), it indicates a switch was made and, therefore, the program is not yet finished. ● The instruction BIT #000 001, R4 is used to test the state of the flag. ● If the flag is clear, indicating no switches were made, then the CPU fetches the HALT instruction and stops. ● If the flag is set, indicating a switch was made, the BNE INIT instruction causes a branch back to the beginning of the program.	371–377

read on ➧

Topic	*Key Points*	*Visual Ref.*
resetting pointers and code	★ Because we are going through the entire program again, it is necessary to reset the pointers and clear the flag (R4) which is actually the same thing as re-initializing the program. ● We can reset the pointers and flag by simply branching back to the initialize portion of the program. ● However, because we know that our limit value will not change, we branch back to the *second* instruction in the initialize portion.	375
sorting many numbers	★ The program can be modified to handle any amount of numbers by making only three changes in the initialize portion of the program. ● MOV #X, R0 – in place of X, use the last address of your list of numbers. ● MOV #Y, R1 – in place of Y, use the starting address of your list. ● MOV #Y+1, R2 – in place of Y+1, use the starting address plus one.	385

read on ➧

Topic	*Key Points*	*Visual Ref.*
not just numbers	★ This program can also be used to sort letters of the alphabet because each letter can be represented by an 8-bit ASCII numerical representation.	386
the finished program	★ The following program will sort a list of numbers and/or letters of the alphabet into their proper sequence.	

```
label      instruction           comments field
           MOV #604, R0           ; set up limit.
INIT:      MOV #600, R1           ; set up both
           MOV #601, R2           ; pointers.
           CLR R4                 ; clear flag.
START:     CMPB (R1)+, (R2)+      ; Compare two
           BMI CHECK              ; numbers.
SWITCH:    MOVB (R1), R3          ; Switch numbers
           MOVB -(R1), -(R2)      ; if second is
           MOVB R3, (R1)+         ; not larger and
           INC R2                 ; restore pointers.
           MOV #000 001, R4       ; Set flag.
CHECK:     CMP R0, R2             ; If limit not
           BNE START              ; reached go back.
RESET:     BIT #000 001, R4       ; Test flag.
           BNE INIT               ; Repeat if flag is 1.
DONE:      HALT                   ; Otherwise stop.
```

read on ➧

Topic	*Key Points*	*Visual Ref.*
eight branch instructions	★ Remember that there are eight branch instructions for testing the states of each of the condition code bits (N, Z, V, C). ● Four instructions test the set state: BMI, BEQ, BVS, and BCS. ● Four instructions test the clear state: BPL, BNE, BVC, and BCC.	388
other branch instructions	★ There are eight more branch instructions that test combinations of bits. However, before describing them, it is necessary to understand signed and unsigned numbers.	389, 390
signed numbers	★ When dealing with both the sign (+ or −) and magnitude of a number, the number is called a "signed" number.	390
unsigned numbers	★ A number in which the sign bit (most significant bit) is considered to be part of the magnitude is known as an "unsigned" number.	390

read on ➧

Topic	*Key Points*	*Visual Ref.*
using the CMP instruction	★ When comparing two numbers, the N bit is set if A is less than B, provided we are dealing with *unsigned* numbers. ★ When comparing two numbers, the V bit is set if A is less than B, provided we are dealing with *signed* numbers. ★ Therefore, when dealing with either signed or unsigned numbers, either the N bit or the V bit is set whenever A is *less than* B. ★ The above conditions are handled by a single branch instruction . . . BLT . . . branch if less than.	391–399
eight more branch instructions	★ There are a total of eight branch instructions that test combinations of condition code bits. ● BLT – branch if less than ● BLE – branch if less than, or equal ● BGT – branch if greater than ● BGE – branch if greater than, or equal ● BLO – branch if lower ● BLOS – branch if lower, or same ● BHI – branch if higher ● BHIS – branch if higher, or same ★ Two of the above instructions are not actually new instructions but simply new mnemonics for two previous instructions. ● BLO and BCS have the same op code (103 400). ● BHIS and BCC have the same op code (103 000).	400–406

read on ➧

subroutine

Topic	*Key Points*	*Visual Ref.*
subroutines	★ A subroutine is used when one task must be performed at different points in the program. An example of such steps would be a series of PRINT instructions. ● The program requests or "calls" the subroutine whenever it needs to use it. ● Once the subroutine is finished, it returns control to the main program. ● Typical subroutines are: multiply and divide; calculating trigonometric functions; and input/output functions such as print. ● Subroutines may use other subroutines. For instance, a square root subroutine may use a divide subroutine. ● Subroutines can be shared by different programs. For example, a print subroutine might be used by an I/O service program, the main program, and an error handling program.	410–414
two problems with subroutines	★ The computer has two distinct problems when dealing with subroutines: ● The main program must tell the CPU where the subroutine is located. ● The subroutine must return control to the proper point in the main program so that the program can continue where it had left off.	415
	★ These two problems are solved by two PDP-11 instructions: ● JSR – jump to subroutine ● RTS – return from subroutine	416

read on ➧

Topic	*Key Points*	*Visual Ref.*
JSR instruction	★ The JSR (jump to subroutine) instruction causes the program to jump to the subroutine and also constructs a "linkage pointer" so that the main program can continue where it left off once the subroutine has been executed.	417–419
JSR format	★ The format of the JSR is as follows: ● Bits 9–15 – these are always octal 004 indicating the op code for a JSR. ● Bits 6–8 – specify the "link" register. Any GPR may be used as the link, except R6. ● Bits 0–5 – a "destination" field that consists of addressing mode and general register fields. This specifies the starting address of the subroutine.	420, 421
	★ Register R7 (the PC) is frequently used for both the link and the destination. For example, you may use JSR R7, SUBR, which is coded 004 767. R7 is the *only* register that can be used for both the link and destination; the other GPRs cannot. Thus, if the link is R5, any register *except R5* can be used for the destination field.	422
JSR operation	★ The JSR instruction is executed in a number of steps: ● When executing a JSR, the CPU first uses the destination field to find the subroutine starting address, which is then held in a temporary register. ● It next saves the current contents of the register to be used as the link by pushing the contents onto the hardware stack.	424–434

(continued on next page)

read on ➧

Topic | *Key Points* | *Visual Ref.*

JSR operation (Cont.)

424–434

- The CPU then saves the main program PC by loading it into the link.
- The final step is to load the subroutine starting address, currently held in the temporary register, into the PC. In effect, it loads a new PC.

typical JSR

★ A typical JSR instruction is:

JSR R5, (R3)

This instruction indicates that register R5 will be used as the link and that R3 is used with addressing mode 1 to serve as the destination field.

JSR functions

★ This JSR instruction functions as follows:

- The CPU looks at the destination field and seeing mode 1, knows that the *starting address* of the subroutine is stored in R3.
- The CPU then saves the subroutine starting address by placing it in a temporary register.
- Because R5 is specified as the link, the CPU decrements the stack pointer (R6) and then moves the data currently stored in R5 onto the hardware stack in order to save it.
- The CPU then stores the current PC in R5 which has been selected as the link register.
- Finally, the CPU moves the subroutine starting address from the temporary register and loads it into the PC. At this point, the computer begins execution of the subroutine.

read on ➧

Topic	*Key Points*	*Visual Ref.*
RTS instruction	★ The RTS (return from subroutine) instruction uses the link to return control to the main program once the subroutine is finished.	440
RTS format	★ The format of the RTS is as follows: ● Bits 3–15 – always contain octal 00020 which is the op code for an RTS. ● Bits 0–2 – specify any one of the GPRs. ● WARNING – The register specified by bits 0–2 *must* be the *same* register used as the link by the associated JSR. In other words, to form a link between the JSR causing the jump and the RTS returning control, the same register must be used by both instructions.	441–444
typical RTS	★ A typical RTS instruction is: RTS R5 The prime function of an RTS is to *restore* information. The instruction does the following: ● It first restores the PC by moving the old PC from the link (in this case, R5) and loading it into the program counter (R7). ● It then restores R5 by "popping" the data from the top of the stack and moving it into R5 (the link). ● At this point, everything is just as it was prior to execution of the JSR instruction.	445, 446

read on ➧

subroutine

Topic	*Key Points*	*Visual Ref.*
summary	★ A JSR is used at any desired point in a program to cause a jump to a specific subroutine. ★ Every subroutine *must* end with an RTS in order to return control to the proper point in the main program.	447

read on ➧

interrupts & traps

Topic	*Key Points*	*Visual Ref.*
interrupts & traps	★ There are three methods of leaving a main program: ● Software exit – the *program* specifies a jump to some *subroutine.* ● Trap exit – *internal hardware* or a special instruction forces a jump to an *error* handling routine. ● Interrupt exit – *external hardware* forces a jump to an interrupt *service* routine.	449–452
	★ In all of the above cases, there is a *jump* to another program. Once that program has been executed, control is returned to the proper point in the main program, unless an error condition forces a HALT.	
interrupts	★ An interrupt forms a "link" back to the main program by taking the current PS word and PC, and "pushing" them both onto the stack.	453
	★ The CPU then retrieves a new PS word and PC from the interrupt vector. The new PC points to the start of the interrupt routine.	454, 455
	★ An RTI (return from interrupt) instruction is used to return control to the main program once the interrupt service routine is finished. ● The RTI format is simple. The entire word is an op code of 000 002. ● The RTI restores control to the main program by "popping" the PS and PC from the stack.	456–461

(continued on next page)

read on ➧

Topic	*Key Points*	*Visual Ref.*
interrupts (Cont.)	★ An RTT (return from trap) instruction, which has an op code of 000 006, performs the same function as an RTI. The only difference is that an RTI permits use of the T-bit for debugging programs while the RTT inhibits the T-bit.	456–461
programming interrupts	★ When dealing with interrupts, a programmer *must* perform three duties: ● Write the interrupt service routine. ● End the routine with an RTI or RTT. ● Load the desired PS and PC in the vector.	462
traps	★ The PDP-11 uses either internal hardware or software to find errors. When an error is spotted, the CPU stops the program and issues a *trap*, causing a jump to a handling routine of some type.	463
types of traps	★ HARDWARE trap – a trap caused by an error condition detected by the CPU. It may cause a jump to an error handling routine. However, it is usual to set up the vector so that the first word points to the second word, and the second word contains a HALT instruction (all 0's).	464
	★ SOFTWARE trap – a trap caused by a trap instruction. The instruction causes a jump to a *special* handling routine.	
hardware traps	★ A *hardware* trap is similar to an interrupt except it is caused by *internal* hardware (the CPU) rather than some *external* device.	465–469
	★ When a *hardware* trap occurs, the PS word and the PC are "pushed" onto the stack.	

read on ➧

Topic	*Key Points*	*Visual Ref.*
hardware traps (Cont.)	★ The trap vector then directs the CPU to an error handling routine or, as mentioned before, the first word points to the second word containing a HALT instruction.	
	● The only difference between a hardware trap and an interrupt is that the hardware trap uses a different vector and is caused by internal hardware.	
	● Return to the main program is accomplished by either an RTI or RTT instruction.	
software traps	★ There are four trap instructions. All trap instructions function in the same manner, but each instruction uses its own vector and, therefore, causes the program to jump to a different handling routine.	470–475
	● IOT (op code 000 004) – vector 020 – this trap instruction is used by the I/O executive routine.	
	● BPT (op code 000 003) – vector 014 – this trap instruction (referred to as "breakpoint trap") is used by ODT, a debugging aid.	
	● EMT (any op code between 104 000 and 104 377) – vector 030 – this trap instruction (emulator) is used by DEC software.	
	● TRAP (any op code between 104 400 and 104 777) – vector 034 – this trap instruction is usually reserved for user programs.	

read on ➧

miscellaneous

Topic | *Key Points* | *Visual Ref.*

miscellaneous instructions

★ There are five miscellaneous instructions:

477 – 483

- HALT (000 000) – used to stop the program.

