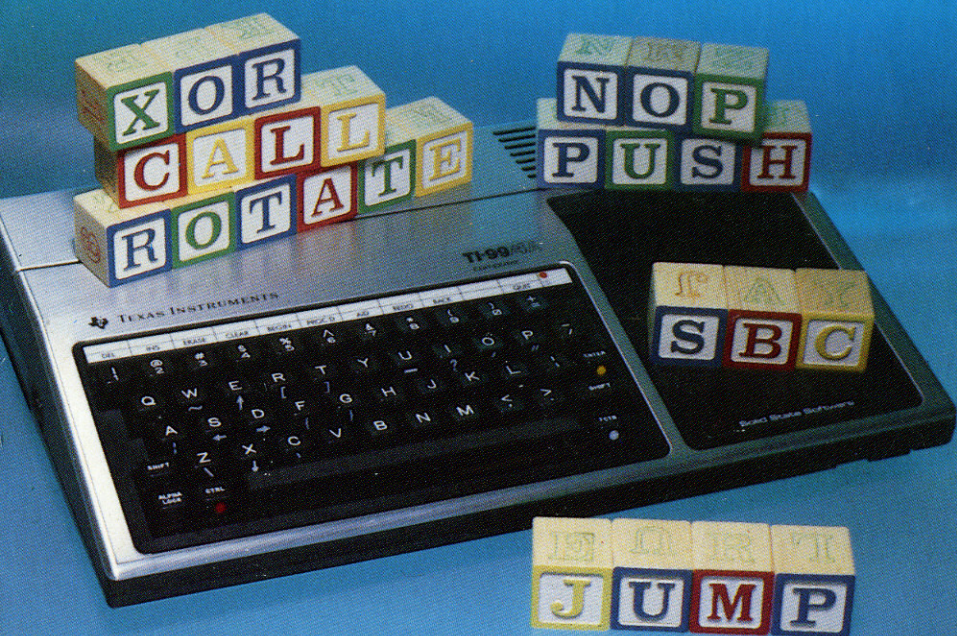


# FUNDAMENTALS OF TI-99/4A ASSEMBLY LANGUAGE

BY M. S. MORLEY



# **FUNDAMENTALS OF TI-99/4A ASSEMBLY LANGUAGE**

**BY M. S. MORLEY**



**TAB BOOKS Inc.**

**BLUE RIDGE SUMMIT, PA. 17214**

**FIRST EDITION**

**FIRST PRINTING**

**Copyright © 1984 by TAB BOOKS Inc.**

**Printed in the United States of America**

**Reproduction or publication of the content in any manner, without express permission of the publisher, is prohibited. No liability is assumed with respect to the use of the information herein.**

**Library of Congress Cataloging in Publication Data**

**Morley, M. S.**

**Fundamentals of TI-99/4A.**

**Includes index.**

**1. TI 99/4A (Computer)—Programming. 2. Assembler language (Computer program language). I. Title.**

**QA76.73.A8M67 1984 001.64'24 83-24189**

**ISBN 0-8306-0722-6**

**ISBN 0-8306-1722-1 (pbk.)**

# Contents

---

<b>List of Programs</b>	<b>v</b>
<b>Preface</b>	<b>vi</b>
<b>Introduction</b>	<b>viii</b>
<b>Part I: Fundamental Concepts</b>	<b>1</b>
<b>1 Programming Languages</b>	<b>3</b>
BASIC—Machine Language—Assembly Language—Summary	
<b>2 Memory</b>	<b>9</b>
Address Bus—Data Bus—Control Bus—ROM and RAM—Summary	
<b>3 Microprocessors</b>	<b>13</b>
Arithmetic Logic Unit—Status Register—General Registers—Program Counter—Control Unit—Summary	
<b>4 Microprocessor Operations</b>	<b>20</b>
Data Transfer—Arithmetic—Logic—Shift and Rotate—Control Transfer—Summary	
<b>5 Addressing Modes</b>	<b>28</b>
Immediate—Direct—Indirect—Indexed—Relative—Summary	
<b>Part II: The 9900 and the TI-99/4A Home Computer</b>	<b>31</b>



<b>6</b>	<b>The TMS9900</b> Architecture—Instruction Set—Addressing Modes—Summary	<b>33</b>
<b>7</b>	<b>The TI-99/4A</b> CPU Memory—Video Data Processor RAM—Graphics Read Only Memory—CRU Bits—Summary	<b>50</b>
<b>8</b>	<b>The Mini Memory Module</b> Loading the Line-by-Line Assembler—Memory Map—Assembler Syntax—Assembler Directives—EASY BUG—Summary	<b>59</b>
	<b>Part III: Programs</b>	<b>75</b>
<b>9</b>	<b>Beginning Programs</b> 16-Bit Data Transfer—64-Bit Data Transfer—16-Bit Addition— 32-Bit Addition—Find the Larger of Two Unsigned Numbers— Sum of Squares—Table of Factorials	<b>77</b>
<b>10</b>	<b>Simple Program Loops</b> 16-Bit Sum of Data—32-Bit Sum of Data—Number of Negative Numbers—Number of Zero, Positive, and Negative Numbers— Find Maximum Value—Find Minimum Byte Value	<b>95</b>
<b>11</b>	<b>Character-Coded Data</b> Length of a String of Characters—Find First Nonblank Character—Find Last Nonblank Character—Replace Leading Zeros with Blanks—Truncate Decimal String to Integer Form— Pattern Match—String Comparison	<b>111</b>
<b>12</b>	<b>Code Conversion</b> Hexadecimal to ASCII—ASCII to Hexadecimal—ASCII to Decimal—Decimal to ASCII—Binary-Coded Decimal to Binary—Binary to BCD—Binary Number to ASCII-Binary String—ASCII-Binary String to Binary Number—Binary Number to ASCII-Hexadecimal String—ASCII-Hexadecimal String to Bi- nary Number—Binary Number to ASCII-Decimal String— ASCII-Decimal String to Binary Number	<b>124</b>
<b>13</b>	<b>Arithmetic Problems</b> 32-Bit by 32-Bit Multiply—64-Bit Division—Square Root— Reciprocal of a Number—Sine of an Angle	<b>154</b>
<b>14</b>	<b>Using the System Utilities</b> Clearing the Screen—Display Text—Generate Cursor— Keyboard Input and Display—Convert String to Number—Raise Number to a Power—Change Screen Color	<b>182</b>
	<b>Appendix : TMS9900 Instruction Set</b>	<b>215</b>
	<b>Index</b>	<b>307</b>

# List of Programs

---

- 9-1. 16-Bit Data Transfer 82
- 9-2. 64-Bit Data Transfer 83
- 9-3. 16-Bit Addition 85
- 9-4. 32-Bit Addition 87
- 9-5. Find the Larger of Two Unsigned Numbers 90
- 9-6. Sum of Squares 91
- 9-7. Table of Factorials 94
- 10-1. 16-Bit Sum of Data 98
- 10-2. 32-Bit Sum of Data 100
- 10-3. Number of Negative Numbers 102
- 10-4. Number of Zero, Positive, and Negative Numbers 105
- 10-5. Find Maximum Value 106
- 10-6. Find Minimum Byte Value 108
- 11-1. Length of a String of Characters 113
- 11-2. Find First Nonblank Character 114
- 11-3. Find Last Nonblank Character 115
- 11-4. Replace Leading Zeros with Blanks 117
- 11-5. Truncate Decimal String to Integer Form 119
- 11-6. Pattern Match 121
- 11-7. String Comparison 123
- 12-1. Hexadecimal to ASCII 127
- 12-2. ASCII to Hexadecimal 129
- 12-3. ASCII to Decimal 130
- 12-4. Decimal to ASCII 131
- 12-5. Binary-Coded Decimal to Binary 135
- 12-6. Binary to BCD 138
- 12-7. Binary Number to ASCII-Binary String 140
- 12-8. ASCII-Binary String to Binary Number 143
- 12-9. Binary Number to ASCII-Hexadecimal String 146
- 12-10. ASCII-Hexadecimal String to Binary Number 148
- 12-11. Binary Number to ASCII-Decimal String 150
- 12-12. ASCII-Decimal String to Binary Number 152
- 13-1. 32-Bit by 32-Bit Multiply 156
- 13-2. 64-Bit Division 160
- 13-3. Square Root 163
- 13-4. Reciprocal of a Number 169
- 13-5. Sine of an Angle 171
- 13-6. Sine of an Angle Using the BL Subroutine Call 177
- 13-7. Sine of an Angle Using the BLWP Subroutine Call 179
- 14-1. Clearing the Screen 184
- 14-2. Display Text 186
- 14-3. Generate Cursor 189
- 14-4. Keyboard Input and Display 192
- 14-5. Convert String to Number 197
- 14-6. Raise Number to a Power 205
- 14-7. Change Screen Color 212

## Preface

---

This book was written to fill a need. I bought my TI-99/4A Home Computer in December, 1982, for two reasons. First, I wanted to get a game machine for my sons, but I also wanted to buy a computer so that they could be exposed to computer programming. I hoped they might even try learning some BASIC. And this they have done, at least to the extent of trying some graphics programs shown in the *TI Beginner's BASIC Teaching Manual* and also typing in programs published in *COMPUTE!* Magazine.

Second, I wanted to learn to write programs in assembly language, having had some experience writing programs in BASIC on a main-frame computer and having also had some experience writing assembly language for a minicomputer. In particular, I was very interested in learning how to program a 16-bit microprocessor, something I had not yet done.

I began pursuing this second purpose within a month or so of purchasing my Home Computer.

At first I looked at the TI Editor/Assembler software package. However, the requirement to buy the Disk Memory System seemed a very high price (\$1000) just to learn assembly language. And then I found out about the Mini Memory module, a software cartridge that also comes with a Line-by-Line Assembler on cassette tape. This was more like it—\$84 and I was in business.

Or so I thought. I thought that the *Mini Memory Owner's Manual* or the *Line-by-Line Assembler Manual* would explain the

9900 instruction set and give me clear program examples illustrating each instruction. I was wrong. Page 9 of the *Mini Memory Owner's Manual* says:

If you intend to use the Mini Memory module for creating your own assembly language programs, it is assumed that you are experienced in TMS9900 assembly language programming and that you are familiar with the internal organization of data and file structures used by the Home Computer. For a complete discussion of these topics, see the *Editor/Assembler* owner's manual.

This certainly wasn't true for me, and I suspected that I was not alone. I purchased the *Editor/Assembler* manual from Texas Instruments for about \$18 plus shipping, and I anxiously looked forward to a good tutorial explanation of the 9900 instruction set. Perhaps it would be similar to what TI had done for BASIC in their *Beginner's BASIC* manual.

But I was wrong. The *Editor/Assembler* manual turned out to be a good reference manual, but it was not, nor ever will be, a self-teaching guide to 9900 assembly language programming.

I decided to write some assembly language programs anyway. What I needed was a good problem set, ranging from very simple assembly language programs to more complex problem solving programs. So I began to look for a book, but no such book existed for the 9900 microprocessor as far as I was able to determine. However, I did find assembly language books about other microprocessors. These assembly language books were tutorial in nature and had similar problem sets. Using these books for program ideas, I wrote many of the programs you will study in this book. Furthermore, after I got going I began to come up with my own program ideas. I also wrote programs that use the internal resources of the TI home computer.

These programs are the basis of this book. I have essentially written the book that I could not find, and I hope it meets the needs of those seeking such a book.

I want to thank Liz Akers and Kim Tabor of TAB BOOKS Inc. for taking a chance on a new author.

I want to thank Texas Instruments for granting permission to reprint the material found in the appendix. This information is not readily available to those who do not work in an electronics engineering environment.

Finally, I want to thank two very good friends, Diane Corbett and Linda Tabor, and my wife, Alice, for typing the manuscript.

# Introduction

---

This book is written for anyone who wants to learn how to program the TI-99/4A Home Computer in assembly language. No prior knowledge of assembly language is assumed. It is assumed, however, that you have at least some experience writing programs in BASIC, TI BASIC, or any other so-called dialect of BASIC and that you are familiar with elementary programming terminology and concepts such as loops and subroutines.

If you are a technician, engineer, or hobbyist who wants to learn 9900 assembly language programming but do not have access to high cost software development systems, then this book is for you.

If you are a TI-99/4A owner who has little or no background in electronics in general and microprocessors in particular and if you want something else to do with your computer, then this book is for you.

If you just want to learn an assembly language, especially a 16-bit microprocessor assembly language, and you are willing to spend less than \$200 (if you haven't yet bought a TI-99/4A), then this book is for you also.

This book will:

- ☐ Teach you the fundamentals of 9900 assembly language.
- ☐ Give you good reference material on the TMS 9900 microprocessor – material that is generally unavailable.



- ☐ Give you a good understanding of how the TI-99/4A works internally.
- ☐ Give you a skill that will help you understand and evaluate the instruction sets of other microprocessors.
- ☐ Give you an appreciation of BASIC and other high-level languages such as FORTRAN and PASCAL, which are simply very complex programs written in assembly language.

This book is divided into three parts: The five chapters in Part I briefly describe the fundamental concepts of programming languages and microprocessor systems. If you are already familiar with these concepts, skip this section. Chapter 1 explains the differences and similarities of BASIC, machine language, and assembly language. Chapter 2 discusses the terminology and operation of the memory in a microcomputer system. Chapter 3 examines the internal organizational features common to most microprocessors. Chapter 4 discusses each of the five basic types of microprocessor operations—data transfer, arithmetic, logic, shift and rotate, and control transfer. Finally, Chapter 5 explains the addressing mode concept.

In Part II the three chapters provide background and procedural information that you will need in order to write TI-99/4A assembly language programs.

Chapter 6 overviews the TMS 9900 microprocessor and discusses the architecture, instruction set, and addressing modes of the 9900. Chapter 7 discusses the internal organization and operation of the TI-99/4A. Chapter 8 explains how to use the Mini Memory module and the Line-by-Line Assembler.

Part III contains six chapters of programs, their listings, and explanations. The programs have been selected and arranged to teach you the 9900 instruction set and how to use other assembly language routines that are stored in the Mini Memory module and the TI-99/4A console.

Chapter 9 contains very simple assembly language programs and explains all the procedural steps required to create and run an assembly language program on the TI-99/4A. Chapter 10 contains programs illustrating the instructions used in program loops. Chapter 11 contains simple programs that process ASCII-encoded strings. Chapter 12 contains 12 useful code-conversion programs. Chapter 13 contains multiprecision arithmetic problem programs. Chapter 14 contains programs that demonstrate how to use the subroutines stored in your computer. In particular, you will learn

how to input data from the keyboard and how to control the screen display.

Finally, the appendix contains detailed instruction set information on opcodes, status register effects, and instruction examples.

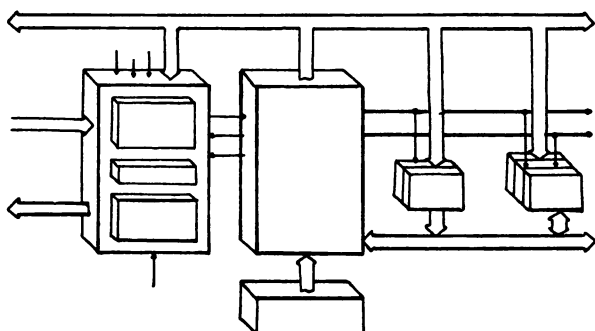
**Part I**

**Fundamental Concepts**

---



# Chapter 1



## Programming Languages

A computer is a data-processing machine designed to solve arithmetic problems and perform other tasks such as accounting, electronic filing, or word processing. A simplified block diagram of a typical home computer is shown in Fig. 1-1.

The brain of this machine is the *microprocessor*, which performs arithmetic and logic functions; receives data from the keyboard; stores data in and retrieves data from its memory; and displays data on the television screen, monitor, or other output device. (A printer will be required if a hard copy of the data is desired or if large quantities of data are to be displayed).

The functions of the computer are controlled by programs stored in its memory. A *program* is a series of instructions written in a language understood by the computer. *Programming* is the art of writing a correct set of instructions. A well-written program uses as few instructions as possible, and works.

Some programs are built in; they have been permanently stored in the computer before it was shipped. These built-in programs are usually designed to enable you to write your own programs and to load and run programs written by the manufacturer or other companies.

