Contents

CHAPTER 1

INTRODUCTION

This book introduces assembly language programming with the TI Home Computer.

1.1 Purpose of the Book

Texas Instruments offers a variety of hardware (equipment) and software to use for running and developing assembly language programs. The software includes the Editor/Assembler package, the line-by-line assembler and the debugger which come with the Mini Memory Module, and the software available with the UCSD p-System.

The specific operation and unique features of these products are described in the documentation that accompanies each one. A lot of this documentation assumes that you have previous assembly language experience and already know the assembly language of the TI Home Computer. If you don't have that kind of experience or knowledge, this book is for you. This book doesn't replace the existing documentation but, rather, supplements it.

The purpose of this book is to help you learn the basic concepts of assembly language programming using the Texas Instruments Home Computer. It's designed to help you learn the TI Home Computer's assembly language instruction set and the structure of assembly language programs. With this knowledge, you can

- understand existing programs

- customize programs

- create your own assembly language programs

With an understanding of assembly language, you begin to know the detailed architecture of the TI Home Computer and can apply your understanding to directly control the computer's programmable components.

language, the translation program is called either an interpreter or a compiler. For an assembly language program, the translation program is called an assembler.

Here are a few "buzzwords" associated with the assembly process. They are:

- source program
- object program
- listing

A "source" program is the collection of assembly language statements which is translated by the assembler. The machine language program that results is called an "object" program. The assembler usually also produces a "listing." A listing is a printed document that shows:

- the assembly language statements that were given to the assembler
- the resulting machine code into which they were translated
- the locations in memory for the machine code
- other information such as a list of symbols used in the program

The listing also contains error messages if the assembler can't understand the assembly language statements, or if for some other reason, it can't produce the correct machine code. You can visualize the assembly process this way.

```
                    ┌──────────┐
                    │  Source  │
                    │ Program  │
                    └────┬─────┘
                         │
                         ▼
                    ┌──────────┐
                    │Assembler │
                    └──┬────┬──┘
                       │    │
              ┌────────┘    └────────┐
              │                      │
              ▼                      ▼
        ┌──────────┐           ┌──────────┐
        │  Object  │           │ Listing  │
        │ Program  │           │          │
        └──────────┘           └──────────┘
```
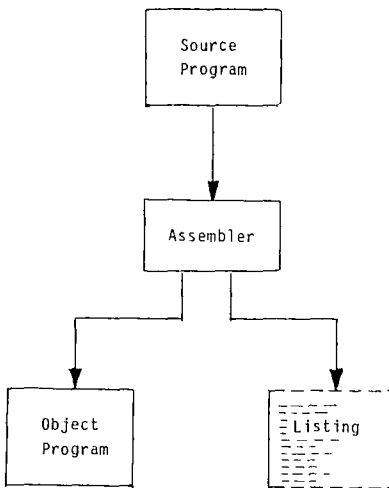
2.4   Procedures for Developing an Assembly Language Program

The steps for developing an assembly language program aren't that much different from the steps used to develop a high-level language program. The steps are as follows:

1. Define the program.
2. Compose the source program.
3. Assemble the source program into an object program.
4. Load the (object) program into memory.
5. Run the program.
6. Test the program.
7. Modify the program.
8. Document the program.

First, define what you want the program to do and how you want it to do its job.

When writing a program in BASIC, you can type in the statements of a program, compose the statements in the right order, and run the program. The BASIC interpreter translates and performs the instructions at one time.

When developing a program in assembly language, you perform these separate steps:

- compose the statements in the source program
- have the source program assembled into an object program
- run the object program

With assembly language, you write the statements of your program and compose them into a source program. Usually, there is an Editor to help in composing the source program. An Editor is a program that lets you type in statements, collect them, and arrange them in the right order. After creating the source program, you use an Assembler to translate the source program into an object program and to produce a listing. Next, you load the object program into memory using a Loader. A loader is a program that reads an object program and stores the object code in memory. Then you run the program.

When you run your program, a Debugger is sometimes available to help you test your program and remove "bugs", or mistakes in the logic of a program. Often, as a result of testing a program, you modify it to fix bugs or change features.

Documentation is an important part of developing any program. By describing a program clearly and completely, you can more easily debug and modify it when necessary. Good documentation helps others understand your program, or, more importantly, helps you understand your program after being away from it.

This description provides an overview of the steps that are used to develop an assembly language program. Other chapters describe these steps in more detail.

2.5 Main Ideas

Computer languages are used to give directions to a computer. There are three levels: high-level language, assembly language, and machine language.

A high-level language is more oriented to human language than machine language. A high-level language is less efficient in terms of the required memory storage and the time required to run the program.

Assembly language is used to express machine language by using characters that people understand.

An assembler is a program that translates the assembler language statements of a source program into the machine code of an object program. The assembler usually produces a printed document called a listing that shows the result of the assembly process.

Chapter 3

THE STRUCTURE OF DATA

An assembly language program, like a high-level language program, consists of a collection of statements. The main purpose of these statements is to give an instruction to the computer or to define data. This chapter examines the structure of data and its relationship to a program.

3.1 Relationship of Data to a Program

Consider the following BASIC language program.

```
100 DATA 3,-8
110 READ A,B
120 C = A + B
```

Statement 100 defines two data items: a value of 3 and a value of −8. Statement 110 assigns the name A to the value 3 and the name B to the value −8. Statement 120 directs the computer to add the value called A and the value called B and call the sum C.

Consider a similar assembly language program for the TI Home Computer.

```
A α    DATA  3
  B    DATA  -8
  C    BSS   2
       MOV   @A,R0
       MOV   @B,R1
       A     R0,R1
       MOV   R1,@C
```

The first three statements define data. The last four statements are instructions that specify an action for the computer to perform.

The first statement assigns the name A to the value 3. The second statement assigns the name B to the value −8. The third statement assigns the name C to a storage location. The BSS is an abbreviation for "Block Starting with Symbol". It reserves a block of memory and assigns a name to the beginning of that block. The 2 in the statement specifies

the number of bytes of memory to reserve. Memory is measured in bytes. A byte is a group of 8 bits.

The fourth statement is an instruction that moves, or copies, the number called A to a register numbered zero. A register is a special storage location that can be accessed faster than other storage locations. The fifth statement is an instruction that moves the number called B to a register numbered one.

The sixth statement is an instruction that adds the number in Register 0 to the number in Register 1 and replaces the number in Register 1 with the sum. The seventh statement is an instruction that moves the number in Register 1 (the sum) to the storage location called C.

The point is, an assembly language program, like a high-level language program, includes statements that define data and statements that direct the computer to perform some action. One step that you must take in assembly language programming is to allocate memory for data and define the structure of that data.

3.2 Bit Quantities

To the computer, all data is simply a collection of one and zero bits. With assembly language, you can directly manipulate individual bits.

The most basic unit of data that a computer can access is a bit. A bit is a single binary digit: a zero or a one. A single bit is usually too small a unit of data to be very useful by itself. More commonly, bits are grouped together to form larger numbers.

Just like a group of 12 doughnuts is called a dozen, names are also given to groups of bits. A group of 8 bits is called a "byte." A group of 4 bits is called a "nibble." Sometimes, it's convenient to refer to a pair of bits by a name. Let's call a two-bit quantity a "niblet" (a petite nibble).

Another name given to a quantity of bits is "word." It's a term given to the maximum number of bits that a computer can handle at one time. The number of bits in a word depends upon the computer. Different computers have different word sizes. If someone asks you how many bits are in a word, you must first know what computer that person is talking about.

Note

The word size of the TI Home Computer is 16 bits.

Sometimes, the expression "double word" is used. Just as you might expect, a double word contains twice as many bits as a word. The number of bits in a double word depends upon the number of bits in a word which, in turn, depends upon the computer.

Here's a summary list of these bit quantities.

Bit	A single binary digit (0 or 1)
Niblet	Two bits
Nibble	Four bits
Byte	Eight bits
Word	The number of bits in a word varies with the computer. For the TI Home Computer, it's 16 bits.
Double Word	The number of bits in a double word depends upon the word size of the computer and equals two times the word size.

3.3 Number Conversions

When writing programs in assembly language, you often deal with word and byte quantities and, sometimes, even smaller quantities of bits. Bits, of course, represent binary numbers. You need to be familiar with the binary number system; because you need to be able to convert a binary number into a decimal value and a decimal value into a binary value.

When reading or writing assembly language programs for the TI Home Computer, you also need to be familiar with the hexadecimal number system. The hexadecimal, or "hex", number system expresses binary values more concisely. For example, rather than writing out a 16-bit number like 1010011110011100, it's more concise to simply write the hexadecimal equivalent value, A79C.

Most of the time, binary numbers are expressed as hex numbers. If you can convert binary numbers to hex equivalents and hex numbers to binary equivalents, it is helpful. Additionally, knowing how to convert hex numbers into decimal equivalents and decimal numbers into hex equivalents is helpful as well.

In summary. knowing how to perform six kinds of number conversions is helpful when learning to program using assembly language. These conversions are:

1. a binary number number to a decimal equivalent

2. a hexadecimal number to a decimal equivalent

3. a binary number to a hexadecimal equivalent

4. a hexadecimal number to a binary equivalent

5. a decimal number to a binary equivalent

6. a decimal number to a hexadecimal equivalent

Let's explore some techniques for performing these number conversions. These are not the only ways to convert numbers, but they'll get you started.

Binary, decimal, and hexadecimal number systems use positional notation. With positional notation, the value of an individual digit in a number depends upon its position in the number.

For example, in comparing the decimal number 735 and the number 357, the 5 digits have different positions and have different values in the two numbers. In the number 735, the 5 digit has a value of 5; in the number 357, the 5 digit has a value of 50. The position of the 5 in these decimal numbers determines its value.

With positional notation, the position of each digit determines its value. To be more specific, the position of a digit determines the power of the radix by which the digit is multiplied. The radix, or the base, of a number system is the number of digits that can be used to express values. For example, the decimal number system has a radix of ten; there are ten digits, 0 through 9, that can be used to express values. The binary number system has a radix of two since there are only two digits, 0 and 1, that can be used to express values.

The value of an individual digit in a number can be determined by this procedure.

- Start at the position of that digit and count the number of other digits to the *right* of it.

- Use this count as an exponent for the radix of the number.

- Multiply the digit times the radix raised to that exponent.

For example, to determine the value of the digit 3 in the decimal number 6357:

- Start at the position of the 3 digit and count the number of digits to the right of it. There are 2 digits to the right of the 3.

```
        1    2
    +--------->
    |
6   3    5    7
```
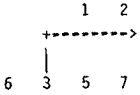
- Use this count (2) as an exponent for the radix of the number. A decimal number has a radix of 10. The number 10 raised to the 2nd power is 100.

```
2  =  10 X 10  = 100         10
```

- Multiply the digit (3) times 100.

```
3 X 100 = 300
```

The value of the digit 3 in 6357 is 300.

Using the same procedure with 6357, you can determine that the value of the 6 digit is 6 10^3, or 6000; the value of the 5 digit is 5 10^1, or 50; and the value of the 7 digit is 7 10^0, or 7.

Note

Any number with a zero exponent equals one. For example,

```
0=1     10
0=1     53
0=1     18927
```

The value of a complete number can be calculated by adding the values of the individual digits. For example, the value of the number 6357 is 6000 + 300 + 50 + 7, or 6357.

Knowing how to perform these calculations with decimal numbers helps you calculate the value of numbers that use other number systems.

3.3.1 Converting a Binary Number to a Decimal Equivalent

The binary number system is the one used by digital computers. The two digits in the binary number system (0 and 1) are used to represent the on/off or true/false states of binary data in a computer.

The binary number system is the "natural" number system for a computer. The natural number system for people is decimal. When confronted with a binary number, you may want to convert it to decimal so you can think about it more easily.

The binary number system uses positional notation just like the decimal number system. You can take advantage of this common element to convert a binary number to an equivalent decimal value. You can use the same technique to evaluate a binary number as you use to evaluate a decimal number.

For example, suppose you want to convert the binary number 10101 into a decimal equivalent. First, determine the value of each digit in the number. Specifically, you only need to determine the value of each 1 digit since the value of each 0 digit is zero.

In the binary number 10101, the leftmost 1 digit has 4 digits to the right of it.

```
    1  2  3  4
+------------>
|
1  0  1  0  1
```
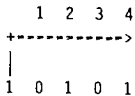
Use this count (4) as an exponent for the radix of the number. The radix of a binary number is 2. The number 2 raised to the 4th power is 16.

```
4  =  2 X 2 X 2 X 2  = 16      2
```

Therefore, the value of the leftmost 1 bit is decimal 16.

The value of the middle 1 bit is 4 and the value of the rightmost 1 bit is 1. The value of the entire binary number is 16 + 4 + 1, or decimal 21.

3.3.2   Converting a Hexadecimal Number to a Decimal Equivalent

The hexadecimal number system is a radix-16 number system. There are 16 unique digits in the hexadecimal number system. The 16 hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. In the hexadecimal number system, the digits 0 through 9 have the same value as the digits 0 through 9 in the decimal number system. The digits A through F represent the decimal values 10 through 15.

The following table illustrates the relationship between a hexadecimal digit and its corresponding binary and decimal equivalent value.

HEXADECIMAL-BINARY-DECIMAL EQUIVALENCY (HBDE) TABLE

Hexadecimal	*Binary*	*Decimal*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Let's refer to this table as the HBDE Table. You can use the HBDE table to determine the equivalent values in the three number systems. For example, to determine the

equivalent decimal value for the hexadecimal digit B, first locate the B in the Hexadecimal column. Then follow across on the same row under the Decimal column and find the equivalent decimal value (11).

The hexadecimal number system is useful because it's a more concise way of expressing binary values. For example, it takes four digits to express the decimal value 10 in binary (1010), but it takes only a single digit (A) to express the same value in the hexadecimal number system.

Just like the binary and decimal number systems, the hexadecimal number system employs positional notation. This commonality helps you convert a hexadecimal number to an equivalent decimal value.

Suppose you want to convert the number hexadecimal 3AD4 to an equivalent decimal value. You can use the same basic procedure you use to convert a binary number to decimal. Start with the leftmost digit and count the number of digits to right of it. There are 3 digits to the right of the leftmost digit.

```
    1  2  3
+--------->
|
3  A  D  4
```
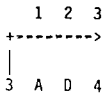
Use this count (3) as an exponent for the radix of the number. The radix of a hexadecimal number is 16. Sixteen raised to the 3rd power is 4096.

```
3 = 16 X 16 X 16 = 4096      16
```

Multiply the digit 3 times 4096.

```
3 X 4096  =  12288
```

Thus, the value of the 3 digit is 12288.

In the same way, evaluate the value of the hex digit A in 3AD4. Use the HBDE Table to find the decimal value for the digit hex A. Its decimal value is 10. Then multiply the decimal value of the digit times 256 (which is 16^2) to determine the value of the digit in the hex number. The value of the A digit in hex 3AD4 is 2560.

```
10 X 256  =  2560
```

Using the same technique, you find the value of the D digit in the hex number is 208 and the value of the 4 digit is 4.

Therefore, the decimal value of the entire hexadecimal number is 12288 + 2560 + 208 + 4 = 15060.

3.3.3   Converting a Binary Number to a Hexadecimal Equivalent

Hexadecimal numbers are more concise than binary numbers. For example, the 4-bit binary number 1100 can be expressed with the single hexadecimal digit C. The concisness of the hexadecimal number system when compared to the binary number system becomes more important as the number of bits in the binary value increase. For example, to express the binary value 1000010011011001 requires 16 bits. The same value can be expressed in hexadecimal with only 4 digits — 84D9.

To convert a binary number to a hexadecimal equivalent, follow these procedures. First, start with the rightmost bit in the number and move toward the left, dividing the bits in the binary number into groups of four. You can add leading zeros to the leftmost group of bits to make the number of bits in that group exactly four.

After dividing the binary number into nibble-sized groups of bits, simply write down the hexadecimal digit that is equivalent to the binary value of each nibble. You can use the HBDE Table for this, or use your memory.

Follow this example to convert the 10-bit binary number 1010100011 into an equivalent hexadecimal number. Start at the right, and moving left-to-right, divide the bits into groups of four.

```
10 1010 0011
```

You can add leading zeros to force the binary number to have an even multiple of four bits.

```
0010 1010 0011
```

Then, using the HBDE Table or your memory, jot down the hex digit that is equal to each nibble.

```
0010 1010 0011
  2    A    3
```

And that's all there is to it.

Use the same procedure and confirm that a binary value of 1001110101101111 is equal to a hexadecimal 9D6F.

3.3.4 Converting a Hexadecimal Number to a Binary Equivalent

Sometimes you may encounter a hexadecimal value (say, from using a debugger to inspect the contents of a memory location) and want to convert the hex value to a binary equivalent. You can perform a hexadecimal-to-binary conversion as follows. Use the HBDE Table or your memory to find the binary nibble that is equivalent to each hexadecimal digit and write down the binary equivalent for each hex digit. You can begin at the left or right hexadecimal digit. Group these nibbles together and you have the binary equivalent.

For example, to convert the hexadecimal number 96C7 to an equivalent binary number, start with the leftmost digit and write down the 4-bit binary equivalent for each hex digit.

```
   9     6     C     7
 1001  0110  1100  0111
```

Hexadecimal 96C7 equals a binary 1001011011000111.

When converting a hexadecimal number into a binary equivalent, it's conventional, though not required, to add enough leading zeros to end up with even multiples of four bits; that is, complete nibbles.

For example, the number hexadecimal 27E6 can be expressed as the 14-bit binary number 10011111100110 (expressed in three-and-a-half nibbles), but it's conventional to add enough leading zeros to end up with whole nibbles. Hexadecimal 27E6 equals binary 0010 0111 1110 0110 (four complete nibbles).

3.3.5 Converting a Decimal Number to a Binary Equivalent

When you want to convert a decimal number to a binary equivalent, you can use the following procedure. To illustrate the procedure, let's convert the decimal number 22 into an equivalent binary value.

Start by making a table like this.

```
|   Radix    |  Decimal   |
+------------+------------+ Remainder |
|     2      |     22     +------------+
|            |            |            |
|            |            |            |
|            |            |            |
|            |            |            |
```

The 2 in the Radix column of the table is the radix of the number system into which the decimal number is being converted. We are converting a decimal number into a binary (radix-2) system. The 22 in the Decimal column is the decimal number to be converted. The column titled Remainder is used to record the remainders from a series of divisions.

Proceed this way. Divide the number in the Radix column, 2, into the number in the Decimal column, 22. Record the quotient under the number in the Decimal column and record the remainder in the same row as the quotient but in the Remainder column.

```
|   Radix    |  Decimal   |
+------------+------------+ Remainder |
|     2      |     22     +------------+
|            |     11     |     0      |
|            |            |            |
|            |            |            |
|            |            |            |
```

Two goes into 22 eleven times with a remainder of 0. The first remainder is the *rightmost* digit of the converted result.

Continue the process by making the quotient of the first division, 11, a dividend for a subsequent division. Divide the new dividend by 2 and, again, record the quotient and the remainder from this second division.

```
|   Radix    |  Decimal   |
+------------+------------+ Remainder |
|     2      |     22     +------------+
|     2      |     11     |     0      |
|            |      5     |     1      |
|            |            |            |
|            |            |            |
```

Two goes into 11 five times with a remainder of 1. The second remainder,1, is the *next digit to the left* in the converted result.

Use this second quotient as the new dividend for a subsequent division by 2 and continue the procedure, recording quotients and remainders as you perform the divisions.

When a division produces a quotient of zero, stop. At that point, the last remainder is the *leftmost* digit of the converted result.

```
|  Radix   |  Decimal  |
+-----------+-----------+ Remainder |
|    2     |    22    +-------------+
|    2     |    11    |   0      |  ------------------+
|    2     |     5    |   1      |  ----------------+  |
|    2     |     2    |   1      |  ------------+   |  |
|    2     |     1    |   0      |  --------+   |   |  |
|          |     0    |   1      |  ----+   |   |   |  |
|          |          |          |      |   |   |   |  |
                                       v   v   v   v  v
                                       1   0   1   1  0
```

Notice that the remainders are written down from right-to-left where the first remainder is the rightmost digit of the converted result.

Thus, a decimal 22 equals a binary 10110.

You can use three leading zeros, if you like, to express the byte value of 00010110.

3.3.6 Converting a Decimal Number into a Hexadecimal Equivalent

Converting a decimal number into an equivalent hexadecimal value follows the same basic process as converting a decimal number into a binary number. The major difference is that you divide the decimal number by 16, rather than by 2. You use 16 because you are converting the decimal number into a radix-16 number.

To convert the decimal number 27823 into a hexadecimal number, use the following procedure. Begin by making a table similar to the one used for a decimal-to-binary conversion.

```
|  Radix   |  Decimal  |
+-----------+-----------+ Remainder |
|    16    |   27823  +-------------+
|          |          |            |
|          |          |            |
|          |          |            |
|          |          |            |
```

Notice that the number in the Radix column is 16, rather than 2, because you're converting to a radix-16 number. The decimal number to be converted is in the Decimal column and the Remainder column is used to record the remainders resulting from a series of divisions.

Proceed as in a decimal-to-binary conversion. Divide the number in the Radix column into the number in the Decimal column. Record the quotient under the number in the Decimal column. Record the remainder in the same row as the quotient but in the Remainder column.

```
|  Radix  |  Decimal  |
+---------------+-----------------+ Remainder |
|   16    |   27823   +-------------+
|         |    1738   |    15     |
|         |           |           |
|         |           |           |
|         |           |           |
|         |           |           |
```

Sixteen divided into 27823 produces a quotient of 1738 with a remainder of 15. The first remainder represents the *rightmost* digit of the converted result. Since 15 is the *decimal* value of the remainder, write down the equivalent hexadecimal digit for the remainder.

```
|  Radix  |  Decimal  |
+---------------+-----------------+ Remainder |  Remainder
|   16    |   27823   +-------------+    (Hex)
|         |    1738   |    15     |  =   F
|         |           |           |
|         |           |           |
|         |           |           |
|         |           |           |
```

Continue the procedure by taking 1738 as the dividend for a subsequent division by 16. Record the quotient and remainder from the second division.

```
|  Radix  |  Decimal  |
+---------------+-----------------+ Remainder |  Remainder
|   16    |   27823   +-------------+    (Hex)
|   16    |    1738   |    15     |  =   F
|         |     108   |    10     |
|         |           |           |
|         |           |           |
|         |           |           |
```

The second remainder of 10 represents the decimal value of the next digit to the left in
the converted result. Express this decimal value as an equivalent hex digit.

```
|   Radix   |  Decimal  |
+-----------+-----------+ Remainder |  Remainder
|    16     |   27823   +-----------+    (Hex)
|    16     |   1738    |    15     |  =   F
|           |    108    |    10     |  =   A
|           |           |           |
|           |           |           |
|           |           |           |
```

Use this new quotient as a dividend for a subsequent division by 16 and continue the
procedure, recording the quotients in the Decimal column and remainders in first decimal
and then hexadecimal form as you perform the divisions.

Stop when a division produces a quotient of zero. At that point, the last remainder
represents the leftmost digit of the converted result.

```
|   Radix   |  Decimal  |
+-----------+-----------+ Remainder |  Remainder
|    16     |   27823   +-----------+    (Hex)
|    16     |   1738    |    15     |  =   F  -----------+
|    16     |    108    |    10     |  =   A  --------+  |
|    16     |     6     |    12     |  =   C  -----+  |  |
|           |     0     |     6     |  =   6  ---+  |  |  |
|           |           |           |           |  |  |  |
|           |           |           |           V  V  V  V
```

Notice that the hexadecimal digit remainders are written down from right-to- left just
like the remainders for a decimal-to-binary conversion.

Therefore, decimal 27823 equals hexadecimal 6CAF.

3.3.7   Number Conversion Shortcuts

As you perform more number conversions, you discover shortcuts that make the process
faster.

You can use the Debugger with the Editor/Assembler Package to convert hexadecimal
numbers to decimal and decimal numbers to hexadecimal.

An even easier way to perform these conversions is to use the Texas Instruments calculator that performs the conversions by simply pressing a few keys. The calculator is called the TI Programmer.

3.4 Data Representation

To the computer, all data is simply a collection of one and zero bits. But these bits can represent different things. The data in a program can represent numeric values, characters, or special codes. For example, in the BASIC statement

```
PRINT A;"B"
```

The A represents a numeric value and "B" is a character.

The same is true with assembly language; data can represent numeric values, characters, or special codes. For example, in the following assembly language statements, A represents a numeric value and B represents the character "B."

```
A   DATA 3
B   TEXT 'B'
```

Sometimes data may represent a special code unique to a program. In a payroll program, for example, the number 33 may mean "overtime."

The information represented by a data quantity depends upon the interpretation of that data quantity. For example, consider the binary byte value 01000001. It might represent a numerical value, a character, or it could be a code meaning a size 8 green Stetson with a polka dot hatband.

3.4.1 Data Representing Numbers

If a data value does represent a number, the number may be an unsigned value or a signed value. For example, if the 16-bit quantity 1111 1111 1111 1011 represents a number and you want to know the decimal value of the number, you can't proceed until you know whether this quantity represents an unsigned or a signed value. Its unsigned (or absolute) value is decimal 65531, but its signed value is -5.

If a binary number represents a signed value, the value is represented in *two's complement notation*. Two's complement notation is the most common way for computers

25

to represent signed numbers. With two's complement notation, positive numbers are expressed as their absolute value, but negative numbers are expressed as the two's complement of their absolute value.

The two's complement of a binary number is the result of taking the one's complement and adding one. The one's complement is the result of inverting (changing the state of) the bits.

As an example, let's form the two's complement of the 16-bit binary number 0000 0000 0000 0110 (the absolute value is decimal 6).

First, form the one's complement by inverting the bits.

```
1111 1111 1111 1001
```

Then, form the two's complement by adding one to the one's complement.

```
  1111 1111 1111 1001        the one's complement
+ 0000 0000 0000 0001        plus one
  ---------------------
  1111 1111 1111 1010        equals the two's complement
```

Taking the two's complement of a number results in a number of equal absolute value, but of opposite sign.

For example, if a binary 0000 0000 0000 0110 is a positive 6, then the two's complement, 1111 1111 1111 1010 represents a negative 6.

There are some rules to observe with two's complement notation. Remember that these rules apply only to signed numbers. If the number doesn't represent a signed number, you don't even have to think about the rules. But if the number does represent a signed number, here are the rules.

The sign of the number is indicated by the leftmost bit, called the "sign bit." A positive number has a zero sign bit and a negative number has a sign bit of one.

```
Sign Bit
   0      =    Positive
   1      =    Negative
```

A positive number with a sign bit of zero represents the absolute value of the number directly. For example, the 16-bit number 0000 0000 0000 1001 has a sign bit of zero and an absolute value of decimal 9. A binary 0000 0000 0000 1001 equals +9.

With a negative number that has a sign bit of one, you must take the two's complement of the number to determine its absolute value. For example, the 16-bit number 1111 1111 1111 0111 has a sign bit of one. It's a negative number, so you must take the two's complement of the number to find out its absolute value.

```
    1111 1111 1111 0111      the number

    0000 0000 0000 1000      the one's complement
 +  0000 0000 0000 0001        plus one
    -----------------------
    0000 0000 0000 1001      equals the two's complement (decimal 9)
```

A binary 1111 1111 1111 0111 equals −9.

A quicker way to take the two's complement of a binary number is to start with the rightmost digit and move to the left, writing down the 0 bits until you come to the first 1 bit. Write down the 1 bit and then invert the rest of the bits to its left. Inverting a bit means to change a zero to one and a one to zero.

Look at this example of how to take the two's complement of a binary 0111 0000 1010 0000.

```
                Moving left-to-right,  (1)
                <--------------------
                0111 0000 1010 0000
                |           |||   |
                +----V-----+|+-->V-+
            (4)    invert   | jot down       (2)
                  these bits | these zero bits
                             |
        jot down this one bit <----+ (3)

    The two's complement of    0111 0000 1010 0000   is
                               1000 1111 0110 0000.
```

With the TI Home Computer, most binary values are expressed as hex equivalents. When you have a hex number that represents a signed number, you can tell the sign of the number by the leftmost hex digit. If the hex digit is 0 through 7, the number is positive. If the hex digit is 8 through F, the number is negative.

For example, if the value hexadecimal C3D2 represents a 16-bit signed number, the number is negative since the leftmost hex digit is greater than 7.

If the value hexadecimal B8A represents a 16-bit signed number, the number is positive since the leftmost hex digit is smaller than 8. Remember that it takes four hex digits to

express a full 16-bit value. Therefore, a leading zero must be attached to B8A for it to represent a 16-bit value, thus B8A equals 0B8A.

Since binary numbers are often expressed in hexadecimal, it is usually more convenient to work directly with the hex digits when taking the two's complement of a number. Here's how to take the two's complement of a number expressed in hexadecimal.

1. Beginning with the rightmost digit and moving to the left, write down any zeros until you come to the first nonzero digit.

2. Subtract the decimal value of this first nonzero digit from 16.

3. Subtract the decimal value of the remaining digits to the left from 15.

4. Record the differences as hex digits.

Here's an example of how to take the two's complement a number expressed as hexadecimal 70A0.

```
    <--------------   Moving left-to-right                   (1)
    15  15  16
 -   7  -0  -A   0
    ------------------
     8   F   6   0
     |   |   |   |
     |   |   |   +--  Write down this zero                   (2)
     |   |   +------  Subtract the first nonzero digit from 16  (3)
     +---+----------  Subtract remaining digits from 15      (4)
```
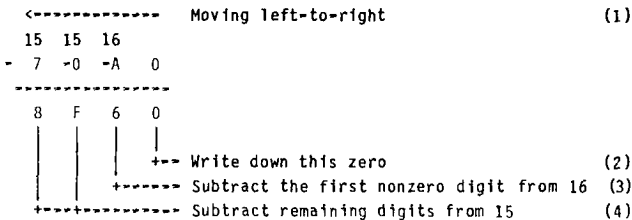
The two's complement (expressed in hex) is 8F60.

When the TI Home Computer is executing instructions that use binary numbers, it handles the numbers in the same whether they represent unsigned or signed values. It's up to the logic of the program to define whether the numbers are signed or not.

Likewise, the computer treats the binary values as integers. It's up to the logic of the program to define data as non-integer numbers.

The unsigned value of a number is called its "logical" value; the signed value is called its "arithmetic" value. For example, the number hexadecimal FFED has a logical value of decimal 65,517 and an arithmetic value of decimal −19.

3.4.2 Data Representing Characters

If a data item represents a character, very likely that character is expressed in ASCII code. ASCII, an abbreviation for American Standard Code for Information Interchange, is the most commonly used code among microcomputers for representing character data.

Each of the 128 characters in the ASCII character set is assigned a unique seven-bit code. The characters and their codes are listed in the ASCII Character Table in Appendix C.

Turn to the ASCII Character Table in Appendix C and you'll find the letter "A" (capital A) has a seven-bit ASCII character code of binary 1000001. If you place a leading zero with the seven-bit code, it becomes the binary byte value 01000001, or a hexadecimal 41. The binary value 01000001, as an ASCII character, represents a capital A.

Most of the time, an ASCII character is expressed as a byte value (8 bits) where the most significant (left-most) bit is called the parity bit. The parity bit is sometimes used for error checking purposes when characters are transmitted between data processing devices over communication lines.

Note

When ASCII characters are discussed in this book, you can assume they're 8-bit character codes where the parity bit is zero.

While looking at the ASCII character table, notice that the 128 characters include both printable and non-printable characters. For example, the character capital B (binary code of 01000010 or hexadecimal 42) is a printable character. However, the character ETX (End of TeXt) is a non- printable character. The ETX character (binary code 00000011 or hexadecimal 03) is a character sometimes used to indicate the end of the text portion of a message that is transmitted over communication lines.

Two non-printable characters that are used often are the carriage return (CR) and line feed (LF) characters. When CR is sent to a terminal such as a printer or a video display, the CR character usually causes the carriage or the cursor to return to the left margin. When LF is sent to a terminal, the LF character usually causes the carriage or cursor to move down to the next line.

Notice that the ASCII character codes for the digits 0 through 9 have sequential values. The character code for "0" is a binary byte value of 00110000 (hex 30), the character code for "1" is a binary value of 00110001 (hex 31), the character code for "2' is a binary byte value of 00110010 (hex 32), and so forth.

The ASCII character codes for the upper-case (capital) letters have sequential values also. The character code for "A" is a binary byte value of 01000001 (hex 41), the character code for "B" is a binary byte value of 01000010 (hex 42), the character code for "C" is a binary byte value of 01000011 (hex 43), and so forth.

Likewise, the ASCII character codes for lower-case letters have sequential values.

3.5 Constants and Variables

In a program, the data can be either a constant or a variable. For example, in the BASIC statement

```
A = 3
```

the 3 is a constant and the A is a variable. The value of 3 is constant; it's always 3. The value of A, however, is variable; its value can change. You can reassign the value of A to 4 as follows.

```
A = 4
```

In the same way, data in an assembly language program can be a constant or a variable. For example, in the assembly language statement

```
A   DATA 3
```

the 3 is a constant and the A is a variable. Specifically, A is the name of a location that contains the value of 3. You can reassign the value of A by putting a different value into that location. One method to reassign the value of A is to use this instruction.

```
MOV RD,@A
```

The instruction replaces the value in location A with a copy of the value in Register 0. If Register 0 has a 4 in it, then the variable A has the value 4.

Character data in a program can also be either constant or variable. For example, in the BASIC statement

```
A$ = "FUDGE"
```

the characters FUDGE are constants and the A$ is a variable.

Likewise, characters in an assembly language program can be constant or variable. For example, the assembly language statement

```
A    TEXT  'FUDGE'
```

assigns the characters FUDGE to the variable A. Specifically, A is the name of the beginning of a series of consecutive memory bytes whose contents are the ASCII character codes for the characters F, U, D, G, and E.

3.6 Main Ideas

This chapter discusses the role of data in a program. All data is represented as binary digits. For convenience, bits are commonly grouped into larger quantities: niblets, nibbles, bytes, words, and double words.

The word size of the TI Home Computer is 16 bits.

Values can be expressed in different number systems. You need to be able to convert values between the binary, hexadecimal, and decimal number systems.

Like high-level languages, data in an assembly language program can represent numbers, characters, or special codes. Character data is usually expressed in ASCII character code as an 8-bit (byte) value with a zero parity bit. Data in a program can be constant or variable.

Chapter 4

THE STRUCTURE OF THE TI HOME COMPUTER

Assembly language is a civilized form of machine language. Using assembly language provides you with precise control of a computer. A specific assembly language reflects the architecture of a specific computer. The assembly language of the TI Home Computer fits the architecture of that computer.

This chapter describes the basic structure of the TI Home Computer and introduces the specific computer parts of importance.

4.1 The Parts of a Computer System

A computer system has three main parts:

1. an input/output section
2. a memory
3. a central processing unit

Although these parts are not always clearly distinguishable, they must all be present in a complete system.

4.1.1 The I/O Section

The input/output section includes devices for sending and retrieving information in and out of the computer system. Some examples of input/output devices are the keyboard, a video display, and a disk drive to name some common devices.

4.1.2 Memory

Every computer system has memory. The computer uses memory to store programs and other data. The computer memory within the computer is of two major types: read-only memory, or ROM, and read/write memory,or RAM.

ROM contains programs and data that cannot be changed. The computer can read the information in ROM but cannot write data into this kind of memory. (There are two kinds of ROM in the TI Home Computer, ROM and GROM, but the distinction is not that important at this point.)

The computer can read from and write to RAM memory. RAM holds programs and data that have been loaded from I/O devices. RAM stores data produced by a program. The TI Home Computer has two kinds of RAM: VDP RAM and CPU RAM. VDP RAM stores information that is displayed on the video screen, and it also stores BASIC language programs. CPU RAM is the read/write memory that the central processor unit (CPU) accesses directly. When an assembly language program's object code is loaded into memory, it must be loaded into CPU RAM.

The TI Home Computer console has over 16 thousand bytes of VDP RAM but only 256 bytes of CPU RAM. Additional CPU RAM is needed for assembly language programs. The TI Memory Expansion Card and the Mini Memory Module contain CPU RAM which can be used with assembly language programs.

A byte is the smallest addressable unit in memory. With the TI Home Computer, each byte of memory has an address. Many of the instructions in the TI Home Computer's instruction set access an individual byte. Most of the instructions, though, access a whole word of memory at a time.

A 16-bit word consists of two 8-bit bytes. A word looks like this.

```
Bit Position:  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
              <------- MSB -------> <------- LSB ------->
                 (Left Byte)          (Right Byte)
```
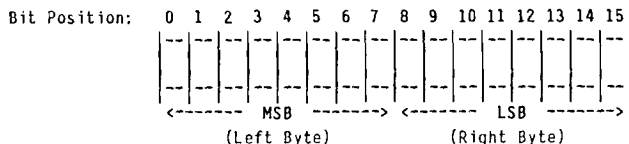
The left byte in a word is named the MSB or Most Significant Byte and the right byte is named the LSB or Least Significant Byte. Notice how the bits are numbered in a word. The leftmost bit is numbered 0 and the rightmost bit is numbered 15.

A memory word contains two bytes. Each byte has its own address; each word has an address. The address of a word and the address of the left byte are the same.

Here's a chart that illustrates the addresses of the first few words in memory.

```
Left Byte             Word           Right Byte
 Address            Address            Address
                +----------------------+
     0          |   Word Address 0     |      1
                +----------------------+
     2          |   Word Address 2     |      3
                +----------------------+
     4          |   Word Address 4     |      5
                +----------------------+
     6          |   Word Address 6     |      7
                +----------------------+
                     .        .        .
                     .        .        .
                     .        .        .
                +----------------------+
    124         |  Word Address 124    |    125
                +----------------------+
    126         |  Word Address 126    |    127
                +----------------------+
                     .        .        .
                     .        .        .
                     .        .        .
```

The first word contains two bytes. The left byte's address is 0 and the right byte's address is 1. The first word's memory address is 0, the same address as the left byte.

The second word's memory address is 2. The second word in memory contains two bytes: a left byte with an address of 2 and a right byte with an address of 3.

Word addresses are numbered by twos (0, 2, 4, 6, etc.) and a word address is always an even number. The left byte of a word is an even number and the right byte of a word is an odd number.

4.1.3   The Central Processing Unit

The central processing unit (CPU) controls a computer system.

All computers perform basically the same operations, but each computer does them differently. The TI Home Computer's CPU utilizes a 9900 family microprocessor chip

that unique characteristics. Since assembly language allows you to control the CPU directly, it's helpful to know the characteristics of its operation. You don't need to know enough to be a computer designer, but enough to understand what you can control in assembly language.

The TI Home Computer's CPU is a 16-bit microprocessor. This means that it can handle 16 bits of data at one time and has word size of 16 bits. It can also operate with byte-sized quantities.

There are three internal registers in the CPU:

- the Program Counter

- the Workspace Pointer

- the Status Register

The Program Counter (PC) is a special register that contains the address of the next instruction to be performed. Before running a program, the Program Counter is loaded with the address of the first machine code instruction. As each instruction is performed, the CPU automatically adjusts the address in the Program Counter to the next address following the current instruction's machine code. As shown in the following illustration, the Program Counter points to the next instruction to be performed by the CPU.

```
                                   Instructions
                          +--------------------------+
                          |    ------------------     |
     Program Counter  ----|    ------------------     |
         (PC)             /    ------------------     |
                          ------> ------------------   |
                          |    ------------------     |
                          |                           |
                          |            .              |
                          |            .              |
                          |            .              |
                          |    ------------------     |
                          +--------------------------+
```
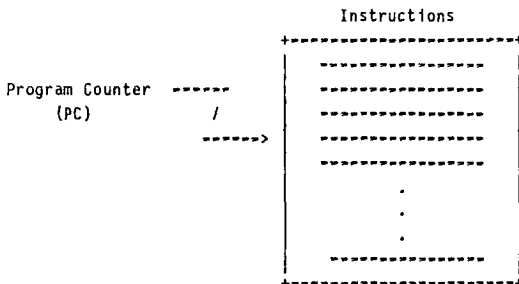
The second internal register is the Workspace Pointer (WP). The Workspace Pointer, like the Program Counter, is simply a register that contains an address. The Workspace Pointer contains the address of a program's workspace. A workspace is a special area of memory whose contents a program can access faster than the rest of memory. A workspace consists of 16 words of memory. Each of the 16 words is called a "working register" or, more often, is referred to as simply a "register". The first word in a workspace is numbered 0 and is named Register 0. The second word is numbered 1 and is named Register 1. The third

word is Register 2 and so forth. The sixteenth word is Register 15. The names and order the the registers in a workspace are shown below.

```
                                Workspace

                    +--------------------------+
          +-->      |      Register  0         |
          |         +--------------------------+
          |         |      Register  1         |
          |         +--------------------------+
Workspace Pointer --+      Register  2         |
     (WP)           +--------------------------+
                                 .
                                 .
                                 .
                    :--------------------------+
                    |      Register 15         |
                    :--------------------------+
```
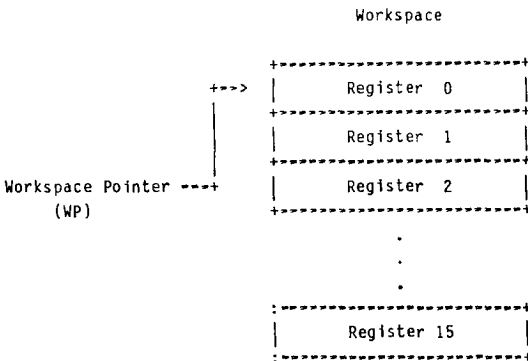
Every program requires a workspace and the CPU has to know the location of the workspace. The CPU uses the address in the Workspace Pointer for the workspace location. Specifically, the address in the Workspace Pointer tells the CPU the location of the first word of the workspace (Register 0) and the CPU knows that Registers 1 through 15 are located in the next 15 words of memory.

Most computers have working registers within the CPU itself. One of the most unique features of the TI Home Computer is that working registers are located in RAM.

For the most part, it's advantageous to have a program use the registers in the workspace to hold the data used by instructions as much as possible. The CPU can access the data in registers faster than data in other areas of memory. The memory outside a workspace is called "general" memory. In addition, using the registers to hold data reduces the amount of memory required to define an instruction's machine code.

The third internal register is the Status Register. The Status Register (SR) holds the individual status bits that are affected by the performance of instructions. Most instructions affect one or more status bits. The status bits are a record of the results of the last instruction.

For example, one of the status bits is the Equal (EQ) status bit. When the CPU performs an add operation, it automatically compares the sum to zero. If the sum is equal to zero, the CPU sets the Equal status bit to one. If the sum is not equal to zero, the CPU clears the Equal status bit to zero.

The state of the status bits can be tested by the "conditional jump" instructions to allow a program to make decisions about what to do next, based upon status conditions resulting from previous instructions.

As a brief example, in BASIC, these statements

```
C = A + B
IF C = 0 THEN 650
```

add the variable A to the variable B, assigns the sum to variable C, and performs a transfer of program control to statement 650 if the variable C equals zero.

In assembly language, these statements

```
A    R0,R1
JEQ  NULL
```

add a number in Register 0 to a number in Register 1, store the sum in Register 1, and perform a transfer of program control to an instruction labeled NULL if the sum equals zero.

When the CPU performs the first instruction (an add operation), it automatically compares the sum to zero and either sets or clears the Equal status bit. The second instruction, Jump if Equal, tests the state of the Equal status bit and performs a transfer of control if the Equal status bit is set to one.

The Status Register, like the Program Counter and the Workspace Pointer, is a 16-bit register. Not all of the bits in the Status Register are used, however.

The Status Register looks like this.

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|L>|A>|EQ|CY|OV|OP|X |  |  |  |  |  |I0|I1|I2|I3|
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
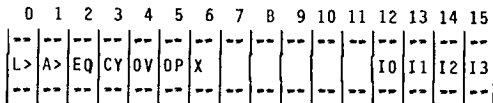
This list names the status bits in the Status Register, their abbreviations and their bit positions.

Name	Abbreviation	Bit Position
Logical Greater Than	L>	0
Arithmetic Greater Than	A>	1
Equal	EQ	2
Carry	CY	3
Overflow	OV	4
Odd Parity	OP	5
Extended Operation	X	6
Not Used	—	7-11
Interrupt Mask	I0-I3	12-15

The ways these status bits are affected by the instructions are explained in the chapters that describe the detailed operation of the instructions and in the instruction summaries in Appendix A.

One of the CPU's input/output ports is the Communication Register Unit (CRU). The CRU is one way that the CPU controls the operation of devices attached to the computer. You can use assembly language instructions to directly control the CRU.

There are some other components inside the computer console that assist the CPU. The TI Home Computer has a video display processor (VDP) to handle the detailed work of displaying information on the video screen, a special component, named a sound generator, to handle producing sound, and other special components for various other functions. All of these components are accessed by instructions to the computer's CPU.

This book provides information for using the instructions that control the CPU.

4.2 Main Ideas

This chapter introduces and describes the basic structure of the TI Home Computer. Other chapters provide more insight into the computer's architecture.

A complete computer system has three parts:

- an I/O section
- a memory
- a central processing unit (CPU)

There are two main kinds of memory: ROM (or read-only memory) and RAM (read-and-write memory). The TI Home Computer has two kinds of RAM: VDP RAM and CPU RAM. When machine code is loaded into RAM, it must be loaded into CPU RAM before it can be performed.

A byte is the smallest addressable unit in memory. Each byte has an address. There are two bytes in each of the TI Home Computer's 16-bit words. The most significant byte in each word has an even address; the least significant byte has an odd address.

The CPU performs instructions. The CPU has three registers of special interest. The Program Counter (PC) remembers the address of the next machine code instruction to be performed. The Workspace Pointer (WP) contains the address of a program's workspace. The Status Register (SR) contains individual status bits that record results from the performance of instructions.

The CRU is a part of the CPU. The CRU is one way that the CPU controls the operation of devices attached to the computer.

Chapter 5

ANATOMY OF ASSEMBLY LANGUAGE STATEMENTS

Most program statements specify an action for the computer to perform or define data. There are some statements that simply make comments and others that give directions to the assembler. This chapter describes how assembly language statements are structured and dissects the statements in a short program.

5.1 Statement Fields

An assembly language statement can contain up to four fields, or groups of information:

- label

- instruction operation code or an assembler directive

- operand(s)

- comment

In a statement, the fields appear in this order:

```
Label      Op-Code or Directive      Operand    Comments
```

A label appears first in a statement, followed by the operation code or assembler directive, followed next by the operand(s), and, finally, the comment.

Although an assembly language statement can contain four fields of information, not all statements use all four fields.

5.1.1 Label Field

The first field is a label. A label names the statement. A label is required only when you want to refer to the statement from another statement.

5.1.2 Instruction Operation Code or Assembler Directive Field

The second field contains either an instruction operation code or an assembler directive. An instruction operation code identifies an operation for the computer to perform. An assembler directive directs the assembler to do something when the program is assembled. Almost every statement has either an instruction operation code or assembler directive.

5.1.3 Operand Field

The third field is the operand field. The operand field identifies the data for an instruction operation or it gives additional information for an assembler directive. Most statements require one or more operands.

5.1.4 Comment Field

The last field is the comment field. The comment field is used to document the statement. It contains information for someone reading the source program. It helps that person understand the program. The comment field is optional but highly recommended.

5.2 Program Example

Let's examine the statements in a program which simply add two numbers together and save the sum.

In a BASIC program you might use these statements.

```
100  X = 2
110  Y = 3
120  Z = X + Y
```

In TI Home Computer assembly language, a similar program that does the same thing looks as follows:

```
START   MOV  @X,R0        PUT X VALUE IN REGISTER 0
        MOV  @Y,R1        PUT Y VALUE IN REGISTER 1
        A    R0,R1        ADD X AND Y
        MOV  R1,@Z        SAVE SUM IN Z
        BLWP @0           EXIT PROGRAM
```

```
X       DATA 2              X EQUALS 2
Y       DATA 3              Y EQUALS 3
Z       DATA 0              Z EQUALS SUM OF X + Y
        END
```

Let's analyze the assembly language program one statement at a time.

First, locate the statement that adds the X and Y values. (The comments to the right of the statements can help you find it.) The third statement adds the two numbers. Look at the statement in more detail. The statement is

```
A    R0,R1          ADD X AND Y
```

The statement has no label. The A is an instruction operation code (often called "op-code"). The A is an operation code for the "Add Words" operation. In assembly language terminology, the A is called a "mnemonic" operation code. The word, mnemonic, is from a Greek word meaning memory aid. A mnemonic helps you remember that A means Add Words.

In the operand field of the Add Words instruction, there are two operands. The first operand is R0; the second is R1. Notice that the two operands are separated by a comma. The operands specify what data to use for the Add Words operation. The operands don't specify the data directly, but, rather, the location of the data.

The purpose of the Add Words instruction is to add two numbers together. The operand field identifies the location of the two numbers. The first number is located in R0 (Register 0). The second number is located in R1 (Register 1). The Add Words instruction adds the two numbers together and puts the sum into the second operand.

The Add Words instruction has a comment field. The comment is ADD X AND Y. It tells you what the instruction does.

When this Add Words instruction is performed, it adds the two numbers in Register 0 and Register 1 and places the sum into Register 1. The value contained in Register 1 before the instruction was performed is replaced by the sum. What happens to the number in Register 0? Nothing. It's still in Register 0.

Before performing the Add Words instruction, the two numbers must be in R0 and R1. The two instructions listed before the Add Words instruction put the two numbers into the registers. The first instruction in the program is a Move Word instruction. The statement is

```
START   MOV  @X,R0          PUT X VALUE IN REGISTER 0
```

START is the label. It's the name given to this statement. The mnemonic operation code is MOV which stands for "Move Word." There are two operands in the operand field. The first operand (@X) identifies the location of the data. The second operand (R0) identifies the location to move the data. The data is moved, actually, it's copied, from one location to another. The first operand (@X) identifies the location of the data to be moved. The at sign (@) means that the location is a general memory location rather than a register or some device attached to the computer. The specific location in memory where the data is located is called X. The second operand (R0) identifies where the data is copied. It's copied into Register 0.

The comment field (PUT X VALUE IN REGISTER 0) tells you what the instruction does.

The second statement in the program is also a Move Word instruction. It causes the data in general memory location Y to be moved where? I hope you said Register 1 (or R1).

The purpose of the two Move Word instructions is to put the two numbers into R0 and R1 so the Add Words instruction can add them.

At this point, you might want to know the reason for putting numbers in registers before adding them. As a matter of fact, you don't have to. You can add the two numbers directly in memory. The purpose for discussing this program is that it helps you understand the difference between addressing data in registers and data in "general" memory.

After the Add Words instruction is performed, the sum is in R1. The instruction immediately after the Add Words instruction is another Move Word instruction which moves the contents of R1 (R1 has the sum) into the contents of memory location Z.

The instruction following this Move Word instruction has a mnemonic operation code of BLWP. The "op-code" stands for Branch and Load Workspace Pointer. It's an instruction that allows you to exit the program. If you write a program in BASIC, you can put an END statement to exit a program. In the same way, the Branch and Load Workspace Pointer instruction is one way to exit an assembly language program. The operand with the BLWP instruction identifies where the exit is. The comment with the BLWP instruction tells you what the instruction does.

Following the Branch and Load Workspace Pointer instruction, the last four statements each contain an assembler directive. A directive does not cause the computer to perform some action when the program runs. A directive gives directions to the assembler when the source program is assembled into an object program.

DATA is an assembler directive rather than an instruction operation code. The DATA directive directs the assembler to reserve a word of memory. Each DATA directive in this program has a label. The labels tell the assembler what to name the words of memory. The operand with each directive tells the assembler what number to put in the word of memory. The comments tell you what the statements do.

The first DATA directive tells the assembler to reserve a word of memory, name that memory location X, and put a 2 in the location. The second DATA directive tells the assembler to reserve a word of memory, name the memory location Y, and put a 3 in the location. The third DATA directive tells the assembler to reserve a word of memory, and name it Z. What number is the assembler to place in the location called Z? Right, a zero.

It's not really important what number is placed in location Z. The DATA statement is a way to make sure a memory location named Z is reserved. When the program runs, it stores the sum in place of the number originally in Z.

One of the program characteristics to note is that it allocates space for all the data values the program uses or produces. In BASIC, you can simply write

```
C = A + B
```

and the BASIC interpreter will find some place for the variable C automatically. In assembly language, however, you must reserve data space explicitly.

The last statement (which has no label) contains the assembler directive END. The END directive tells the assembler that this statement ends the program.

In BASIC, the END statement tells the BASIC interpreter to stop running a program, but in assembly language, an END directive simply marks the physical end of the program. The last statement in every assembly language program should have an END directive.

The important thing to notice is that END is a directive to the assembler and not an instruction to be performed by the computer. It simply tells the assembler to stop translating and is not an instruction.

5.3 Statement Syntax

Contrary to what my Auntie Blossom thinks, syntax is not a government levy on immoral deeds. Rather, syntax is a term for the orderly arrangement of the fields in a statement.

The English language has rules of syntax that govern sentence construction. For example, the first letter of the first word in a sentence is capitalized; words are separated by spaces; items in a list are separated by commas; and sentences are terminated by a period, question mark, or exclamation mark. Likewise, assembly language statements also follow rules.

There are rules of syntax for writing assembly language programs with the Editor/Assembler package. Other assemblers may have slightly different rules.

Rules for labels

- A label, if one is used, must come first in a statement.

- A label must have at least one character and no more than six characters.

- The first character of a label must be the first character on the line (even ahead of any space).

- The first character of a label must be a letter (A through Z).

- Any following characters in a label can be letters or numbers (1 through 9).

- A statement can have only a label. In this case, the label is associated with the following statement.

- When a label is used in a statement with an operation code or assembler directive, there must be at least one space between the last character of the label and the first character of the operation code or directive. If a label is not used in a statement with an operation code or directive, there must be at least one space before the first letter of the operation code or directive.

Operation code/assembler directive rules

- An operation code or assembler directive is the second field in a statement.
- About the only thing you need to remember about them is to spell them correctly. For example, the operation code for Move Word is MOV, not MOVE.

Operand field rules

- The third field of a statement is the operand field.

- There are a few operation codes and directives which don't require any operands, but most do.

- There must be at least one space between the last character of the operation code or directive and the first character of the operand field.

- If there's more than one operand, the individual operands are separated by a comma.

- There can be no spaces between the first character and the last character of the operand field unless the spaces appear between a pair of apostrophes.

Comment field rules

- The last field in a statement is the comment field.

- There must be at least one space between the last character of the operand field and the first character of the comment field.

- The comment field can contain any printable characters, including spaces, and can extend to the end of the line.

The fields in a statement must be separated from each other by at least one space. Although only one space is required, it is common practice to align the fields of a statement in columns. This makes the source statement easier to read. For example, the following program is syntactically correct, but difficult to read.

```
MOV @X,R0 PUT X VALUE IN REGISTER 0
MOV @Y,R1 PUT Y VALUE IN REGISTER 1
A R0,R1 ADD X AND Y
MOV R1,@Z SAVE SUM IN Z
BLWP @0 EXIT PROGRAM
X DATA 2 X EQUALS 2
Y DATA 3 Y EQUALS 3
Z DATA 0 Z EQUALS SUM OF X + Y
```

By arranging the statements so that each field is aligned in a column with the same field in the other statements, the program is easier to read.

```
        MOV  @X,R0         PUT X VALUE IN REGISTER 0
        MOV  @Y,R1         PUT Y VALUE IN REGISTER 1
        A    R0,R1         ADD X AND Y
        MOV  R1,@Z         SAVE SUM IN Z
        BLWP @0            EXIT PROGRAM
   X    DATA 2             X EQUALS 2
   Y    OATA 3             Y EQUALS 3
   Z    DATA 0             Z EQUALS SUM OF X + Y
```

Exception to rules

Finally, nearly all rules have exceptions. There is a special kind of statement in assembly language which is largely free from the constraints of syntax. This free spirit is the comment statement. With a comment statement, you can use a whole line to make comments about the program, call attention to the brilliance of your clever design, or chat about your cat's new kittens. To designate a comment line, place an asterisk (*) as the first character in the statement. After the asterisk, you can put any characters you want.

For example, we can add some comment statements to the program as follows.

```
*  THIS PROGRAM ADDS TWO NUMBERS TOGETHER.
*  THE FIRST NUMBER IS STORED IN MEMORY LOCATION X AND
*  THE SECOND NUMBER IS STORED IN MEMORY LOCATION Y.
*  THE SUM IS STORED IN MEMORY LOCATION Z.
*

        MOV  @X,R0         PUT X VALUE IN REGISTER 0
        MOV  @Y,R1         PUT Y VALUE IN REGISTER 1
        A    R0,R1         ADD X AND Y
        MOV  R1,@Z         SAVE SUM IN Z
        BLWP @0            EXIT PROGRAM
   X    DATA 2             X EQUALS 2
   Y    DATA 3             Y EQUALS 3
   Z    DATA 0             Z EQUALS SUM OF X + Y
```

In assembly language, a comment statement is like a REMark statement in BASIC.

Any numeric constants in a statement are treated as decimal values by the assembler unless you indicate otherwise. For example, in the statement

```
   DATA    11
```

the constant 11 is assumed to be decimal eleven.

If you want to specify a hexadecimal number, you can do so by putting a greater-than symbol (>) in front of the number. For example, in the statement

```
DATA    >11
```

the constant >11 is hexadecimal 11 which is equal to decimal 17.

After working with this first program, you may have some more questions such as:

- How do you know exactly what those mnemonic op-codes stand for?

- How do you know exactly what those instructions do?

- How do you know that DATA is an assembler directive and not an instruction operation code?

- Is this the only sequence of instructions you can write to add two words together?

- Why does the program reserve a word of memory for the numbers instead of a byte?

Don't despair. The answers to all these questions are found in the following chapters.

5.4   Main Ideas

When writing assembly language statements, you must follow certain syntax rules which govern the way you write statements.

An assembly language statement can contain up to four fields:

- a label

- an instruction operation code or an assembler directive

- one or more operands

- comments

A label is required when you want to refer to the statement from another statement. When a label is used, it must come first in the statement.

49

Most statements require either an instruction operation code or an assembler directive. An instruction operation code specifies an action for the computer to perform. An assembler directive gives directions to the assembler when the source program is assembled into machine code.

Operands identify the data to be used for an instruction operation or give additional information to be used with an assembler directive.

Comments describe the purpose of the statement.

Each of the four fields must be separated by at least one space. It is common practice, however, to align each of the fields in columns.

A comment statement has an asterisk as the first character. A comment statement in assembly language is like a REMARK statement in BASIC.

The last statement in an assembly language program should contain an END directive. The END directive tells the assembler to stop translating. The END directive does not result in any machine code which is performed by the computer.

Chapter 6

INSTRUCTION SET OVERVIEW

This chapter provides an overview of the TI Home Computer assembly language instruction set. It introduces all the operation codes in the instruction set, classifies the instructions according to the operation they perform, and briefly describes what each instruction does.

The first field in a statement is a label. The second field is either an instruction operation code or an assembler directive.

The purpose of an instruction operation code is to define an operation for the computer to perform. These defined operations make up the computer's instruction set. The TI Home Computer has 69 operation codes in its instruction set.

