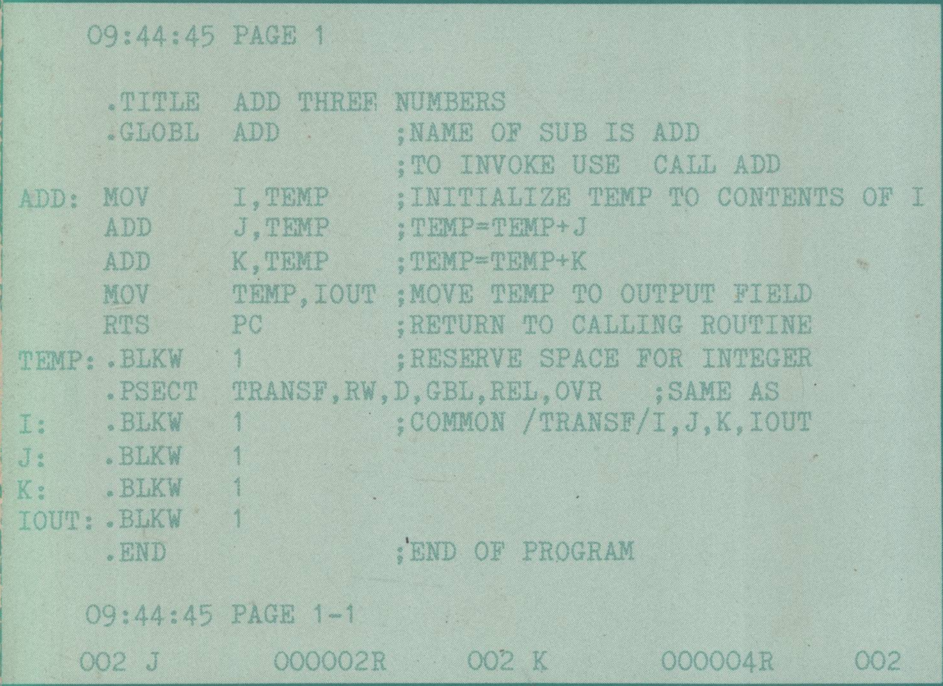
MACRO-11 PROGRAMMING

AND PDP-11 ORGANIZATION



```
     09:44:45 PAGE 1

      .TITLE  ADD THREE NUMBERS
      .GLOBL  ADD         ;NAME OF SUB IS ADD
                          ;TO INVOKE USE  CALL ADD
ADD:  MOV     I,TEMP      ;INITIALIZE TEMP TO CONTENTS OF I
      ADD     J,TEMP      ;TEMP=TEMP+J
      ADD     K,TEMP      ;TEMP=TEMP+K
      MOV     TEMP,IOUT   ;MOVE TEMP TO OUTPUT FIELD
      RTS     PC          ;RETURN TO CALLING ROUTINE
TEMP: .BLKW   1           ;RESERVE SPACE FOR INTEGER
      .PSECT  TRANSF,RW,D,GBL,REL,OVR   ;SAME AS
I:    .BLKW   1           ;COMMON /TRANSF/I,J,K,IOUT
J:    .BLKW   1
K:    .BLKW   1
IOUT: .BLKW   1
      .END                ;END OF PROGRAM

     09:44:45 PAGE 1-1

    002 J         000002R      002 K        000004R      002
```

Gerald W. Cichanowski

MACRO-11 Programming And PDP-11 Organization

Gerald W. Cichanowski

Michigan State University

Engineering Press, Inc. **San Jose, California 95103-0001**

Printed in the United States of America

Library of Congress Cataloging In Publication Data

Cichanowski, Gerald W., 1952-
MACRO-11 programming and PDP-11 organization.