### BASIC

All home computers have been programmed by the manufacturer to allow the user to write programs in BASIC, perhaps the



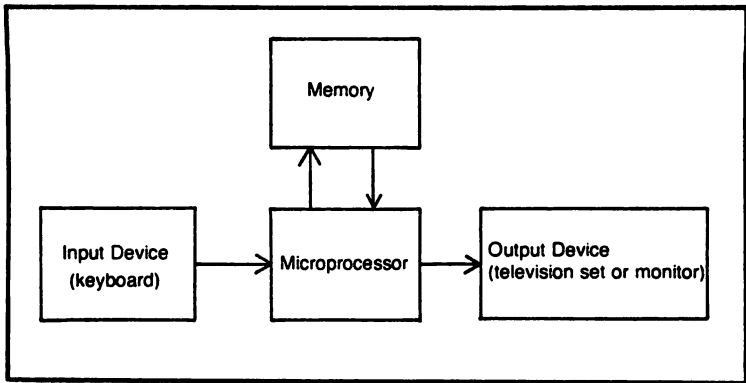


Fig. 1-1. Basic components of a typical home computer.

easiest programming language to learn. BASIC stands for Beginners All-purpose Symbolic Instruction Code. The simplicity of BASIC is demonstrated by the program shown in Fig. 1-2.

Notice the following things about this BASIC program:

☐ It has *line numbers*. Line 10 reads INPUT. Line 40 reads END.

☐ It uses plain English language words: INPUT, PRINT, and END. These words are called *statements*.

☐ The form of the equation on line 20 differs very little from the form used in algebra books or technical literature. Note that an asterisk is used to indicate multiplication in BASIC.

Obviously, few BASIC programs are this simple. However, this program demonstrates the essential features of BASIC. Note that simple programs are easy to write in almost any computer language and that complex programs are difficult to write because they are complex, not necessarily because they are written in BASIC.

The best thing about BASIC is that you can learn this programming language without understanding the internal organization and operation of the computer. You don't even have to know how to type, but it helps.

Fig. 1-2. Sample BASIC program.

```
10 INPUT C
20 F=(9/5)*C+32
30 PRINT F
40 END
```

## MACHINE LANGUAGE

BASIC is often referred to as a *high-level language*. FORTRAN, COBOL, PASCAL, and ADA are also high-level languages. The term high-level language is used to distinguish these languages from *machine language*, which is the most fundamental computer language.

Machine language utilizes only two symbols: 0 (zero) and 1 (one). These symbols correspond to the two possible states of the basic memory unit. The basic memory unit, or cell, is like a switch and is either on or off. Thus, the cell has a *binary* nature, and the number system used by the computer is the *binary number system*.

BASIC is actually a program which resides in the computer memory in the form of 1s and 0s. Furthermore, programs written in BASIC must be translated into machine language before they can be executed, or run. This process of translation is carried out key-stroke by keystroke as the operator types. Thus, when you as the operator type the word INPUT on the keyboard, the computer actually receives a series of 1s and 0s as follows:

0100100101001110010100000101001101010010

Also, the program BASIC consists of many subprograms, called *subroutines*, all (of course) written in machine language. For each BASIC statement, such as INPUT or PRINT, and for each arithmetic operation such as addition or subtraction, there is a corresponding machine-language subroutine having one or more instructions. A machine-language subroutine to perform addition is shown in Fig. 1-3.

This program performs the following steps:

1. Get first number.
2. Add to second number.
3. Store the result.

Note that there are two columns of numbers. The numbers on

0111110100000100	1100000001100000
0111110100000110	0111111000000010
0111110100001000	1010000001100000
0111110100001010	0111111000000100
0111110100001100	1100100000000001
0111110100001110	0111111000000000

Fig. 1-3. Sample binary machine language program.

7D04	C060
7D06	7E02
7D08	A060
7D0A	7E04
7D0C	C801
7D0E	7E00

Fig. 1-4. Sample hexadecimal machine language program.

the left are memory locations, called *addresses* and correspond conceptually to the line number in a BASIC program. The numbers on the right are the data located at the memory addresses. The data may be instructions as well as numbers and characters to be processed. The *instruction* corresponds conceptually to the statement in BASIC.

It should be apparent that writing programs in machine language is tedious and prone to error. Even after careful proofreading, there could easily be one or more errors in the program shown in Fig. 1-3.

Machine-language programming is made less tedious and subject to fewer errors if we use *hexadecimal* notation. Hexadecimal uses the symbols 0 through 9 and A through F. One hexadecimal number takes the place of four binary numbers, as shown in Table 1-1. Using this table you can convert the binary machine-language program, shown in Fig. 1-3, to hexadecimal. The result is shown in Fig. 1-4.

## ASSEMBLY LANGUAGE

*Assembly language* is a symbolic form of machine language.

Table 1-1. Binary to Hexadecimal Conversion.

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

E3	LWPI	WS
	MOV	@M1,R1
	A	@M2,R1
	MOV	R1,@M3
	B	*R11

Fig. 1-5. Sample assembly language program.

Abbreviations, or *mnemonics*, are used in place of 1s and 0s or hexadecimal notation. A sample assembly language program is shown in Fig. 1-5.

Each line of an assembly-language program has three sections, or fields. The first field is the *label* field. The label is similar to the *line number* in BASIC. While each line in BASIC must be numbered, in assembly language the label is optional.

The second field is the *opcode* field. The opcode is the operation to be performed. Moving data from one memory location to another, adding, and subtracting are a few of the operations that the computer may perform. The opcode is similar to the statement in BASIC.

The third field is the *operand* field. The operand may be a number (to be added to another number, for example), or it may be the number of a memory location (a memory address where a number is located). There may be one or two operands in the operand field. If there are two operands, they are separated by a comma.

Before an assembly-language program can be run, it must be translated into machine language. Fortunately this need not be done by hand. There is a program which translates the assembly language mnemonics into machine language. This program is called an *assembler*. Each assembly language instruction (opcode plus operand) is translated into one binary instruction. The assembly-language program is called the *source program*. The binary result (usually represented in hexadecimal) is called the *object code*.

The main advantage of assembly language is that it gives the programmer direct control of the internal computer memory location. The result is that assembly-language programs use fewer memory locations and run faster than BASIC programs.

The disadvantage of assembly language is that it is more difficult to learn than BASIC. Recall that to learn BASIC it is not necessary to understand the internal operation of the computer. This is not true for assembly language. Furthermore, there are as many assembly languages as there are microprocessors. Thus, to learn assembly language you must learn about microprocessors in

general, and in particular, you must learn at least some things about the microprocessor that you want to program.

## **SUMMARY**

In this chapter, I discussed three programming languages: BASIC, machine language, and assembly language.

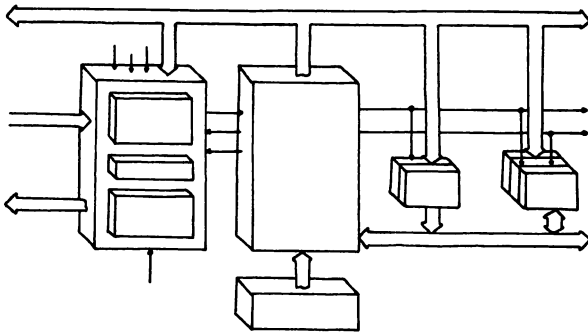
BASIC is an easy programming language to learn. The programmer is not required to understand the internal organization and operation of the computer to use it.

Machine language is the most fundamental programming language, utilizing only two symbols: 0 (zero) and 1 (one). The use of hexadecimal notation makes machine language easier to read and write.

Assembly language is a symbolic form of machine language. Assembly language uses mnemonics in place of the binary code. To learn assembly language, the programmer must have some knowledge of the internal operation of the computer.



## Chapter 2



## Memory

The memory is that part of the computer where programs and data are stored. Recall that the assembly-language programmer has direct control over each memory location. Thus, understanding the terminology and operation of the memory is essential to learning assembly-language programming.

### ADDRESS BUS

The basic interconnection between the microprocessor and the memory is shown in Fig. 2-1. Each *bus* is a group of electrical interconnections. The *width* of a bus is the number of lines in that bus. Each line is called a *bit*.

The microprocessor sends out addresses of memory locations on the address bus. The address is in the form of a binary number. Recall that the binary number system uses only two symbols: 0 (zero) and 1 (one). The width of the address bus determines the maximum number of memory locations that the computer may address according to the following equation:

$$A = 2^n$$

A equals the maximum addressable memory and n equals the width of the address bus. Thus a 16-bit address bus can address  $2^{16}$ , or 65536, memory locations. A 20-bit address bus can address  $2^{20}$ , or 1048576, memory locations.

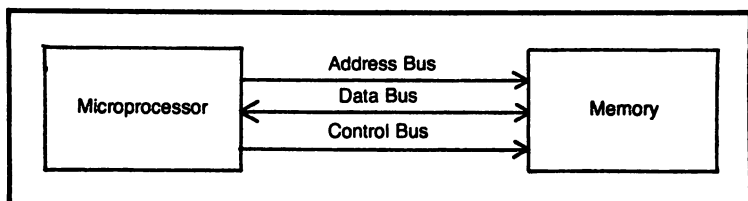


Fig. 2-1. Interconnection between the memory and the microprocessor.

In assembly language, memory locations are usually specified using hexadecimal notation. Thus, the range of addresses for a computer having a 16-bit address bus would be 0000 to FFFF, or from 0 to 65535 in decimal notation. Note that 0, not 1, is the first location. Also, don't worry about not being able to quickly switch back and forth between hexadecimal and decimal. Even experienced programmers must either perform the conversion by hand, or use a special-purpose calculator. Later on, I'll discuss number systems and how to convert from one system to another.

## DATA BUS

The data bus is *bi-directional*. That is, data can be transferred from the microprocessor to the memory or from the memory to the microprocessor. However, data cannot be transferred in both directions at the same time.

Like the address bus, the information on the data bus is in the form of a binary number. Although the width of the data bus may be any length, it is usually one of the following widths: 4, 8, 16, or 32 bits. These widths usually correspond to the data width of the microprocessor. Thus, a 4-bit microprocessor has a 4-bit data bus, an 8-bit microprocessor has an 8-bit data bus, and so forth. This means that an 8-bit microprocessor, for example, has been designed to process, or perform operations on, 8-bit binary numbers.

Also like the address bus, the width of the data bus determines the largest number that the computer can process according to the following equation:

$$D = 2^n - 1$$

D equals the largest number and n equals the width of the data bus. Thus a computer with an 8-bit data bus can process any number up to 255. A computer with a 16-bit data bus can process a number up to 65535. By *process* I mean that the computer can perform a

single operation on a number no larger than that determined by its bus width. Larger numbers, however, may be processed by multiple operations.

Generally, there is one memory address per the number of bits in the data bus. This is true for four and eight bit computers. However, for 16-bit computers there is one address per eight bits, called *bytes*. This gives the 16-bit computer additional flexibility. It may transfer either one byte of data or two bytes of data at the same time. Without this flexibility, memory space would be wasted in those cases where the data was only eight bits wide.

## CONTROL BUS

The control bus consists of two lines: the *read* control line and the *write* control line. These lines are used in conjunction with the address and data buses in two basic modes.

In the read mode, the microprocessor sends out an address to the memory. The read line momentarily goes from the 1 state to the 0 state and back to the 1 state. While the read line is in the 0 state, the memory sends the microprocessor the data contained in the corresponding memory location. The write line is in the 1 state during the read operation.

In the write mode, the microprocessor sends out both address and data information to the memory. The write line momentarily goes from the 1 state to the 0 state and back to the 1 state. While the write line is in the 0 state, the binary information on the data bus is written into the designated memory location. Previous data in this location is overwritten. The read line is in the 1 state during the write operation.

Both of these operations take less than one microsecond for most types of microprocessors. One microsecond is one one-millionth of a second ( $1/1,000,000$  or  $0.000001$  seconds). This means that a typical microprocessor may perform one million memory operations per second.

## ROM AND RAM

There are basically two types of memory inside a home computer: read only memory (ROM) and read/write memory (RWM).

Read only memory is exactly what its name implies: you can only read the contents of this memory. The data in ROM cannot be changed. The ROM in the computer contains prewritten programs and associated data. These prewritten programs (which allow you

to write and run other programs) cannot be accidentally erased. Even if you inadvertently try to write over a ROM location, nothing will happen.

On the other hand, the read/write memory can be read or overwritten. This memory is *volatile*. That means that when you turn off the power the contents of the RWM is lost. Also, when you turn the power on, the state of the RWM is indeterminate. The read/write memory is that area where programs are temporarily stored. To save new programs they must be stored on either cassette tape or floppy disk. Old programs can be loaded into this memory from cassette or disk.

Read/write memory is also called random access memory, or RAM for short. *Random access* means that any memory location may be accessed in any order. This term distinguishes this type of memory from other types of so-called serial memories, such as first-in-first-out (FIFO) or last-in-first-out (LIFO), which may not be randomly accessed. Note, however, that read only memory is also a random access memory although it is never referred to as RAM. I'm sure there's a story behind this inconsistency, but I don't know what it is.

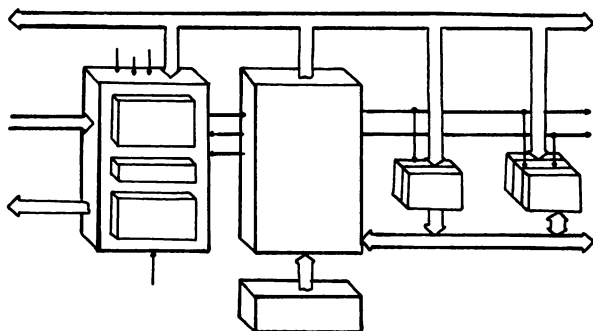
## SUMMARY

In this chapter, I discussed the basic terminology and operation of the memory.

The microprocessor and the memory are connected by three buses: the address bus, the data bus, and the control bus.

The two basic types of memory inside the home computer are the read only memory and the read/write memory - the ROM and RAM. The data in ROM is essentially permanent and can only be read. The data in RAM may be changed as often as the programmer desires. The data in RAM is lost when power is removed.

## Chapter 3



### Microprocessors

In the last chapter, I examined the interconnection, or interface, between the memory and the microprocessor and discussed the terminology and operation of the memory. In this chapter, I will go inside the microprocessor, the so-called brain of the home computer.

Recall that there are as many assembly languages as there are microprocessors. There are two reasons for this: first assembly-language instructions are directly related to the internal design, or *architecture*, of the microprocessor; and second the architecture of each microprocessor is different.

In this book, you are going to learn how to write assembly language programs on the Texas Instruments TI-99/4A home computer. The microprocessor inside the TI-99/4A is called the TMS9900, also made by Texas Instruments. The 6502 microprocessor, made by several manufacturers, is inside the ATARI, VIC-20, and Apple II computers. The Intel 8088 microprocessor is inside the IBM Personal Computer. The internal designs of these microprocessors are all different, and the assembly language-instructions for each microprocessor (and home or personal computer) is consequently different.

To write assembly language programs for the TI-99/4A, you will have to learn at least some things about the internal design of the TMS9900. Before I do that, however, I am going to discuss the terminology and internal design of microprocessors in general. This



will make it much easier to understand the TMS9900 later on.

Figure 3-1 shows a block diagram of a generalized microprocessor. This drawing does not represent any commercial microprocessor and has been simplified in order to clearly explain those features that are common to many different types of microprocessors. Note the following primary functions within the microprocessor:

- ☐ The arithmetic logic unit (ALU).
- ☐ The status register.
- ☐ The general registers.
- ☐ The program counter.
- ☐ The control unit.

The multiplexers and demultiplexers are secondary functions

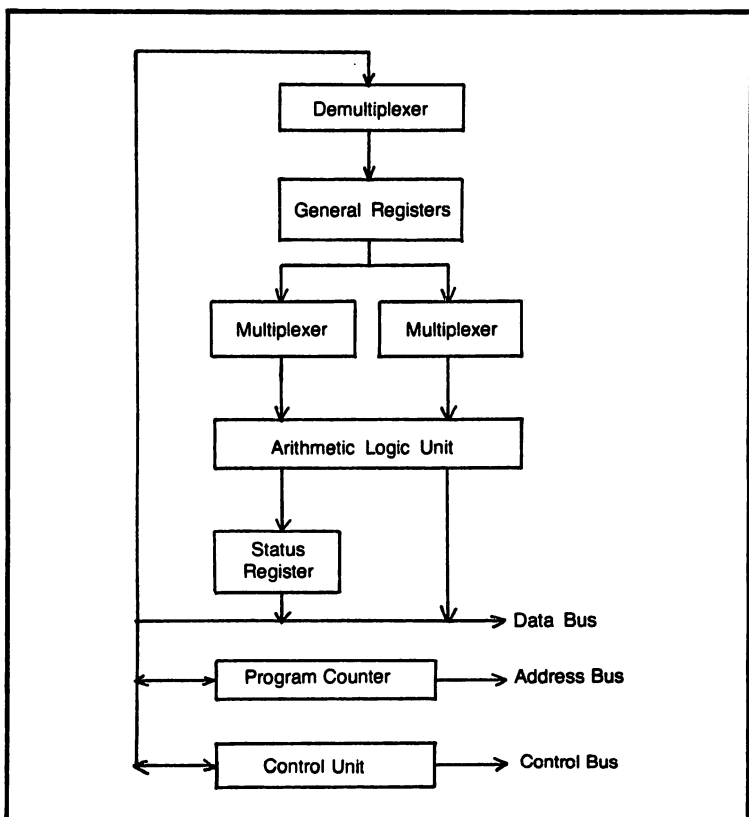


Fig. 3-1. Block diagram of a generalized microprocessor.