6.1 Functional Categories

There are several ways to classify the instructions. To begin, let's classify the instructions by functional categories based upon what kind of function they perform.

The instructions can be classified into seven functional categories.

1. Data Movement

2. Compare

3. Jump

4. Arithmetic

5. Logical

6. Branch and Subroutine

7. CRU and External

Let's look at the specific instructions that belong in each functional category.

6.2 Data Movement Instructions

The data movement instructions are those which move data. The main job of a data movement instruction is to move data or rearrange data.

The 12 data movement instructions are listed below.

Mnemonic *Op-code*	*Instruction Name*
MOV	Move Word
MOVB	Move Byte
SWPB	Swap Bytes
LI	Load Immediate
LWPI	Load Workspace Pointer Immediate
LIMI	Load Interrupt Mask Immediate
STWP	Store Workspace Pointer
STST	Store Status
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
SRC	Shift Right Circular
SLA	Shift Left Arithmetic

The most often-used instruction in the entire instruction set is the Move Word instruction (MOV). It moves (copies) a word (16 bits) from one location to another. Its little brother, the Move Byte instruction (MOVB), moves a byte (8 bits) from one location to another.

With the TI Home Computer, a word has 16 bits. Since there are 8 bits in a byte, there are two bytes in a word: a left byte and a right byte. You can visualize a word like this.

```
                     A word

<--------------------- 16 Bits --------------------->
----------------------------------------------------
|       Left  Byte      |      Right Byte           |
----------------------------------------------------
<------- 8 bits -------> <------- 8 bits ------->
```
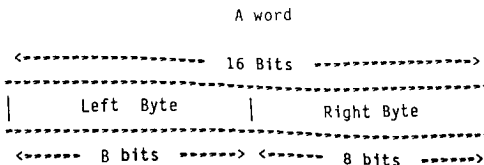
The Swap Bytes instruction (SWPB) simply exchanges the two bytes in a word. Why would you want to do that, you ask? You'll see some uses for it when you learn more about addressing formats.

The Load Immediate instruction (LI) puts a constant into a register; the constant appears directly in the operand field of the instruction.

The Load Workspace Pointer Immediate instruction (LWPI) puts an address into the Workspace Pointer. The Workspace Pointer is the special CPU register that tells the computer the locations of the working registers. The Load Interrupt Mask Immediate instruction (LIMI) puts a number into the computer's interrupt mask. LIMI helps control peripheral devices attached to the computer.

The Store Workspace Pointer instruction (SWPI) copies the contents of the Workspace Pointer into a working register. The SWPI is used to remember the contents of the Workspace Pointer. The Store Status instruction (STST) copies the contents of the Status Register into a working register. It's sometimes used to remember the condition codes before performing another operation.

There are four shift instructions:

- Shift Right Logical (SRL)

- Shift Right Arithmetic (SRA)

- Shift Right Circular (SRC)

- Shift Left Arithmetic (SLA)

The shift instructions move bits within a register to different positions. You can review some examples of how shift instructions are used in Chapter 11.

6.3 Compare Instructions

The Compare instructions compare values and determine their relationships. The 5 instructions in this group are listed below.

53

Mnemonic	*Instruction Name*
Op-code	
C	Compare Words
CB	Compare Bytes
CI	Compare Immediate
COC	Compare Ones Corresponding
CZC	Compare Zeros Corresponding

The Compare Words instruction (C) compares two 16-bit values. The Compare Bytes instruction (CB) compares two 8-bit values. The Compare Immediate instruction (CI) compares two 16-bit values; one is in a register and the other is a constant that appears directly in the operand field of the instruction.

The Compare Ones Corresponding instruction (COC) analyzes specific bits in a word and determines whether they are all ones. The Compare Zeros Corresponding instruction (CZC) analyzes specific bits in a word and determines whether they are all zeros.

6.4 Jump Instructions

The jump instructions are very important because they allow you to make decisions in a program. The 13 jump instructions are listed below.

Mnemonic	*Instruction Name*
Op-code	
JEQ	Jump if Equal
JNE	Jump if Not Equal
JOC	Jump On Carry
JNC	Jump if No Carry
JNO	Jump if No Overflow
JOP	Jump if Odd Parity
JH	Jump if High
JHE	Jump if High or Equal
JLE	Jump if Low or Equal
JL	Jump if Low
JGT	Jump if Greater Than
JLT	Jump if Less Than
JMP	Jump Unconditionally

The first twelve jump instructions are conditional ones. They may, or may not, cause a jump (go to an instruction) based upon certain conditions. The conditional jump instructions let you make decisions about what to do next in a program.

The thirteenth jump instruction, the JMP instruction, is an unconditional one. It causes a jump to a specific instruction unconditionally.

The jump instructions are limited to relatively short-range transfers of control; that is, they can only jump to instructions that are relatively close to them. Each of the jump instructions is discussed in detail in a later chapter.

6.5 Arithmetic Instructions

The Arithmetic instructions are those which perform arithmetic operations. The 13 Arithmetic instructions are listed below.

Mnemonic Op-code	*Instruction Name*
AI	Add Immediate
A	Add Words
AB	Add Bytes
S	Subtract Words
SB	Subtract Bytes
INC	Increment
INCT	Increment by Two
DEC	Decrement
DECT	Decrement by Two
NEG	Negate
ABS	Absolute Value
MPY	Multiply
DIV	Divide

The Add Immediate instruction, AI, adds a 16-bit constant to the contents of a register and replaces the original contents of the register with the sum.

The Add Words instruction, A, adds two 16-bit numbers and produces a 16-bit sum. The Add Bytes instruction, AB, adds two 8-bit numbers and produces an 8-bit sum.

The Subtract Words instruction, S, subtracts a 16-bit number from another and produces a 16-bit difference. The Subtract Bytes instruction, SB, subtracts an 8-bit number from another and produces an 8-bit difference.

There are four instructions that increase or decrease an operand by a fixed amount. The Increment instruction, INC, increases an operand by one and the Increment by Two instruction, INCT, increases an operand by two. The Decrement instruction, DEC,

decreases an operand by one and the Decrement by Two instruction, DECT, deceases an operand by two.

Perhaps you are wondering why there are instructions that increase or decrease an operand by fixed amounts of one or two and not amounts like three, ten, or thirteen-and-a-half. The answer is that these amounts are useful for address manipulations. For example, if you increment an address value by one, you point to the next byte address. If you decrement an address value by one, you point to the previous byte address. If you increment an address value by two, you point to the next word address. If you decrement an address value by two, you point to the previous word address.

The Negate instruction, NEG, negates a value by forming the two's complement) of the value. The Absolute Value instruction (ABS) forms the absolute value of a number.

The Multiply instruction, MPY, multiplies two 16-bit numbers together and results in a 32-bit product. The Divide instruction (DIV) divides a 16-bit divisor into a 32-bit dividend and produces a 16-bit quotient and 16-bit remainder. The Multiply and Divide instructions are both unsigned operations; that is, the numbers are treated as absolute values by the computer.

6.6 Logical Instructions

The logical instructions are those which perform the AND, OR, exclusive OR, and NOT logic operations or they perform functions related to logic operations. The ten instructions in this group are listed below.

Mnemonic Op-code	*Instruction Name*
ANDI	And Immediate
SZC	Set Zeros Corresponding
SZCB	Set Zeros Corresponding Byte
ORI	Or Immediate
SOC	Set Ones Corresponding
SOCB	Set Ones Corresponding Byte
XOR	Exclusive Or
INV	Invert
CLR	Clear
SETO	Set to One

The AND Immediate instruction, ANDI, performs a logical AND operation between the contents of a register and a constant value. The Set Zeros Corresponding instruction, SZC,

performs an operation similar to a logical AND operation between two 16-bit quantities. The Set Zeros Corresponding Byte, SZCB, instruction performs an operation similar to a logical AND operation between two 8-bit quantities.

The OR Immediate instruction, ORI, performs a logical OR operation between the contents of a register and a constant value. The Set Ones Corresponding instruction, SOC, performs a logical OR operation between two 16-bit quantities. The Set Ones Corresponding Byte instruction, SOCB, performs a logical OR operation between two 8-bit quantities.

The Exclusive Or instruction, XOR, performs an exclusive OR logical operation between two 16-bit quantities.

The Invert instruction, INV, inverts the bits in an operand. When an INV instruction is used, all the one bits are changed to zero bits and all the zero bits are changed to one bits. This procedure produces the one's complement of a number.)

The Clear instruction, CLR, provides a simple way of setting the contents of an operand to zero. The Set to One instruction (SETO) sets the contents of an operand to binary ones.

6.7   Branch and Subroutine Instructions

The group of instructions called Branch and Subroutine instructions call subroutines, return from subroutines, or perform long-range transfers of control. The six instructions in this group are listed below.

Mnemonic Op-code	*Instruction Name*
BL	Branch and Link
B	Branch
X	Execute
XOP	Extended Operation
BLWP	Branch and Load Workspace Pointer
RTWP	Return with Workspace Pointer

The Branch and Link instruction, BL is a subroutine-calling instruction. The Branch instruction, B is used to return from a subroutine that is called with a Branch and Link instruction. The Branch instruction also performs a long-range unconditional transfer of control whenever it's needed.

The Execute instruction, X, performs a one-instruction subroutine. You can use it to

perform (execute) an instruction at another location. After that instruction is performed, control returns to the instruction immediately following the Execute instruction.
Both the Extended Operation instruction, XOP, and the Branch and Load Workspace Pointer instruction, BLWP, are instructions for calling a subroutine when the calling program and the subroutine each have their own set of working registers. These two instructions perform what is called a "context switch". The Return with Workspace Pointer instruction (RTWP) is used to return from a subroutine which is called by a context switch.

6.8   CRU and External Instructions

Finally, there's a group called CRU and External instructions. The 10 instructions in this group are listed below.

Mnemonic Op-code	*Instruction Name*
SBO	Set Bit to One
SBZ	Set Bit to Zero
TB	Test Bit
LDCR	Load Communication Register Unit
STCR	Store Communication Register Unit
IDLE	Idle
RSET	Reset
LREX	Load or Restart Execution
CKON	Clock On
CKOF	Clock Off

The first five instructions are the CRU instructions. They're I/O instructions that transfer data between the CPU and peripheral devices.

The last five instructions are the External instructions. They can be used to control peripheral devices or perform other functions unique to a particular application.

These instructions comprise the TI Home Computer assembly language instruction set. Each instruction's operation is described in detail in following chapters.

Chapter 7

ADDRESSING FORMATS: GENERAL

Addressing formats, or addressing modes, is a term that refers to the different ways that the computer addresses data. The operand field in an instruction specifies the data, or the device, used in an operation. Usually, an operand specifies the address of data rather than the actual data. There are different ways to specify the address of a data item. This chapter introduces the TI Home Computer assembly language addressing formats and describes those formats classified as general addressing modes.

7.1 Addressing Formats Overview

There are eight addressing formats used by the TI Home Computer. They're listed below.

1. Register Direct
2. Register Indirect
3. Register Indirect Autoincrement   General
4. Memory (Direct)/"Symbolic"   Addressing
5. Memory (Indexed)/"Indexed"   Modes

6. Immediate

7. PC-Relative

8. CRU
   Single-bit
   Multi-bit

The first five are general addressing modes. You need to remember which ones are general addressing modes.

The sixth is immediate addressing. The seventh is PC-relative addressing. PC stands for Program Counter.

The eighth addressing mode is CRU addressing. This mode is used for CRU instructions. The CRU is an input/output (I/O) channel or "port." There are two variations of CRU addressing: single-bit and multi-bit.

This chapter describes the five general addressing modes.

7.2   General Addressing Modes

The five general addressing modes are:

- Register Direct

- Register Indirect

- Register Indirect Autoincrement

- Memory (Direct), often referred to as "Symbolic" addressing

- Memory (Indexed), often referred to as "Indexed" addressing

To learn how each of the general addressing modes operates you can follow several examples of each addressing mode used with the Move Word instruction. The Move Word instruction is the one most frequently used in programs. Before exploring the addressing modes, however, review the Move Word instruction.

Each instruction is described in a summary found in Appendix A. The instruction summaries are arranged alphabetically according to the mnemonic operation codes. Turn to the Move Word instruction summary. The mnemonic operation code is MOV.

An instruction summary describes a specific instruction. Each summary follows the same format.

The first line contains the name of the instruction on the left and the instruction's mnemonic operation code on the right. The name of this instruction is Move Word and s mnemonic operation code is MOV.

he second line provides the mnemonic operation code, the number of operands required r the instruction, and the kind of addressing formats the operands can have. An

instruction requires none, one, or two operands. If no operands are required, nothing appears on the second line with the mnemonic operation code. If two operands are required, a comma separates them. Otherwise, the instruction requires one operand.

The following codes are used for the operands.

S   indicates a *general source* operand. It means the operand can use any of the five *general* addressing modes. If two operands are required, it is the first one. It's called a *source* operand because it's the operand that is the source, or supplies, the data for an operation.

D   indicates a *general destination* operand. The operand can use any of the five *general* addressing modes. If two operands are required by the instruction, it is the second one. The term *destination* operand is used to indicate it's the destination that receives the result of an operation.

R   means that the operand must be one of the sixteen working registers. (The operand can use only register direct addressing.)

C   indicates a count value and must be a number from 0 through 15.

IOP   indicates an immediate operand. The operand uses only immediate addressing. The operand is treated as a data item rather than the address of a data item. Immediate operands are 16-bit values.

Target   indicates the operand specifies the target for a jump instruction. This code appears only in a jump instruction summary.

Displacement   indicates a displacement for a CRU single-bit instruction. The value of the displacement must be from −128 through +127.

Notice that the MOV instruction requires two operands, an S and a D. Both operands can use any of the five general addressing modes.

The third item of information in an instruction summary is titled "Result". The result is a summarized description of the instruction's operation. A pair of parentheses can be read as "the content of." For example, (S) means "the content of the S operand." For the

MOV instruction, the content of the source operand replaces the content of the destination operand.

The fourth item in the instruction summary is titled "Operation." It is a narrative description of the instruction's operation. The MOV instruction copies a word from the source operand address to the destination operand address.

The fifth item, titled "Status Bits Affected," includes a graphic description of the status bits affected by the instruction's operation. The specific status bits affected by the instruction are shown within the Status Register box.Only those status bits that are affected appear within the Status Register box. Status bits not affected by the instruction are not shown. The MOV instruction affects only the Logical Greater Than (L>), Arithmetic Greater Than (A>), and Equal (EQ) status bits.

Below the Status Register box is a description of how the status bits are affected by the instruction. (In the case of the conditional jump instructions, there's a description of the status bits analyzed by the jump instruction.) With the MOV instruction, the three status bits are affected based upon comparing the word operand to zero.

The sixth item, "Notes," is a collection of notes, examples, and suggested uses for the instruction.

The seventh item, "Machine Code," describes the instruction's machine code.

The line labeled "Hex" contains the hex digits that correspond to the first word of the binary machine code. Only those hex digits are given for which the corresponding machine code nibble is completely defined. A hex digit is not shown for any nibble which contains bits that are not completely defined; that is, the bits vary depending upon the operand(s). Hex digits are shown only for the first word because any other words of machine code always vary depending upon the operand(s). In the case of a MOV instruction, only the first nibble of machine code is completely defined. The nibble contains a binary value of 1100 (a hex C).

The line labeled "Binary" contains the state of the specific bits in the machine code that are fixed and do not change for the instruction. Any group of bits which do vary depending upon the operand(s) contains a code in that field. These codes and their meanings are described in Chapter 19 which describes the structure of machine code.

With this overview of the format of an instruction summary and a closer look at the MOV instruction, let's explore how the general addressing modes work.

The Move Word instruction is a good choice for illustrating the general addressing modes. Not only is it the most commonly used instruction, but it's also the kind of instruction that has two operands, both of which can use any of the five general addressing modes. This is the most flexible kind of instruction in the instruction set.

7.2.1 Register Direct Addressing

The first of the five general addressing modes is register direct addressing. Register direct addressing is used when the data is located directly in one of the sixteen working registers in a workspace. A workspace is a special area of memory that can be accessed faster than any other area of memory. A workspace consists of 16 words of memory. The first word is called Register 0, or R0. The second word is called Register 1. The third word is called Register 2, or R2 and so forth. The sixteenth word (the last word) in a register set is called Register 15, or R15.

Consider this instruction.

```
MOV R10,R1
```

Both the first operand and the second operand are using register direct addressing. The instruction copies the content of Register 10 to Register 1. After the instruction is performed, the content of Register 1 is the same as the content of Register 10.

```
         Unchanged          Changed
           After              After
        +-----------+      +-----------+
   R10  |  1 2 3 4  | ------> |  1 2 3 4  |  R1
        +-----------+      +-----------+
```
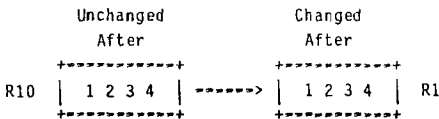
The first operand, named a "source" operand, supplies the data for the operation. The second operand, named a "destination" operand, receives the result of the operation. In this example, Register 10 is the source operand; Register 1 is the destination operand.

By convention, an R precedes a register number. This convention helps the reader of the program to understand that the number in the operand field is a register number.

To specify that you want an operand to use Register direct addressing, write an R and then the register number.

When the assembler, which translates an assembly language instruction into machine code, encounters an operand that can use any of the five general addressing modes, the

```
                              Memory

  MOV  R10,*R3            Before      After
                     +>->-->>--->--+->->>>>>>>>>+
                     :             :            :
                     :             :            :
                     +>>>>>>>>>>>>>+>>>>>>>>>>>>+
              R0     |             |            |
                     +>->>->>>->>>>+>>>->>->>>>>+
                     :             :            :
                     :             :            :
                     +>>>>>>>>>>>>>+>>>>>>>>>>>>+ Memory Pointer
              R3     |  B 3 7 E    |  B 3 7 E   | >>>>>>>>>>>>>+
                     +>>>>>>>>>>>>>+>>>>>>>>>>>>+             |
                     :             :            :            |
                     :             :            :            |
       Data Moved    +>>>>>>>>>>>>>+>>>>>>>>>>>>+            |
       +>>>>>>>> R10  |  A 0 6 2    |  A 0 6 2   |            |
       |             +>>>>>>>>>>>>>+>>>>>>>>>>>>+            |
       |             :             :            :            |
       |             :             :            :            |
       |             +>>>>>>>>>>>>>+>>>>>>>>>>>>+            |
       +>>>>>>>>> B37E |  > > >     |  A 0 6 2   | <>>>>>>>>>>>+
                     +>>>>>>>>>>>>>+>>>>>>>>>>>>+
                     :             :            :
                     :             :            :
                     +>>>>>>>>>>>>>+>>>>>>>>>>>>+
```
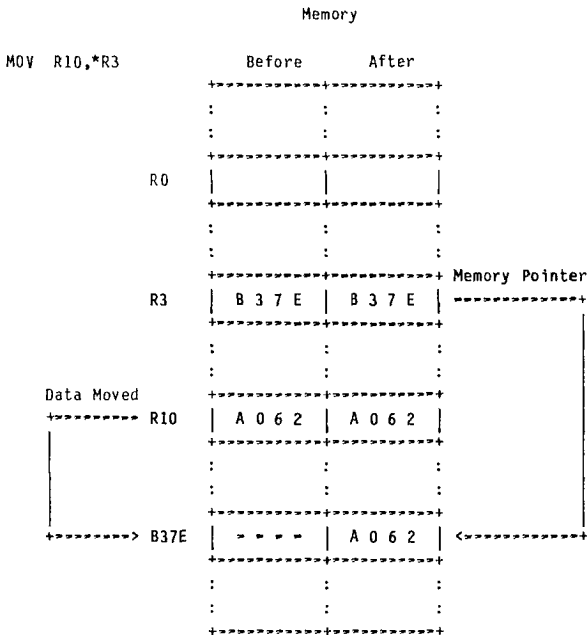
Suppose there is a number in memory location hexadecimal B7E2 that you want to copy into Register 1. And suppose as luck (or planning) would have it, there is a hexadecimal B7E2 in Register 11. What instruction could you write to copy that number into Register 1? You could write the instruction

```
  MOV  *R11,R1
```

After the instruction is performed, Register 1 has a copy of what is in memory location hexadecimal B7E2 and Register 11 still has the hexadecimal B7E2 in it.

7.2.3 Register Indirect Autoincrement Addressing

The third of the five general addressing modes is register indirect autoincrement addressing. It works almost exactly like register indirect addressing. With register indirect

autoincrement addressing mode, a register contains the address of the data, but after the data is accessed, the content of the register is automatically incremented.

Register indirect autoincrement addressing mode is specified by writing an asterisk followed by a register number (with its R prefix) and followed by a plus (+) sign.

In the following instruction

```
MOV  R10,*R3+
```

the source operand is using register direct addressing and the destination operand is using register indirect autoincrement addressing mode.

Assume that before the instruction is performed, Register 10 contains the number hexadecimal A062 and that Register 3 contains a hexadecimal B37E.

When the instruction is peformed, the number hexadecimal A062 in Register 10 is copied into memory location hexadecimal B37E and the content of Register 3 is automatically incremented by two (to hexadecimal B380).

67
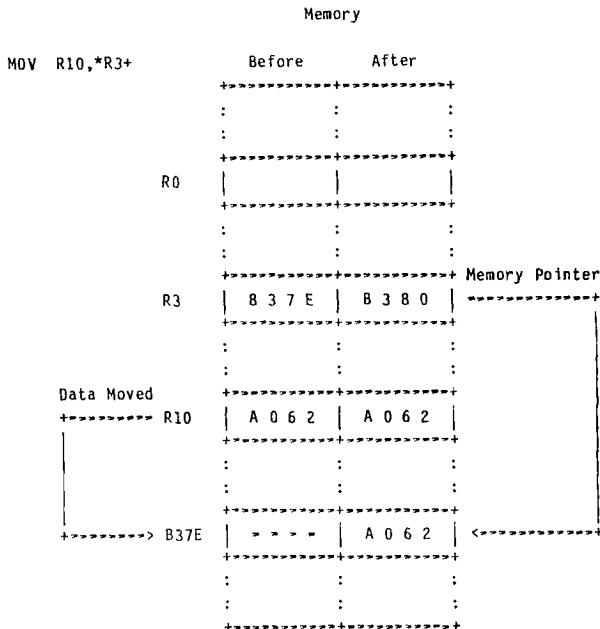
Notice that the address in the register is automatically incremented after the data at that address is accessed.

The content of the indirect register are incremented by two because the Move Word instruction peforms a word operation. If the instruction using register indirect autoincrement addressing mode performs a byte operation (as does a Move Byte instruction, for example), the content of the indirect register is incremented by one.

Autoincrementing by two allows a word-operation instruction to access a word and automatically adjusts the address in the indirect register to the next *word* in memory. Autoincrementing by one allows a byte-operation instruction to access a byte and automatically adjusts the address in the indirect register to the next *byte* in memory.

Using register indirect autoincrement addressing mode is helpful when accessing several sequential data items in a list. You can point to the first item in the list by putting the address of that item into a register. Then, by using register indirect autoincrement addressing mode, you can access the item and have the address in the indirect register automatically adjusted so that it points to the next sequential data item. Each time you access an item, the address in the indirect register is automatically adjusted to point to the next item.

The TI Home Computer has a register indirect autoincrement addressing mode but it does not have an autodecrement addressing mode.

Suppose there are several data items in consecutive words of memory beginning at location hexadecimal A0A4 and you want to copy the first word to Register 7 and automatically be ready to access the second word. And suppose that hexadecimal A0A4 happens to be in Register 10. What instruction could you write to do this? The instruction would look like this.

```
MOV  *R10+,R7
```

It copies the contents of memory location hexadecimal A0A4 into Register 7 and automatically adjusts the address in Register 10 to hexadecimal A0A6, the address of the second word.

7.2.4 Symbolic Addressing

The fourth general addressing modes is direct memory addressing, or as it is more often called, "symbolic" addressing. Symbolic addressing is used to address directly a data item in general memory by putting its address directly in the operand field.

If the data item is in a register, you can use register direct addressing. But if it's not in a register, you can address it directly using symbolic addressing.

Symbolic addressing is specified by writing the address of the memory location preceded by an "at" sign (@).

In the following instruction

```
MOV  @>B37E,R10
```

the source operand uses symbolic addressing and the destination operand is uses register direct addressing. Suppose that memory location hexadecimal B37E contains the number hexadecimal A062 before the instruction is performed. After the instruction is performed, Register 10 contains the number hexadecimal A062 also.

```
                         Memory

        MOV  @>B37E,R10   Before      After
                         +>>>>>>>>>>+>>>>>>>>>>+
                         :         :         :
                         :         :         :
                         +>>>>>>>>>>+>>>>>>>>>>+
          R0             |         |         |
                         +>>>>>>>>>>+>>>>>>>>>>+
                         :         :         :
                         :         :         :
       Data Moved        +>>>>>>>>>>+>>>>>>>>>>+
       +-->>>>>>> R10    | > > > > | A 0 6 2 |
       |                 +----------+----------+
       |                 :         :         :
       |                 :         :         :
       |                 +----------+----------+
       +--------- B37E   | A 0 6 2 | A 0 6 2 |
                         +----------+----------+
                         :         :         :
                         :         :         :
                         +----------+----------+
```
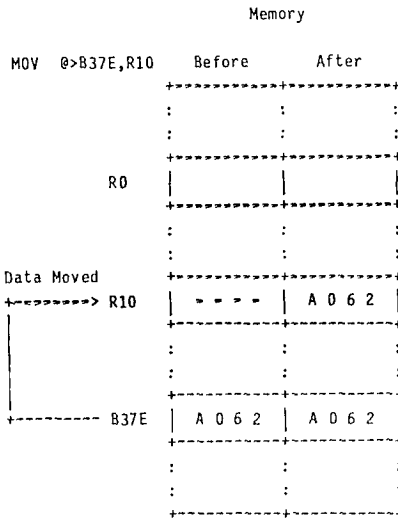
This addressing mode is called symbolic addressing because, most of the time, the operand uses a symbolic address rather than a numeric address. For example, if location hexadecimal B37E were assigned the name DOG, the same instruction could be written as

```
MOV  @DOG,R10
```

Suppose there's a word of data at a memory location named CAT and you want to copy it to Register 15. What instruction could you write?

This instruction would do the job:

```
MOV  @CAT,R15
```

What if you want to copy the data item in Register 0 to a memory location named GERBIL. What instruction could you write to do it?

The instruction below serves the purpose.

```
MOV  R0,@GERBIL
```

7.2.5 Indexed Addressing

The fifth of the five general addressing modes is commonly called indexed addressing. Indexed addressing is used when you want to specify a data item with a combination of symbolic addressing and an address in a register.

Indexed addressing mode is specified by writing a memory location preceded by an "at" sign, @, that is followed by a register number in parenthesis. The register within the parenthesis is called the index register.

The following instruction uses indexed addressing for the source operand and register direct addressing for the destination operand. The index register is Register 3.

```
MOV  @PICKLE(R3),R10
```

Since Register 10 is the destination operand, a word of data is moved into Register 10. The location of the data item moved into Register 10 is determined by adding the content of the index register to the address value of the memory location in the source operand. The sum is the address of the data item.

Suppose that Register 3 contains the number 4 and that the address value of PICKLE is hexadecimal B37E. The effective address of the source operand is hexadecimal B37E plus 4, or hexadecimal B382. The contents of memory location hexadecimal B382 (PICKLE+4) is copied into Register 10.

```
                            Memory

MOV @PICKLE(R3),R10           Before        After
                      +------------+------------+
                      :            :            :
                      :            :            :
                      +------------+------------+
                R0    |            |            |
                      +------------+------------+
                      :            :            :
                      :            :            :
                      +------------+------------+
                R3    |  0 0 0 4   |  0 0 0 4   |
                      +------------+------------+
                      :            :            :
                      :            :            :
  Data Moved          +------------+------------+
  +----> R10          |  - - - -   |  A 0 6 2   |
  |                   +------------+------------+
  |                   :            :            :
  |                   :            :            :
  |                   +------------+------------+
  |        PICKLE     |            |            |
  |                   +------------+------------+
  |        PICKLE+2   |            |            |
  |                   +------------+------------+
  +------ PICKLE+4    |  A 0 6 2   |  A 0 6 2   |
                      +------------+------------+
                      :            :            :
                      :            :            :
                      +------------+------------+
```
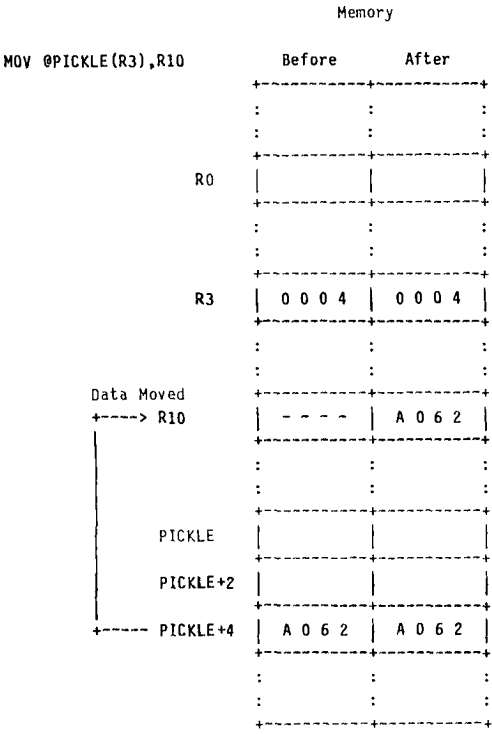
If Register 10 contains hexadecimal B37E, the following instruction accomplishes the same thing.

```
MOV @4(R10),R9
```

Indexed addressing is useful for accessing specific data items from a set of data. For example, assume that memory location LIMITS is the first word of a table of several contiguous words of data. Then this instruction can be used to access a partiacular word in the table and copy that word into Register 5:

```
MOV  @LIMITS(R2),R5
```

The particular word accessed is determined by the number in Register 2. If Register 2 contains 0, the first word is accessed. If Register 2 contains 2, the second item is accessed. If Register 2 contains 4, the third item is accessed; and so forth.

To understand indexed addressing better, study the following two instructions. Both accomplish the same objective.

```
MOV  @0(R6),R0

MOV  *R6,R0
```

Both instructions copy a word into Register 0. The address of the word of data that's copied is determined by the address in Register 6. In the first instruction, Register 6 is an index register. In the second instruction, Register 6 is an indirect register.

With indexed addressing, *except for register 0*, any of the registers can be used as an index register. Register 0 can't be used as an index register due to the structure of the machine code.

7.3   Word and Byte Addressing

The Move Word instruction performs a word operation. It uses a 16-bit value for its operation. Some instructions perform byte operations and use 8-bit values.

An example of an instruction that performs a byte operation is the Move Byte (MOVB) instruction. It operates almost exactly like the MOV instruction except that it copies an 8-bit value from one location to another.

The MOVB instruction requires two operands. Like the MOV instruction, both operands can use any of the five general addressing modes. In the case of the MOVB instruction, however, the operands are byte addresses rather than word addresses.

For example, this instruction moves a byte from byte address decimal 100 to byte address hexadecimal A084:

```
MOVB  @100,@>A084
```

Assume that word address decimal 100 contains a hexadecimal 9E63 and word address hexadecimal A084 contains a hexadecimal C072 before the instruction is performed. The

instruction moves the content of byte address decimal 100, the byte value hex 9E, to byte address hexadecimal A084.

After the instruction is performed, byte address hexadecimal A084 has a byte value of hex 9E also.

```
Word Address        Before      After

    (100)       =    >9E63      >9E63
  (>A084)       =    >C072      >9E72
```

Whenever a byte operation is performed with an operand using register direct addressing, the left byte of the register is always used. For example, before the following instruction is performed, assume that Register 4 contains a hexadecimal D19F and memory word hexadecimal A084 contains a hexadecimal C072.

```
MOVB R4,@>A084
```

After the instruction is performed, byte address hexadecimal A084 has a copy of the value in the left byte of Register 4.

```
                 Before      After

  (R4)      =    >D19F      >D19F
  (A084)    =    >C072      >D172
```

Recall that the left byte of a word has an even-numbered address and the right byte of a word has an odd-numbered address.

Let's look at this instruction:

```
MOVB  *R7,R11
```

Suppose that before the instruction is performed, Register 7 contains a hexadecimal A085, Register 11 contains a hexadecimal D19F, and memory word hexadecimal A084 contains a hexadecimal C072.

After the instruction is performed, Register 7 still contains a hexadecimal A085, Register 11 contains a hexadecimal 729F, and memory word hexadecimal A084 still contains a hexadecimal C072.

		Before	After
(R7)	=	>A085	>A085
(R11)	=	>D19F	>729F
(A084)	=	>C072	>C072

It's possible that sometimes the computer can be directed by an instruction to perform a word operation with an odd-numbered address. For example, suppose that Register 7 contains a hexadecimal A085, Register 11 contains a hexadecimal D19F, and memory word hexadecimal A084 contains a hexadecimal C072 before the following instruction is performed.

```
MOV  *R7,R11
```

The instruction asks for the performance of a word operation, Move Word, with the odd-numbered address, hexadecimal A085. In this situation, the computer rounds the odd-numbered address down to the next lower even number to establish word alignment for the operation. In this example, the computer rounds the odd-numbered address hexadecimal A085 down to A084 and performs the operation with the contents of word address hexadecimal A084.

		Before	After
(R7)	=	>A085	>A085
(R11)	=	>019F	>C072
(A084)	=	>C072	>C072

You probably wouldn't intentionally use an odd-numbered address for a word operation, but if you do, the computer rounds the odd-numbered address down to the next lower even address.

7.4   A Look at Another Instruction (Add Words)

Several examples have illustrated the five general addressing modes using the Move Word instruction. The Move Word instruction is one that uses two operands, both of which let you use any of the general addressing modes.

A similar instruction is the Add Words instruction. Look at the Add Words, mnemonic op-code A, in Appendix A. Notice that both the first and second operand can use any of the general addressing modes. (The operands have S and D codes.) The Add Words instruction adds the content of the first operand to the content of the second operand, placing the results in the second operand.

For example, suppose that Register 14 contains the number 23 and that Register 3 contains the number 54, then the instruction

```
A    R14,R3
```

adds 23 to 54 and places the sum, 77, in Register 3. The contents of Register 14 is still 23.

As another example, the instruction

```
A    *R2,@GENIE
```

adds the number whose address is in Register 2 to the contents of the memory location called GENIE. The sum is placed in memory location GENIE.

An example program in the next chapter uses the Add Words instruction.

7.5 Summary

This chapter introduces the general addressing modes. The five general addressing modes and the assembly language syntax for specifying each addressing mode is listed below.

General Addressing Mode	Assembly Language Syntax
1. Register Direct	Rx
2. Register Indirect	*Rx
3. Register Indirect Autoincrement	*Rx+
4. Memory (Direct)/"Symbolic"	@location
5. Memory (Indexed)/"Indexed"	@location(Ry)

x is any number 0 through 15
y is any number 1 through 15
location is a numeric or symbolic address

Chapter 8

ADDRESSING FORMATS: IMMEDIATE AND PC-RELATIVE

The previous chapter identifies the TI Home Computer's eight addressing modes and describes the operation of the first five addressing modes (the ones which together are classified as general addressing modes).

This chapter describes two more addressing formats: immediate and PC-relative. The eighth addressing mode, CRU addressing, is described in another chapter.

This chapter illustrates the immediate and PC-relative addressing formats and how to structure a program loop in assembly language.

8.1 Immediate Addressing

The sixth addressing mode is immediate addressing. Immediate addressing is not a general addressing mode. With a general addressing mode, the operand specifies the address of a data item rather than the data item itself. With immediate addressing, the data item itself appears directly, or "immediately," in the operand field.

8.1.1 The Load Immediate Instruction

As an example of an instruction that uses immediate addressing, look in Appendix A at the instruction summary for the Load Immediate instruction. The mnemonic op-code is LI.

Notice that the instruction requires two operands in the operand field. The first is an R-type operand. The R means that the first operand must be a register; that is, only register direct addressing can be used for that operand. The second operand is an IOP-type operand. IOP means that the operand is an immediate operand. The instruction uses the second operand as a data value rather than the address of a data value.

Consider the following Load Immediate instruction.

```
LI   R7,26
```

The first operand is R7. The second operand is 26. The first operand uses register direct addressing, which is the only addressing mode that can be used for the first operand. The second operand uses immediate addressing, which is the only addressing mode that can be used for the second operand. The second operand is said to be an immediate operand.

```
   Register 7              Immediate Operand
   +-------------+
   |             |  <------------    26
   +-------------+
```
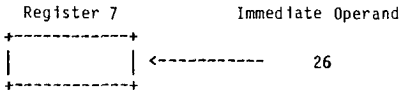
The instruction copies the immediate operand to into the register. The Load Immediate instruction is useful when you want to put a specific data value in a register.

8.1.1.1   Comparison of LI instruction with MOV Instruction

You can also use a Move Word instruction to put a data value into a register. For example, you can use a Move Word instruction like

```
MOV  R2,R7
```

to put a 26 into R7. Of course, Register 2 must have a 26 in it first. It would be simpler just to use a Load Immediate instruction to put the 26 directly into Register 7.

You could also use a Move Word instruction like this

```
MOV  @LAMP,R7
```

to copy a 26 into R7 (assuming, that memory location LAMP contains a 26). This Move Word instruction requires two words of memory for its machine code and another word of memory, called LAMP, is needed to hold the 26. The Move Word instruction, therefore, requires three words of memory; wheras the instruction

```
LI   R7,26
```

requires only two words of memory for its machine code. The advantage of the Load Immediate instruction is that is saves memory.

The Move Word instruction, however, has the advantage of allowing you to use a wide

variety of addressing modes. Remember the Load Immediate instruction only lets you use register direct addressing for the first operand and only immediate addressing for the second operand. The Move Word instruction lets you use your choice of any of the five general addressing modes for both operands. For example, if you want to copy a word stored in a general memory location to another general memory location, you can do it with a Move Word instruction. You can't do it, though, with a Load Immediate instruction.

Suppose you want to copy a number in general memory location PENCIL to memory location PAPER. The single instruction

```
MOV @PENCIL,@PAPER
```

works fine, but you couldn't use a single Load Immediate instruction.

If you were to write an instruction like

```
LI   @PAPER,@PENCIL
```

the assembler would not translate it because the first operand is required to be a register and the second operand must be a data item, not the address of a data item.

Both the Load Immediate and the Move Word instruction have advantages in some situations. You can choose the best instruction for a particular situation.

8.1.1.2   Using the LI Instruction in a Loop

You can use the Load Immediate instruction to put a constant into a register. You may want to use this instruction at the beginning of a loop. A loop performs a series of instructions repeatedly. Usually, the loop repeats a fixed number of times or until some condition is true.

Suppose you want to repeat a series of instructions four times. In BASIC, you can build a FOR-NEXT loop like this.

```
900 FOR Z = 1 TO 4
910 -------
9XX     .
9XX     .
9XX -------
9XX NEXT Z
```

Assembly language, does not have FOR and NEXT instructions. You must control the loop explicitly by adjusting the loop count and analyzing the resulting loop count. You must build a loop which is more like this.

```
900 Z = 4
910 -------
9XX   .
9XX   .
9XX -------
9XX Z = Z -1
9XX IF Z <> 0 THEN 910
```

To build this kind of loop in assembly language, put the loop count (4) into a register, perform the series of instructions, subtract one from the loop count in the register; if the loop count is not zero, perform the loop again. When the loop count becomes zero, the program falls out of the loop and goes on to the next instruction after the loop. Here's a general structure of the loop:

```
              <Set  loop  count  in  a  register>
+-------->   - - - - - - - - - - - - - - - - - -
|                              .
|                              .
|             <Perform the instructions in the loop>
|                              .
|                              .
|            - - - - - - - - - - - - - - - - - - -
|             <Subtract  one  from  the  loop count>
+---------    <If  loop  count   not equal to  zero>
```
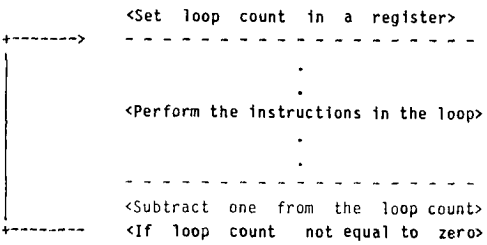
The Load Immediate instruction is useful for setting up the loop count in the register. You can use any one of the sixteen registers to hold the loop count. If you use Register 8, the loop looks like this.

```
              LI   R8,4   SET LOOP COUNT IN R8
+-------->   - - - - - - - - - - - - - - - - - - - -
|                              .
|                              .
|             <Perform the instructions in the loop>
|                              .
|                              .
|            - - - - - - - - - - - - - - - - - - -
|             <Subtract  one  from  the  loop count>
+---------    <If  loop  count   not equal to  zero>
```
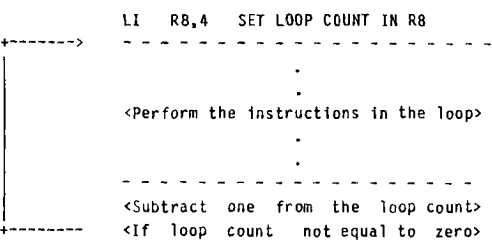
8.1.2   The Add Immediate Instruction

While exploring the subject of immediate addressing, let's look at another example of an instruction that uses immediate addressing. Locate the instruction summary for the Add Immediate instruction in Appendix A. (The mnemonic op-code is AI).

Like the Load Immediate instruction, the Add Immediate instruction requires two operands. The first operand must be a register; the second operand must be an immediate value.

The Add Immediate instruction adds the immediate operand to the contents of the register and leaves the sum in the register. Notice from the instruction summary that several status bits are affected by the Add Immediate instruction. The instruction automatically compares the sum to zero and either sets or clears the Arithmetic Greater Than, Logical Greater Than, and Equal status bits. For example, if the sum equals zero, the Equal status bit is set to one. If the sum does not equal zero, the Equal status bit is cleared to zero.

You can use the Add Immediate instruction in the above loop. You can use it to subtract one from the content of the loop counter register. You can subtract by adding a negative number. For example, an instruction like

```
AI   R8,-1
```

subtracts one from the contents of Register 8.

With an Add Immediate instruction, the loop looks like this.

```
                LI   R8,4   SET LOOP COUNT IN R8
+------->        - - - - - - - - - - - - - - - - - - -
|                             .
|                             .
|               <Perform the instructions in the loop>
|                             .
|                             .
|                - - - - - - - - - - - - - - - - - -
|               AI   R8,-1  SUBTRACT ONE FROM LOOP COUNT IN R8
+--------       <If  loop  count   not equal to  zero>
```
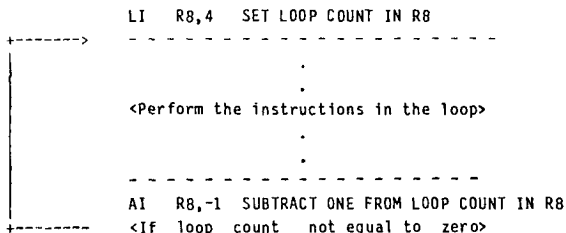
8.2   PC-Relative Addressing

The seventh addressing mode is PC-relative or Program Counter- relative addressing. The only instructions that use PC-relative addressing are the jump instructions. PC-

relative addressing is the only addressing mode the jump instructions can use. Let's review jump instructions and see how they use PC-relative addressing.

If you recall, there are thirteen jump instructions. Twelve are conditional jump instructions. They may, or may not, cause a transfer of program control depending upon whether certain conditions are true or untrue. Each conditional jump instruction is designed to analyze a particular condition. If that condition is true, the jump instruction causes a jump to a target instruction. If the condition is not true, the program goes on to the next sequential instruction.

A conditional jump instruction determines whether a condition is true or not by analyzing one or more status bits. For example, one conditional jump instruction is the Jump if Equal instruction (JEQ). It determines whether or not to jump by analyzing the Equal status bit. If the status bit is true, or a one, the Jump if Equal instruction causes a jump; if the status bit is not true, or a zero, it doesn't jump and program control continues to the next sequential instruction.

Another conditional jump instruction that analyzes the Equal status bit is the Jump if Not Equal instruction (JNE). If the Equal status bit is not true (zero), the Jump if Not Equal instruction causes a jump. If the Equal status bit is true (one), the Jump if Not Equal instruction does not jump. You can see that the JEQ instruction and the JNE instruction analyze the same status bit but check for different conditions.

One unconditional jump instruction exists among the thirteen jump instructions. This instruction is called Jump Unconditionally and its mnemonic op-code is JMP. The JMP instruction doesn't analyze any status bits. It simply causes a jump no matter what.

8.2.1 Jump Instruction Targets

Every jump instruction requires an operand. The purpose of the operand is to specify the next instruction to be performed next if there is a jump. The instruction that receives control from a jump instruction is called the target of the jump.

The operand of a jump instruction uses PC-relative addressing. The target of the jump is specified as relative to the Program Counter. The Program Counter, or PC, is a special counter in the computer that keeps track of the address of the next instruction to be performed. Although it's called a counter, the PC is simply a special register that holds the address of the next instruction to be performed.

Whenever an instruction is performed, the computer automatically adjusts the address in the Program Counter to the address following the instruction. When a jump instruction

is performed that results in a jump, the value of the jump instruction's operand is added to the contents of the Program Counter. When the computer finishes the jump instruction and is ready to perform the next instruction, it goes to the adjusted address in the Program Counter to get the next instruction; it jumps.

The jump instruction's operand specifies how much to add to the contents of the Program Counter to reach the target of the jump; therefore, the operand of a jump instruction is a "PC-relative" address.

Another way to think about the target of a jump instruction is to think about its relative distance from the location of the jump instruction. After all, when a jump instruction is performed, the address in the Program Counter is always the next word after the address of the jump instruction itself.

8.2.1.1 Distance to the Target

The target for a jump instruction must be relatively close to the location of the jump instruction itself. Without a lengthy explanation here, the jump range of a jump instruction is limited to plus 256 bytes and minus 254 bytes from the location of the instruction.

8.2.1.2 Methods for Specifying a Target

When writing programs in assembly language, there are three ways you can specify the target of a jump instruction.

1. The best way for many situations is to use the name (label) attached to the target instruction. For example, if you want your program to jump when the Equal Status bit is true, you can write an instruction like this:

    ```
    JEQ  MICKEY
    ```

    MICKEY is a label attached to the target instruction. In this example, the target instruction must be labeled MICKEY or the assembler cannot translate the instruction into the right machine code.

2. You can specify a numeric address. After all, a label is simply a name assigned to the numeric location of a statement so you can use the numeric address itself. For example, you can write an instruction like this:

    ```
    JEQ  42826
    ```

83

42826 is the address of the target instruction.

3. You can specify how far to jump (rather than where to jump). Here's an example.

```
JEQ  $+36
```

This instruction causes a jump if the the Equal status bit is true to a location that is a distance of plus 36 bytes from the location of the JEQ instruction. The $ symbol means the location of the statement in which it appears. If this JEQ instruction is located at address 42000, then the target is located at address 42036.

As another example, the instruction

```
JEQ  $-100
```

specifies a target that is a distance of minus 100 bytes from the location of the JEQ instruction. If this JEQ instruction is located at address 53982, its target is at address 53882.

These are the three ways to specify the target of a jump instruction:

- use a label

- use a numeric address

- use a dollar sign and a relative distance from the location of the jump instruction.

Using a label is usually the best way. Regardless of which method you use, the target of the jump instruction must be within range. That range is not more than -254 to $+256$ bytes from the location of the jump instruction.

```
    Range of a Jump Instruction

+---->    - 254 Bytes
|                 :
|                 :
+---- Jump Instruction
|                 :
|                 :
+---->    + 256 Bytes
```
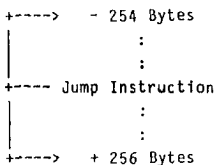
```
                   .
<Perform the instructions in the loop>
                   .
                   .
- - - - - - - - - - - - - - - - - - - -
AI   R8,-1  SUBTRACT ONE FROM LOOP COUNT IN R8
JNE  LOOP   IF COUNT NOT ZERO, GO TO LOOP START
```

Next, use the Load Immediate instruction again to set a register, say Register 9, to zero. This register accumulates the results of the repetitive addition.

```
        LI   R7,5   PUT NUMBER TO BE SQUARED IN R7
        MOV  R7,R8  COPY NUMBER INTO R8 FOR LOOP COUNT
        LI   R9,0   INITIALIZE CUMULATIVE SUM TO ZERO
LOOP    - - - - - - - - - - - - - - - - - - - -
                       .
                       .
        <Perform the instructions in the loop>
                       .
                       .
        - - - - - - - - - - - - - - - - - - - -
        AI   R8,-1  SUBTRACT ONE FROM LOOP COUNT IN R8
        JNE  LOOP   IF COUNT NOT ZERO, GO TO LOOP START
```

After all these initialization procedures, constructing the body of the loop is straight forward. Add the number in Register 7 to the number in Register 9.

```
        LI   R7,5   PUT NUMBER TO BE SQUARED IN R7
        MOV  R7,R8  COPY NUMBER INTO R8 FOR LOOP COUNT
        LI   R9,0   INITIALIZE CUMULATIVE SUM TO ZERO
LOOP    A    R7,R9  ADD NUMBER TO CUMULATIVE SUM
        AI   R8,-1  SUBTRACT ONE FROM LOOP COUNT IN R8
        JNE  LOOP   IF COUNT NOT ZERO, GO TO LOOP START
```

The program repeatedly adds the 5 in Register 7 to the contents of Register 9 and subtracts one from the loop count in Register 8 until the loop count is zero.

The program is almost complete. It now adds five to the number in Register 9 until the loop count in Register 8 is zero. Then the program comes to the end of the loop and goes to the JNE instruction. The problem is: there isn't an instruction following the Jump if Not Equal instruction.

Ending an assembly language program is different from ending a BASIC program. When

a BASIC program is finished and the last instruction is performed, the BASIC interpreter waits for the next instruction. The BASIC interpreter is a program that interprets and performs the instructions in a program. When the program runs out of instructions, the BASIC interpreter is still controlling the computer.

In assembly language program, however, the program controls the computer directly. Once you have used your program to have the computer do what you want, you must include an instruction to have your program give control to another program. If you fail to include such an instruction, the computer's response is unpredictable.

At this point, let's introduce an instruction that will keep the computer under control. Right after the JNE instruction, let's place a "Go-Home" instruction.

```
        LI    R7,5   PUT NUMBER TO BE SQUARED IN R7
        MOV   R7,R8  COPY NUMBER INTO R8 FOR LOOP COUNT
        LI    R9,0   INITIALIZE CUMULATIVE SUM TO ZERO
LOOP    A     R7,R9  ADD NUMBER TO CUMULATIVE SUM
        AI    R8,-1  SUBTRACT ONE FROM LOOP COUNT IN R8
        JNE   LOOP   IF COUNT NOT ZERO, GO TO LOOP START
        BLWP  @0         OTHERWISE, GO HOME
```

The Go-Home instruction has been added to the program. Later, a discussion of exactly how the BLWP instruction operates is given. For right now, understanding what it does in this program is sufficient. It causes the computer to display the title screen, as if you had just turned the computer on.

One more statement is needed. If you recall, every program ends with an END statement, a statement containing an END directive. The END directive marks the end of the program and instructs the assembler to stop assembling.

```
        LI    R7,5   PUT NUMBER TO BE SQUARED IN R7
        MOV   R7,R8  COPY NUMBER INTO R8 FOR LOOP COUNT
        LI    R9,0   INITIALIZE CUMULATIVE SUM TO ZERO
LOOP    A     R7,R9  ADD NUMBER TO CUMULATIVE SUM
        AI    R8,-1  SUBTRACT ONE FROM LOOP COUNT IN R8
        JNE   LOOP   IF COUNT NOT ZERO, GO TO LOOP START
        BLWP  @0         OTHERWISE, GO HOME
        END
```

Now you have an assembly language program that does something useful. When the program is finished, it relinquishes control of the computer in an orderly fashion.

Once the program construction is complete you've reached a major milestone, but you're

8.2.2 Using a Jump Instruction in a Loop

You can use a conditional jump instruction in the example loop.

As discussed previously, you can use the Add Immediate instruction to subtract one from the loop counter register each time the loop is performed. The Add Immediate instruction automatically compares the result remaining in the register with zero and affects several status bits, including the Equal status bit. If the result in the loop counter register equals zero, the Equal status bit is set to one. If the result does not equal zero, the Equal status bit is cleared to zero.

After the Add Immediate instruction is performed, you can have the program check if the result of subtracting one from the loop counter produced a zero or not. If the result is not zero, you want the program to jump back and perform the series of instructions in the loop again. If the result is zero, you know that the loop has been performed enough times and you can allow the program to precede to the next instruction after the loop.

You can use a conditional jump instruction after the Add Immediate instruction to check if the Equal status bit was set or not. There are two jump instructions that check the state of the Equal status bit: Jump if Equal (JEQ) and Jump if Not Equal (JNE). You want the program to jump back to the beginning of the loop if the result in the loop counter is not equal to zero, so use the JNE instruction.

The JNE instruction, like all jump instructions, must specify a target in the operand field. The target of the JNE instruction is the first instruction within the loop. Choose a name for that instruction, say LOOP, and attach the name as a label to the instruction. Then use that name as the target for the JNE instruction. The program segment now looks like this.

```
        LI   R8,4   SET LOOP COUNT IN R8
LOOP    - - - - - - - - - - - - - - - - - - - -
                       .
                       .
        <Perform the instructions in the loop>
                       .
                       .
        - - - - - - - - - - - - - - - - - - -
        AI   R8,-1  SUBTRACT ONE FROM LOOP COUNT IN R8
        JNE  LOOP   IF COUNT NOT ZERO, GO TO LOOP START
```

8.3 Building a Program Example

This shell allows for a loop to perform four times. By putting a different number into Register 8, you determine the number of times the loop is performed. For example, if you put 100 into Register 8, the loop is performed 100 times.

This loop can be modified slightly to do something specific. Let's have the program square a number. To square a number, you multiply the number times itself.

Multiplication is simply repetitive addition. For example. if you want to find out how much 8 times 3 is, you can get the answer by adding 8 three times or by adding 3 eight times.

To square a number, you multiply the number times itself. You can achieve the same result by repeatedly adding the number until you've added it a number of times equal to the number itself. For example, you can square the number 4 by adding it four times.

```
4+4+4+4 =16
```

To modify the loop to square a number, start by using a Load Immediate instruction to put the number you want to square in a register, say Register 7. Let's choose the number 5 and square it.

```
        LI   R7,5   PUT NUMBER TO BE SQUARED IN R7
        LI   R8,4   SET LOOP COUNT IN R8
LOOP    - - - - - - - - - - - - - - - - - -
                          .
                          .
        <Perform the instructions in the loop>
                          .
                          .
        - - - - - - - - - - - - - - - - - -
        AI   R8,-1  SUBTRACT ONE FROM LOOP COUNT IN R8
        JNE  LOOP   IF COUNT NOT ZERO. GO TO LOOP START
```

Use a Move Word instruction instead of the second Load Immediate instruction to copy the number into the loop counter register (Register 8).

```
        LI   R7,5   PUT NUMBER TO BE SQUARED IN R7
        MOV  R7,R8  COPY NUMBER INTO R8 FOR LOOP COUNT
LOOP    - - - - - - - - - - - - - - - - - -
                          .
```

not through yet. Next, you run the program.

In the next chapter, you can use this program to learn more about the Editor, the Assembler, the Loader, and the Debugger utility programs which come with the Editor/Assembler package. (A utility program helps you develop your application program.)

8.4 Summary

This chapter introduces the immediate and PC-relative addressing modes. The Load Immediate and Add Immediate instructions are two instructions that use immediate addressing. The jump instructions use PC-relative addressing.

In this chapter, the program loop that squares a number illustrates the use of instructions that employ immediate and PC-relative addressing. In the following chapters, the program loop is used to help you learn to use the utility programs to develop assembly language programs.

Chapter 9

INTRODUCTION TO
THE EDITOR AND ASSEMBLER

Chapters 9 and 10 explain the role of utility programs in the development of assembly language programs. A utility program is one that is designed to aid in the development of another program. Utility programs for assembly language include editors, assemblers, loaders, and debuggers.

- An editor helps you compose a source program.

- An assembler translates a source program into an object program.

- A loader loads the machine code of an object program into memory.

- A debugger helps you test a program and detect bugs.

There are several utility packages available to help you develop assembly language programs. For example, Texas Instruments offers an Editor/Assembler package which includes an editor, assembler, loader, and debugger. The Mini Memory Module that includes a line-by-line assembler and a debugger and the UCSD p-System with an assembler and loader are both available.

This book describes the use of the TI Editor/Assembler package as an example of the role of utilities in developing a program and as an example of what the utilities do. If you don't have the Editor/Assembler package, you can use these examples as guidelines and adapt the concepts to the utility programs that you use.

Chapters 9 and 10 use the program from Chapter 8 (the program that squares a number) to illustrate the use of the utility programs with the Editor/ Assembler package. Chapter 9 introduces the Editor and Assembler utility programs. Chapter 10 introduces the Loader and Debugger.

If you have the Editor/Assembler package, you can use the guidelines that are presented to help you edit, assemble, load, and run the program you developed in the previous chapter.

NOTE

The descriptions of the operation of the utility programs are simplified descriptions. The Editor/Assembler manual contains cautionary notes, hints, troubleshooting suggestions, and more precise details about the utility programs.

If you have the Editor/Assembler manual, read Sections 1 and 2 of the manual before continuing.

9.1   The Editor

The Editor is a utility program that helps you create a source program. You can use the Editor to type your program and compose the statements like you want them to appear before passing the source program to the Assembler.

9.1.1   Bringing Up the Editor/Assembler Package

The steps for getting the Editor/Assembler package up and running are:

Make sure the computer and its associated equipment are connected correctly and plugged into power outlets.

Turn on all the peripheral devices such as the Peripheral Expansion System, the disk drives, printer, etc.

Turn on the computer. The title screen is displayed.

Insert the Editor/Assembler command module into the slot on the console. The screen goes blank momentarily and then the title screen reappears.

Press any one of the keys to have the master selection screen appear on the display.

Press the number that selects the Editor/Assembler package. The number varies based upon whether you have the model 99/4 or 99/4A. After pressing the right number, the Editor/Assembler selection screen is displayed.

9.1.2 Using the Editor to Compose the Source Program

Here's the procedure for calling up the Editor and using it to compose the source program.

Bring up the Editor/Assembler package and get the Editor/Assembler selection screen displayed.

Insert the diskette labeled Part A into Disk Drive 1.

Press the 1 key to select the Editor. After pressing 1, the Editor's selection screen appears on the display. The Editor selection screen lets you choose the following functions.

- Load an existing source program from disk into memory

- Edit a source program in memory

- Save a source program from memory to disk

- Print a source program

- Delete (purge) a source program in memory.

Press the 2 key to edit the source program. After pressing the 2 key, the Editor displays the message

```
ONE MOMENT PLEASE...
```

and the Editor utility program is loaded from disk into memory. Then the Editor clears the screen and you can begin typing in the source statements. The Editor displays the message

```
*EOF (VERSION X.Y)
```

where X.Y is the version number of the Editor, to mark the end of the program being edited.

Use the Editor to type the program statements and arrange the fields as you want them. Remember the syntactical rules for the statements. Use a <tab>, function, 7 to align the fields of the statements.