Bibliography: p.
Includes index.
1. PDP-11 (Computer)--Programming. 2.MACRO-11 (Computer programming language) I. Title. II. Title: MACRO-11 programming and P.D.P.-11 organization.
QA76.8.P2C5 1982 001.64'2 82-11498
ISBN 0-910554-38-2

Engineering Press, Inc.
P. O. Box 1
San Jose, California 95103-0001

Preface

This book was written from my lecture notes for a course in assembly language programming, taught at St. Mary's College, Winona, MN 55987. The course is one semester in length and required for Computer Science majors at St. Mary's. The background of the students who take the course consists of a one semester course in FORTRAN programming, and a one semester course, which we call Advanced Programming. In the second course, students complete complex programs as assignments, and are given an overview of the field of computer science. There they are introduced to the topics of assemblers, compilers and operating systems, from a functional, rather than a theoretical or implementational point of view.

Although in the course, the students become familiar with PDP-11 assembler programming, the purpose of this course is not to develop assembler programmers. The course gives the student an understanding of how computer hardware is organized, and how it interacts with software. For those whose only goal is to write MACRO-11 programs, chapter 19 can be omitted, without a loss of needed material.

Chapter 19 is intended to describe in general terms the makeup and operation of computer hardware. It is not intended to describe in detail how the PDP-11's function.

The exercises and examples in the book begin by using MACRO-11 subroutines which are called from FORTRAN mainline programs. This is done for two reasons. First, it allows the discussion of input/output to be delayed until near the end of the book. Secondly, my contention is that on those occasions when writing code in assembler is justified, that code should be written as subroutines which are called from high-level languages.

Contents

Preface v

1. Number Systems 1
2. PDP-11 Data Representation 9
3. Instruction Formats 21
4. PDP-11 Organization 29
5. MACRO-11 35
6. Some Basic Instructions Using Relative Addressing 51
7. Changing Program Flow 63
8. Registers As Accumulators 71
9. EIS Instructions 79
10. The Macro in MACRO-11 89
11. Immediate Addressing 99
12. Instruction Modification And Indexing 105
13. Byte Operations 115
14. Indirect Addressing 121
15. Stacks 131
16. Logical Instructions 139
17. Subroutines 153
18. Input/Output 171
19. Hardware 189
20. Relocation 205
21. Floating Point Operations 211

Appendix

A. PDP-11 Instruction Set 217
B. Addressing Modes 223
C. ASCII Characters 224
D. RSTS/E Directives 225
E. RSTS/E Error Messages 241

Index 245

1

Number Systems

Meaning of Numbers

'Numbers' are symbolized by placing together a series of characters, which we recognize as having numeric value. The standard characters we are used to dealing with are the following; 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. We normally associate the 'values' zero, one, two, etc., with them. When we place more than one character together, say '123', the meaning of the symbols becomes harder to decipher. In order to establish the meaning of the number, we must know what 'base' the number is in. If we go back to our example 123, normally we associate the value one hundred twenty three with it. This is because normally we deal with base ten arithmetic. As a result, the symbols represent values through the following process. Starting from right to left, the value the symbols represent is the sum of 3 x 1 + 2 x 10 + 1 x 100.

To say a number is in a certain base, say b, we are saying two things about the number. First, we are saying that there will be a collection of distinct symbols used to represent numbers in that base. There will be b symbols in that set. For example, a number in base ten is represented by the symbols 0,1, ... 9, likewise a number in base two would be represented by the symbols 0 and 1. Also, by saying a number is in base b, we are saying the value given to the number represented by the symbols is as follows. For some arbitrary base b number;

D(n)D(n-1)...D(2)D(1)D(0).D(-1)D(-2)...D(-m)

The value is as follows.

$$D(n)xb^{n} + D(n-1)xb^{n-1} + \ldots + D(1)xb^{1} + D(0)xb^{0} + D(-1)xb^{-1} + \ldots + D(-m)xb^{-m}$$

Example

1011.011 base 2 has the following value.

$$1x2^{3} + 0x2^{2} + 1x2^{1} + 1x2^{0} + 0x2^{-1} + 1x2^{-2} + 1x2^{-3} = 11.375 \text{ base } 10$$

Converting Between Bases

In the first part we saw that the meaning a group of symbols takes on, is a function of the base used to express the number. Often times, we will have a number in one base, and we will need to express it in another. The most straightfoward way of performing this conversion is the following. When we are evaluating the expression:

$$D(n)xb^{n} + D(n-1)xb^{n-1} + \ldots + D(0)xb^{0} + D(-1)xb^{-1} + \ldots + D(-m)xb^{-m}$$

simply perform the arithmetic operations in the new base. Unfortunately, most people, especially the author, are not adept at performing these types of calculations in a base other than ten. Therefore, this technique is normally only useful for converting a number from a base other than ten to base ten.