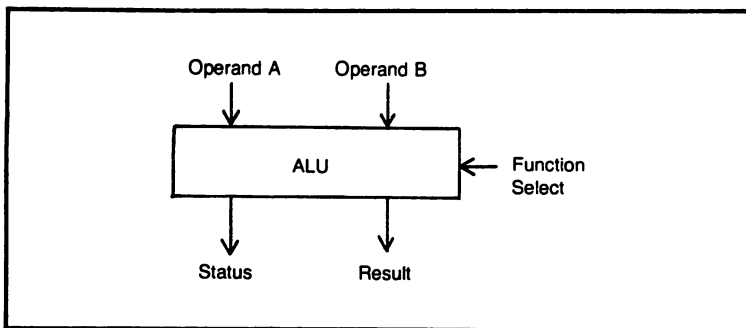


Fig. 3-2. Inputs and outputs of the ALU.

which facilitate the movement of data between the main blocks and also to the so-called *outside world*—the memory, the keyboard, and the display unit.

## ARITHMETIC LOGIC UNIT

Let's begin with the arithmetic logic unit, or ALU, which is considered the heart of the microprocessor. As its name implies, this unit performs arithmetic and logic operations on either one or two *operands*. In Fig. 3-2, these operands are called operand A and operand B. Note that two outputs are generated each time an operation is performed: one is the *result* of the operation and the other is the *status*, or *condition code*.

Operand A, operand B, and the result are binary numbers. The number of bits in both of these numbers corresponds to the width of the data bus which I discussed in the last chapter. The TMS9900, for example, is a 16-bit microprocessor. The data bus is 16 bits wide and the ALU operands and result are also 16 bits wide. Thus, a 16-bit ALU performs arithmetic and logic operations on 16 bits of data at one time. Operating on 32 or 64 bits would require multiple operations.

The *function select* shown in Fig. 3-2 represents one or more control lines which determine which of the following typical operations the ALU is to perform:

- ☐ A plus B.
- ☐ A minus B.
- ☐ B minus A.
- ☐ A OR B.
- ☐ A AND B.
- ☐ A exclusive-OR B.

- ☐ Set result to all zeros.
- ☐ Set result to all ones.

Plus and minus are arithmetic operations (addition and subtraction). OR, AND, and exclusive-OR are logic operations. Set result to all zeros and set result to all ones are neither arithmetic nor logic operations (they do not involve input operands at all), but are used when it is desirable to write all zeros or all ones to a particular memory location. Each of these and other microprocessor operations will be explained in the next chapter.

## STATUS REGISTER

As mentioned before, the ALU has two outputs: the result and the status. The result, obviously, is the answer. If the operation was addition, then the result would be a sum. The status, on the other hand, gives us some additional information about the result. The status word is a group of bits. The state (0 or 1) of each bit corresponds to the occurrence or nonoccurrence of a particular condition. The state of each condition is stored in the status register. (*Register* is another name for a storage location.) One or more status bits is updated at the end of each operation.

The four most common conditions that are monitored are called carry, overflow, zero, and negative. When two numbers are added, a carry may occur. If a carry does occur, then the *carry* bit is set to 1, otherwise it is reset to 0. The *overflow* bit is set when the result of the operation is too large or too small to be correctly represented in 2s complement form (to be explained). The *zero* bit will be set if the result is zero, and the *negative* bit will be set if the result is less than zero.

The status of the bits is used as a basis for making decisions during program operation. The programmer may want to branch, or jump, to another part of the program depending on the state of one of the status bits. He may want to perform one task if the result is zero and a different task if the result is negative. Thus, just as the BASIC programmer uses the IF-THEN-ELSE combination statement to jump to another location in his program, the assembly language programmer uses a *jump on zero* or other conditional branching instruction which causes the computer to go to another memory location (other than the next one in sequence) based on the condition code contained in the status register.

## GENERAL REGISTERS

A set of general purpose registers is often included in the

microprocessor to improve processor speed. It takes the processor less time to read or write to an internal register than it does to read or write to an external memory location.

Programming is also simplified. Earlier microcomputers had only one register for ALU operations. It was called the *accumulator*. To perform an ALU operation, the programmer had to first copy data from an external memory location into the accumulator. Next he added, for example, the contents of the accumulator to the contents of a memory location (the address of which was previously stored in a specific internal register) and then stored the result back into the accumulator. Before the programmer could perform another ALU operation he had to store the contents of the accumulator in an external memory location. Thus, it took at least four assembly-language instructions to perform a simple addition. If, however, the microprocessor has a number of general purpose internal registers, then many ALU operations may be performed

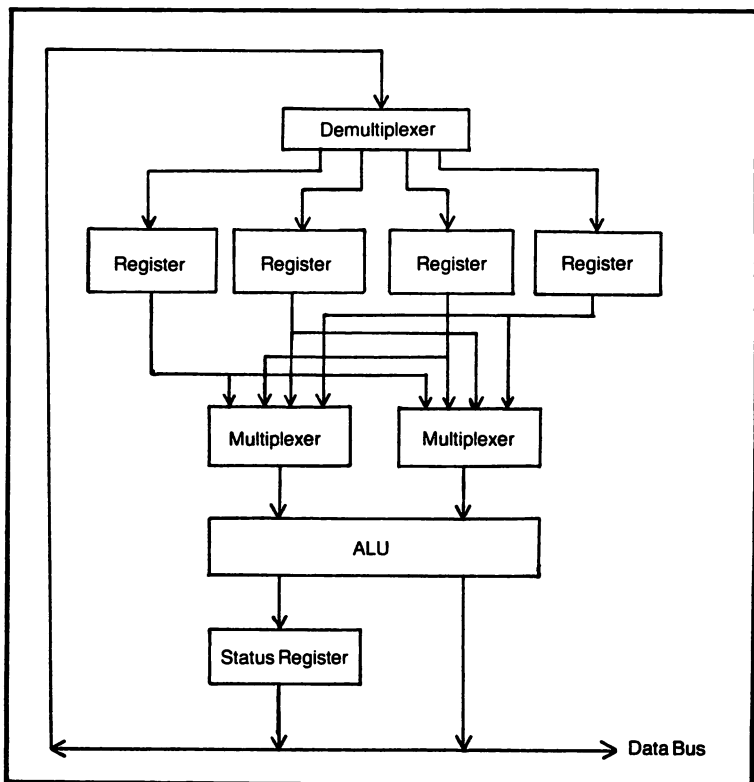


Fig. 3-3. ALU and register set interconnection.

and it may not be necessary to ever store the results in external memory.

Figure 3-3 shows the interconnection between the ALU and the general register set. This figure shows only four registers, but some commercial microprocessors have 16 or more. Note that the purpose of the multiplexers is to select which register's data will be the *source* for operand A and which will be the source for operand B. A multiplexer is conceptually identical to the channel switch on your television set.

The demultiplexer selects which registers will be the *destination* for an ALU operation or a memory operation. For an ALU operation the demultiplexer selects which register will be the storage location for the result. For a memory operation, the demultiplexer selects which register will contain a copy of the data in the memory.

## PROGRAM COUNTER

The program counter contains the memory address of the next instruction to be executed. Normally all instructions are executed sequentially. After an instruction has been read into the microprocessor from the memory, the program counter is automatically incremented by one. Most programs, however, involve jumping, or branching, to some other location in memory. In these cases, the program counter is first loaded with the new location and then incremented thereafter or until another jump instruction is encountered.

When the microprocessor is instructed to begin a subroutine, the contents (plus one) of the program counter is saved either in another internal register or an external memory location. The address of the first instruction in the subroutine is then loaded into the program counter and incremented until a *return* instruction is encountered. Then the old program counter value is loaded into the program counter register and the program continues, executing each instruction sequentially until encountering either a jump instruction or a subroutine call.

## CONTROL UNIT

The control unit is the most complex of all the microprocessor functions. It is shown as a simple block in Fig. 3-1. Not shown is the fact that the control unit is connected to every other block.

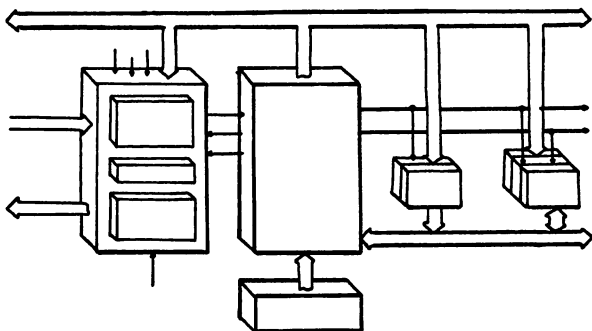
The control unit causes the addresses to be sent out on the address bus to the memory. The control unit sends out a read

command to the memory and reads the instruction (or data) that comes back on the data bus. The control unit then decodes the instruction and sends out appropriate signals to the ALU, the multiplexers, demultiplexers, registers, and the program counter. If the instruction was to store data in the memory, then the control unit sends out the address and the data and a write command to the memory.

## **SUMMARY**

A typical microprocessor consists of five major functional blocks. The ALU performs arithmetic and logic operations. The status register records the effects of the ALU operations. The general register set provides convenient additional storage locations for the programmer and, additionally, improves the execution time for a program. The program counter contains the address of the next instruction to be executed. The control unit decodes instructions, sets up all internal conditions necessary to execute the instructions, and sends out read and write commands to the memory.

## Chapter 4



### Microprocessor Operations

Microprocessors perform five basic types of operations: data transfer operations, arithmetic operations, logic operations, shift and rotate operations, control transfer operations.

#### DATA TRANSFER

The *move* instruction is probably the most used instruction in the entire instruction set of any microprocessor. This instruction is used to transfer data from one location (the source) to another location (the destination). These locations may be almost anywhere in the microcomputer system. Thus, I may move data from the memory to a register, from a register to the memory, from one register to another register, or from one memory location to another memory location.

By the word *move* I really mean that the data is copied. After I move data from a memory location to a register, for example, the original data is still in the memory—it has not been lost, nor is the memory location empty. However, any data in the register (to which I moved data) has been overwritten and is lost (unless previously moved elsewhere, of course).

*Load* and *store* instructions are variations on the basic move instruction. Generally, I *load* a register with data, where the data to be loaded is located in the memory location immediately following the memory location that contains the load instruction. Thus, in a load instruction, only the destination needs to be specified because

the source is implied. Similarly, I store a result in a register or memory location.

**ARITHMETIC**

Microprocessors perform two basic arithmetic operations: addition and subtraction. Recall that microprocessors perform operations on binary numbers and that the largest possible number is set by the width of the data bus or, more particularly, the number of bits that the ALU has been designed to operate upon. Thus, a 16-bit microprocessor may perform addition or subtraction on any two 16-bit binary numbers. Let me illustrate how the computer performs both of these operations. For simplicity I will use only 4-bit numbers.

**Binary Addition**

Suppose I wanted to add the number 9 to the number 3. The operation is shown in both decimal and binary. The binary equivalents came from Table 4-1.

Decimal

9  
+3  

---

12

Binary

1001  
+0011  

---

1100

Binary addition is performed in the same manner that decimal addition is performed. I start in the right hand column (the *least significant bit*, or LSB) and move left one column at a time until I

**Table 4-1. Decimal-Binary Equivalents.**

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



reach the leftmost column (the *most significant bit*, or MSB). When I add two single-bit binary numbers, there are only four possible combinations:

- ☐ 0 plus 0 equals 0. Carry equals 0.
- ☐ 0 plus 1 equals 1. Carry equals 0.
- ☐ 1 plus 0 equals 1. Carry equals 0.
- ☐ 1 plus 1 equals 0. Carry equals 1.

The solution to the simple addition problem is as follows:

1. Start at the right hand column. 1 plus 1 equals 0 with 1 to carry.

$$\begin{array}{r} 1 \\ 1001 \\ +0011 \\ \hline 0 \end{array}$$

2. Move one column to the left. Since I have a carry, I must add twice. 1 (the carry) plus 0 equals 1 (no carry). 1 (result from last add) plus 1 equals 0 with 1 to carry. The result of steps 1 and 2 are as follows:

$$\begin{array}{r} 11 \\ 1001 \\ +0011 \\ \hline 00 \end{array}$$

3. Move one column to the left. 1 (the carry) plus 0 is 1 (no carry). 1 (result from last add) plus 0 is 1 (no carry). The result of steps 1, 2 and 3 are:

$$\begin{array}{r} 11 \\ 1001 \\ +0011 \\ \hline 100 \end{array}$$

4. Move one column to the left. 1 plus 0 equals 1 (no carry). Addition is complete:

$$\begin{array}{r} 11 \\ 1001 \\ +0011 \\ \hline 1100 \end{array}$$

## Binary Subtraction

Binary subtraction, unfortunately, is not as straightforward as binary addition. Most computers perform binary subtraction by what is called *the addition of the 2s complement*. In simple terms, this means that instead of subtracting one number from another the computer adds two *signed* numbers. The form for signed numbers is called 2s complement.

Table 4-2 gives the 2s complement of all integers between +7 and -8. Note that this is the maximum number range for a 4-bit microprocessor. The maximum number range for an 8-bit microprocessor is +127 to -128. The maximum number range for a 16-bit microprocessor is +32,767 to -32,768.

Also note that the 2s complement of a positive number is the same as its binary representation. The 2s complement of a negative number is formed by first changing all 1s to 0s and 0s to 1s and then adding 1. For example, let's convert the number -3 to its 2s complement form. First, I change 0011 to 1100. Then I add 0001 to 1100 and get 1101. The leftmost bit is called the sign bit: 0 indicates a positive number, 1 indicates a negative number.

As an example of binary subtraction, let's subtract 4 from 7. This is the same as adding +7 and -4. First, I convert each number to its 2s complement form as outlined above or by looking up the numbers in Table 4-2. The 2s complement for +7 is 0111 and the 2s complement for -4 is 1100. Next, I add the 2s complement forms of the numbers:

**Table 4-2. 2s Complement of Signed Numbers.**

Signed Decimal	Binary	2s Complement
7	0111	0111
6	0110	0110
5	0101	0101
4	0100	0100
3	0011	0011
2	0010	0010
1	0001	0001
0	0000	0000
-1	-0001	1111
-2	-0010	1110
-3	-0011	1101
-4	-0100	1100
-5	-0101	1011
-6	-0110	1010
-7	-0111	1001
-8	-1000	1000

$$\begin{array}{r}
 0111 \\
 +1100 \\
 \hline
 0011
 \end{array}$$

The final carry is ignored. The result is in 2s complement form. Using Table 4-2, notice that the number is +3, which is the correct answer.

Now let's subtract 7 from 4:

Decimal	2s Complement
+4	0100
-7	+1001
<u>-3</u>	<u>1101</u>

When you look up 1101 in Table 4-2 you will see that it is equivalent to -3. Since the result is negative, the status register records this condition by setting the negative bit to 1.

What happens if I subtract 7 from -4?

Decimal	2s Complement
-4	1100
-7	+1001
<u>-11</u>	<u>0101</u>

When you look up 0101 in Table 4-2 you will see that it is equivalent to +5, which is not the correct answer. This is because the correct answer is outside the maximum allowable range for a 4-bit microprocessor. Consequently, the overflow bit in the status register is set to 1. Note that it is the programmer's responsibility to make sure he is subtracting numbers that will give him legal results.

## LOGIC

Microprocessors perform three basic logic operations: AND, OR and exclusive-OR.

The logic operation results on two single-bit operands are shown in Table 4-3. This table shows the corresponding logic operation results for the four possible combinations of 0 and 1 of the input operands. Logic operations on 4, 8, or 16-bit operands are performed on a single-bit basis.