After you've typed the statements, return to the Editor selection screen by pressing the <escape> key, function 9, *twice*.

To save the program on a diskette, press the 3 key. The Editor displays the prompt

```
VARIABLE 80 FORMAT (Y/N)?
```

Press Y to choose the variable 80 format. This tells the Editor to use the least amount of disk space possible for storing the program. The Editor responds by displaying the prompt

```
FILE NAME?
```

Place a diskette that has been initialized by the Disk Manager into a disk drive and type in

```
DSKn.XXXXXXXX
```

where n is the number of the disk drive where you put the diskette to save the source program and XXXXXXXX is the name you choose for the source program file. For example, if you put a disk into Disk Drive 1, you might type in

```
DSK1.SQRSRC
```

You can use "SRC" as the last 3 letters of a Source file name to remind you that it is an assembly language source file. Remember to press the <enter> key after typing in a file name.

The Editor saves the source program on disk and returns to the Editor selection screen.

Press the <escape> key to return to the Editor/Assembler selection screen.

9.2 The Assembler

Here are the steps for assembling the program.

Get the Editor/Assembler selection screen displayed.

Insert the Part A diskette from the Editor/Assembler package in Disk Drive 1.

Press the 2 key to select the Assembler. After pressing 2, the prompt

```
LOAD ASSEMBLER (Y/N)
```

is displayed. (If the Assembler is already in memory, you don't get the prompt.)

When you press Y, the message

```
ONE MOMENT PLEASE...
```

is displayed and the Assembler is loaded from disk into memory.

After the Assembler is loaded, the prompt

```
SOURCE FILE NAME?
```

is displayed.

Make sure the diskette with the source file is in a disk drive and then type the file name of the source program as follows. (If you have only one disk drive, you must remove the Part A diskette from the drive and place the diskette with the source file in that drive.)

```
DSKn.XXXXXXXX
```

The n is the number of the disk drive that contains the source program diskette in it and XXXXXXXX is the name of the source program file. After typing in the source file name and pressing the <enter> key, the prompt

```
OBJECT FILE NAME?
```

is displayed. The Assembler is asking for a name to assign to the object program that is stored on disk as a result of assembling the source program.

Type in

    DSKn.YYYYYYYY

where n is the disk drive number that contains the diskette to receive the object program and YYYYYYYY represents the name you choose for the object program file. For example, if the program disk is in Disk Drive 1, you might type DSK1.SQUARE. After typing in the object file name and pressing the <enter> key, the prompt

    LIST FILE NAME?

is displayed. The Assembler is asking you to name the location for the listing.

If you have a printer, type in one of the following.

   1.   If you have a serial printer attached to the RS232 Interface unit, type in RS232.BA=n where n is the baud rate (speed in bits per second) of your printer. For example, it you have a 300 baud printer, type RS232.BA=300. If you have a 1200 baud printer, type RS232.BA=1200, etc.

   2.   If you have a parallel printer attached to the RS232 Interface unit, type in PIO. to select the printer.

   3.   If you have a thermal printer, type in TP to select the thermal printer.

If you don't have a printer, you can type

    DSKn.ZZZZZZZZ

where n represents the number of the disk drive that is to receive the listing and ZZZZZZZZ is the name you choose for the listing file. For example, if the program disk is in Disk Drive 1, you might type in DSK1.SQRLIST. After typing in the list file name, the prompt

    OPTIONS?

is displayed.

Each option has a character code. The option character codes and their meanings are as follows:

Code   Meaning

R   Instructs the Assembler to expect an R prefix with register numbers in the source program statements such as R8 for Register 8. If you do not use the R option and you put R's in front of the register numbers, the Assembler flags those statement as being wrong.

L   Instructs the Assembler to produce a listing when it assembles. Even though you type in a file name in response to the LIST FILE NAME? prompt, you still must use the L option to actually get a listing.

S   Instructs the Assembler to produce a symbol table with the listing. A symbol table is a list of all the names in your program and the address or value assigned with each name. A symbol table is especially useful when you have a long program and a listing of several pages. It helps you find the statement where a symbol is defined.

C   Instructs the Assembler to store the object program on disk in a compressed format. This option saves disk space.

In response to the OPTIONS? prompt, type in RLSC to choose all the options. Type in all four letters as they appear without spaces. The letters can be in any order. Then press <enter>.

After typing in the options, the message

ASSEMBLER EXECUTING

is displayed at the bottom of the screen. The Assembler assembles the source program on disk and builds an object program on disk. It also produces a listing. You can hear the assembler turning on the disk and, if a printer is connected, the listing is printed. If the assembler encounters any statements it does not understand or finds something it cannot assemble, it displays an error message on the screen. The error message also appears on the listing.

When the assembler is finished, the total number of errors is displayed and the message

```
PRESS ENTER TO CONTINUE
```

is displayed.

Press the <enter> key and the Editor/Assembler selection screen is displayed again.

If there are any errors in the assembly, use the Editor to load the source program from disk into memory and correct any statements that are syntactically incorrect.

Once you get an error-free assembly, take a moment to look at the listing produced by the Assembler. If you have a printed listing, remove it from the printer and place it before you. If you directed the listing to a disk file, get the listing displayed on the screen. Here's the way you get a listing displayed on the screen.

Get the Editor/Assembler selection screen displayed.

Press the 1 key to choose the Editor. After pressing 1, the Editor selection screen appears on the display.

Insert the Part A diskette is in Disk Drive 1 and press the 1 key to choose the Load option. After pressing 1, the message

```
ONE MOMENT PLEASE . . .
```

is displayed as the Editor is loaded from diskette into memory and then the prompt

```
FILE NAME?
```

is displayed.

Make sure the diskette with the list file is in a disk drive. Then type in the file name of the list file as follows:

```
DSKn.XXXXXXXX
```

The n is the number of the disk drive that contains the list file and XXXXXXXX represents the name of the list file. After typing in the file

name and pressing <enter>, the list file is loaded into the edit buffer. You can examine it using the Editor. If you should get the message

```
CONTROL CHARACTERS REMOVED
PRESS ENTER TO CONTINUE
```

press <enter>. The Editor is simply telling you it has removed some control characters that are needed if the file is sent to a printer. These control characters are still present in the file on the diskette.

The Editor selection screen is displayed.

Press the 2 key to choose the Edit option. The list file is displayed on the screen.

Study the listing and make sure you know what it's telling you. Here is a copy of a printed listing.


```
                    IDENTIFIES ASSEMBLER THAT PRODUCED LISTING

                                                        PAGE NUMBER
  99/4 ASSEMBLER
  VERSION 1.2                                          PAGE 0001
    0001 0000 0207          LI    R7,5        PUT NUMBER TO BE SQUARED IN R7
         0002 0005
    ...  0004 C207          MOV   R7,R8       COPY NUMBER INTO R8 FOR LOOP COUNT
    ...  0006 0209          LI    R9,0        INITIALIZE CUMULATIVE SUM TO ZERO
         0008 0000
    0004 000A A247   LOOP   A     R7,R9       ADD NUMBER TO CUMULATIVE SUM
    0005 000C 0228          AI    R8,-1       SUBTRACT ONE FORM LOOP COUNT IN R8
         000E FFFF
    0006 0010 16FC          JNE   LOOP        IF COUNT NOT ZERO, GO TO LOOP START
    0007 0012 04_0          BLWP  @0            ELSE GO HOME
         0014 0000
    0008                    END
```

```
MACHINE CODE                              SOURCE PROGRAM STATEMENTS
MEMORY LOCATIONS WHERE
MACHINE CODE IS LOADED
(RELATIVE)

LINE NUMBERS
```

```
  99/4 ASSEMBLER
  VERSION 1.2                                          PAGE 0002
    * LOOP   000A     R0    0000     R1    0001     R10    000A
       R11    000B     R12   000C     R13   000D     R14    000E
       R15    000F     R2    0002     R3    0003     R4     0004
       R5     0005     R6    000E     R7    0007     R8     0008
       R9     0009
  0000 ERRORS
```

The listing consists of two pages. The first page shows the source program statements, the machine code into which they are assembled, and other information. The second page is a symbol table. It's a list of all the symbols in the source program.

On the first page, the left-most column contains the line numbers for the source statements. The line numbers are in decimal. The second column shows the relative memory locations where the machine code will be loaded. The actual locations for the machine code is determined by the Loader when the object program is loaded. These relative locations are in hexadecimal. The third column lists the machine code values of the assembled source statements. The numbers are in hexadecimal. The source statements are listed to the right of the third column.

Some source statements produce more than one word of machine code. For example, statement number 1 (the LI R7,5 statement) produced two words of machine code. The first machine code word occupies relative word address 0000. The second machine code word occupies relative word address 0002. The first machine code word is hexadecimal 0207; the second machine code word is hexadecimal 0005.

Statement number 2 (the MOV instruction) requires only one word of machine code. The machine code word is hexadecimal C207 and occupies relative word address 0004.

Notice the END directive, statement number 8, doesn't produce any machine code words and doesn't require any memory words for machine code.

The second page is a symbol table. Each of the symbols used in the program is listed in alphabetical order from left-to-right on each line. Beside each symbol is a hexadecimal number which is the value of that symbol.

For example, the symbol LOOP is a label in the program and its value is hexadecimal 000A. The number is the relative word address of the instruction's machine code. The symbol LOOP has a relative address code of hexadecimal 000A.

The other symbols in the table are automatically assigned values by the assembler when you choose the R assembler option. The R option tells

the assembler to associate the symbol R0 with the value 0; the symbol R1, with the value 1; and so forth.

The last line of the listing tells you how many errors were found when the program was assembled. You want this number to be zero.

At this point, you have a listing, a source program, and an object program. The next step is to load and run the object program.

9.3 Summary

This chapter summarizes the use of the Editor and Assembler utility programs to edit and assemble the program from the previous chapter.

The Editor helps you compose or change a source program. The Assembler translates the source program into an object program for the computer to run. The Assembler also creates a listing to show the results of the assembly process.

The following chapter summarizes the use of the Loader and Debugger utility programs.

Chapter 10

INTRODUCTION TO
THE LOADER AND DEBUGGER

Chapter 9 uses a program example to illustrate the use of the Editor and Assembler utility programs in the Editor/Assembler package. This chapter uses the object program produced by the Assembler to illustrate the use of the Loader and Debugger included with the Editor/Assembler package.

10.1 Using the Loader

Here are the steps for using the Loader to load the object program.

Get the Editor/Assembler selection screen.

Press the 3 key to select the LOAD AND RUN choice. After pressing the 3 key, the prompt

FILE NAME?

is displayed.

Make sure you've got the diskette containing the object program in a disk drive, then type

DSKn.ZZZZZZZZ

The n is the number of the disk drive in which the diskette with the object program is inserted. ZZZZZZZZ is the name of the object program. Always press the <enter> key after typing in a file name. The Loader loads the object program into memory and the prompt

FILE NAME?

is displayed again. The Loader is asking for the name of another object program to load. You can load more than one program into memory.

Another program that you can load with this program is the Debugger program. The Debugger program is used to control your application program, for checking the results, and even helping you detect and remove bugs from your program should you have any. The Debugger is a helpful companion for your program.

The Debugger is on the Part A diskette that is included with the Editor/Assembler package. Place that diskette into a disk drive and respond to the FILE NAME? prompt by typing

DSKn.DEBUG

The n is the number of the disk drive where the Part A diskette is installed. DEBUG is the name of the DEBUG object program. The Loader loads the DEBUG program into memory along with the application program and the prompt

FILE NAME?

is displayed again. Since there are not any more programs to load, simply press the <enter> key.

The following prompt is displayed:

PROGRAM NAME?

The Loader is asking for the name of the program to run. You can run the program directly; but, if you do, it takes over the computer, does its job, and returns to the title screen. You will not see the results. Instead, run the Debugger, and then have the Debugger run the program when you're ready. You can also use the Debugger to help you check the results.

In response to the PROGRAM NAME? pompt, type

DEBUG

and press the <enter> key. The Debugger starts running.

10.2 Using the Debugger

When the Degugger starts running, it displays the identification message

*** 99/4 DEBUGGER ***

and then displays a dot on the screen (the Debugger is a program of few words). The dot is the Debugger's way of asking for what you want to do. You respond to the Debugger by typing in a single-character code followed by some other information. The information is based upon what you want the Debugger to do.

There are over 20 commands for the Debugger. Although you are not going to use all of them at this point, you can learn to use the ones required to run this first program.

First, use the Debugger to look at the object code that was loaded into memory by the Loader. Unless the program specifies otherwise, the Loader loads the first program into memory beginning at address hexadecimal A000. Let's use a Debugger command to examine the memory location and see if the object code is really there.

10.2.1 The Memory Inspect/Change Command (M)

To examine the contents of memory with the Debugger, follow this procedure. In response to the dot prompt, type M (for Memory Inspect/Change). Don't press the <enter> key.

Following the M, type the address for the contents you want to examine. Type in A000. The Debugger understands this number is hexadecimal. In fact, the Debugger assumes all numbers are hexadecimal.

After typing in A000, press the <enter> key. All Debugger commands are terminated by the <enter> key.

The Debugger responds by displaying

A000=0207

This tells you that the contents of memory address hexadecimal A000 is hexadecimal 0207.

Recall from looking at the listing, or look at the listing now if you had it printed,) that the first assembly language instruction (The LI R7,5 instruction) results in two words of machine code. The first word of machine code is hexadecimal 0207. So, it appears that the first word of machine code was loaded at address hexadecimal A000.

According to the listing, the second word of machine code is 0005. This means that the next word of memory (address A002) should have a 0005 in it. Does it? Let's find out.

After displaying the content of memory location hexadecimal A000, the Debugger waits to give you a chance to change the content of that memory location. If you don't want to change the content, but you want to inspect the next location, simply press the space bar.

The Debugger responds to the space by displaying on the next line

```
A002=0005
```

This shows that the contents of the next word (whose address is A002) is 0005.

Look at the next several memory locations to see what they contain. Keep pressing the space bar until the address A014 appears. The display should look like this.

```
A000 = 0207
A002 = 0005
A004 = C207
A006 = 0209
A008 = 0000
A00A = A247
A00C = 0228
A00E = FFFF
A010 = 16FC
A012 = 0420
A014 = 0000
```

These are the addresses and machine code values for the object program. Confirm from the listing that the machine code values and their relative locations are correct.

Address A014 is the last location into which your program's machine code was loaded. If you press the space bar too many times and pass address A014, that's okay. You're just looking at the machine code for the Debugger program.

When you're finished examining memory locations, press the <enter> key to terminate the Memory Inspect/Change command and the Debugger gives you another dot prompt for another command.

10.2.2   The (Internal) Registers Command (R)

Another Debugger command you can use is the R command. It lets you inspect and, optionally, change the contents of the Workspace Pointer, Program Counter, and Status Register.

To use the command, type in R. Don't press the <enter> key.

The Debugger responds by displaying

```
W=XXXX     (where XXXX is some four-digit hexadecimal
              number)
```

The Debugger is showing you the contents of the Workspace Pointer (WP). The Workspace Pointer tells the computer what area of memory to use for a program's workspace. For now, type in 2000. Don't press <enter>. You're telling the Debugger to let your program use the area of memory beginning at hexadecimal 2000 for its workspace.

Press the space bar. The Debugger displays on the next line

```
P=YYYY     (where YYYY is some four-digit hexadecimal
              number)
```

The Debugger is showing you the contents of the Program Counter (PC). You may remember that the Program Counter is the computer register that tells the computer the address of the next instruction to be performed in a program. The address of the first instruction is A000.

Type A000 followed by a space. The Debugger puts A000 into the Program Counter and on the next line displays

```
S=ZZZZ     (where ZZZZ is some four-digit hexadecimal
              number)
```

The Debugger is showing you the content of the Status Register (SR). The Status Register is the computer register in which the computer stores status conditions resulting from the performance of instructions. The Status Register contains the status bits that the conditional jump instructions use to make decisions.

Before running a program, it's a good idea to set the Status Register to zero. Simply type a 0 followed by the <enter> key. The Debugger puts a zero into the Status Register and displays a dot prompt again.

Note

The R command does not directly put values into the CPU's Workspace Pointer, Program Counter, or Status Register. The R command saves these values in memory; these values are placed in the CPU's internal registers when you tell the Debugger to run a program.

10.2.3 The Breakpoint Command (B)

You can use a Breakpoint command to control how much of the program to run.

The B command lets you set a breakpoint. A breakpoint is a roadblock in a program. When the program comes to a breakpoint, it stops and gives control to the Debugger.

Without a breakpoint, if you tell the Debugger to run the program, the program starts at the first instruction and keeps performing instructions until it performs the BLWP @0 instruction. When it performs that Go-Home instruction, the computer returns to the title screen just as when you first turn on the computer.

Before you allow the computer to go home, use a breakpoint to stop the program so you can check the results produced by the program

Set a roadblock (a breakpoint) at the Go-Home instruction. You can allow the program to run until it comes to the Go-Home instruction and then have the Debugger stop the program before it goes any further. Here's how you do that.

In response to the dot prompt, simply type a B.

Now tell the Debugger where to set the breakpoint. Set the breakpoint at the BLWP instruction. The beginning address of the BLWP instruction is A012. You can confirm this from the listing. Following the B, type A012 and press the <enter> key. These steps set a "trap" at the BLWP instruction. A trap is the location of a breakpoint.

You may get the message, "BKPT USES 2 WORDS." That's fine, just keep going.

You have used the Debugger to check the machine code in memory and verify that it's what you expected. You have used the R command to inspect the content of the Workspace Register, to set the content of the Program Counter to the starting address of the program, and to zero out the content of the Status Register. And you've used the Breakpoint command to set a trap for the program so the Debugger can stop the program before it goes to the title screen.

10.2.4   The Execute Command (E)

To actually run the program, here's all you do. In response to the dot prompt, simply type in E (for Execute) and press the <enter> key.

The Debugger starts running the program at the address you set in the Program Counter and allows the program to run until it comes to the breakpoint. When the program reaches the breakpoint, the Debugger takes over and displays

       B   2000   A012   3000

The B means the Debugger has hit a breakpoint. The first number is the address of the program's workspace. The second number is the address in the Program Counter when the breakpoint was encountered (it's the address of the breakpoint). The third number is the contents of the Status Register.

Now, you can use the Debugger to check the results.

If everything went according to plan, the program should have squared the number 5 and left the square in Register 9.

10.2.5   The Working Register Inspect/Change Command (W)

You can use the W command to inspect and, optionally, change the contents of working registers. Here are the steps to follow.

In response to the Debugger's dot prompt, type in the letter W. (Don't press the <enter> key yet.)

The W command tells the Debugger you want to inspect the content of a register but the Debugger needs to know which one. After the W, type the number of the register you want to inspect. Type 9 (not R9, the Debugger doesn't understand R prefixes). After typing 9, press the <enter> key.

The Debugger responds by displaying

     R9=0019

This message tells you that the content of Register 9 is 0019. Register 9 has the square of 5 in it. Remember, the Debugger only speaks hexadecimal. Register 9 holds a *hexadecimal* 19, which is a decimal 25.

After examining Register 9, press <enter> to get a dot prompt from the Debugger.

10.2.6   The Hex-to-Decimal Conversion Command (>)

Often when using the Debugger, you need to convert a hexadecimal number to a decimal equivalent. The Debugger provides a convenient way to do those conversions. Here's how.

First, make sure you have a dot prompt from the Debugger. In response to the dot prompt, type a greater-than sign (>). This is a command to the Debugger to convert a hex number to a decimal number. Then, type the hex number, say 19, and press the <enter> key. The Debugger responds by displaying

```
>.19
    =25
```

This shows that hex 19 equals decimal 25.

There are some other Debugger commands which are especially useful.

10.2.7   The Set Bias Commands (X, Y, and Z)

By looking at a listing, you can see the machine code that was produced from the assembly language statements. You also can see the relative locations in which the machine code values are placed in memory when the object program is loaded. Normally, when the first object program is loaded into memory, it's loaded beginning at address hexadecimal A000. When a program is loaded into memory, it's sometimes a brain twister to transform a relative address from the listing into the physical address where the program was actually loaded.

Again, the Debugger can help. One feature of the Debugger is setting a bias. Here's the way it works.

Suppose you have the program in memory beginning at address hexadecimal A000 and you want to look at the machine code for the instruction

```
JNE LOOP
```

From the listing, you discover that the instruction's machine code is a hexadecimal 10· distance from the beginning of the program. You could do a mental calculation and add

a hex 10 to the beginning of the program (hex A000) and come up with the physical location hex A010 (hex A000 + hex 0010). An easier way, however, is to use a bias.

To set a bias, follow this procedure. In response to the dot prompt, type in an X. The Debugger displays

```
.X ZZZZ        (where ZZZZ is some four-digit hexadecimal
               number).
```

To set an X bias at A000, type in A000 and press the <enter> key. You've just set a bias of A000. Now, whenever you use any Debugger command that requires an address (like the Memory Inspect/Change command) you can use X as part of the address calculation. For example, if you want to examine the contents of the memory address that is a relative hex 10 from the starting point of the program (hex A000) you can type in X as part of the address.

For example, in response to the dot prompt, type in M, then type in 10X, and press the

<enter> key.

The Debugger responds by displaying

```
A010=16FC
```

It automatically added hex 10 (the displacement) to hex A000 (the bias).

Press <enter> to get a dot prompt from the Debugger.

The Debugger lets you establish up to three bias. The biases are called X, Y, and Z.

10.2.8 More Experiments with the Program

If you run the square program again just the way it is, it would repeat the results from the first time you ran it. It would square the number 5. If you want the program to do something different (like square a different number), you must have it do something different.

Suppose you want the program to square the number 6. You must change the program so that it starts with a 6 in Register 7. One way to do that is to change the first statement in the program from

```
LI    R7,5
```

to

```
LI    R7,6
```

This change requires changing the source program, reassembling, and reloading. This approach is a time-consuming process. You can change the machine code directly in memory. This approach takes more knowledge of the machine code than explored to this point.

Use the Debugger to put a 6 into Register 7 before running the program again and have the Debugger start running the program at the second instruction, rather than the first.

Here's how you can do that. Use the W command to change the contents of Register 7. You can do it this way.

Get a dot prompt from the Debugger. Then, type in W 7 and press the <enter> key. The Debugger responds by displaying

```
R7=0005
```

This tells you that the current content of Register 7 is 5. The program put a 5 in Register 7 when it ran the first time. The Debugger gives you a chance to change the content of Register 7 if you want to. Change the content of Register 7 to 6 by simply typing in 6 followed by the <enter> key.

If you were to run the program again, it would start off with a 6 in Register 7. But there's a problem. If you start running the program at the first instruction, then the first thing the program does is put a 5 into Register 7 and you'll end up with the square of 5 again.

But, if start running the program at the second instruction, the program uses the 6 in Register 7 and squares it.

How can you start running the program at the second instruction? Use the R command to set the Program Counter to the address of the second instruction. Do it this way.

Get a dot prompt and then type the letter R.

The Debugger responds by displaying

```
W=2000
```

Hexadecimal 2000 is the address in memory for the workspace that the program is using for the registers. Since you don't want to change that address, simply press the space bar.

The Debugger responds by displaying

```
P=A012
```

Hexadecimal A012 is the address where the program stopped the last time you ran it. To change this number to the address of the second instruction, you can type either A004 or you can type 4X if you set the X bias to A000. Press the space key after the entry.

The Debugger responds by displaying

```
S=ZZZZ     ( ZZZZ is some four-digit hexadecimal number)
```

Zero out the Status Register by typing a 0 and press the <enter> key. After pressing the <enter> key, the Debugger gives you a dot prompt.

Now you're ready to run the program again, but this time the program starts running at the second instruction (at address hex A004) and with a 6 in Register 7.

Before running the program, set a breakpoint by typing in B A012 (or you can type B 12X, if you set the X bias to A000) and press the <enter> key.

After setting the breakpoint, run the program by typing in E and press the <enter> key.

The program runs until it reaches the breakpoint, then the Debugger takes over. The Debugger reacts by displaying the three addresses, including the address of the breakpoint (A012), and a dot prompt.

Check the results of the program by inspecting the contents of Register 9. Use the W command to see what's in the register. There should be a hexadecimal 0024 in Register 9. Use the hexadecimal-to-decimal conversion command (>) to confirm that a hexadecimal 24 is equal to a decimal 36 (the square of 6).

10.2.9   The Decimal-to-Hex Conversion Command (.)

There's another Debugger command you can use to convert a decimal number to a hexadecimal equivalent.

Suppose you want to convert a decimal number like 100 to a hexadecimal equivalent.

Here's how you can use the Debugger to do that conversion for you.

In response to the dot prompt, type a decimal point and type the decimal number you want to convert.

Type in .100 and press the space bar or the <enter> key. The Debugger shows you the hexadecimal equivalent value. The Debugger shows you that a decimal 100 is equal to a hex 64.

10.2.10 The Hexadecimal Arithmetic Command (H)

Another useful Debugger command is the hexadecimal arithmetic command. It lets you type in two hex numbers and it shows:

- the sum of the two numbers

- the difference of the first number minus the second

- the product of multiplying the two numbers

- the quotient and remainder resulting from dividing the first number by the second

The results are given in hexadecimal.

Try this. In response to the dot prompt, type in the letter H. The H is the command to perform hexadecimal arithmetic. Then type two hex numbers separated by a space, or a comma, and followed by pressing the <enter> key. For example, type

```
H 20,6
```

and press the <enter> key.

The Debugger responds by displaying

```
H1=0020  H2=0006  H1+H2=0026
H1-H2=001A  H1*H2 = 0000 00C0
H1/H2=0005 R 0002
```

The Debugger shows you that it's naming the first number (hex 20) H1 and the second number (6) H2.

- H1 plus H2 (or hex 20 plus 6) is hex 26

- H1 minus H2 (or hex 20 minus 6) is hex 1A

- H1 times H2 (or hex 20 times 6) is hex C0. Notice that the product is given as an 8-digit hex number

- H1 divided by H2 (or hex 20 divided by 6) results in a quotient of 5 and a remainder of 2. (Hex 20 is a decimal 32 and 32 divided by 6 is 5, with a remainder of 2.)

10.2.11   The Quit Command (Q)

The Q command lets you leave the Debugger.

It works like this. In response to the Debugger dot prompt, simply type in Q and press the <enter> key. The Debugger runs the program beginning at the address in the Program Counter. But if the Program Counter contains zero, the Debugger returns to the Editor/Assembler selection screen.

At this point, you can let the program run without a breakpoint. Use the R command to set the Program Counter to A000 and the Status Register to zero.

Then type Q and press the <enter> key. The program runs and the master title screen appears after the BLWP instruction is performed.

10.3   Summary

Chapters 9 and 10 describe:

- how to use the Editor to create or change a source program

- how to use the Assembler to assemble a program

- how to read a listing

- how to load an object program into memory along with the Debugger

- how to use several Debugger commands

In the following chapters, the rest of the instructions are examined in detail. Also, you can learn more about assembler directives, the Loader, the Debugger, and more of the techniques of assembly language programming.

The next chapter examines the Data Manipulation instructions.

Chapter 11

DATA MOVEMENT INSTRUCTIONS

This chapter introduces the Data Movement group of instructions. The main job of these instructions is to move data or to rearrange data. There are 12 Data Movement instructions. The instructions are listed below with their names, operation codes, and a description of the kinds of addressing modes you can use with the instructions.

In the following list, G means that an operand is a general addressing mode operand (one that can use any of the five general addressing modes). An R means that an operand must be a working register and can use only the register direct addressing mode). An IOP means that an operand must use immediate addressing. IOP is a data value, rather than the address of a data value. A C means that the operand is a count value and it must be a number from 0 through 15.

Name	Operation Code	Addressing Mode
Move Word	MOV	G,G
Move Byte	MOVB	G,G
Swap Bytes	SWPB	G
Load Immediate	LI	R,IOP
Load Workspace Pointer Immediate	LWPI	IOP
Load Interrupt Mask Immediate	LIMI	IOP
Store Workspace Pointer	STWP	R
Store Status	STST	R
Shift Right Logical	SRL	R,C
Shift Right Arithmetic	SRA	R,C
Shift Right Circular	SRC	R,C
Shift Left Arithmetic	SLA	R,C

11.1 The Move Instructions (MOV and MOVB)

The Move Word (MOV) and Move Byte (MOVB) instructions copy a data item from one location to another. The MOV instruction moves a word (16 bits) and the MOVB instruction moves a byte (8 bits).

11.1.1 The Move Word Instruction (MOV)

The Move Word instruction was introduced in Chapter 7. It moves, or copies, a word (16 bits) from one location to another. It requires two operands; both operands can use any of the five general addressing modes. The first operand is called the source operand; the second is called the destination operand. The source operand specifies the location of the word that is moved; the destination operand specifies where the copy is placed. Since the source operand appears to the left of the destination operand in the operand field, you can visualize a left-to-right movement between the operands from the location specified by the source operand into the location specified by the destination operand.

```
              +--------------+
              |              |
              |              V
 MOV     Source Operand,Destination Operand
```
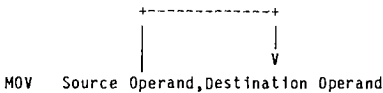
The data word that is moved is automatically compared to zero and its relationship to zero affects the Logical Greater Than (L>), Arithmetic Greater Than (A>), and the Equal (EQ) status bits.

As an example, suppose memory location hexadecimal A102 contains the value hexadecimal 9ABC, and suppose memory location B87E contains hexadecimal 5D6F. Further suppose Register 10 contains a hexadecimal A102 and Register 3 contains a hexadecimal B800 before the following instruction is performed.

```
 MOV   *R10,@>7E(R3)
```

After the instruction is performed, memory location hex B87E contains a hex 9ABC. The contents of memory location A102 is still 9ABC and the contents of Registers 3 and 10 are unchanged.

```
 Location       Before     After
 (R3)     =     >B800      >B800
 (R10)    =     >A102      >A102
 (>A012)  =     >9ABC      >9ABC
 (>BB7E)  =     >5D6F      >9ABC
```

The data word is moved; hex 9ABC is compared to zero and affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits as follows.

Logical Greater Than (L>) Status Bit = 1

Arithmetic Greater Than (A>) Status Bit = 0

Equal (EQ) Status Bit = 0

Imagine that the computer responds to these questions in deciding whether to set or clear these status bits.

The computer asks if hex 9ABC is logically greater than zero; that is, if 9ABC is thought of as a logical value (an absolute or unsigned number, is it bigger than zero. It is, so the computer sets the Logical Greater Than status bit true (a one).

The computer asks if hex 9ABC is arithmetically greater than zero; that is, if 9ABC is thought of as an arithmetic value (a signed number), is it larger than zero. It isn't, so the computer clears the Arithmetic Greater Than status bit to zero (meaning that the condition is not true).

If you think about hex 9ABC as a signed number using two's complement notation, what is the sign of the number? It's negative because the sign bit is one. In binary, hex 9ABC is 1001101010111100. You can determine if a hex number that represents a signed number is positive or negative by looking at the left-most digit. If the most-significant digit is 7 or less, the number is positive; and if it is 8 or greater, the number is negative.

The computer asks if hex 9ABC equals zero. It doesn't, so the computer clears the Equal status bit.

11.1.2   The Move Byte Instruction (MOVB)

The Move Byte instruction (MOVB) is the little brother of the Move Word instruction. It does the light duty work. It moves an 8-bit chunk of data.

The MOVB instruction requires two operands that can use any of the five general addressing modes. The first operand, called the source operand, specifies where the byte is to be copied; and the second operand, called the destination operand, specifies where the data is copied.

Both the source and destination operands are byte addresses. As an example, suppose memory word A012 contains a hex 9ABC, memory word B87E contains 5D6F, Register 10 contains hex A103 and Register 3 contains hex B800 before the following instruction is performed.

```
MOVB  *R10,@>7E(R3)
```

After the instruction is performed, memory word hex B87E contains hex BC6F. The instruction moves the contents of byte address hex A103, hex BC, to byte address hex B87E, the left byte of word address hex B87E.

Location		Before	After
(R3)	=	>B800	>B800
(R10)	=	>A103	>A103
(>A012)	=	>9ABC	>9ABC
(>B87E)	=	>5D6F	>BC6F

Just like the Move Word instruction, the Move Byte instruction has the computer compare the moved value to zero and affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

In this example, the byte value, hex BC, is compared to zero, causing the Logical Greater Than status bit to be set, the Arithmetic Greater Than status bit to be cleared, and the Equal status bit to be cleared.

Because the Move Byte instruction performs a byte operation, which means it uses an 8-bit value, the Odd Parity (OP) status bit is also affected based on the number of one bits in byte. The byte value hex BC is a binary 10111100. There are 5 one bits, an odd number, in the byte, so the Odd Parity status bit is set one (true).

The Odd Parity status bit is affected only by byte operations, and not word operations. It is affected by the number of one bits in a byte. If there is an odd number of one bits (1, 3, 5, or 7), the Odd Parity status bit is set. If there is an even number of one bits (2, 4, 6, or 8), the Odd Parity status bit is cleared.

You might wonder exactly what use is the Odd Parity status bit? Often, byte values represent ASCII character codes. For some communication applications, parity, the number of one bits, is used to detect possible errors in the transmission of the character codes. After a byte operation, the Odd Parity status bit tells you if the byte had even or odd parity.

11.2 The Swap Bytes Instruction (SWPB)

The Swap Bytes instruction requires only one operand and can use any of the five general addressing modes. The instruction simply exchanges the two bytes in a word, exchanging the left byte with the right byte. No status bits are affected by the instruction.

Although the operand can specify a general memory location, the SWPB instruction is used most often to exchange the two bytes in a register. SWPB is used to put the right

byte of a register into the left-byte position, so the byte can be accessed by a byte-operation instruction using register direct addressing mode.

For example, suppose you want to copy the right byte of Register 6 to memory location MEOW. You can use the instruction

```
MOVB  R6,@MEOW
```

to move a byte in Register 6 to memory location MEOW, but it's the left byte of Register 6 that is moved. (Any time the computer performs a byte operation using register direct addressing, it can only access the left byte of the register.) If you want to move the right byte of Register 6 to memory location MEOW, you can use a Swap Byte instruction to exchange the bytes first, as follows.

```
SWPB  R6
```

11.3 The Load Immediate Instruction (LI)

The Load Immediate instruction (LI) is probably an old friend by now. It was used in the program example that was presented in a previous chapter.

The LI instruction has two operands. The first is an R-type operand. This means that the first operand must be a register. It can only use register direct addressing. The second operand is an IOP-type operand. This means that it's an immediate operand and uses the immediate addressing mode.

The LI instruction places the immediate operand into the register. It's useful for initializing the contents of a register to a constant. In the previous program example, you've seen how to use it for establishing a loop count in a register, for example.

An immediate operand is always a 16-bit value. There are no 8-bit immediate operands.

The Load Immediate instruction copies the immediate operand into the register. Just as with the Move Word instruction, the computer automatically compares the value of the word to zero and affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits in the same way.

The Load Immediate instruction is used to set an address value in a register. Suppose you want to access several sequential data items in a list. Register indirect autoincrement addressing mode is designed especially for that. To use that addressing mode, you need to have an address in a register. The Load Immediate instruction can help. Suppose the

data items are located in memory beginning at address hex CA62. You can use the LI instruction to put the first data item's address in a register, such as Register 8.

```
LI  R8,>CA62
```

Then the instruction

```
MOV  *R8+,R0
```

moves the first word to R0 and automatically points R8 to the second word so that you are ready to access it with another instruction.

In the previous example, suppose memory location hex CA62 is labeled LIST, then the instruction

```
LI  R8,LIST
```

puts a hex CA62 in R8. Remember, with a Load Immediate instruction, the second operand is an immediate operand; it's treated as the data value itself, not the address of data.

An instruction like

```
MOV  @LIST,R8
```

moves the contents of address LIST into R8, but

```
LI  R8,LIST
```

moves the address value of LIST into R8.

11.4   The Load Internal Registers Instructions (LWPI and LIMI)

Two instructions load values into two of the computer's special internal registers. The Load Workspace Pointer Immediate (LWPI) instruction loads a value into the Workspace Pointer, and the Load Interrupt Mask Immediate (LIMI) instruction loads a value into the interrupt mask portion of the Status Register (LIMI).

11.4.1 The Load Workspace Pointer Immediate Instruction (LWPI)

The Load Workspace Pointer Immediate Instruction (LWPI) is designed specifically to put an address value into the Workspace Pointer. You may remember that the Workspace Pointer is a special computer register that holds the address that tells the computer the location of the program's register set in memory.

The LWPI instruction has only one operand, an immediate operand. A copy of the immediate operand is placed in the Workspace Pointer. Sometimes you may want to define explicitly the location of a program's register set. The LWPI instruction lets you do that.

As an example, suppose you want to use the area of memory beginning at hex DE80 for a register set. The instruction

    LWPI >DE80

sets the Workspace Pointer to hex DE80.

Just as with any instruction using immediate addressing, the immediate operand can be given a name. For example, if memory location hex DE80 is named WRKSPC, then the instruction

    LWPI WRKSPC

puts a hex DE80 into the Workspace pointer.

11.4.2 The Load Interrupt Mask Immediate Instruction (LIMI)

The Load Interrupt Mask Immediate instruction (LIMI) sets a value into the interrupt mask. The interrupt mask is the low-order (rightmost) four bits of the Status Register. The interrupt mask is used by the computer to help control peripheral devices.

Like the LWPI instruction, the LIMI instruction has one operand, an immediate operand. Recall that all immediate operands are 16-bit values. The interrupt mask, however, is only 4-bits big. With the Load Interrupt Mask Immediate instruction, only the low-order nibble (4 bits) of the immediate operand is placed into the interrupt mask.

For example, the instruction

    LIMI 4

causes a 4 to be placed into the interrupt mask.

The instruction

```
LIMI  >1234
```

also causes a 4 to be placed into the interrupt mask.

11.5   The Store Internal Registers Instructions (STWP and STST)

Two instructions copy values from two of the computer's special internal registers into a program's working registers. One instruction, Store Workspace Pointer (STWP), copies the value in the Workspace Pointer into a working register. The other instruction, Store Status (STST), copies the value in the Status Register into a working register.

These two instructions are not used often in most programs.

11.5.1   The Store Workspace Pointer Instruction (STWP)

The Store Workspace Pointer instruction (STWP) puts the address of the program's working registers (which is in the Workspace Pointer) into one of the working registers. It's a way of remembering the address of the working registers.

Here's an example of how it works. Suppose these two instructions were in a program.

```
LWPI  >C2E0
STWP  R9
```

The STWP instruction stores a hex C2E0 into Register 9.

11.5.2   The Store Status Instruction (STST)

The Store Status instruction (STST) copies the 16-bit value in the Status Register into a working register. It's a way of remembering what's in the status register.

As an example, the instruction

```
STST  R15
```

copies the current contents of the Status Register into Register 15.

11.6   The Shift Instructions (SRL, SRA, SRC, and SLA)

There are four instructions which rearrange bits in a register. These are the shift instructions. They are most often used in applications where individual bits represent one-bit information items.

A shift instructions requires two operands. The first is an R-type operand and can use only register direct addressing. The second is a C-type operand and is a number in the range of 0 through 15.

The first operand identifies the register that contains the bits to be shifted. The second operand specifies how many bit positions to shift. A word must be in a register before you can shift it. You can't directly shift the contents of a general memory location.

The four shift instructions are alike in many ways. They each require that a value be in a register before it can be shifted. The second operand identifies how much to shift. The second operand can be 0 or a non-zero number of 1 through 15. If the second operand is a non-zero number, the contents of the register are shifted that number of positions. If the operand is 0, the contents of the register are shifted the number of positions equal to the number in the rightmost nibble of Register 0. When the operand is 0 and the rightmost nibble of Register 0 contains a non-zero number, that number in Register 0 is the number of bits shifted. When the operand is 0 and the rightmost nibble of Register 0 is also 0, the bits are shifted 16 positions.

Also, after the shift operation is performed, the computer compares the result in the register to zero and affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits accordingly. Each of the instructions also affects the Carry status bit. The state of the last bit shifted out of the register is recorded in the Carry status bit. If the last bit shifted out is a one, the Carry status bit is set; if the last bit shifted out is a zero, the Carry status bit is cleared. You might say that the last bit shifted out leaves its footprint in the Carry status bit.

Those are the ways the instructions are alike. Now look at the ways in which they're different.

11.6.1   The Shift Right Logical Instruction (SRL)

The Shift Right Logical instruction (SRL) shifts the bits in a register to the right the number of positions determined by the second operand. The bits shifted out of the right end of the register are gone. (They are said to fall into the "bit bucket", the fictitious final resting

place for departed bits.) As the bits are shifted to the right, *zero bits* fill the vacated bit positions on the left. The state of the last bit shifted out of the right end of the register is recorded in the Carry status bit. After the shift, the 16-bit result in the register is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits accordingly.

```
              Bits Move This Way
              ------------------>
              +--+--+-+-/ /-+--+--+
    0 ---->   |  |  |  | ... |  |  |  |
              +--+--+-+-/ /-+--+--+
```

Take an example. Suppose register 2 contains a hex C873 or a binary 1100100001110011.

The instruction

```
    SRL  R2,4
```

shifts the contents of Register 2 four bit positions to the right and zero bits fill the vacated bit positions on the left.

After the instruction is performed, Register 2 contains a hex 0C87 or a binary 0000110010000111. The Carry status bit is not set, because the last bit shifted out was a zero bit. The Logical Greater Than status bit is one, the Arithmetic Greater Than status bit is one, and the Equal status bit is zero as a result of comparing the result (hex 0C87) to zero.

```
                          (R2) Before
            0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
            | 1| 1| 0| 0| 1| 0| 0| 0| 0| 1| 1| 1| 0| 0| 1| 1|
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
              |  .  .  .  .  .  .  .  .  .  .  .  |
              +-----------+                    +-----------+
              |                                |
              V     (R2) After                 V
            0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    0 --->  | 0| 0| 0| 0| 1| 1| 0| 0| 1| 0| 0| 0| 0| 1| 1| 1|
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```
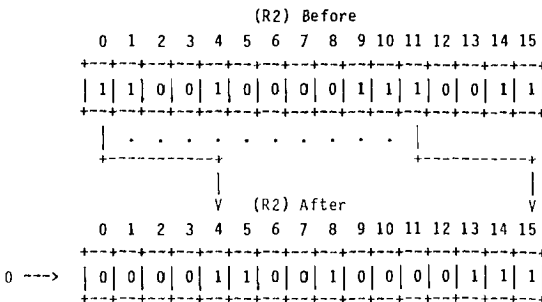
The instruction is called Shift Right Logical because it treats the contents of the register as a logical, or unsigned, value. Shifting a number to the right is a simple way of performing a division by a power of two. For example, if Register 2 contains a hex 8004

(a binary 1000 0000 0000 0100) and you shift the contents one position to the right, the contents become a binary 0100 0000 0000 0010 (a hex 4002).

The number is divided by two. Shift the number right again and it's divided by two again. It becomes a binary 0010 0000 0000 0001 or hex 2001.

This works as long as you're thinking about the number as an unsigned value (which means that you don't care what happens to the sign bit because the number has no sign).

Notice that with the SRL instruction, zero bits always fill the vacated bit positions on the left.

11.6.2 The Shift Right Arithmetic Instruction (SRA)

The Shift Right Arithmetic instruction (SRA) works almost exactly like the Shift Right Logical instruction. The only difference is what happens to the vacated bit positions. With the SRA instruction, the vacated bit positions are filled with bits equal to the state of the original sign bit (the leftmost bit).

```
        Bits Move This Way
        ---------------->
        +--+--+-/ /-+--+--+
+---->  |  |  | ... |  |  |
|       +--+--+-/ /-+--+--+
|Sign Bit|
+--------+
```
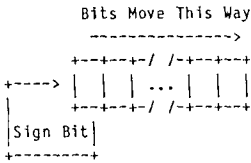
Consider an example like the one from the previous section. Suppose register 2 contains a hex C873 or a binary 1100100001110011.

The instruction

```
    SRA  R2,4
```

causes the contents of Register 2 to be shifted four bit positions to the right and one bits fill the vacated bit positions on the left since the sign bit is a one.

After the instruction is performed, Register 2 contains a hex FC87 or a binary 1111110010000111. The Carry status bit is not set, because the last bit shifted out was a zero bit. The Logical Greater Than status bit is one, the Arithmetic Greater Than status bit is zero, and the Equal status bit is zero as a result of comparing the result (hex FC87) to zero.

```
                    (R2)  Before
        0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
        +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
        | 1| 1| 0| 0| 1| 0| 0| 0| 0| 1| 1| 1| 0| 0| 1| 1|
        +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
         |  .  .  .  .  .  .  .  .  .  .  |
         +-----------+                    +-----------+
                     |                                |
                     V    (R2)  After                 V
        0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
        +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 +----> | 1| 1| 1| 1| 1| 1| 0| 0| 1| 0| 0| 0| 0| 1| 1| 1|
 |      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                     |
 +---------------------+
```
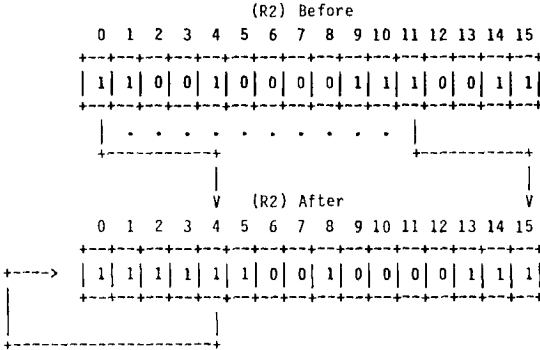
The instruction is called Shift Right Arithmetic because it treats the value in the register as a signed number. Shifting a number to the right is a simple way of dividing the number by a power of two. If you're thinking about the number as a signed number, however, you need to maintain the sign.

Suppose that Register 2 contains a hex FFFA (a binary 1111 1111 1111 1010). If you're thinking about the number as a signed number, it's a minus 6. It's a minus number because the sign bit is a one. The absolute value, 6, is found by taking the two's complement of the number.

What do you get if you divide -6 by two? You get -3. If you shift the contents of Register 2 one bit position to the right, you get a binary 1111 1111 1111 1101 (or a -3). You get the correct signed number as long as you maintain the sign bit.

Notice that with the SRA instruction, *the state of the original sign bit* fills the vacated bit positions on the left.

11.6.3   The Shift Right Circular Instruction (SRC)

The Shift Right Circular instruction (SRC) rotates the contents of a register. This instruction works just about like the SRL and SRA instructions where bits are shifted right in a register. The difference is what happens to the bits shoved out of the right end. With the SRC instruction, when a bit is shifted out of the right end of a register, rather than landing in the bit bucket, it walks around and hops right back into the register on the left side. Effectively, the bits are simply rotated to the right in the register.

```
       Bits Move This Way
       -~----------------->
       +--+--|--|-/ /-+--+--+
  +-->|  |  |  |...|  |  |---+
  |    +--+--+--+-/ /-+--+--+      |
  |                                |
  +----------------------------+
```
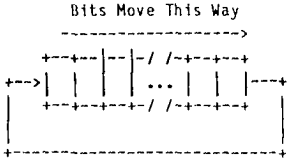
Suppose Register 2 contains a hex C873 (a binary 1100 1000 0111 0011). The instruction

    SRC R2,4

shifts the bits four positions to the right and the bits displaced on the right fill the vacated bit position on the left. The result in Register 2 is a binary 0011 1100 1000 0111 (or hex 3C87).

```
                          (R2) Before
             0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
            | 1| 1| 0| 0| 1| 0| 0| 0| 0| 0| 1| 1| 1| 0| 0| 1| 1|
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
               |  .  .  .  .  .  .  .  .  .  .  |
               +----------+                    +----------+
                          |                               |
                          V    (R2) After                 V
             0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  +-->  | 0| 0| 1| 1| 1| 1| 0| 0| 1| 0| 0| 0| 0| 1| 1| 1| ---+
  |         +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+      |
  |                                                               |
  +------------------------------------------------------------+
```
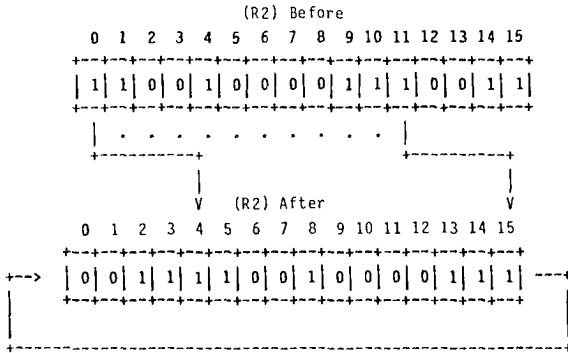
The Carry status bit is zero because the last bit shifted out is a zero. With the SRC instruction, the Carry status bit is always left equal to the state of the leftmost bit in the register.

The Logical Greater Than status bit and the Arithmetic Greater Than status bit are one; the Equal status bit is zero as a result of comparing the result in Register 2 to zero.

11.6.4   The Shift Left Arithmetic Instruction (SLA)

The Shift Left Arithmetic instruction (SLA) is the only shift instruction that directly shifts the contents of a register to the left. The bits shifted out of the left end of the register fall into the bit bucket and zero bits fill the vacated bit positions on the right.

```
    Bits Move This Way
 <---------------------
+--+--|--|-/ /-+--+--+
|  |  |  |  ... |  |  |<--- 0
+--+--+--+-/ /-+--+--+
```
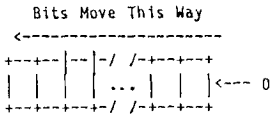
The instruction is called Shift Left Arithmetic because it treats the value in the register as a signed number. It pays attention to the sign bit. The SLA instruction is the only shift instruction that affects the Overflow status bit. It sets the Overflow status bit to one should the sign bit change any time during the shift operation; otherwise, the instruction clears the Overflow status bit.

For example, suppose Register 2 contains a hex C873 (a binary 1100 1000 0111 0011). The following instruction shifts the contents of Register 2 four positions to the left and fills the vacated bit positions on the right with zero bits.

```
SLA  R2,4
```

The result in Register 2 is a binary 1000 0111 0011 0000 (or hex 8730).

```
                      (R2) Before
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  | 1| 1| 0| 0| 1| 0| 0| 0| 0| 1| 1| 1| 0| 0| 1| 1|
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                   |  .  .  .  .  .  .  .  .  .  . |
                   |                              |
  +-----------+                        +----------+
  |                                     |
  V                  (R2) After         V
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  | 1| 0| 0| 0| 0| 1| 1| 1| 0| 0| 1| 1| 0| 0| 0| 0| <--- 0
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```
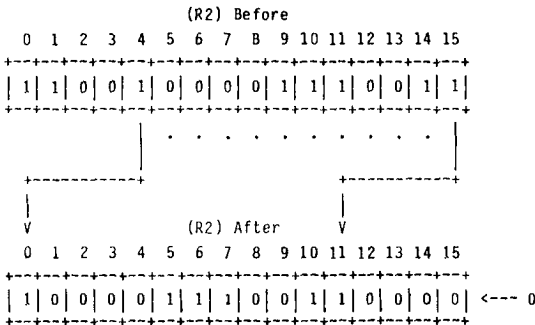
The Overflow status bit becomes a one because the sign bit changed during the shift operation. Even though the sign bit is the same after the shift as it was before the shift, the sign bit did change during the shift operation (at least one zero bit passed through the sign bit).

Shifting a number to the left is a simple way of multiplying by a power of two. If you're thinking of the number as a signed number, that's true as long as you preserve the sign (don't change the sign bit).

The computer doesn't know how you're thinking about the number, as a signed number or as an unsigned number. But, in case you are thinking about it as a signed number, the computer tells you if the sign bit changes by setting the Overflow status bit.

As an example, suppose Register 2 contains a hex 4002 (a binary 0100 0000 0000 0010). The instruction

```
SLA  R2,1
```

shifts the contents of Register 2 one bit position to the left, leaving in Register 2 a binary 1000 0000 0000 0100 (or hex 8004).

```
                    (R2) Before
      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
     +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
     | 0| 1| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 1| 0|
     +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
          |       .  .  .  .  .  .  .  .  .  .  .  . |
          |--|                                  +--+
          |                                     |
          V               (R2) After            V
      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
     +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
     | 1| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 1| 0| 0| <--- 0
     +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```
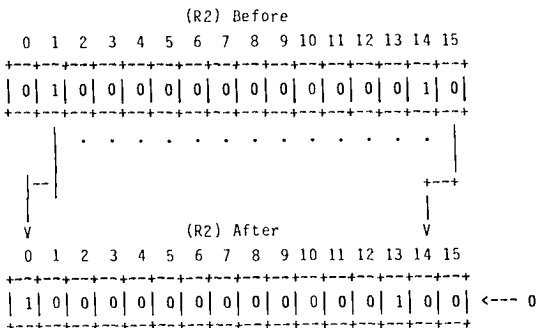
If you think of hex 4002 as an unsigned number, then hex 8004 is, indeed, two times hex 4002. But, if you think of hex 4002 as a signed number, then hex 8004 is not two times hex 4002. Why? Because the sign bit is different; it changed. If you think of hex 4002 as a signed number, it's a positive number (the sign bit is a zero), but the sign of hex 8004 is negative (the sign bit is a one).

11.6.5   Using Register 0 for a Shift Count

The purpose of the second operand with a shift instruction is to tell the computer how many bit positions to shift. That number must be in the range of 0 through 15. A non-zero number (1 through 15) tells the computer directly how many bits to shift, but an operand of 0 tells the computer that it's to look in Register 0 for the shift count. Specifically, the computer looks in the rightmost nibble of Register 0 for the shift count.

Consider these three instructions.

```
LI   R0,>D573
LI   R11,>1234
SLA  R11,0
```

The SLA instruction shifts the bits in Register 11 to the left three positions. The second operand is 0, which directs the processor to find the shift count in Register 0. Since the rightmost nibble of Register 0 is 3, the contents of Register 11 are shifted 3 positions to the left, so that a hex 91A0 remains in the register.

```
                     (R11) Before
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  | 0| 0| 0| 1| 0| 0| 1| 0| 0| 0| 1| 1| 0| 1| 0| 0|
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
               |    .  .  .  .  .  .  .  .  .  |
               |                              |
    +--------- |                   +----------+
    |          |                   |
    V          |       (R11) After V
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  | 1| 0| 0| 1| 0| 0| 0| 1| 1| 0| 1| 0| 0| 0| 0| 0| <--- 0
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```
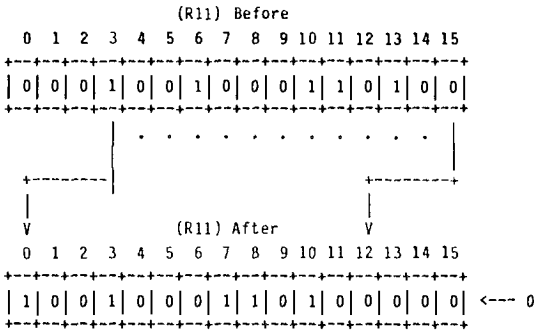
Consider these instructions.

```
LI   R0,0
LI   R11,>1234
SLA  R11,0
```

The SLA instruction shifts the bits in Register 11 to the left 16 positions, because the second operand is 0 and the rightmost nibble of Register 0 is 0. The SLA instruction shifts the bits 16 positions and fills the vacated bit positions on the right with zeros. Upon completion of the SLA instruction, Register 11 contains zero.

11.6.6   Testing the Carry Status Bit

With the shift instructions, the Carry status bit is set to the state of the last bit shifted out of the register. There are two instructions that let you check the state of the Carry status bit: Jump On Carry (JOC) and Jump if No Carry (JNC). The JOC causes a jump if the Carry status bit is one; the JNC causes a jump if the Carry status bit is zero.

11.7 Program Example

The following is a printed listing from the assembly of a source program that uses several of the data movement instructions.

```
99/4 ASSEMBLER                                              PAGE 0001
VI    1 1.2
                              IDT  'BITCNTR'
      0000 02E0              LWPI WS          INITIALIZE WORKSPACE POINTER
           004E'
0003  0004 0200              LI   R0,)1000    POINT R0 TO BITS
           1000
0004  0008 0201              LI   R1,COUNTS   POINT R1 TO COUNTS STORAGE
           003E'
0005  000C 0202              LI   R2,16       SET LOOP COUNTER (BYTES TO EXAMINE)
           0010
0006  0010 0203    NXTBYT LI  R3,0           INIT BIT COUNT TO ZERO
           0000
0007  0014 0204              LI   R4,8        SET LOOP COUNTER (BITS TO EXAMINE)
           0008
0008  0018 D170              MOVB *R0+,R5     COPY A BYTE INTO R5
0009  001A 0A15    SHIFT  SLA  R5,1          SHIFT OUT A BIT
0010  001C 1702              JNC  BITIS0      IF BIT IS ZERO, JUMP
0011  001E 0223              AI   R3,1         ELSE ADD TO BIT COUNT
           0001
0012  0022 0224    BITIS0 AI  R4,-1          DECREMENT BIT LOOP COUNT
           FFFF
0013  0026 16F9              JNE  SHIFT       IF NOT ZERO, GO EXAMINE NEXT BIT
0014                SWPB R3                   ELSE PUT BIT COUNT IN R3 LEFT BYT
0015        DC43              MOVB R3,*R1+     .. STORE IT
0016        0222              AI   R2,-1       DECREMENT BYTE LOOP COUNTER
           FFFF
0017        16EF              JNE  NXTBYT      IF NOT ZERO, GO GET NEXT BYTE
0018        0420              BLWP @0          ELSE GO HOME
           0000
0019        COUNTS           16               BIT COUNTS STORED HERE
0020  0046  WS               32               WORKSPACE
0021
```

```
99/4 ASSEMBLER
VERSION 1.2                                                PAGE 0002
'  BITIS0  0022   '  COUNTS  003E   '  NXTBYT  0010      R0    0000
   R1      0001      R10     000A      R11     000B      R12   000C
   R13     000D      R14     000E      R15     000F      R2    0002
   R3      0003      R4      0004      R5      0005      R6    0006
   R7      0007      R8      0008      R9      0009   '  SHIFT 001A
'  WS      0046
0000 ERRORS
```

The program counts the number of one bits in each of 16 successive bytes of memory and stores each count in 16 other successive bytes of memory.

```
            Bits                           Counts
         +-------+                       +-------+
1st Byte |   :   |2nd Byte   1st Count|   :   |2nd count
         +-------+                       -------+
3rd Byte |   :   |4th Byte   3rd Count|   :   |4th count
         +-------+                     +-------+
             :                             :
             :                             :
         +-------+                     +-------+
15th Byte|   :   |16th Byte 15th Count|   :   |16th Count
         +-------+                     +-------+
```
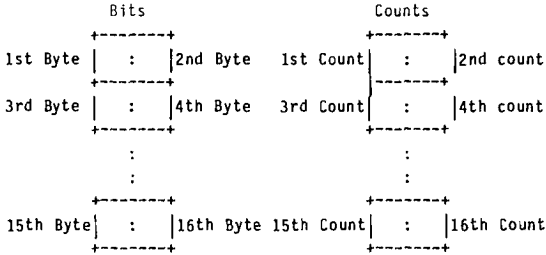
In general, this is how the program works. The program copies a byte of memory into
a register and then shifts each bit of the byte out of the register one bit at a time. As each
bit is shifted out, the program analyzes the Carry status bit. If the Carry status bit is one,
this means the bit shifted out is a one bit and the program adds one to a count of the
number of bits. If the Carry status bit is zero after the shift, the shifted bit is a zero, so
the program does not add to the count of the number of one bits.