Example

45.3 base 6 becomes

$4x6^1 + 5x6^0 + 3x6^{-1} = 29.5$ base 10.

Since the previous method, which we will call the 'expansion' method, is only useful for going to base ten, we will look at another technique, which will be useful for converting from base ten to another. In order to do this, we must split our number into two parts. The first part, which we will call the integer part, is that which is to the left of the radix point. The rest of the number, that to the right of the radix point, is called the fractional part.

To convert the integer part of the number we will use a technique called the 'division/remainder' technique. The process is as follows.

1. In the old base, divide the number by the new base obtaining a quotient and a remainder.

2. The remainder becomes the next digit to the left of the radix point.

3. If the quotient is zero, you are finished, otherwise replace the original number by the quotient and repeat the process starting at step one.

Example

123 base 10 to base 8 is done as follows.

```
123 / 8  = 15 remainder 3
 15 / 8  =  1 remainder 7
  1 / 8  =  0 remainder 1
```

Which yields the following base 8 number, 173.

Example 2

124 base 10 to base 16.

```
124 / 16 = 7 remainder 12
  7 / 16 = 0 remainder  7
```

This yields 7C as the base 16 equivalent of 124 base 10. Note that a base 16 number needs 16 symbols to represent the values in base ten, of zero to fifteen. This is done by using the symbols 0 ... 9 and additionally the symbols A, B, C, D, E and F to represent the values ten through fifteen.

Converting Fractions

The technique we will use to convert the fractional part of a number is the 'multiplication' method and is as follows.

1. In the old base, multiply the original number by the new base, obtaining the following, an integer part and a fractional part.

2. The integer part becomes the next digit to the right of the radix point.

3. If the fractional part is zero, or you have reached the number of places to the right of the radix point you wish to carry, you are finished. Otherwise, replace the original number with the fraction obtained in the multiplication and repeat the process starting at step 1.

Note: a fraction in one base may not have a finite counterpart in another. Unfortunately, our only solution to this problem is to settle for an approximation to the number.

Example

Convert .25 base ten to base two.

```
 .25 x 2 = 0.5
 .5  x 2 = 1.0
```
This yields the base two number .01

Example 2

Convert .2 base ten to base 8.

```
 .2 x 8 = 1.6
 .6 x 8 = 4.8
 .8 x 8 = 6.4
 .4 x 8 = 3.2
 .2 x 8 = 1.6 Note, the pattern will now repeat.
```

This yields the following repeating base eight fraction.

.14631463...etc.

A Shortcut

Very often we will need to convert an integer between base 8 and base 2, or base 16 and base 2. Since in each pair, both bases are a power of two, this conversion becomes very simple. When converting from base 2 to base 8, starting from the right, group the digits into groups of three. Each group of three binary digits now represents one base eight digit. To reverse the process, simply take each octal digit and express it as a group of three binary digits. If you want to convert between base two and base sixteen, use the same procedure, just group the binary digits into groups of four.

Example

Convert 10101111011 base two to base eight.

The groups of digits become

10 101 111 011, which represent the octal digits
2 5 7 3.

Thus the base eight number is 2573.

Example 2

Convert 732 base 8 to binary.

Each octal digit now becomes three binary digits.

7 3 2 becomes 111 011 010, for the binary number 111011010.

Example 3

Convert 10101111011 binary to hexadecimal.

The groups of digits become

101 0111 1011, yielding the base sixteen number 57B.

Notation

Throughout this book, many numerical quantities are used. If the base of the numbers is not obvious from the context, they will be written in the following fashion.

123 (8)

The number in parenthesis indicates the base of the number.