As an example, let operand A equal 0101 and operand B equal 1100. Starting at either the left or right, look up the logic results in

**Table 4-3. Logic Operation Results on Two Single-Bit Operands.**

Operand A	Operand B	Logic Operation Result		
		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 4-3 for each pair of bits. Starting at the leftmost bit, operand A equals 0 and operand B equals 1. Therefore A AND B equals 0; A OR B equals 1; and A XOR B equals 1. In the same manner, you can look up the logic operation results for the next three pairs of bits. The results are summarized as follows:

0101 AND 1100 equals 0100.

0101 OR 1100 equals 1101.

0101 XOR 1100 equals 1001.

## SHIFT AND ROTATE

Microprocessors perform two basic shift operations: shift left and shift right. Rotate operations are specialized shift operations.

### Shift

In a shift left operation, all bits in a register or memory location are shifted one position to the left. The leftmost bit is transferred to the carry in the status register. The rightmost bit, after the shift, is set to 0. This operation is typically called *arithmetic shift left*.

Let's take a 4-bit number and perform an arithmetic shift left. Let's also assume the carry equals 0. Before the shift I have the following:

Carry	Register Contents
0	1011

After a left shift the carry and register contents look like this:

Carry	Register Contents
1	0110

In a shift right operation, all bits in a register or memory location are shifted right one position. The rightmost bit is transferred to the carry, and the leftmost bit, after the shift, is set to 0. This operation is typically called *logic shift right*. Let's perform a logic shift right.

Before		After	
Register	Carry	Contents	Carry
1011	0	0101	1

Another shift right operation is called *arithmetic shift right*. The difference between this operation and the logic shift right is that the leftmost bit remains unchanged after an arithmetic shift right:

Before		After	
Register	Carry	Contents	Carry
1011	0	1101	1

## Rotate

In rotate operations, the bits are circulated back into the register. The carry may or may not be included in the loop. For example, in a rotate left operation including carry, all bits would be shifted to the left one position (same as the shift operation described above). The leftmost bit would be transferred to the carry (same as before), while the carry would be transferred to the rightmost bit (different than before).

## CONTROL TRANSFER

Microprocessors perform three basic control transfer operations. They are unconditional jump, conditional jump, and jump to subroutine.

In an unconditional jump the computer is instructed to load a new address into the program counter and then begin executing instructions starting at that address. An unconditional jump instruction in assembly language is the same as the GOTO statement in BASIC.

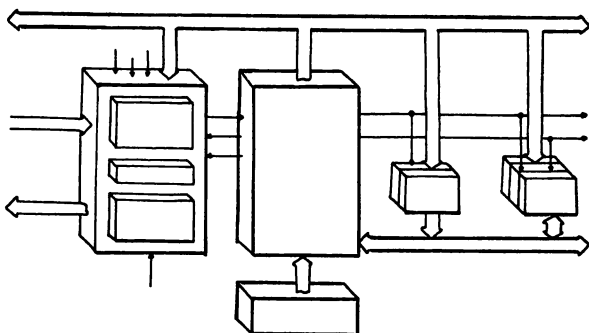
In a conditional jump, the program counter is loaded with a new instruction address only if and when certain conditions have occurred. The microprocessor reads the appropriate status register bits to see if the condition has occurred. If so, then a jump occurs. Otherwise the next consecutive instruction is executed. This is the same as the IF-THEN statement in BASIC.

Subroutines are subprograms with one or more instructions in them. Subroutines begin with a label that is used to call it up. Subroutines end with a return instruction. When the computer encounters a subroutine call, it loads the address of the first instruction in the subroutine and then begins executing the subroutine. Subroutines may contain conditional or unconditional jumps as well as subroutines. When a subroutine contains a subroutine, then the two subroutines are said to be *nested*. This instruction is like the GOSUB statement in BASIC.

## **SUMMARY**

Microprocessors perform five basic types of operation: data transfer, arithmetic, logic, shift and rotate, and control transfer. Move, load, and store are data transfer operations. Add and subtract are arithmetic operations. AND, OR, and XOR are logic operations. Shift and rotate operations involve the transfer of bits in a register or memory location to the left or right. Conditional jumps, unconditional jumps, and subroutine calls are control transfer operations.

## Chapter 5



### Addressing Modes

One final concept remains to be explained before we discuss the TMS9900 microprocessor and the TI-99/4A home computer and begin writing assembly language programs. That concept is called the *addressing mode*.

Microprocessors perform operations on data and place the results in registers or memory locations. The data to be processed and the address of the result are called *operands*.

The operand itself or the location of the operand within the microcomputer system is determined by what is called the addressing mode, which must be specified for each operand in the assembly language instruction.

If the operand is data, then the programmer must specify the data itself or where the data can be found. If the operand is an address, then the programmer must specify either the address directly, or indirectly by specifying the register or memory location where that address is stored.

There are five basic addressing modes that are common to most microprocessors:

- ☐ Immediate
- ☐ Direct
- ☐ Indirect
- ☐ Indexed
- ☐ Relative

A single instruction with two operands may specify one addressing mode for the first operand and a different addressing mode for the second operand. Let me illustrate each of these addressing modes.

## IMMEDIATE

Immediate addressing applies to data operands only. In the instruction, load into register 1 the number 23, the number 23 is the data. The addressing mode is called *immediate* because this instruction takes two memory locations; the first memory location contains the machine code (binary) corresponding to "load into register 1"; the memory location *immediately* following contains the binary equivalent of 23.

## DIRECT

Direct addressing applies only to address operands. In the previous instruction, load into register 1 the number 23, register 1 is an address operand. It is specified *directly*, in contrast to indirectly (which we will get to in a moment).

This mode is also called *register direct*, contrasting this mode with memory direct addressing. The instruction, move the contents of memory location 7E00 to register 1, is an example where both operands are addresses, and both are specified directly. Nevertheless, the TMS9900 differentiates between register direct and memory direct addressing and uses different *mnemonics* (symbolic abbreviations) for each. Memory direct addressing is called *symbolic* addressing in TMS9900 assembly language.

Note in the last example that a data operand is concealed in the words "the contents of." Thus, it is data that is moved, even though the instruction contains an address operand. In fact, data operands are specified directly only in the immediate addressing mode. Confusing, isn't it?

## INDIRECT

When the indirect addressing mode is specified, then the address of either the data or the address must be looked up. In the instruction, load the number 23 into the memory location the address of which is stored in register 1, the microprocessor must read the contents of register 1 to find out the destination address for the number 23.

As another example, move the contents of register 1 to the



memory location the address of which is stored in register 2, the first operand (the contents of register 1) is specified using the direct addressing mode (indirectly specifying a data operand) and the second operand (register 2) is specified using the indirect addressing mode. The words “the memory location the contents of which is stored in” specify the addressing mode. These words are reduced to a single symbol in TMS9900 assembly language—the asterisk (\*).

## **INDEXED**

When the indexed addressing mode is specified, the address is formed by adding the contents of a register with a constant. In the example, move the number 23 to the memory location the address of which is the sum of the contents of register 2 and the number 5, if register 2 contained the address 7D00, then the number 23 would be stored in memory location 7D05.

## **RELATIVE**

Relative addressing is specified for jump, or branch, instructions only. Jump forward 16 memory locations is an example of relative addressing.

In this example, the number 16 is added to the address in the program counter. This new address (the program counter plus 16) is loaded into the program counter and is the address of the next instruction to be executed. Note that the new program counter address is relative to the old program counter address. With respect to the old address, the new address is 16 memory locations forward.

Relative addressing is not the only way to specify a jump or branch address. I could instruct the computer to jump to the address stored in register 1 (indirect) or jump to address 7E0A (direct), for example.

## **SUMMARY**

Microprocessors perform operations on data and place results in registers or memory locations. The data to be processed and the address of the result are called operands.

The operand itself or the location of the operand within the microcomputer system is determined by what is called the addressing mode, which must be specified for each operand in the assembly language instruction.

There are five basic addressing modes common to most microprocessors: immediate, direct, indirect, indexed, and relative.

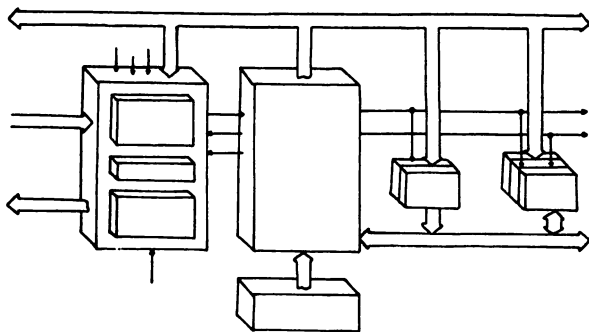
## **Part II**

### **The 9900 and the TI-99/4A Home Computer**

---



# Chapter 6



## The TMS9900

The TMS9900 is a 16-bit microprocessor. It is capable of addressing up to 65,536 bytes of memory; it performs both word and byte operations, uses memory-to-memory architecture, and has 69 instructions and 7 addressing modes. In this chapter, I will examine the primary architectural features, the instruction set, and the addressing modes of the TMS9900, hereafter simply referred to as the 9900.

### ARCHITECTURE

The following section briefly describes the primary architectural features of the 9900. More detailed information can be obtained by writing to Texas Instruments and asking for the TMS9900 data sheet.

#### Memory Addressing

The 9900 has a 15-bit address bus and a 16-bit data bus. This means that the 9900 can address up to 32,768 words of 16-bit data.

Internally, the 15-bit address bus is left-justified one bit to form a 16-bit address. This means that all 16-bit data is essentially stored at even addresses. However, because of the internal 16-bit address bus, 8-bit byte data may also be addressed, and both byte and word operations may be performed.

For example, the word at address 7D00 (even address) is composed of two bytes. The address of the most significant byte is

7D00, and the address of the least significant byte is 7D01. The 9900 can perform a word operation on the word at 7D00 or it can perform a byte operation on the byte at 7D00 or on the byte at 7D01. However, when the 9900 performs a byte operation on either the byte at 7D00 or the byte at 7D01, it reads both bytes, temporarily stores them internally, and then performs the operation on the byte specified in the instruction.

## Program Counter

The program counter is a 16-bit register and contains the address of the next instruction to be executed. The program counter address is always an even address. After an instruction is fetched from memory (but before the instruction is executed), the program counter is incremented by two.

## Workspace

A unique feature of the 9900 is its lack of an internal general purpose register set. Instead, a set of sixteen 16-bit registers, called the *workspace*, resides in external memory. The address of the first register in the workspace is contained in an internal register called the *workspace pointer* (WP).

Any area in the unused external read/write memory (the RAM) may be designated as the workspace by the programmer. Let's suppose, for example, that you decide to place the first register of the workspace at memory address 70B8. This is accomplished by

**Table 6-1. Example  
Workspace for WP-70B8.**

Memory Address	Register
70B8	0
70BA	1
70BC	2
70BE	3
70C0	4
70C2	5
70C4	6
70C6	7
70C8	8
70CA	9
70CC	10
70CE	11
70D0	12
70D2	13
70D4	14
70D6	15

loading the number 70B8 into the WP register. Thereafter, memory locations 70B8 through 70D6 are referred to as registers 0 through 15 of the workspace (See Table 6-1).

The 9900 does not have an internal accumulator. Instead, all 16 external workspace registers may be used as accumulators. This is called *memory-to-memory architecture*. In the earlier microprocessors, data had to be first moved to an internal accumulator before an operation could be performed on it. With memory-to-memory architecture, data may be processed and stored back into memory by using a single assembly language instruction.

Note that the programmer may allocate several areas in memory to be workspaces. Changing workspaces is as simple as loading a new workspace pointer. Changing workspaces is called *making a context switch* or *changing the program environment*. Thus, a context switch or a program environment change can be accomplished using a single instruction. This saves both in the number of instructions and in execution time since the programmer does not have to worry about saving the contents of his workspace registers. With other microprocessors, the programmer would have to save each register, using at least one instruction per register to be saved.

### Communications Register Unit

Another unique feature of the 9900 is the communications-register unit (CRU). Twelve lines of the address bus are used in conjunction with the CRUIN (CRU input) line, the CRUOUT (CRU output) line, and the CRUCLK (CRU clock) line to individually address up to 4096 input lines and up to 4096 output lines. The data bus is not used. CRU interface logic is required to decode the individual addresses and store data sent out on the CRUOUT line. A simplified diagram of the CRU interface is shown in Fig. 6-1.

When an input line is selected, the logic state (0 or 1) is read via the CRUIN line and stored in the 9900. When an output line is selected, a 0 or 1 is sent out on the CRUOUT line and stored in an

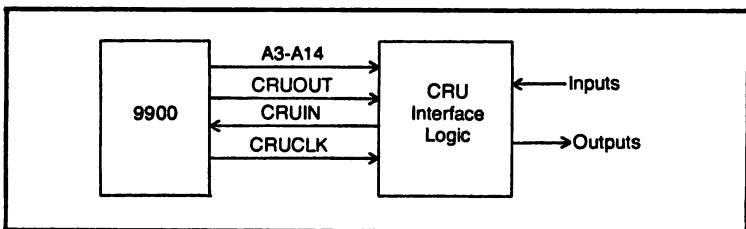


Fig. 6-1. 9900 CRU interface.

external latch (a single-bit storage location) when the CRUCLK line goes from the 0 state to the 1 state and back to the 1 state (a pulse, in other words).

In the TI-99/4A home computer, the keyboard keys, the joystick positions, and the cassette input and output lines are all connected to the 9900 via a CRU interface logic integrated circuit called the TMS9901 Programmable Systems Interface.

## Interrupts

The 9900 has the capability of being interrupted during processing. When an interrupt is requested (depressing the FUNC and = keys on the TI-99/4A, for example), the interrupting device must supply a 4-bit *interrupt priority code*. This code is read by the computer and compared with a number called the *interrupt mask* which is stored internally and may be changed by the programmer.

If the interrupt mask has been set to 5, for example, then external devices with interrupt priority codes 0 through 5 will be allowed to interrupt whatever the computer was doing. Note that 0 is the highest priority and 15 (F in hexadecimal) is the lowest. If in our example an interrupting device supplied a code of 6 or higher, then the computer would ignore the request.

If the code passed the priority test, then the computer would look up new values for the program counter (PC) and workspace pointer (WP). These values must be stored beforehand in reserved

**Table 6-2. Interrupt Vector Locations.**

Priority Code	Memory Address Containing New Workspace Pointer Value	Memory Address Containing New Program Counter Value
0	0000	0002
1	0004	0006
2	0008	000A
3	000C	000E
4	0010	0012
5	0014	0016
6	0018	001A
7	001C	001E
8	0020	0022
9	0024	0026
10	0028	002A
11	002C	002E
12	0030	0032
13	0034	0036
14	0038	003A
15	003C	003E

**Table 6-3. XOP Vector Locations.**

XOP Number	Memory Address Containing New Workspace Pointer Value	Memory Address Containing New Program Counter Value
0	0040	0042
1	0044	0046
2	0048	004A
3	004C	004E
4	0050	0052
5	0054	0056
6	0058	005A
7	005C	005E
8	0060	0062
9	0064	0066
10	0068	006A
11	006C	006E
12	0070	0072
13	0074	0076
14	0078	007A
15	007C	007E

memory locations called *interrupt vector* locations. There is a different set of memory locations for the new PC and WP values corresponding to each interrupt priority code. These locations are listed in Table 6-2.

Having located the new PC and WP values, the computer next saves the old PC, the old WP, and the contents of the status register (ST) in the new workspace registers 13, 14, and 15, respectively. The computer also loads the new PC and WP values at this time and proceeds to execute the so-called *service routine* that corresponds to the interrupt. When interrupt processing is completed, the old PC, WP, and ST contents are restored.

### Extended Operations

Just as memory locations are reserved by the 9900 to be used as interrupt vectors, other memory locations are reserved for what are called *extended operation* (XOP) vectors, or memory locations which contain new WP and PC values.

The 9900 programmer has the option of defining up to 16 subprograms which may be called by a single XOP instruction. Each XOP routine is assigned a number by the programmer who stores corresponding WP and PC values in the reserved memory locations prior to calling the routine. The 16 XOP vectors are given in Table 6-3. Note that there are two vectors per XOP, one for the new WP value and one for the new PC value.



## Status Register

The 9900 contains a 16-bit status register (ST). These bits are defined as follows:

☐ Bit 0 LGT, logical greater than. Set to one when an unsigned number (0 to 65,536 decimal) is compared to another unsigned number and the first number is greater than the second.

☐ Bit 1 AGT, arithmetic greater than. Set to one when a signed number (−32,768 to +32,767) is compared to another signed number and the first number is less positive than the second number.