Each of the eight bits in a byte are analyzed and the accumulated count is stored into
a byte of memory. The program then accesses the second byte of memory, analyzes each
bit, counts the number of one bits, stores the count, and accesses the next byte of memory.
The program continues this repetitive process until the bits in all sixteen bytes have been
counted and the counts stored in memory.

Review the listing. All of the instructions were introduced in previous chapters.

The program contains four assembler directives: an IDT in the first statement, an END
in the last statement, and two BSS directives just before the END directive.

The IDT and END directives are like bookends for a source program. The IDT directive
identifies the name of the program and is optional. The END directive is required. It tells
the assembler to stop assembling.

The IDT directive is optional, but if it is used, it must come before any instruction or any
other directive that defines data within the program. The IDT directive simply names
the program.

The operand field of an IDT directive is the name assigned to the program. The name
must be enclosed in single quote marks (apostrophes) and is limited to a maximum of
eight characters.

The last statement of every source program should have an END directive. If it doesn't, the assembler either doesn't stop assembling when it should, or it gives you an error message. Notice that END is a directive to the assembler and is not an instruction to the computer. END simply tells the assembler when to stop translating, but it results in no machine code instructions.

The END directive can have an operand. The operand, if used, is the name of a statement in the program. The operand identifies the entry point of the program (the name of the instruction to perform first). If you use an operand with the END directive, the program begins running at that entry point as soon as the object program is loaded.

There are two BSS directives in the program. The BSS (Block Starting with Symbol) directive tells the assembler to set aside a block of memory. The label is the name assigned to the beginning of the block. The operand tells the assembler how much memory, in bytes, to set aside.

In this program, the first BSS directive (statement 19) reserves 16 bytes of memory and the first location in the block is named COUNTS. The second BSS directive (statement 20) reserves 32 bytes (16 words) of memory and the first location in the block is named WS (as in WorkSpace). This 16-word block is the area of memory that the program uses for its workspace.

Examine the instructions. The first instruction (statement 2) is Load Workspace Pointer Immediate (LWPI). It explicitly puts into the Workspace Pointer the address of the block of memory to be used for the program's working registers. Notice WS is the operand and WS is the label attached to a 32-byte block of memory at the end of the program. WS has an address value and that address value is the immediate operand which is loaded into the Workspace Pointer. The relative address value of WS is hexadecimal 46.

The next three instructions (statements 3, 4, and 5) are Load Immediate instructions which perform initialization prior to examining the bits in all the bytes. The first of these three instructions puts into Register 0 the beginning address of the bytes to be examined. This address is hex 1000. (This is an arbitrary choice and points to a location in the computer's fixed-contents memory, or ROM). The second Load Immediate instruction puts into Register 1 the beginning address of the memory locations where the bit counts are stored. The immediate operand is COUNTS. This the name of the area of memory reserved by the first BSS directive. The third Load Immediate instruction puts a 16 into Register 2. The number 16 is a loop count equal to the number of bytes to examine.

There are two more Load Immediate instructions (statements 6 and 7) after the first three. The first one initializes Register 3 to zero. As the one bits in each byte are counted, the

count is accumulated in Register 3. Register 3 must start off with a zero in it or the count will be wrong. The second of these two Load Immediate instructions puts an 8 into Register 4. The 8 is a loop count equal to the number of bits to examine in each byte.

Notice the structure of the program. There is a loop within a loop. The outer loop is performed 16 times or once for each byte and the inner loop is performed 8 times or once for each bit in a byte. The outer loop begins at the instruction labeled NXTBYT; the inner loop begins at the instruction labeled SHIFT.

After the LI instructions, there is a Move Byte instruction (statement 8). It accesses the byte pointed to by the address in Register 0 and copies the byte into Register 5. Since this is a byte operation using register direct addressing for the destination operand, the byte is moved into the left byte of Register 5. Notice the MOVB instruction uses register indirect autoincrement addressing mode for the source operand. Once the byte is accessed, the address in Register 0 is automatically incremented by one since MOVB performs a byte operation. After the MOVB instruction is performed, Register 0 points to the next sequential byte in memory.

The SLA instruction, labeled SHIFT, begins the inner loop. The instruction shifts the contents of Register 5 one bit position to the left. The left half of Register 5 has a copy of the byte taken from memory. Remember what happens to the last bit shifted out of a register? Its state (1 or 0) is recorded in the Carry status bit. After the SLA instruction is performed, the Carry status bit tells you whether the bit is a one or zero.

A Jump if No Carry, JNC, instruction is next (statement 10). If the Carry status bit is zero, the Jump if No Carry instruction jumps. If the Carry status bit is one, it doesn't jump. If the JNC instruction does jump, it goes to the instruction labeled BITIS0 (as in "Bit Is 0"). It jumps when the status bit is zero and skips the next instruction (which means the bit is not counted).

Statement 11, the AI instruction, adds one to the contents of Register 3. The AI instruction adds one to the accumulated count of one bits in Register 3 if the last bit shifted out of the register is a one.

The following AI instruction, labeled BITIS0, subtracts one from the inner loop counter (the one used to count the number of bits to examine in each byte). The AI instruction was used in the program example in the previous chapters.

Recall that the sum of an AI instruction is compared to zero and the Equal status bit is affected by that comparison.

Following the second AI instruction is the Jump if Not Equal instruction (statement 13). It analyzes the Equal status bit and jumps to the instruction labeled SHIFT if the Equal status bit is not set. The JNE instruction closes the inner loop and causes a jump back to the beginning of the inner loop until the count in Register 4 goes to zero. When the inner loop is performed 8 times, the program falls out of the loop to the SWPB instruction.

The Swap Bytes instruction, statement 14, is performed when all eight bits in a byte have been examined. At this step in the program, there is a count of the number of one bits in Register 3. The count is a 16-bit number but since that count is never larger than 8, the count is contained in the least significant (rightmost) byte of Register 3. The count needs to be stored in a byte of memory. In order to move a single byte of data in a register, the data needs to be in the left byte of the register. The SWPB instruction swaps the two bytes in Register 3 so that the right byte is placed in the left-byte position. The count is now ready to be moved.

The Move Byte instruction, statement 15, moves the count from the left byte of Register 3 into the byte of memory pointed to by the address in Register 1. Notice that the destination operand is using register indirect autoincrement addressing mode. As soon as a count is placed in memory, the address in Register 1 is automatically adjusted to point to the next byte.

The Add Immediate instruction, statement 16, decrements the outer loop counter (in Register 2).

Another JNE instruction, statement 17, follows and closes the outer loop by causing a jump to NXTBYT if the loop count is not yet zero.

The last instruction is the "Go-home" instruction (BLWP). The instruction was used in the program example in the previous chapters.

That's the program. If you have the equipment and the utility programs, you can edit, assemble, load, and run it.

Before running the program, set a breakpoint at the BLWP instruction. Use the Debugger to examine the 16 bytes of memory beginning at hex 1000 and to examine the counts stored in the 16 bytes of memory beginning at COUNTS.

This program illustrates how to use some of the data movement instructions. The next chapter introduces the Compare instructions.

Chapter 12

COMPARE INSTRUCTIONS

This chapter introduces the group of Compare instructions. The main job of these instructions is to compare values and establish the relationships of the values, or to analyze specific bits in data. There are 5 instructions in this group. The instructions are listed below with their names, operation codes, and a description of the kinds of addressing modes you can use with the instructions.

In the following list, G indicates a general addressing mode operand (one that can use any of the five general addressing modes). R indicates a working register, which means the operand can use only register direct addressing mode. IOP indicates that an operand must use immediate addressing and it is a data value, rather than the address of a data value.

Name	Operation Code	Addressing Mode
Compare Words	C	G,G
Compare Bytes	CB	G,G
Compare Immediate	CI	R,IOP
Compare Ones Corresponding	COC	G,R
Compare Zeros Corresponding	CZC	G,R

12.1 The Compare Values Instructions (C, CB, and CI)

The first three instructions, Compare Words, Compare Bytes, and Compare Immediate, compare two values and establish the relationships between the values by affecting status bits. With all three instructions, the data values are not changed. The two values are simply compared and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

12.1.1   The Compare Words Instruction (C)

The Compare Words instruction (C) compares two words together. The instruction requires two operands. Both operands can use any of the five general addressing modes. The word addressed by the first operand is compared to the word addressed by the second operand and the comparison affects the status bits.

As an example, assume that memory location BEAGLE contains a −100 (hex FF9C) and Register 14 contains a 13 (hex 000D). The instruction

```
C    @BEAGLE,R14
```

compares −100 to 13 and establishes their relationships by affecting the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

The Logical Greater Than status bit is set to a one since the absolute value of −100 is larger than the absolute value of 13. The Arithmetic Greater Than status bit is cleared to a zero because −100 is not arithmetically greater than 13, in fact, its signed value is smaller than 13. And the Equal status bit is cleared to zero since the two values are not equal.

After the instruction is performed, memory location BEAGLE still contains a hex FF9C and Register 14 still contains a hex 000D.

12.1.2   The Compare Bytes Instruction (CB)

The Compare Bytes instruction (CB) works just like the Compare Words instruction except it compares two bytes, rather than words. Another difference is that the CB instruction affects the Odd Parity status bit and the C instruction does not. The Odd Parity status bit is affected based upon the number of bits in the first operand.

Assume that the memory word with address hex D36C contains a hex 6A8F, Register 10 contains a hex D360, and Register 4 contains a hex F20B. The following instruction compares the byte value hex 8F to the byte value hex F2. The contents of byte address hex D36D is hex 8F and the left byte of Register 4 is hex F2.

```
CB   @13(R10),R4
```

The comparison causes the Logical Greater Than status bit to be cleared to zero. The Arithmetic Greater Than status bit to be cleared to zero (since hex 8F or −113 is arithmetically smaller than hex F2 or −14). And the Equal status bit is cleared to zero.

The Odd Parity status bit is set to one since a binary 1000 1111 (hex 8F) contains an odd number of one bits.

12.1.3 The Compare Immediate Instruction (CI)

The Compare Immediate instruction (CI), like the Compare Words instruction, compares two words. And like the Compare Words instruction, the Compare Immediate instruction requires two operands. However, with the Compare Immediate instruction, the first operand is limited to register direct addressing and the second operand is limited to immediate addressing.

The Compare Immediate instruction compares the contents of a register to an immediate operand. It's often used to compare a variable address value in a register being used as an index register or indirect register with a specific anticipated address value. For example, suppose a list of data is being accessed in a loop and the data is accessed using Register 6 as an indirect register for register indirect autoincrement addressing mode. Rather than using a loop count to determine when all the data has been accessed, the program can, instead, use a CI instruction to await the appearance of the address at the end of the table. Let's say the last data word is at address hex FC20. The instruction

```
CI  R6,>FC20
```

can be used to determine when the last item is accessed.

The contents of Register 6 is autoincremented to hex FC22 when the last word in the table is accessed. The contents of Register 6 becomes logically greater than hex FC20 after the last word is accessed.

You can visualize the loop like this.

```
---------------
      .
      .
MOV   *R6+,R0        ACCESS A LIST ITEM
---------------
      .
      .
CI    R6,>FC20       END OF LIST?
```

12.2 Using the Jump if Low or Equal Instruction (JLE)

After the Compare Immediate instruction, you can use a Jump if Low or Equal (JLE) instruction to close the loop.

```
        --------------
                .
                .
LOOP    MOV   *R6+,R0        ACCESS A LIST ITEM
        --------------
                .
                .
        CI    R6,>FC20       END OF LIST?
        JLE   LOOP           JUMP IF NOT END?
```

The JLE instruction causes a jump if the Logical Greater Than status bit is zero or the Equal status bits is a one. In this example, it causes a jump to LOOP as long as the content of Register 6 is logically less than or equal to hex FC20. When the content of Register 6 becomes logically greater than >FC20, which occurs after the last data word in the list is accessed, it lets the program fall out of the loop to the instruction following the JLE.

12.3 The Compare Bits Instructions (COC and CZC)

The Compare Ones Corresponding (COC) and Compare Zeros Corresponding (CZC) instructions analyze individual bits in a word.

12.3.1 The Compare Ones Corresponding Instruction (COC)

The Compare Ones Corresponding instruction (COC) analyzes specific bits in a word to determine if those selected bits are all ones. If they are, it sets the Equal status bit to one. If they're not, it clears the Equal status bit to zero. The only thing affected by the COC instruction is the Equal status bit. No other status bits are affected, and the contents of neither operand are changed.

The instruction requires two operands. The first operand can use any of the five general addressing modes, but the second operand can use only register direct addressing.

The first operand is the address of a "bit mask." The bit mask is a word used to select bit positions in another word. The position of the one bits in the bit mask select bits in the same positions in another word.

Here's an example. Suppose memory location FONZOE contains a hex A6F0, and Register 7 contains a hex 953D.

The instruction

```
COC  @FONZOE,R7
```

compares (analyzes) the bits in Register 7 selected by the bit mask to see if they are all ones.

The bit mask is a hex A6F0, or a binary 1010 0110 1111 0000. It looks like this.

```
              < -- Bit Mask  (in memory location FONZOE) -- >

Positions -  |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Bits -   | 1| 0| 1| 0| 0| 1| 1| 0| 1| 1| 1| 1| 0| 0| 0| 0|
             ----------------------------------------------------
             <--  A  --> <--  6  --> <--  F  --> <--  0  -->
```

Bit positions 0, 2, 5, 6, 8, 9, 10, and 11 are one bits.

Register 7 contains a hex 953D, or binary 1001 0101 0011 1101. It looks like this.

```
              < --------- Contents of Register 7  -------- >

Positions -  |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Bits -   | 1| 0| 0| 1| 0| 1| 0| 1| 0| 0| 1| 1| 1| 1| 0| 1|
             ----------------------------------------------------
             <--  9  --> <--  5  --> <--  3  --> <--  D  -->
```

You can imagine the computer performs the instruction this way. The bit mask is a checklist. Everywhere there is a one bit in the bit mask, the computer checks the corresponding bit position in the second operand (Register 7) to see if the bit is a one or not. If the bit is a one, the computer makes a check on the checklist. If all the selected bits are one bits, the computer sets the Equal status bit to one, indicating that all the bits are ones. If any or all of the selected bits are not a one bit, the computer clears the Equal flag to zero, indicating they are not all equal to ones.

In this example, the computer checks bits positions 0, 2, 5, 6, 8, 9, 10, and 11 in Register 7. That's where the bit mask says to look. The computer finds that the bits in positions

0, 5, 10, and 11 are one bits. But the bits in positions 2, 6, 8, and 9 are not one bits; therefore, the computer clears the Equal status bit.

Suppose that Register 7 contains hex B7F9 (a binary 1011 0111 1111 1001). It looks like this.

```
         < --------- Contents of Register 7 -------- >

Positions ~ |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
     Bits ~ | 1| 0| 1| 1| 0| 1| 1| 1| 1| 1| 1| 1| 1| 0| 0| 1|
            ----------------------------------------------
            <-- B --> <-- 7 --> <-- F --> <-- 9 -->
```

In this case, the computer sets the Equal status bit to one because each of the selected bit positions in Register 7 contains a one.

12.3.2   The Compare Zeros Corresponding Instruction (CZC)

The Compare Zeros Corresponding instruction (CZC) analyzes specific bits in a word to determine if those selected bits are all zeros. If they are, it sets the Equal status bit to one. If they are not, it clears the Equal status bit to zero. The only thing affected by the CZC instruction is the Equal status bit. No other status bits are affected, and the contents of neither operand are changed.

The instruction requires two operands. The first operand can use any of the five general addressing modes, but the second operand can use only register direct addressing.

The first operand is the address of a "bit mask." The bit mask is a word used to select bit positions in another word. The position of the one bits in the bit mask select bits in the same positions in another word.

Take an example. Suppose memory location FONZOE contains a hex A6F0 and Register 7 contains a hex 953D.

The instruction

```
CZC  @FONZOE,R7
```

compares (analyzes) the bits in Register 7 selected by the bit mask to see if they're all zeros.

The bit mask is a hex A6F0 (a binary 1010 0110 1111 0000). It looks like this.

```
              < -- Bit Mask  (in memory location FONZOE) -- >

Positions ~  |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
     Bits ~  | 1| 0| 1| 0| 0| 1| 1| 0| 1| 1| 1| 1| 0| 0| 0| 0|
             -------------------------------------------------
             <--  A  --> <--  6  --> <--  F  --> <--  0  -->
```

Bit positions 0, 2, 5, 6, 8, 9, 10, and 11 are one bits.

Register 7 contains a hex 953D or a binary 1001 0101 0011 1101. It looks like this.

```
              < --------- Contents of Register 7 -------- >

Positions ~  |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
     Bits ~  | 1| 0| 0| 1| 0| 1| 0| 1| 0| 0| 1| 1| 1| 1| 0| 1|
             -------------------------------------------------
             <--  9  --> <--  5  --> <--  3  --> <--  D  -->
```

You can imagine the computer performs the instruction this way. The bit mask is a checklist. Everywhere a one bit occurs in the bit mask, the computer checks the corresponding bit position in the second operand (Register 7) to see if the bit is a zero. If the bit is a zero, the computer makes a check on the checklist. If all the selected bits are zero bits, the computer sets the Equal status bit to one, indicating that all the bits there are zeros. If any or all of the selected bits are not a zero bit, the computer clears the Equal flag to zero, indicating they are not all zeros.

In this example, the computer checks bits positions 0, 2, 5, 6, 8, 9, 10, and 11 in Register 7 where the bit mask says to look. The computer finds that the bits in positions 2, 6, 8, and 9 are zero bits. But the bits in positions 0, 5, 10, and 11 are not; therefore, the computer clears the Equal status bit.

145

Suppose that Register 7 contains hex 410D or a binary 0100 0001 0000 1101. It looks like this.
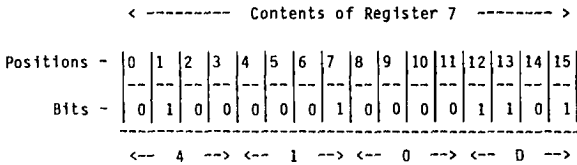
```
< --------- Contents of Register 7  -------- >

Positions -  |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Bits -   | 0| 1| 0| 0| 0| 0| 0| 1| 0| 0| 0| 0| 1| 1| 0| 1|

             <-- 4  --> <-- 1  --> <-- 0  --> <-- D  -->
```

In this case, the computer sets the Equal status bit to one because each of the selected bit positions in Register 7 contains a one.

12.4 Program Example

Here's the listing of a program that uses two of the Compare instructions.

```
9974 ASSEMBLER
VERSION 1.7                                                 [    PAGE 0001
 0001                          IDT  'SORT'        PROGRAM TO SORT IN     JDING ORDER
 0002 0000 02E0               LWPI WS             INITIALIZE WORKSPACE COUNTER
      0002 003C'
 0003 0004 0200        L1     RO,FILE             POINT RO TO BEGINNING OF FILE
      0006 002C'
 0004 0008 C040  PASS   MOV   RO,R1              POINT R1 TO WHERE RO POINTS
 0005 000A CC5,0 COMPAR C     *RO,*R1*            COMPARE TWO WORDS
 0006 000C 1205         JLE   INORDR              JUMP IF IN ORDER
 0007 000E C0A1         MOV   @-2(R1),R2          SAVE SMALLER NUMBER IN R2
      0010 FFFE
 0008 0012 C850         MOV   *RO,@-2(R1)         PUT BIGGER NUMBER WHERE SMALLER WAS
      0014 FFFE
 0009 0016 C402         MOV   R2,*RO              PUT SMALLER NUMBER WHERE IT BELONGS
 0010 0018 0281 INORDR  CI    R1,FILEND           FINISHED THIS PASS?
      001A 003A'
 0011 001C 12F6         JLE   COMPAR              IF NOT, JUMP
 0012 001E 0220         AI    RO,2                IF SO, MOVE RO DOWN A WORD
      0020 0002
 0013 0022 0280         CI    RO,FILEND           FINISHED ALL PASSES?
      0024 003A'
 0014 0026 16F0         JNE   PASS                IF NOT, JUMP
 0015 0028 0420         BLWP  @0                  IF SO, GO HOME
      002A 0000
 0016 002C 0005  FILE   DATA  3,6,2,1,7,2,10,5
      002E 0006
      0030 0002
      0032 0001
      0034 0007
      0036 0002
      0038 000A
      003A 0005
 0017        003A' FILEND EQU  $-2                FILEND=LOC. OF LAST ITEM
 0018 003C   WS    BSS   32                  WORKSPACE
 0019              END
```

146

```
99/4 ASSEMBLER
VERSION 1.2                                                      PAGE 0002
   '  COMPAR  000A   '  FILE   002C   '  FILEND  003A   '  INDRDR  0018
   '  PASS    000B      R0     0000      R1      0001      R10     000A
      R11     000B      R12    000C      R13     000D      R14     000E
      R15     000F      R2     0002      R3      0003      R4      0004
      R5      0005      R6     000E      R7      0007      R8      0008
      R9      0009   '  WS     003C
   0000 ERRORS
```

This program sorts numbers into ascending order; that is, it puts the smallest number first, the next larger number next, and the largest number is put last.

The program demonstrates the use of two compare instructions and includes a new assembler directive.

In general, the program works like this. It starts with a file of unsorted data words in memory and sorts the words in ascending order, leaving the sorted words in the same memory locations.

The program uses a technique called a replacement sort. It starts at the beginning of the file and makes several passes through the file. On the first pass, the program compares pairs of words and exchanges their positions, if necessary, so that at the end of the first pass, the smallest number in the file is in the first location.

For the second pass, the program begins with the second data word (the smallest is already in the first location) and compares pairs of words, exchanging their positions if necessary, so that at the end of the second pass, the next highest number (or a number equal to the smallest number) is in the second position.

For the third pass, the program begins with the third data word, compares the remaining words in the file, and leaves the next highest number in the third position.

On each pass, the program begins further in the file and leaves the next highest number in its proper position. The program continues making passes on the data until, on the last pass, the program ends up with the largest number in the last position.

Take a look at the listing. Notice there is a C instruction, statement 5, and two CI instructions, statements 10 and 13, in the program. There are also a couple of JLE instructions, statements 6 and 11. The Jump if Low or Equal instruction was introduced in this chapter. The rest of the instructions you have seen before.

The program also has a few directives like the bookends, IDT and END. Just above the END directive is a BSS directive that is labeled WS. This directive reserves 32 bytes (16 words) for the program's working registers.

147

Look at the DATA directive that is labeled FILE. The DATA directive tells the assembler to set aside some memory words for the program to use and to put a specific value into the words. The operands identify how many words to set aside and the values to put in those words.

Since there are 8 operands with this DATA directive, the assembler sets aside 8 words of memory. The assembler places 3 in the first word, 6 in the second word, 2 in the third word, and so forth. The first word is labeled FILE. These 8 words are the data file that the program sorts, and the file begins with the first data word (labeled FILE).

Look at the EQU directive. The EQU directive equates a name to a value. The label tells the assembler what to call the value and the operand identifies the value.

In this program, the EQU directive assigns the name FILEND to the value of $ − 2. A dollar sign symbol is recognized by the assembler as the location of the statement in which it appears. The location of this EQU statement is at the end of the data file which is at the word immediately after the last data word. Therefore, the value of $ − 2 is the location of the last data word. The assembler is told to call the location of the last data word FILEND.

The EQU directive does not reserve any memory locations. It simply tells the assembler what to call a location.

Let's look at the instructions. The entry point of the program is the LWPI instruction in statement 2.

The next instruction is a Load Immediate (statement 3). This instruction puts the address value for FILE in Register 0. FILE is the name assigned to the beginning data word in the file. File has a relative address value of hexadecimal 2C.

The instruction labeled PASS copies the contents of Register 0, the address of where to start the current pass, into Register 1. At the beginning of a pass, Register 0 and Register 1 point to the same word.

The next instruction, labeled COMPAR, compares the word pointed to by Register 0 to the word pointed to by Register 1. Notice the second operand uses register indirect autoincrement addressing mode. After making a comparison, Register 1 points to the next word.

Following the Compare Words instruction is a JLE instruction in statement 6. The JLE instruction causes a jump to the instruction labeled INORDR if the word pointed to Register 0 is less than or equal to the word pointed to by Register 1 when the Compare

instruction was performed. The JLE jumps if the two numbers were already in order; that is, the smaller word is already ahead of the larger word or the two numbers are the same. It does not jump if the two numbers are not in order. If the number pointed to by Register 0 is larger than the one pointed to by Register 1, an exchange needs to be made.

Statements 7, 8, and 9) are MOV instructions that exchange the position of two numbers. The first MOV instruction copies the smaller number into R2. The second MOV instruction moves the larger number into the space that was occupied by the smaller number. The third MOV instruction copies the smaller number into the place where the larger number was.

The first two MOV instructions use indexed addressing. The operand $@-2(R1)$ uses Register 1 as an index register. When these instructions are performed, Register 1 points to the location following the second value compared. The contents of Register 1 was autoincremented when the second value was accessed (statement 5). So, to refer back to that value, a minus 2 needs to be added to the address in Register 1.

After comparing two values and making an exchange if necessary, the instruction labeled INORDR is performed. INORDR is a Compare Immediate instruction that compares the address value in Register 1 to the immediate operand FILEND. Remember that FILEND is the name of the location of the last data word.

After this CI instruction, the JLE instruction (statement 11) jumps if the address value in Register 1 is less than or equal to FILEND. As long as it is, the pass is not complete, there are more words to compare, and the JLE jumps to COMPAR. The address value in Register 1 becomes bigger than FILEND when the last word in the file is accessed. At that time, the contents of Register 1 is autoincremented to an address value larger than FILEND and the JLE instruction allows the program to fall out of the loop to the Add Immediate instruction at statement 12.

The AI instruction adds 2 to the address value in Register 0 in preparation for the next pass. But before performing another pass, the program determines if another pass is necessary.

The next instruction, statement 13, is a CI instruction which compares the address value in Register 0 with FILEND. If the address value is not yet equal to FILEND, there are more passes to complete and the JNE instruction (statement 14) causes a jump to PASS.

When the address value in Register 0 is equal to FILEND (when Register 0 has been bumped to point to the last data item), the JNE instruction allows the program to fall out of the outer loop and go on to the next instruction.

The last instruction (statement 15) is the Go-home instruction.

You can use the assembler to assemble the program and then use the Loader to load th
resulting object program. Load the Debugger along with the program and use th
Debugger to control the program.

Set a breakpoint at the BLWP instruction and run the program. Use the Debugger t
examine the 8 words of memory beginning at FILE to confirm that the numbers wer
sorted correctly.

Then use the Debugger's Memory Inspect/Change command to change the contents c
the file. Run the program again and check the results. It should sort as well the secon
time as it did the first.

This an example of how you can use the compare instructions. The next chapter discusse
the jump instructions.

Chapter 13

THE JUMP INSTRUCTIONS

This chapter introduces the Jump instructions. The main purpose of the Jump instructions is to make decisions in a program. These decisions are based upon an evaluation of the status bits that are affected by the performance of previous instructions. It's important to notice that the Jump instructions do not affect the status bits; they simply examine them. After a jump instruction is performed, the status bits are in the same state as they were before the instruction was performed.

Some of these instructions have been introduced already, such as the JNE, JNC, and JLE instructions.

Below is a list of all of the 13 Jump instructions, their names, their mnemonic operation codes, and the conditions that cause them to jump.

Name	Operation Code	Jump Conditions
Jump if Equal	JEQ	EQ = 1
Jump if Not Equal	JNE	EQ = 0
Jump On Carry	JOC	CY = 1
Jump if No Carry	JNC	CY = 0
Jump if No Overflow	JNO	OV = 0
Jump if Odd Parity	JOP	OP = 1
Jump if High	JH	L$>$ = 1
Jump if High or Equal	JHE	L$>$ = 1 or EQ = 1
Jump if Low or Equal	JLE	L$>$ = 0 or EQ = 0
Jump if Low	JL	L$>$ = 0 and EQ = 0
Jump if Greater Than	JGT	A$>$ = 1
Jump if Less Than	JLT	A$>$ = 0 and EQ = 0
Jump Unconditionally	JMP	Always

There are 12 conditional jump instructions which may or not cause a jump, based upon the condition of the status bits. The thirteenth jump instruction is unconditional and jumps under any conditions.

All jump instructions use PC-relative addressing and have a limited transfer-of-control range. A jump instruction can jump only as far as 254 bytes behind its location and only up to 256 bytes ahead of its location.

A jump instruction requires one operand. The operand designates the target of the jump. The target can be specified in three ways.

One way and, usually, the best way, is to use a name as a target. The operand is the name (label) attached to the target instruction. For example,

        JMP   CREEPY

where CREEPY is the name of the target instruction.

A second way is to use a numeric address for a target. For example,

        JGT   56984

where 56984 is the physical address for the target.

A third way is to use a dollar sign reference to specify how far to jump based upon the location of the instruction. For example,

        JOP $+4

where $ means the location of the jump instruction and +4 is the distance (displacement) in bytes of the target from this location.

No matter which of the three ways you choose to specify the target of a jump, it must be within range.

Let's look now at the jump instructions and some examples of how to use them.

13.1   The Equal Testing Instructions (JEQ and JNE)

Both the Jump if Equal (JEQ) and Jump if Not Equal (JNE) instructions only examine the Equal status bit. The JEQ jumps if it's one; the JNE jumps if it's zero.

Most instructions affect the Equal status bit. For example, when the arithmetic instruction AI is performed, the result is compared to zero. When the data movement instruction MOV is performed, the data value is compared to zero. When the compare instruction CB is performed, a byte value is compared to another byte value. All of these instructions affect the Equal status bit. You've also seen that the COC and CZC instructions affect only the Equal status bit.

13.1.1 The Jump if Equal Instruction (JEQ)

The Jump if Equal instruction (JEQ) causes a jump if the Equal status bit is one.

As an example, the following JEQ instruction causes a jump if two byte values are the same.

```
CB   @DAISY,*R9
JEQ  SAME
```

13.1.2 The Jump if Not Equal Instruction (JNE)

The Jump if Not Equal instruction (JNE) causes a jump if the Equal status bit is zero.

As an example, the following JNE instruction causes a jump if the result of the AI instruction is not zero.

```
AI   R8,-1
JNE  LOOP
```

13.2 The Carry Testing Instructions (JOC and JNC)

The Jump on Carry (JOC) and Jump if No Carry (JNC) instructions examine only the Carry status bit. The JOC instruction jumps if it's one. The JNC instruction jumps if it's zero.

Several instructions affect the Carry status bit. An arithmetic instruction like AI affects the Carry status bit as a result of the add operation. The shift instructions record the state of the last bit shifted out of a register in the Carry status bit.

13.2.1 The Jump On Carry Instruction (JOC)

The Jump On Carry instruction (JOC) jumps if the Carry status bit is one. As an example, the following JOC instruction jumps if the sign bit of the number in Register 5 is one.

```
SLA   R5,1
JOC   ONEBIT
```

13.2.2   The Jump if No Carry Instruction (JNC)

The Jump if No Carry instruction (JNC) jumps if the Carry status bit is zero. As an example, the following JNC instruction jumps if the number in Register 3 is an even number (the rightmost bit is zero).

```
SRC   R3,1
JNC   EVEN
```

13.3   The Jump if No Overflow Instruction (JNO)

The Jump if No Overflow instruction (JNO) jumps if the Overflow status bit is zero. It's the only jump instruction that evaluates the Overflow status bit.

The Overflow status bit is affected by many of the arithmetic instructions. It's also affected by the SLA instruction. As an example, suppose Register 12 contains a hex F96E before these two instructions are performed.

```
SLA   R12,4
JNO   OK
```

The JNO jumps because the sign bit does not change during the shift.

13.4   The Jump if Odd parity Instruction (JOP)

The Jump if Odd Parity instruction (JOP) jumps if the Odd Parity status bit is one. This jump instruction is the only one that evaluates the Odd Parity status bit.

The Odd Parity status bit is affected by byte operations. It's set to one if there's an odd number of one bits in the byte result. It's cleared to zero if there's an even number of one bits.

Suppose the memory word at address hex D3A2 contains a hex DAC6 and Register 6 contains a hex D3A3 before the following instructions are performed.

```
MOVB   *R6,@DEALER
JOP    ODD
```

The JOP does not jump because the byte value moved (hex C6) has an even number of one bits.

13.5 The Logical Evaluation Instructions (JH, JHE, JLE, and JL)

There are four jump instructions that let you make decisions based upon a logical evaluation of values. They evaluate either the Logical Greater Than status bit alone, or the Logical Greater Than and Equal status bits together.

Most instructions affect the Logical Greater Than status bit. For example, when the arithmetic instruction AI is performed, the result is compared to zero. When the data movement instruction MOV is performed, the data value is compared to zero. When the compare instruction CB is performed, one byte value is compared to another byte value. All of these instructions affect the Logical Greater Than status bit.

The status bit is affected based upon a "logical" evaluation of data; that is, based on the absolute, or unsigned, value.

13.5.1 The Jump if High Instruction (JH)

The Jump if High instruction (JH) evaluates only the Logical Greater Than status bit and jumps if it's one.

Suppose the fourth word in a file beginning at KNOTS contains a hex 2FB9, Register 9 contains 6, and Register 7 contains a hex 8C3C before the following instructions are performed.

```
C    @KNOTS(R9),R7
JH   BIGGER
```

The JH instruction does not jump because the absolute value of hex 2FB9 is smaller than 8C3C.

13.5.2 The Jump if High or Equal Instruction (JHE)

The Jump if High or Equal instruction evaluates both the Logical Greater Than and Equal status bits. It jumps if either the Logical Greater Than or Equal status bit is one.

In the following example, the JHE instruction jumps since the result left is Register 7 is equal to zero.

```
LI    R0,0
SRL   R7,0
JHE   BILKO
```

13.5.3   The Jump if Low or Equal Instruction (JLE)

The Jump if Low or Equal instruction (JLE) evaluates both the Logical Greater Than and Equal status bits. It jumps if the Logical Greater Than status bit is zero or if the Equal status bit is one.

As an example, suppose that memory word hex AF9C contains a hex 2FB9, Register 12 contains a hex AF9D, and Register 1 contains a hex B93C before the following instructions are performed.

```
CB     *R12,R1
JLE    MARGIN
```

The JLE instruction jumps because the byte value hex B9, the content of byte address hex AF9D, is equal to the left byte of Register 1.

13.5.4   The Jump if Low Instruction (JL)

The Jump if Low instruction (JL) evaluates both the Logical Greater Than and Equal status bits. It jumps only if both status bits are zero.

As an example, suppose that Register 10 contains a hex AE78 before the following instructions are performed.

```
CI     R10,>AE78
JL     TOOLOW
```

The JL instruction does not jump because the hex AE78 in Register 10 is not smaller than the immediate value; it's equal to it.

13.6   The Arithmetic Evaluation Instructions (JGT and JLT)

There are two jump instructions that allow you to make decisions based upon an arithmetic evaluation of values. They evaluate either the Arithmetic Greater Than status bit alone or the Arithmetic Greater Than and Equal status bits together.

Most instructions affect the Arithmetic Greater Than status bit. For example, when the arithmetic instruction AI is performed, the result is compared to zero. When the data movement instruction MOV is performed, the data value is compared to zero. When the compare instruction CB is performed, a byte value is compared to another. All these instructions affect the Arithmetic Greater Than status bit.

The status bit is affected by an arithmetic evaluation of data; that is, based upon the signed value of the data.

13.6.1 The Jump if Greater Than Instruction (JGT)

The Jump if Greater Than instruction (JGT) evaluates only the Arithmetic Greater Than status bit, and jumps if it's one.

Suppose the fourth word in a file beginning at KNOTS contains a hex 2FB9, Register 9 contains 6, and Register 7 contains a hex 8C3C before the following instructions are performed.

```
C     @KNOTS(R9),R7
JGT   GRATER
```

The JGT instruction jumps because the signed value of hex 2FB9 (a positive number) is greater than 8C3C (a negative number).

13.6.2 The Jump if Less Than Instruction (JLT)

The Jump if Less Than instruction (JLT) evaluates both the Arithmetic Greater Than and Equal status bits. The JLT instruction jumps if both status bits are zero.

As an example, suppose the memory word with address hex BD74 contains a hex 2C8E, Register 10 contains a hex BD75, and Register 3 contains a hex FE94 before the following instructions are performed.

```
CB    *R10,R3
JLT   LESSER
```

The JLT jumps because the contents of byte address hex BD75, or hex 8E, is arithmetically less than the left byte of Register 3, or hex FE. The signed value of hex 8E is -114; the signed value of hex FE is -2.

13.7 The Jump Unconditionally Instruction (JMP)

The Jump Unconditionally instruction (JMP) does not evaluate status bits. It jumps under any condition.

You can use the JMP instruction to transfer control to another instruction as long as that instruction is within range. You can use the JMP instruction so that, effectively, it is a conditional jump instruction. For example, in the following program segment, the JMP instruction is, effectively, a jump on even parity instruction. Suppose Register 3 contains a hex 2D8C before these instructions are performed.

```
MOVB  R3,R8
JOP   $+4
JMP   EVEN
```

The JOP does not jump because the parity of hex 2D is even. It allows the program to go on to the JMP instruction that does jump. Effectively, the JMP instruction is a jump on even parity. The JOP instruction's operand ($+4) causes the JOP instruction to simply skip over the JMP instruction if the Odd Parity status bit is one.

13.8 Program Example

Here's the listing of a program that uses several jump instructions.

```
9976 ASSEMBLER
VERSION 1.1                                                       PAGE 0001
  0001                          IDT   'PARITYCK'   COMPARE PARITY OF TWO BYTES
  0002  0000 0.F0               LWPI  WS           INITIALIZE WORKSPACE
        0002  0074'
  0003  0004 0200          LI    R0,FILE      POINT TO FIRST WORD IN FILE
        0006  0054'
  0004  0008 0201     LOOP LI    R1,-1        INIT SAME/DIFFERENCE FLAG TO SAME
        000A FFFF
  0005  000C C0D0               MOV   *R0,R2       COPY TWO BYTES INTO R2
  0006  000E D082               MOVB  R2,R2        LEFT BYTE ODD PARITY?
  0007  0010 1C02               JOP   ODD1         IF SO, JUMP
  0008  0012 0221               AI    R1,1         ELSE BUMP FLAG
        0014 0001
  0009  0016 06C2     ODD1 SWPB  R2           EXCHANGE BYTES
  0010  0018 D082               MOVB  R2,R2        OTHER BYTE ODD PARITY?
  0011  001A 1C02               JOP   ODD2         IF SO, JUMP
  0012  001C 0221               AI    R1,1         ELSE BUMP FLAG
        001E 0001
  0013  0020 C041     ODD2 MOV   R1,R1        SAME OR DIFFERENT PARITIES?
  0014  0022 1601               JNE   SAME         IF SAME PARITY, JUMP
  0015  0024 C402               MOV   R2,*R0       ELSE REVERSE BYTES IN MEMORY
  0016  0026 0220     SAME AI    R0,2         POINT R0 TO NEXT WORD
        0028 0002
  0017  002A 0280               CI    R0,FILEND    END OF FILE?
        002C 0062'
  0018  002E 12EC               JLE   LOOP         IF NOT, GO EXAMINE NEXT BYTES
  0019  0030 0420               BLWP  @0           ELSE GO HOME
        0032 0000
  0020  0034     WS   BSS   32           WORKSPACE
  0021  0054 FILE BSS   16           FILE OF WORDS
  0022  0062' FILEND EQU   *-2          NAME OF END OF FILE
  0023               END
```

```
   99/4 ASSEMBLER
VERSION 1.2                                                  PAGE 0002
      ' FILE    0054   ' FILEND  00E2   ' LOOP   000B   ' ODD1    0016
      ' ODD2    0020     R0      0000     R1     0001     R10     000A
        R11     000B     R12     000C     R13    000D     R14     000E
        R15     000F     R2      0002     R3     0003     R4      0004
        R5      0005     R6      0006     R7     0007     R8      0008
        R9      0009   ' SAME    0026   ' WS    0034
   0000 ERRORS
```

The program analyzes the two bytes in each word of a file. If the two bytes have the same parity, either both even or both odd, the program does nothing to the word. However, if the parity of the two bytes is different, the program exchanges the position of the two bytes in the word.

The program uses a "flag" that helps to determine if two bytes have the same or different parity. A flag is a special code defined by a program and indicates whether a condition is true or not. Like most flags, this one indicates two conditions.

1. A zero value means the parity of the two bytes is different.

2. A non-zero value means the parity of the two bytes is the same. A -1 means both bytes have odd parity and a $+1$ means both bytes have even parity.

Look at the listing. Notice there is a mixture of word and byte operations. There are instructions from the data movement group, the compare group, and the jump group.

There are several assembler directives in the program, all of which have been introduced. Notice the BSS directive, labeled FILE, defines a 16-byte, or 8-word, block of memory. This block contains the data analyzed by the program. Notice also that the EQU directive defines the address of the last word in the file as FILEND.

Now, look at the instructions. Statement 2, the entry point of the program, is the LWPI instruction that sets up the workspace.

Statement 3 is a Load Immediate instruction that points Register 0 to the first word in the file.

In statement 4, the LI instruction initializes the flag to -1. The program starts a loop assuming that both bytes in the word have odd parity.

In statement 5, the MOV instruction copies a word from the file into Register 2.

The MOVB instruction, statement 6, moves a byte in Register 2 back into Register 2. The instruction actually moves the left byte of Register 2 back into the left byte position. After

159

the instruction is performed, the contents of Register 2 is exactly the same as before the instruction was performed. Seems useless, doesn't it? Something has changed, however. As a result of moving the byte, the computer got a chance to analyze the byte and affect the status bits. One of the status bits affected is the Odd Parity status bit.

After the MOVB instruction, the JOP instruction, statement 7, jumps to ODD1 if the parity of the left byte of the word is odd. It skips the AI instruction, statement 8, if the parity is odd. If the parity is even, the AI instruction is performed which adds one to the contents of Register 1 and the contents of Register 1 becomes zero.

The SWPB instruction, labeled ODD1, exchanges the two bytes in Register 2. The former right byte is now in the left byte position and vice versa.

The MOVB instruction at statement 10 has the computer affect the status bits, including the Odd Parity status bit.

If the other byte has odd parity, the JOP instruction at statement 11 skips the AI instruction at statement 12 and jumps to ODD2. If the other byte has even parity, the AI instruction at statement 12 adds one to the contents of Register 1.

When the instruction labeled ODD2 is reached. Register 1 contains either zero or a non-zero value. If the parity of the two bytes is different, it contains zero. If the parity of the two bytes is the same, it contains a non-zero value; either a -1 if the parity of both bytes is odd, or a $+1$ if the parity of both bytes is even.

At this point, also, Register 2 contains the two bytes in reverse order from how they were in the file.

The MOV instruction at ODD2 copies the flag from Register 1 back into Register 1 so that the status bits are affected.

After the MOV instruction, the JNE instruction at statement 14 jumps to SAME if the flag is non-zero; otherwise, the MOV instruction at statement 15 moves the swapped bytes in Register 2 back into the memory word they came from. The AI instruction labeled SAME moves the pointer in Register 0 to the next word in the file.

The Compare Immediate instruction at statement 17 compares the address value in Register 0 with the address of the last word in the file. If the last word in the file has not been analyzed, the JLE instruction at statement 18 jumps to LOOP to close the loop. When the last word has been analyzed. the JLE instruction allows the program to fall down to the Go-Home instruction.

When you are ready to try this program, you can use the Assembler to assemble the program and the Loader to load the resulting object program. Load the Debugger along with the program and use the Debugger to control the program.

Before running the program, use the Debugger to inspect and change the contents of the file to data values of your choosing. Mix it up a bit. Choose words that have bytes of different parity and the same parity.

Set a breakpoint at the BLWP instruction and run the program. Use the Debugger to examine the file and confirm the program worked correctly.

This chapter illustrates the use of several jump instructions. The next chapter introduces the Arithmetic instructions.

Chapter 14

THE ARITHMETIC INSTRUCTIONS

This chapter introduces the Arithmetic group of instructions. These are the instructions that perform arithmetic operations on data. There are 13 instructions in this group. The instructions are listed below with their names, operation codes, and a description of the kinds of addressing modes you can use with the instructions.

In the following list, G means that an operand is a general addressing mode operand and can use any of the five general addressing modes. An R means that an operand must be a working register which means it can use only register direct addressing mode. IOP means that an operand must use immediate addressing; the operand is a data value, rather than the address of a data value.

Name	Operation Code	Addressing Mode
Add Immediate	AI	R,IOP
Add Words	A	G,G
Add Bytes	AB	G,G
Subtract Words	S	G,G
Subtract Bytes	SB	G,G
Increment	INC	G
Increment by Two	INCT	G
Decrement	DEC	G
Decrement by Two	DECT	G
Negate	NEG	G
Absolute Value	ABS	G
Multiply	MPY	G,R
Divide	DIV	G,R

14.1 The Add Instructions (AI, A, and AB)

The add instructions add two numbers together and produce a sum. The additi
operation affects the Carry and Overflow status bits. The sum of the addition is compar
to zero and this comparison affects the Logical Greater Than, Arithmetic Greater Tha
and Equal status bits.

14.1.1 The Add Immediate Instruction (AI)

The Add Immediate instruction (AI) is probably familiar. The AI instruction w
introduced in Chapter 8. Perhaps, you recall how it works. It requires two operands. Th
first operand is a register; the second operand is an immediate value. The immediat
value is added to the contents of the register and the sum replaces the contents of th
register. Both addends are 16-bit numbers.

Also the sum is automatically compared to zero. Based upon this comparison, the Logic
Greater Than, Arithmetic Greater Than, and Equal status bits are affected.

The Carry and Overflow status bits are affected by addition. The Carry status bit
affected based upon a logical (or unsigned) evaluation of the result and the Overflo
status bit is affected based upon a signed evaluation of the results. These status bits te
you whether the answer is right or wrong.

Let's review what you have already learned about numbers. If you're given a numbe
like hex 89AB, and asked how much that number is in decimal, you really can't say unt
you have some more information. You need to know whether the number is signed
unsigned. If the number is unsigned, or a "logical" number, its absolute value is decim
35243. But if the number is signed, or an "arithmetic" number, it represents −30293.
hex 89AB represents a signed number, the number is negative since the sign bit is on
And the absolute value of the number is hex 7655 (hex 7655 is the two's complement
hex 89AB.

When an instruction is given to the computer that includes performing an addition, th
computer doesn't whether the numbers are signed or unsigned. The computer simpl
adds the numbers and provides enough information in the status bits for you to interpr
the results.

You can interpret the Logical Greater Than, Arithmetic Greater Than, and Equal stat
bits to determine the relationship of the result to zero. You can also interpret the Carr
and Overflow status bits to determine if the answer is right or wrong.

Following an add operation, the Carry status bit tells you whether the answer is right or wrong based upon a logical evaluation of the answer. The Overflow status bit tells you whether the answer is right or wrong based upon an arithmetic evaluation of the the answer.

Take an example. Suppose Register 7 has the number hex 6AC5 in it. The instruction

```
AI   R7,>3438
```

adds hex 6AC5 and hex 3438. The sum is hex 9EFD no matter how you interpret the numbers. But whether that sum is right or wrong does depend upon how you interpret **the numbers.**

If you interpret the two numbers as unsigned numbers, the answer is right. But if you interpret the numbers as signed numbers, the answer is wrong. Hex 6AC5 is a positive number and hex 3438 is also positive, but the sum, hex 9EFD is negative. Adding two positive numbers should not produce a negative sum.

The carry status bit is affected by the computer based upon a signed evaluation of the sum. If the Carry status bit is zero, the unsigned sum is correct, but if the Carry status bit is one, the unsigned sum is wrong.

The Overflow status bit is affected by the computer based upon a signed evaluation of the sum. If the Overflow status bit is zero, the signed sum is correct, but if the Overflow status bit is one, the signed sum is wrong.

Take another example. Suppose Register 7 still has a hex 6AC5 in it. The instruction

```
AI   R7,>B827
```

produces a sum of hex 22EC.

The Carry status bit is set to one. The real sum of hex 6AC5 and hex B827 is hex 122EC. It requires 17 bits to express the real sum but the computer only has 16 bits. For this reason, the Carry status bit is set to one which tells you the unsigned sum is wrong.

The Overflow status bit is zero. The signed result is correct. If you interpret hex 6AC5 as a signed number, it's positive. If you interpret hex B827 as a signed number, it's negative. The instruction is adds a positive number to a negative number, and the sum, hex 22EC, is a smaller positive number.

Here are the rules for determining if the overflow state occurs and the Overflow status bit is set to one. Think of the numbers as signed numbers. If the two numbers have opposite signs (one positive, the other negative), the overflow state can't occur, so the Overflow bit is not set. However, if the two numbers have the same sign (both positive or both negative), the overflow state is possible and actually occurs if the sign of the result is opposite that of the two addends.

Take a third example. Suppose Register 7 still has a hex 6AC5 in it. The instruction

```
AI  R7,>14D6
```

produces a sum of hex 7F9B in Register 7. No matter how you interpret the numbers, the sum is correct. The unsigned sum is correct because it can be expressed in 16 bits. The signed sum is correct because the two numbers are positive and the result is also positive. (The overflow state is possible but it does not occur.)

You can have a situation where the sum is wrong no matter how you think about it. Suppose Register 7 has a hex 82D8 in it. The instruction

```
AI  R7,>A72C
```

produces a 16-bit sum of hex 2A04. The sum is wrong no matter how you interpret the numbers. The real unsigned sum is hex 12A04 which requires 17 bits to express; therefore, the 16-bit sum hex 2A04 is the wrong unsigned sum and the Carry status bit is set to one. Interpreting the two addends as signed numbers, hex 82D8 is negative and hex A72C is also negative. The sum, hex 2A04, is positive. The instruction added two numbers of the same sign and produced a sum of different sign; therefore, the signed sum is also wrong and the Overflow status bit is set to one.

The Carry and Overflow status bits are affected by most of the arithmetic instructions.

14.1.2   The Add Words Instruction (A)

The Add Words instruction (A) adds two 16-bit numbers together. It requires two operands, both of which can use any of the five general addressing modes. The number specified by the first operand address is added to the number specified by the second operand address and the sum replaces the contents of the second operand address. The addition affects the Carry and Overflow status bits. The sum is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

As an example, suppose memory word ZEBRA contains a hex 1234, memory word hex B74E contains a hex 8AEE, and Register 9 has a zero in it before this instruction is performed.

    A   @ZEBRA,@>B74E(R9)

The instruction adds hex 1234 and hex 8AEE, producing a sum of hex 9D22. The Logical Greater Than status bit is one, the Arithmetic Greater Than status bit is zero, and the Equal status bit is zero. The Carry status bit is zero and the Overflow status bit is zero.

Following this instruction, would a JNO instruction cause a jump? It would jump since the Overflow status bit is not set. Would a JOC instruction cause a jump? It wouldn't jump since the Carry status bit is not set. Would a JHE instruction cause a jump. It would jump because the Logical Greater Than status bit is set.

14.1.3   The Add Bytes Instruction (AB)

The Add Bytes instruction (AB) is like the Add Words instruction except that it adds two bytes and it affects the Odd Party status bit.

As an example, suppose memory word ZEBRA contains a hex 1234, memory word hex B74E contains a hex 8AEE, and Register 9 has a one in it before this instruction is performed.

    A   @ZEBRA,@>B74E(R9)

The instruction adds hex 12 and hex EE, producing a sum of zero. The sum replaces the contents of byte address hex B74F so that word address hex B74E contains hex 8A00.

The Logical Greater Than status bit is zero, the Arithmetic Greater Than status bit is zero, and the Equal status bit is one. The Carry status bit is one and the Overflow status bit is zero.

Following this instruction, would a JNO instruction cause a jump? Yes, it would jump since the Overflow status bit is not set. Would a JOC instruction cause a jump? Yes, it would jump since the Carry status bit is set. Would a JHE instruction cause a jump. Yes, it would jump because the Equal status bit is set.

14.2   The Subtract Instructions (S and SB)

The subtract instructions subtract one number from another to get a difference. The subtraction operation affects the Carry and Overflow status bits. The result is compared to zero and this comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

14.2.1   The Subtract Words Instruction (S)

The Subtract Words instruction (S) requires two operands, both of which can use any of the five general addressing modes. The first operand specifies the address of the number to subtract; the second operand is the address from which the number is subtracted.

As an example, suppose memory word ZEBRA contains a hex 1234, memory word hex B74E contains a hex 8AEE, and Register 9 has a zero in it before this instruction is performed.

```
S    @ZEBRA,@>B74E(R9)
```

The instruction subtracts hex 1234 from hex 8AEE, producing a difference of hex 78BA. The Logical Greater Than status bit is one, the Arithmetic Greater Than status bit is one, and the Equal status bit is zero. The Carry status bit is one and the Overflow status bit is one.

The computer performs the subtraction operation by adding the two's complement of the first operand (hex EDCC) to hex 8AEE.

14.2.2.   The Subtract Bytes Instruction (SB)

The Subtract Bytes instruction (SB) works like the Add Words instruction except that it subtracts a byte value from another byte value. In addition, it affects the Odd Parity status bit.

As an example, suppose memory word ZEBRA contains a hex 1234, memory word hex B74E contains a hex 8AEE, and Register 9 contains a one before this instruction is performed.

```
SB   @ZEBRA,@>B74E(R9)
```

The instruction subtracts hex 12 from hex EE, resulting in a difference of hex DC. The Logical Greater Than status bit is one, the Arithmetic Greater Than status bit is zero, and the Equal status bit is zero. The Carry status bit is one and the Overflow status bit is zero.

14.3 The Increment and Decrement Instructions (INC, INCT, DEC, and DECT)

The increment and decrement instructions are arithmetic instructions that use fixed numbers for one of their operators. Each instruction requires only one operand that can use any of the five general addressing modes.

These four instructions are useful for addressing manipulations. For example, if you are using a register for indirect addressing or indexed addressing, you can adjust the address value in the register to adjacent addresses using these instructions. Incrementing the contents by one (INC) points the register to the next sequential byte address. Decrementing the contents by one (DEC) points to the previous byte address. Incrementing the contents by two (INCT) points the register to the next sequential word address. Decrementing the contents by two (DECT) points to the previous word address.

The DEC instruction is also especially useful for loop control operations. Very often a program subtracts one from a loop counter each time the loop is performed. The DEC instruction is ideal for this. The programs you've seen up to this point have used the Add Immediate instruction for this operation. Now that it's been introduced, the DEC instruction is a better choice.

Likewise, the INC instruction can be used for loop control. You can use a negative value for the initial loop count and increment the contents toward zero with each iteration of the loop.

14.3.1 The Increment Instruction (INC)

The Increment instruction (INC) adds one to a number. It has only one operand that can use any of the five general addressing modes.

It is an arithmetic operation and affects the Carry and Overflow status bits. The result of the incrementing is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

As an example, the following instruction adds one to the contents of memory location 60000.

```
INC   60000
```

14.3.2   The Increment by Two Instruction (INCT)

The Increment by Two instruction (INCT) adds two to a number. It has one operand that can use any of the five general addressing modes.

The instruction performs an arithmetic operation and affects the Carry and Overflow status bits. The result of the incrementing is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

As an example, the following instruction adds two to the contents of the memory location pointed to by Register 6.

```
INCT  *R6
```

14.3.3   The Decrement Instruction (DEC)

The Decrement instruction (DEC) subtracts one from a number. The single operand can use any of the five general addressing modes.

It is an arithmetic operation and affects the Carry and Overflow status bits. The result of the decrementing is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

As an example, the following instruction subtracts one from the contents of Register 8.

```
DEC  R8
```

14.3.4   The Decrement by Two Instruction (DECT)

The Decrement by Two instruction (DECT) subtracts two from a number. Its lone operand can use any of the five general addressing modes.

It is an arithmetic operation and affects the Carry and Overflow status bits. The result of the incrementing is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

As an example, the following instruction subtracts two from the contents of Register 14.

```
DECT  R14
```

14.4   The Negate Instruction (NEG)

The Negate instruction (NEG) negates a number by forming the two's complement of the number. It has one operand that can use any of the five general addressing modes.

As an example, suppose memory location LIZARD contains a 3 (hex 0003) before the following instruction is performed.

```
NEG  @LIZARD
```

The instruction leaves a hex FFFD (a −3) in LIZARD.

The result of the negation operation is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

The computer performs this instruction by forming the two's complement of the original value. The two's complement is formed by, first, taking the one's complement and then adding one. The addition of one to the one's complement affects the Carry and Overflow status bits.

In case you play assembly language trivia, the Carry status bit is set to one only when the original value is zero. Any other value causes the Carry status bit to be cleared to zero.

Whenever you negate a number, you end up with a number of equal absolute value, but of opposite sign. For example, if Register 11 has a hex FFFF (a −1) in it, the instruction

```
NEG  R11
```

leaves a hex 0001 (a +1) in Register 11.

There is an exception, however. If the original value happens to be hex 8000 which is the smallest, or most negative, number possible in 16 bits, the computer can't produce a positive equivalent. If this happens, the Overflow status bit is set to one. Any other value clears the Overflow status bit.

14.5   The Absolute Value Instruction (ABS)

The Absolute Value instruction (ABS) does just what its name implies. It takes the absolute value of a number. It has one operand that lets you use any of the five general addressing modes.

Effectively, the instruction works this way. If the number specified by the operand is positive value, the number is left unchanged. If the number is a negative value, the two complement of the number is formed.

The Carry and Overflow status bits are affected when the two's complement of a numb is formed. The Logical Greater Than, Arithmetic Greater Than, and Equal status bits a affected also, but they are affected based upon a comparison of the *original number* zero.

For example, suppose memory location BORNEO contains a hex FB39 (a negativ number). The instruction

    ABS @BORNEO

leaves a hex 04C7 (the positive counterpart to hex FB39) in BORNEO. The Logical Great Than status bit is one. The Arithmetic Greater Than status bit is zero because FB39 not arithmetically greater than zero. And the Equal status bit is zero.

As with the NEG instruction, the Overflow status bit is set to one only when the origin value is hex 8000. In this case, hex 8000 remains in the location.

14.6   The Multiply and Divide Instructions (MPY and DIV)

The arithmetic instructions include a single instruction multiply and a single instructic divide. Some computers don't have multiply and divide operations in their instructic set. You have to write a program of several instructions to perform multiplication ar division.

The Multiply and Divide instructions treat the numbers as unsigned numbers. If you' thinking of the numbers as signed numbers, you have to keep track of the signs.

14.6.1   The Multiply Instruction (MPY)

The Multiply instruction (MPY) multiplies two 16-bit numbers and produces a 32-b product. The instruction requires two operands. The first operand can use any of the fiv general addressing modes. The second operand uses only register direct addressing moc and the second number must be in a register.

The number addressed by the first operand is multiplied by the number in the registt and the 32-bit product goes into the second operand. The product goes into the registe

This brings us to a question. How do you get a 32-bit product into a 16-bit register? The answer is, you don't. That 's like trying to squeeze a number 14 foot into a size 7 shoe. Here's how the computer handles this situation. The computer puts the most significant 16 bits of the product into the register and the least significant 16 bits spill into the next register. For example, if the second operand is Register 8, the 32-product goes into Registers 8 and 9.