- NOP (000 240 or 000 260) – no operation. This instruction might be used, for example, if an entire program had been coded and then, for some reason, an instruction was deleted. Rather than recode the entire program, a NOP could be used in place of the deleted instruction.

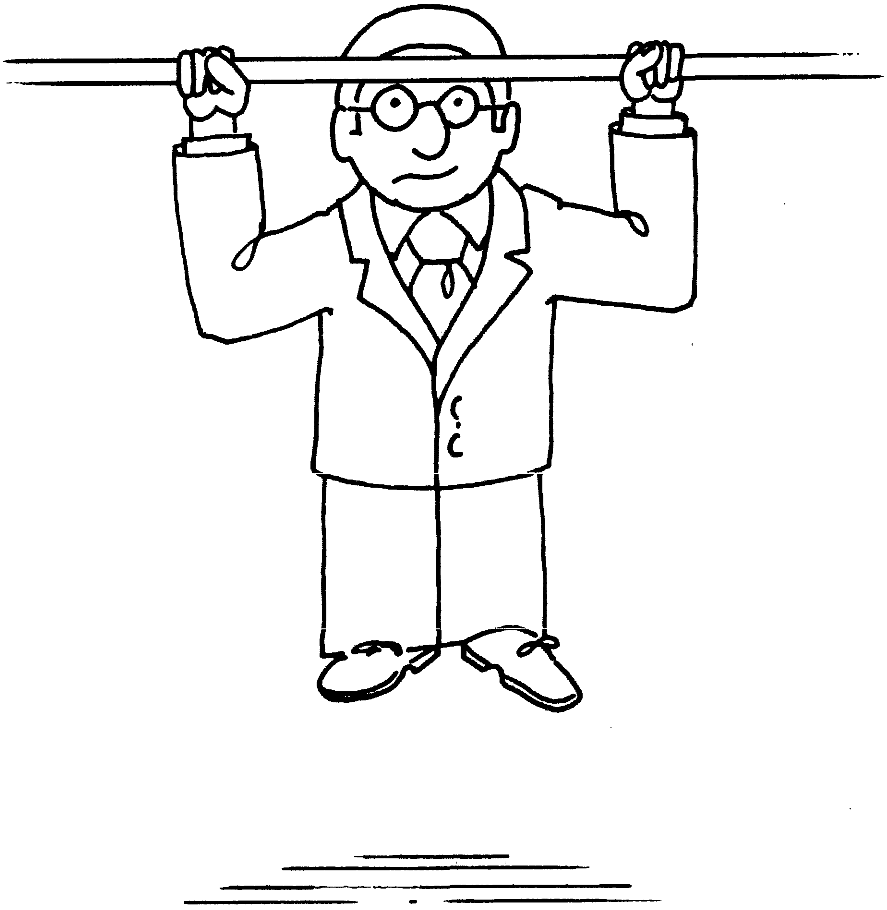
- RESET (000 005) – places an initialize signal on the Unibus so that every bus device is initialized to the state it held when power was first applied to the system. Note that RESET *has no effect* on the GPRs.

- WAIT (000 001) – forces the CPU to release the bus and wait for some external device to assume control of the bus and issue an interrupt.

- JMP (jump) – causes the program to go to a new location. Notice that it technically has the format of a single-operand instruction. The destination field specifies the location to jump to. An explanation of the prime differences among JMP, BR, JSR, etc. is given in the REFERENCE section of this workbook.

read on ➧

STUDY EXERCISES



read on ➧

PDP-11 INSTRUCTION SET

- **INTRODUCTION**

The study exercises in this section of your workbook have two purposes:

1. To reinforce your understanding of the PDP-11 instruction set.
2. To teach you some additional programming techniques not covered in the audio/visual portion of this course.

- **WHAT YOU NEED**

Before you start the study exercises, make sure that you have:

1. A PDP-11 instruction card and/or processor handbook.
2. Scratch paper.

- **EXERCISES**

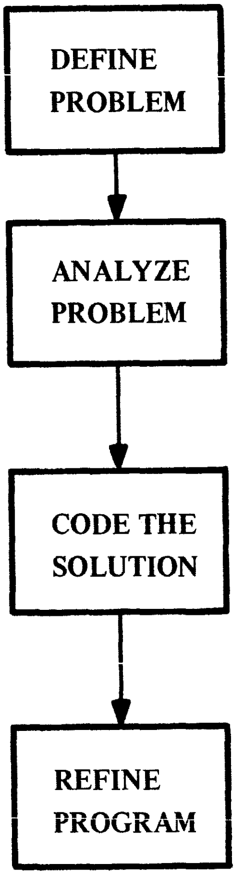
1. Do all of the numbered exercises in each section of this workbook.
2. At the end of each section is a page of *optional* MINI-EXERCISES which stress coding, decoding, and understanding of individual instructions.

read on ➧

study exercises

THE SECRET OF GOOD PROGRAMMING

The following four basic steps are essential to all programming. You will be successful if you always perform these steps whenever writing programs. As you go through subsequent exercises, we will help you learn each one of these steps.

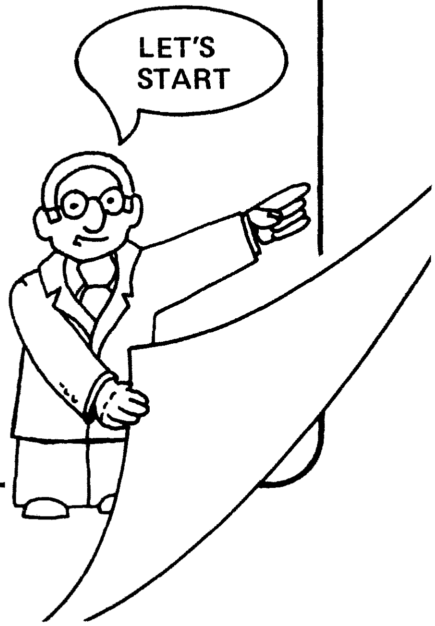


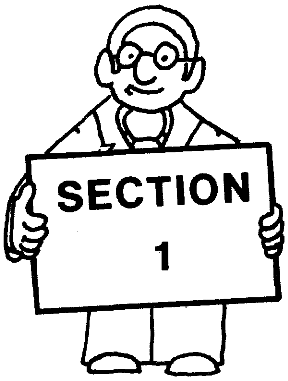
What needs to be done?

Make a flowchart that solves the problem.

Select appropriate instructions for the flowchart.

Eliminate as many instructions as you possibly can.





SINGLE-OPERAND INSTRUCTIONS

COVERAGE

The following exercises will make use of many of the following single-operand instructions:

CLR
COM
INC
DEC
NEG
TST
ASL
ASR
ROL
ROR

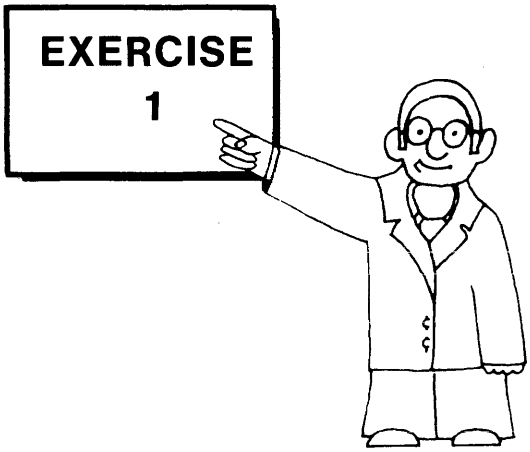
A SPECIAL AID

We know that using nothing but single-operand instructions can limit the types of programs you can write. However, by simply adding a BRANCH instruction, much more flexible programming is possible. Rather than use some general term, we are going to let you use an actual instruction which is:

BNE – Branch if *not* equal to *zero*

Thus, if the result of the previous instruction is *not* zero, the BNE causes a branch. If the result *is zero,* no branch occurs.

read on ➧



Our PROBLEM is to clear out a series of memory locations prior to loading them with new information.

Because this is the very first exercise in the workbook, we'll give you a little help. On the following page we've defined the problem, listed the known factors, and included a blank flow diagram.

YOUR JOB IS TO:

1. Fill in the flow diagram.

2. Implement the flow with appropriate instructions.

3. Refine your program, if possible.

read on ➧

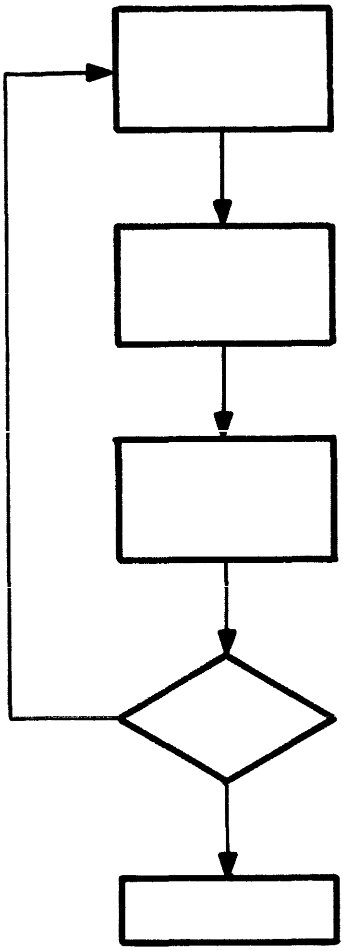
EXERCISE 1

Problem Clear out 12 sequential memory word locations beginning at address 600, then stop.

Known Factors 600 is the first address of the memory block (assume this value is already stored in R0).
12 decimal word locations can be represented by octal 14 (This value is stored in R1)

A Hint Use a loop and tally.