☐ Bit 2 EQ, equal to. Set to one when one number is compared to another number and the first number is equal to the second number.

☐ Bit 3 C, carry. Set to one when an arithmetic operation results in a carry. Also set to one or cleared to zero in a shift or rotate operation, depending on the state of the bit transferred to the carry bit location.

☐ Bit 4 OV, overflow. Set to one when the result of an arithmetic operation results in a number too large or too small to be correctly represented in 2s complement form.

☐ Bit 5 OP, odd parity. Set to one when the number of one bits in the result is odd. For example, if the result was the number 0110001010110001 (62D1 hexadecimal), then the OP bit would be set to one because the number of one bits in the result is 7, an odd number.

☐ Bit 6 X, extended operation. Set to one when the XOP instruction is used.

☐ Bits 7-15 of the status register are not status bits in the usual sense. Bits 7-11 are reserved for TI Model 990/10 computer applications. Bits 12-15 is the 4-bit storage location for the interrupt mask.

## INSTRUCTION SET

The following section contains a brief description of the 9900 assembly language instructions. Detailed information on each instruction is contained in Appendix A. The instruction set is divided into nine groups: data transfer, arithmetic, comparison, logic, shift, conditional jump, CRU, and control.

### Data Transfer Instructions

**LI, Load Immediate.** Example: LI R7,5. The 16-bit binary equivalent of the decimal number 5 is loaded into workspace regis-

ter 7. Note that the first operand (R7 in this example) must be a workspace register.

**LIMI, Load Interrupt Mask Immediate.** Example: LIM I 5. The 4-bit binary equivalent of the decimal number 5 is loaded into bits 12-15 of the status register. Interrupt priority codes 0-5 are enabled, codes 6-15 will be ignored.

**LWPI, Load Workspace Pointer Immediate.** Example: LWPI >70B8. The number 70B8 is loaded into the WP register. Thus 70B8 becomes the address of register 0 of the workspace. Note that the symbol > is used to denote hexadecimal representation in 9900 assembly language. The absence of the > before a number indicates that the number is in decimal representation.

**MOV, Move Word.** Example: MOV R1, R3. The 16-bit contents of register 1 is moved to register 3. The result is that registers 1 and 3 have identical contents.

**MOVB, Move Byte.** Example: MOVB@>7D00, R1. The 8-bit contents of memory location 7D00 is moved to the upper (or leftmost or most significant) 8-bit byte of register 1. Note that the symbol@denotes the symbolic addressing mode in 9900 assembly language. Addressing modes will be discussed in the next section.

**SWPB, Swap Bytes.** Example: SWPB R3. The most significant byte of register 3 is moved to the least significant byte position of register 3 and the least significant byte is moved to the most significant byte position.

**STST, Store Status.** Example: STST R3. The contents of the status register are stored in workspace register 3. Note that the operand (R3 in this example) must be a workspace register.

**STWP, Store Workspace Pointer.** Example: STWP R3. The contents of the WP register are stored in workspace register 3. Note that the operand must be a workspace register.

## Arithmetic Instructions

**A, Add Words.** Example: A R1, R2. The word stored in register 1 is added to the word stored in register 2 and the sum is stored in register 2. Hence, the previous contents of register 2 are copied over.

**AB, Add Bytes.** Example: AB R1, R2. The most significant byte of register 1 is added to the most significant byte of register 2 and the sum is stored in the most significant byte position of register 2.

**AI, Add Immediate.** Example: AI R1,>C. The number 000C (12 in decimal) is added to the contents of register 1 and the sum is

stored in register 1. Note that the first operand (R1 in this example) must be a workspace register.

**S, Subtract Words.** Example: S@>7E00,@>7E02. The word at memory location 7E00 is subtracted from the word at memory location 7E02 and the difference is stored at memory location 7E02.

**SB, Subtract Bytes.** Example: SB@>7301, R1. The byte at memory location 7E01 is subtracted from the most significant byte of register 1 and the difference is stored in the most significant byte position of register 1.

**INC, Increment.** Example: INC R1. The number 1 is added to the contents of register 1 and the sum is stored in register 1.

**INCT, Increment by Two.** Example: INCT R1. The number 2 is added to the contents of register 1 and the sum is stored in register 1.

**DEC, Decrement.** Example: DEC@>7E00. The number 1 is subtracted from the contents of memory location 7E00 and the difference is stored in memory location 7E00.

**DECT, Decrement by Two.** Example: DECT R4. The number 2 is subtracted from the contents of register 4 and the difference is stored in register 4.

**NEG, Negate.** Example: NEG@>7E00. The data in memory location 7E00 is replaced by its 2s complement.

**ABS, Absolute Value.** Example: ABS R5. The data in register 5 is replaced by its absolute value.

**MPY, Multiply.** Example: MPY@>7D00, R5. The 16-bit data in memory location 7D00 is multiplied by the 16-bit data in register 5. The 16 most significant bits of the 32-bit product are stored in register 5 and the 16 least significant bits of the 32-bit product are stored in register 6. Note that the second operand (R5 in this example) must be a workspace register and that the result is stored in the designated register and the designated register plus one.

**DIV, Divide.** Example: DIV R4, R5. The 32-bit data contained in registers 5 and 6 (register 5 containing the 16 most significant bits) is divided by the 16-bit data in register 4. The 16-bit quotient is stored in register 5 and the remainder is stored in register 6. Note that the second operand (R5 in this example) must be a workspace register.

## Comparison Instructions

**C, Compare Words.** Example: C R1, R2. The 16-bit data in

register 1 is compared to the 16-bit data in register 2. The comparison is done on both a signed and unsigned number basis. On a signed number basis, if the data in register 1 is more positive than the data in register 2, then the AGT (arithmetic greater than) status bit is set to one. On an unsigned number basis, if the data in register 1 is greater than the data in register 2, then the LGT (logical greater than) status bit is set to one. In either case, if the 16-bit data in register 1 is identical to the 16-bit data in register 2, then the EQ (equal) status bit is set to one.

**CB, Compare Bytes.** Example: CB@>7D01, R2. The 8-bit data in memory location 7D01 is compared to the most significant byte of register 2.

**CI, Compare Immediate.** Example: CI R9,>F330. The 16-bit data in register 9 is compared to F330. Note that the first operand must be a workspace register.

**COC, Compared Ones Corresponding.** Example: COCR1, R2. The data in register 1 is compared to the data in register 2. If for each one bit (a bit with a value of one) in register 1 there is a one in register 2 in the same position (bit 3 in both registers equals one, for example), then the EQ (equal) status bit is set to one. Note that the second operand (R2 in this example) must be a workspace register.

**CZC, Compare Zeros Corresponding.** Example: CZC R1, R2. The data in register 1 is compared to the data in register 2. If for each one bit (a bit with a value of one) in register 1 there is a zero in register 2 in the same position (bit 3 of register 1 equals one and bit 3 of register 2 equals zero, for example), then the EQ (equal) status bit is set to one. Note that the second operand (R2 in this example) must be a workspace register.

## Logic Instructions

**ANDI, AND Immediate.** Example: ANDI R0,>6D03. The 16-bit data in register 0 is logically ANDed with the data value 6D03 on a bit by bit basis, and the result is placed in register 0. Note that the first operand must be a workspace register and that the second operand must be a data value.

**ORI, OR Immediate.** Example: ORI R5,>6D03. The 16-bit data in register 5 is logically ORed with the data value 6D03 on a bit by bit basis, and the result is placed in register 5. Note that the first operand must be a workspace register and that the second operand must be a data value.

**XOR, exclusive-OR.** Example: XOR@>7E00, R2. The con-

tents of memory location 7E00 is logically exclusive-ORed with the contents of register 2 and the result is placed in register 2. Note that the second operand must be a workspace register.

**INV, Invert.** Example: INV R1. The contents of register 1 is logically inverted and the result is placed in register 1. This means that all ones in the register are changed to zeros and all zeros are changed to ones.

**CLR, Clear.** Example: CLR R1. 0000 is placed in register 1.

**SETO, Set to One.** Example: SETO R1. FFFF (1111111111111111 in binary) is placed in register 1.

**SOC, Set Ones Corresponding.** Example: SOC R3, >7E00. The contents of register 3 is logically ORed with the contents of memory location 7E00 and the result is placed in memory location 7E00. (Why didn't TI just call this instruction OR?)

**SOCB, Set Ones Corresponding—Byte.** SOCB R5,R8. The most significant byte of register 5 is logically ORed with the most significant byte of register 8 and the result is placed in the most significant byte position of register 8. (Why not ORB or OR Byte?)

**SZC, Set Zeros Corresponding.** Example: SZC R5, R3. For each one in register 5 the corresponding bit in register 3 is reset to zero. Suppose bit 7 of register 5 was equal to one. This instruction would reset bit 7 of register 3 to zero.

**SZCB, Set Zeros Corresponding—Byte.** Example: SZCB @>7E00,@>7E01. For each one in the byte at memory location 7E00, the corresponding bit in the byte at memory location 7E01 is reset to zero.

## Shift Instructions

**SRA, Shift Right Arithmetic.** Example: SRA R1,6. The contents of register 1 is shifted to the right one bit six times. Bit 0 (the leftmost, most significant bit for TI microprocessors) is shifted to bit 6, bit 1 is shifted to bit 7, and so forth. Vacated bit positions are filled with the starting value (one or zero) of bit 0. In this example, the carry status bit will contain the value of bit 10 of register 1's original contents. Note that the first operand must be a workspace register. The second operand is the shift count and is a number between 0 and 15. If the number is zero, then the shift count equals the value of the four least significant bits of register 0. If the four least significant bits of register 0 equal 0, then a 16-bit shift will be performed.

**SLA, Shift Left Arithmetic.** Example: SLA R10,5. The

contents of register 10 are shifted to the left one bit five times. Vacated bit positions are filled with zeros. The carry status bit is equal to the value of the last bit shifted out to the left.

**SRL, Shift Right Logical.** Example: SRL R0,3. The contents of register 0 is shifted to the right one bit three times. Vacated bit positions are filled with zeros. The carry status bit is equal to the value of the last bit shifted out to the right.

**SRC, Shift Right Circular.** Example: SRC R2,7. The contents of register 2 is shifted to the right one bit seven times. Bit 15 (the rightmost, least significant bit for TI microprocessors) is transferred to bit 0 each time a shift occurs. Hence, the contents of the specified register are said to be *circulated* or *rotated*. The carry status bit is not in the loop but will contain the value of the last bit shifted out to the right.

### Unconditional Branch Instructions

**B, Branch.** Example: B@>2166. The memory address 2166 is loaded into the program counter and becomes the address of the next instruction to be executed.

**BL, Branch and Link.** Example: BL@>7D00. The memory address 7D00 is loaded into the program counter and the old value of the program counter is stored in workspace register 11.

**BLWP, Branch and Load Workspace Pointer.** Example: BLWP@>2100. The data at memory location 2100 is loaded into the workspace pointer register. The data at memory location 2102 is loaded into the program counter. The old values of the workspace pointer and program counter are stored in the new workspace registers 13 and 14, respectively. The contents of the status register are stored in register 15.

**XOP, Extended Operation.** Example: XOP@>7E00,2. Extended operation number 2 is called. The 16-bit contents of memory address 0048 is loaded into the workspace pointer register. The 16-bit contents of memory address 004A is loaded into the program counter. (See Table 6-2 for XOP vector locations.) The 16-bit contents of memory location 7E00 is loaded into the new workspace register 11. The old workspace pointer, program counter, and status are stored in the new workspace registers 13, 14, and 15, respectively. Note that the first operand is a memory address of a data value to be passed to and used by the XOP subprogram. (A dummy address must be used if the subprogram does not require a variable to be passed to it.) The second operand is the XOP subprogram number between 0 and 15.

**RTWP, Return with Workspace Pointer.** No operands in the instruction operand field. The contents of registers 13, 14, and 15 are loaded into the workspace pointer register, the program counter, and the status register, respectively.

**JMP, Unconditional Jump.** Example: **JMP -5**. The program counter is incremented by 2 and decremented by 10. Note that the basic memory word width of the 9900 is 16 bits. Instructions have even addresses and, thus, the value of the program counter is always an even address. Also, after an instruction is fetched (and before it is executed) the program counter is incremented by two. The operand in the jump instruction is called the displacement and is the relative number of program counter addresses forward (plus sign) or backward (minus sign) from the value of the program counter after the instruction has been fetched. Suppose our example instruction (**JMP -5**) was located at memory address 7D18. Before execution the program counter equals 7D1A. Then the program counter is decreased by 2 five times and equals 7D10. Schematically we can see that 7D10 corresponds to a displacement of -5 for this example:

Program Counter	Displacement
7D10	-5
7D12	-4
7D14	-3
7D16	-2
7D18	-1
7D1A	0

No doubt this is confusing. You will see, however, that in assembly language you can label a memory location with a mnemonic (such as J1, for example). Thus, when you use the **JMP** instruction with a label (**JMP J1**, for example), the assembler will compute the proper machine code for the displacement.

**X, Execute.** Example: **X@>7D00**. The instruction located at 7D00 is executed.

### Conditional Jump Instructions

Conditional jumps to other locations in the program occur only if certain status bits meet the condition required by the conditional jump instruction. Conditional jump instructions have the same form as the unconditional jump (**JMP**) instruction. For each of the following instructions, the operand is a displacement value as explained

for the JMP instruction. Therefore, I will not use examples in this section.

**JH, Jump if Higher.** A jump will occur only if LGT (the logical greater than status bit) equals 1 and if EQ (the equal status bit) equals 0.

**JL, Jump if Lower.** A jump will occur only if LGT equals 0 and if EQ equals 0.

**JHE, Jump if Higher or Equal.** A jump will occur only if LGT equals 1 or EQ equals 1.

**JLE, Jump if Lower or Equal.** A jump will occur only if LGT equals 0 or EQ equals 1.

**JGT, Jump if Greater Than.** A jump will occur only if AGT (the arithmetic greater than status bit) equals 1.

**JLT, Jump if Less Than.** A jump will occur only if AGT equals 0 and EQ equals 0.

**JEQ, Jump if Equal.** A jump will occur only if EQ equals 1.

**JNE, Jump if Not Equal.** A jump will occur only if EQ equals 0.

**JOC, Jump On Carry.** A jump will occur only if C (the carry status bit) equals 1.

**JNC, Jump on No Carry.** A jump will occur only if C equals 0.

**JNO, Jump on No Overflow.** A jump will occur only if OV (the overflow status bit) equals 0.

**JOP, Jump if Odd Parity.** A jump will occur only if OP (the odd parity status bit) equals 1.

## **CRU Instructions**

**SBO, Set Bit to Logic One.** Example: SBO 15. A logic one is stored in an external latch. The address of the latch (single-bit storage location) is the sum of the number 15 and the 12-bit address stored in bits 3 through 15 of workspace register 12. The operand (15 in this example) is called the displacement value and is a number between -128 to +127.

**SBZ, Set Bit to Logic Zero.** Example: SBZ 2. A logic zero is stored in an external latch. The address of the latch is the sum of the number 2 and the 12-bit address stored in bits 3 through 14 of workspace register 12.

**TB, Test Bit.** Example: TB 4. The logic state of the selected single-bit input line is stored in the EQ (Equal) status bit location. The address of the selected input line is the sum of the number 4 and the 12-bit address stored in bits 3 through 14 of workspace register 12.



**LDCR, Load CRU.** Example: LDCR@>7E00, 9. The 9 least significant bits of the 16-bit data stored at memory location 7E00 is transferred to external single-bit latches. The least significant bit (bit 15 for TI microprocessors) is transferred to the latch designated by the 12-bit address (the *base address*) stored in bits 3 through 14 of workspace register 12. The next least significant bit (bit 14) is transferred to the latch designated by the base address plus one, and so forth. Note that the operand (9 in this example) is the number of bits to be transferred and must be a number between 0 and 15. If the number is 0 then a 16-bit transfer will be performed.

**STCR, Store CRU.** Example: STCR@>7E02, 5. The logic states of 5 successive external lines are transferred to memory location 7E02. The address of the first line is designated by the 12-bit address (the base address) stored in bits 3 through 14 of workspace register 12. The logic state of the first line is stored in the least significant bit (bit 7) of the byte located at 7E02. The logic state of the next line (base address plus one) is stored in the next least significant bit (bit 6), and so forth.

## Control Instructions

See Appendix A for details on these instructions. TI's Editor/Assembler manual recommends that these instructions not be used on the TI-99/4A home computer even though the instructions are recognized and assembled. These instructions will not be used in this book.