Look at the following example. Suppose memory location HOGG contains a 3, Register 5 contains 4, and Register 6 contains 5 before the following instruction is performed.

```
MPY  @HOGG,R5
```

The computer multiplies 3 (hex 0003) times 4 (hex 0004), producing a 32-bit product of 12 (hex 0000 000C). The most significant 16 bits of the product (hex 0000) goes into Register 5 and the least significant 16 bits (hex 000C) goes into Register 6.

```
            Before          After

(HOGG) =   >0003           >0003
(R5)   =   >0004           >0000
(R6)   =   >0005           >000C
```

No status bits are affected by the MPY instruction.

You need to be aware that whatever is in the register following the one specified as the second operand is overlayed as a result of the multiplication.

You can use any register for the second operand. If you use Register 15, the product is placed in Register 15 and in the general memory location following Register 15. If you use Register 15 as the second operand for a MPY instruction, make sure the word of memory following the workspace can be written over.

14.6.2   The Divide Instruction (DIV)

The Divide instruction (DIV) divides a 16-bit divisor into a 32-bit dividend. It produces a 16-bit quotient and a 16-bit remainder. The instruction requires two operands. The first operand can use any of the five general addressing modes. The second operand uses only register direct addressing mode.

With the Divide instruction, the second operand is the first register of an implied register pair. The 32-bit dividend is in the register pair. The first 16 bits of the dividend (the most

173

significant word) are in the first register and the second word (least significant 16 bits) of the dividend) are in the second register of the pair.

```
   <------32-bit Divided------>
   +--------------+--------------+
   |    MSW       |    LSW       |
   +--------------|--------------+
        Rn             Rn+1
```
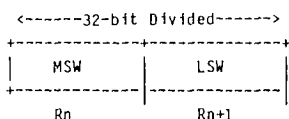
The number addressed by the first operand is divided into the 32-bit number in the register pair. The resulting 16-bit quotient goes into the first register of the pair and the 16-bit remainder goes into the second register of the pair.

```
   <--16 bits--->   <---16 bits--->
   +---------------|---------------+
   |   Quotient    |   Remainder   |
   +---------------|---------------+
        Rn               Rn+1
```
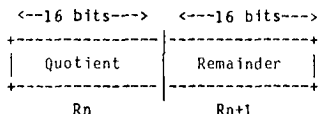
Look at an example. Suppose memory location HOGG contains a 3, Register 5 contains 0, and Register 6 contains (hex E) 14 before the following instruction is performed.

```
DIV  @HOGG,R5
```

The computer divides 3 (hex 0003) into 14 (a 32-bit hex 0000 000E in Register 5 and 6. It produces a 16-bit quotient of 4 which goes into Register 5 and a 16-bit remainder of 2 which goes into Register 6. When finished, the instruction leaves a hex 0004 in Register 5 and a hex 0002 in Register 6.

```
              Before         After
(HOGG) =      >0003          >0003
(R5)   =      >0000          >0004
(R6)   =      >000E          >0002
```

The DIV instruction affects one status bit — the Overflow status bit.

Prior to performing the divide operation, the computer compares the 16-bit divisor with the first word of the dividend which is the contents of the register in the second operand. If the divisor is smaller than the first word of the dividend, the computer sets the Overflow status bit to one and doesn't perform the division. If the divisor is smaller than the first word of the dividend, the quotient will exceed 16 bits and, under those conditions, the computer sets the Overflow status bit and doesn't divide.

As an example, suppose memory location R2D3 contains a hex 0003, Register 10 contains a hex 0005, and Register 11 contains a hex 0000. The instruction

```
DIV  @R2D3,R10
```

causes the Overflow status bit to be set. The contents of memory location R2D3, Register 10, and Register 11 are unchanged.

This comparison of the divisor to the most significant word of the dividend prior to performing the division prevents the computer from attempting to divide by 0 (one of those irrational acts which produces a result approaching infinity and threatens the stability of the cosmos).

14.7   Program Example

The following program performs a signed multiplication of two 16-bit numbers and produces a 32-bit signed result. The program expects the two numbers to be already in Register 0 and Register 1 when it starts running. It leaves the 32-bit signed product in Registers 0 and 1.

The program uses the Multiply instruction (MPY). Since the Multiply instruction only multiplies unsigned (absolute) values, the program must determine the sign of the numbers and the sign of the product.

If you multiply two numbers together, the result is positive when both numbers are either positive or negative. The result is negative if the numbers have opposite signs.

The program checks the sign of both numbers. If they are both positive, the number is already expressed as an absolute value and the numbers can be multiplied directly. The absolute value of the product expresses the positive result.

If the two numbers are both negative, the program forms the absolute values of them and multiplies the absolute values. The absolute value of the product expresses the positive result.

If the numbers have opposite signs, the program forms the absolute value of both numbers and multiplies them. The result is the absolute value of the negative product. The program then must express the 32-bit absolute product as a 32-bit signed number. The two's complement of a 32-bit number is formed by taking the two's complement of the least significant word and the one's complement of the most significant word. For example,

the 32-bit two's complement of hex 0000 000C (an absolute value of 12) is hex FFFF FFF₄ (a −12).

There's one exception however. If the least significant word of the 32-bit product is zero then the two's complement of the most significant word is formed rather than the one's complement. For example, the 32-bit two's complement of hex 0001 0000 (an absolute value of 65,536) is hex FFFF 0000 (a −65,536).

The program uses several instructions introduced in this chapter, including the ABS, MPY, NEG, and DEC instructions. The program takes advantage of the fact that the ABS instruction affects the Arithmetic Greater Than status bit, as well as the other status bits, based upon a comparison of the original value to zero.

Look at the listing of the program.

```
99/4 ASSEMBLER
VERSION 1.2                                                      PAGE 0001
 0001                          IDT   'SIGNMULT'   SIGNED MULTIPLY
 0002  0000  02E0             LWPI  WS            INITIALIZE WORKSPACE
       0002  0022'
 0003  0004  0740             ABS   R0            FORCE X TO POSITIVE
 0004  0006  1104             JLT   NEGX          IF X NEGATIVE, JUMP
 0005  0008  0741             ABS   R1            ELSE FORCE Y POSITIVE
 0006  000A  1104             JLT   NEGY          IF Y NEGATIVE, JUMP
 0007  000C  3801  MPYPOS     MPY   R1,R0         MULTIPLY X AND Y (SAME SIGNS)
 0008  000E  1007             JMP   EXIT          GO TO EXIT
 0009  0010  0741  NEGX       ABS   R1            FORCE Y POSITIVE
 0010  0012  11FC             JLT   MPYPOS        GO MULTIPLY (SAME SIGNS)
 0011  0014  3801  NEGY       MPY   R1,R0         MULTIPLY X AND Y (DIFFERENT SIGNS)
 0012  0016  0500             NEG   R0            TAKE TWO'S COMPLEMENT OF MSW
 0013  0018  0501             NEG   R1            TAKE TWO'S COMPLEMENT OF LSW
 0014  001A  1301             JEQ   EXIT          IF LSW ZERO, JUMP
 0015  001C  0600             DEC   R0            ELSE FORM ONE'S COMPLEMENT OF MSW
 0016  001E  0420  EXIT       BLWP  @0            GO HOME
       0020  0000
 0017  0022  WS         BSS   22            WORKSPACE
 0018                          END
```

```
99/4 ASSEMBLER
VERSION 1.2                                                      PAGE 0002
  ' EXIT     001E    ' MPYPOS   000C    ' NEGX     0010    ' NEGY     0014
    R0       0000      R1       0001      R10      000A      R11      000B
    R12      000C      R13      000D      R14      000E      R15      000F
    R2       0002      R3       0003      R4       0004      R5       0005
    R6       0006      R7       0007      R8       0008      R9       0009
  ' WS       0022
  0000 ERRORS
```

Following the LWPI instruction (statement 2), the ABS instruction (statement 3) takes the absolute value of the number in Register 0 which is called the X value. The absolute value is left in Register 0 and the first three status bits, including the Arithmetic Greater Than and Equal status bits, are affected based upon a comparison of the original value to zero.

If the original X value was negative, the JLT instruction at statement 4 jumps to NEGX. Otherwise, the program goes on to the next instruction. The ABS instruction (statement 5) forms the absolute value of Y if X was positive.

If the Y value was negative, the JLT instruction at statement 6 jumps. Otherwise, the program goes on to the MPY instruction labeled MPYPOS. This multiply instruction is performed only if both numbers have the same sign. It multiplies the absolute values in Register 0 and Register 1 and leaves the absolute value of the 32-bit product in Registers 0 and 1. Since this MPY instruction is performed only if the two numbers have the same sign, the absolute value of the product expresses the positive product directly. Therefore, the program performs a JMP instruction (statement 8) to the Go-home instruction labeled EXIT.

If the X value is negative, the ABS instruction labeled NEGX receives control. It forms the absolute value of Y and leaves the absolute value in Register 1.

The JLT instruction (statement 10) jumps to the Multiply instruction labeled MPYPOS is Y is negative. This jump is taken only when X is negative and Y is negative. When the jump is taken, Register 0 has the absolute value of X and Register 1 has the absolute value of Y.

If the JLT instruction (statement 10) does not jump, program control passes to the MPY instruction at statement 11. This multiply instruction is performed only if the X and Y values have opposite signs. When it's performed, the absolute value of X is in Register 0 and the absolute value of Y is in Register 1. After it is performed, the absolute value of the 32- bit product is in Registers 0 and 1; the program must take the two's complement of this 32-bit number.

The NEG instruction at statement 12 forms the two's complement of the most significant word of the product. The next NEG instruction at statement 13 forms the two's complement of the least significant word of the product. The result is compared to zero and affects several status bits, including the Equal status bit. The Equal status bit is set to one if the result is zero and the result is zero only if the original value was zero.

Next, the JEQ instruction at statement 14 jumps to EXIT if the least significant word of the product is zero. Otherwise, program control passes to the next instruction (statement 15).

The DEC instruction (statement 15) is performed when the least significant word of the product is non-zero. In that case, the two's complement of the most significant word of the product in Register 0 is reduced to the one's complement by subtracting one from the contents of Register 0.

The program terminates at the Go-home instruction labeled EXIT.

Use the assembler to assemble the program and then use the Loader to load the resulting object program. Load the Debugger with the program and use the Debugger to control the program.

Before running the program, use the Debugger to place numbers in the program's Registers 0 and 1.

Set a breakpoint at the BLWP instruction and run the program.

After running the program, use the Debugger to look at the same registers for the results.

Run the program several times with different numbers in R0 and R1. Use two positive numbers, two negative numbers and two numbers of different signs.

This chapter illustrates the use of the Arithmetic instructions. The next chapter introduces the Logical instructions.

Chapter 15

THE LOGICAL INSTRUCTIONS

This chapter introduces the group of logical instructions. The main job of these instructions is to perform the logical operations of AND, OR, Exclusive OR, or related operations on data. There are 10 logical instructions. They are listed below with their names, operation codes, and a description of the kinds of addressing modes you can use with the instructions.

In the following list, G means that an operand is a general addressing mode operand and can use any of the five general addressing modes. An R means that an operand must be a working register and it can use only register direct addressing mode. An IOP means that an operand must use immediate addressing and the operand is a data value, rather than the address of a data value.

Name	Operation Code	Addressing Mode
And Immediate	ANDI	R,IOP
Set Zeros Corresponding	SZC	G,G
Set Zeros Corresponding Byte	SZCB	G,G
Or Immediate	ORI	R,IOP
Set Ones Corresponding	SOC	G,G
Set Ones Corresponding Byte	SOCB	G,G
Exclusive Or	XOR	G,R
Invert	INV	G
Clear	CLR	G
Set to One	SETO	G

Most of the instructions in this group work with individual bits in a data quantity and define the state of the selected bits.

The first three of these instructions (ANDI, SZC, and SZCB) perform a logical AND operation, or something closely related to the AND operation, on data.

The next three instructions, ORI, SOC, and SOCB, perform a logical OR operation on data. The XOR instruction performs an exclusive OR operation on data.

The INV instruction performs a logical NOT operation on the bits in a word. The last two instructions (CLR and SETO) set the content of a word to predefined values.

15.1   The AND Operation Instructions (ANDI, SZC, and SCZB)

There are three instructions that perform a logical AND operation or something closely related to the AND operation.

The AND operation selects the state of a bit based upon the state of two other bits. In the following truth table for the AND operation, X is the state of one bit and Y is the state of the other bit. Notice that the result is a one only if both X and Y are ones.

```
              AND
          Truth Table

    X bit │ Y bit │ Result
    ------------------------------
       0  │   0   │    0
       0  │   1   │    0
       1  │   0   │    0
       1  │   1   │    1
```

The AND operation is useful for selectively turning off bits in a data quantity. As an example, consider an AND operation between two byte values. One byte is called X and the other is called Y. The AND operation is performed on each of the eight pairs of X and Y bits and produces an 8-bit result called R.

```
X  =     0 0 1 1 1 0 1 0
Y  =     0 1 0 1 1 1 0 0
         ------------------
R  =     0 0 0 1 1 0 0 0
```

Notice that for each X bit that is zero, the corresponding R bit is zero. For each X bit that is one, the corresponding R bit is the same state as the Y bit.

Call the 8-bit X value a "bit mask". Everywhere there is a zero in the bit mask, the corresponding bit in the R byte is zero; everywhere there's a one in the bit mask, the corresponding bit in the R byte is the same as the Y bit. Effectively, the bit mask is forcing zeros at selected locations of the Y value and leaving unselected locations unchanged.

The AND operation is useful for turning off, or setting to zero, selected bits in a data quantity.

15.1.1   The And Immediate Instruction (ANDI)

The And Immediate instruction (ANDI) performs a logical AND operation on two word values. The instruction requires two operands, the first uses only register direct addressing and the second is an immediate operand.

The instruction performs a logical AND operation between the contents of the register and the immediate operand. The result replaces the content of the register.

The result is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

As an example, suppose Register 6 contains a hex 5C69 before the following instruction is performed

```
ANDI   R6,>3A0F
```

The instruction performs an AND operation between a bit in the register and a corresponding bit in the immediate operand. The AND result of that pair of bits replaces the bit in the register. As a result of this instruction, a hex 1809 is left in Register 6.

```
IOP        =  0011  1010  0000  1111  =  >3A0F
(R6) Before  =  0101  1100  0110  1001  =  >5C69
----------------------------------------------------
(R6) After   =  0001  1000  0000  1001  =  >1809
```

The hex 1809 is compared to zero, causing the Logical Greater Than status bit to be one, the Arithmetic Greater Than status bit to be one, and the Equal status bit to be zero.

15.1.2   The Set Zeros Corresponding Instruction (SZC)

The Set Zeros Corresponding Instruction (SZC) performs an operation similar to a logical AND operation on two word values. The instruction requires two operands, both of which can use any of the five general addressing modes.

The instruction performs a logical AND operation between the *complement* of the first value and the uncomplemented second value. The result replaces the contents of the

second operand. The result is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

As an example, suppose Register 6 contains a hex 3A0F and memory word TWEETE contains a hex 5C69 before the following instruction is performed.

```
SZC  R6,@TWEETE
```

The instruction performs an AND operation between the complement of a bit in Register 6 and the corresponding bit in TWEETE. The AND result of that pair of bits replaces the bit in the TWEETE. As a result of this instruction, a hex 4460 is left in TWEETE.

```
                (R6)  =  0011  1010  0000  1111 = >3A0F
                -----------------------------------------
Complement of (R6)  =  1100  0101  1111  0000 = >C5F0
(TWEETE) Before     =  0101  1100  0110  1001 = >5C69
----------------------------------------------------
(TWEETE) After      =  0100  0100  0110  0000 = >4460
```

The hex 4460 is compared to zero, causing the Logical Greater Than status bit to be one, the Arithmetic Greater Than status bit to be one, and the Equal status bit to be zero.

The instruction is called Set Zeros Corresponding because the one bits in the first operand (the bit mask) set zeros in the corresponding bits of the second operand. Zero bits in the first operand leave the corresponding bits in the second operand unchanged.

15.1.3   The Set Zeros Corresponding Byte Instruction (SZCB)

The Set Zeros Corresponding Byte instruction (SZCB) works just like the Set Zeros Corresponding (SZC) instruction except it uses two byte values, rather than word values. It additionally affects the Odd Parity status bit. The instruction requires two operands, both of which can use any of the five general addressing modes.

As an example, suppose Register 6 contains a hex 3A0F and memory word TWEETE contains a hex 5C69 before the following instruction is performed.

```
SZCB  R6,@TWEETE
```

The instruction performs an AND operation between the complement of a bit in the left byte of Register 6 and the corresponding bit in byte address TWEETE. The result of that

pair of bits replaces the bit in TWEETE. As a result of this instruction, a hex 44 is left in byte address TWEETE. (A hex 4469 is left in word address TWEETE).

```
              (R6)   =  0011  1010  0000  1111 = >3A0F
                       -----------------------------
  Compliment of (R6)  =  1100  0101  1111  0000 = >C5F0
  (TWEETE) Before     =  0101  1100  0110  1001 = >5C69
                       -----------------------------
  (TWEETE) After      =  0100  0100  0110  1001 = >4469
                                     +---VV---+
                                     NOT AFFECTED
```

The byte result, hex 44, is compared to zero, causing the Logical Greater Than status bit to be one, the Arithmetic Greater Than status bit to be one, and the Equal status bit to be zero. The Odd Parity status bit is a zero.

15.2   The OR Operation Instructions (ORI, SOC, and SOCB)

There are three instructions which perform a logical OR operation.

The OR operation selects the state of a bit based upon the state of two other bits. In the following truth table for the OR operation, X is the state of one bit and Y is the state of the other bit. Notice that the result is a one if either X or Y is one.

```
              OR
          Truth Table

  X bit │ Y bit │  Result
  ----------------------------
     0  │   0   │    0
     0  │   1   │    1
     1  │   0   │    1
     1  │   1   │    1
```

The OR operation is useful for selectively turning on bits in a data quantity. As an example, consider an OR operation between two byte values. One byte is called X and the other is called Y. The OR operation is performed on each of the eight pairs of X and Y bits and produces an 8-bit result called R.

```
  X  =     0 0 1 1 1 0 1 0
  Y  =     0 1 0 1 1 1 0 0
          ------------------
  R  =     0 1 1 1 1 1 1 0
```

Notice that for each X bit that is one, the corresponding R bit is one. For each X bit that is zero, the corresponding R bit is the same state as the Y bit.

Call the 8-bit X value a bit mask. Everywhere there is a one in the bit mask, the corresponding bit in the R byte is one; everywhere there's a zero in the bit mask, the corresponding bit in the R byte is the same as the Y bit. Effectively, the bit mask is forcing ones at selected locations of the Y value and leaving unselected locations unchanged.

The OR operation is useful for turning on (setting to one) selected bits in a data quantity.

15.2.1 The Or Immediate Instruction (ORI)

The Or Immediate instruction (ORI) performs a logical OR operation on two word values. The instruction requires two operands, the first uses only register direct addressing and the second is an immediate operand.

The instruction performs a logical OR operation between the contents of the register and the immediate operand. The result replaces the contents of the register. The result is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

As an example, suppose Register 6 contains a hex 5C69 before the following instruction is performed

```
ANDI   R6,>3A0F
```

The instruction performs an OR operation between a bit in the register and a corresponding bit in the immediate operand. The OR result of that pair of bits replaces the bit in the register. As a result of this instruction, a hex 7E6F is left in Register 6.

```
        IOP  =  0011  1010  0000  1111  =  >3A0F
(R6) Before  =  0101  1100  0110  1001  =  >5C69
-----------------------------------------
(R6) After   =  0111  1110  0110  1111  =  >7E6F
```

The hex 7E6F is compared to zero, causing the Logical Greater Than status bit to be one, the Arithmetic Greater Than status bit to be one, and the Equal status bit to be zero.

15.2.2   The Set Ones Corresponding Instruction (SOC)

The Set Ones Corresponding Instruction (SOC) performs a logical OR operation on two word values. The instruction requires two operands, both of which can use any of the five general addressing modes.

The instruction performs a logical OR operation between the first value and the second value. The result replaces the contents of the second operand. This result is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

As an example, suppose Register 6 contains a hex 3A0F and memory word TWEETE contains a hex 5C69 before the following instruction is performed

```
SOC  R6,@TWEETE
```

The instruction performs an OR operation between a bit in Register 6 and the corresponding bit in TWEETE. The result of that pair of bits replaces the bit in the TWEETE. As a result of this instruction, a hex 7E6F is left in TWEETE.

```
        (R6)         =  0011  1010  0000  1111  = >3A0F
   (TWEETE) Before  =  0101  1100  0110  1001  = >5C69
   -------------------------------------------------
   (TWEETE) After   =  0111  1110  0110  1111  = >7E6F
```

The hex 7E6F is compared to zero, causing the Logical Greater Than status bit to be one, the Arithmetic Greater Than status bit to be one, and the Equal status bit to be zero.

The instruction is called Set Ones Corresponding because the one bits in the first operand (the bit mask) set ones in the corresponding bits of the second operand. Zero bits in the first operand leave the corresponding bits in the second operand unchanged.

15.2.3   The Set Ones Corresponding Byte Instruction (SOCB)

The Set Ones Corresponding Byte instruction (SOCB) works just like the Set Ones Corresponding (SOC) instruction except it uses two byte values, rather than word values. It additionally affects the Odd Parity status bit. The instruction requires two operands, both of which can use any of the five general addressing modes.

As an example, suppose Register 6 contains a hex 3A0F and memory word TWEETE contains a hex 5C69 before the following instruction is performed.

```
SOCB  R6,@TWEETE
```

The instruction performs an OR operation between a bit in the left byte of Register ι
and the corresponding bit in byte address TWEETE. The result of that pair of bits replace
the bit in TWEETE. As a result of this instruction, a hex 7E is left in byte address TWEETE
A hex 7E69 is left in word address TWEETE.

```
   (R6)           =  0011  1010  0000  1111  =  >3A0F
(TWEETE) Before =  0101  1100  0110  1001  =  >5C69
-----------------------------------------------------
(TWEETE) After  =  0111  1110  0110  1001  =  >7E69
                                +---VV---+
                                NOT AFFECTED
```

The byte result, hex 7E, is compared to zero, causing the Logical Greater Than status bit
to be one, the Arithmetic Greater Than status bit to be one, and the Equal status bit to
be zero. The Odd Parity status bit is a zero.

15.3 The Exclusive Or Instruction (XOR)

The Exclusive OR instruction (XOR) is the only instruction that performs an exclusive
OR operation.

The Exclusive OR operation selects the state of a bit based upon the state of two other
bits. In the following truth table for the Exclusive OR operation, X is the state of one bit
and Y is the state of the other bit. Notice that the result is a one only if X *or* Y is a one,
but not both (X and Y must be different).

```
        Exclusive OR
        Truth Table

X bit | Y bit |  Result
-------------------------------
   0       0        0
   0       1        1
   1       0        1
   1       1        0
```

The Exclusive OR operation is useful for selectively changing the state of bits in a data
quantity. As an example, consider an Exclusive OR operation between two byte values.

One byte is called X and the other is called Y. The Exclusive OR operation is performed on each of the eight pairs of X and Y bits and produces an 8-bit result called R.

```
X  =      0 0 1 1 1 0 1 0
Y  =      0 1 0 1 1 1 0 0
          -----------------
R  =      0 1 1 0 0 1 1 0
```

Notice that for each X bit that is one, the corresponding R bit is changed. For each X bit that is zero, the corresponding R bit is the same state as the Y bit.

Call the 8-bit X value a bit mask. Everywhere there is a one in the bit mask, the corresponding bit in the R byte is changed; everywhere there's a zero in the bit mask, the corresponding bit in the R byte is the same as the Y bit. Effectively, the bit mask is inverting bits at selected locations of the Y value and leaving unselected locations unchanged.

The Exclusive OR operation is useful for inverting or changing the state of selected bits in a data quantity.

The Exclusive OR instruction (XOR) performs an Exclusive OR operation on two word values. The instruction requires two operands, the first can use any of the five general addressing modes and the second uses only register direct addressing.

The instruction performs an Exclusive OR operation between the contents of the first operand and the register. The result replaces the contents of the register. This result is compared to zero and that comparison affects the Logical Greater Than, Arithmetic Greater Than, and Equal status bits.

As an example, suppose memory word TWEETE contains a hex 5C69 and Register 6 contains a hex 3A0F before the following instruction is performed.

```
    XOR  @TWEETE,R6
```

The instruction performs an Exclusive OR operation between a bit in TWEETE and the corresponding bit in Register 6. The Exclusive OR result of that pair of bits replaces the bit in Register 6. As a result of this instruction, a hex 6666 is left in Register 6.

```
(TWEETE)    =  0101  1100  0110  1001  =  >5C69
(R6) Before =  0011  1010  0000  1111  =  >3A0F
---------------------------------------------------
(R6) After  =  0110  0110  0110  0110  =  >6666
```

187

The hex 6666 is compared to zero, causing the Logical Greater Than status bit to be o₁
the Arithmetic Greater Than Status bit to be one, and the Equal status bit to be zer₍

15.4 The Invert Instruction (INV)

The Invert instruction (INV) performs a logical NOT function. It inverts the state of tl
bits in a word. The instruction requires one operand and the operand can use any of tl
five general addressing modes. The result of the operation is compared to zero and tl
Logical Greater Than, Arithmetic Greater Than, and Equal status bits are affected bas₍
upon that comparison.

As an example, assume that Register 2 contains 8. The following INV instruction inver
the bits in the fifth word of a memory file called COMPS.

```
INV  @COMPS(R2)
```

If that word contains a hex 5E6D before the instruction is performed, it contains a he
A192 after the instruction is performed.

```
(COMPS (R2))  Before = 0101 1110 0110 1101 = >5E6D
              After  = 1010 0001 1001 0010 = >A192
```

The Logical Greater Than status bit is one, the Arithmetic Greater Than status bit is zer
and the Equal status bit is zero as a result of the instruction.

When you invert the state of each bit in a data quantity, you take the one's compleme₁
of the value. The Invert instruction forms the one's complement of a word.

15.5 The Initialize to Constant Instructions (CLR and SETO)

There are two instructions that initialize a location to a constant value. These instructio₁
are useful for setting locations to common initial conditions.

15.5.1 The Clear Instruction (CLR)

The Clear instruction (CLR) initializes a word to zero. The instruction requires on
operand that can use any of the five general addressing modes. No status bits are affecte₍

Programs very often initialize storage locations to zero before performing operation
Earlier, the LI instruction was used to initialize a register to zero. The Clear instructio₁
however, is a more effective way.

e,   For example, the instruction

```
CLR  R8
```

sets the contents of Register 8 to zero.

Notice that the CLR instruction always addresses a word location, not a byte.

15.5.2   The Set to One Instruction (SETO)

The Set to One instruction (SETO) initializes a word to minus one (hex FFFF). The instruction requires one operand which can use any of the five general addressing modes. No status bits are affected.

The value hex FFFF, which is often called minus one, because that's its value if you interpret it as a signed number, is sometimes used as a marker for the end of a file or a special code within a program.

As an example of how it operates, the instruction

```
SETO  *R7
```

sets the contents of the location pointed to by Register 7 to hex FFFF.

Notice that the SETO instruction always addresses a word location, not a byte.

15.6   Program Example

The following program examines the two bytes in each word of a ten-word file. If both bytes contain either an odd number or an even number, the program clears the word in a corresponding word of a second ten-word file; otherwise, the corresponding word of the second file is set to hex FFFF. The program takes advantage of the fact that an odd number has a one bit in the least significant bit position and an even number has a zero bit in the least significant bit position. The program illustrates the use of several of the logical instructions introduced in this chapter. Look at the listing.

```
99/4 ASSEMBLER
VERSION 1.2                                                    PAGE 0001
0001                              IDT  'EVENODD'   DETECT EVEN AND ODD NUMBERS
0002 0000 02E0                    LWPI WS          POINT TO WORKSPACE
     0002 0054'
0003 0004 0202                    LI   R2.BUFFER   POINT TO DATA WORDS
     0006 002C'
0004 0008 0203                    LI   R3.FLAGS    POINT TO SAME/DIFFERENT FLAGS
     000A 0040'
0005 000C 0204                    LI   R4.10       SET A COUNTER
     000E 000A
0006 0010 C0B2      GTWORD MOV    *R2+.R0          GET A DATA WORD (AUTOINCREMENT)
0007 0012 0240             ANDI   R0.)0101         TURN OFF ALL BITS EXCEPT LSB'S
     0014 0101
0008 0016 C040             MOV    R0.R1            COPY RESULT
0009 0018 06C1             SWPB   R1               EXCHANGE THE TWO BYTES IN THE COPY
0010 001A 2840             XOR    R0.R1            BOTH SAME OR DIFFERENT?
0011 001C 04F3             CLR    *R3+             ASSUME SAME - SET "SAME" FLAG
0012 001E 1302             JEQ    SAME             YES. THEY ARE SAME
0013 0020 05E3             INV    @-2(R3)          DIFFERENT - SET "DIFFERENT" FLAG
     0022 FFFE
0014 0024 0604      SAME   DEC    R4               DECREMENT COUNTER
0015 0026 15F4             JGT    GTWORD           IF COUNTER () 0. CHECK ANOTHER WORD
0016 0028 0420             BLWP   @0               GO HOME
     002A 0000
0017 002C         BUFFER BSS    20               DATA WORDS HERE
0018 0040         FLAGS  BSS    20               FLAGS PUT HERE
0019 0054         WS     BSS    32               WORKSPACE
0020                     END
```

```
99/4 ASSEMBLER
VERSION 1.2                                                    PAGE 0002
 ' BUFFER  002C    ' FLAGS   0040   ' GTWORD  0010     R0     0000
   R1      0001      R10     000A     R11     000B      R12    000C
   R13     000D      R14     000E     R15     000F      R2     0002
   R3      0003      R4      0004     R5      0005      R6     0006
   R7      0007      R8      0008     R9      0009    ' SAME   0024
 ' WS      0054
0000 ERRORS
```

The BSS directive labeled BUFFER at statement 17 reserves a block of memory for the ten-word file containing the pairs of bytes to be analyzed. The BSS directive labeled FLAGS at statement 18 reserves a block of memory for the ten words used to mark the relationship of the bytes in BUFFER.

The first instruction (statement 2), an LWPI, sets up the Workspace Pointer.

Statements 3, 4, and 5 are Load Immediate instructions. The first one points Register 2 to the file of words. The second one points Register 3 to the file FLAGS. The third one initializes a loop count of 10 in Register 4.

The MOV instruction labeled GTWORD copies a word from the BUFFER file into Register 0. Notice it uses register indirect autoincrement addressing mode. After this instruction is performed, Register 0 contains the two bytes of a word and Register 2 points to the following word in the file.

In statement 7, the ANDI instruction turns off or forces to zero all the bits in Register 0 except the rightmost bit in each byte. The MOV instruction at statement 8 copies the result into Register 1. The SWPB instruction (statement 9) exchanges the two bytes in Register 1

The XOR instruction (statement 10) performs an Exclusive OR operation between the contents of Register 0 and Register 1. The result is left in Register 1. The result in register 1 is either a value of zero or a value of one. If the rightmost bit in each byte is the same, the result is zero. If the rightmost bit in each byte is different, the result is one. The instruction compares the result to zero and affects several status bits, including the Equal status bit.

The CLR instruction (statement 11) clears to zero the corresponding word in the FLAGS file. Notice the instruction uses register indirect autoincrement addressing mode, so that after it is performed, Register 3 points to the next word in the FLAGS file. The instruction assumes the two bytes are the same. The CLR instruction doesn't change any status bits. The status bits are in the same state they were after the XOR instruction was performed.

The JEQ instruction (statement 12) analyzes the Equal status bit that was affected by the XOR instruction. It jumps to SAME and skips the next instruction if the result of the XOR instruction is zero which means the bytes are the same. Otherwise, it lets the program go on to the next instruction, the INV instruction.

If the two bytes are different, the INV instruction at statement 13 is performed. It reaches back to the previous word in the FLAGS file and inverts it. Since the previously performed CLR instruction set that word to zero, the INV instruction changes it to all one bits (hex FFFF).

The DEC instruction (labeled SAME) subtracts one from the loop count in Register 4 and closes the loop to GTWORD if the loop count is not yet zero; otherwise, it lets the program fall out of the loop to the Go-home instruction (BLWP).

Use the assembler to assemble the program and then use the Loader to load the resulting object program. Load the Debugger with the program and use the Debugger to control the program.

Before running the program, use the Debugger to place values in the BUFFER file.

Set a breakpoint at the BLWP instruction and run the program.

After running the program, use the Debugger to look at the FLAGS file for the results.

This chapter illustrates the use of the Logical instructions. The next chapter introduces the Branch and Subroutine instructions.

Chapter 16

BRANCH AND SUBROUTINE INSTRUCTIONS

This chapter reviews the concept of subroutines and describes those instructions which are used with subroutines and long-range transfers of control (branches). This chapter also describes context switching, the instructions that can cause a context switch, and explains the events that happen as a result of a context switch.

16.1 Subroutines

Nearly every language offers the ability to define and use subroutines. A subroutine is normally used in a program when a function needs to be performed several times at different locations in the program. By creating a subroutine to perform that function, it can be called from anywhere in the program where that function is needed.

You can imagine that a subroutine looks like this.

```
                    A Subroutine

                 +----------------------+
Entry Point ---> |  -------------    ----+
                 |  -------------       |
                 |  -------------       |
                 |        .             |
                 |        .             >  Instructions
                 |        .             |
                 |  -------------       |
                 |  -------------       |
Exit Point <---  |  -------------    ----+
                 |  ----------       ----+
                 |        .             |
                 |        .             >  Data Area
                 |        .             |
                 |  ----------       ----+
                 +----------------------+
```
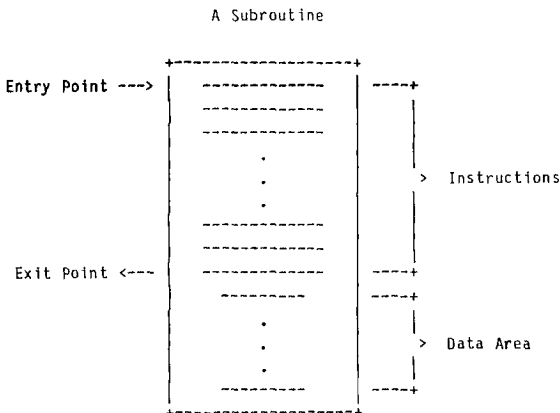
A subroutine consists of a set of instructions and, possibly, an associated area of data. The subroutine has an entry point; that is, a point where it receives control. Although it's possible to construct a subroutine with more than one entry point, most subroutines should have only one. The subroutine has an exit point; that is, the last instruction in the subroutine which is performed and the one that returns control to the program that called the subroutine. It's possible that a subroutine may have more than one exit point, but good programming practice recommends that you use only one.

Within a complete program, you can have several individual programs segments. Some of these program segments may be subroutines that can be called from statements in other parts of the program. Other program segments can be "calling programs" that contain statements which call subroutines. When a calling program calls a subroutine, the subroutine receives program control, performs its job, and then exits by returning control to the calling program. Usually, the calling program is returned control at the location immediately following the instruction that called the subroutine. The location where a calling program is returned control from a subroutine is called the return point.

```
                    A Program
          +---------------------+
          |   -------------      |
          |   -------------      |
    +----  |  calling statement  |
          |   return point      |  <---+
          |         .           |      |
          |         .           |      |
          |   -------------      |      |
          |   -------------      |      |
          |   -------------      |      |
    +--->  |    entry point      |   ---+
          |         .           |      |
          |         .           |      |  > Subroutine
          |         .           |      |
          |    exit point       |  ----+ ---+
          +---------------------+
```
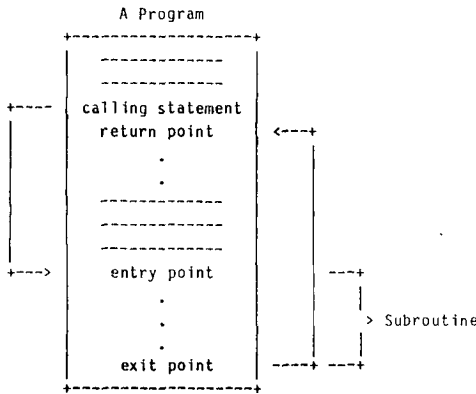
Often, data must be exchanged between a calling program and a subroutine. There are several ways of exchanging data. One way is to use defined areas of memory. For example, assume that the calling program and subroutine are designed to use locations X and Y for passing data. These are specific memory locations used only for passing data. The calling program places the data to be processed in memory location X and then calls the subroutine. The subroutine takes the data from location X, performs the operation on the data, and places the results in memory location Y. The subroutine exits and returns control to the calling program. The calling program is designed to look in memory location Y for the result of the subroutine's operation.

There are other ways of passing data between calling programs and subroutines. Another way is to use several areas of memory for passing data. In this case, the calling program's logic selects an area of memory to use and places the data in that chosen area. When the calling program calls the subroutine, it passes to the subroutine the address of the memory area containing the data to be processed. When the subroutine receives control, it extracts the data from that area and processes it. The subroutine also might select an area of memory in which to place the result. After choosing an area and placing the results there, the subroutine returns to the calling program and passes back to the calling program the address where the result was placed.

A third common technique for exchanging data between a calling program and a subroutine is to use the working registers. With this technique, the calling program simply places data in one or more of the working registers and calls the subroutine. The subroutine retrieves the data from the register(s), performs its operation on the data, places the results in one or more registers, and returns control to the calling program. The calling program looks in the register(s) for the result.

Look now at some of the ways that you can call subroutines in assembly language with the TI Home Computer. There are two categories of subroutine- calling techniques. One category is where the calling program and subroutine share the same set of working registers; the second category is where the calling program and the subroutine have different sets of registers. This technique of assigning a different set of working registers to the calling program and the subroutine is called "context switching." Look first at those ways of calling subroutines that don't use context switching.

16.2 Non-Context Switching Subroutine Calls

There are three instructions that can be used with subroutines without causing a context switch:

- Branch and Link (BL)

- ranch (B)

- Execute (X)

16.2.1 The Branch and Link Instruction (BL)

The Branch and Link instruction (BL) calls a subroutine. Both the subroutine and the calling program share the same set of registers.

The BL instruction has one operand that can use any of the five general addressing mod
The instruction transfers program control to the location of the instruction specified
the operand. The return address which is the address of the location immediate
following the BL instruction is placed in Register 11.

As an example, suppose there's a subroutine in a program with an entry point of SUI
(SUBR is the label attached to the first instruction to be performed in the subroutin
The following instruction calls the subroutine

```
BL   @SUBR
```

and the address of the location immediately following the BL instruction is placed
Register 11.

When the subroutine is finished and ready to return control to the calling program,
can do so by going to the address contained in Register 11. It can do this by using a Bran
instruction.

16.2.2   The Branch Instruction (B)

The Branch instruction (B) is similar to the Branch and Link instruction. The B instructi
has one operand that can use any of the five general addressing modes and it causes
transfer of program control to the location specified by the operand. The B instructi
is normally used to exit from a subroutine that's called by a Branch and Link instructio

For example, the instruction

```
B    *R11
```

transfers program control to the address in Register 11.

In fact, a Branch instruction with this particular operand is used so often in TI Hom
Computer assembly language programs that a pseudo-instruction has been given to
A pseudo-instruction is a mnemonic operation code that is used in place of anoth
operation code and assumes a specific operand. The pseudo-instruction, RT, when place
in the operation code field of a statement, results in machine code that is the same
that for B *R11.

```
RT  =  B  *R11
```

The B instruction is normally the last instruction performed by a subroutine called wi

a BL instruction, but the Branch instruction can be used anywhere in a program that you want to perform an unconditional transfer of control.

The B instruction is similar to the JMP instruction. Both instructions cause an unconditional transfer of control within a program. But the Branch instruction has a big advantage over the JMP instruction. The B instruction can transfer control anywhere, whereas the JMP instruction has a limited transfer-of-control range. Also, the B instruction has a much wider choice of addressing modes available to it, since it can use any of the five general addressing modes; the JMP instructions is limited to PC-relative addressing.

There are some advantages to the JMP instruction, however. It requires only one word of machine code where the B instruction might require two words. Also, the JMP instruction usually takes less time for the computer to perform than the B instruction.

The limited transfer-of-control range of the JMP instruction is often not a severe handicap. Persons who study such things tell us that, in a high percentage of cases, when a program transfers control to another instruction, that instruction is within a relatively short distance of the instruction transferring control.

The bottom line is this. Use a JMP instruction whenever you can for an unconditional transfer of control. If you can't reach the target with a JMP instruction, use a B instruction.

16.2.3 The Execute Instruction (X)

There's another instruction which is classified as a subroutine instruction. The Execute instruction (X) performs a one-instruction subroutine call.

The Execute instruction has one operand and can use any of the five general addressing modes. The operand is the address of an instruction. The Execute instruction performs the one instruction at that address and then returns to the location following the Execute instruction.

For example, suppose there are these instructions in a program.

```
                              :
                              :
              --------------------
+------              X   @PICKET
              --------------------        <------+
|                             .                  |
|                             .                  |
+------>     PICKET A    R0,RI        ------+
                              :
                              :
```
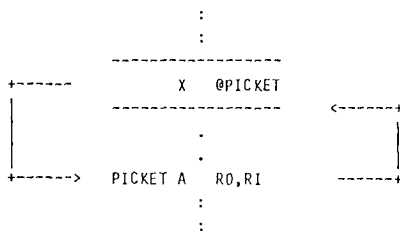
The X instruction causes the computer to perform the instruction labeled PICKET. After performing that instruction, the computer returns to the location following the X instruction.

There are a couple of things to be wary of when using the Execute instruction. For example, if the instruction performed by an Execute instruction requires more than one word of machine code, the locations immediately following the Execute instruction's machine code are used as the addresses for the data. Also, if the instruction performed by an Execute instruction is a jump instruction that results in a transfer of control, the jump is made a relative distance from the location of the Execute instruction rather than from the location of the jump instruction. When using the Execute instruction, proceed with caution.

16.3   Context-Switching Subroutine Calls

Recall that context switching is a way of calling a subroutine so that the calling program and the subroutine can have their own set of registers. There are two instructions that cause a context switch — BLWP and XOP — and one instruction that reverses a context switch — RTWP.

16.3.1   The Branch and Load Workspace Pointer Instruction (BLWP)

The BLWP instruction has one operand and can use any of the five general addressing modes. The operand specifies the address of a two-word "vector" in memory that the computer uses to perform a context switch.

A context switch vector is composed of two adjacent words in memory. The first word contains the 16-bit address of the subroutine's workspace; the second word contains the 16-bit address of the subroutine's entry point.

```
                      A Two-Word Vector

                +-----------------------------------------+
First Word      |     Address of Subroutine's Workspace    |
                +-----------------------------------------+
Second Word     |   Address of Subroutine's Entry Point    |
                +-----------------------------------------+
```
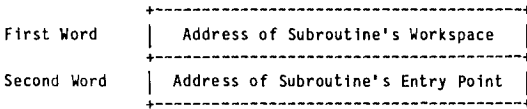
For example, suppose there are these statements in a program.

```
                :
                :
          BLWP @GIZZ
                :
                :
GIZZ      DATA SUBWSP
          DATA SUBENT
                :
                :
```

The BLWP instruction calls a subroutine with a context switch. The operand used with the BLWP instruction identifies the location of the two-word context switching vector. GIZZ is a label attached to the first word of a two-word vector. The first word is the address of the workspace used by the subroutine, SUBWSP, and the second word is the address of the subroutine's entry point, SUBENT.

When a subroutine is called as a result of a context switch, the subroutine can use its own set of working registers. The subroutine's registers are different from the set of registers used by the calling program.

When a context switch is performed, the computer automatically saves the old program context in the subroutine's workspace. Specifically, the computer saves what was in the Workspace Pointer, Program Counter, and Status Register at the moment immediately before the context switch in the bottom three registers of the subroutine's workspace. The computer saves the contents of the Workspace Pointer in Register 13, the contents of the Program Counter in Register 14, and the contents of the Status Register in Register 15 of the subroutine's workspace.

```
Saving an Old Program Context
in the Subroutine's Workspace
```

```
        +--------------------------------+
        |                  .             |
        |                  .             |
        |                  .             |
        |                  .             |
        |                  .             |
        +--------------------------------+
R13     |   Contents of Workspace Pointer |
        +--------------------------------+
R14     |   Contents of Program Counter  |
        +--------------------------------+
R15     |   Contents of Status Register  |
        +--------------------------------+
```
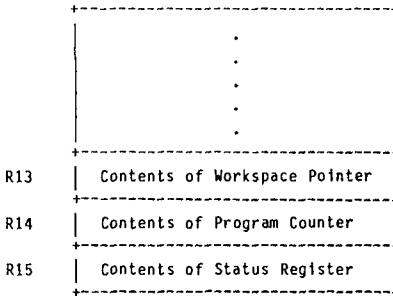
As you might expect, the contents of these internal registers are saved so that the conten
can be eventually restored to the registers. When the subroutine finishes, it can exit an
return program control to the calling program by using a Return with Workspace Point
(RTWP) instruction.

16.3.2   The Return with Workspace Pointer Instruction (RTWP)

The Return with Workspace Pointer instruction (RTWP) reverses a context switch. It
one of the few instructions that doesn't require an operand. The operation of the RTW
instruction is simple. It places the contents of Register 13 into the computer's Workspac
Pointer, moves the contents of Register 14 into the Program Counter, and moves th
contents of Register 15 into the Status Register.

An RTWP Instruction Reverses a Context Switch

```
         Computer                            Memory
    +-------------------+           +----------------------+
    |                   |           |                      |
    |  +-----------+    |           |    Workspace         |
    |  |    WP     |<-|------+      |   +--------------+    |
    |  +-----------+    |      |     |   |       .      |    |
    |                   |      |     |   |       .      |    |
    |                   |      |     |   +--------------+    |
    |  +-----------+    |      |     +---|     R13      |    |
    |  |    PC     |<-|--+   +-------|---+--------------+    |
    |  +-----------+    | |         |   |     R14      |    |
    |                   | +---------|---+--------------+    |
    |  +-----------+    |           |   |     R15      |    |
    |  |    SR     |<-|--------------|---+--------------+    |
    |  +-----------+    |           |                      |
    |                   |           |                      |
    +-------------------+           +----------------------+
```
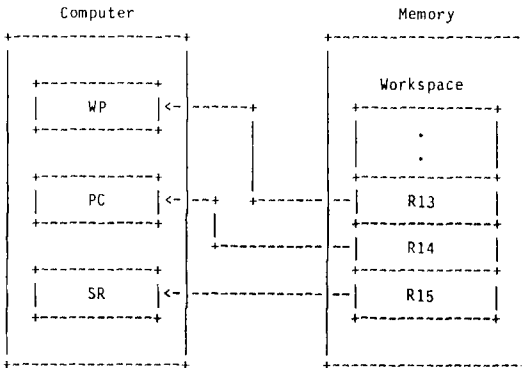
As soon as the RTWP instruction finishes, the computer uses the workspace which is addressed in the Workspace Pointer and performs the instruction which is addressed in the Program Counter. The Status Register contains whatever was in Register 15. The calling program continues with the program context that it had before calling the subroutine.

An RTWP instruction is normally the last instruction performed in a subroutine called as a result of a context switch.

16.3.3   The Extended Operation Instruction (XOP)

There's a second instruction that causes a context switch. It's the Extended Operation instruction (XOP). There are several differences, to distinguish an XOP context switch from a BLWP context switch. Like the BLWP instruction, the XOP instruction requires a two-word vector for the context switch. But with the XOP instruction, the vector must be located at a very precise location within a limited area of memory. This area of memory is the XOP vector memory space. The vector for an XOP instruction must be located in the area of memory between word addresses hexadecimal 40 and hexadecimal 7E, inclusive.

```
        XOP Vector Memory Space

          +----------------+    ------+
  >0040   |   XOP 0  WS    |          |
          +----------------+          > XOP 0 Vector
  >0042   |   XOP 0  PC    |          |
          +----------------+    ------+
  >0044   |   XOP 1  WS    |          |
          +----------------+          > XOP 1 Vector
  >0046   |   XOP 1  PC    |          |
          +----------------+    ------+
    :            :              :
    :            :              :
          +----------------+    ------+
  >007C   |   XOP 15 WS    |          |
          +----------------+          > XOP 15 Vector
  >007E   |   XOP 15 PC    |          |
          +----------------+    ------+
```
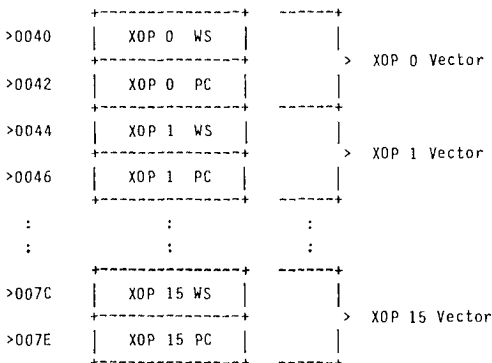
The XOP instruction requires two operands. The second operand is a C-type operand; it's a number that ranges from 0 through 15. The second operand identifies the precise location of the vector for the XOP instruction. An operand of 0 tells the computer to use

the first pair of words in the XOP vector memory space for the vector or memory locations hexadecimal 40 and 42. An operand of 1 tells the computer to use the second pair of words in that memory space for the vector or memory locations hexadecimal 44 and 46. An operand of 2 tells the computer to use the third pair of words for the vector, and so forth. An operand of 15 tells the computer to use the last pair of words for the vector or memory locations hexadecimal 7C and 7E.

This area of memory from hexadecimal 40 through hexadecimal 7E is in the TI Home Computer's ROM. The contents of these vectors can't be changed. Some of the TI Home Computers have vectors defined for XOP numbers 1 and 2; some have vectors defined only for XOP number 2.

The XOP instruction has two operands. The second operand identifies the address of information passed automatically to the subroutine. The address of the first operand is automatically placed in Register 11 of the subroutine's workspace. It's the *address value* of the operand and not the content of the address that is placed in Register 11. For example, the instruction

```
XOP   @PARAM,2
```

puts the address value of PARAM, not the contents of location PARAM, in Register 11 of the subroutine's workspace.

An RTWP instruction is used to exit a subroutine called by XOP instruction.

16.4   Context Switching and Interrupts

The two instructions, BLWP and XOP, cause a context switch. A context switch is also performed by the computer in response to an interrupt signal from an I/O device. The number of the interrupting device tells the computer where to find the two-word vector. The vectors for interrupt-initiated context switches are located in the area of memory between word addresses 0 and hexadecimal 3E, inclusive. This area of memory is in ROM and the contents of the vectors can't be changed.

16.5   Program Example

The following program is designed to illustrate different ways of calling subroutines. The program makes use of the Branch and Link instruction (BL) to call a subroutine within the program itself and it uses the Branch and Load Workspace Pointer instruction (BLWP) to call two subroutines resident within the TI Home Computer's ROM.

The program generates the Morse code for messages typed in on the computer's keyboard. When you run the program, you can press an alphabetic key and the program immediately sounds the Morse code for that character. If you press a key other than an alphabetic character, A through Z, the program sounds a reject signal. If you press the <enter< key, the program stops and returns to the title screen.

The program translates each character entered on the keyboard into Morse code and sounds the Morse code for each character. The program uses a subroutine called KSCAN to read the characters from the keyboard and uses a subroutine called SOUND to sound the Morse code characters. These subroutines are located in the TI Home Computer's ROM. A third subroutine called DELAY is included in the program itself and produces a time delay that determines how long a sound is heard. The length of the sound depends upon whether a dot or a dash is being sent.

Look at the program listing.

```
99/4 ASSEMBLER
    IN 1.2                                              PAGE 0001
    ...                   IDT  'MORSE'    TRANSLATE CHARACTERS TO MORSE CODE
    ...                *
    ...                *  EXTERNAL REFERENCES
    ....                  REF  KSCAN,SOUND
    0205              *
    000F             *  EQUATED VALUES
    ...       1194    DOTIME EQU  4500         DELAY FOR DOT TONE
    ...              *
    ...   .... 02F0    LWPI WS               INILIALIZE WORKSPACE
          ...   ..
    0010 ...       .  GETKEY SB   a)8374,a)8374   SELECT ENTIRE KEYBOARD
          ..  8374
          .... 8274
    0011 000A 04I0     BLWP aKSCAN           CHECK KEYBOARD
    000C 0003
    0012 000E D0I0     MOVB a)837C,R0        READ KEYBOARD STATUS
    0010 837C
    0013 0012 I020     COC  aKEYMSK,R0       CHECK KEYBOARD STATUS
    001. 0038'
    0014 001E 1EFE     JNE  GETKEY           JUMP IF NO KEY YET
    0015 0018 D020     MOVB a)8375,R0        KEY PRESSED, PUT ASCII CODE IN R0
    001A 8375
    0016 001C 024a     ANDI R0,)7F00         STRIP OFF PARITY BIT
    001E 7F00
    0017 0020  ...'    CB   R0,aCHARA        COMPARE CODE TO "A"
    0022  ..  '
    0018 0024 ....I    JL   NALPHA           JUMP IF NOT ALPHABETIC, MAY BE CR
    0019 002E 9800     CB   R0,aCHARZ        COMPARE CODE TO "Z"
    002S .
    0020 ..       .    JH   NOGOOD           JUMP IF NOT ALPHABETIC
    0021 ..    . .'    MOV  R0,R3            COPY CHAR CODE TO R3 (LEFT BYTE)
    0022 ..   .'       SWPB R3               PUT CHAR IN RIGHT BYTE
    0023 0030 ....     AI   R3,-65           SUBTRACT CODE FOR "A" = INDEX
    003C FFBF
    0024 0034 0A13     SLA  R3,1             MULTIPLY INDEX BY 2
    0025 0036 F1.3     MOV  aMCTABL(R3),R4   GET TABLE ENTRY IN R4
    003S ..  '
    0026 ..    .' .    MOV  R4,R3            COPY TABLE ENTRY TO R3
    0027 ..   .'       SRL  R3,8             RIGHT JUSTIFY ELEMENT COUNT
    0028 .... SENDEL LI  R10,)9100           TURN ON
          .... 9100
```

```
0029 0042 D8CA          MOVB R10,@SOUND    TONE
     0044 2200
0030 0046 04C2          CLR   R2           PUT ZERO IN R2
0031 0048 0914          SRL   R4,1         SHIFT NEXT ELEMENT CODE INTO CARRY
0032 004A 170?          JNC   DOT          JUMP IF DOT
0033 004C .'            AI    R2,DDTIME*2  ADD DELAY FOR DASH
     ?0^5  .
0034 .  .' ....   DOT   AI    R2,DOTIME    ADD DELAY FOR DOT
     ...... 1194
0035 0054 06A0          BL    @DELAY       DELAY AND END TONE
     0056 0082'
0036 0058 020?    LI    R2,DOTIME          GET INTER-ELEMENT DELAY TIME
     ..     1194
0037 ?.'  06A0          BL    @DELAY       DELAY AFTER ELEMENT
     .'   0082'
..   .' . 0603          DEC   R3           DECREMENT ELEMENT   '.T
.?' ...... 16ED          JNE   SENDEL       JUMP IF MORE ELEMI'   TO SEND
..... 0064 10CF          JMP   GETKEY         ELSE GO GET ANO'.... CHAR
```

```
99/4 ASSEMBLER
VERSION 1.2                                                    PAGE 0002
0041 006E '.'   .:'  NALPHA CB   R0,@CHARCR  IS CHAR A CARRIAGE RETURN?
     0068 ?'
0042 006A ..... JEQ   EXIT               IS SO, GO EXIT
0043 006C 020A  NOGOOD LI   R10,)F400    TURN ON
     006E F400
0044 0070 '.'         MOVB R10,@SOUND    NOISE
     0072 ....
0045 0074 0202        LI    R2,DOTIME*2  SET DELAY TIME FOR NOISE
     0076 2328
0046 0078 06A0        BL    @DELAY       DELAY AND TURN OFF NOISE
     007A 0082'
0047 007C 10C3        JMP   GETKEY       GO GET NEXT CHAR
0048 007E 0420  EXIT  BLWP @0            GO HOME
     0080 0000
0049                  *
0050 0082 0BFC  DELAY SRC   R12,15       KILL TIME
0051 0084 0602        DEC   R2           DECREMENT DELAY COUNT
0052 0086 1EFD        JNE   DELAY        JUMP IF MORE DELAY
0053 0088 020A        LI    R10,)9FFF    TURN OFF
     008A 9FFF
0054 008C D80A        MOVB R10,@SOUND    TONE
     008E 0072'
..   0090 06CA        SWPB R10           TURN OFF
.?'  0092 D80A        MOVB R10,@SOUND    NOISE
     0094 008E'
0057 0096 045B        B    *R11          RETURN TO CALLER
0058                  *
0059                  *   DATA CONSTANTS
0060                  *
0061 .?'  2000  KEYMSK DATA )2000         KEY ''
..   .?'  41    CHARA  .    'A'           CHAR    FOR "A"
..  ' .?' 5A    C     .'    'Z'           CHAR    FOR "Z"
.?' .... ØD    C           )0D            CHAR .... FOR CARRIAGE RETURN
```

```
99/4 ASSEMBLER
VERSION 1.2                                                    PAGE 0003
006E                  *
006F                  *   TRANSLATION LOOK-UP TABLE
006G                  *
0069 009E 0202  MCTABL DATA )0202         A = .__
0070 00A0 0401        DATA )0401          B = _...
0071 00A2 0405        DATA )0405          C = _._.
0072 00A4 0301        DATA )0301          D = _..
0073 00A6 0100        DATA )0100          E = .
0074 00A8 0404        DATA )0404          F = .._.
0075 00AA 0303        DATA )0303          G = __.
0076 00AC 0400        DATA )0400          H = ....
0077 00AE 0200        DATA )0200          I = ..
0078 00B0 040E        DATA )040E          J = .___
0079 00B2 0305        DATA )0305          K = _._
0080 00B4 0402        DATA )0402          L = ._..
0081 00B6 0203        DATA )0203          M = __
```

```
0082 00B8 0201    DATA )0201      N = __.
0083 00BA 0307    DATA )0307      O = ___
0084 00BC 040E    DATA )040E      P = .__.
0085 00BE 040B    DATA )040B      Q = __._
0086 00C0 0302    DATA )0302      R = .__
0087 00C2 0300    DATA )0300      S = ...
0088 00C4 0101    DATA )0101      T = _
0089 00C6 0304    DATA )0304      U = .._
0090 00C8 040B    DATA )040B      V = ..._
0091 00CA 0306    DATA )0306      W = .__
0092 00CC 0409    DATA )0409      X = _.._
0093 00CE 040D    DATA )040D      Y = _.__
0094 00D0 0403    DATA )0403      Z = __..
0095              *
0096 00D2    WS    BSS  32         WORKSPACE
0097              END
```

```
     99/4 ASSEMBLER
     VERSION 1.2                                          PAGE 0004
   ' CHARA   009A   ' CHARCR  009C   ' CHARZ   009B   ' DELAY   0082
   ' DOT     0050     DOTIME  1194   ' EXIT    007E   ' GETKEY  0004
   ' KEYMSK  0098   E KSCAN   000C   ' MCTABL  009E   ' NALPHA  00EE
   ' NOGOOD  00EC     R0      0000     R1      0001     R10     ...
     R11     000B     R12     000C     R13     000D     R14     ...
     R15     000F     R2      0002     R3      0003     R4      ...
     R5      0005     R6      0006     R7      0007     R8      ....
     R9      0009   ' SENDEL  003E   E SOUND   0094   ' WS      00D2
     0000 ERRORS
```

The statements with an asterisk in the label field; for example, statements 2 and 3, are comments. Statement 4 is a REF directive. The REF directive references symbols that are defined some place other than the program in which the REF directive appears. The operands for the REF directive are KSCAN and SOUND. These two symbols are the names of entry points into two subroutines that reside in the computer's ROM. (See Chapter 18 for further discussion of the REF directive.) KSCAN is the name of a subroutine which scans the keyboard to see if a key has been pressed. Each time the KSCAN subroutine is called, it affects a status byte at memory location <837C. If a key has been pressed since the last time the KSCAN subroutine was called, a bit in the status byte in set to one; otherwise, the bit is cleared to zero.