R0	R1
600	14



TURN FOR ANSWER

ANSWER FOR EXERCISE 1

YOUR SOLUTION MAY BE DIFFERENT

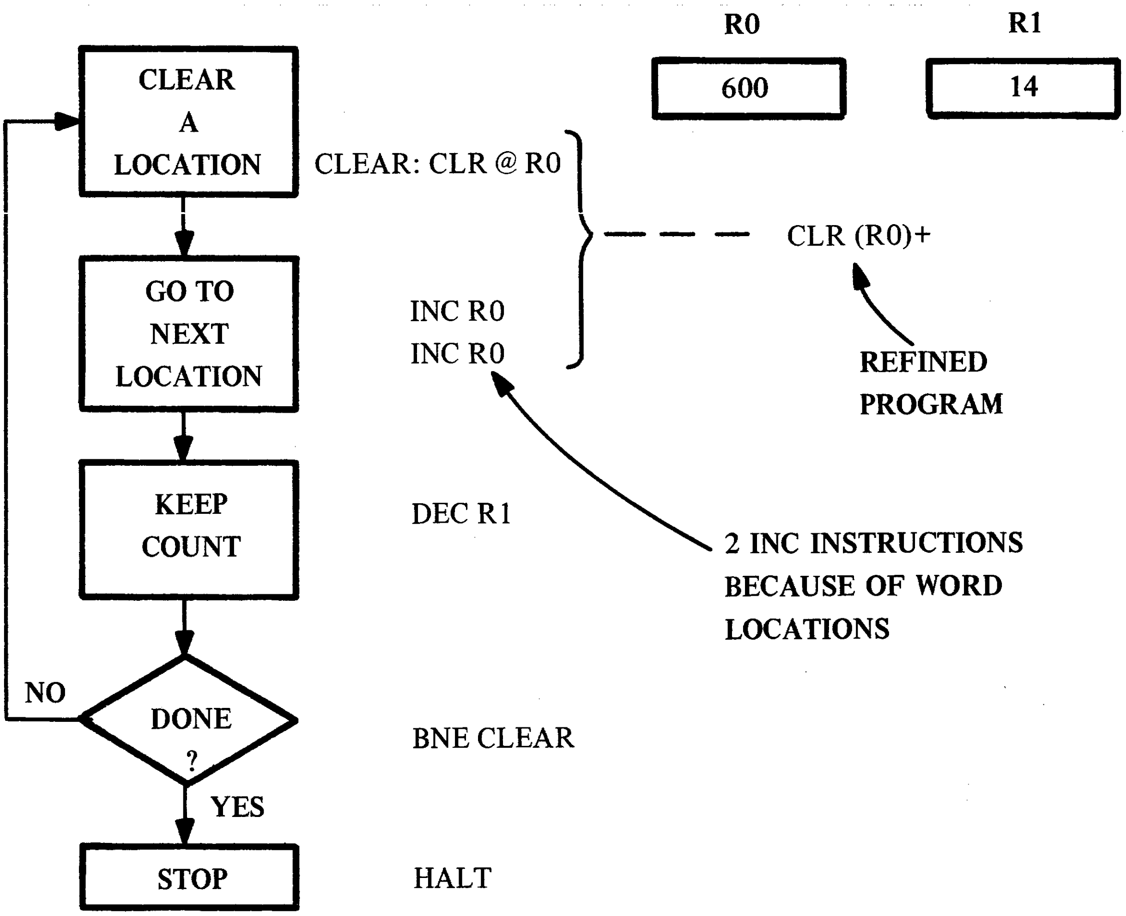
This program can be solved in a variety of ways. We are showing you only ONE POSSIBLE SOLUTION.

If your answer is different, it must contain:

1. A method of referencing sequential addresses (we used R0 as a pointer and incremented it after each operation).

2. A method of counting the number of locations cleared (we used R1 as a counter and decremented it).

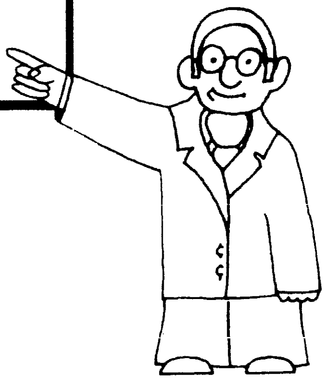
3. A method of knowing when to stop (we used a BNE in our loop so we could stop when the counter reached zero).



```
CLEAR: CLR @ R0
       INC R0
       INC R0
       DEC R1
       BNE CLEAR
       HALT
```

read on

EXERCISE 2



Problem

A table of operands is stored in memory and we need to decrement each operand in the table by one. The table contains 20_{10} (24_8) operands; each occupies one byte location.

Known Factors

The starting address of the table is 1000.

Your Job

Write a program to solve this problem. Assume a count of 24_8 is stored in R0. Use the INDEX mode to address operands stored in the table.

read on ➧

ANSWER FOR EXERCISE 2

We're showing you one possible solution to this problem. Notice how we went about it.

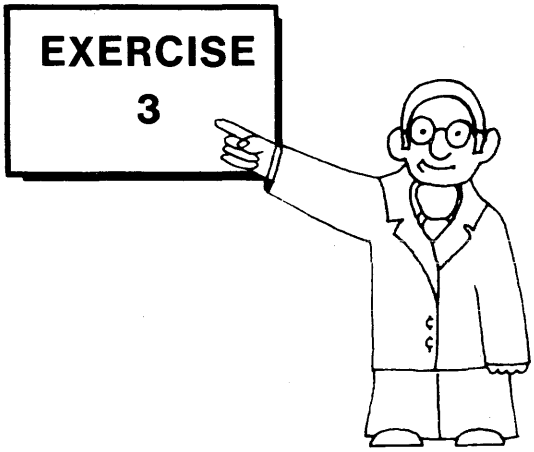
We start by clearing R1. Then we use a decrement byte (DECB) instruction to decrement the first operand stored in the table. The effective address of this operand is calculated by summing the contents of R1 (zero) with an index word of 1000.

Next, we increment the contents of R1. This allows us to access the second operand in the table on the next pass through this program. We also decrement the count stored in R0. Then we check if the count is zero. If the count is not zero, the branch instruction (BNE) returns us to the DECB instruction and the loop is repeated.

This process continues until all operands in the table have been decremented. The last decrement of R0 results in zero so that the branch condition is no longer fulfilled and the program goes to the next instruction, which is HALT.

```
            (R0)   =   24

            CLR R1

LOOP:       DECB 1000 (R1)

            INC R1

            DEC R0

            BNE LOOP

            HALT
```

read on ➧



Examine the following program and then answer the questions listed below.

```
(R0)   =   064 000

           ASR  R0
           ASR  R0
           ASR  R0
           ASR  R0
           HALT
```

1. What is the purpose of the program?

2. What does R0 contain when the program stops?

3. Write a *simple* program that will *multiply* the number 123 by 2^7. Assume that the initial conditions are as follows:

   (R0) = 123
   (R1) = 7

read on ➧

ANSWERS FOR EXERCISE 3

1. What is the purpose of the program?

   TO DIVIDE A NUMBER BY 16 (2^4)

2. What does R0 contain when the program stops?

   R0 contains 003 200

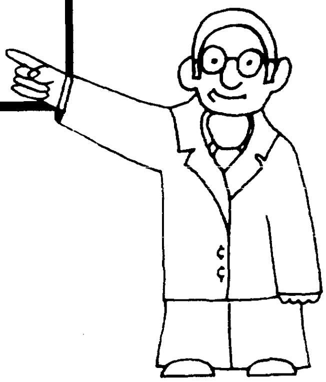
3. Write a *simple* program that will multiply the number 123 by 2^7.

R0	R1
123	7

```
LOC:      ASL R0
          DEC R1
          BNE LOC:
          HALT
```

read on

EXERCISE 4



Look at this program, then answer the following questions.

```
(R0)   =   042 345
C-Bit  =   0

           ROR R0
           SWAB R0
           ROL R0
           HALT
```

1. What is the state of the N condition code bit after ROL?

2. What is the state of the Z condition code bit after ROL?

3. When the program stops, what value is in register R0?

read on ➧

ANSWERS FOR EXERCISE 4

There is only ONE VALID ANSWER for each question. In case your answers are incorrect, we have shown you what happens step-by-step as the program is executed.

1. The N condition code is set after ROL.
2. The Z condition code is clear after ROL.
3. The value in register R0 is: 162 104

REGISTER R0

	C-BIT	BINARY	OCTAL
Initial condition	0	0 100 010 011 100 101	042 345
ROR R0	1	0 010 001 001 110 010	021 162
SWAB R0	0	0 111 001 000 100 010	071 042
ROL R0	0	1 110 010 001 000 100	162 104

Negative number (N=1)
Non-zero condition (Z=0)

HALT

read on ➧

MINI-EXERCISES

These exercises are optional and are provided if you would like practice in coding, decoding, and understanding single-operand instructions.

1. R0 contains the value 177 704. Write the mnemonic for each instruction and indicate the result.

   a. 005 200
   b. 105 200
   c. 005 010
   d. 000 300

   NOTE
   This is not a program. Treat each instruction separately. The same value of 177 704 is in R0 in each case.

2. Each of the following instructions is either illegal or causes a problem. Explain why in each case.

   a. 005 007
   b. 100 325 (used for a swap byte operation)
   c. 106 206

3. Write the following instructions in octal form.

   a. COM @(R5)+
   b. DECB @4(R3)
   c. TST @#177514
   d. ASLB -(R4)

4. The following conditions exist. Write the instruction mnemonic.

Address		Content
(R1)	=	200
(200)	=	214
(214)	=	026

NOTE - These conditions are the same for each of the four cases below. Treat each case separately.

   a. Increment contents of location 200; increment address 214.
   b. Complement the number contained at address 214.
   c. Divide the contents of address 200 by 2, using an addressing mode with R1.
   d. Clear the register.

read on ➧

MINI-EXERCISES
(ANSWERS)

1.	a.	INC R0	(R0)	=	177 705	177 704 is address of R4, which cannot be used by program. This address is only used by console switches when manually accessing R4.
	b.	INCB R0	(R0)	=	177 705	
	c.	CLR @R0	(R0)	=	No change	
			(R4)	=	No change	
	d.	SWAB R0	(R0)	=	142 377	

2. a. CLR R7 ← Destroys the PC

b. Cannot use byte operation with a SWAB

c. ASRB R6 ← Destroys the SP

3. a. 005135

b. 105373
000004 ← Index word

c. 005737
177514

d. 106344

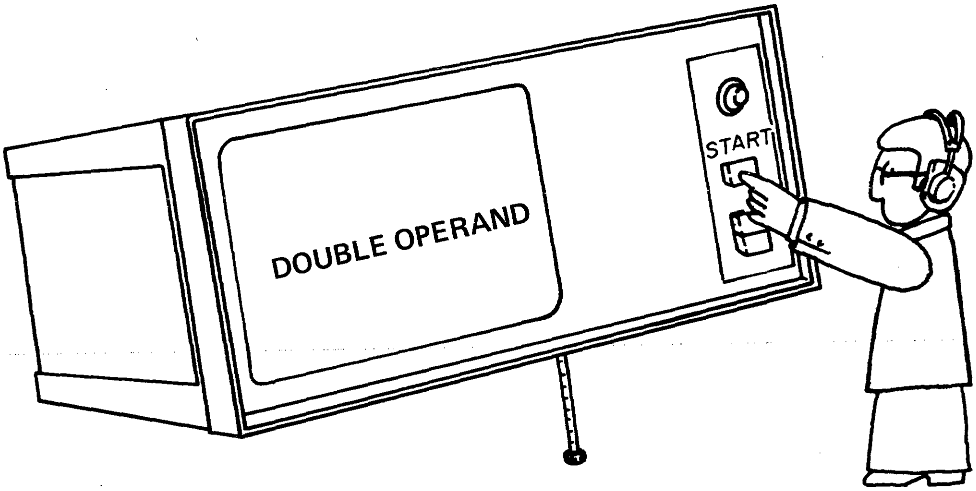
4. a. INC @R1; INC #214

b. COM @(R1)+ or COM 14(R1)

c. ASR @R1

d. CLR R1

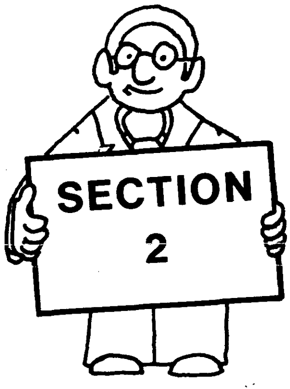
read on



RETURN TO A/V

FILM CARTRIDGES C & D

exercises: section 2



DOUBLE-OPERAND INSTRUCTIONS

COVERAGE

The following exercises will make use of many of the double-operand instructions:

MOV	ADD	BIT	BIC
CMP	SUB		BIS

KEY POINTS

Although they have many uses, double-operand instructions are most often used for three prime functions:

1. Initialization – setting up registers and memory locations for pointers, counters, etc.
2. Comparison – comparing two values to determine what course of action to take next.
3. Computation – performing arithmetic operations such as add and subtract.