**LREX, Load or Restart Execution.**

**CKOF, Clock Off.**

**CKON, Clock On.**

**RSET, Reset.**

**IDLE, Idle.**

## ADDRESSING MODES

The 9900 has seven addressing modes: immediate, register direct, register indirect, register indirect with autoincrement, symbolic (memory direct), indexed, relative.

### Immediate

Seven instructions use the immediate addressing mode:

☐ LI, Load Immediate.

☐ LIML, Load Interrupt Mask Immediate.

- ☐ LWPI, Load Workspace Pointer Immediate.
- ☐ AI, Add Immediate.
- ☐ CI, Compare Immediate.
- ☐ ANDI, AND Immediate.
- ☐ ORI, OR Immediate.

Example: LI R1 >,A70C. The data A70C is loaded into workspace register 1. This instruction requires two words of memory. The first word contains the machine code for LI R1 and the memory word *immediately* following contains the number A70C. Note that no other addressing modes are allowed with the immediate instructions. Also, no other instructions may use the immediate addressing mode.

### Register Direct

If the register direct addressing mode is specified, then the data to be processed is found in the workspace register specified *directly* in the instruction. If the operand is a destination address (where the result will be stored), then the address is specified directly. No special symbols are used to indicate this mode.

Example: MOV R1,R2. The contents of register 1 is moved to register 2. Both operands in this example are specified using the register direct addressing mode.

### Register Indirect

If the register indirect addressing mode is specified, then the address of the data to be processed is found in the specified workspace register. Hence, the location of the data is specified *indirectly*. If the operand is a destination address, than that address is contained in the specified workspace register. Hence, the destination address is specified indirectly. The asterisk (\*) symbol is used to designate the register indirect addressing mode.

Example: MOV \*R1, \*R2. Move data. The source address (the address where the data to be moved is located) is stored in register 1. The destination address (the address to which the data will be moved) is stored in register 2.

### Register Indirect with Autoincrement

If the register indirect with autoincrement mode is specified, then the address of the data to be processed is found in the specified workspace register. Additionally, the address in the specified reg-

ister is incremented after the instruction has been executed. The address will be incremented by one if the data to be processed is a byte. The address will be incremented by two if the data to be processed is a word.

If the operand is a destination address, then that address is contained in the specified workspace register. Additionally, the destination address is either incremented by one or two after the instruction has been executed—by one if the operation is a byte operation, by two if the operation is a word operation.

A plus (+) sign is used to designate autoincrement.

Example: `MOV *R1+, *R2+`. The source address (the address of the data to be processed) is contained in register 1. The destination address (the address of the memory location to which the data will be moved) is contained in register 2. Since this is a word operation, then the contents of both registers will be incremented by two after the instruction is executed.

## Symbolic

If symbolic addressing is specified, the data to be processed is found at the memory location specified directly in the instruction. If the operand is a destination address, then that address is specified directly in the instruction. TI uses the @ symbol to indicate this mode. Symbolic addressing is also called memory direct addressing.

Example: `MOV@ >7E00, R1`. The data at memory location 7E00 is moved to workspace register 1.

Example: `MOV R1, @>7E00`. The data in register 1 is moved to memory location 7E00.

## Indexed

If indexed addressing is specified, then the source or destination address is formed by adding a constant to the contents of a workspace register (called the *index register*). The constant is preceded by an @ sign and the workspace register to be used as the index register is enclosed in parentheses.

Example: `MOV R1, @2(R8)`. The data in register 1 is moved to memory. The destination address is formed by adding the number 2 to the contents of register 8.

Note that any workspace register may be used as an index register except register 0.

## **Relative**

Relative addressing is used by the JMP(unconditional jump) instruction and the twelve conditional jump instructions. In these instructions, the new program counter address is specified in terms of the number of program addresses forward (plus sign) or backward (minus sign) from the value of the program counter after the instruction has been fetched. The number of addresses forward or backward is called the displacement. In general terms, the new program counter address (the jump address) is the value of the program counter plus two plus twice the displacement. The displacement value is limited to the range  $-128$  to  $+127$ .

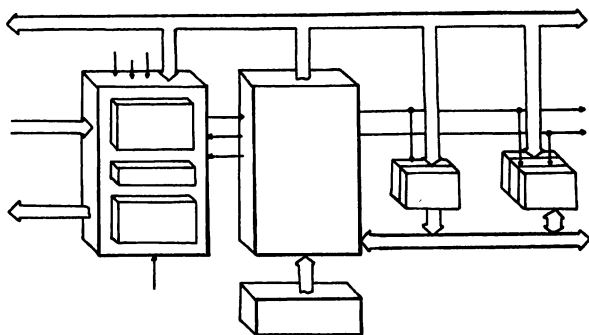
Example: JMP  $-5$ . The program counter is incremented by two and then decremented by ten.

## **SUMMARY**

The primary architectural features, instruction set, and addressing modes of the 9900 have been discussed.

The 9900 is a 16-bit microprocessor, addresses up to 65,536 bytes of memory, performs both byte and word operations, and uses memory-to-memory architecture (no internal accumulators—uses external workspace registers in memory). It has 69 instructions and 7 addressing modes.

## Chapter 7



### The TI-99/4A

The TI-99/4A is one of the best bargains in the home computer marketplace. While the price has undergone many swings and TI has announced that they will no longer produce the machine, it offers more capabilities for the money than any other home computer.

It was \$199 after a \$100 rebate when I bought mine. For my \$199 I got the console, video interface cable and RF modulator, power supply module and power cable, 193-page *User's Reference Guide*, and a 143-page *Beginner's BASIC* teaching manual.

Internally, the TI-99/4A home computer has a 16-bit microprocessor (the TMS9900), something no other home computer on the market has at this time. The VIC-20, the Radio Shack Color Computer, the Sinclair, the ATARI 400/800, and the Commodore-64 all have 8-bit microprocessors. Even in the higher priced machines (\$1000-5000) which are sometimes bought for the home (though generally they are bought for business), very few have a 16-bit microprocessor. The IBM Personal Computer is perhaps the most popular, and it has the Intel 8088 16-bit microprocessor, but it costs at least twenty times more than the TI. Obviously, the IBM PC is the better computer; but if you're looking for a low-priced 16-bit computer, the TI-99/4A is the one.

Looking at the TI-99/4A home computer from the standpoint of learning how to write assembly language programs, you can see that this machine is a very good choice for a low cost learning system. It

is possible to buy an inexpensive 8-bit microprocessor learning systems, such as the Heathkit ET3400 or even the VIC-20 if you buy the assembly language plug-in cartridge, but I know of no 16-bit microprocessor learning system other than the TI-99/4A at a similar price.

Intel offers a system design kit, the SDK-86, for the 8086 16-bit microprocessor (the 8088 used in the IBM PC Personal Computer is an 8-bit data bus version of the 8086 but is still considered a 16-bit processor because internally it has the 8086 16-bit data bus and ALU). This kit costs about \$700 and is a stripped down system—no power supply included, no cover (all components exposed), and most of all no assembler. All assembly language programs must be translated by the programmer into machine code and entered via a hexadecimal numeric keypad.

TI offers a similar kit for the TMS9995, a newer enhanced version of the 9900 having an internal 256 byte RAM, internal clock generator, internal timers, and an 8-bit data bus. This kit does come with an assembler in ROM but still costs about \$500. A power supply is not included, and all components are exposed.

Neither of these kits is for the first time assembly language programmer. These kits are actually low cost software development and prototyping systems for companies who are in the business of developing microcomputer-based products. By low cost, I mean low compared to the more sophisticated software development systems and in-circuit emulators which cost from \$20,000 to \$50,000.

So, for 16-bit microprocessor students with a limited budget, the TI-99/4A is currently the only choice. The only additional cost to be able to learn assembly language on the TI home computer is

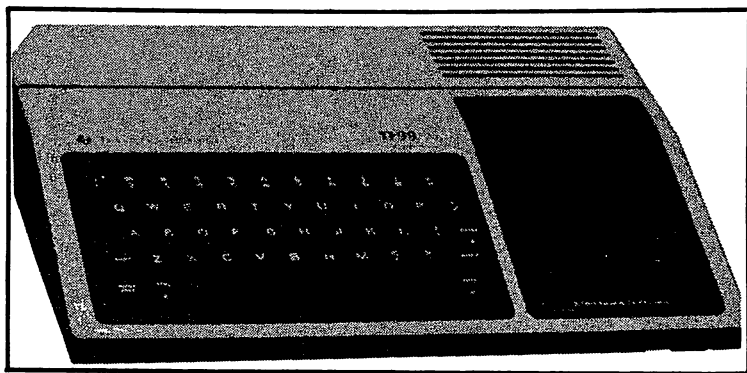


Fig. 7-1. The TI-99/4A home computer console.

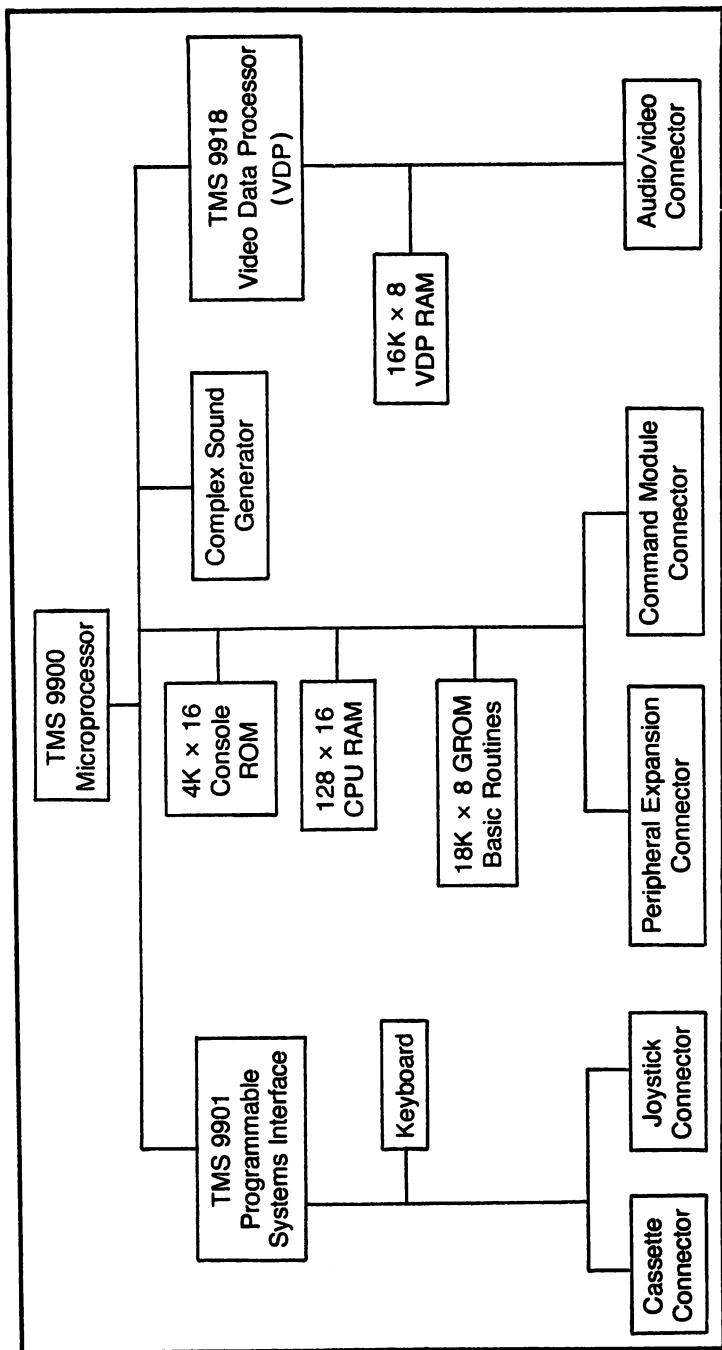


Fig. 7-2. Simplified block diagram of the TI-99/4A home computer console.

**Table 7-1. CPU Memory Map.**

First Address (hexadecimal)	Last Address (hexadecimal)	Description	Number of bytes
0000	1FFF	Console ROM	8,192
2000	3FFF	Memory Expansion	8,192
4000	5FFF	Peripheral Expansion	8,192
6000	7FFF	Command Module ROM/RAM	8,192
8000	9FFF	CPU RAM & Memory Mapped Devices	8,192
A000	FFFF	Memory Expansion	24,576

the cost of the Mini Memory plug-in command module. This module costs less than \$100 (mine was \$84 plus tax). It comes with a Line-by-Line Symbolic Assembler on cassette tape.

A picture of the TI-99/4A home computer console is shown in Fig. 7-1 and a simplified block diagram of the internal organization of the console is shown in Fig. 7-2. In the rest of this chapter, I will discuss the memory organization of the TI-99/4A, which has not one, but three separate memories. In particular, I will focus on how memory locations are allocated within the computer. Such information is essential for the assembly language programmer. Memory allocation will be shown in tables that are called *memory maps*.

## CPU MEMORY

The CPU memory map is shown in Table 7-1. This memory contains up to 65,536 locations with each location containing 8 bits, or one byte. 8,192 bytes (or 4,096 words of 16-bit data) are contained in the *console ROM* (see Fig. 7-2) and 256 bytes (128 words) are contained in the *CPU RAM*.

The address of the first memory location in console ROM is 0000, and the last address is 1FFF (hexadecimal for 8191). This ROM is called the system monitor ROM and controls the basic computer operation; it displays the so-called START screen on your TV set or monitor, allows you to get into BASIC, and so forth.

The address of the first memory location in the CPU RAM is 8300 (hex), the last address is 83FF (hex). This 256 byte read/write memory is called a *scratch pad* and is used by various programs as a workspace register area and as a general purpose temporary storage area for variables.

Another area of the CPU Memory is allocated for memory-mapped devices. Although space has been reserved for 7,936 bytes, only 11 addresses have been decoded internally. These addresses



and their use are shown in Table 7-2. Note that these addresses are not the addresses of specific 8- or 16-bit locations, such as ROM, RAM, or general purpose registers. TI uses these decoded address lines essentially as control lines to read and write data or address information from or to one or more registers contained in these memory-mapped devices.

Four devices are controlled by these 11 decoded address lines in conjunction with the data bus. Three of the four devices are in the console: the TMS9918A Video Data Processor, the Complex Sound Generator, and the graphics read only memory (GROM). The one external device for which there is internal address decoding is the speech module.

The remaining CPU Memory is reserved for memory expansion (addresses 2000-3FFF and A000-FFFF), peripheral expansion (4000-5FFF) and command module ROM/RAM (6000-7FFF). All of these are outside the console.

Memory expansion and peripheral expansion are accomplished by connecting external memory (ROM or RAM) or peripheral devices (such as an RS-232 interface for a printer) to the 44-pin input/output port on the right side of the console. TI sells a Peripheral Expansion System to facilitate expansion. This box has its own power supply and can hold up to seven accessories plus one disk drive.

Command module ROM/RAM is connected to the system through the command module slot just to the right of the keyboard. See Fig. 7-3.

## VIDEO DATA PROCESSOR RAM

The second memory in the TI-99/4A is the Video Data Proces-

**Table 7-2. Memory Mapped Devices.**

Address (hexadecimal)	Function
8400	Sound
8800	VDP Read Data
8802	VDP Read Status
8C00	VDP Write Data
8C02	VDP Write Address
9000	Speech Read
9400	Speech Write
9800	GROM Read Data
9802	GROM Read Address
9C00	GROM Write Data
9C02	GROM Write Address

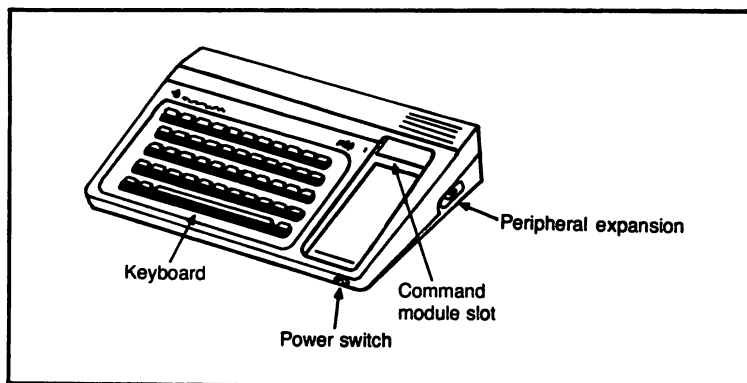


Fig. 7-3. Front view of console.

sor (VDP) RAM. This memory is completely separate from the CPU Memory and has its own address space. It is completely controlled by the TMS9918A Video Data Processor integrated circuit and contains 16,384 bytes of read/write memory. This memory is the so-called 16K RAM you read and hear about in the advertisements for the TI-99/4A home computer.