SOUND is the name of a subroutine that produces tones or sound with the sound processor.

In statement 7, the symbol DOTIME is equated to the value 4500. This value is a number that determines the number of times to perform a program loop in the DELAY subroutine and, effectively, determines the length of the sound produced by the sound processor.

The entry point of the program is the LWPI instruction at statement 9 which initializes the Workspace Pointer. The Subtract Bytes instruction (labeled GETKEY) zeroes out byte address <8374. This is the byte address used by the KSCAN subroutine to determine whether it should look at the whole keyboard or only a part of the keyboard. Putting a zero in the byte causes KSCAN to look at the whole keyboard.

205

The BLWP instruction at statement 10 calls the KSCAN subroutine. Upon return from the KSCAN subroutine, the MOVB instruction at statement 12 copies the status byte affected by the KSCAN subroutine into the left byte of Register 0. Then the Compare Ones Corresponding instruction at statement 13 checks the status bit in that byte. If the bit is set, it means that a key was pressed; if the bit is zero, no key was pressed. The JNE instruction at statement 14 causes a jump to GETKEY if the bit is zero and the program calls KSCAN again. The program remains in this loop, repeatedly calling KSCAN until a key is pressed. When a key is pressed, the KSCAN subroutine places the character code for that key in byte address <8375. When a key is pressed, the program falls out of the loop and the Move Byte instruction at statement 15 copies the ASCII character code into the left byte of Register 0.

The And Immediate instruction at statement 16 isolates the 7-bit ASCII character code in the left byte of Register 0. Since the program can only produce the Morse code for alphabetic characters, the program checks the character to determine if it's alphabetic. The Compare Bytes instruction at statement 17 compares the ASCII character code in Register 0 with the ASCII character code for the letter A (hexadecimal 41). If the character code in Register 0 is less than hex 41, the character is not alphabetic and the Jump if Low instruction at statement 18 causes a jump to the instruction labeled NALPHA. If the character code in Register 0 is hex 41 or greater, the Compare Bytes instruction at statement 19 compares it with the ASCII character code for the letter Z (hexadecimal 5A). If the character code in Register 0 is greater than hex 5A, the character is not alphabetic and the Jump High instruction at statement 20 causes a jump to the instruction labeled NOGOOD. If the program reaches the Move Word instruction at statement 21, the character is alphabetic and the Move Word instruction copies the character code into Register 3 (the left byte). The Swap Bytes instruction at statement 22 puts the character code into the right byte of Register 3 which right justifies the code.

At this point, it would be helpful to look at the structure of the lookup table that is used to translate the ASCII character codes of the characters into Morse code. The table begins with the DATA directive labeled MCTABL, statement 69. Each of the alphabetic characters has a one-word entry in the table, starting with the character A and ending with the character Z. Each one-word entry consists of two bytes. The left byte is the number of Morse code elements (dots and dashes) for the character. The right byte defines what those elements are and the order of the elements. In the right byte, the elements for a character appear right-to-left. A zero represents a dot and a one represents a dash. The first word in the table is labeled MCTABL and is the entry for the letter A. In Morse code, the letter A consists of two elements; a dot followed by a dash. In the table entry for A, notice the left byte contains a 2 (for two elements) and the right byte contains a 2. The binary byte value for 2 is 0000 0010. The zero in the rightmost bit position represents the dot and the one in the next position to the left represents the dash.

Take another example. Find the entry for the letter C in the table, the third word. The left byte is 4, meaning there are four elements in the Morse code. The right byte is 5 (a binary 0000 0101). The rightmost bit (a one) represents the first element, a dash; the next bit to the left (a zero) represents the second element, a dot; the next bit to the left (a one) represents the third element, a dash; and the next bit to the left (a zero) represents the fourth element, a dot.

The instructions beginning with statement 23 form an index to the lookup table. The Add Immediate instruction at statement 23 subtracts the character code for the letter A from the character code in Register 3. The result is a number in the range of 0 through 25. The Shift Left Arithmetic instruction at statement 24 multiplies the result in Register 3 by two. The result in Register 3 is a word index into the lookup table that selects a specific entry based upon the ASCII character code of the key entered.

The Move Word instruction at statement 25 uses indexed addressing to select the appropriate table entry and moves the entry to Register 4. The Move Word instruction at statement 26 copies the entry to Register 3. The Shift Right Logical instruction at statement 27 shifts the left byte of the entry into the right byte position of Register 3 and leaves zeros in the left byte of Register 3 (it right justifies the element count in Register 3). At this point, the element count is right justified in Register 3 and the bits representing the elements are in the right byte of Register 4.

The instructions beginning at statement 28 sound the Morse code. The Morse code for each character consists of a series of dot and dashes. There is a unique pattern of dots and dashes for each character. The sound for a dash is three times longer than the sound for a dot. There is an period of silence after each element equal in length to the dot time.

The Load Immediate instruction labeled SENDEL puts a hexadecimal 91 in Register 10 which is the value of a command to produce a tone with the sound processor and the Move Byte instruction at statement 29 sends the command to the sound processor. The Clear instruction at statement 30 zeros out Register 2. The Shift Right Logical instruction at statement 31 shifts an element bit out of Register 4 and the state of that bit is copied into the Carry status bit. The Jump if No Carry instruction at statement 32 causes a jump to the instruction labeled DOT if the bit is a zero. Otherwise, if the bit is one (representing a dash), the Add Immediate instruction at statement 33 adds two times the dot time to Register 2. The Add Immediate instruction at statement 34 adds one dot time to the contents of Register 2. When the program reaches statement 35, Register 2 has one of two values in it: a value equal to the dot time or a value equal to three times the dot time. The value in Register 2 determines the length of delay before turning off the sound; it determines the length of the sound.

The Branch and Link instruction at statement 35 calls the DELAY routine. The DELAY

routine, starting with statement 50, begins with a three-instruction loop that is performe a number of times, depending upon the value in Register 2. The loop takes a finite amou of time to perform and while the loop is being performed, the sound processor is makir a sound. When the loop is finished, the subroutine turns off the sound process (statements 53 through 56) and returns to the calling program (statement 57).

The program receives control from the DELAY subroutine at statement 36. The Loa Immediate instruction at statement 36 sets the delay time equal to a dot time and tl Branch and Link instruction at statement 37 calls the DELAY subroutine to wait for o dot time before sounding another element.

The program receives control again from the DELAY subroutine at statement 38. Tl Decrement instruction at statement 38 decrements the element count in Register 3. If the are more elements left to send in the character, the Jump if Not Equal instruction statement 39 causes a jump to the instruction labeled SENDEL and the next element identified and sent.

When no more elements remain to be sent, the JMP instruction at statement 40 caus a jump to the instruction labeled GETKEY and the program waits for the operator to pre another key.

The instruction labled NALPHA receives control if the ASCII character code is less th hex 41. The Compare Bytes instruction at statement 41 compares the character code that produced by the <enter< key (hex D). If the operator pressed the <enter< ko the JEQ instruction at statement 42 jumps to the instruction labeled EXIT, the Go-Hor instruction.

The instruction labeled NOGOOD receives control if the ASCII character code is great than hex 5A, the code for the letter Z. The series of instructions from statement 43 throu statement 46 sound a reject signal with the sound processor. The JMP instruction statement 47 jumps to the instruction labled GETKEY and the program waits for tl operator to press another key.

If you have the equipment, go ahead edit, assemble, load, and run the program.

This chapter illustrates the use of the Branch and Subroutine instructions. The n chapter introduces the CRU and External instructions.

Left margin stray characters (column fragments):
d
ıt
g
r

ᴅ
e
e

e
e
t
s

s
s

ı
ɔ
;
ə

r
ı
t
ə

t

Chapter 17

CRU AND EXTERNAL INSTRUCTIONS

All computers, no matter how complex or how simple, have some way of exchanging data with input and output devices. The TI Home Computer has different ways to exchange data with I/O devices. One of these ways is the Communication Register Unit (CRU). This chapter describes the CRU, the instructions that are used with the CRU, and the CRU addressing formats.

17.1 The Communication Register Unit (CRU)

The CRU is a serial I/O port that is part of the computer's central processor. Serial means that the data exchanged between the processor and I/O devices are exchanged in serial form, or one bit at a time. When performing CRU input or output operations, the processor uses a single line to bring information into the processor from an input device and another single line to send data to an output device. Each of these lines carries one bit of data at a time. The input line is called CRUIN; the output line is called CRUOUT.

Just as addresses are used to select specific memory locations to supply or receive data for an operation, addresses are also used to select the specific input or output devices that supply or receive each bit of data when a CRU I/O operation is performed.

There are five instructions in the TI Home Computer's instruction set that are used for CRU input and output operations. Three of these instructions are single-bit CRU instructions; that is, only one bit of data is sent or received with each instruction. The other two instructions are multi-bit CRU instructions; that is, they can be used to send or receive more than one bit of data. Although a multi-bit CRU instructions can cause a transfer of up to 16 bits of data, each bit is sent or received serially on the CRUIN or CRUOUT line.

Among the five CRU instructions, three instructions cause data to be sent out to a device and two instructions cause data to be brought in from a device.

Each of the five instructions can be classified as either a single-bit or a multi-bit instruction and each of them can be classified as an input or output instruction. You can

even classify them both ways at the same time, as shown below.

```
            CRU Instructions

                | Input  | Output|
   ----------------+--------+--------+
   Single-Bit   |   TB   |  SBO   |
                |        |  SBZ   |
   ----------------+--------+--------+
   Multi-Bit    |  STCR  |  LDCR  |
   ----------------+--------+--------+
```

17.2   The CRU Single-Bit Instructions (SBO, SBZ, and TB)

The three single-bit instructions receive or send only one bit of data. The TB (Test Bit) instruction receives a single bit of data from an input device. The SBO and SBZ instructions send a single bit of data to an output device. The SBO (Set Bit to One) instruction sends a one bit; the SBZ (Set Bit to Zero) instruction sends a zero bit.

Each of the single-bit CRU instructions requires only one operand. The operand is called a displacement and is a number from -128 through $+127$. The displacement is added to a base address. The sum of the displacement and the base address is the address of the device. The base address must be in Register 12.

When a CRU instruction is performed, the computer always uses a base address in Register 12. It's the programmer's job to make sure that the correct base address is in the register before the CRU instruction is performed.

Register 12, like all other working registers, holds 16 bits. When Register 12 is used to hold a base address for a CRU operation, the programmer must put the base address in bit positions 3 through 14 of Register 12. This 12-bit value in Register 12 is called the "CRU hardware base address." This address is the actual one that the computer hardware uses to address a device.

```
                  Register 12
   | 0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|15|
   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |  |  |  |  |<--- CRU Hardware Base Address --->|  |
   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```
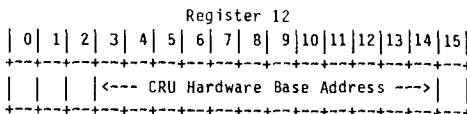
When Register 12 holds a base address for CRU operations, it doesn't matter what the bits in positions 0, 1, and 15 are. In most cases, though, these bit positions contain zeros.

When these bit positions contain zeros, the entire 16-bit value in Register 12 is called the "CRU software base address." This is address is the one that the program (software) puts in the register.

```
                  Register 12
    | 0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|15|
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    | 0| 0| 0|<--- CRU Hardware Base Address --->| 0|
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    <--------  CRU Software Base Address  -------->
```
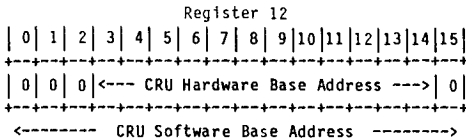
Notice that the CRU hardware base address is simply shifted one bit position to the left in Register 12. Shifting a number to the left is the same as multiplying the number times 2. This means that the CRU software base address is two times the CRU hardware base address.

```
CRU Software Base Address = 2 X CRU Hardware Base Address
```

Or, to say the same thing another way, the CRU hardware base address is one- half the CRU software base address.

```
CRU Hardware Base Address = 1/2 CRU Software Base Address
```

Register 12 actually contains two base addresses at the same time, but there is a fixed relationship between the two of them.

All CRU instructions require that a base address be established in Register 12. The Load Immediate (LI) instruction can be used to establish the base address.

For example, to establish a hardware base address of hexadecimal 40 in Register 12, you could use the following instruction.

```
LI  R12,>80
```

Or, if you don't want to go through the mental gymnastics of multiplying the CRU hardware base address times two, you can let the assembler calculate the CRU software base address for you. You can write the instruction this way.

```
LI  R12,>40*2
```

Most TI assemblers (the line-by-line assembler with the Mini Memory Module is one exception) calculate the expression >40*2 as hexadecimal 40 times 2, or hexadecimal 80.

When a single-bit CRU instruction is performed, the address of the selected device
the sum of a base address in Register 12 and the displacement which appears in
operand field of the instruction. The sum is a bit address; it's the address of a single
of data.

The displacement operand of a single-bit CRU instruction is a number that is added
the CRU hardware base address or to say it another way: *The displacement of a C*
single-bit instruction is added to the CRU hardware base address in register 12.

For example, suppose Register 12 contains hexadecimal 2A6. The software base add
is hex 2A6, and the hardware base address is hex 153. A single- bit CRU instruction v
an operand of 2 addresses bit address hex 155.

With that background, let's see how the three single-bit CRU instructions work.

17.2.1   The Set Bit to One Instruction (SBO)

The Set Bit to One instruction (SBO) sends a one bit to an output device. The instruct
requires one operand that is a displacement added to the CRU hardware base addr
in Register 12. The sum is the address that selects a specific device. The displacem
must be a number from -128 through $+127$.

For example, consider the following program segment.

```
LI    R12,>200
        .
        .
        .
SBO   12
```

The LI instruction establishes a CRU software base address of hexadecimal 200
Register 12. Consequently, the CRU hardware base address is hexadecimal 100. WI
the SBO instruction is performed, a one bit is sent to the device whose bit addres
hexadecimal 10C which is the hardware base address plus the displacement, decir
12.

17.2.2   The Set Bit to Zero Instruction (SBZ)

The Set Bit to Zero instruction (SBZ) sends a zero bit to an output device. The instruct
requires one operand which is a displacement added to the CRU hardware base addr

is
ie
it

to
U

ss
h

in Register 12. The sum is the address which selects a specific device. The displacement must be a number from −128 through +127.

For example, consider the following program segment.

```
LI    R12,>1E3*2
      .
      .
      .
SBZ   -9
```

The LI instruction establishes a CRU hardware base address of hexadecimal 1E3 in Register 12. When the SBZ instruction is performed, a single zero bit is sent to the device with bit address hexadecimal 1DA. The bit address is the sum of the hardware base address, hex 1E3, plus the displacement, decimal −9.

17.2.3   The Test Bit Instruction (TB)

n
s
t

The Test Bit Instruction (TB) is the only single-bit instruction that performs an input operation. It reads one bit of data from an input device and places the state of that bit into the Equal status bit. The instruction requires one operand which is a displacement added to the CRU hardware base address in Register 12. The sum is the address which selects a specific device. The displacement must be a number from −128 through +127.

For example, consider the following program segment.

```
LI    R12,>39C*2
      .
      .
      .
TB    23
```

n
n
is
l

The LI instruction establishes a CRU hardware base address of hexadecimal 39C in Register 12. When the TB instruction is performed, a single bit is read in from the device with bit address hexadecimal 3B3 (the sum of the hardware base address, 39C, plus the displacement, decimal 23).

n
s

The device might be a switch where a one bit means the switch is on and a zero bit means the switch is off. Following the TB instruction, the state of the switch is recorded in the Equal status bit. You can use a conditional jump instruction to determine if the switch is on or off. A JEQ instruction causes a jump if the switch is on, and a JNE instruction causes a jump if the switch is off.

In the following program segment, the JNE instruction cause a jump to the instruction labeled OFF if the switch is off; that is, where the state of the tested bit is zero.

```
TB    23
JNE   OFF
```

17.3   The CRU Multi-Bit Instructions (LDCR and STCR)

There are two CRU instructions that transfer more than one bit of data. The LDCR (Load Communication Register) instruction sends a number of bits out serially on the CRUOUT line to output devices with consecutive addresses. The STCR (Store Communication Register) instruction reads in a number of bits serially on the CRUIN line from input devices with consecutive addresses.

Each of the multi-bit CRU instructions requires two operands. The first operand can use any of the five general addressing modes and is the word or byte address for the data bits. The second operand is a count that specifies how many data bits to transfer. The count is a number that must be in the range of 0 through 15.

A non-zero count, 1 through 15, specifies directly the number of bits transferred. A count of 0 means that 16 bits are transferred.

Just as with the single-bit CRU instructions, when a multi-bit CRU instruction is performed, the computer always uses a base address in Register 12. It is the programmer's job to make sure that the correct base address is in the register before the CRU instruction is performed.

17.3.1   The Load Communication Register Instruction (LDCR)

The Load Communication Register instruction (LDCR) transfers a number of bits from memory to output devices with consecutive bit addresses. The instruction requires two operands. The first operand can use any of the five general addressing modes and is the word or byte address of the memory location containing the bits to be transferred. The second operand is a number in the range of 0 through 15 which specifies how many bits to transfer. A number of 0 means that 16 bits are transferred.

If the second operand is a number from 1 through 8, the first operand is a byte address. If the second operand is a number from 9 through 15 or is a 0, the first operand is a word address.

The base address in Register 12 determines the address of the device to which the first data bit is sent. Subsequent bits are sent to devices having the next consecutive sequential addresses.

The bit sent to the first device comes from the rightmost bit in the byte or word. The second bit sent out comes from the next bit to the left in the byte or word. Any other bits sent out come from the next bits to the left in the byte or word; that is, bits are sent out from the byte or word from right to left.

For example, consider the following program segment.

```
LI    R12,>200
        .
        .
        .
LDCR  @PUTTY,10
```

Suppose word address PUTTY contains hexadecimal 9ABC, a binary 1001 1010 1011 1100.

The first bit sent out comes from the rightmost bit in PUTTY or bit position 15. It goes to the device whose address is hexadecimal 100. Hex 100 is the CRU hardware base address in Register 12. The second bit sent out comes from bit position 14 in PUTTY and goes to the device with an address of hexadecimal 101. The third bit sent out comes from bit position 13 in PUTTY and goes to the device with an address of hexadecimal 102. Ten bits are transferred. The last bit sent out comes from bit position 6 in PUTTY and goes to the device with address hexadecimal 109.

```
           +-------------------> 10th bit to >109
           |                  .
           |                  .
           |        +-----------> 3rd bit to >102
           |        | +--------> 2nd bit to >101
           |        | | +-----> 1st bit to >100
           |        | | |
     | 0|      | 6|   |13|14|15|
     +--+ -/- +--+ -/- +--+--+--+
PUTTY | 1| . . | 1| . . | 1| 0| 0|
     +--+ -/- +--+ -/- +--+--+--+
```
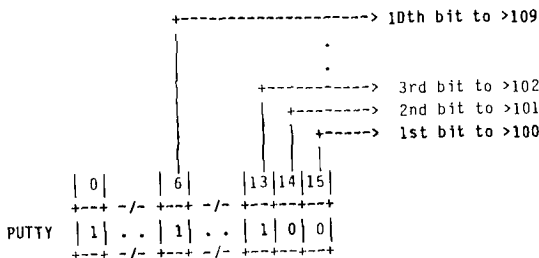
17.3.2 The Store Communication Register Instruction (STCR)

The Store Communication Register instruction (STCR) transfers a number of bits in memory from output devices with consecutive bit addresses. The instruction requires tv operands. The first operand can use any of the five general addressing modes and is t word or byte address of the memory location that receives the transferred bits. The seco operand is a number in the range of 0 through 15 that specifies the number of bits to sen A number of 0 means that 16 bits are transferred.

If the second operand is a number from 1 through 8, the first operand is a byte addre If the second operand is a number from 9 through 15 or is a 0, the first operand is a wo address.

The base address in Register 12 determines the address of the device from which tl first data bit is transferred. Subsequent bits are transferred from devices having the ne consecutive sequential addresses.

The bit transferred from the first device goes into the rightmost bit in the byte or wor The second bit goes to the next bit to the left in the byte or word. Any other bits transferr in go to the next bits to the left; that is, bits transferred in fill the byte or word from rig to left. Any unfilled bit positions in the byte or word are forced to zero.

For example, consider the following program segment.

```
LI    RI2,>38D*2
         .
         .
         .
STCR R9,5
```

Suppose Register 9 contains hexadecimal F72D (a binary 1111 0111 0010 1101) before t STCR instruction is performed.

The STCR instruction transfers five bits into Register 9. A count of 5 establishes the fi operand as a byte address. Since register direct addressing is used for the byte operatic the left byte of Register 9 receives the five data bits. The first bit transferred goes to position 7 in Register 9; the second bit goes to bit position 6; the third bit goes to bit positi 5; and the fifth, and last bit, transferred in goes to bit position 3 in Register 9. Since positions 0 through 2 in the left byte are unfilled, these bit positions are forced to ze The right byte of the register is unaffected.

The first bit transferred is determined by the base address in Register 12. Since Regis 12 contains a hardware base address of hexadecimal 38D, the first bit comes from t

device with an address of hex 38D. The second bit comes from the device with an address of hex 38E, and so forth. The fifth bit comes from the device with an address of hex 391.

Suppose these devices are individual switches where a one bit means a switch is on and a zero bit means the switch is off. Further suppose that the switches at the following addresses are on or off as indicated below.

```
Switch Address          State
    >38D                 On
    >38E                 Off
    >38F                 Off
    >390                 On
    >391                 On
```

After the STCR instruction is performed, Register 9 contains hexadecimal 192D.

```
         +-------------------< 5th bit from >391
         | +-----------------< 4th bit from >390
         | | +---------------< 3rd bit from >38F
         | | | +-------------< 2nd bit from >38E
         | | | | +-----------< 1st bit from >38D
         | | | | | |
    | 0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|15|
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
R9  | 0| 0| 0| 1| 1| 0| 01 1| 0| 0| 1| 0| 1| 1| 0| 1|
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    < Zeros>                  <---- Unaffected ---->
```
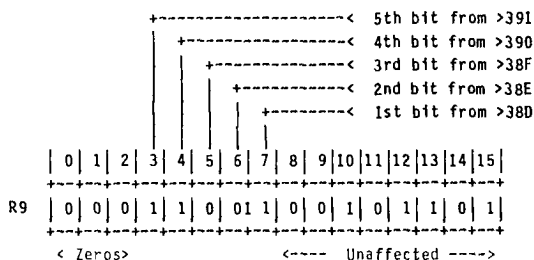
17.4   The External Instructions (IDLE, RSET, LREX, CKON, CKOF)

There are five instructions in the TI Home Computer's instruction set classified as external instructions. These instructions are reserved for very special functions within the computer and, generally, should not be used in your programs.

Each of these instructions causes the central processor to generate specific signals that can trigger functions defined by other electronic components in the computer. The inappropriate use of these instructions can cause unpredictable results. None of these five instructions require an operand.

17.4.1   The Idle Instruction (IDLE)

The Idle instruction (IDLE) places the central processor in the idle state. When an IDLE instruction is performed, the computer stops performing any other instructions and simply

performs the IDLE instruction over and over again. The computer remains in the idle state until an interrupt signal occurs from some device.

17.4.2 The Reset Instruction (RSET)

The Reset instruction (RSET) puts zeros into the interrupt mask which is the rightmost four bits in the Status Register. This is a way of preventing all but the most important interrupt signals from causing an interrupt.

17.4.3 The Other External Instructions (LREX, CKON, and CKOF)

The other External instructions (LREX, CKON, and CKOF) do not directly affect the operation of the central processor.

Program Example

In the last chapter, the example program accepted an alphabetic key pressed on the keyboard and translated and sounded the Morse code for that that character. This program simulates a telegraph key. Whenever you press the <function> key, a sound is made. When you release the key, the sound stops. The program uses one of the CRU instructions to determine when a specific key is pressed and when it's released.

Look at the listing for the program.

```
      99/4 ASSEMBLER
      VERSION 1.2                                                    PAGE 0001
      0001                          IDT   'SOUND'     MAKE A SOUND BY PRESSING A KEY
      0002                    *
      0003                    *  EXTERNAL  REFERENCES
      0004                          REF   SOUND        REFERENCE SOUND PORT
      0005                    *
      0006                    *  EQUATED  VALUES
      0007      2000    DEBNCE EQU   >2000         DELAY TIME TO WAIT ON BOUNCING KEY
      0008      0007    FUNCTN EQU   7             DISPLACEMENT FOR FUNCTION KEY
      0009                    *
      0010 0000 02E0    LWPI  WS           INITIALIZE WORKSPACE POINTER
           0002 0028'
      0011 0004 04CC    CLR   R12          POINT TO KEYBOARD
      0012 0006 1F07 CHEKEY TB    FUNCTN     TEST KEY
      0013 0008 13FE    JEQ   $-2          WAIT UNTIL IT'S DOWN
      0014 000A 020A    LI    R10,>9100    TURN ON
           000C 9100
      0015 000E D820    MOVB  R10,@SOUND   TONE
           0010 0000
      0016 0012 0202    LI    R2,DEBNCE    INIT R2 TO DEBOUNCE DELAY COUNT
           0014 2000
      0017 0016 0602    DEC   R2           WAIT FOR KEY
      0018 0018 16FE    JNE   $-2            TO STOP BOUNCING
      0019 001A 1F07    TB    FUNCTN       TEST KEY
      0020 001C 16FE    JNE   $-2          WAIT UNTIL IT'S UP
```

```
0021  001E  020A      LI    R10,)9F00    TURN OFF
      0020  9F00
0022  0022  D80A      MOVB  R10,@SOUND    TONE
      0024  0010'
..'   0026  10EF      JMP   CHEKEY    GO WAIT FOR KEY TO BE PRESSED AGAIN
..'.              *
....  0028    WS     BSS   32          WORKSPACE
002E              END
```

```
99/4 ASSEMBLER
VERSION 1.2                                              PAGE 0002
'  CHEKEY   000E    DEBNCE   2000    FUNCTN   0007    R0      0000
   R1       0001    R10      000A    R11      000B    R12     000C
   R13      000D    R14      000E    R15      000F    R2      0002
   R3       0003    R4       0004    R5       0005    R6      0006
   R7       0007    R8       0008    R9       0009  E SOUND   0024
 ' WS       002E
 0000 ERRORS
```

The REF directive (statement 4) references the symbol SOUND, a byte address that is used to give commands to the sound processor.

There are two EQUated values. DEBNCE is equated to hexadecimal 2000 in statement 7. DEBNCE is a loop counter used to create a program controlled timing loop to wait for the key to stop bouncing (making intermittent contact) when the operator presses it. The second EQUated value is FUNCTN. FUNCTN is equated to the value 7 in statement 8. This value is the CRU I/O address of the <function> key on the keyboard. This is the key the operator presses to make a sound.

The entry point of the program is the LWPI instruction at statement 10. Statement 11 establishes a base address of zero in Register 12 for CRU addressing.

The TB instruction labeled CHEKEY tests the state of the <function> key. The state of the key is a logic one if it is not pressed and a logic zero if it is pressed. If the key is not pressed, the JEQ instruction at statement 13 jumps back to the TB instruction to test the key again. The program remains in this two-instruction loop, repeatedly testing the key until the key is pressed.

When the key is pressed, the program falls out of the loop and the two instructions at statements 14 and 15 command the sound processor to make a sound. The program then initializes the contents of Register 2 to DEBNCE (statement 16) and performs a two-instruction programmed control timing loop (statements 17 and 18). The purpose of this loop is to wait for the key to stop bouncing before checking for the release of the key.

In statement 19, the program uses another TB instruction to determine when the key is released. The two-instruction loop composed of statements 19 and 20 is performed repeatedly until the key is released. At that time, the program falls out of the loop and the two instructions at statements 21 and 22 command the sound processor to be silent.

Then the JMP instruction at statement 23 jumps back to the CHEKEY instruction to wa for the key to be pressed again. The program is composed as an infinite loop. It has exit point.

Edit, assemble, load, and run the program.

This chapter describes the CRU and External instructions. Now all of the instructio in the instruction set have been described. The remaining chapters discuss other assemb language concepts and describe the structure of the TI Home Computer's machine cod

Chapter 18

OTHER ASSEMBLER LANGUAGE CONCEPTS

This chapter discusses a variety of topics relevant to assembly language programming:

- operand expressions

- program relocation

- assembler directives

- assembler errors

- a comparison of some of the different utility packages for running and developing assembly language programs

18.1   Operand Expressions

Expressions are used in the operand field of a statement. An expression can include one or more constants or symbols and arithmetic operators. The most common arithmetic operators are these.

Arithmetic Operator	Meaning
+	positive or addition.
—	minus or substraction.
*	multiplication.
/	division.

As an example, in the statement

```
ORANGE MOV  @PEEL+2,R6
```

PEEL + 2 is an expression. The instruction moves the contents of the location with the address PEEL + 2 to register 6. If PEEL has an address value of hexadecimal B4A2, then the address value of PEEL + 2 is hexadecimal B4A4.

Expressions can be simple or they can be complex. An expression can consist of simply a constant. The following statements are examples of this.

```
SBO  9
TB   -3
```

Expressions may include several constants and symbols with more than one arithmetic operator. For example, the statement

```
CLR  @QUARK+14/6*2
```

clears the contents of the word with the address that is determined by the expression QUARK+14/6*2. When evaluating an expression, most assemblers perform the arithmetic operations from left to right. In this example, suppose the address value of QUARK is hexadecimal A6B4 (decimal 42676). The assembler evaluates the expression left to right like this.

```
QUARK       =    42676
QUARK+14    =    42690
QUARK+14/6=      7115
QUARK+14/6*2   14230      (hexadecimal 3796)
```

The instruction clears the contents of location hexadecimal 3796. This example is extreme. Most likely, you won't encounter expressions that complex.

18.2 Relocation

The Assembler included with the Editor/Assembler package is a relocatable assembler. This means that it can assemble a source program and construct an object program so the object code can be loaded at different locations in memory. The object program requires a relocating loader to be able to load the object code into different locations.

There may be some statements, though, that you don't want to be relocatable. For example, you might want the constant ten in a specific, physical memory location with an address that remains the same. Sections of a program may be relocatable and other sections non-relocatable (absolute).

During assembly, the Assembler uses a location counter to assign location values to program statements. As each statement is assembled, the location counter is incremented by the length of the assembled item.

The $ symbol is used to represent the current value of the location counter. When the $ symbol is used in an expression in the operand field of a statement, you can read the symbol as "this location." For example, the statement

```
JMP   $+8
```

can be read as jump to "this location" plus 8.

18.3   Assembler Directives

An assembler directive gives directions to the assembler during the assembly process. The previous chapters have introduced a few assembler directives such as BSS, DATA, and END. This section describes several of the more commonly used directives.

18.3.1   Directives that Define the Contents of Memory

Some directives define the contents of memory. These directives include DATA, BYTE, and TEXT.

18.3.1.1   The DATA Directive

The DATA directive defines a *word* of memory with a specific value in it. For example, the statement

```
DATA   10
```

defines a word of memory that contains ten.

You can assign a name to the value with the DATA directive. For example, the statement

```
DECA   DATA 10
```

assigns the name DECA to the constant 10.

223

You can use symbols in expressions in the operand field of a DATA directive.
example, the statement

```
DONUT   DATA   HOOPLA-6
```

assigns the name DONUT to a memory word containing the value of HOOPLA m
6.

You can use more than one operand in the operand field of a DATA directive.
example, the statement

```
CTABLE   DATA   5,4,3,2,1,0,-1,-2,-3,-4,-5
```

defines a table of eleven consecutive words. The first word, containing a constant c
is named CTABLE. The second word, containing a constant of 4, could be addressed v
the expression CTABLE+2.

18.3.1.2   The BYTE Directive

The BYTE directive is similar to the DATA directive except it defines a *byte* of mem
with a specific value it, rather than a word.

For example, the statement

```
CRUZO   BYTE 8,-128,>40,0
```

defines the content of four consecutive bytes in memory. The first byte contains the va
8; the second byte contains the value minus 128; the third byte contains the va
hexadecimal 40; and the fourth byte contains the value 0. The name of the first byt
CRUZO.

If Register 9 contains the value 2 and Register 4 contains hexadecimal 0A7C, then
statement

```
AB   @CRUZO(R9),R4
```

adds the two byte values hexadecimal 40 and 0A.

.or

18.3.1.3 The TEXT Directive

The TEXT directive causes the assembler to put the ASCII character codes for specific characters into consecutive bytes of memory. The characters whose character codes are assembled are in the operand field surrounded by single quote marks (apostrophes).

ıus

For example, the statement

```
MESG   TEXT  'GERONIMO!'
```

places the ASCII character codes for the characters G, E, R,O, N, I, M, O, and ! in consecutive bytes of memory.

The name MESG is assigned to the address of the first character.

Suppose the assembler's location counter contains hexadecimal 1C6 when this TEXT directive is encountered.

The Assembler places the following hexadecimal values into the following words of memory.

Word Address	Contents
>01C6	>4745
>01C8	>524F
>01CA	>4E49
>01CC	>4D4F
>01CE	>21??

Immediately after assembling the TEXT directive, the location counter contains hexadecimal 01CF and the contents of byte address hexadecimal 01CF is not yet defined.

The TEXT directive is often used to compose a message that can be displayed on a screen or printed. Sometimes a message may have several lines of text. To end a line of text and begin another, you can embed the ASCII character codes for a carriage return and line feed within the text string.

For example, the statements

```
PROMPT TEXT  'WHEN READY'
       BYTE  >0D,>0A
       TEXT  'PRESS ANY KEY'
```

causes the characters WHEN READY to appear on one line and the characters PRESS ANY KEY to appear on another line. Hexadecimal 0D is the ASCII character code for a carriage return and hexadecimal 0A is the ASCII character code for a line feed. There are some video displays, printers, and other similar devices which may not require the line feed character in order to put characters on another line.

18.3.2 The EVEN Directive

Sometimes, especially following a BYTE or TEXT directive, the location counter value is an odd number. The EVEN directive forces the location counter value to the next larger even number so that the object code assembled after the EVEN directive will begin on a word boundary.

For example, suppose the statement

```
BYTE  -88,12,-1
```

left the location counter with a value of hexadecimal 13D (an odd number). An EVEN directive following the BYTE directive

```
BYTE  -88,12,-1
EVEN
```

forces the location counter value to hexadecimal 13E, the next larger even value. If the location counter value is already an even number, the EVEN directive doesn't change it.

18.3.3 Directives that Reserve But Do Not Define the Contents of Memory

Two directives that reserve memory space for use in a program but don't define the contents of those memory locations are BSS and BES.

18.3.3.1 The Block Starting with Symbol Directive (BSS)

The Block Starting with Symbol Directive (BSS) reserves one or more bytes of memory but doesn't define the values those bytes contain. The operand of the BSS directive specifies how many bytes to reserve. For example, the statement

```
BSS 20
```

reserves 20 bytes (10 words) of memory.

You can use a label with the BSS directive. A label is the name given to the first location of the area of memory.

For example, the statement

```
BUFFER BSS 80
```

reserves 80 bytes (40 words) of memory and BUFFER is the name assigned to the first location.

The BSS directive is often used to reserve an area of memory for a program's workspace. For example, the statement

```
WSP     BSS   32
```

reserves a 32-byte (16-word) area of memory and assigns the name WSP to the first location. The statement

```
LWPI   WSP
```

can be used to load the Workspace Pointer with the address value of WSP.

18.3.3.2   The Block Ending with Symbol Directive (BES)

The Block Ending with Symbol directive (BES), like the BSS directive, also reserves a block of memory. The BES directive, though, assigns to the label the value of the address immediately following the block of memory.

For example, the statement

```
STACK  BES  >100
```

reserves a 256-byte (hexadecimal 100) area of memory. If the value of the location counter is hexadecimal 10E when the BES directive is encountered, the location counter value is advanced to hexadecimal 20E and the label STACK is assigned the address value hexadecimal 20E.

227

18.3.4   Directives that Initialize the Location Counter

Two directives that initialize the assembler's location counter are RORG and AOR

18.3.4.1   The Relocatable Origin Directive (RORG)

The Relocatable Origin Directive (RORG) causes the section of the program that foll
to be relocatable. It permits the object code for that section of the program to be loa
into different physical memory locations. With the Editor/ Assembler packa
assembler, the object code is relocatable by default and an RORG directive isn't nee
unless you want it.

If an operand is used with the RORG directive, the location counter is set to that va
If no operand is used, the location counter is set to zero or to the last value it had w
assembling the last relocatable section of the program.

For example, the statement

```
RORG
```

specifies that the following section of the program is relocatable and, if this is the
relocatable section of the program, the location counter is set to zero.

The statement

```
RORG  $+16
```

advances the location counter by 16 from its current value.

18.3.4.2   The Absolute Origin Directive (AORG)

The Absolute Origin directive (AORG) causes the section of the program following
be non-relocatable. It causes the object code for that section of the program to be loa
into specific and fixed (absolute) memory locations.

If an operand is used with the AORG directive, the location counter is set to that va
For example, the statements

```
AORG >FFFC
DATA LOADWP
DATA LDADPC
```

cause the two word values LOADWP and LOADPC to occupy the fixed locations hexadecimal FFFC and FFFE.

18.3.5   The Equate Directive (EQU)

The Equate directive (EQU) assigns (or equates) a name to a value without reserving a word in the program's memory space. For example, the statement

```
TWELVE EQU   12
```

assigns the name TWELVE to the constant 12. TWELVE can then be used anywhere in a statement where the constant 12 can be used. For example, the statement

```
LI  R7,TWELVE
```

is the equivalent to the statement

```
LI  R7,12
```

As another example, the statement

```
NEG  @BLADE(TWELVE)
```

is equivalent to the statement

```
NEG  @BLADE(12)
```

which is equivalent to the statement

```
NEG  @BLADE(R12)
```

You can even use the equated name with an assembler directive. For example, the statement

```
DATA TWELVE
```

reserves a word of memory with a content of 12.

18.3.6   The Book End Directives (IDT and END)

There are two directives that you can think of as bookends for a program. These directives are the IDT and END directives.

18.3.6.1   The End Directive (END)

The End directive (END) should be in the last statement of a program. It tells the Assembler to stop assembling. You can use a label with the END directive. A label is simply assigned the value of the location counter when the directive is encountered. You can also use an operand with the END directive. The operand lets you define the entry point of the program. That is, it specifies the instruction to be performed first when the program runs.

For example, the statement

```
END  OPEN
```

identifies OPEN as the name of the instruction to be performed first when the program runs. OPEN should be a label attached to that instruction.

Note

With the Editor/Assembler package's Loader, using an operand with an END directive causes the program to start running as soon as it's loaded.

18.3.6.2   The Identification Directive (IDT)

The Identification directive (IDT) is optional. However, if it's used, it should be the first statement in a program. It assigns a name to the program. The name of the program is specified in the operand field. The name can have up to eight characters and the characters are surrounded by single quote marks (apostrophes). For example, the statement

```
IDT  'MODULE X'
```

assigns the name MODULE X to a program.

18.3.7   The External Linkage Directives (DEF and REF)

When creating lengthy programs, it is often convenient to divide the program into separately assembled modules and have a linking loader load the object programs into memory together. The DEF and REF directives help link together separately assembled programs.

18.3.7.1 The External Definition Directive (DEF)

The External Definition directive (DEF) identifies those symbols that are defined in a program and that can be referenced by other programs. To be used by other programs, a symbol must appear in the label field of a statement in the program and also be included in the operand of the DEF statement

For example, the statement

```
DEF   OPEN,TWELVE
```

identifies the symbols OPEN and TWELVE as symbols that can be referenced by other programs. These symbols must be defined by the program that includes the DEF directive. To be defined, a symbol must appear in the label field of a statement.

18.3.7.2 The External Reference Directive (REF)

The External Reference directive (REF) identifies those symbols which are used in a program and defined in another program. These symbols are included in the operand field of the REF statement. For example, the statement

```
REF   PIGARN,CRAZY8,CUPID
```

identifies the symbols PIGARN, CRAZY8, and CUPID as symbols used in the program and defined in another program.

As an example, the statement

```
MOV   *R6+,@PIGARN
```

moves a word to memory location PIGARN that is defined in a different program from this one.

When programs are assembled separately and reference symbols between them, the object programs must be loaded and linked together by a linking loader. Before you run a program that references a symbol in another program, the program that defines the symbol must be loaded. The loader included with the Editor/Assembler package is capable of loading and linking programs together.

18.4 Assembler Errors

Some things can go wrong when you assemble a program. Whenever the assembler fin
a statement that it can't assemble or encounters a situation that it can't handle, it gi
you an error message. These error conditions are classified as fatal or nonfatal.

Fatal errors are grim. The assembler just can't go on. Fortunately, fatal errors do
happen often. Fatal errors occur when the assembler can't read or write to a disk for so
reason, or the assembler runs out of memory. If the assembler encounters a fa
condition, it displays an error message on the screen and stops the assembly proces

Nonfatal error conditions don't stop the assembly process. They nearly always result fr
writing a statement incorrectly. These are the error conditions that you'll encounter m
often. Write enough programs and you'll get quite a collection of them. When
assembler encounters a nonfatal error, it displays the statement in error and
appropriate admonishment on the screen. And to further add to your embarasseme
it also prints the error message right on the listing.

Even a program that's not up to the assembler's standards has some value. It can se
as a bad example. Here's the listing of a program which has, perhaps, a high enou
percentage of errors to qualify for a world record.

```
     9974 ASSEMBLER
    VERSION 1.2                                              PAGE 0001
      0001                  IDT  'NAME TOO LONG'   SHOULD BE 8 OR LESS CHARACTERS
    *****  OUT OF RANGE - 0001
      0002 0000 1000   A COMMENT LINE SHOULD BEGIN WITH AN ASTERICK (*).
    *****  SYMBOL TRUNCATION -   0002
    *****   ··.  IN MNEMONIC - 0002
      0003 ···· ·. ·       LWPI WS         OPERANDS MUST MATCH DEFINED SYMBOLS
           ··· ·.··.·
    *****  UNDEFINED SYMBOL - 0003
      0004 0006 C040   STARTUP MOV R0.R1    LABELS MUST BE 6 CHARS OR LESS
    *****  SYMBOL TRUNCATION -   0004
      0005 0008 1000           MOVE R1.R2   OPCODES MUST BE SPELLED RIGHT
    *****  INVALID MNEMONIC - 0005
      0006 000A 1000    LABEL MOV  R2.R3    LABELS MUST BEGIN IN FIRST COLUMN
    *****  INVALID MNEMONIC - 0006
      0007 000C 1000   MOV    R4.R5         OPCODES CAN'T BEGIN IN FIRST COLUMN
    *****  INVALID MNEMONIC - 0007
      0008 000E 1095      JMP  $+300        JUMP TARGETS MUST BE IN RANGE
    *****  OUT OF RANGE - 0008
      0009 0010 0206      LI   R6.@BUFF     DON'T USE @ WITH IMMEDIATE OPERANDS
           0012 0000
    *****  SYNTAX ERROR - 0009
      0010 0014 2820      XOR  @MASK.+R7    XOR 2ND OPERAND MUST BE REG DIRECT
           0016 001B'
    *****  SYNTAX ERROR - 0010
      0011 ··         BUFF   .       FORTY  OPERAND SYMBOLS MUST BE DEFINED
    *****   ·.    ·    ··:·   · ··.:
      0012 ··   ··:·  ··.    BUFF/5C6F      SPACES MUST BE BETWEEN OPERANDS
    *****  SYNTAX ERROR  .012
      0013 001A 1000   WSP  BSS             BSS DIRECTIVE REQUIRES AN OPERAND
    *****   ·   INED SYMBOL - ·· ·
      0014 ···· 1000   THE LAST     EMENT SHOULD CONTAIN AN END DIRECTIVE
```

```
*****  INVALID MNEMONIC - 0014
  0015               END
    THE FOLLOWING SYMBOLS ARE UNDEFINED:
COMMEN
WS
MOVE
| ORFI
|
L̲̲̲
LAST
*****  END ASSUMED - 0015


  99/4 ASSEMBLER
VERSION 1.2                                             PAGE 0002
   ' A      0000    U BSS     0000    ' BUFF    001B   U COMMEN  0000
  U FORTY   0000    U LABEL   0000    U LAST    0000    ' MASK    001B
   ' MOV    ...     U MOVE    0000      R0      0000      R1      0001
     R10    ...       R11     000B      R12     000F      R13     000D
     R14    ...       R15     000F      R2      ...       R3      0003
     R4     ...       R5      0005      R6      ...       R7      0007
     R8     0008      R9      0009    ' STARTU  ...     ' THE     001A
  U WS      0000    ' WSP     001A
  0016 ERRORS
```

At the bottom of the listing, notice the message ***** END ASSUMED – 0015. It means the source program does not contain an END directive. Statement number 15 was added by the assembler when it encountered the end of the source program disk file.

18.5   Comparison of Utility Packages

A variety of assemblers and associated utility programs available from Texas Instruments and, perhaps, from other sources that you might use to assemble a program. The specific assembler you choose depends upon what kind of equipment you have and how much money you want to spend.

This book illustrates the use of the assembler in the Editor/Assembler package as an example of the features common to most assemblers. Other assemblers may have some more features, fewer features, or different features.

Here's a brief comparison of some of the assemblers and associated utility programs that are available.

18.5.1   Editor/Assembler Package

The Editor/Assembler package includes four utility programs:

• an editor for composing source programs

233

- a relocatable assembler (one which can produce relocatable object code)

- a relocating linking loader (one which can load and link several individual object programs

- an extensive debugger

The package comes with a comprehensive manual that describes the detailed architecture of the TI Home Computer, including the programmable components that directly control graphics, sound, and speech.

To run the Editor/Assembler package requires at least one disk drive and a Memory Expansion unit in addition to the computer console and a display.

18.5.2 The Mini Memory Module

The Mini Memory Module is a Command Module that contains 4K bytes of battery-backed CPU RAM and ROM-resident programs. These programs allow you to tie TI BASIC programs and assembly language programs together. The Mini Memory Module comes with an assembler on a cassette tape and a debugger that's resident in the Mini Memory Module's ROM.

You can load the assembler from a cassette into the RAM and use it to assemble programs. The assembler is a line-by-line assembler which means that each statement is assembled and the resulting object code stored in memory as each statement is typed.

This is different from using the Editor/Assembler package where you use an editor to type in all the source statements before they are assembled. With the Mini Memory Module, there's no editor. You simply type in the statements on the keyboard and they're assembled as you type them. If you need to reassemble a program again, you type it all again.

The object code created by the line-by-line assembler is stored into the CPU RAM in the Mini Memory Module. You can't assemble very large programs, though. The Mini Memory Module has only 4K bytes of CPU RAM and the assembler needs about half of that 4K for itself. (You can assemble large programs by adding a Memory Expansion unit.)

The line-by-line assembler is a nonrelocatable assembler and accepts only a few directives.

There's no loader. The object code is stored directly in memory as the source statements are assembled. A debugger is included in the Mini Memory Module's ROM and has a minimal number of commands to help you debug a program.

The big advantage of the Mini Memory Module is that it gives you assembly language capability without having to purchase a disk drive or a Memory Expansion unit.

18.5.3   The p-System Assembler and Linker

The UCSD p-System gives you the ability to develop and run programs written in other high-level languages besides BASIC. In addition, an assembler is available for use with the UCSD p-System that lets you develop assembly language programs that can be used with these other languages.

The assembler is relocatable and includes a larger number of directives than the one with the Editor/Assembler package. The assembler is also a macro assembler, which means you can create your own macro statements. When the assembler encounters a macro statement, it generates a whole series of individual source statements and assembles the source statements as if they were included directly in the source program.

A linker is also available which can link assembly language object programs with high-level language programs.

The UCSD p-System requires a Memory Expansion unit, a p-System peripheral, and at least one disk drive in addition to the computer console and a display.

18.5.4   Other Assemblers

Other assemblers exists that could conceivably be used to develop assembly language programs for the TI Home Computer. Texas Instruments provides other assemblers intended for those individuals and organizations that develop commercial programs. These assemblers are designed to run on minicomputers or bigger computers.

There are also assemblers that run on lower-cost equipment. As one example, Texas Instruments manufactures a single-board microcomputer called the University Module. Included in the ROM on the board is a line-by-line assembler similar to the line-by-line assembler that comes with the Mini Memory Module.

These assemblers all recognize the same instruction mnemonic operation codes, expect basically the same syntax, and accept all or many of the same assembler directives discussed in this book.

Chapter 19

MACHINE CODE FORMATS

This chapter explores the structure of the TI Home Computer's machine code. A knowledge of machine code can give you an edge when you need to debug a program.

19.1 Relationship of Machine Code to Assembly Language

Machine language is what the computer needs in order to understand what to do. Assembly language is a domesticated form of machine language. It gives you the advantage of directly controlling a computer without having to deal with the ones and zeros of machine code.

Although you normally would not choose to write programs in machine language and you don't have to when there's an an assembler available, it's often helpful to understand the format of machine code. When debugging a program, for example, you might need to change an instruction. Rather than take the time to leave the debugger, use an editor to change the statement, re-assemble, and reload the object program, it's faster to use the debugger to directly change the machine code. Also, sometimes, you may be using the debugger to examine the machine code of a program and you want to know the kind of instruction you're looking at. In these situations, a knowledge of how machine code is structured is helpful.

19.2 Determining the Number of Words of Machine Code

All machine language instructions of the TI Home Computer require an even number of bytes of machine code. All instructions require either two, four, or six bytes of machine code or one, two, or three words. You can easily determine the number of words of machine code that an instruction requires by examining the assembly language form of the instruction.

All instructions require at least one word of machine code. A few instructions always require two words of machine code. Some instructions may require two, or even three, words of machine code, depending upon the addressing modes used by the operands.

The only instructions that always require two words of machine code are those with an immediate operand. You can identify these instructions easily because the last letter of the mnemonic operation code is the letter, I; for example, LI, AI, ORI, etc.

The instructions that have operands that can use the general addressing modes may require two or three words of machine code. With these instructions, an additional word of machine code is required for each operand that uses either symbolic addressing or indexed addressing. These are the operands that require an at sign (@) with them. An instruction that has only one operand that can use the general addressing modes may require either one or two words of machine code. It depends upon the specific addressing mode used by that operand. An instruction which has two operands where both of the operands can use the general addressing modes may require one, two, or three words of machine code.

An additional word of machine code is required for each valid at sign (@) in the instruction. For example, the instruction

```
MOV   *R7+,@ARGIE
```

requires one additional word of machine code for a total of two words.

The instruction

```
SZC   @WILMA,@ARGIE
```

requires a total of three words of machine code (one word plus two more for the two at signs).

With this background, let's see how the machine code is structured.

19.3   Machine Code Fields

Just as assembly language statements contain different fields of information, machine code also has different fields. Within each instruction's machine code, there is a field of bits with a unique and fixed pattern that identifies the instruction's operation. For example, there's a unique pattern of bits that identifies a MOV instruction; there's a unique pattern that identifies a CLR instruction; and there's a unique pattern that identifies each of the other instructions.

There are some fields within the machine code with contents that vary depending upon the operands used with the instruction.

You can determine which bits in the machine code of an instruction are fixed and which ones are variable by examining the instruction summary for the instruction. Appendix A contains the instruction summaries which are arranged in alphabetical order according to the instructions' mnemonic operation codes. For example, turn to the instruction summary for the MOV instruction and look at the last item in the instruction summary, titled "Machine Code." The line labeled "Binary" contains the state of the specific bits in the machine code that are fixed. With the MOV instruction, the first four bits in the first word are fixed. These four bits are a binary 1100. In machine code, a binary 1100 in the first four bits of an instruction means "MOV."

Look at the instruction summary for the A instruction. With the A instruction, the first four bits are also fixed. These four bits are a binary 1010. In machine code, a binary 1010 in the first four bits of an instruction means "A."

Often, more than four bits are fixed in the machine code. After all, if only four bits were used to determine operation codes, the computer could only have 16 instructions.

Turn to the instruction summary for the COC instruction and look at the machine code format. The first six bits are fixed. They are a binary 001000.

Turn to the instruction summary for the SRL instruction and look at the machine code format. The first eight bits are fixed. They are a binary 00001001.

Turn to the STWP instruction summary and look at the machine code format. The first twelve bits are fixed. They are a binary 000000101010.

There are a few instructions in which all the machine code bits are fixed. For example, turn to the RTWP instruction summary and look at the machine code format. All sixteen bits in the one word of machine code are fixed. In machine code, a binary 0000 0011 1000 0000 (or hex 0380) means "RTWP."

There are no variable bits in the machine code of the RTWP instruction because the instruction has no operands. Variable bits appear in the machine code of those instructions that have operands. The contents of the variable fields depend upon the specific operands used with the instruction.

In the machine code description of an instruction summary the line titled "Hex" contains the hexadecimal digits that correspond to the fixed bits of the machine code. Only those hex digits are shown for which the corresponding machine code nibble is completely defined. A hex digit is not shown for any nibble which contains bits that vary depending upon the operands. Hex digits are shown only for the bits in the *first* word of machine

code because the bits in any other word of machine code always vary depending u|
the operands.

Let's explore these variable fields more closely.

19.4   The R Field

Turn to the instruction summary for the STWP instruction and look at the machine c|
description. The first twelve bits are fixed; the last four bits are variable. The last f
bits make up an R field which is indicated by the R in that field. The STWP instruc|
requires only one operand and that operand must be a register (the operand uses c
register direct addressing). The R field in the machine code holds the register num
of the operand. For example, the instruction

        STWP   R6

results in a binary 0110 in the R field.

The instruction

        STWP R9

results in a binary 1001 in the R field. The entire word of machine code for the instruc|

        STWP R9

is a binary 0000 0010 1010 1001 (or hexadecimal 02A9).

An R field in a machine code word is always four bits big and holds the number (
register.

19.5   The C Field

Turn to the instruction summary for the SRL instruction and look at the machine c|
description. The first eight bits are fixed and the last eight bits are variable. The last e
bits include a four-bit R field and a four-bit C field. The SRL instruction requires
operands: a register number (R) and a count (C). The R field holds the register num
and the C field holds the count.

For example, the instruction

```
SRL  R10,2
```

results in an R field of binary 1010 and a C field of binary 0010. The complete word of machine code is a binary 0000 1001 0010 1010 or hexadecimal 092A.

The C field in a machine code word allows for four bits and holds a count in the range of 0 through 15.

19.6 The IOP Field

Turn to the LI instruction summary in Appendix A and look at the machine code description. The Load Immediate instruction has an immediate operand and requires two words of machine code. The requirement for a second word of machine code is indicated in the machine code description by a second word with a solid bottom line.

In the first word of the machine code, the first twelve bits are fixed and the last four bits constitute an R field. The second word of machine code contains the 16-bit immediate operand.

The LI instruction requires two operands: a register number and an immediate operand. In the machine code, the R field holds the register number and the immediate operand holds the immediate operand. Immediate operands are always 16 bits; there are no 8-bit immediate operands.

As an example, the instruction

```
LI  R10,>1234
```

results in an R field of binary 1010 and an immediate operand field of binary 0001 0010 0011 0100.

The first word of machine code is binary 0000 0010 0000 1010 (hexadecimal 020A); the second word of machine code is binary 0001 0010 0011 0100 (hexadecimal 1234).

As another example, the instruction

```
LI  R12,1234
```

(where 1234 is a decimal number) results in the following two words of machine code.

	Binary	Hexadecimal
First Word	0000 0010 0000 1100	020C
Second Word	0000 0100 1101 0010	04D2

An immediate operand can be a negative number. For example, the instruction

    LI R0,-1

results in the following two words of machine code.

		Binary	Hexadecimal
First	Word	0000 0010 0000 0000	0200
Second	Word	1111 1111 1111 1111	FFFF

The immediate operand field is always is 16 bits, is always the second word of the machine code, and contains the immediate operand.

19.7 General Addresssing Mode Fields

Those instructions that allow an operand to use the five general addressing modes result in a more complex machine code structure.

Turn to the instruction summary for the MOV instruction and look at the machine code description. The first four bits are fixed, a binary 1100, and the next twelve bits vary according to the operands used with the instruction. These twelve bits include four fields: a two-bit Td field, a four-bit Rd field, a two-bit Ts field, and a four-bit Rs field.

19.7.1 The Ts and Td Fields

The two-bit Ts and Td fields specify the specific addressing modes that the operands use. The two-bit codes for the Ts and Td fields are as follows.

Ts and Td Code	*Addressing Mode*
00	Register Direct
01	Register Indirect
11	Register Indirect Autoincrement
10	Either Symbolic (Direct Memory) or Indexed, depending upon the accompanying Rd or Rs field

19.7.2 The Rs and Rd Fields

The Rs and Rd fields hold the register number specified by the source operand (Rs) and the register number specified by the destination operand (Rd), if a register is specified.

For example, in the instruction

```
MOV  R9,R14
```

both operands are using register direct addressing. Register 9 is the source operand; Register 14 is the destination operand.

The Td field specifies the type of addressing mode for the *destination* operand. In this example, the destination operand is using register direct addressing and the Td field is a binary 00.

The Rd field specifies the register number used in the *destination* operand if a register number is used. In this example, the destination operand uses Register 14 and the Rd field is a binary 1110.

The Ts field specifies the type of addressing mode for the *source* operand. In this example, the source operand uses register direct addressing and the Ts field is a binary 00.

The Rs field specifies the register number used in the *source* operand if a register number is used. In this example, the source operand uses Register 9 and the Ts field is a binary 1001.

The complete word of machine code is structured as follows.

Fixed Bits	Td	Rd	Ts	Rs
1100	00	1110	00	1001

The 16-bit machine code word is a binary 1100 0011 1000 1001, or hexadecimal C389.

Notice this. In the machine code, the codes for the destination operand appear to the left of the codes for the source operand. This format is just opposite of the order of the operands in the assembly language statement in which the source operand appears to the left of the destination operand.

Consider a second example. The instruction

```
MOV  *R8,R5
```

uses register indirect addressing for the source operand and register direct addressing for the destination operand.

243

Chapter 19

In the machine code, the Ts field is a binary 01 and the Rs field is a binary 1000. Td field code is binary 00 and the Rd field is a binary 0101.

The complete word of machine is structured as follows.