SPECIAL NOTE

In this section of the workbook, perform exercises 1–4. (Mini-exercises are optional.)

read on ➧

HOW TO USE LABELS

Labels and symbols are often used to make our programs more understandable, to define values, and to provide convenient branching points.

Although labels and symbols are used with the PDP-11 assembler, we are going to begin using them in the remainder of this workbook so you will become familiar with them.

More complete descriptions of label use and restrictions are presented in the *PDP-11 Paper Tape Software Handbook*. We want to explain briefly what labels and symbols are.

LABELS

A label is a user-defined symbol which is assigned the value of the current location counter. It is a symbolic method of referring to a specific location within a program. For example:

```
200          START:     CLR R1
```

indicates that the value 200 is assigned to the label START. Thereafter, any reference to START is a reference to location 200. When we say, BRANCH START, we are saying, "branch to location 200." Other labels may be used to indicate other important program locations such as INIT: (initialize), MULTPY: (multiply), and CONT: (continue).

When using labels with the assembler, certain conventions must be followed. For example, a label must be six letters or digits or less; no label can begin with a number; no spaces can occur in a label; and each label must be terminated by a colon (:).

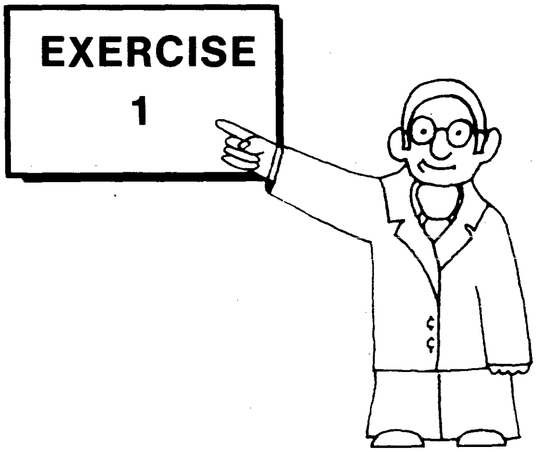
SYMBOLS

Symbols may be equated with a value. An equal sign must separate the symbol from the expression defining the symbol. For example:

```
A = 1
```

indicates that the symbol A represents the value 1 whenever that symbol appears in the program.

read on ➧



Problem

Write a program that multiplies N times 3.

Restrictions

1. Do not use a branch (write a straight line program).
2. Use GPR's R0 and R1 only. If possible, use only R0.
3. Use only labels and symbols in your program.

Your job

1. Write the program, observing the above restrictions.
2. When completed, assign memory addresses to each instruction and convert symbols to real values. (This step is handled automatically by the PDP-11 assembler.)
3. Use a starting address of 600 and assume N = 7.

read on ➧

ANSWER FOR EXERCISE 1

STEP 1

USING TWO REGISTERS

```
 INIT:      CLR R1
            MOV #N, R0

START:      ADD R0, R1
            ADD R0, R1
            ADD R0, R1
            HALT
```

USING ONE REGISTER

```
 INIT:      CLR R0

START:      ADD #N, R0
            ADD #N, R0
            ADD #N, R0
            HALT
```

STEP 2

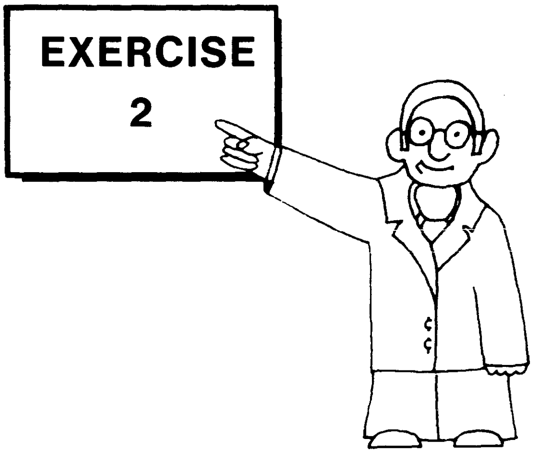
USING TWO REGISTERS

```
600     INIT:     CLR R1
602               MOV #7, R0
604               000 007
606   START:      ADD R0, R1
610               ADD R0, R1
612               ADD R0, R1
614               HALT
```

USING ONE REGISTER

```
600      INIT:     CLR R0
602    START:      ADD #7, R0
604                000 007
606                ADD #7, R0
610                000 007
612                ADD #7, R0
614                000 007
616                HALT
```

read on ➧



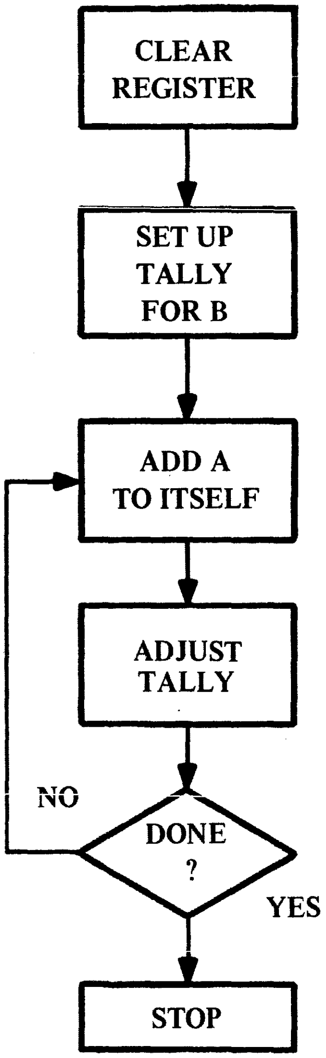
Problem Multiply A by B using repetitive additions and a program loop.

Your job

1. Make a flow diagram.
2. Implement the flow diagram using PDP-11 instructions.

read on ➧

ANSWER FOR EXERCISE 2



```
INIT:    CLR  R0

         MOV  B, R1

START:   ADD  A, R0

         DEC  R1

         BNE  START

         HALT
```

read on

EXERCISE 3

Problem

We want to add all the values in a series of memory locations from location 4000 through location 4076. We want the total result in register R0 and we want to make sure the program stops after it has added the last value.

Your job

The program below will do just what we want. However, *two instructions are missing.* Add the correct instructions to the program.

```
BEGIN   =   4000
END     =   4076
```

Program

```
INIT:     CLR R0
          MOV #BEGIN, R1
          MOV #END, R2

START:    missing instruction #1
          missing instruction #2
          BLOS START  (branch if lower or same)
          HALT
```

read on ➧

ANSWER TO EXERCISE 3

PROGRAM

```
INIT:       CLR R0
            MOV #BEGIN, R1
            MOV #END, R2

START:      ADD  (R1)+, R0
            CMP  R1, R2

            BLOS START

            HALT
```

EXPLANATION

Each time the ADD instruction is executed, the CPU retrieves a value from memory (using the address stored in R1), adds the value to the accumulated sum in R0, and then autoincrements R1 so that it points to the next value.

The CMP instruction is used to determine if:

(1) BEGIN < END
(2) BEGIN = END
(3) BEGIN > END.

If BEGIN < END, the BLOS instruction causes a branch back to START and another addition is performed.

If BEGIN = END, the BLOS instruction still causes a branch back to START (i.e., since R1 is autoincremented *after* the addition, we have not yet added the *last* value).

If BEGIN > END (i.e., when R1 is autoincremented to 4100), the branch condition is no longer satisfied and, consequently, the program will HALT.

read on ➧

EXERCISE 4

This exercise consists of four separate parts: A, B, C, and D.

A. Bit 7 of the I/O control and status register (CSR) is the READY flag. The following program waits for ready by continually looping until the flag is set, at which time the program continues.

Your job is to replace the single-operand TSTB instruction with a double-operand instruction that does the same job.

```
LOOP: TSTB @#CSR        ; test CSR for ready condition
      BPL LOOP          ; branch if plus
```

B. Study the conditions and program given below. Then explain what job the program is doing. DBR signifies a data buffer for an I/O device, such as a printer.

```
(R0)  =  100
(R1)  =  600
(R2)  =   32
```

addresses 600 through 632 contain the numbers 0 through 32_8, in random order

```
START:    ADD R0, (R1)
LOOP:     BITB #200, @#CSR
          BPL LOOP
          MOV (R1), @#DBR
          INC R1
          DEC R2
          BNE START
          HALT
```

C. Refine the above program by eliminating one instruction.

D. The following program multiplies 16 by 3. Although the program works, it *might* produce an erroneous result because we left out one instruction. What is that instruction?

```
MOV #16, R0
ADD R0, R1
ADD R0, R1
ADD R0, R1
HALT
```

read on ➧

ANSWERS FOR EXERCISE 4

A. BITB #200, @#CSR

B. *This answer is correct*: The program adds the value 100 to a memory location, then moves the resultant value to an I/O device, and proceeds to the next location, stopping after 32 octal locations have been dealt with.

This answer is even more correct: the program converts the contents of each memory location to an alphabetical equivalent (by adding 100 to obtain the ASCII code) and then sends the data to an I/O device for printing the appropriate character.

C.

```
ADD R0, (R1)
MOVB (R1), @#DBR  }
INC R1            }   MOVB (R1)+, @#DBR
DEC R2
BNE
HALT
```

D. The first instruction in the program should be: CLR R1. If we don't do this, there may be a value in R1 that can give us an erroneous result. The program will provide the correct answer only if the contents of R1 is zero initially.

read on ➧

MINI-EXERCISES

These exercises are optional and are provided if you would like more practice in coding, decoding, and understanding double-operand instructions.

1. Write the mnemonic for each of the following instructions and indicate the result of the instruction. Note the conditions given. Assume these same conditions exist for each instruction.