Exercises

1.1 Convert the following base ten numbers to binary.

a. 263

b. 174

c. 25

1.2 Convert the following octal numbers to decimal.

a. 36

b. .732

c. 177

1.3 Convert the base ten numbers in 1.1 to octal and hexadecimal.

1.4 Convert the following binary numbers to octal and hexadecimal.

a. 10110110

b. 11100011

c. 10101010

1.5 Convert the following octal numbers to binary.

a. 176

b. 377

c. 17770

1.6 Convert the following hexadecimal numbers to binary.

a. ABC

b. 1EFO

c. 377

2
PDP-11 Data Representation

The primary storage unit of a computer is a 'bit', BInary digiT. A bit is a unit which is capable of storing one binary digit, 0 or 1. Bits are put together into groups to form 'bytes'. A byte is a group of bits which is capable of storing a single alphanumeric character. On the PDP-11 a byte is a group of eight bits. Bytes in turn are grouped together into units which will hold an integer value. On the PDP-11 this is two bytes or sixteen bits. This group of bits is referred to as a 'word'.

Characters are stored in a computer's memory by coding them as binary numbers. The length of the number will correspond to the length of a byte on the computer we are dealing with. Many computer systems, including the PDP-11, use a standard code for the characters. This code is known as ASCII, see appendix C. The rightmost seven bits of a byte correspond to the ASCII code for a given character. The leftmost bit is reserved as a parity bit. The parity bit is used for error detection. Its use will be explained in a later chapter.

Sign-Magnitude

Integer values in most computers are stored in a single word. For positive integers this becomes simple. Simply code the integer as a binary number, with the number of digits equal to the number of bits in a computer's word. Negative numbers however, present a special problem.

One technique for storing negative numbers, the 'sign-magnitude' notation, is as follows. If a computer has a word of length n, code the integer as a binary number of length n-1. Then in the leftmost bit of the computer word, place a one if the number is negative or a zero if the number is positive. This technique has the advantage that it is very simple to form the negative of a number. A number and its positive counterpart are the same, except for one bit, the leftmost or sign bit. However, there is a major disadvantage to this form of storing integers. The disadvantage becomes evident when we try to add or subtract two numbers. To see the problem, we will work through the procedure of adding two arbitrary numbers. If a and b are two numbers, they are added as follows. First, we look at the sign of the two numbers. If they are the same, we simply add the numbers and give the sum the sign that both a and b had. If a and b do not have the same sign, this becomes more difficult. If this is the case, we must determine which of a and b has the largest magnitude. Then we subtract the magnitude of the smaller number from the magnitude of the larger one. The sum is the result, once it has been given the sign of the number with the largest magnitude.

Example

```
  23           -10           -23          23
 +22          +-11          + 10        +-10
-----        ------        ------       -----
 +45          - 21

                             23           23
                           - 10         - 10
                           ------       -----
                             13           13

                            -13          +13
```

This technique, as you should be able to see, presents the problem, that the process of adding and subtracting two numbers becomes fairly involved. Since we are dealing with computers, we need to consider the difficulty of implementing some form of hardware to perform the operations. Two other forms of representing negative numbers exist. These forms have the advantage that adding and subtracting numbers with mixed signs is much simpler.

One's Complement

The first of these forms is known as 'one's complement' notation. In this form, to store the negative of an integer, we take the positive form of it, add zeroes to the left until we have the number of digits which will fit in one of our computer's words. Then wherever there is a zero, we put a one and wherever there is a one we put a zero.

Example

If we are dealing with three bit numbers, the one's complement of 0, 1, 2 and 3 are as follows.

Number	Complement
000 +0	111 -0
001 +1	110 -1
010 +2	101 -2
011 +3	100 -3

Addition and subtraction of one's complement numbers is very simple. If a and b are two numbers in one's complement notation, to add the numbers, simply ignore the signs and add the numbers. You will obtain a sum the same length as the original numbers and possibly a carry to the left. If there is a carry to the left, simply add it to the original sum. This process is known as 'end-round-carry'. The result will now be of the correct sign and magnitude, assuming we have not generated a sum which is too large to fit in one of our computer's words. To subtract one number from another, simply take the one's complement of it and then add the two numbers.

Example

```
   1          001
 +-2         +101
 ----        ----
  -1          110

   3          011
 +-2         +101
 ----        -----
   1         1)000
                 1
             -----
               001
```