The VDP RAM contains the current data for the video display. Everything you see on your screen is contained in binary form in the VDP RAM and was placed on the screen by the Video Data Processor integrated circuit. This memory contains the pattern descriptor table (used for defining up to 256 patterns or characters), the color table, screen image table (which specifies the characters that occupy each of the 768 screen positions), sprites (moving graphics), sprite descriptor table (similar to the pattern descriptor table), and sprite motion table. See Table 7-3.

Table 7-3. VDP RAM Memory Map.

First Address	Last Address	Description	Number of Bytes
0000	02FF	Pattern Name Table	768
0300	037F	Sprite Attribute List	128
0380	03FF	Pattern Color Table	128
0400	077F	Sprite Descriptor Blocks	896
0780	07FF	Sprite Velocity Table	128
0800	0FFF	Pattern Generator Area	2048
1000	137F	Free Memory Space	896
1380	34FF	Program File Load Buffer Area	8576
3500	3FFF	Reserved for Disk Device Service Routine	2816

**Table 7-4. VDP RAM with BASIC Interpreter.**

First Address	Last Address	Description	Number of Bytes
0000	02FF	Screen	768
0300	031F	Color and Sprite Table	32
0320	03BD	Crunch Buffer	158
03BE	03FF	BASIC Temporaries and Interpreter Roll-Out Area	66
0400	05FF	Character Tables	512
0600	3FFF	BASIC Tables and Crunched Program	14,848

When BASIC is in use, the VDP RAM also contains the user's program and all the information required by the BASIC Interpreter program (recall that BASIC is itself a program that allows you to write in BASIC language) in order to convert the user's program to machine code. See Table 7-4.

Keep in mind that the VDP RAM space cannot be accessed directly. This RAM is controlled by the VDP chip which interfaces to the 9900 microprocessor via the data bus and four decoded address lines which function as read/write control lines—to write data to and read data from special purpose registers inside the VDP chip. Thus, the assembly-language programmer may read and modify the VDP RAM indirectly by accessing the VDP registers.

Another way to access the VDP RAM is the use of the EASY BUG debugging program, which I will discuss in the next chapter.

## **GRAPHICS READ ONLY MEMORY**

The third memory in the TI-99/4A home computer is the graphics read only memory (GROM). This memory is completely separate from the CPU Memory and the VDP RAM and has its own address space.

The TI-99/4A console contains three GROM chips, each with 6,144 bytes of ROM. These chips are not standard ROM devices which are interfaced to a microprocessor via the address and data busses. These GROM chips, which are unique to TI as far as I know, have their own 13-bit program counter inside. Also, like the VDP RAM, they interface to the 9900 through four decoded address lines (see Table 7-2) which function as read/write control lines.

The three GROM chips in the console contain the BASIC

language routines. See the box labeled “18K × 8—BASIC Routines” in Fig. 7-2. Up to five more GROM chips may be added externally via the command module slot, giving the computer an additional 30K bytes of memory.

## CRU BITS

In addition to the three memories just discussed, up to 4096 single-bit lines or storage locations may be addressed by the 9900’s communications register unit (CRU).

Only one device in the TI-99/4A home computer console communicates with the 9900 through the CRU—the TMS9901 Programmable Systems Interface chip. This chip is used to interface the keyboard, cassette tape recorder (external), and joysticks (external) to the 9900 microprocessor. The TMS9901 contains all the interface circuitry required to address 32 input/output lines.

**Table 7-5. Allocation of TMS 9901 CRU Bits.**

CRU Bits	Functions
0	Control
1	External
2	VDP Vertical Synchronization
3	Clock interrupt, keyboard ENTER line, joystick fire button
4	Keyboard “L” line, joystick left
5	Keyboard “P” line, joystick right
6	Keyboard “0” (zero) line, joystick down
7	Keyboard SHIFT line, joystick up
8	Keyboard SPACE line
9	Keyboard “Q” line
10	Keyboard “L” line
11	Not used
12	Reserved
13-15	Not used
16	Reserved
17	Reserved
18	Bit 2 of keyboard select
19	Bit 1 of keyboard select
20	Bit 0 of keyboard select
21	ALPHA LOCK
22	Cassette control 1
23	Cassette control 2
24	Audio gate
25	Magnetic tape output
26	Reserved
27	Magnetic tape input
28-31	Not used

CRU Bits	Functions
0-2047	Internal
2048-2175	Disk Controller
2176-2303	Reserved
2304-2431	RS232, ports 1 and 2
2432-2559	Reserved
2560-2687	RS232, ports 3 and 4
2688-2815	Reserved
2816-2943	Reserved
2944-3071	Thermal Printer
3072-3957	Future Expansion
3958-4095	P-Code Peripheral

**Table 7-6. CRU  
Allocation for the TI-99/4A.**

Each of these lines is individually addressable by the CRU. The TMS9901 CRU bits and their functions are listed in Table 7-5.

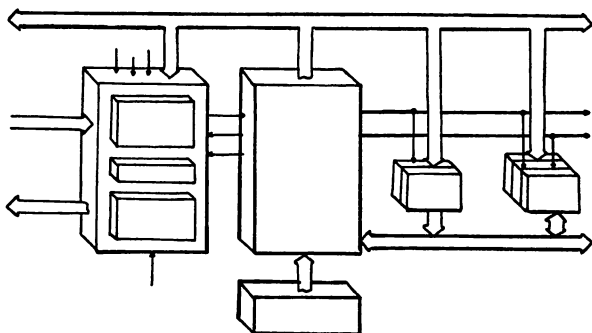
Note that CRU bits 0-15 are input lines. All but bit 0 are also interrupt lines and, when active (logic 0), generate an interrupt request and a priority code which is read by the 9900. (See Table 6-2 for interrupt vectors, the addresses of memory locations which contain new workspace pointers and program counter values which correspond to the priority codes.)

The entire CRU allocation for the TI-99/4A is given in Table 7-6. Bits 0-1023 (decimal) are reserved for internal use. Bits 1024-2047 are reserved for future internal use (new console designs?). And bits 2048-4095 are reserved for peripherals which are connected to the system via the 44-pin port on the right side of the computer console.

## SUMMARY

Before memory expansion the TI-99/4A console includes a little over 8K bytes of CPU Memory, 16K bytes of VDP RAM, and 18K bytes of GROM, for a total of 42K bytes. A fully expanded system could address 128K bytes - 64K bytes of CPU Memory, plus 16K VDP RAM, plus 48K of GROM.

## Chapter 8



### The Mini Memory Module

Unless you knew what you were looking for or took the time to read the small print on the front of the box (the cover of the enclosed manual to be exact), you would never know from its name that the Mini Memory command module is a low-cost, assembly-language development tool. The very name Mini Memory hardly invites a second look to see what this remarkable command module has to offer:

- ☐ 14K bytes of additional memory—4K bytes of ROM (CPU memory address space 6000-6FFF), 4K bytes of RAM (CPU memory address space 7000-7FFF), and 6K bytes of GROM.

- ☐ 7 additional BASIC subprograms contained in the GROM.

- ☐ Built-in battery. You may save either BASIC or assembly language programs on the Mini Memory module instantaneously (versus slowly on cassette). The module should not be used as a long-term storage medium but is excellent for temporary (days or longer if not statically discharged) storage, especially during program development. Data is saved even when the module is removed from the console.

- ☐ The EASY BUG program in the ROM, most useful for executing and debugging your assembly language programs.

- ☐ The Line-by-Line Symbolic Assembler, which is stored on cassette tape and is loaded into the Mini Memory RAM.

All this for less than \$100. This may not seem a bargain for

many TI home computer owners, but for the person interested in writing or learning how to write assembly-language programs, the Mini Memory module with the Line-by-Line Assembler is an excellent low-cost program development tool and educational device.

The alternative method of writing assembly-language programs on the TI-99/4A is to use the Editor/Assembler software package which costs about the same as the Mini Memory module but cannot be used unless your system includes the following accessories:

- ☐ Peripheral Expansion System, about \$200
- ☐ Disk Drive, about \$350
- ☐ Disk Drive Controller, about \$200
- ☐ Memory Expansion Unit (32K bytes of RAM), about \$250

The total, about \$1000, is more than ten times the cost of either the Mini Memory module or the Editor/Assembler alone.

There is no doubt that the Editor/Assembler is the more powerful assembly-language development software and that the disk system with the ability to write and save multiple files is better. It just costs more. And chances are, if you buy all the required accessories, then you will probably want a printer (about \$650, if you buy TI's) which requires the RS-232 interface (about \$150).

I do not recommend that you buy the Editor/Assembler and the required accessories unless you are an experienced programmer (having written assembly-language programs for at least one type of microprocessor) and have a lot of money. I suspect that the average owner of the TI home computer will never buy the big system. Thus, for the average owner, the Mini Memory module is an excellent choice and a good place to begin to learn about assembly language. If you are an average owner and later on you lose interest in assembly language (and many will just because it is harder than BASIC and takes more time to learn and has fewer applications to the average user), then you've only invested about \$100 instead of \$1000 or \$1800.

Having said all that, I suggest that if you haven't bought the Mini Memory module yet that you go out and get one now so that you can follow along as I discuss the Mini Memory module and the Line-by-Line Assembler.

## **LOADING THE LINE-BY-LINE ASSEMBLER**

To load the Line-by-Line Assembler, perform the following steps:

1. With power off, properly connect your cassette tape recorder to the console. In case you can't remember, hook up the CS1 wires as follows: Red wire to the microphone jack, Black wire to the remote jack, and White wire to the earphone jack.
2. Insert the Line-by-Line Assembler cassette into the tape recorder.
3. Plug in the Mini Memory module into the command module slot.
4. Turn on the console power.
5. Turn on the television set, channel 3 or 4, and set the rf modulator switch to the modulator position. Make sure the rf modulator channel switch is set to the same channel as the tv. The so-called START screen is displayed.
6. Press any key to begin. The screen looks like this:

**TEXAS INSTRUMENTS  
HOME COMPUTER**

**PRESS  
1 FOR TI BASIC  
2 FOR EASY BUG  
3 FOR MINI MEMORY**

7. Press 3 for Mini Memory. The screen looks like this:

**\* MINI MEMORY \***

**PRESS:  
1 TO LOAD AND RUN  
2    RUN  
3    RE-INITIALIZE**

**© 1981 TEXAS INSTRUMENTS**

LOAD AND RUN applies to assembly language programs developed with the Editor/Assembler package. RUN applies to assembly language programs previously loaded into the Mini Memory RAM. The RUN option also allows you to build new assembly-language programs. More about this shortly.

8. Press 3 to RE-INITIALIZE. This step clears the Mini Memory RAM to accept new programs. Old programs and data are lost. If the memory has never been initialized, then the screen temporarily goes blank and the Mini Memory selection list (shown above) reappears. Otherwise the following screen is displayed:



**\* INITIALIZE MEMORY \***  
**MEMORY ALREADY INITIALIZED**  
**HIT "PROC'D" TO CONFIRM**

If this screen is displayed, then press PROC'D (the FUNC and 6 keys at the same time).

9. Press QUIT (FUNC and =). The START screen is displayed.
10. Press any key. The master selection list is displayed (as in Step 6).
11. Press 2 for EASY BUG. The screen displays the EASY BUG commands and special function keys shown in Fig. 8-1.
12. Press any key. A question mark will appear at the lower left hand corner of your screen.
13. Press the letter L.
14. Follow the displayed directions to load the data (the Line-by-Line Assembler and the NEW, OLD, and LINES programs) from the cassette to the Mini Memory RAM. If everything went right the screen looks like this:

== COMMAND TYPES ARE ==	
MXXXX	MODIFY CPU MEMORY
GXXXX	DISPLAY GROM MEMORY
VXXXX	MODIFY VDP MEMORY
EXXXX	EXEC. ASSEMBLY PROGRAM
CXXXX	CRU SINGLE-BIT I/O
SXXXX	SAVE CPU MEMORY TO CS1 (STARTING AT XXXX)
L	LOAD STORAGE FROM CS1
-- SPECIAL FUNCTION KEYS ARE	
AID	DISPLAY THIS SCREEN
PERIOD	ABORT A COMMAND
ENTER	ENTER COMMAND/DATA
MINUS	DISPLAY LAST MEMORY (CURRENT UNCHANGED)
SPACE	DISPLAY NEXT MEMORY (CURRENT UNCHANGED)
*NOTE*	CPU RAM 8370-83FF IS RESERVED FOR EASY BUG

Fig. 8-1. EASY BUG commands and special function keys.

```

?L
* REWIND CASSETTE TAPE      CS1
  THEN PRESS ENTER
* PRESS CASSETTE PLAY      CS1
  THEN PRESS ENTER
* READING
* DATA OK
* PRESS CASSETTE STOP      CS1
  THEN PRESS ENTER
?

```

15. Press QUIT.
16. Press any key.
17. Press 3 for Mini Memory
18. Press 2 to RUN. The screen displays:

```

* RUN *
PROGRAM NAME?

```

Directly below the P in PROGRAM there will be a flashing cursor—a white rectangle.

19. Type NEW and press ENTER. The following screen is displayed:

```

LINE BY LINE ASSEMBLER
© 1982 TEXAS INSTRUMENTS

```

**7D00 045B**

There is also a solid square white flashing cursor in the second space to the right of the B of the 4-digit hexadecimal code 045B. You are looking at the memory address and the present contents of the first memory location in the Mini Memory RAM that is available for new assembly language programs. The assembler is waiting for you to enter your first assembly language instruction. Before we do this, let's discuss the memory allocation of the Mini Memory module.

## **MEMORY MAP**

Notice that the first memory address that the assembler displays is 7D00. Now look at the Mini Memory map shown in Table

**Table 8-1. Mini Memory Map with Line-By-Line Assembler Loaded.**

First Address	Last Address	Segment Description	Number of Bytes
6000	6FFF	ROM Routines	4096
7000	70B6	Reserved RAM	168
70B8	70D6	User Program Workspace	32
70D8	7116	Reserved RAM	64
7118	7CD6	Line-By-Line Assembler	3008
7CD8	7CFE	Symbol Table	40
7D00	7FE6	User Program	760
7FE8	7FEE	User Program Name/Address	8
7FF0	7FF6	OLD Name/Address	8
7FF8	7FFE	NEW Name/Address	8

8-1. This table shows the Mini Memory memory allocation after the Line-by-Line Assembler has been loaded.

The Mini Memory adds 8K bytes to the CPU memory (not the VDP RAM), 4K bytes of ROM, and 4K bytes of RAM. The Line-by-Line Assembler itself uses approximately 3K bytes as you can see from the table. 232 bytes of RAM are reserved for use by the assembler and the *utilities* (subprograms stored in the Mini Memory ROM which are used by the LINES program and may be used by programs that you write). Also, 32 bytes are reserved for the user's workspace, sixteen 16-bit general purpose registers beginning at address 70B8.

The nominal 40-byte symbol table is also reserved for use by the assembler. This table keeps track of any addresses or data values that you have named when you created your assembly language program. These names and their values are called *references* and each one takes 4 bytes of memory—2 bytes for a two-character name and 2 bytes for the value. Thus, nominally you are allowed 10 references. If you go over this amount, the assembler will write over the beginning of your program—that is, if you started your program at address 7D00. Note that there is no requirement to start at 7D00, even though the assembler has been programmed to display this address first.

Program names (such as NEW, OLD, or LINES) are stored at the end of the RAM area. Also, the memory address where the program starts is stored in this area. The *starting address* is the address of the first instruction to be executed. Subprograms or data values could be stored at lower addresses. Therefore, the starting address is not necessarily the address of the first piece of code or

data in the program. For example, the starting address (the address of the first instruction to be executed) for the LINES graphics demonstration program is 7D9E. However, subprograms used by the LINES program are stored in the RAM space 7CD6-7D9C. 7CD6 contains the first RAM code of the LINES program, but 7D9E contains the first instruction to be executed. The starting address is also called the *entry point*.

Up to 6 characters are allowed for a program name. This takes 6 bytes. The starting address takes 2 bytes. Thus, the total for each name/address group takes 8 bytes. When we first load the assembler, the LINES name/address is stored in the space 7FE8-7FEE; the OLD name/address is stored in the space 7FF0-7FF6; and the NEW name/address is stored in the space 7FF8-7FFE. Also, the entire LINES code (subprograms, main program, and data) is stored in the space 7CD6-7FB0. What does this mean? First, there is very little space left over for your program—56 bytes. Second, if you start your program at 7D00, you will be writing over the LINES program. Thus, don't expect LINES to work after you enter your program. To run LINES after you have created a new program you must reload the assembler.