   (R1) = 005 (006) = 004
   (R2) = 004 (005) = 003
   (R3) = 006 (004) = 002

   a. 110 102
   b. 020 213
   c. 061 202
   d. 030 213

2. What is wrong with the following instructions?

   a. ADDB R1, R2
   b. BIS R0, R7

3. Write the following instructions in octal form.

   a. CMP @(R3)+, @-(R2)
   b. BICB R1, (R2)+
   c. SUB - (R1), @6(R4)
   d. BIT #30, (R3)

4. Use only *one* double-operand instruction to do each of the jobs listed below. Note the conditions given.

   (R1) = 2, (002) = 006, (026) = 005.

   a. Subtract 005 from 006.
   b. Set up locations 2 and 26 so they both contain 006.

read on

MINI-EXERCISES
(ANSWERS)

1.
   a. MOVB R1, R2 — Both R1 and R2 now contain 005
   b. CMP R2, (R3) — No change in R2 or R3. Z-bit is set.
   c. ADD (R2), R2 — R2 contains 006
   d. BIT R2, (R3) — No change in R2 or R3. Z-bit is cleared.

2.
   a. No byte operation is available with the ADD or SUB instructions.
   b. Destroys the program counter (PC).

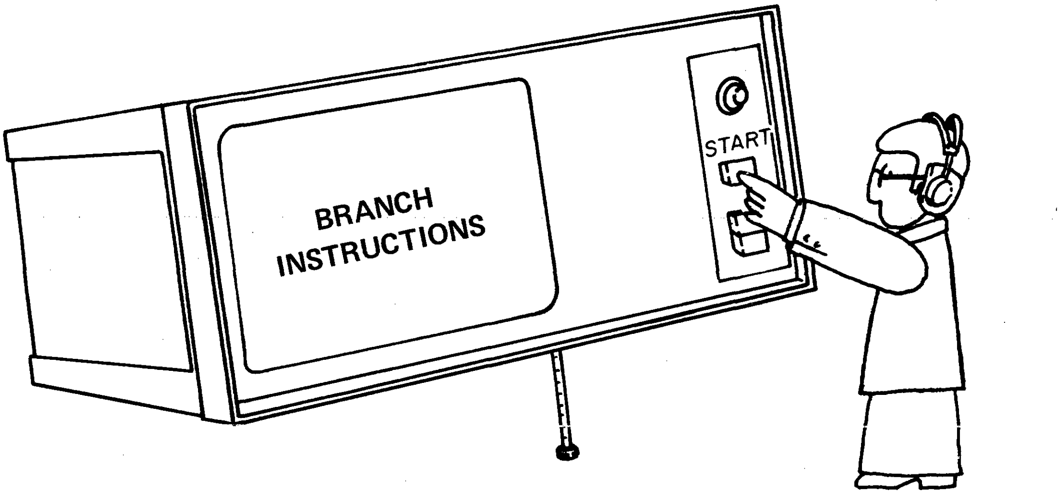
3.

```
a.   023352
b.   140122
c.   164174
     000006 <---- index word
d.   032713 \
     000030 /   immediate addressing mode
```

4.

```
                          OCTAL
a.   SUB  @#26, @#2    -{ 163737       NOTE: Absolute mode for
                        { 000026             both src and dst
                        { 000002

b.   MOV  @#2, @#26    -{ 013737       NOTE: Absolute mode for both
                        { 000002             src and dst. In a MOV
                        { 000026             instruction, src does not
                                             change but dst does change.
                                             Therefore, both locations
                                             now contain 006.
```

read on ➧



RETURN TO A/V
FILM CARTRIDGES E, F, & G

BRANCH INSTRUCTIONS

COVERAGE

The following exercises will make use of the single unconditional BR instruction and many of the sixteen *conditional* branch instructions.

KEY POINTS

Branch instructions add extreme versatility to programming. Because we do not want to write lengthy and unwieldy programs, we often have "decision blocks" to tell us which of two paths to take. *Branch instructions implement these decision blocks.*

read on ➧

EXERCISE 1

The following program negates all numbers in a block of memory. In other words, it makes all positive numbers negative and all negative numbers positive. Examine the program, then answer the questions.

Conditions:

Conditions:	BEGIN	=	Starting address of memory block.
	LIMIT	=	Final address of memory block.

NOTE

The number sign before BEGIN means we are using the defined number which is the starting address.

Program:

```
INIT:      MOV #BEGIN, R0
           MOV #LIMIT+2, R1

START:     NEG (R0)+
           CMP R0, R1
           BNE   START
           HALT
```

1. Why did we load #LIMIT+2 rather than just #LIMIT into R1?

2. Suppose we did load #LIMIT into R1. What branch instruction could we use in place of BNE?

3. Suppose we load #LIMIT into R1 and then change the compare instruction to CMP R1, R0. What branch instruction could we use in place of BNE?

read on ➧

ANSWERS FOR EXERCISE 1

1. Why did we load #LIMIT+2 rather than just #LIMIT into R1?

   Loading #LIMIT causes the CMP to stop the program when the pointer reaches the last location. In order to NEG the last location, we must load #LIMIT+2.

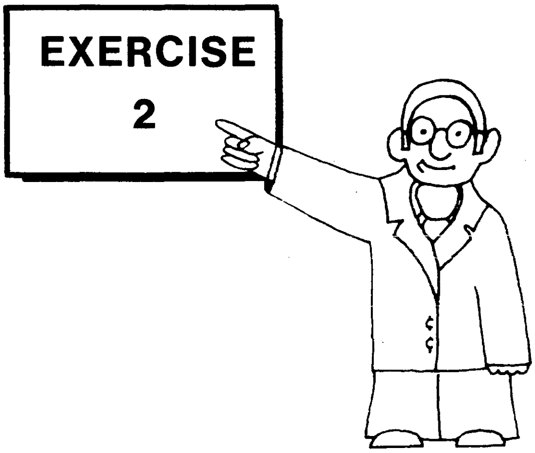
2. Suppose we did load #LIMIT into R1. What branch instruction could we use in place of BNE?

   BLOS (branch if lower or same)

3. Suppose we load #LIMIT into R1 and then change the compare instruction to CMP R1, R0. What branch instruction could we use in place of BNE?

   BHIS (branch if higher or same)

read on ➧



Remember the following program from step 2 of the previous exercise?

```
INIT:       MOV #BEGIN, R0
            MOV #LIMIT, R1
            START:
            NEG (R0)+
            CMP R0, R1
            BLOS START
            HALT
```

Problem The program, as written, will negate all numbers in a given memory block. However, suppose some of the numbers are *already* negative.

Your job Modify the above program so that it will only NEGATE positive numbers and will ignore negative numbers.

read on ➧

ANSWER FOR EXERCISE 2

YOUR ANSWER MAY BE DIFFERENT – There are many methods of solving this particular problem. We have indicated two methods below.

Our preference is method #1 because of two reasons:

1. It has one less instruction.
2. It uses the single-operand TST rather than the double-operand BITB instruction, thereby making it slightly faster.

Notice the COMMENTS we have added to the first example. This is often done in programming to make the program easier to understand by the user. Notice that each COMMENTS line begins with a semicolon.

METHOD #1

```
INIT:      MOV #BEGIN, R0       ; Set up starting address
           MOV #LIMIT, R1       ; and limit.

START:     TST (R0)             ; Is number negative?
           BMI CONT             ; If so, skip next step.
           NEG (R0)             ; If not, make it negative.

CONT:      ADD #2, R0           ; Increment pointer by 2.
           CMP R0, R1           ; Compare pointer with limit.
           BLOS START           ; If pointer is lower or same as
                                ; limit, go back.
           HALT                 ; Otherwise stop.
```

METHOD #2

```
INIT:      MOV #BEGIN, R0
           MOV #LIMIT, R1

START:     BIT #100 000, (R0)
           BMI CONT
           NEG (R0)

CONT:      INC R0
           INC R0
           CMP R0, R1
           BLOS START
           HALT
```

read on ➧

EXERCISE 3

Here is the updated program from the previous exercise. Now it will negate only positive numbers and ignore negative numbers.

```
INIT:       MOV #BEGIN, R0
            MOV #LIMIT, R1

START:      TST (R0)
            BMI CONT
            NEG (R0)

CONT:       ADD #2, R0
            CMP R0, R1
            BLOS START
            HALT
```

ANOTHER MODIFICATION

The whole purpose of our program so far is to make sure we have nothing but negative numbers. What about zero? If a location contains zero, negating it leaves it at zero.

Your Job

Assume that we have a program called PRINT which will print out the address of the current memory location.

Modify the above program so that it will branch to PRINT whenever it finds a location with zero.

Add "comments" to any new instructions you add to the program.

read on ➧

ANSWER TO EXERCISE 3

```
INIT:       MOV #BEGIN, R0
            MOV #LIMIT, R1

START:      TST (R0)
            BMI CONT
            NEG (R0)

            BEQ PRINT               ; If result is zero,
            (or BCC PRINT)          ; branch to PRINT program;
                                    ; otherwise, keep going.

CONT:       ADD #2, R0
            CMP R0, R1
            BLOS START
            HALT
```

read on ➧

EXERCISE 4

Here is the complete program from the previous exercise. Notice that we are showing the first and last instruction of the PRINT program. Notice also, that we have assigned specific memory locations to each instruction.

```
600     INIT:      MOV #BEGIN, R0
602                XXX XXX
604                MOV #LIMIT, R1
606                XXX XXX

610     START:     TST (R0)
612                BMI CONT
614                NEG (R0)
616                BEQ PRINT

620     CONT:      ADD #2, R0
622                000 002
624                CMP R0, R1
626                BLOS START
630                HALT
632     PRINT:     CLR R2
634                   .
636                   .
640                   .
642                BR CONT
```

Your job Calculate the proper *offset* value for each of the four branch instructions.

BMI CONT has an offset of ________
BEQ PRINT has an offset of ________
BLOS START has an offset of ________
BR CONT has an offset of ________

read on ➧

ANSWER FOR EXERCISE 4

REMEMBER

The offset is always calculated from the updated Program Counter which is one word location beyond the branch instruction you are dealing with.

The answers are:

BMI	offset is plus 2
BEQ	offset is plus 5
BLOS	offset is minus 10_8 (or 370)
BR	offset is minus 12_8 (or 366)

By the way, notice that for the BR offset, we had to count back from location 644 (not shown). Because we went back *ten decimal* word locations, the negative offset is *octal twelve.*

read on ➧

EXERCISE 5

Comment

Examining unknown data against a known value is done with a CMP instruction.

Problem

Our problem is to examine the contents of a specific memory location and make some judgment about the data. Each case listed below will look for a different condition.

Conditions

The following conditions exist for each of the cases listed below:

R0 contains the address of a memory location.
R1 contains an *unsigned* value of 000 005.

Your Job

Below are four different instructions. In each case, write the particular branch instruction that will satisfy the specified condition. Assume that we are working with *unsigned* values.

1. Branch if memory location contains 000 005 or *more.*

   CMP (R0), R1
   ____________(fill in appropriate branch)

2. Branch if location contains more than 000 005.

   CMP (R0), R1
   ____________(fill in appropriate branch)

3. Branch if location contains less than 000 005.

   CMP (R0), R1
   ____________(fill in appropriate branch)

4. Branch if location contains 000 005.

   CMP (R0), R1
   ____________(fill in appropriate branch)

read on ➧

ANSWERS FOR EXERCISE 5

		NOTES
1.	BHIS or BCC	By reversing the compare (CMP R1, @R0) we could have used a BLOS rather than a BHIS.
2.	BHI	
3.	BLO or BCS	
4.	BEQ	Here we are only interested in equivalency.

read on ➧

SOME HINTS

By now, you are beginning to realize how versatile the branch instructions are. You might probably be wondering how to know when to use specific branches because many seem to do the same or similar jobs. Well, here's a few tips you might like to use.