```
Fixed Bits    Td    Rd    Ts    Rs
   1100       00   0101   01   1000
```

The 16-bit machine code word is a binary 1100 0001 0101 1000, or hexadecimal C

Look at a third example. The instruction

```
MOV  *R2+,*R15
```

uses register indirect autoincrement addressing for the source operand and regi indirect addressing for the destination operand.

In the machine code, the Ts field is a binary 11 and the Rs field is a binary 0010. Td field is binary 01 and the Rd field is a binary 1111.

The complete word of machine is structured as follows.

```
Fixed Bits    Td    Rd    Ts    Rs
   1100       01   1111   11   0010
```

The 16-bit machine code word is a binary 1100 0111 1111 0010, or hexadecimal C

Take another example. The instruction

```
MOV  @>A062(R10),R7
```

uses indexed addressing for the source operand and register direct addressing for destination operand.

In the machine code, the Ts field is a binary 10 and the Rs field is a binary 1010. Td field is a binary 00 and the Rd field is a binary 0111.

The complete word of machine is structured as follows.

```
Fixed Bits    Td    Rd    Ts    Rs
   1100       00   0111   10   1010
```

The 16-bit machine code word is a binary 1100 0001 1110 1010, or hexadecimal C1

A Ts or Td field code of binary 10 is used to specify both indexed and symbolic (di

·

he memory) addressing. In the case of symbolic addressing, though, no register number is used and the accompanying Rs or Rd field is set to a binary 0000.

As an example, the instruction

```
MOV  @>B83E,@>A062(R5)
```

58. uses symbolic addressing for the source operand and indexed addressing for the destination operand.

In the machine code, the Ts field is a binary 10 and the Rs field is a binary 0000. The Td field is also a binary 10 and the Rd field is a binary 0101.

ter The complete word of machine is structured as follows.

Fixed Bits	Td	Rd	Ts	Rs
1100	10	0101	10	0000

The 16-bit machine code word is a binary 1100 1001 0110 0000, or hexadecimal C960.

A Ts or Td field code of binary 10 is used to indicate both symbolic and indexed addressing. When the computer interprets the machine code and sees a Ts or Td field with a binary 10, the computer doesn't know whether the operand is using symbolic addressing or indexed addressing until it looks at the accompanying Rs or Rd field. If the accompanying Rs or Rd field contains a non-zero value. the operand is using indexed addressing and the number in the Rs or Rd field is the number of the index register. If the accompanying Rs or Rd field contains zero, the operand is using symbolic addressing.

Notice that an Rs or Rd field of zero when used with a Ts or Td field of binary 10 means Symbolic addressing. Therefore, Register 0 can't be used as an index register. If you specify Register 0 as an index register in an assembly language instruction, the assembler places the four-bit binary number of the index register (0000) in the Rs or Rd field and sets the accompanying Ts or Td field to a binary 10. When the computer performs the machine code, it responds as if the operand is using symbolic addressing.

Whenever the machine code contains a Ts or Td field code of binary 10, an additional word of machine code is required. The additional word holds the 16-bit memory address specified in the operand.

An additional word of machine code is required for each Ts or Td field code of binary 10. If both the Ts and Td field are a binary 10, two additional words are required, and the address value of the memory location specified in the source operand precedes the address value of the memory location specified in the destination operand.

Notice in the machine code description of the MOV instruction that the machine code may require more than one word. The possibility of more than one word is indicated by the dashed lines at the bottom of the second and third lines.

For example, the instruction

```
MOV  @>B83E,@>A062(R5)
```

requires two additional words of machine code, one for the source operand memory address and one for the destination operand memory address. The three words of machine code are as follows.

```
First  Word      >C960
Second Word      >B83E
Third  Word      >A062
```

As a second example, the instruction

```
MOV  R4,@WINKLE
```

requires one additional word of machine code or a total of two words.

The first word of machine code is hexadecimal C804; the second word of machine code contains the address value of WINKLE.

There are some instructions that have only one operand that can use the general addressing modes.

Turn to the instruction summary for the SWPB instruction and look at the machine code description. The first ten bits are fixed and the next six bits consist of a two-bit Ts field and a four-bit Rs field. The SWPB instruction has one operand, called a source operand, that can use the general addressing modes.

For example, the instruction

```
SWPB  R11
```

uses register direct addressing for the source operand.

In the machine code, the Ts field code is a binary 00 and the Rs field is a binary 1011.

The complete word of machine code is structured as follows.

```
Fixed Bits     Ts     Rs
0000011011     00     1011
```

16-bit machine code word is a binary 0000 0110 1100 1011, or hexadecimal 06CB.

As another example, the instruction

    SWPB @SABRE

uses symbolic addressing for the source operand.

The instruction requires two words of machine code. In the first word, the Ts field code is a binary 10 and the Rs field is a binary 0000.

The first word of machine is structured as follows.

    Fixed Bits   Ts   Rs
    0000011011   10   0000

The first word machine code word is a binary 0000 0110 1110 0000, or hexadecimal 06E0. The second word contains the address value of SABRE.

Each valid @ sign in an instruction results in a Ts or Td field of binary 10 and requires an additional word of machine code.

19.8 The Displacement Field

A Displacement field appears only in the machine code format of the single-bit CRU instructions (SBO, SBZ, and TB). The Displacement field holds an eight-bit value which the computer adds to the hardware base address in Register 12 when a single-bit CRU instruction is performed.

Turn to the instruction summary for the SBO instruction and look at the machine code description. The instruction requires only one word of machine code. The first eight bits are fixed, a binary 0001 1101, and the last eight bits make up the variable Displacement field. The assembler puts into this field the value of the Displacement operand in the assembly language instruction. For example, the instruction

    SBO 3

results in a Displacement field of binary 0000 0011. The 16-bit machine code word is a binary 0001 1101 0000 0011, or hexadecimal 1D03.

The instruction

```
SBO  -10
```

where 10 is a decimal number results in a Displacement field of binary 1111 0110
16-bit machine code word is a binary 0001 1101 1111 0110, or hexadecimal 1DF6.

The Displacement operand with the single-bit CRU instructions is limited to the
of −128 to +127 because that's the range of the eight-bit signed number that goes i
Displacement field of the machine code. If you use a Displacement operand outsid
range, the assembler flags it as an error.

The eight-bit Displacement field is used only with the single-bit CRU instruction

19.9   The PC Word Displacement Field

A PC Word Displacement field appears only in the machine code of the jump instruc
It's used to tell the computer how many words to add to the
address in the Program Counter (PC) to cause a jump to the right location.

Turn to the instruction summary for the JNE instruction and look at the machine
description. The instruction requires only one word of machine code. The first eigh
are fixed (a binary 0001 0110) and the last eight bits make up the variable PC
Displacement field. The assembler places in this field a value equal to the numb
words that must be added to the address in the Program Counter to cause a jump t
correct target location.

The PC Word Displacement field is an eight-bit field and, therefore, the number i
field is limited to the range of an eight-bit signed number (−128 to +127).

Recall that the Program Counter is the register in the computer's central processo
is continually updated to hold the address of the next instruction to be performe
the computer performs an instruction, the address in the Program Count
automatically increased to the address immediately following the address o
instruction currently being performed. This is done automatically because instruc
usually are performed in sequential order. So, when an instruction in a location is
performed, the address in the Program Counter points to the next location.

This is true of all instructions, including the jump instructions. When a jump instru
is performed, the Program Counter is pointing to the location immediately after the
instruction. If a jump is made, the computer adds the PC Word Displacement i
machine code to the *word* address in the Program Counter. The sum is the addr
the instruction to be performed next.

Notice that it's the number of *words* (the *word* displacement) that is added to the *word* address in the Program Counter.

When writing a jump instruction in assembly language, you specify the distance to the target in terms of the location of the instruction, not where the Program Counter points when the machine code is performed. And the distance to the target is measured in bytes, not words.

The assembler that translates an assembly language jump instruction into machine code has a way of resolving all this. It translates the distance to the target *measured in bytes from the location of the instruction* into a PC Word Displacement in the machine code which measures the distance to the target in *words from the address in the Program Counter.*

Whenever the assembler encounters a jump instruction, it simply takes the byte displacement specified by the Target operand and divides it by two to convert the byte displacement to a displacement measured in words. Then it subtracts one from this amount to compensate for the fact that the Program Counter points to the location following the jump instruction when the machine code instruction is performed.

The formula the assembler uses for translating a Target operand in assembly language to a PC Word Displacement in machine code is

    (N/2) - 1

where N is the signed displacement, measured in bytes, to the target from the location of the instruction.

Take an example. The instruction

    JNE $+10

specifies a target for the jump which is ten bytes ahead of the location of the JNE instruction. The value of N is +10. The assembler applies the formula and derives plus 4.

    (+10/2) - 1 = +4

The assembler places a plus 4 in the eight-bit PC Word Displacement field (a binary 0000 0100). The entire 16-bit machine code word is a binary 0001 0110 0000 0100, or hexadecimal 1604.

Notice in the formula that N is a *signed* number. For example, the instruction

```
JNE  $-10
```

specifies a target that is ten bytes behind the location of the JNE instruction and the value of N is -10.

The assembler applies the formula and derives minus 6.

```
(-10/2) - 1  =  -6
```

The assembler places a minus 6 (a binary 1111 1010) in the eight-bit PC Word Displacement field. The entire 16-bit machine code word is a binary 0001 0110 1111 1010, or hexadecimal 16FA.

The assembler performs the same calculation even if the operand of the jump instruction is the label of the target instruction. The assembler simply calculates the distance in bytes to the target and then applies the same formula to determine the PC Word Displacement.

For example, the instruction

```
JNE   ROSCOE
```

specifies as a target the instruction labeled ROSCOE. Suppose the JNE instruction occupies location hexadecimal A34E and the instruction labeled ROSCOE occupies location hexadecimal A32C.

```
Location                Instruction

    :                        :
    :                        :
  >A32C        ROSCOE   ------------
    :                        :
    :                        :
  >A34E                 JNE  ROSCOE
    :                        :
    :                        :
```

The distance from the JNE instruction to its target can be calculated as follows.

```
   >A34E    Location of JNE Instruction
 - >A32C    Location of Target Instruction
   -------
   >0022    Displacement in Bytes to the Target
```

Since hexadecimal 22 is a decimal 34 and since the target is behind the jump instruction (a minus direction), the displacement to the target is −34. This is the number that the assembler uses to calculate the PC Word Displacement in the machine code.

```
   (-34/2) - 1  =  -18
```

The PC Word Displacement is −18 decimal, or −12 hexadecimal.

The assembler places a minus hexadecimal 12 in the eight-bit PC Word Displacement field (a binary 1110 1110).

The entire 16-bit machine code word is a binary 0001 0110 1110 1110, or hexadecimal 16EE.

Chapter 20

SUMMARY

This book has covered a lot of ground. The structure of assembly language programs and the structure of data used by those programs has been introduced. The instructions and addressing modes of the TI Home Computer's instruction set have been presented with several examples of how to use the instructions. The function of assembly language development tools has been described and the example programs in the book can be used to experiment with assembly language utility programs such as assemblers and debuggers. The basic concepts of assembly language programming in general and, specifically, for the TI Home Computer has been covered.

The purpose of this book is to lay a foundation to help you understand programs written in assembly language for the TI Home Computer and to help you get started in creating your own programs. Hopefully, it has achieved that purpose for you. Good programming!

Appendix A

Instruction Summaries

Appendix A

INSTRUCTION OPERATION CODES IN ALPHABETICAL ORDER

Op-Code	Syntax	Op-Code	Syntax
A	S, D	LDCR	S, C
AB	S, D	LI	R, IOP
ABS	S	LIMI	IDP
AI	R, IDP	LREX	
ANDI	R, IDP	LWPI	IDP
B	S	MOV	S, D
BL	S	MOVB	S, D
BLWP	S	MPY	S, R
C	S, D	NEG	S
CB	S, D	ORI	R, IOP
CI	R, IOP	RSET	
CKON		RTWP	
CKDF		S	S, D
CLR	S	SB	S, D
COC	S, R	SBO	Displacement
CZC	S, R	SBZ	Displacement
DEC	S	SETO	S
DECT	S	SLA	R, C
DIV	S, R	SOC	S, D
IDLE		SOCB	S, D
INC	S	SRA	R, C
INCT	S	SRC	R, C
INV	S	SRL	R, C
JEQ	Target	STCR	S, C
JGT	Target	STST	R
JH	Target	STWP	R
JHE	Target	SWPB	S
JL	Target	SZC	S, D
JLE	Target	SZCB	S, D
JLT	Target	TB	Displacement
JMP	Target	X	S
JNC	Target	XOP	S, C
JNE	Target	XOR	S, R
JND	Target		
JOC	Target		
JOP	Target		

Add Words   A

Mnemonic and Addressing Modes: A S,D

Result: (S) + (D) --> (D)

Operation: Adds the contents of the first operand to the contents of the
    second operand. Replaces the contents of the second operand with the
    sum. Both operands are word addresses.

Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |L>|A>|EQ|CY|OV|  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
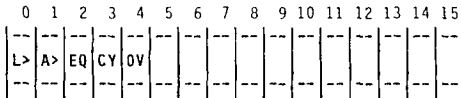
    The addition affects the Carry and Overflow status bits.
    The sum is compared to zero and the Logical Greater Than,
    Arithmetic Greater Than, and Equal status bits are
    affected accordingly.

Notes: The Add Words instruction offers the widest choice of
    addressing modes for an add operation.

    Example: A R4,@ALPHA

                  Before   After

    (R4)   =   >5A74   >5A74
    (ALPHA) =   >BC5A   >16CE

                               L> = 1, A> = 1, EQ = 0
                               CY = 1, OV = 0

Machine Code:

```
   Hex       <--- A --->  <--- - --->  <--- - --->  <--- - --->
             0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary   | 1| 0| 1| 0| T d|   R d  | T s|   R s  |
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                  Source (or Destination) Address
            - - - - - - - - - - - - - - - - - - - - - -
                        Destination Address
            - - - - - - - - - - - - - - - - - - - - - -
```

                  Length: 1 or 2 or 3 words

Add Bytes   AB

Mnemonic and Addressing Modes: AB S,D

                        Bytes
Result: (S) + (D) -----> (D)

Operation: Adds the contents of the first operand to the contents of the
    second operand. Replaces the contents of the second operand with the
    sum. Both operands are byte addresses.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 1D 11 12 13 14 15
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   |L>|A>|EQ|CY|DV|OP|  |  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
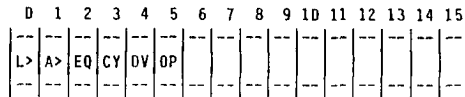
The addition affects the Carry and Overflow status bits.
The sum is compared to zero and the Logical Greater Than,
Arithmetic Greater Than, and Equal status bits are
affected accordingly. The DP status bit is set to one if
there's an odd number of one bits in the sum; otherwise;
it's cleared to zero.

Notes: The Add Bytes instruction adds two 8-bit numbers together.

    Example: AB R4,@ALPHA

                   Before   After

    (R4)   =   >8074   >8074
    (ALPHA) =   >BC5A   >3C5A

                                 L> = 1, A> = 1, EQ = 0
                                 CY = 1, OV = 1, OP = 0

Machine Code:

    Hex   <--- B ---> <--- - ---> <--- - ---> <--- - --->
              0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary | 1| 0| 1| 1| T d | R d | T s | R s |
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
             | Source (or Destination) Address |
             |- -|
             | Destination Address |
             |- -|

              Length: 1 or 2 or 3 words
```

Absolute                                                          ABS


Mnemonic and Addressing Modes:  ABS S

Result:  $|(S)|$  --> (S)

Operation:  Takes the absolute value of the contents of the operand.

Status Bits Affected:

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
    |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    |L>|A>|EQ|  |OV|  |  |  |  |  |  |  |  |  |  |  |
    |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

The content of the operand before the absolute value is
taken is compared to zero and the Logical Greater Than,
Arithmetic Greater Than, and Equal status bits are
affected accordingly. If the content of the operand is
>8000, the Overflow status bit is set; otherwise, it's
cleared to zero.

Notes: The Absolute instruction forces a value to a positive number. If
the value is already positive, it's unchanged. If the value is
negative, it's forced to its two's complement value. However, if the
value is hex 8000 (the smallest negative number and which doesn't have
a positive counterpart), the computer sets the Overflow status bit to
indicate it can't form a positive number.

Example: ABS   @NUMBER

                 Before   After

(NUMBER) =   >FFF9   >0007

                           L> = 1, A> = 0, EQ = 0
                           OV = 0

Machine Code:

```
Hex       <--- 0 ---> <--- 7 ---> <--- - ---> <--- - --->
           0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary    | 0| 0| 0| 0| 0| 1| 1| 1| 0| 1| T  s|   R  s   |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
          |              Source Address                   |
          |- - - - - - - - - - - - - - - - - - - - - - - -|
```

                Length: 1 or 2 words

Add Immediate   AI

Mnemonic and Addressing Modes: AI   R,IOP

Result: (R) + IOP --> (R)

Operation: Adds the 16-bit immediate operand to the contents of the register.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |L>|A>|EQ|CY|OV|  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

The addition affects the Carry and Overflow status bits.
The sum is compared to zero and the Logical Greater Than,
Arithmetic Greater Than, and Equal status bits are
affected accordingly.

Notes: The AI instruction is useful for adding a specific constant to the
contents of a register.

Example: AI   R3,4

                Before   After

(R3)   =   >BC5A   >BC5E

                              L> = 1, A> = 0, EQ = 0
                              CY = 0, OV = 0

Machine Code:

```
Hex        <--- 0 --->  <--- 2 --->  <--- 2 --->  <--- - --->
            0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary    | 0| 0| 0| 0| 0| 0| 1| 0| 0| 0| 1| 0|     R      |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                          Immediate Operand
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

Length: 2 words

And Immediate   ANDI

Mnemonic and Addressing Modes: ANDI R,IOP

Result: (R) AND IOP --> (R)

Operation: Performs a logical AND operation between the bits in the
     register and the bits in the immediate operand. The result replaces the
     contents of the register.

Status Bits Affected:

```
  0  I  2  3  4  5  6  7  8  9 10 I1 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|L>|A>|EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

     The result is compared to zero and the Logical Greater
     Than, Arithmetic Greater Than, and Equal status bits are
     affected accordingly.

Notes: The ANDI instruction is useful for forcing selected bits in a
     register to zero.

     Example: ANDI R0,>A60F

                  Before   After

     (RD)   =   >BC5A   >A4DA

                               L> = 1, A> = 0, EQ = 0

     (RO) Before = 10II 11D0 0I01 1D10
     IOP   = 1010 01I0 0000 1lI1

     (RD) After = 1010 01D0 0DD0 1010

Machine Code:

```
  Hex      <--- 0 ---> <--- 2 ---> <--- 4 ---> <--- - --->
           0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  Binary  | 0| 0| 0| 0| 0| 0| 1| 0| 0| 1| 0| 0|     R     |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
          |                 Immediate Dperand              |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                        Length: 2 words

Branch   B

Mnemonic and Addressing Modes: B   S

Result: S --> (PC)

Operation: Transfers program control to the operand address.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

No status bits are affected.

Notes: The B instruction performs a long-distance transfer of program
         control. The B instruciton performs an unconditional transfer of
         control as does the JMP instruction. But whereas the JMP instruction
         allows only a short-distance transfer of control, the B instruction
         permits a transfer of control to any location in the memory space.

   Example: B @TICKET

   Causes a transfer of program control to location TICKET.

Machine Code:

```
   Hex        <--- 0 --->  <--- 4 --->  <--- - --->  <--- - --->
              0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary    | 0| 0| 0| 0| 0| 1| 0| 0| 0| 0| 1| T s  |  R s   |
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
             |              Source Address                    |
             |- - - - - - - - - - - - - - - - - - - - - - - - -|
```

                       Length: 1 or 2 words

Branch and Link   BL

Mnemonic and Addressing Modes: BL   S

Result: (PC) --> (R11)
          S   --> (PC)

Operation: Transfers program control to the operand address and saves the
    address following the BL instruction in Register 11.

Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

    No status bits are affected.

Notes: The BL instruction calls a subroutine. The return address is
    saved in Register 11. The subroutine can return to the calling
    program by performing a Branch instruction to the address in
    Register 11 (B *R11).

```
    Example:  Calling Program                  Subroutine
              ---------------       +---> SUBR  ----------
                   :                |           :
                   :                |           :
                 BL  @SUBR        --+          ----------
              ---------------    <------------  B  *R11
                   :
                   :
```

Machine Code:

```
    Hex      <--- 0 ---> <--- 6 ---> <--- - --->  <--- - --->
             0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  Binary   | 0| 0| 0| 0| 0| 1| 1| 0| 1| 0| T  s|    R  s  |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
           |              Source Address                   |
           | - - - - - - - - - - - - - - - - - - - - - - - |
```

                    Length: 1 or 2 words

                                                              263

Branch and Load Workspace Pointer   BLWP

Mnemonic and Addressing Modes: BLWP S

Result: (S) --> (WP)
          (S + 2) --> (PC)
          (old WP) --> (new R13)
          (old PC) --> (new R14)
          (old SR) --> (new R15)

Operation: Performs a context switch using the two-word vector
    specified by the operand address. The operand is the address
    of the first word of the vector and it contains the address of
    a new workspace. The second word of the vector contains the
    address of the program to which a transfer of control is made.
    When the BLWP instruction is perfomed, the current contents of
    the WP, PC, and SR are stored in Registers 13, 14, and 15,
    respectively, of the new workspace.

Status Bits Affected:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

    No status bits are affected.

Notes: The BLWP instruction can be used to perform a context switch
    to any subroutine or program in the memory address space.

    Example: Calling Program   Called Program
                       :   :
                 BLWP @SQVECT ------+   SUBWSP BSS 32
                 -------------- <--+ | :
                       : | +---> SUBENT -------
                       : | :
                 SQVECT DATA SUBWSP | :
                        DATA SUBENT +-------   RTWP
```
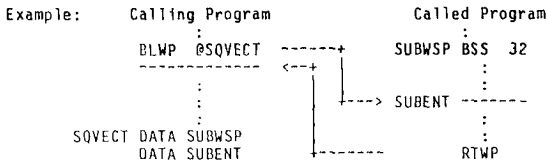
    A context switch is performed using the contents of SQVECT as
    the address for a new workspace and the contents of SQVECT+2 as
    the address of a program to which a transfer of program control is made.

Machine Code:

    Hex   <--- 0 ---> <--- 4 ---> <--- - ---> <--- - --->
              0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary | 0| 0| 0| 0| 0| 1| 0| 0| 0| 0| T s| R s |
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
             | Source Address |
             ~- ~
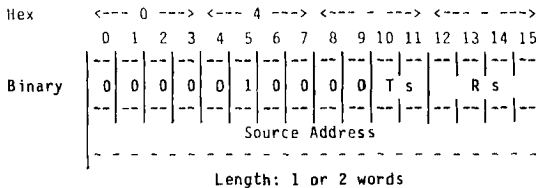
                       Length: 1 or 2 words

Compare Words   C

Mnemonic and Addressing Modes: C   S,D

Result: L>, A>, and EQ status bits affected based upon
         (S):(D)

Operation: Compares the contents of the first operand to the contents
    of the second operand. The operands are word addresses. The
    Logical Greater Than, Arithmetic Greater Than, and Equal status
    bits are affected based upon the results of the comparison.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |L>|A>|EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
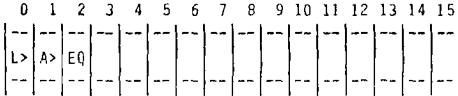
    The contents of the first operand is compared to the
    contents of the second operand and the Logical Greater
    Than, Arithmetic Greater Than, and Equal status bits are
    affected accordingly.

Notes: The Compare Word instruction compares two 16-bit
    values together. Neither the contents of the source
    operand or the contents of the destination operand are
    changed.

    Example: C   @ALPHA,R4

                 Before   After

    (ALPHA) =   >A374   >A374
    (R4)   =   >6E2F   >6E2F

                               L> = 1, A> = 0, EQ = 0

Machine Code:

```
    Hex      <--- 8 --->  <--- - --->  <--- - --->  <--- - --->
             0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary | 1| 0| 0| 0| T d |  R d   | T s |  R s   |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
           |         Source (or Destination) Address        |
           - - - - - - - - - - - - - - - - - - - - - - - - -
           |              Destination Address               |
           - - - - - - - - - - - - - - - - - - - - - - - - -
```
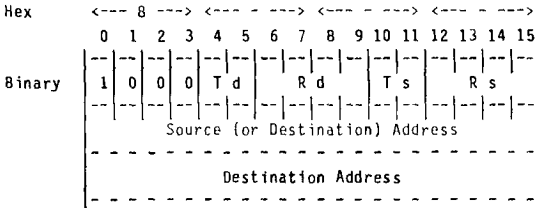
                Length: 1 or 2 or 3 words

Compare Bytes   CB

Mnemonic and Addressing Modes: CB S,D

Result: L>, A>, and EQ status bits affected based upon
         (S):(D). OP status bit affected based upon number
         of one bits in (S).

Operation: Compares the contents of the first operand to the contents
         of the second operand. The operands are byte addresses. The Logical
         Greater Than, Arithmetic Greater Than, and Equal status bits are
         affected based upon the results of the comparison. The Odd Parity
         status bit is affected based upon the number of one bits in the source
         operand.

Status Bits Affected:

```
    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
 |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
 | L>| A>| EQ|   |   |   | OP|   |   |   |   |   |   |   |   |   |
 |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
```

The contents of the first operand is compared to the
contents of the second operand and the Logical Greater
Than, Arithmetic Greater Than, and Equal status bits are
affected accordingly. The Odd Parity status bit is
affected based upon the number of one bits in the
source operand.

Notes: The Compare Word instruction compares two 8-bit values together.
     Neither the contents of the source operand or the contents of the
     destination operand are changed.

     Example: C @ALPHA(R7),R4

                  Before   After

     (ALPHA) =   >74A3   >74A3
     (R4)   =   >F26E   >F26E
     (R7)   =   >0001   >0001

                          L> = 0, A> = 0, EQ = 0, OP = 0

Machine Code:
   Hex   <--- 9 ---> <--- - ---> <--- - ---> <--- - --->
            0   1   2   3   4   5   6   7   8   9 10 11 12 13 14 15
            |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
   Binary | 1 | 0 | 0 | 1 | T d | R d | T s | R s |
            |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
            | Source (or Destination) Address |
            |- -|
            | Destination Address |
            |- -|
                        Length: 1 or 2 or 3 words

266

Compare Immediate   CI

Mnemonic and Addressing Modes: CI   R,IOP

Result: L>, A>, and EQ status bits affected based upon
          (R):(IOP).

Operation: Compares the contents of the register to the immediate
    operand.   The Logical Greater Than, Arithmetic Greater Than, and
    Equal status bits are affected based upon the results of the comparison.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |L>|A>|EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

    The contents of the register is compared to the contents
    of the immediate operand and the Logical Greater Than,
    Arithmetic Greater Than, and Equal status bits are
    affected accordingly.

Notes: The Compare Immediate instruction can be used to check for a
    specific address value in a register when the register is being used
    with register indirect autoincrement or indexed addressing in a loop.

    Example: CI   R10,TBLEND

                    Before   After

    (R10)   =   >A49C   >A49C
    TBLEND   =   >A49C   >A49C

                    L> = 0, A> = 0, EQ = 1

Machine Code:

```
    Hex        <--- 0 --->  <--- 2 --->  <--- 8 --->  <--- - --->
               0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
              |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary    | 0| 0| 0| 0| 0| 0| 1| 0| 1| 0| 0| 0|      R      |
              |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                             Immediate Operand
              |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                             Length: 2 words
```


                                                                                     267

Clock Off   CKOF

Mnemonic and Addressing Modes: CKOF

Result: Sends signals out on address lines and CRUOUT line.

Operation: When the CKOF instruction is performed, the binary values
   1, 1, and 0 appear on address lines A0, A1, and A2, respectively,
   in conjunction with a synchronizing pulse on the CRUCLK line.

Status Bits Affected:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

No status bits are affected.

Notes: The CKOF instruction may be used to implement functions unique
   to hardware surrounding the central processor.

Machine Code:

```
Hex      <--- 0 --->  <--- 3 --->  <--- C --->  <--- 0 --->
         0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary  | 0| 0| 0| 0| 0| 0| 1| 1| 1| 1| 0| 0| 0| 0| 0| 0|
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

Length: 1 word

Clock On   CKON

Mnemonic and Addressing Modes: CKON

Result: Sends signals out on address lines and CRUOUT line.

Operation: When the CKON instruction is performed, the binary values
    1, 0, and 1 appear on address lines A0, A1, and A2, respectively,
    in conjunction with a synchronizing pulse on the CRUCLK line.

Status Bits Affected:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

    No status bits are affected.

Notes: The CKON instruction may be used to implement functions unique
    to hardware surrounding the central processor.

Machine Code:

```
Hex      <--- 0 --->  <--- 3 --->  <--- A --->  <--- 0 --->
          0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary   | 0| 0| 0| 0| 0| 0| 1| 1| 1| 0| 1| 0| 0| 0| 0| 0|
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                        Length: 1 word

Clear   CLR

Mnemonic and Addressing Modes: CLR S

Result: 0 --> (S)

Operation: Forces the content of the operand to zero.

Status Bits Affected:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

No status bits are affected.

Notes: The CLR instruction is useful for initializing a word to zero.

Example: CLR @ZFFLAG

                 Before   After

(ZFFLAG) =   >72E9   >0000

                             No status bits affected.

Machine Code:

```
Hex       <--- 0 ---> <--- 4 ---> <--- - ---> <--- - --->
          0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary   | 0| 0| 0| 0| 0| 1| 0| 0| 1| 1| T  s |   R  s   |
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
         |              Source Address                    |
         - - - - - - - - - - - - - - - - - - - - - - - - -
```
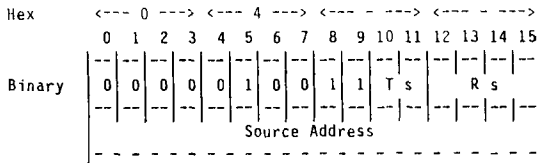
Length: 1 or 2 words

Compare Ones Corresponding   COC

Mnemonic and Addressing Modes: COC S,R

Result: (S) AND complement (R); set Equal status bit if
          result zero.

Operation: Sets the Equal status bit if there are all one bits in the
    contents of the second operand corresponding to the bit positions
    where there are one bits in the contents of the first operand.

Status Bits Affected:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

    The Equal status bit is set if there are all one bits
    in the contents of the second operand corresponding to
    the bit positions where there are one bits in the
    contents of the first operand; otherwise, it's cleared.

Notes: The COC instruction can be used to test the state of specific bits
    in the contents of a register.

    Example: COC @ALPHA,R4

                 Before   After   Binary

    (ALPHA) =   >3A74   >3A74   0011 1010 0111 0100
    (R4)   =   >BC5A   >BC5A   1011 1100 0101 1010
                               EQ = 0

Machine Code:

```
Hex          <--- 2 --->  <--- - --->  <--- - --->  <--- - --->
              0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary      | 0| 0| 1| 0| 0| 0|    R     | T  s |    R  s |
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
            |              Source Address                  |
            |- - - - - - - - - - - - - - - - - - - - - - - -|
```

                 Length: 1 or 2 words

271
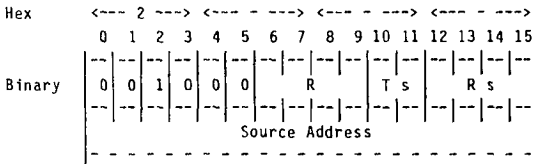
Appendix A

Compare Zeros Corresponding   CZC

Mnemonic and Addressing Modes: CZC S,R

Result: (S) AND (R); set Equal status bit if result zero.

Operation: Sets the Equal status bit if there are all zero bits in the
        contents of the second operand corresponding to the bit positions
        where there are one bits in the contents of the first operand.

Status Bits Affected:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
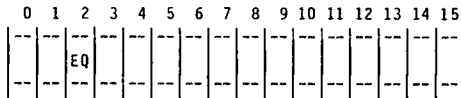
The Equal status bit is set if there are all zero bits
in the contents of the second operand corresponding to
the bit positions where there are one bits in the
contents of the first operand; otherwise, it's cleared.

Notes: The CZC instruction can be used to test the state of specific bits
        in the contents of a register.

        Example: CZC @ALPHA,R4

                      Before   After   Binary

        (ALPHA) =   >3A74   >3A74   0011 1010 0111 0100
        (R4)   =   >C482   >C482   110D 0100 1DDD 0D10

                                    EQ = 1

Machine Code:

```
 Hex      <--- 2 --->  <--- - --->  <--- - --->  <--- - --->
          0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 Binary   | 0| 0| 1| 0| 0| 1|    R    |  T s  |    R s    |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
          |              Source Address                   |
          |- - - - - - - - - - - - - - - - - - - - - - - -|
```

                      Length: 1 or 2 words

272
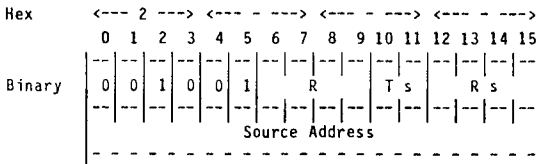
Decrement   DEC

Mnemonic and Addressing Modes: DEC   S

Result: (S) - 1 --> (S)

Operation: Subtracts one from the contents of the operand.

Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |L>|A>|EQ|CY|OV|  |  |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

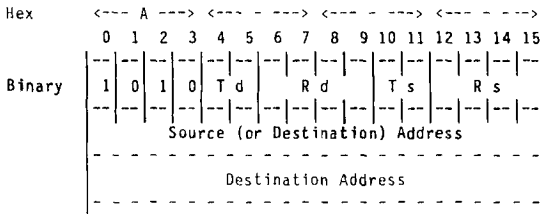The computer adds a negative 1 (hexadecimal FFFF) to the
contents of the operand. The addition affects the Carry
and Overflow status bits. The result is compared to zero
and the Logical Greater Than, Arithmetic Greater Than,
and Equal status bits are affected accordingly.

Notes: The Decrement instruction is useful for decrementing a byte
         address in a register which is being used for indirect addressing or
         indexed addressing. It's also useful for adjusting a loop counter in a
         program loop.

   Example: DEC   R2

                 Before   After

   (R2)   =   >D009   >0008

                               L> = 1, A> = 1, EQ = D
                               CY = 1, OV = 0

Machine Code:

```
   Hex      <--- 0 ---> <--- 6 ---> <--- - ---> <--- - --->
            0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary  | 0| 0| 0| 0| 0| 1| 1| 0| 0| 0| T  s |   R  s  |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
           |              Source Address              |
           - - - - - - - - - - - - - - - - - - - - - - -
```

                 Length: 1 or 2 words

273
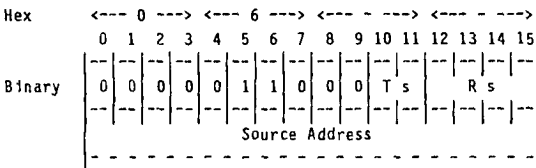
Decrement by Two   DECT

Mnemonic and Addressing Modes: DECT   S

Result: (S) - 2 --> (S)

Operation: Subtracts two from the contents of the operand.

Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|L>|A>|EQ|CY|OV|  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
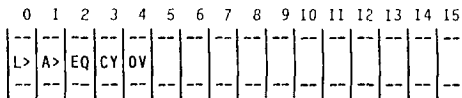
The computer adds a negative 2 (hexadecimal FFFE) to the
contents of the operand. The addition affects the Carry
and Overflow status bits. The result is compared to zero
and the Logical Greater Than, Arithmetic Greater Than,
and Equal status bits are affected accordingly.

Notes: The Decrement by Two instruction is useful for decrementing
a word address in a register which is being used for indirect
addressing or indexed addressing.

Example: DECT   R2

                Before   After

(R2)   =   >0009   >0007

                                L> = 1, A> = 1, EQ = 0
                                CY = 1, OV = 0

Machine Code:

```
Hex      <--- 0 ---> <--- 6 ---> <--- - ---> <--- - --->
          0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary  | 0| 0| 0| 0| 0| 1| 1| 0| 0| 1| T  s|    R  s  |
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
        |                Source Address                 |
        |- - - - - - - - - - - - - - - - - - - - - - - -|
```
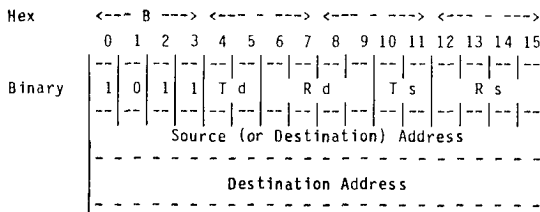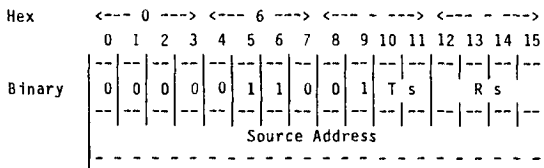
Length: 1 or 2 words

Divide   DIV

Mnemonic and Addressing Modes: DIV S,R

Result: (R and R+1) / (S) --> (R) Quotient and
                                   (R+1) Remainder

Operation: Divides the word value in the first operand (the divisor) into
the 32-bit dividend in the destination register and the next register.
The destination register contains the most significant 16 bits of the
dividend and the next register contains the least significant 16 bits.
After the division, the destination register contains the 16-bit
quotient and the next register contains the 16-bit remainder. Before
the division, the 16-bit divisor in the first operand is compared to the
16-bit value in the destination register. If the divisor is less than,
or equal to, this most significant word of the dividend, the quotient
would exceed 16 bits and, in such a case, the computer sets the Overflow
status bit and does not perform the divide. The numbers are treated as
unsigned values.

Status Bits Affected:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |OV|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
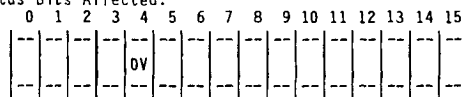
The Overflow status bit is set if the 16-bit divisor
is less than, or equal to, the contents of the
destination register; otherwise it's cleared to zero.

Notes: The Divide instruction divides a 16-bit number into a 32-bit number.
It's an unsigned division; the computer ignores the sign of the numbers.

```
Example:        DIV  @DIVISR,R5
                Before          After

(DIVISR) =      >0008           >0008
(R5)     =      >0000           >000C
(R6)     =      >0064           >0004

                                OV = 0
```

Machine Code:

```
Hex     <--- 3 --->  <--- - --->  <--- - --->  <--- - --->
         0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary  | 0| 0| 1| 1| 1| 1|     R     | T  s|     R  s   |
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
        |              Source Address                    |
        | - - - - - - - - - - - - - - - - - - - - - - - -|
```

Length: 1 or 2 words

275
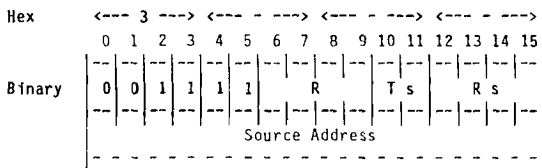
Idle   IDLE

Mnemonic and Addressing Modes: IDLE

Result: Idle the computer
         Sends signals out on address lines and CRUCLK line.

Operation: Places the computer into the idle state. The computer performs
    the IDLE instruction continuously until an interrupt occurs. When an
    interrupt occurs that causes a context switch, the return address saved
    is the location following the IDLE instruction. When the IDLE
    instruction is performed, the binary values 0, 1, and 0 appear on
    address lines A0, A1, and A2, in conjunction with a synchronizing pulse
    on the CRUCLK line.

Status Bits Affected:

```
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
 |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
 | --| --| --| --| --| --| --| --| --| --| --| --| --| --| --| --|
 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
 | --| --| --| --| --| --| --| --| --| --| --| --| --| --| --| --|
 |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
```
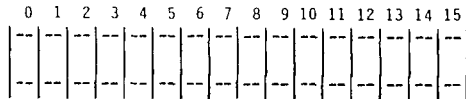
    No status bits are affected.

Notes: The IDLE instruction may be used to implement functions unique to
    hardware surrounding the central processor.

Machine Code:

```
  Hex      <--- 0 --->  <--- 3 --->  <--- 4 --->  <--- 0 --->
           0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
         |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
  Binary | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
         |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
```

                          Length: 1 word

276
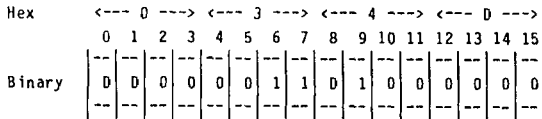
Increment   INC

Mnemonic and Addressing Modes: INC   S

Result: (S) + 1 --> (S)

Operation: Adds one to the contents of the operand.

Status Bits Affected:

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   |L>|A>|EQ|CY|OV|  |  |  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

The addition affects the Carry and Overflow status bits.
The sum is compared to zero and the Logical Greater Than,
Arithmetic Greater Than, and Equal status bits are
affected accordingly.

Notes: The Increment instruction is useful for incrementing a byte address
in a register which is being used for indirect addressing or indexed
addressing.   It's also useful for adjusting a loop counter in a program
loop.

Example: INC   R2

               Before   After

(R2)   =   >0009   >000A

                               L> = 1, A> = 1, EQ = 0
                               CY = 0, OV = 0

Machine Code:

```
Hex       <--- 0 ---> <--- 5 ---> <--- - ---> <--- - --->
          0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary  | 0| 0| 0| 0| 0| 1| 0| 1| 1| 0| T  s|     R  s |
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
        |            Source Address                     |
        |- - - - - - - - - - - - - - - - - - - - - - - -|
```

                    Length: 1 or 2 words

                                                                      277

Increment by Two   INCT

Mnemonic and Addressing Modes: INCT S

Result: (S) + 2 --> (S)

Operation: Adds two to the contents of the operand.

Status Bits Affected:

```
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|L> |A> |EQ |CY |OV |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
```

The addition affects the Carry and Overflow status bits.
The sum is compared to zero and the Logical Greater Than,
Arithmetic Greater Than, and Equal status bits are
affected accordingly.

Notes: The Increment by Two instruction is useful for incrementing a word
        address in a register which is being used for indirect addressing or
        indexed addressing.

        Example: INCT R2

                     Before   After

        (R2)   =   >0009   >000B

                                  L> = 1, A> = 1, EQ = 0
                                  CY = 0, OV = 0

Machine Code:

```
Hex       <--- 0 --->  <--- 5 --->  <--- - --->  <--- - --->
          0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
         |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
Binary   | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | T s |   | R s |
         |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
                              Source Address
         - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```
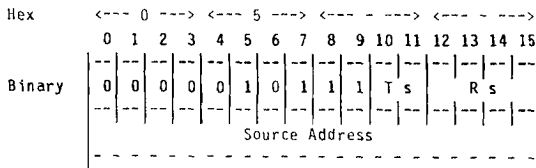
                        Length: 1 or 2 words

Invert   INV

Mnemonic and Addressing Modes: INV   S

Result: Complement (S) --> (S)

Operation: Inverts the state of each bit in the operand. Leaves the one's
   complement of the original value in the operand.

Status Bits Affected:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|L>|A>|EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

   The result is compared to zero and the Logical Greater
   Than, Arithmetic Greater Than, and Equal status bits are
   affected accordingly.

Notes: The INV instruction is a logical NOT operation. It changes all the
   one bits in a word to zeros and all the zero bits to ones.

   Example: INV   @PNFLAG

                 Before   After

   (PNFLAG) =   >FFFF   >0000

                              L> = 0, A> = 0, EQ = 1

Machine Code:

```
   Hex        <--- 0 --->  <--- 5 --->  <--- - --->  <--- - --->
              0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary    | 0| 0| 0| 0| 0| 1| 0| 1| 0| 1| T  s|     R  s  |
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                              Source Address
             |- - - - - - - - - - - - - - - - - - - - - - - -|
```

                 Length: 1 or 2 words

279

Jump if Equal   JEQ

Mnemonic and Addressing Modes: JEQ   Target

Result: If EQ = 1, jump to target
          If EQ = 0, continue to following instruction

Operation: Jumps to the target instruction if the Equal status bit is one.
   Otherwise, there is no jump and program control continues to the
   following instruction. The target must be within -254 to +256 bytes
   from the location of the JEQ instruction.

Status Bits Affected:

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

   No status bits are affected.
   The EQ status bit is analyzed.

Notes: The JEQ instruction can be used to check the state of the Equal
   status bit resulting from a previous instruction's operation.   It's
   often used at the end of a loop to determine if the loop count is zero.

   Example: JEQ   GALLOP

   The JEQ instruction jumps to the instruction labeled GALLOP if the Equal
   status bit is one.

Machine Code:

```
   Hex        <--- 1 ---> <--- 3 ---> <--- - ---> <--- - --->
              0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary   | 0| 0| 0| 1| 0| 0| 1| 1|  PC Word Displacement  |
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                          Length: 1 word

Jump if Greater Than   JGT

Mnemonic and Addressing Modes: JGT Target

Result: If A> = 1, jump to target;
          otherwise, continue to following instruction

Operation: Jumps to the target instruction if the Arithmetic Greater Than
    status bit is one. Otherwise, there is no jump and program control
    continues to the following instruction. The target must be within -254
    to +256 bytes from the location of the JGT instruction.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

    No status bits are affected.
    The A> status bit is analyzed.

Notes: The JGT instruction can be used to check the state of the Arithmetic
    Greater Than status bit resulting from a previous instruction's
    operation. The JGT can be used to evaluate the arithmetic (signed)
    result of an operation.

    Example: JGT GALLOP

    The JGT instruction jumps to the instruction labeled GALLOP if the
    Arithmetic Greater Than status bit is one.

Machine Code:

```
    Hex      <--- 1 ---> <--- 5 ---> <--- ~ ---> <--- ~ --->
             0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary | 0| 0| 0| 1| 0| 1| 0| 1|  PC Word Displacement |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

                        Length: 1 word
```

Jump if High   JH

Mnemonic and Addressing Modes: JH Target

Result: If L> = 1, jump to target;
         otherwise, continue to following instruction

Operation: Jumps to the target instruction if the Logical Greater Than
    status bit is one. Otherwise, there is no jump and program control
    continues to the following instruction. The target must be within -254
    to +256 bytes from the location of the JH instruction.

Status Bits Affected:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

   No status bits are affected.
   The L> status bit is analyzed.

Notes: The JH instruction can be used to check the state of the Logical
    Greater Than status bit resulting from a previous instruction's
    operation. The JH can be used to evaluate the logical (unsigned) result
    of an operation.

    Example: JH GALLOP

    The JH instruction jumps to the instruction labeled GALLOP if the
    Logical Greater Than status bit is one.

Machine Code:

```
Hex        <--- 1 --->  <--- B --->  <--- - --->  <--- - --->
           0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary    | 0| 0| 0| 1| 1| 0| 1| 1|   PC Word Displacement |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                         Length: 1 word

Jump if High or Equal   JHE

Mnemonic and Addressing Modes: JHE   Target

Result: If L> = 1 or EQ = 1, jump to target;
          otherwise, continue to following instruction

Operation: Jumps to the target instruction if the Logical Greater Than or
     Equal status bit is one. Otherwise, there is no jump and program
     control continues to the following instruction. The target must be
     within -254 to +256 bytes from the location of the JHE instruction.

Status Bits Affected:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
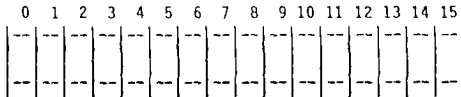
     No status bits are affected.
     The L> and EQ status bits are analyzed.

Notes: The JHE instruction can be used to check the state of the Logical
     Greater Than and Equal status bits resulting from a previous
     instruction's operation. The JHE can be used to evaluate the logical
     (unsigned) result of an operation.

     Example: JHE   GALLOP

     The JHE instruction jumps to the instruction labeled GALLOP if the
     Logical Greater Than or Equal status bit is set to one.

Machine Code:

```
Hex        <--- 1 --->  <--- 4 --->  <--- - --->  <--- - --->
           0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary    | 0| 0| 0| 1| 0| 1| 0| 0|  PC Word Displacement  |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
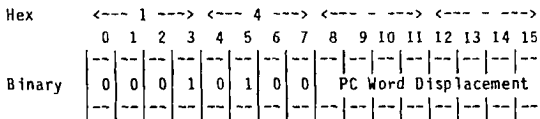
                          Length: 1 word

Jump if Low   JL

Mnemonic and Addressing Modes: JL   Target

Result: If L> = 0 and EQ = 0, jump to target;
          otherwise, continue to following instruction

Operation: Jumps to the target instruction if the Logical Greater Than
   status bit is zero and the Equal status bit is zero. Otherwise, there
   is no jump and program control cor' .es to the following instruction.
   The target must be within -254 to -. · bytes from the location of the JL
   instruction.

Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 1D 11 12 13 14 15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
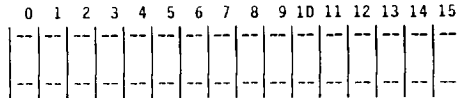
No status bits are affected.
The L> and EQ status bits are analyzed.

Notes: The JL instruction can be used to check the state of the Logical
   Greater Than and Equal status bits resulting from a previous
   instruction's operation. The JL can be used to evaluate the logical
   (unsigned) result of an operation.

   Example: JL   GALLOP

   The JL instruction jumps to the instruction labeled GALLOP if the
   Logical Greater Than status bit is zero and the Equal status bit is zero.

Machine Code:

```
   Hex        <--- 1 ---> <--- A ---> <--- - ---> <--- - --->
              0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary   | D| D| 0| 1| 1| D| 1| 0|  PC Word Displacement  |
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

                          Length: 1 word
```
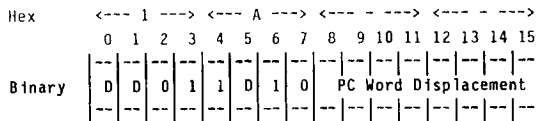
Jump if Low or Equal   JLE

Mnemonic and Addressing Modes: JLE   Target

Result: If L> = 0 or EQ = 1, jump to target;
          otherwise, continue to following instruction

Operation: Jumps to the target instruction if the Logical Greater Than
             status bit is zero or the Equal status bit is one. Otherwise, there is
             no jump and program control continues to the following instruction. The
             target must be within -254 to +256 bytes from the location of the JLE
             instruction.

Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

   No status bits are affected.
   The L> and EQ status bits are analyzed.

Notes: The JLE instruction can be used to check the state of the Logical
         Greater Than and Equal status bits resulting from a previous
         instruction's operation. The JLE can be used to evaluate the logical
         (unsigned) result of an operation.

   Example: JLE   GALLOP

   The JLE instruction jumps to the instruction labeled GALLOP if the
   Logical Greater Than status bit is zero or if the Equal status bit is one.

Machine Code:

```
 Hex        <--- 1 --->  <--- 2 --->  <--- - --->  <--- - --->
            0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 Binary   | 0| 0| 0| 1| 0| 0| 1| 0|  PC Word Displacement  |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                        Length: 1 word

285
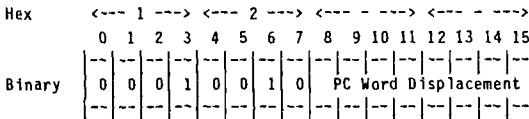
Jump if Less Than   JLT

Mnemonic and Addressing Modes: JLT Target

Result: If A> = 0 and EQ = 0, jump to target;
          otherwise, continue to following instruction

Operation: Jumps to the target instruction if the Arithmetic Greater Than
      status bit is zero and the Equal status bit is zero. Otherwise, there
      is no jump and program control continues to the following instruction.
      The target must be within -254 to +256 bytes from the location of the
      JLT instruction.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

      No status bits are affected.
      The A> and EQ status bits are analyzed.

Notes: The JLT instruction can be used to check the state of the Arithmetic
      Greater Than and Equal status bits resulting from a previous
      instruction's operation.   The JLT can be used to evaluate the arithmetic
      (signed) result of an operation.

      Example: JLT GALLOP

      The JLT instruction jumps to the instruction labeled GALLOP if the
      Arithmetic Greater Than status bit is zero and the Equal status bit is
      zero.

Machine Code:

```
Hex       <--- 1 ---> <--- 1 ---> <--- - ---> <--- - --->
          0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary  | D| 0| D| 1| D| 0| 0| 1|  PC Word Displacement  |
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                        Length: 1 word

Jump Unconditional   JMP

Mnemonic and Addressing Modes: JMP Target

Result: Jump to target

Operation: Jumps to the target instruction. The target must be within -254
    to +256 bytes from the location of the JMP instruction.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

    No status bits are affected.
    No status bit are analyzed.

Notes: The JMP instruction performs a short-range transfer of program
    control. The JMP instruction peforms an unconditional transfer of
    control as does the B instruction. The JMP instruction, however,
    requires only one word of machine code and the B instruction may require
    two words.

    Example: JMP   GALLOP

    The JMP instruction jumps to the instruction labeled GALLOP.

Machine Code:

```
    Hex       <--- 1 --->  <--- 0 --->  <--- - --->  <--- - --->
              0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary  | 0| 0| 0| 1| 0| 0| 0| 0|  PC Word Displacement  |
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                         Length: 1 word

`

Jump if No Carry   JNC

Mnemonic and Addressing Modes: JNC   Target

Result: If CY = 0, jump to target
          If CY = 1, continue to following instruction

Operation: Jumps to the target instruction if the Carry status bit is zero.
   Otherwise, there is no jump and program control continues to the
   following instruction. The target must be within -254 to +256 bytes
   from the location of the JNC instruction.

Status Bits Affected:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
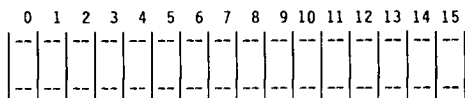
   No status bits are affected.
   The CY status bit is analyzed.

Notes: The JNC instruction can be used to check the state of the Carry
   status bit resulting from a previous instruction's operation.

   Example: JNC   GALLOP

   The JNC instruction jumps to the instruction labeled GALLOP if the Carry
   status bit is zero.

Machine Code:

```
Hex      <--- 1 ---> <--- 7 ---> <--- - ---> <--- - --->
          0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary   | 0| 0| 0| 1| 0| 1| 1| 1|   PC Word Displacement |
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
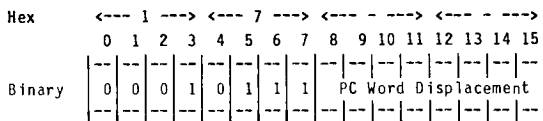
                          Length: 1 word

Jump if Not Equal   JNE

Mnemonic and Addressing Modes: JNE   Target

Result: If EQ = 0, jump to target
          If EQ = 1, continue to following instruction

Operation: Jumps to the target instruction if the Equal status bit is zero.
   Otherwise, there is no jump and program control continues to the
   following instruction. The target must be within -254 to +256 bytes
   from the location of the JNE instruction.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

   No status bits are affected.
   The EQ status bit is analyzed.

Notes: The JNE instruction can be used to check for the state of the Equal
   status bit resulting from a previous instruction's operation.   It's
   often used at the end of a loop to determine if the loop count is zero.

   Example: JNE   GALLOP

   The JNE instruction jumps to the instruction labeled GALLOP if the Equal
   status bit is zero.

Machine Code:

```
   Hex        <--- 1 ---> <--- 6 ---> <--- - ---> <--- - --->
              0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary    | 0| 0| 0| 1| 0| 1| 1| 0|  PC Word Displacement |
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
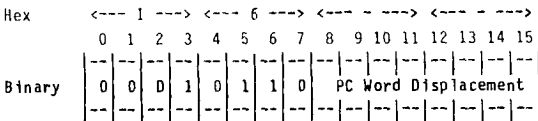
                        Length: 1 word

Jump if No Overflow   JNO

Mnemonic and Addressing Modes: JNO   Target

Result: If OV = 0, jump to target
          If OV = 1, continue to following instruction

Operation: Jumps to the target instruction if the Overflow status bit is
     zero. Otherwise, there is no jump and program control continues to the
     following instruction. The target must be within -254 to +256 bytes
     from the location of the JNO instruction.

Status Bits Affected:

```
  0  1  2  3  4  5  6  7  8  9 10 1I 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
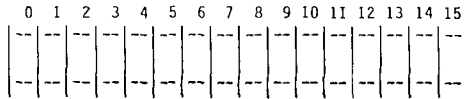
     No status bits are affected.
     The OV status bit is analyzed.

Notes: The JNO instruction can be used to check the state of the Overflow
     status bit resulting from a previous instruction's operation.

     Example: JNO   GALLOP

     The JNO instruction jumps to the instruction labeled GALLOP if the
     Overflow status bit is zero.

Machine Code:

```
Hex       <--- 1 ---> <--- 9 ---> <--- - ---> <--- - --->
          0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary   | 0| 0| 0| 1| 1| 0| 0| 1|  PC Word Displacement  |
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

                      Length: 1 word
```
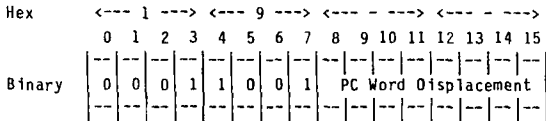
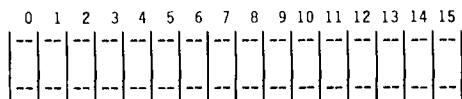Jump On Carry   JOC

Mnemonic and Addressing Modes: JOC   Target

Result: If CY = 1, jump to target
          If CY = 0, continue to following instruction

Operation: Jumps to the target instruction if the Carry status bit is one.
     Otherwise, there is no jump and program control continues to the
     following instruction. The target must be within -254 to +256 bytes
     from the location of the JOC instruction.
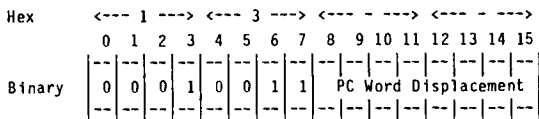

Status Bits Affected:

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
    |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    |                                                |
    |                                                |
    |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
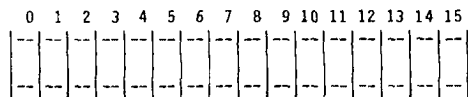
     No status bits are affected.
     The CY status bit is analyzed.

Notes: The JOC instruction can be used to check the state of the Carry
     status bit resulting from a previous instruction's operation.

     Example: JOC GALLOP

     The JOC instruction jumps to the instruction labeled GALLOP if the Carry
     status bit is one.