Testing for Errors

The C, V, and N bits are most often used to check for error conditions. If that's what you want to do, then use one of these branches:

BCC	BVC	BMI
BCS	BVS	BPL

Comparisons

When we are comparing two numbers only to find out if they match (if they are equal or not), then we normally use the instructions BEQ or BNE. If we want to branch if the two numbers are *equal,* we use BEQ (branch on zero or match). If we want to branch if they are *not equal,* we use BNE (branch on no zero or no match).

Greater or Less (Signed Conditional Branches)

Here are a few tips that will help you select the proper branch when trying to find out if the result is greater or less. Remember two points:

1. These branches are normally used after a CMP instruction and greater or less refers to the SOURCE. In other words, branch if the SOURCE is greater, for example.

2. These branches are used when we are concerned with the *sign* of the word and are, therefore, referred to as signed conditional branches.

BGT	Branch if greater than	SRC > DST
BGE	Branch if greater than or equal	SRC ⩾ DST
BLT	Branch if less than	SRC < DST
BLE	Branch if less than or equal	SRC ⩽ DST

(continued on next page)

read on ➧

SOME HINTS (Continued)

Higher or Lower (Unsigned Conditional Branches)

The following four branch instructions are also normally used after a CMP instruction but, because they are used with words which we do not read as plus or minus numbers, they are referred to as *unsigned* conditional branches.

BHI	Branch if higher	SRC > DST
BHIS	Branch if higher or same	SRC ⩾ DST
BLO	Branch if lower	SRC < DST
BLOS	Branch if lower or same	SRC ⩽ DST

Whenever we are comparing *addresses,* the *unsigned* conditional branch instructions should be used. For example, BLOS should be used rather than BLE because BLE will interpret an address of 100 000 or greater as a *negative* number.

read on ➧

EXERCISE 6

Problem

We are dealing with payroll data that consists of a series of 16-bit words. The *high byte* of each word contains the employee's badge number, the *low byte* contains an octal number ranging from 0 to 13. These numbers represent salary levels within *three wage classes* to identify which employees get paid weekly, bi-monthly, or monthly.

It is now time to make out *weekly* paychecks. Unfortunately, employee information has been stored in a random order. The problem is to extract the *badge numbers* of those employees that are to receive a *weekly* paycheck.

Conditions

Employee payroll numbers are assigned as follows:

0 to 3	–	Wage class I	(weekly)
4 to 7	–	Wage class II	(bi-monthly)
10 to 13	–	Wage class III	(monthly)

600 is the starting address of memory block containing the employee payroll information
1264 is the final address of this memory block.

Your job

Write a program that will:

1. Search through the memory block and find all employee payroll numbers representing wage class I.
2. Each time an appropriate number is found, store the employee's badge number (just the high byte) on a "last in, first out" stack which begins at location 4000.

Restrictions

Do not use the hardware stack.

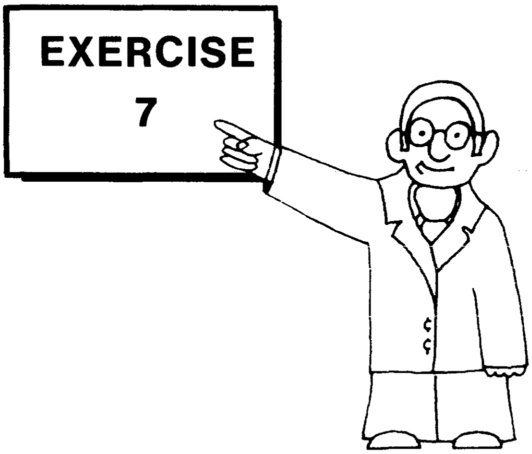
read on ➧

ANSWER FOR EXERCISE 6

Of course there are a variety of ways to solve this problem so we are just showing you one method.

```
INIT:       MOV #600, R0            ; Set up an address pointer
            MOV #4001, R1           ; Set up a stack pointer

START:      CMPB (R0)+, #3          ; Compare the contents of the first low
                                    ; byte with the number 3 and increment
                                    ; the pointer by one.

            BHI CONT                ; If the number is more than 3, branch
                                    ; to continue.

STACK:      MOVB (R0), -(R1)        ; Otherwise, decrement the stack
                                    ; pointer and move the high byte
                                    ; containing the badge number
                                    ; onto the stack.


CONT:       INC R0                  ; Advance R0 to examine next low byte.

            CMP #1264, R0           ; Compare the limit with the
                                    ; updated pointer.


            BHIS START              ; If the limit is higher or the same
                                    ; as the pointer, go back and examine
                                    ; the next low byte.

            HALT                    ; Otherwise, stop.
```

read on ➧



We have assigned memory locations to the program that you wrote in the previous example.

Your job Calculate the offset for all branch instructions

```
200    INIT:     MOV #600, R0
202              000 600
204              MOV #4001, R1
206              004 001
210    START:    CMPB (R0)+, #3
212              000 003
214              BHI CONT              BHI ________ ?
216    STACK:    MOVB (R0), -(R1)
220    CONT:     INC R0
222              CMP #1264, R0
224              001 264
226              BHIS START            BHIS ________ ?
230              HALT
```

read on

ANSWER FOR EXERCISE 7

The correct offset values are:

BHI (offset is plus 1)
BHIS (offset is minus 10_8 or 370)

A comment regarding the BHI instruction

The BHI instruction has an offset of 1 (not 2). Remember, the offset is always given in *words.*

Some notes on BHIS (offset minus 10)

1. Did you remember that you had to count back from location 230 because that is where the PC is located?

2. You probably counted back 8 words but remember that the offset is expressed in octal, and is therefore minus ten.

3. If you didn't count the words, did you use the formula?

$$\text{offset} = \frac{\text{effective address minus PC}}{2}$$

which would be:

$$\text{offset} = \frac{210 \text{ minus } 230}{2} = \frac{-20}{2} = -10$$

An important note

Although the previous example appears to be decimal division, it is octal. Here is an example that more clearly shows the effect of octal division. Assume that the PC is at location 212 and the effective address is 222.

$$\text{offset} = \frac{\text{effective adrs. -PC}}{2} = \frac{222 - 212}{2} = \frac{10}{2} = 4$$

In this case the offset is 4 words forward, not five. Remember that we are dealing with OCTAL numbers.

read on ➧

A WORD ABOUT BRANCH SYNTAX

Up to now, we have been talking about branching – for example, back -3 or forward +5 words. When writing a program in *octal code,* this means that we not only must count the number of words to the branch location, but we must also convert negative offsets into their two's complement form.

Fortunately, most programs are written using symbolic names and labels which are then translated into machine language by an assembler. When using the assembler, the programmer simply writes the branch mnemonic (such as BNE) and follows it with the label of the branch location.

For example, if it is desired to branch to location 210, which is assigned the label START, the programmer writes: BNE START

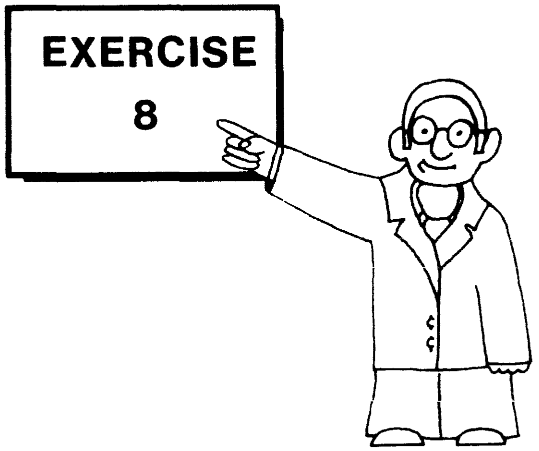
If the desired branch location exceeds the offset limit, the assembler will print out an error message for the programmer.

In the remainder of this workbook, assembler syntax will be used for branches unless otherwise specified.

An Example

```
210     START:     ADD
212                SUB
214                DEC
216                BNE START ◄———— Causes a branch back of
220                HALT            -4 words to location 210
```

read on ➧



We goofed! After spending over a week writing an extensive program, our boss informed us that we had made a number of errors. He was kind enough to point out the errors, but didn't explain them to us.

Please tell us what is wrong in each of the following cases.

1. We wanted to branch forward 310 locations. The boss said our offset was wrong. Why?

2. At one point in our program, we had constructed a loop which contained an increment instruction. We decided to get out of the loop once the carry bit was set indicating we had incremented once too often. We did it this way but were told it was wrong. Why?

```
INC R0
BCS LOC
```

(exercise continued on page 105)

read on ➧

EXERCISE 8 (CONTINUED)

3. We wanted to compare two values, and then check each one of the condition code bits separately, branching to appropriate points based on the results. We did it this way:

```
CMP X,Y
BCS LOC1
CMP X,Y
BVS LOC2
CMP X,Y
BEQ LOC3
CMP X,Y
BMI LOC4
```

Our boss said there was an easier way. What is it?

4. When we added two numbers, it was important to know if either the carry bit was set or if the result was zero. We did it this way but out boss asked us to shorten it. How can we do it?

```
ADD X,Y
BCS LOC
BEQ LOC
```

read on ➧

ANSWERS FOR EXERCISE 8

1. We can only branch forward a maximum of 177 octal locations. A positive 310 is an invalid offset.

2. If we would have looked at our instruction card, we would have seen that the INC instruction does not affect the state of the C bit.

3. Why use all those CMP instructions? A branch *tests* condition codes but *never changes* them. It is much simpler to write:

```
CMP X,Y
BCS LOC1
BVS LOC2
BEQ LOC3
BMI LOC4
```

4. Notice that there are two branch instructions that branch back to the *same point* in the program. There is one instruction that tests both the conditions we are interested in. Therefore, we could have shortened the program by writing:

```
ADD X,Y
BLOS LOC
```

read on ➧

MINI-EXERCISES

These exercises are optional and are provided if you would like more practice in coding, decoding, and understanding branch instructions.

1. Write the mnemonic for each of the following instructions and indicate the offset value.

   a. 002 403
   b. 100 776
   c. 101 214
   d. 001 736

2. What is wrong with the following instructions?

   a. BR LOC (offset of +200)
   b. BNE R2
   c. BR LOC (offset of - 1)

3. Write the following instructions in octal form.

   a. BHIS LOC1 (offset of +73)
   b. BCC LOC2 (offset of +73)
   c. BR LOC3 (offset of - 22)
   d. BVS LOC4 (offset of - 127)

4. Select the appropriate branch instruction for the following conditions (use the mnemonic).

   a. Branch if higher or same.
   b. Branch if either C or Z bit is zero.
   c. Branch if either C or Z bit is one.
   d. Branch if plus.

read on ➧

MINI-EXERCISES
(ANSWERS)

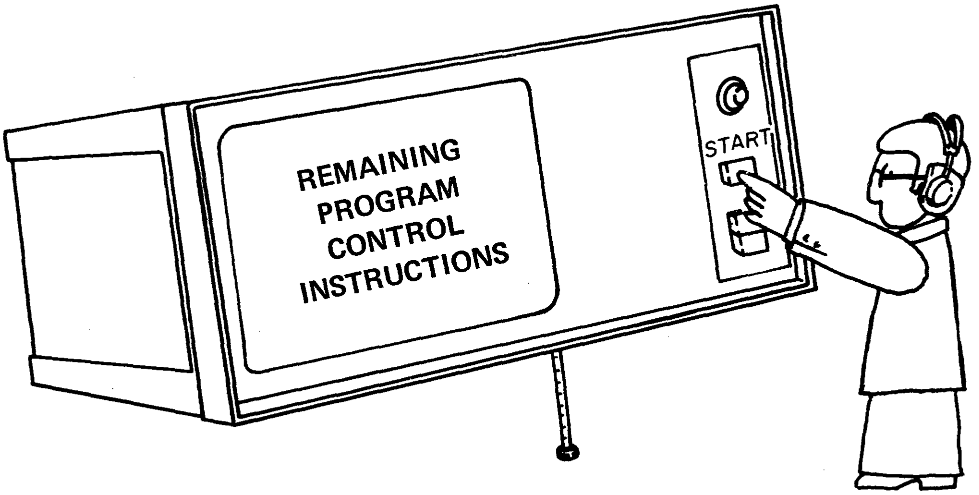
1. a. BLT (offset +3)
   b. BMI (offset -2)
   c. BHI (offset -164)
   d. BEQ (offset -42)

2. a. A positive offset can never exceed a maximum of 177.
   b. Branch instructions do not deal with registers. The only item that can follow a branch mnemonic is an offset.
   c. Causes a branch back to the branch instruction. Thus, we are locked into a perpetual loop.

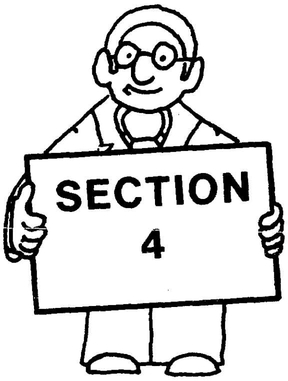
3. a. 103 073 } ← Remember? BHIS and BCC are the same.
   b. 103 073
   c. 000 756
   d. 102 651

4. a. BHIS or BCC
   b. BHI
   c. BLOS
   d. BPL

read on ➧



RETURN TO A/V
FILM CARTRIDGE H



OTHER PROGRAM CONTROL INSTRUCTIONS

COVERAGE

The following exercises cover three groups of program control instructions:

a. Subroutine – JSR and RTS
b. Intr. & Trap – RTI, IOT, EMT, TRAP, etc.
c. Miscellaneous – HALT, WAIT, RESET, NOP, JMP

KEY POINTS

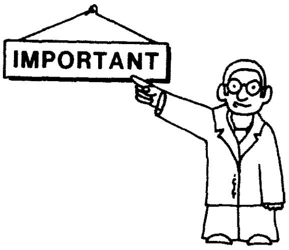
The three groups of instructions are used primarily as follows:

a. *Subroutine* – JSR causes the program to jump to a subroutine and then RTS returns the program back to the proper point when the subroutine has been executed.

b. *Intr. & Trap* – the RTI permits return from an interrupt or trap by "popping" the old PC and PS off the stack. The various "trap" instructions permit a trap to be initiated by the program rather than by hardware.

c. *Miscellaneous* — provide the normal program command functions of stopping, waiting, resetting, and jumping.

read on ➧



A FEW WORDS ABOUT SUBROUTINES

WHY USE SUBROUTINES?

There are three good reasons for using subroutines whenever possible:

1. *Memory savings* – it is only necessary to store *one* copy of the routine in memory.
2. *Time savings* – you, as a programmer, need to code the routine only once.
3. *Flexibility* – the routine can be used by the main program or by other routines.

WHAT ONE LOOKS LIKE

The subroutine itself looks just like any other portion of the program except for the first and last lines.

1. The first instruction of the subroutine is prefaced by a functional label to identify it. For example, the label might say "MULT" (multiply). We could then enter this subroutine by using the instruction JSR Rn, MULT. Notice that Rn indicates the *link register.*
2. The last instruction of a subroutine is *usually* an RTS instruction. In this case, RTS Rn would return us to the main program at the instruction that immediately follows JSR Rn, MULT.

HOW TO AVOID TROUBLE

Remember the importance of the link register? It is used to keep track of the proper point in the main program that we wish to return to.

The link register specified in the RTS instruction must be the *same register* that was used in the corresponding JSR instruction. For example, if the MULTIPLY subroutine ended with RTS R3, then "call" it with the instruction, JSR R3, MULT.

read on ➧

EXERCISE 1

We have written a subroutine that prints out characters of data on a line printer. We call this subroutine "PRINT."

Below, you see a portion of a main program which is followed by the PRINT subroutine.

Your job

1. Notice that the instruction following the MOV is missing. Write the instruction that will cause the main program to jump to the PRINT subroutine.

2. After the subroutine has been executed, control is returned to the program. Which instruction is executed *after* the RTS R4 instruction?

Program

```
            MOV (R3)+, R1

            ________________ (missing instruction)

            INC R2

            CMP R2, R3

            BEQ LOC
             .
             .
             .
             .
             .
             .
             .
             .
             .
PRINT:      TSTB @#CSR
             .
             .
             .
             .
             .
             .
             .

            RTS R4
```

read on ➧

ANSWER FOR EXERCISE 1

1. JSR R4, PRINT — If you used any register other than R4, it is wrong because the JSR instruction must use the same register for a "link" as the RTS instruction that ends the routine.

2. INC R2 — While executing the JSR instruction, the PC was pointing to INC R2 (the next instruction).

   This PC was loaded into the link (R4) by the JSR. When the PRINT subroutine executed the RTS R4 instruction, the old PC was taken from the link and became the current PC which points to INC R2.

read on ➧

EXERCISE 2

We want to ask you a few questions about the following program.

Conditions R0 contains the address of memory location 600.
R1 contains a count of 200 bytes.
R2 contains an unknown value.
LPB signifies a *data buffer* for a printer.
LPS signifies a *control and status register* for the same printer.

Program

```
START:      JSR R2, LOOP
            MOVB (R0), @#LPB  ⎫
                           or ⎬  MOVB (R0)+, @#LPB
            INC R0            ⎭
            DEC R1
            BNE  START
            HALT

LOOP:       BITB #200, @#LPS
            BPL LOOP
            RTS R2
```

1. What is the purpose of this program?

2. What is the purpose of the LOOP subroutine?

3. After RTS R2 is executed, what does the PC point to?

4. Later, we want to use the data originally stored in R2. Is this possible?

5. Instead of R2, could we use R1 for the JSR and RTS instructions?

read on ➧

ANSWERS FOR EXERCISE 2

1. The program basically moves the contents of a memory location to a line printer for printing. It automatically steps through memory locations and stops once it has printed 200 bytes of data.

2. The LOOP subroutine tests the ready flag in the printer's control and status register (LPS) and returns control to the main program once the printer is ready to receive data.

3. After execution of RTS R2, the PC points to MOVB (R0)+, @#LPB which was the next instruction to be executed after the JSR.

4. Of course. Whatever had been stored in R2 was saved because the JSR pushed the data on the stack before it used R2 as a link. Once the RTS has been executed, the data is popped off the stack to restore R2 to its original condition.

5. Yes. R0 is an address pointer and R1 is a counter. However, if we used JSR R1, LOOP, the count in R1 would be saved on the stack and then restored by the RTS R1 instruction. The main program would not even realize that R1 had been used as a link.

read on ➧

EXERCISE 3

1. We spent two weeks writing a program containing 500 separate instructions. We made one modification as shown.

```
            original                            modification

    400     MOVB (R0), R1               400     MOVB (R0)+, (R1)
    402     INC R0                      402
    404     NEG R3                      404     NEG R3
```

Questions:

a. What happens if we leave it this way?

b. What instruction can we put in location 402 that will make the program keep running but have no other effect?

2. We are writing a program and have found that we are at a point where we cannot proceed any further until an I/O device services our data. What should we do?

3. In our program, we need to branch *unconditionally* from location 1000 to location 4700. What instruction can we use?

4. In studying a trap error handling routine, we noticed that the routine ended with a return from interrupt (RTI) instruction. Using an RTI with a trap routine seemed odd. Is it?

5. Prior to starting a program, we wanted to make certain that all GPRs were cleared. In order to save time, we used the RESET instruction. Did this one instruction clear all the registers including the SP and PC?

read on ➧

ANSWERS FOR EXERCISE 3

1. a. If we leave it this way the program stops when it reaches location 402 because all 0's is the op code for a HALT.

   b. NOP. Note that a NOP instruction has many useful functions in spite of its name. By using a NOP here, we eliminate the need for assigning new locations to our 500 instruction program.

2. Use a WAIT instruction.

3. Use a JMP instruction. A JMP is the same as an unconditional branch with one important exception. JMP is not limited to the number of locations forward or backward it can jump to. Remember that the JMP is a single-operand instruction and the destination part of the instruction can be used with any addressing mode.

4. No. The RTI and RTT instructions are both used to exit from interrupt and trap handling routines. The RTI permits use of the T-bit while RTT inhibits the T-bit until completion of the instruction following the RTT.

5. No. Although RESET initializes *all* devices on the Unibus, it has no effect on the GPRs.

read on ➧

MINI-EXERCISES

These exercises are optional and are provided if you would like more practice in coding, decoding, and understanding program control instructions.

1. Write the mnemnoic for each of the following instructions.

   a. 000 135
   b. 004 512
   c. 000 001
   d. 000 002

2. What is wrong with the following instructions?

   a. RTS (R4)+
   b. RESET R7
   c. JSR @R5, (R4)+
   d. JMP+277

3. Write the following instructions in octal form.

   a. TRAP
   b. JSR R5, (R2)+
   c. JMP (R4)
   d. HALT

4. Write the octal code for two different NOP instructions.

5. Write the octal code for two different instructions used for exiting *trap* service routines.

6. The following conditions exist:

   (R0) = 2 (005) = 400
   (R1) = 6 (006) = 600

   Write an instruction mnemonic that causes a jump to a subroutine that starts at memory address 400.

read on ➧

MINI-EXERCISES
(ANSWERS)

1. a. JMP @(R5)+
   b. JSR R5, (R2)
   c. WAIT
   d. RTI

2. a. Addressing modes cannot be used with an RTS instruction.
   b. RESET is not a single-operand instruction, therefore, no register or operand can be used with it.
   c. Addressing modes cannot be used with the link register.
   d. A JMP does not use an offset. To specify the jump point, the standard destination field is used. Thus, if we wanted to jump to location 277, we might use JMP (R0) where R0 contained the number 277. NOTE: Mode 0 cannot be used with a JMP because it is impossible to transfer program control to a register.