Machine Code:

```
    Hex        <--- 1 --->  <--- 8 --->  <--- - --->  <--- - --->
               0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
              |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary    | 0| 0| 0| 1| 1| 0| 0| 0|  PC Word Displacement |
              |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                         Length: 1 word

291
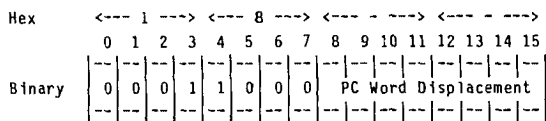
Jump if Odd Parity   JOP
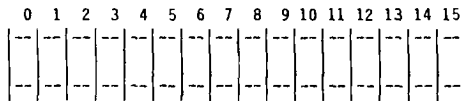

Mnemonic and Addressing Modes: JOP   Target

Result: If OP = 1, jump to target
          If OP = 0, continue to following instruction

Operation: Jumps to the target instruction if the Odd Parity status bit is
             one. Otherwise, there is no jump and program control continues to the
             following instruction. The target must be within −254 to +256 bytes
             from the location of the JOP instruction.
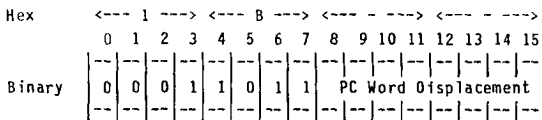
Status Bits Affected:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

          No status bits are affected.
          The OP status bit is analyzed.

Notes: The JOP instruction can be used to check the state of the Odd Parity
         status bit resulting from a previous instruction's operation.

         Example: JOP   GALLOP

         The JOP instruction jumps to the instruction labeled GALLOP if the Odd
         Parity status bit is one.

Machine Code:

```
Hex       <--- 1 ---> <--- C ---> <--- - ---> <--- - --->
           0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary    | 0| 0| 0| 1| 1| 1| 0| 0|  PC Word Displacement |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
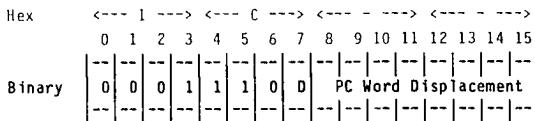
                          Length: 1 word

Load Communicaton Register Unit   LDCR

Mnemonic and Addressing Modes: LDCR   S,C

Result: (S) --> The number of CRU selected bit addresses
          determined by C.

Operation: Sends out the r   · r of bits specified by C from the source
            operand to consecutive   bit addresses. If C is zero, 16 bits are
            sent out. If the number or bits sent out is greater that 8, the source
            operand is a word address; otherwise, it's a byte address. The bits
            sent out from the contents of the word or byte addressed by the source
            operand are sent out from right to left (the rightmost bit in the word
            or byte is sent out first). The CRU hardware base address (the contents
            of R12, bits 3 through 14) selects the first CRU bit address and
            subsequent bits sent out go to the subsequent CRU bit addresses. The
            contents of R12 and the source   ·· and remain unchanged. The bits are
            sent out sequentially on the C   line.

Status Bits Affected:
     0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
    |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
L>	A>	EQ			OP										

    The entire content of the source operand (not just the
    transferred bits) is compared to zero and the Logical
    Greater Than, Arithmetic Greater Than, and Equal status
    bits are affected accordingly. If the source operand is
    a byte address, the Odd Parity status bit is affected
    based upon the number of one bits in the contents of the
    source operand.

Notes: The LDCR instruction sends out a multiple number of bits (up to a
        maximum of 16) to a series of CRU bit addresses.

    Example: LI   R3,>1E7F
                  LI   R12,>200
                  LDCR R3,0
    Sends out 16 bits from the contents of Register 3 to CRU bit addresses
    hexadecimal 100 through 10F.
                              L> = 1, A> = 1, EQ = 0

Machine Code:
    Hex   <--- 3 ---> <--- - ---> <--- - ---> <--- - --->
              0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary | 0| 0| 1| 1| 0| 0| C | T s | R s |
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
             | Source Address |
             | - |

                     Length: 1 or 2 words

Load Immediate   LI

Mnemonic and Addressing Modes: LI   R,IOP

Result: IOP --> (R)

Operation: Places the I6-bit immediate operand into the contents of the
    register.

Status Bits Affected:

```
 0  I  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|L>|A>|EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
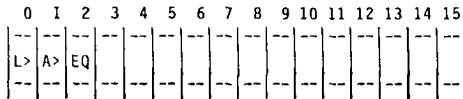
    The immediate operand is compared to zero and the Logical
    Greater Than, Arithmetic Greater Than, and Equal status
    bits are affected accordingly.

Notes: The LI instruction is used often to initialize a register with a
    constant; for example, a loop counter or an address to be used with
    indirect or indexed addressing.

    Example: LI   R9,100

                 Before   After

    (R9) =   >BC5A   >0064

                               L> = 1, A> = 1, EQ = 0

Machine Code:

```
Hex        <--- 0 ---> <--- 2 ---> <--- 0 ---> <--- ~ --->
            0  I  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary     | 0| 0| 0| 0| 0| 0| 1| 0| 0| 0| 0| 0|     R     |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                          Immediate Operand
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                       Length: 2 words

Load Interrupt Mask Immediate   LIMI

Mnemonic and Addressing Modes: LIMI IOP

Result: IOP Bits 12 through 15 --> (ST) Bits 12 through 15

Operation: Replaces the contents of the Interrupt Mask (the rightmost four
    bits of the Status Register) with bits 12 through 15 (the rightmost
    nibble) of the immediate operand.

Status Bits Affected:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |I0|I1|I2|I3|
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

    The interrupt mask is set to the value of the right
    most nibble of the immediate operand.

Notes: The LIMI instruction is used to set the interrupt mask to a specific
    number. This number is used by the computer to determine which
    interrupt signals are allowed to cause an interrupt and which ones are
    not. When an interrupt signal occurs, the number of the interrupting
    device (called its interrupt "level") is compared to the number in the
    interrupt mask. The interrupt level must be less than or equal to the
    number in the interrupt mask before the device is allowed to interrupt
    the current program. (A level 0 device is always allowed to interrupt.)

    Example: LIMI 6

    The number 6 is placed in the interrupt mask. Only devices with
    interrupt levels of 0 through 6 are allowed to interrupt the current
    program.

Machine Code:

```
  Hex      <--- 0 ---> <--- 3 ---> <--- 0 ---> <--- 0 --->
           0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  Binary | 0| 0| 0| 0| 0| 0| 1| 1| 0| 0| 0| 0| 0| 0| 0| 0|
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                          Immediate Operand
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
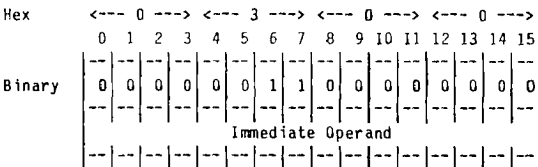
                    Length: 2 words

Load or Restart Execution   LREX

Mnemonic and Addressing Modes: LREX

Result: Sends signals out on address lines and CRUOUT line.

Operation: When the LREX instruction is performed, the binary values 1, 1,
    and 1 appear on address lines A0, A1, and A2, respectively, in
    conjunction with a synchronizing pulse on the CRUCLK line.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

No status bits are affected.

Notes: The LREX instruction may be used to implement functions unique to
    hardware surrounding the central processor.

Machine Code:

```
Hex       <--- 0 --->  <--- 3 --->  <--- E --->  <--- 0 --->
          0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary  | 0| 0| 0| 0| 0| 0| 1| 1| 1| 1| 1| 0| 0| 0| 0| 0|
        |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                        Length: 1 word

Load Workspace Pointer Immediate   LWPI

Mnemonic and Addressing Modes: LWPI IOP

Result: IOP --> (WP)

Operation: Replaces the contents of the Workspace Pointer with the
    immediate operand.

Status Bits Affected:

```
    0  I  2  3  4  5  6  7  8  9 I0 II I2 13 14 15
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
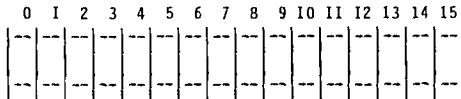
    No status bits are affected.

Notes: The LWPI instruction is used to define the area of memory for a
    program to use as its workspace. The immediate operand is placed into
    the Workspace Pointer. This operand should be the address of the first
    word of the workspace (the address of Register 0). The computer uses
    the next I5 contiguous words in memory as Registers 1 through 15.

    Example: LWPI WSP

    The address value of WSP is placed in the Workspace Pointer.

    A BSS directive can be used to reserve the 32-byte (16-word) workspace.

    Example: WSP   BSS 32

Machine Code:

```
    Hex      <--- 0 ---> <--- 2 ---> <--- E ---> <--- 0 --->
             0  1  2  3  4  5  6  7  B  9 10 11 I2 13 I4 15
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary  | 0| 0| 0| 0| 0| 0| 1| 0| 1| 1| 1| 0| 0| 0| 0| 0|
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                            Immediate Operand
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
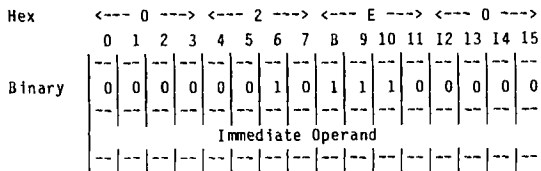
                     Length: 1 word

Move Word   MOV

Mnemonic and Addressing Modes: MOV S,D

Result: (S) --> (D)

Operation: Replaces the contents of the second operand with a copy of the
    contents of the first operand. Both operands are word addresses.

Status Bits Affected:

```
    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |L>|A>|EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
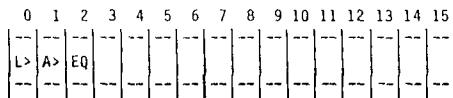
    The moved word is compared to zero and the Logical
    Greater Than, Arithmetic Greater Than, and Equal
    status bits are affected accordingly.

Notes: You can use the MOV instruction to copy a word from one location to
    another. It's often used to copy a word from one general memory
    location to another general memory location. It can be used to
    initialize the contents of an operand at the start of a program and to
    save the results of an operation.

    Example: MOV @ALPHA,R4

                 Before   After

    (ALPHA) =   >3A74   >3A74
    (R4)   =   >BC5A   >3A74

                              L> = 1, A> = 1, EQ = 0

Machine Code:

```
  Hex     <--- C ---> <--- - ---> <--- - ---> <--- - --->
           0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  Binary  | 1| 1| 0| 0| T d |  R d  | T s |  R s  |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
          |        Source (or Destination) Address        |
          - - - - - - - - - - - - - - - - - - - - - - - -
          |              Destination Address              |
          - - - - - - - - - - - - - - - - - - - - - - - -
```
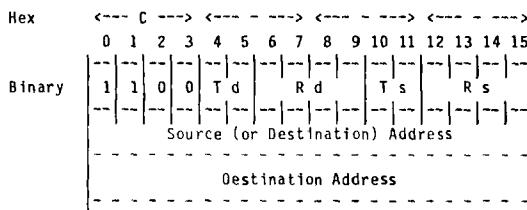
              Length: 1 or 2 or 3 words

Move Byte   MOVB

Mnemonic and Addressing Modes: MOVB S,D


```
          Byte
Result:  (S) ----> (D)
```

Operation: Replaces the contents of the second operand with a copy of the
    contents of the first operand. Both operands are byte addresses. The
    contents of the unaddressed byte in a word are unaffected.

Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |L>|A>|EQ|  |  |  |OP|  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

The moved byte is compared to zero and the Logical
Greater Than, Arithmetic Greater Than, and Equal
status bits are affected accordingly. The Odd Parity
status bit is set to one if there's an odd number of one
bits in the byte; otherwise, it's cleared to zero.

Notes: You can use the MOVB instruction to copy a byte from one location to
    another. It can be used to initialize the contents of an operand at the
    start of a program and to save the results of an operation.

    Example: MOVB @ALPHA(R1),R4

                 Before   After

    (ALPHA) =   >3A74   >3A74
    (R1)   =   >D001   >0001
    (R4)   =   >BC5A   >745A

                 L> = 1, A> = 1, EQ = 0, OP = 0

Machine Code:

```
   Hex      <--- D ---> <--- - ---> <--- - ---> <--- - --->
            0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary  | 1| 1| 0| 1| T d|   R d  | T s|   R s  |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
           |       Source (or Destination) Address       |
           |- - - - - - - - - - - - - - - - - - - - - - - |
           |             Destination Address             |
           |- - - - - - - - - - - - - - - - - - - - - - - |
```
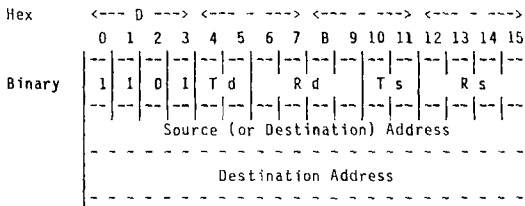
                 Length: 1 or 2 or 3 words

Multiply   MPY

Mnemonic and Addressing Modes: MPY S,R

Result: (S) * (R) --> (R and R+1)

Operation: Multiplies the contents of the first operand by the contents of
     the second operand. The most significant 16-bits of the 32-bit product
     replaces the contents of the destination register and the least
     significant 16 bits of the product replaces the contents of the next
     register. If Register 15 is the destination register, the least
     significant 16 bits of the product replaces the contents of the word
     following Register 15. The numbers are treated as unsigned values.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
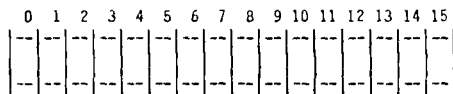
     No status bits affected.

Notes: The Multiply instruction multiplies two 16-bit numbers and produces
     a 32-bit product. It's an unsigned multiplication; the computer ignores
     the sign of the numbers.

     Example: MPY @FACTOR,R5

                      Before   After

     (FACTOR) =   >0023   >0023
     (R5)   =   >0005   >0000
     (R6)   =   >A687   >00AF

                            No status bits affected.

Machine Code:

```
   Hex      <--- 3 --->  <--- - --->  <--- - --->  <--- - --->
            0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary  | 0| 0| 1| 1| 1| 0|    R    | T s|    R s   |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
           |            Source Address                      |
           | - - - - - - - - - - - - - - - - - - - - - - - -|
```
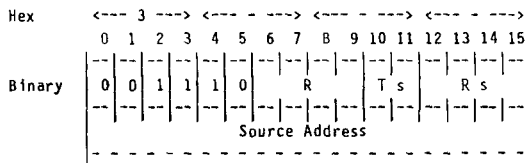
                  Length: 1 or 2 words

Negate   NEG

Mnemonic and Addressing Modes: NEG S

Result: ~(S) --> (S)

Operation: Negates (forms the two's complement of) the contents of the
    operand.

Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |L>|A>|EQ|CY|OV|  |  |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
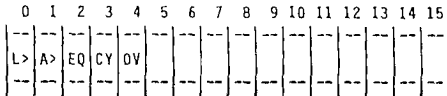
    The computer negates the contents of the operand by
    taking the one's complement of the contents and adding
    one. The addition affects the Carry and Overflow status
    bits. The result is compared to zero and the Logical
    Greater Than, Arithmetic Greater Than, and Equal status
    bits are affected accordingly.

Notes: The Negate instruction negates a 16-bit number.

    Example: NEG @DTFLAG

                 Before   After

    (DTFLAG) =   >0005   >FFFB

                               L> = 1, A> = 0, EQ = 0
                               CY = 0, OV = 0

Machine Code:

    Hex   <--- 0 ---> <--- 5 ---> <--- - ---> <--- - --->
             0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary | 0| 0| 0| 0| 0| 1| 0| 1| 0| 0| T s| R s|
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
           | Source Address |
           |- -|
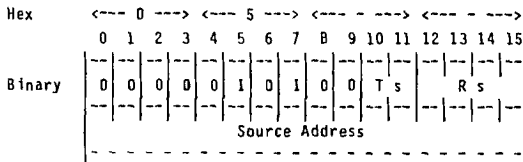
                 Length: 1 or 2 words
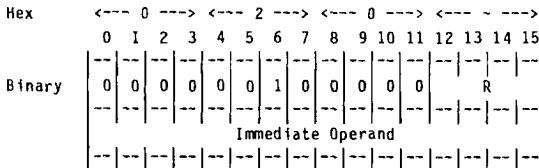
Or Immediate   ORI

Mnemonic and Addressing Modes: ORI   R,IOP

Result: (R) OR IOP --> (R)

Operation: Performs a logical OR operation between the bits in the register
    and the bits in the immediate operand. The result replaces the contents
    of the register.

Status Bits Affected:

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   |L>|A>|EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
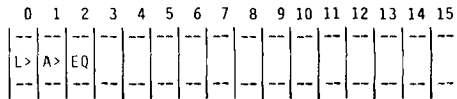
    The result is compared to zero and the Logical Greater
    Than, Arithmetic Greater Than, and Equal status bits are
    affected accordingly.

Notes: The ORI instruction is useful for forcing selected bits in a
    register to one.

    Example: ORI   R0,>A60F

                  Before   After

    (R0)   =   >BC5A   >BE5F

                                L> = 1, A> = 0, EQ = 0

    (R0) Before = 1011 1100 0101 1010
    IOP   = 1010 0110 0000 1111

    (R0) After = 1011 1110 0101 1111

Machine Code:

```
   Hex        <--- 0 ---> <--- 2 ---> <--- 6 ---> <--- - --->
               0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary    | 0| 0| 0| 0| 0| 0| 1| 0| 0| 1| 1| 0|     R      |
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
             |                 Immediate Operand              |
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
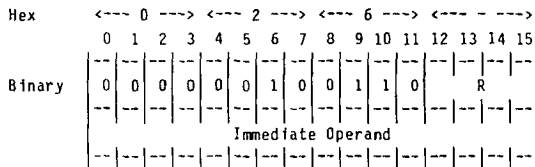
                      Length: 2 words

Reset   RSET

Mnemonic and Addressing Modes: RSET

Result: Force interrupt mask to zero.
          Sends signals out on address lines and CRUOUT line.

Operation: Forces the interrupt mask (the rightmost nibble in the Status
             Register) to zero. When the RSET instructionn is performed, the binary
             values 0, 1, and I appear on address lines A0, A1, and A2, respectively,
             in conjunction with a synchronizing pulse on the CRUCLK line.

Status Bits Affected:

```
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|   |   |   |   |   |   |   |   |   |   |   |   |I0 |I1 |I2 |I3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
```
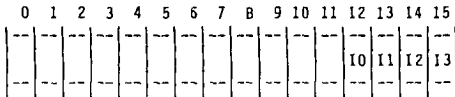
          The interrupt mask (status bits 12 through 15) are
          forced to zero.

Notes: The RSET instruction may be used to implement functions unique to
         hardware surrounding the central processor.

Machine Code:

```
Hex       <--- 0 --->  <--- 3 --->  <--- 6 --->  <--- 0 --->
          0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
         |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
Binary   | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
         |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
```

                              Length: 1 word

303
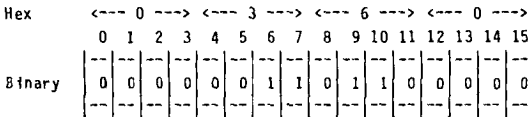
Return with Workspace Pointer   RTWP

Mnemonic and Addressing Modes: RTWP

Result: (R13) --> (WP)
          (R14) --> (PC)
          (R15) --> (SR)

Operation: Reverses a context switch. The contents of the Workspace
     Pointer, Program Counter, and Status Register are replaced with the
     contents of Registers 13, 14, and 15, respectively, of the current
     workspace.

Status Bits Affected:

```
      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
    |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    |L>|A>|EQ|CY|OV|OP|X |  |  |  |  |  |I0|I1|I2|I3|
    |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

     The contents of the Status Register is replaced with
     the contents of Register 15.

Notes: The RTWP instruction is normally used as the last instruction
     performed by a subroutine which is called by a context switch (by an XOP
     or BLWP instruction or by an interrupt).

```
    Example:     Calling Program              Called Program
                       :                            :
                 BLWP  @SQVECT  -------+     SUBWSP BSS  32
                 -------------   <--+   |            :
                       :          |   |            :
                       :          |   +---> SUBENT -------
                                  |                :
             SQVECT DATA  -       |                :
                    DATA   `.     +-------        RTWP
```
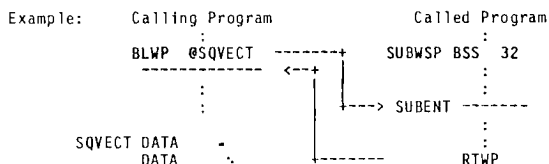
Machine Code:

```
    Hex      <--- 0 ---> <--- 3 ---> <--- B ---> <--- 0 --->
             0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary | 0| 0| 0| 0| 0| 0| 1| 1| 1| 0| 0| 0| 0| 0| 0| 0|
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                        Length: 1 word

Subtract Words   S

Mnemonic and Addressing Modes: S   S,D

Result: (D) - (S) --> (D)

Operation: Subtracts the contents of the first operand from the contents of
    the second operand. Replaces the contents of the second operand with
    the difference.   Both operands are word addresses.

Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |L>|A>|EQ|CY|OV|  |  |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
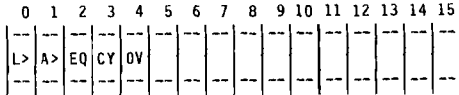
The computer adds the two's complement of the contents of
the first operand to the contents of the second operand.
The addition affects the Carry and Overflow status bits.
The sum is compared to zero and the Logical Greater Than,
Arithmetic Greater Than, and Equal status bits are
affected accordingly.

Notes: The Subtract Words instruction subtracts one 16-bit number from
    another.

    Example: S   R4,@ALPHA

                  Before   After

    (R4)   =   >07A4   >07A4
    (ALPHA) =   >BC5A   >B4B6

                                L> = 1, A> = 0, EQ = 0
                                CY = 1, OV = 0

Machine Code:

```
 Hex     <--- 6 ---> <--- - ---> <--- - ---> <--- - --->
          0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 Binary  | 0| 1| 1| 0| T  d |  R  d   |  T  s |   R  s  |
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
         |          Source (or Destination) Address      |
         |- - - - - - - - - - - - - - - - - - - - - - - -|
         |                                               |
         |              Destination Address              |
         |- - - - - - - - - - - - - - - - - - - - - - - -|
```
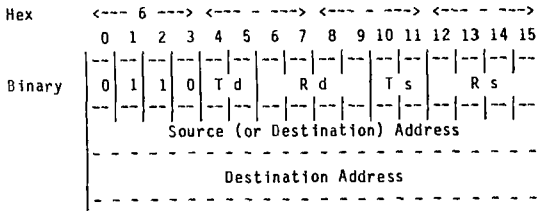
                  Length: 1 or 2 or 3 words

Subtract Bytes   SB

Mnemonic and Addressing Modes: SB   S,D

                    Bytes
Result: (D) - (S) -----> (D)

Operation: Subtracts the contents of the first operand from the contents of
    the second operand.   Replaces the contents of the second operand with
    the difference.   Both operands are byte addresses.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   |L>|A>|EQ|CY|OV|OP|  |  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
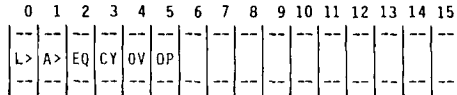
    The computer adds the two's complement of the contents of
    the first operand to the contents of the second operand.
    The addition affects the Carry and Overflow status bits.
    The sum is compared to zero and the Logical Greater Than,
    Arithmetic Greater Than, and Equal status bits are
    affected accordingly.   The OP status bit is set to one if
    there's an odd number of one bits in the sum; otherwise;
    it's cleared to zero.

Notes: The Subtract Bytes instruction subtracts one B-bit
    number from another.

    Example: SB   R4,@ALPHA

                  Before   After

    (R4)   =   >07A4   >07A4
    (ALPHA) =   >BC5A   >B55A

                                L> = 1, A> = 0, EQ = 0
                                CY = 1, OV = 0
Machine Code:
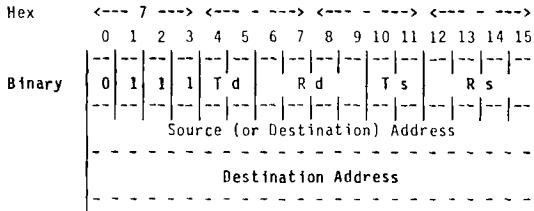
```
    Hex      <--- 7 ---> <--- - ---> <--- - ---> <--- - --->
             0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary  | 0  1  1  1| T d|   R d  | T s|   R s  |
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
            |         Source (or Destination) Address        |
            |- - - - - - - - - - - - - - - - - - - - - - - - |
            |              Destination Address               |
            |- - - - - - - - - - - - - - - - - - - - - - - - |
                     Length: 1 or 2 or 3 words
```

Set Bit to One   SBO

Mnemonic and Addressing Modes: SBO Displacement

Result: One bit --> Selected CRU Bit Address

Operation: Sends a one bit to a selected CRU bit address. The bit address
   is the 12-bit sum of the CRU hardware base address (the contents of R12,
   bits 3 through 14) and the displacement operand (a number in the range
   of -128 through +127). The 12-bit sum appears on the computer's address
   lines A3 through A14. Address lines A0 through A2 are forced to zero.
   The one bit goes out on the CRUDUT line in conjunction with a
   synchronizing pulse on the CRUCLK line.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

   No status bits are affected.

Notes: The SBO instruction sets a CRU device at the selected address to a
   logic one.

   Example: LI   R12,>400
                SBO 1B

   Sets CRU bit address hexadecimal 212 to a logic one. No status bits are
   affected.

Machine Code:

```
   Hex       <--- 1 --->  <--- D --->  <--- - --->  <--- - --->
             0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary  | 0| 0| 0| 1| 1| 1| 0| 1|       Displacement      |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
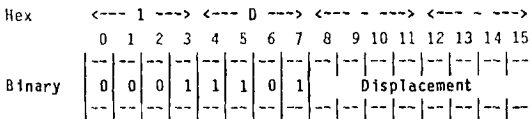
                         Length: 1 word

Set Bit to Zero   SBZ

Mnemonic and Addressing Modes: SBZ Displacement

Result: Zero bit --> Selected CRU Bit Address

Operation: Sends a zero bit to a selected CRU bit address. The bit address
is the 12-bit sum of the CRU hardware base address (the contents of R12,
bits 3 through 14) and the displacement operand (a number in the range
of -128 through +127). The 12-bit sum appears on the computer's address
lines A3 through A14. Address lines A0 through A2 are forced to zero.
The zero bit goes out on the CRUOUT line in conjunction with a
synchronizing pulse on the CRUCLK line.

Status Bits Affected:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
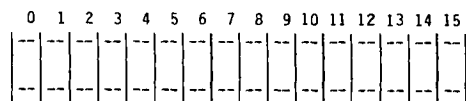
No status bits are affected.

Notes: The SBZ instruction sets a CRU device at the selected address to a
logic zero.

Example: LI   R12,>300
              SBZ -18

Sets CRU bit address hexadecimal 16E to a logic zero. No status bits
are affected.

Machine Code:

```
Hex        <--- 1 ---> <--- E ---> <--- - ---> <--- - --->
            0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary     | 0| 0| 0| 1| 1| 1| 1| 0|    Displacement      |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

                       Length: 1 word
```

Set to Ones   SETO

Mnemonic and Addressing Modes: SETO S

Result: >FFFF --> (S)

Operation: Forces the content of the operand to hexadecimal FFFF.

Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

   No status bits are affected.

Notes: The SETO instruction initializes all the bits in a word to ones.
         This value is sometimes used as a special marker to mark the end of a
         table *or* sometimes used for a flag condition. When considered as a
         signed number, hexadecimal FFFF is -1.

   Example: SETO @ZFFLAG

                Before   After

   (ZFFLAG) =   >72E9   >FFFF

                            No status bits affected.

Machine Code:

```
   Hex       <--- 0 ---> <--- 7 ---> <--- - ---> <--- - --->
             0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary  | 0| 0| 0| 0| 0| 1| 1| 1| 0| 0| T s|     R s    |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
           |              Source Address                    |
           |- - - - - - - - - - - - - - - - - - - - - - - - |
```
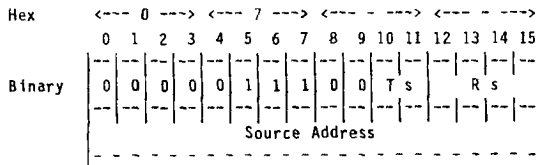
                        Length: 1 or 2 words

Shift Left Arithmetic   SLA

Mnemonic and Addressing Modes: SLA R,C

```
Result:          <--- C  places ----
                 0  1          14 15
                 +--+--+-- / --+--+--+
         (R) =  | X| X| . . . | X| X|  <--- 0
                 +--+--+-- / --+--+--+
```

Operation: Shifts the contents of the register to the left by the number of
    bit positions specified by the count, C. If C is 0, the shift count is
    specified by the number in the rightmost nibble of Register 0. If C is
    0 and the rightmost nible of register 0 is zero, the bits in the
    register are shifted 16 posistins.   Fills the vacated bit positions
    with zeros. The state of the last bit shifted out is recorded in the
    Carry status bit.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   |L>|A>|EQ|CY|OV|  |  |  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

    The result is compared to zero and the Logical Greater
    Than, Arithmetic Greater Than, and Equal status bits are
    affected accordingly. The Carry status bit is a copy of
    the last bit shifted out. The Overflow status bit is set
    to one if the sign bit (bit position 0) changes anytime
    during the shift; otherwise, it's cleared to zero.

Notes: The SLA instruction can be used to perform a multiplication by 2.
    In order for the result to be correct, the sign bit should not change
    during the shift.

    Example: SLA R8,3

                 Before   After

      (R8) =   >9A74   >D3A0

                              L> = 1, A> = 0, EQ = 0,
                              CY = 0, OV = 1
Machine Code:

```
Hex        <--- 0 --->  <--- A --->  <--- - --->  <--- - --->
           0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary    | 0| 0| 0| 0| 1| 0| 1| 0|      C     |      R     |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                            Length: 1 word
```

Set Ones Corresponding   SOC

Mnemonic and Addressing Modes: SOC   S,D

Result: (S) OR (D) --> (D)

Operation: Sets to one all those bits in the contents of the destination
     operand that correspond to the position of one bits in the source
     operand.   Leaves unchanged those bits in the contents of the destination
     operand that correspond to the position of zero bits in the source
     operand.   This is a logical OR operation between the bits in the
     contents of the source operand and the bits in the contents of the
     destinaion operand. Both operands are word addresses.

Status Bits Affected:

```
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|L> |A> |EQ |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
```

     The result is compared to zero and the Logical Greater
     Than, Arithmetic Greater Than, and Equal status bits are
     affected accordingly.

Notes: The SOC instruction can be used to force selected bits in a word to
     one.   The computer ORs the contents of the source operand with the
     contents of the destination operand.

     Example: SOC   @BITMSK,R1

                        Before   After

     (BITMSK) =   >A6OF   ·ˉ
     (R1)   =   >BC5A

                                   L> = 1, A> = 0, EQ = 0

     (BITMSK)   =   1010 0110 0000 1111
     (R1)   Before   =   1011 1100 0101 1010

     (R1)   After   =   1011 1110 0101 1111

Machine Code:

```
  Hex      <--- E ---> <--- - ---> <--- - ---> <--- - --->
           0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
          |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
  Binary  | 1 | 1 | 1 | 0| T d |   R d     | T s |   R s     |
          |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
          |       Source (or Destination) Address          |
          - - - - - - - - - - - - - - - - - - - - - - - - -
          |            Destination Address                 |
          - - - - - - - - - - - - - - - - - - - - - - - - -
               Length: 1 or 2 or 3 words
```
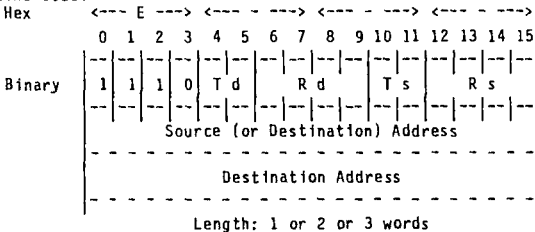
Set Ones Corresponding Byte   SOCB

Mnemonic and Addressing Modes: SOCB   S,D

Result: (S) OR (D) --> (D)

Operation: Sets to one all those bits in the contents of the destination
   operand that correspond to the position of one bits in the source
   operand. Leaves unchanged those bits in the contents of the destination
   operand that correspond to the position of zero bits in the source
   operand. This is a logical OR operation between the bits in the
   contents of the source operand and the bits in the contents of the
   destination operand. Both operands are byte addresses.

Status Bits Affected:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|L>|A>|EQ|  |  |OP|  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

   The result is compared to zero and the Logical Greater
   Than, Arithmetic Greater Than, and Equal status bits are
   affected accordingly. The Odd Parity status bit is set
   if the byte result contains an odd number of one bits;
   otherwise, it's cleared to zero.

Notes: The SOCB instruction can be used to force selected bits in a byte to
   one. The computer ORs the contents of the source operand with the
   contents of the destination operand.

   Example: SOCB @BITMSK,R1
                 Before   After
   (BITMSK) =   >A60F   : ·
   (R1)   =   >BC5A   >BE5A
                                 L> = 1, A> = 0, EQ = 0, OP = 0

   (BITMSK)   =   1010 0110 0000 1111
   (R1)   Before =   1011 1100 0101 1010

   (R1)   After   =   1011 1110 0101 1010
                                 <- Not ->
                                 changed

Machine Code:

```
Hex        <--- F --->  <--- - --->  <--- - --->  <--- - --->
           0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary    | 1| 1| 1| 1| T d |  R d  | T s |  R s  |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
          |        Source (or Destination) Address        |
          |- - - - - - - - - - - - - - - - - - - - - - - -|
          |            Destination Address               |
          |- - - - - - - - - - - - - - - - - - - - - - - -|
              Length: 1 or 2 or 3 words
```

Shift Right Arithmetic   SRA

Mnemonic and Addressing Modes: SRA R,C
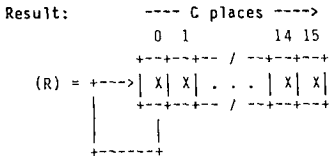
Result: ---- C places ---->
                    0 1   14 15
                    +--+--+-- / --+--+--+
    (R) = +--->| X| X| . . . | X| X|
              | +--+--+-- / --+--+--+
              | |
              +-~----+

Operation: Shifts the contents of the register to the right by the number
    of bit positions specified by the count, C. If C is 0, the shift count
    is specified by the number in the rightmost nibble of Register 0. If C
    is 0 and the rightmost nibble of register 0 is zero, the bits in the
    register are shifted 16 positions. The vacated bit positions are filled
    with a copy of the sign bit (bit 0). The state of the last bit shifted
    out is recorded in the Carry status bit.

Status Bits Affected:

     0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
    |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
L>	A>	EQ	CY												
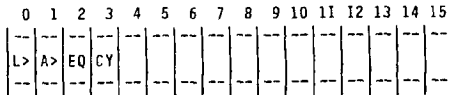
    The result is compared to zero and the Logical Greater
    Than, Arithmetic Greater Than, and Equal status bits are
    affected accordingly. The Carry status bit is a copy of
    the last bit shifted out.

Notes: The SRA instruction can be used to perform a division by 2 when the
    number in the register is an signed value.

    Example: SRA RB,2

                  Before   After

    (R8) =   >9A74   >E69D

                                L> = 1, A> = 0, EQ = 0, CY = 0

Machine Code:

    Hex   <--- 0 ---> <--- 8 ---> <--- - ---> <--- - --->
             0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary | 0| 0| 0| 0| 1| 0| 0| 0| C | R |
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                              Length: 1 word

                                                                        313

Shift Right Circular   SRC

Mnemonic and Addressing Modes: SRC R,C

Result: ---- C places ---->
                   0 1   14 15
                 +--+--+-- / --+--+--+
   (R) = +--->| X| X| . . . | X| X|---+
         | +--+--+-- / --+--+--+ |
         +--------------------------+

Operation: Shifts the contents of the register to the right by the number
            of bit positions specified by the count, C. If C is 0, the shift count
            is specified by the number in the rightmost nibble of Register 0. If C
            is 0 and the rightmost nibble of register 0 is zero, the bits in the
            register are shifted 16 positios. Each bit shifted out of bit position
            15 (the right end of the register) goes to bit position 0 (the left end
            of the register). The state of the last bit shifted out is recorded in
            the Carry status bit.

Status Bits Affected:

     0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
   |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
L>	A>	EQ	CY												

     The result is compared to zero and the Logical Greater
     Than, Arithmetic Greater Than, and Equal status bits are
     affected accordingly. The Carry status bit is a copy of
     the last bit shifted out.

Notes: The SRC instruction can be used to rearrange the position of bits in
        a word without changing their order. Notice that after a SRC
        instruction is performed, the sign bit (bit position 0) is the same as
        the state of the Carry status bit.

   Example: SRC R8,2

                 Before   After

     (R8) =   >9A74   >269D

                               L> = 1, A> = 1, EQ = 0, CY = 0
Machine Code:

   Hex   <--- D ---> <--- B ---> <--- - ---> <--- - --->
            0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary | 0| 0| 0| 0| 1| 0| 1| 1| C | R |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                            Length: 1 word

Shift Right Logical   SRL

Mnemonic and Addressing Modes: SRL   R,C

Result: ---- C places ---->
                  0 1   14 15
                 +--+--+-- / --+--+--+
   (R) = +--->| X| X| . . . | X| X|
                 +--+--+-- / --+--+--+

Operation: Shifts the contents of the register to the right by the number
   of bit positions specified by the count, C. If C is 0, the shift count
   is specified by the number in the rightmost nibble of Register 0. If C
   is 0 and the rightmost nibble of register 0 is zero, the bits in the
   register are shifted 16 positions. The vacated bit positions are filled
   with zeros. The state of the last bit shifted out is recorded in the
   Carry status bit.

Status Bits Affected:

   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
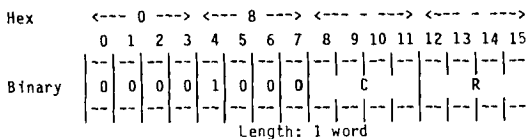 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
L>	A>	EQ	CY												

   The result is compared to zero and the Logical Greater
   Than, Arithmetic Greater Than, and Equal status bits are
   affected accordingly. The Carry status bit is a copy of
   the last bit shifted out.

Notes: The SRL instruction can be used to perform a division by 2 when the
   number in the register is an unsigned value. It's also useful in
   situations where each bit is a data item; for example, the state of a
   specific switch. You might use the SRL instruction to check the
   condition of the swithch by shifting its bit image into the Carry
   status bit.

   Example: SRL   R8,2

                Before   After

      (R8) =   >3A74   >0E9D

                L> = 1, A> = 0, EQ = 0, CY = 0

Machine Code:

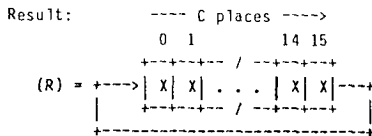   Hex   <--- 0 ---> <--- 9 ---> <--- - ---> <--- - --->
            0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary | 0| 0| 0| 0| 1| 0| 0| 1| C | R |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                              Length: 1 word

315
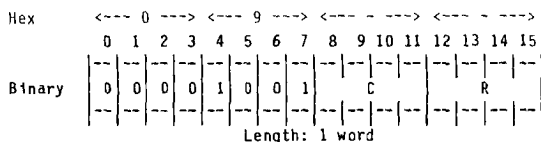
Store Communicaton Register Unit   STCR

Mnemonic and Addressing Modes: STCR S,C

Result: The number of CRU selected bit addresses determined
          by C --> (S)

Operation: Reads the number of bits specified by C into the source operand
            from consecutive CRU bit addresses. If C is zero, 16 bits are read in.
            If the number of bits specified is greater that 8, the source operand is
            a word address; otherwise, it's a byte address. The bits read into the
            contents of the word or byte addressed by the source operand fill the
            word or byte from right to left. Any unfilled bit positions in the
            source operand are forced to zero. The CRU hardware base address (the
            contents of R12, bits 3 through 14) selects the first CRU bit address
            and subsequent bits read in come from the subsequent CRU bit addresses.
            The content of R12 is unchanged. The bits are read in sequentially on
            the CRUIN line.

Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |L>|A>|EQ|  |  |OP|  |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

The entire content of the source operand (not just the
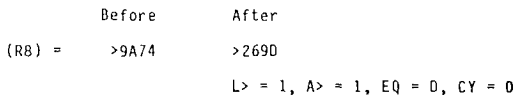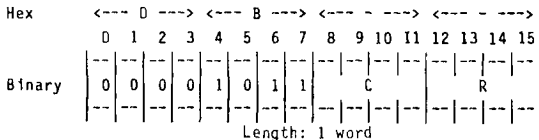transferred bits) is compared to zero and the Logical
Greater Than, Arithmetic Greater Than, and Equal status
bits are affected accordingly. If the source operand is
a byte address, the Odd Parity status bit is affected
based upon the number of one bits in the contents of the
source operand.

Notes: The STCR instruction reads in a multiple number of bits (up to a
        maximum of 16) from a series of CRU bit addresses.

        Example: LI   R12,>400
                     STCR @STATUS,4

Reads in 4 bits from CRU bit addresses hexadecimal 200 through 203 into
the contents of byte address STATUS, bits 7 through 4 (from right to left).

Machine Code:
```
   Hex      <--- 3 --->  <--- - --->  <--- - --->  <--- - --->

            0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  Binary  | 0| 0| 1| 1| 1| 0|   C   |  T  s |   R  s |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
          |            Source Address                     |
          |- - - - - - - - - - - - - - - - - - - - - - - -|

                     Length: 1 or 2 words
```

Store Status   STST

Mnemonic and Addressing Modes: STST R

Result: (ST) --> (R)

Operation: The contents of the register is replaced with a copy of the
    contents of the Status Register.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

    No status bits are affected.

Notes: You can use the STST instruction to save the contents of the Status
    Register. This might be useful for determining the state of the X
    status bit (for which there is no conditional jump instruction) or for
    analyzing the number in the interrupt mask.

    Example: STST R15

    The contents of the Status Register is copied into Register 15.

Machine Code:

```
    Hex       <--- 0 --->  <--- 2 --->  <--- C --->  <--- - --->
              0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary   | 0| 0| 0| 0| 0| 0| 1| 0| 1| 1| 0| 0|     R     |
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
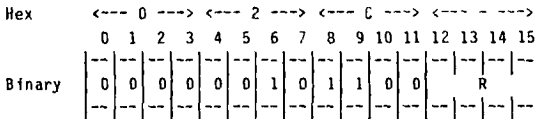
                        Length: 1 word

Store Workspace Pointer   STWP

Mnemonic and Addressing Modes: STWP R

Result: (WP) --> (R)

Operation: The contents of the register is replaced with a copy of the
      contents of the Workspace Pointer.

Status Bits Affected:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
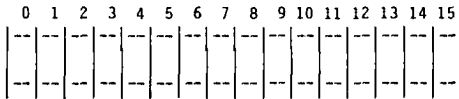
No status are bits affected.

Notes: You can use the STWP instruction to save the address of the current
      workspace. This is sometimes useful in a subroutine so that the
      subroutine can remember the address of its current workspace, use a
      different workspace for a while, and then restore the address of its
      orignal workspace to the Workspace Pointer.

Example: STWP R13

The address value in the Workspace Pointer is copied into Register 13.

Machine Code:

```
Hex       <--- 0 --->  <--- 2 --->  <--- A --->  <--- - --->
          0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
         |--|--|--|--|--|--|--|--|--|--|--|--| --|--|--|--|
Binary   | 0| 0| 0| 0| 0| 0| 1| 0| 1| 0| 1| 0|      R      |
         |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
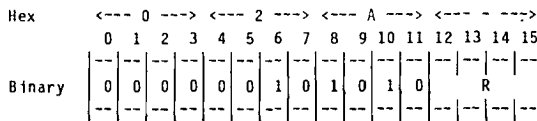
Length: 1 word

Swap Bytes   SWPB

Mnemonic and Addressing Modes: SWPB   S

Result: (S) Bits 0 through 7 --> (S) Bits 8 through 15
          (S) Bits 8 through 15 --> (S) Bits 0 through 7

Operation: Swaps the two bytes in a word.

Status Bits Affected:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
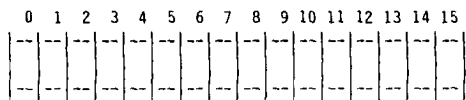
   No status bits are affected.

Notes: Although you can use the SWPB instruction to swap the two bytes in a
         general memory word, it's used more often to exchange the two bytes in a
         register. In order to perform a byte operation with the contents of a
         register, the byte must be in the left half of the register. The SWPB
         instruction can be used to place the right byte of a register in the
         left half.

   Example: SWPB   R4

                Before   After

   (R4)   =   >BC5A   >5ABC

                No status bits affected.

Machine Code:

```
   Hex      <--- 0 ---> <--- 6 ---> <--- - ---> <--- - --->
            0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary  | 0| 0| 0| 0| 0| 1| 1| 0| 1| 1| T s|   R s      |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
           |              Source Address                    |
           |- - - - - - - - - - - - - - - - - - - - - - - - |
```
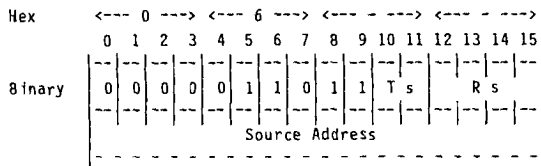
                Length: 1 or 2 words

Set Zeros Corresponding   SZC

Mnemonic and Addressing Modes: SZC S,D

Result: Complement (S) AND (D) --> (D)

Operation: Sets to zero all those bits in the contents of the destination
   operand that correspond to the position of one bits in the source
   operand. Leaves unchanged those bits in the contents of the destination
   operand that correspond to the position of zero bits in the source
   operand. Both operands are word addresses.

Status Bits Affected:

```
      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
    |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    |L>|A>|EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
    |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

The result is compared to zero and the Logical Greater
Than, Arithmetic Greater Than, and Equal status bits are
affected accordingly.

Notes: The SZC instruction can be used to force selected bits in a word to
   zero. The computer ANDs the complement of the contents of the source
   operand with the contents of the destination operand.

Example: SZC @BITMSK,R1

               Before   After

(BITMSK) =   >A60F   >A60F
(R1)   =   >BC5A   >1850

                             L> = 1, A> = 1, EQ = 0

(BITMSK)   = 1010 0110 0000 1111
(R1)   Before   = 1011 1100 0101 1010

(R1)   After   = 0001 1000 0101 0000

Machine Code:
     Hex   <--- 4 ---> <--- - ---> <--- - ---> <--- - --->
              0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
     Binary | 0| 1| 0| 0| T d | R d | T s | R s |--|
            |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                        Source (or Destination) Address
            -
                           Destination Address
            -
                      Length: 1 or 2 or 3 words

Set Zeros Corresponding Byte   SZCB

Mnemonic and Addressing Modes: SZCB   S,D

Result: byte
        Complement (S) AND (D) ---> (D)

Operation: Sets to zero all those bits in the contents of the destination
    operand that correspond to the position of one bits in the source
    operand. Leaves unchanged those bits in the contents of the destination
    operand that correspond to the position of zero bits in the source
    operand. Both operands are byte addresses.

Status Bits Affected:
      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
    |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
L>	A>	EQ			OP										

    The result is compared to zero and the Logical Greater
    Than, Arithmetic Greater Than, and Equal status bits are
    affected accordingly. The Odd Parity status bit is set
    if the byte result contains an odd number of one bits;
    otherwise, it's cleared to zero.

Notes: The SZCB instruction can be used to force selected bits in a byte to
    zero. The computer ANDs the complement of the contents of the source
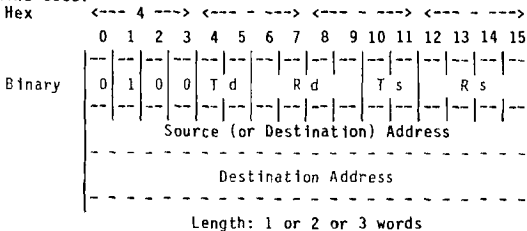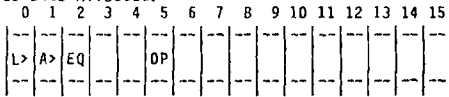   operand with the contents of the destination operand.

    Example: SZCB @BITMSK,R1

                 Before   After
    (BITMSK) =   >A60F   >A60F
    (R1)   =   >BC5A   >185A

                                 L> = 1, A> = 1, EQ = 0, OP = 0

    (BITMSK)   = 1010 0110 0000 1111
    (R1)   Before   = 1011 1100 0101 1010

    (R1)   After   = 0001 1000 0101 1010
                              <- Not ->
                              changed

Machine Code:
    Hex   <--- 5 ---> <--- - ---> <--- - ---> <--- - --->
            0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary| 0| 1| 0| 1| T d| R d | T s| R s|
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
                   Source (or Destination) Address
          |- |
                      Destination Address
          |- |
                  Length: 1 or 2 or 3 words

321
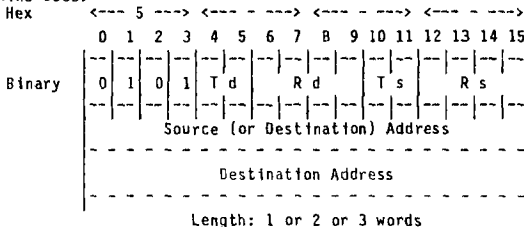
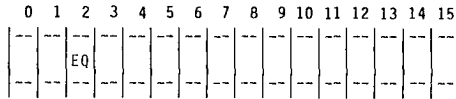Test Bit   TB

Mnemonic and Addressing Modes: TB Displacement

Result: State of bit at selected CRU Bit Address --> EQ
     status bit

Operation: Reads the state of the bit at the selected CRU bit address. The
     state of the bit is recorded in the Equal status bit. The bit address
     is the 12-bit sum of the CRU hardware base address (the contents of R12,
     bits 3 through 14) and the displacement operand (a number in the range
     of -128 through +127). The 12-bit sum appears on the computer's address
     lines A3 through A14. Address lines A0 through A2 are forced to zero.
     The selected bit is read in on the CRUIN line.

Status Bits Affected:

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
  |  |  |EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
  |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
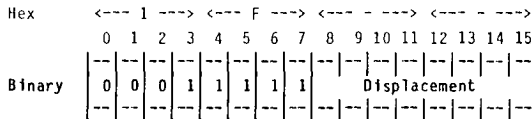
     The Equal status bit is set to the state of the bit read in.

Notes: The TB instruction reads the state of a CRU input bit so that its
     state can be tested.

     Example: LI   R12,>400
                  TB   0
                  JEQ   ON

     The JEQ instruction causes a jump to the instruction labeled ON if the
     CRU bit at address hexadecimal 200 is a one.

Machine Code:

```
    Hex        <--- 1 --->  <--- F --->  <--- - --->  <--- - --->
               0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
              |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
    Binary    | 0| 0| 0| 1| 1| 1| 1| 1|      Displacement     |
              |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

                          Length: 1 word

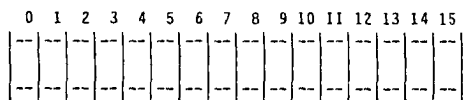Execute   X

Mnemonic and Addressing Modes: X   S

Result: Performs the instruction at the operand address.
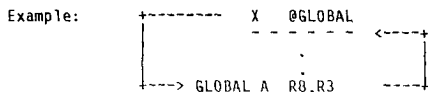

Operation: Performs (executes) the instruction specified by the operand
             address and program control returns immediately to the location
             following the X instruction, unless the executed instruction is one
             which performs a transfer of program control (e.g, a branch or jump
             instruction). If the executed instruction is a jump which causes a
             transfer of control, the jump is made relative to the location of the X
             instruction. If the executed instruction requires more than one word of
             machine code, the word or words following the X instruction are used for
             operand addresses.

Status Bits Affected:

```
 0  1  2  3  4  5  6  7  8  9 10 II 12 13 14 15
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```
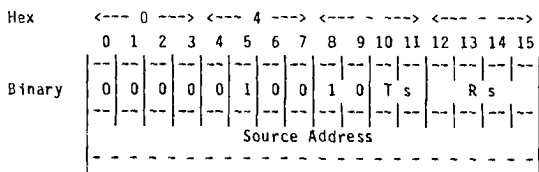
         No status bits are affected by the X instruction itself,
         but the executed instruction affects the status bits.

Notes: The X instruction is sometimes useful in situations where a program
         constructs the machine code for an instruction and then performs it.
         Proceed with caution when using the X instruction.

         Example: +--------   X   @GLOBAL
                         | - - - - - -   <----+
                         | . |
                         | . |
                         +----> GLOBAL A R8,R3   ----+

         Executes the Add Words instruction at GLOBAL and returns to the location
         following the X instruction.

Machine Code:

```
Hex        <--- 0 ---> <--- 4 ---> <--- - ---> <--- - --->
            0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary     | 0| 0| 0| 0| 0| 1| 0| 0| 1| 0| T  s|    R  s  |
           |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
           |                  Source Address              |
           - - - - - - - - - - - - - - - - - - - - - - - -
           |                                              |
                     Length: 1 or 2 words
```

Extended Operation   XOP

Mnemonic and Addressing Modes: XOP S,C

Result: (>40 + 4 x C) --> (WP)
         (>42 + 4 x C) --> (PC)
         (old WP) --> (new R13)
         (old PC) --> (new R14)
         (old SR) --> (new R15)
              S   --> (new R11)

Operation: Performs a context switch using the two-word vector specified by
            the second operand. The second operand is multiplied by four and added
            to hexadecimal 40. The result is the address of the first word of the
            two-word vector and it contains the address of a new workspace. The
            second word contains the address of a program to which a transfer of
            control is made.   When the XOP instruction is perfomed, the current
            contents of the WP, PC, and SR are stored in Registers 13, 14, and 15,
            respectively, of the new workspace. Also, the address of the first
            operand is stored in Register 11 of the new workspace.

Status Bits Affected:

```
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|   |   |   |   |   |   | X |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
```
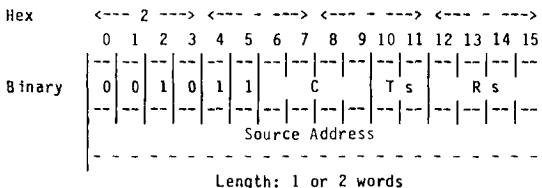
        The Extended Operation status bit is set.

Notes: The XOP instruction performs a context switch using a vector within
        a fixed area of memory.

        Example: XOP @PARAM,1

        A context switch is performed using the contents of memory locations
        hexadecimal 44 and 46 as a two-word vector. The address value of PARAM
        is placed in Register 11 of the new workspace.

Machine Code:

```
Hex        <--- 2 --->  <--- - --->  <--- - --->  <--- - --->
           0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
Binary    | 0| 0| 1| 0| 1| 1|    C    |  T s  |  R s    |
          |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
          |              Source Address                  |
          |- - - - - - - - - - - - - - - - - - - - - - - -|
                     Length: 1 or 2 words
```

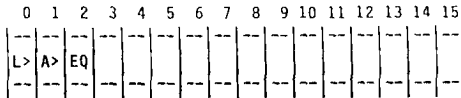Exclusive Or   XOR

Mnemonic and Addressing Modes: XOR S,R

Result: (S) XOR (R) --> (R)

Operation: Performs a logical exclusive OR operation between the bits in
    the source operand and the bits in the destination register. The result
    replaces the contents of the destination register.
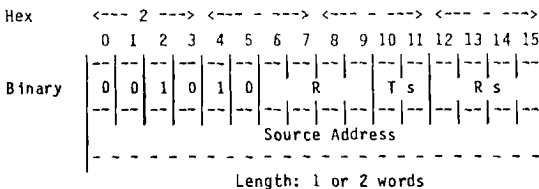
Status Bits Affected:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
 |L>|A>|EQ|  |  |  |  |  |  |  |  |  |  |  |  |  |
 |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
```

    The result is compared to zero and the Logical Greater
    Than, Arithmetic Greater Than, and Equal status bits are
    affected accordingly.

Notes: The XOR instruction is useful for selectively complementing bits in
    a register. Bits in the destination register are complemented for which
    there are one bits in the corresponding bit positions of the source
    operand.

    Example: XOR @BITMSK,R1

                 Before   After

    (BITMSK) =   >A60F   >A60F
    (R1)   =   >BC5A   >1A55

                               L> = 1, A> = 1, EQ = 0

    (BITMSK)   = 1010 0110 0000 1111
    (R1) Before = 1011 1100 0101 1010

    (R1) After   = 0001 1010 0101 0101

Machine Code:

```
   Hex        <--- 2 --->  <--- - --->  <--- - --->  <--- - --->
              0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
   Binary    | 0| 0| 1| 0| 1| 0|    R   | T  s |    R  s |
             |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
             |              Source Address                 |
             |- - - - - - - - - - - - - - - - - - - - - - -|
                     Length: 1 or 2 words
```

Appendix B

Number Tables

Hexadecimal	Binary	Decimal
D	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Even	Byte			Odd	Byte		
Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec
0	0	0	0	0	0	0	0
1	4,096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,766	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,066	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,584	E	224	E	14
F	61,440	F	3,840	F	240	F	15

Appendix C

ASCII Character Table

ASCII CHARACTER SET

Character	Binary	Hex	Decimal
NUL-Null	000 0000	00	0
SOH-Start of Heading	000 0001	01	1
STX-Start of Text	000 0010	02	2
ETX-End of Text	000 0011	03	3
EOT-End of Transmission	000 0100	04	4
ENO-Enquiry	000 0101	05	5
' . Acknowledge	000 0110	06	6
BEL-Bell	000 0111	07	7
BS-Backspace	000 1000	08	8
HT-Horizonatal Tab	000 1001	09	9
LF-Line Feed	000 1010	0A	10
VT-Vertical Tab	000 1011	0B	11
FF-Form Feed	000 1100	0C	12
CR-Carriage Return	000 1101	0D	13
SO-Shift Out	000 1110	0E	14
SI-Shift In	000 1111	0F	15
OLE-Data Link Escape	001 0000	10	16
DC1-Device Control 1	001 0001	11	17
OC2-Device Control 2	001 0010	12	18
DC3-Device Control 3	001 0011	13	19
OC4-Device Control 4	001 0100	14	20
'.'·Negative Acknowledge	001 0101	15	21
·'·Synchronous Idle	001 0110	16	22
ETB-End of Transmission Block	001 0111	17	23
CAN-Cancel	001 1000	18	24
EM-End of Medium	001 1001	19	25
SUB-Substitute	001 1010	1A	26
ESC-Escape	001 1011	1B	27
FS-File Separator	001 1100	1C	28
GS-Group Separator	001 1101	1D	29
RS-Record Separator	001 1110	1E	30
US-Unit Separator	001 1111	1F	31
Space	010 0000	20	32
!	010 0001	21	33
"	010 0010	22	34
#	010 0011	23	35
$	010 0100	24	36
%	010 0101	25	37
&	010 0110	26	38
'	010 0111	27	39
(	010 1000	28	40
)	010 1001	29	41
*	010 1010	2A	42
+	010 1011	2B	43
,	010 1100	2C	44
~	010 1101	2D	45
.	010 1110	2E	46
/	010 1111	2F	47

ASCII CHARACTER SET (Continued)

Character	Binary	Hex	Decimal
a	110 0001	61	97
b	110 0010	62	98
c	110 0011	63	99
d	110 0100	64	100
e	110 0101	65	101
f	110 0110	66	102
g	110 0111	67	103
h	110 1000	68	104
i	110 1001	69	105
j	110 1010	6A	106
k	110 1011	6B	107
l	110 1100	6C	108
m	110 1101	6D	109
n	110 1110	6E	110
o	110 1111	6F	111
p	111 0000	70	112
q	111 0001	71	113
r	111 0010	72	114
s	111 0011	73	115
t	111 0100	74	116
u	111 0101	75	117
v	111 0110	76	118
w	111 0111	77	119
x	111 1000	78	120
y	111 1001	79	121
z	111 1010	7A	122
	111 1011	7B	123
	111 1100	7C	124
	111 1101	7D	125
	111 1110	7E	126
DEL-Delete, Rubout	111 1111	7F	127

Index