3. a. Any octal value between 104 400 and 104 777 is correct.
   b. 004 522
   c. 000 114
   d. 000 000

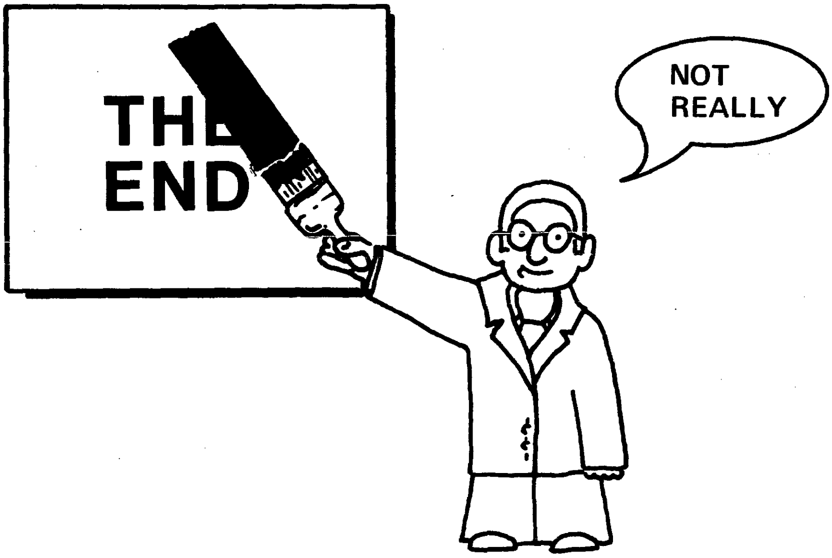
4. 000 240 and 000 260

5. 000 002 (RTI) and 000 006 (RTT)

6. JSR R4, 3(R0)

   NOTE:
   1. Any GPR can be used in place of R4.
   2. The index mode adds the value 3 to the contents of R0 to obtain 5. The value 5 is then used as the effective address of the operand (400).

read on ➧



This is the END of the workbook exercises but only the BEGINNING of PDP-11 programming.

In the next study unit we will delve even deeper into typical PDP-11 programs and software.

Your programming PROFICIENCY depends on PRACTICE.

We have tried to help by giving you some sample exercises. However, we urge you to continue practicing on your own. Write your own programs. Code them in both octal and mnemonic form. If any part of this workbook presented particular difficulty, review it and practice more programming related to that area.

NOTE

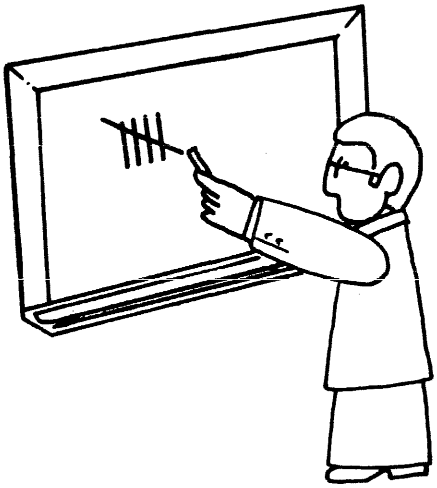
Demonstration programs and additional programming exercises are contained in the document entitled *Supplementary Programming and Console Exercises.*

read on ➧

REFERENCE MATERIAL

read on ➧

PROGRAMMING TECHNIQUES



COUNTING

There are two basic methods of keeping a tally by using any one of the general-purpose registers:

1. Load a *positive* number, *decrement* for each count, and branch on a zero condition (BEQ).

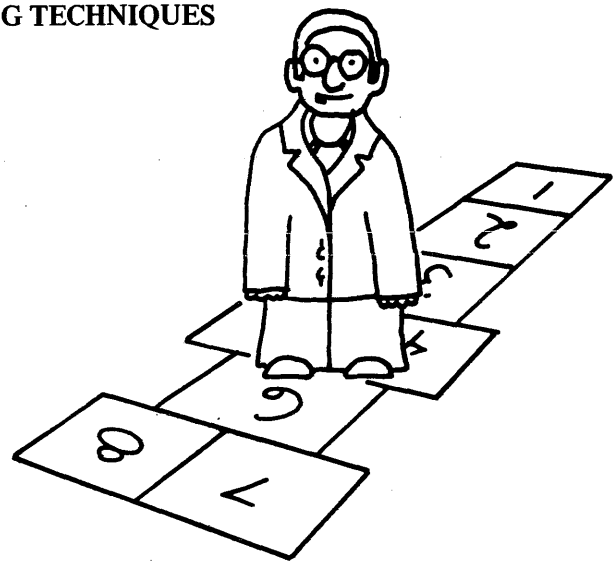
2. Load a *negative* number, *increment* for each count, and branch on a zero condition (BEQ).

If possible, avoid using the stack pointer (R6) or the program counter (R7) for counters unless you realize the effect this will have on the hardware stack and the program sequence.

It is possible to use a memory location as a counter but this is highly inefficient because it requires use of bus cycles that are not needed if GPRs are used.

read on ➧

PROGRAMMING TECHNIQUES



JUMPING AROUND

You already know a number of instructions that permit jumping to various points in a program. We're going to simply highlight the basic differences so you will know which one to use, depending on the job you want to do.

BR Although an *unconditional* branch seems to be the same as a JMP (but with a *limited* range), it actually can be used to advantage. Whenever you need to jump less than 200 locations, use the BR because it does not tie up any GPRs or require any bus cycles.

OTHER BRANCHES The 16 *conditional* branches have one advantage over all other types of jump instructions. They cause a branch (or jump) only when some specific condition exists.

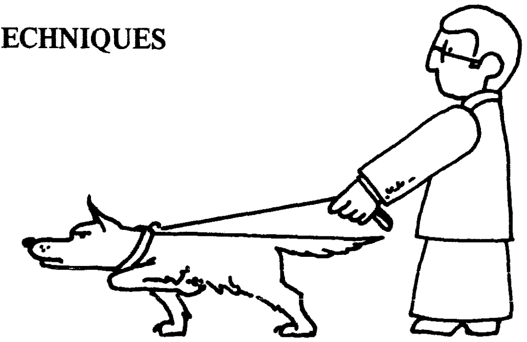
JMP The JMP instruction is more versatile than BR because it is not limited by range and can be used with all the registers and addressing modes except mode 0.

JSR The actual jump initiated by a JSR is the same as that caused by a JMP. However, the JSR makes a link so the program can return to the point where it left off. A JMP moves us someplace else but does not allow us to return. The JSR saves the contents of a GPR and then uses that GPR to save the PC so we can return to the next instruction in the main program.

TRAP Any TRAP instruction effectively causes a jump. However, it again provides a means of returning to the main program because it stores the PC and PS on the stack. The RTI or RTT instruction at the end of the trap routine can "pop" the PC and PS, thereby returning control to the main program.

read on ➧

PROGRAMMING TECHNIQUES



USING POINTERS

Pointers are useful tools for stepping through a series of consecutive operands. A pointer moves either forward, when it is incremented, or backward, when it is decremented. A pointer can be used to access either words or bytes data.

When dealing with pointers, it is more efficient to use the autoincrement and autodecrement modes than to use the INC and DEC instructions. However, there are some pitfalls that must be avoided. There is also a problem when the pointer reaches its limit.

AUTO-INCREMENT

This is perhaps used more often for pointers than any other mode. CLR (R0)+ is an example. We first do what the instruction says (such as CLR), and then automatically move the pointer to the next sequential location.

LIMITS

When dealing with pointers, we often *compare* the pointer with a limit to know when to stop. Since auto-increment moves the pointer *after* instruction execution, the condition POINTER = LIMIT means we have not done the required job on the *last* entry. To ensure the last entry is handled, we could use a BLOS (branch if lower *or same*) instruction in place of a BLO (branch if lower) instruction.

As we said above, comparing a pointer with a limit often results in failing to perform the required job for the last data entry if we are not careful. We can solve this problem in one of three ways:

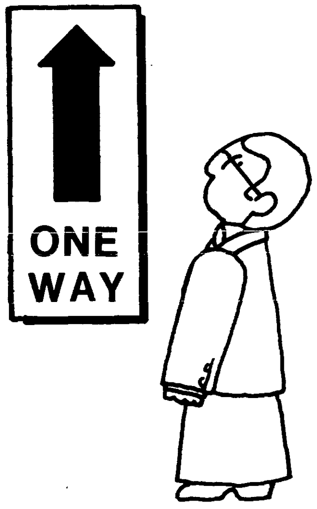
a. Use the auto-decrement mode instead of auto-increment.

b. Use the auto-increment mode but set the limit value one word (or byte) *past* our desired limit.

c. Choose the appropriate branch instruction. For example, a BLOS instruction rather than a BLO.

read on ➧

PROGRAMMING TECHNIQUES



UNDERSTANDING SYMBOLS

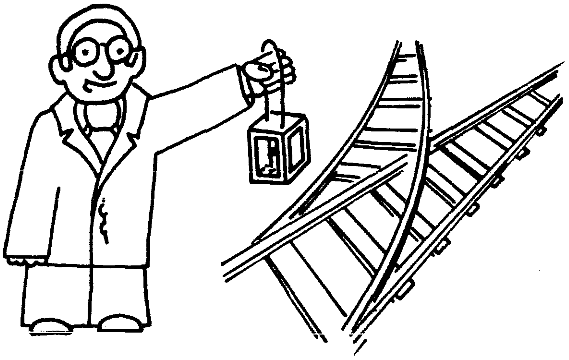
When referring to available PDP-11 literature, you may find that a number of special symbols are used to describe certain features of individual instructions. The more commonly used symbols are explained below.

( )	=	Indicates "the contents of". For example, (R5) means "the contents of R5".
SS or SRC	=	source field
DD or DST	=	Destination field
←	=	"becomes", or "moves into". For example, (dst)←(src) means that the source *becomes* the destination or that the source *moves into* the destination location.
↑ (SP)	=	"popped" or removed from the hardware stack
↓ (SP)	=	"pushed" or added to the hardware stack
∧	=	logical AND
∨	=	logical inclusive OR (either one or both)
∀	=	logical exclusive OR (either one but not both)
~	=	logical NOT

read on ➧

PROGRAMMING TECHNIQUES

BRANCH CONDITIONS



Because branch instructions are used so often, it is quite helpful to know exactly which condition codes are being tested by a specific instruction. All 16 conditional branches are listed below along with the condition code or codes tested by the instruction.

BNE	Branches if Z bit is clear
BEQ	Branches if Z bit is set
BPL	Branches if N bit is clear
BMI	Branches if N bit is set
BCC	Branches if C bit is clear
BCS	Branches if C bit is set
BVC	Branches if V bit is clear
BVS	Branches if V bit is set
BGT	Branches if N and V are both clear Branches if N and V are both set
BGE	Same as BGT but also branches if Z bit set
BLT	Branches if N or V *but not both* are set
BLE	Same as BLT but also branches if Z bit set
BHI	Branches if both C and Z bits are clear
BHIS	Branches if C bit is clear (same as BCC)
BLO	Branches if C bit is set (same as BCS)
BLOS	Branches if C or Z or both are set