How much RAM area is left for a program? If you start at 7D00, don't use over 10 references, and save 8 bytes for name/address storage, then you have 760 bytes—which is plenty for beginning programs.

Now Let's find out how to use the Line-by-Line Assembler.

## ASSEMBLER SYNTAX

The assembly language instruction line is composed of three sections, or *fields*: the label field, the opcode field, and the operand field.

The Line-by-Line Assembler allows only a two-character wide label, such as J1 or WP. Labels essentially give the memory location a symbolic name that can be referenced later on in the program. The use of labels instead of 4-digit hexadecimal numbers within the program usually reduces the amount of typing, relieves the programmer from some mental bookkeeping, and makes the program easier to read at a later date. The label is conceptually identical to the line number in BASIC. A BASIC program might contain the statement GOTO 10, for example. An assembly language program might contain the instruction JMP J1, where J1 is the label of the instruction which the programmer wants to execute next (instead of the next one in sequence).

Note that the label is optional in assembly language. If you don't want to use a label, just press the space bar and the cursor will move to the next field, which is the opcode field.

The opcode field is from one to four characters wide and contains one of the 69 TMS9900 instruction mnemonics such as MOV, A, ANDI, or MPY. The opcode field always begins in column 4.

The third field is the operand field. It does not start at any particular column number, but is separated from the opcode field by at least one space. The operand field contains one or two operands, depending on the instruction specified in the opcode field. Mnemonics are used in the operand field to specify both the operand and the addressing mode. An operand is either the data to be processed or a workspace register number or memory location where either the data is to be found (the source address) or where the result is to be stored when the operation is completed (the destination address). The addressing mode specifies how the source or destination address is to be determined.

Comments may be placed immediately after the operand field. Normally comments are very useful for program readability. Comments should indicate the intention of the instruction. "Store results" is a better comment than move "R1 to memory XYZ." Good comments are useful both as you write the program and as you try to read the program days or weeks after the program was written when the assembly language mnemonics no longer look familiar.

The TI Line-by-Line Assembler, however, saves neither the assembly-language mnemonics nor the comments, both of which make up what is called the *source code*. After you end a session using the Line-by-Line Assembler, the source code is lost and only the *object code* remains. The object code is the memory addresses and their contents. Thus, unfortunately, there is no value in typing comments in the comment field. However, you should keep your own copy of the source code of your program and you should use comments on your copy, whether handwritten or typed.

Now let's enter an assembly language instruction to see how the Line-by-Line Assembler works. Assuming that the assembler has been loaded and you haven't yet changed any data in the memory, the first line displayed by the assembler looks like this:

7D00 045B

Note the number of 045B. This is a current contents of memory

location 7D00, one of the memory locations used by the LINES program (which we are going to write over). 045B is the machine code for B \*R11, a mnemonic which means branch to the address contained in workspace register 11.

Now press the space bar. The cursor moved to column 4, the beginning of the opcode field. Thus, the code you are about to enter will not have a label.

Type MOV R1,R2. Don't put a period after it and don't press ENTER. The line looks like this:

```
7D00 045B    MOV R1,R2
```

Now press ENTER. The result is as follows:

```
7D00 C081    MOV R1,R2
7D02 C101
```

The assembler changed 7D00's contents from 045B to C081 and then displayed the next memory address and data. The assembler assembles code after each line is entered. That is why it is called a *line-by-line* assembler in contrast to the Editor/Assembler which will assemble an entire file of source code.

Note that the Line-by-Line Assembler displays two bytes of data on each line. Thus the number in the address column increments by two as you build or step through a program. In the above display, the byte of data at 7D02 is C1. The byte of data at 7D03 is 01. The next address to be displayed is 7D04.

Before I continue, let's put the data back the way it was. Press the space bar, type AORG >7D00, and press ENTER. The result is as follows:

```
7D00 C081    MOV R1,R2
7D02 C101    AORG >7D00
7D00 C081
```

Note that I did not change the contents of 7D02. I only told the assembler to display memory location 7D00 for us again. Now I can restore 7D00's original contents by pressing the space bar, typing B \*R11, then pressing enter:

```
7D00 C081    MOV R1,R2
7D02 C101    AORG >7D00
7D00 045B    B *R11
7D02 C101
```

AORG stands for *absolute origin* and is one of seven directives, or commands recognized by the Line-by-Line Assembler. I will discuss these directives in the next section.

If you want to break at this point, type END in the opcode field, and press ENTER. The assembler displays 0000 UNRESOLVED REFERENCES. Press ENTER. The assembler displays PRESS ENTER TO CONTINUE. Press ENTER. The Mini Memory selection list is displayed. Press QUIT. Turn the tv and the console off.

## ASSEMBLER DIRECTIVES

The following section describes each of the seven directives recognized by the Line-by-Line Assembler.

**AORG—Absolute Origin.** This command tells the assembler to display the address and data of another memory location. This command is used to correct previous entries or to jump ahead to enter more code. For example, if you wanted to locate your data (to be processed by your program) at 7F00 and you were at 7E12 after finishing your program, then you would type AORG >7F00 in the opcode field and press ENTER. The assembler would then display 7F00 and its current contents. You could then enter data using the DATA command.

**BSS—Block Starting with Symbol.** This command is used to reserve a block of memory to be used by your program. Suppose the following line is displayed by the assembler:

```
7D00 XXXX
```

XXXX means that the contents could be anything. Press the space bar to get to the opcode field. Type BSS 32 and press ENTER. The assembler displays the following:

```
7D00 XXXX
```

```
7D20 XXXX
```

```
BSS 32
```

In this example, 32 bytes (or 16 words) are reserved. Odd numbers may be used but the assembler will round up to an even number. This command does not affect data in memory. Note that this command is similar to the AORG command, except that this command causes the assembler to jump ahead (and not back) a designated number of bytes rather than to an explicit or absolute memory location.

**DATA—Enter Data.** This command is used to enter specific data values into the memory. Again, suppose the following assembler display:

```
7D00 XXXX
```

Press the space bar. Type **DATA >6043** and press ENTER. The display now looks like this:

```
7D00 6043
7D02 XXXX
DATA >6043
```

Press the space bar, type **DATA 15** and press ENTER. The result is as follows:

```
7D00 6043    DATA >6043
7D02 000F    DATA 15
7D04 XXXX
```

Note that data values in hexadecimal representation must be preceded by the **>** symbol, otherwise the assembler assumes that the value is in decimal form. The assembler accepts only hexadecimal or decimal. Decimal numbers are converted by the assembler automatically, stored in memory in binary, and displayed in hexadecimal.

Note also that the **DATA** command is used to enter 16-bit data. To enter a single byte at an even address, you must enter two bytes at the same time. Normally, if you did want to enter a single byte at an even address for some reason you would make the second byte zero, although any 8-bit number would work. If you are entering data in hexadecimal, this is an easy operation. For example, if you wanted to enter 0F in location 7D04, you would enter the two byte number 0F00 at that location as follows:

```
7D04 0F00    DATA >0F00
7D06 XXXX
```

Performing this operation in decimal is more difficult. For example, to enter the decimal number 15 at location 7D06, you must first multiply 15 times 256. This effectively shifts the number by two hexadecimal digits. The display would look like this:

```
7D06 0F00    DATA 3840
7D08 XXXX
```



If you had just typed DATA 15 instead of DATA 3840 then 0F would have been entered into memory location 7D07:

```
7D06 000F    DATA 15
7D08 XXXX
```

**EQU—Equate.** This command is used to equate an address or data value to a one- or two-character symbolic name. This is especially useful if an address or data value is used more than once in your program. It is simply easier to type CD in place of >A55A, for example, if the value A55A is used several times in your program.

In general, it is a good practice to use symbolic names in your program rather than explicit numbers. The symbols are often more meaningful later on. Note that this applies more to the Editor/Assembler than to the Line-by-Line Assembler, because as we have said the Line-by-Line Assembler does not save the source code. However, you yourself may save your source program for future reference. All of the exercises in this book were written by hand on paper first, entered into the assembler second, debugged using EASY BUG third, and were finally typed on paper for a future occasion—which turned out to be the writing of this book.

Equates are done at the beginning of a program and do not affect data in the memory location currently displayed. Assume the following assembler display:

```
7D20 0420
```

If you were to type CD EQU >A55A and press ENTER, the assembler would display:

```
7D20 0420 CD EQU >A55A
7D20 0420
```

The assembler would store CD and the value A55A in the Symbol Table and display the same memory address and data that was being displayed when we entered the equate.

**SYM—Display Symbol Table.** This command causes the assembler to display all the references that have been entered into the Symbol Table. References include those created by the EQU command and those created when a label has been used in the program. A one- or two-character name in the label field automatically equates that name with the address of the instruction being entered.

The following program uses five references:

```
7D00 XXXX WP EQU >70B8
7D00 XXXX M1 EQU >7D00
7D00 XXXX M2 EQU >7D02
7D00 XXXX      AORG >7D04
7D04 3001 M3   DATA >3001
7D06 02E0      LWPI WP
7D08 70B8
7D0A 04C1      CLR R1
7D0C C0A0      MOV@M3,R2
7D0E 7D04
7D10R11FF      JLT J2
7D12R13FF      JEQ J2
7D14 0581 J1    INC R1
7D16 0A12      SLA R2, 1
7D18 15FD      JGT J1
7D1A C802 J2    MOV R2,@ M1
7D1C 7D00
7D12*1303
7D10*1104
7D1E C801      MOV R1,@ M2
7D20 7D02
7D22 045B      B *R11
```

This program is shown as it was entered. WP, M1, and M2 were made references by use of the EQU command. M3 and J1 were made references by being placed in the label field.

J2 is different. It was first used at memory location 7D10. JLT J2 is a conditional jump instruction—jump to J2 if the jump condition is met. However, at this point in the program J2 has not yet appeared in a label field nor was J2 defined by an EQU command, and the assembler does not yet know how many memory locations forward are specified by this instruction. That is why the R appears between the address and data columns on that line. At this point J2 is considered an unresolved reference. As soon as J2 is used in a label field the reference will be resolved and the assembler will finish assembling the opcodes for the previous memory locations for which it was not resolved. This is done at memory location 7D1A. The assembler resolves the J2 references and displays the previous memory locations with an asterisk between the address and data.

Using the SYM command after entering the above program will cause the assembler to display three categories of references:

- ☐ Resolved references.
- ☐ Unresolved references (word).
- ☐ Unresolved references (jump).

Unresolved *word references* are unresolved references in any instruction except a jump instruction. For example, at memory location 7D0C of the above program, if I hadn't previously equated M3 with a specific memory address (by use of EQU, or by putting M3 in a label field), then M3 would be an unresolved word reference.

**TEXT—Enter Text.** This command is used to enter a string of characters into memory without having to first convert the characters to ASCII (a one-byte code for each character) and then enter the ASCII values using the DATA command. When you use the TEXT command, the assembler automatically converts the characters to ASCII and stores them in successive memory locations. The word MICROPROCESSOR would be entered as follows:

```

7D00 4D49      TEXT 'MICROPROCESSOR'
7D02 4352
7D04 4F50
7D06 524F
7D08 4345
7D0A 5353
7D0C 4F52
7D0E XXXX

```

The ASCII code for M is 4D, the code for I is 49, and so forth. The text to be entered is enclosed in single quotes. Blanks within the quotes are recognized and encoded. If an odd number of characters is entered, then 00 will be added to the last ASCII byte to make the number of bytes even.

**END, End Program.** This command is used to exit the assembler. After you type the END command in the opcode field and press ENTER, the assembler displays the number of unresolved references. If the number is zero, press ENTER. The assembler will display PRESS ENTER TO CONTINUE. Press ENTER again to return to the Mini Memory selection list, and press QUIT to return to the START screen.

If there are any unresolved references, press the space bar. This returns you to the assembler which displays the last address and data. Type SYM in the opcode field. The assembler will display

the unresolved references. Then use the AORG command to move to any location that needs to be corrected.

```
7D00R10FF    JMP J1
7D02 XXXX    END
              0001 UNRESOLVED REFERENCES
7D02 XXXX    SYM
UNRESOLVED REFERENCES (JUMP)
J1-7D00
7D02 XXXX    AORG >7CFE
7CFE C081 J1 MOV R1,R2
7D00*10FE
7D00 10FE    END
              0000 UNRESOLVED REFERENCES
```

## **EASY BUG**

Return to the master selection list and press 2 for EASY BUG. The screen displays and summarizes the EASY BUG commands and special function keys as shown in Fig. 8-1. By now these commands should be self-explanatory. However, when you write and execute the first assembly language program, the EASY BUG commands and special keys will be explained as you use them. For now, note that the four Xs after a command letter indicate that a hexadecimal memory address is to be supplied by the user. If you enter more than four digits, only the last four are used by EASY BUG. If you enter less than four digits, then whatever number you enter is considered the last digits of a four-digit address.

EASY BUG is a program which allows you to display and modify memory locations, execute assembly language programs, load assembly language programs from cassette and store assembly language programs on cassette (Earlier in this chapter I used EASY BUG to load the Line-by-Line Assembler.)

I am going to use EASY BUG to execute the programs in the remainder of this book. I will also use EASY BUG to display certain memory locations before and after I execute the program. And I will use EASY BUG to modify certain memory locations so that I can try my program on more than one data value without having to return to the assembler.

## **SUMMARY**

The Mini Memory module adds 14K bytes of memory to your

system; gives you 7 additional BASIC subprograms; has a built-in battery which maintains power to the Mini Memory RAM when the module is removed from the console; contains the EASY BUG program; and comes with the Line-by-Line Assembler on a cassette tape.

The Line-by-Line Assembler allows you to build your own assembly language programs using instruction mnemonics and labels. The assembler allows forward references even though code is assembled on a line-by-line basis.

The EASY BUG program allows you to debug and execute your assembly language program, read and modify the CPU and VDP RAM locations, read GROM locations, read and modify CRU bits, and load and store programs on an external cassette tape recorder.

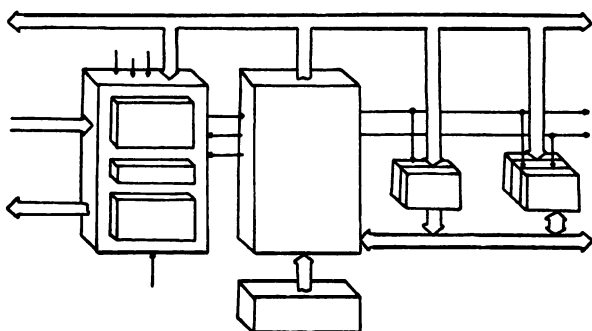
# **Part III**

## **Programs**

---



# Chapter 9



## Beginning Programs

In this chapter, you will create and execute some very simple assembly language programs using the Line-by-Line Assembler and EASY BUG. Each program will illustrate one or two new instructions or addressing modes. Each program will be discussed step by step. After you have entered the program into memory using the assembler, you will use EASY BUG and perform the following tasks:

- ☐ Execute the program.
- ☐ Examine the memory locations that should have been altered by the program or contain the results of the program.
- ☐ Modify memory locations that contain input data.
- ☐ Rerun the program on new input data.
- ☐ Examine the target memory locations again to verify that the program works successfully on more than one data value or set of data values.

Each listing follows the discussion of the program and has ample room to the right of each line to add in your own comments. You will find it very helpful to document all of these programs with your own comments.

### 16-BIT DATA TRANSFER

Let's enter Program 9-1. The procedure for entering (starting) the Line-by-Line Assembler is repeated here for your convenience.



This procedure assumes that the Line-by-Line Assembler has already been loaded into the Mini Memory RAM by using the EASY BUG program, and the START screen is displayed.

1. Press any key. The master selection list is displayed.
2. Press 3 for Mini Memory. The selection list is displayed.
3. Press 2 for RUN. **PROGRAM NAME** is displayed.
4. Type **NEW** and press **ENTER**.

The screen displays the assembler title, copyright, and two 4-digit hexadecimal numbers. The first number is memory address 7D00 and the second number is the contents of that address, as follows:

7D00 XXXX

Your display does not show XXXX. However, I am using four X's to indicate that the contents could be anything at this point, and since we are going to enter a new program we don't care.

Now type the first line but don't press ENTER. The line looks like this:

7D00 XXXX M1 DATA >2E56

As soon as you press ENTER, the assembler changes the data in memory location 7D00 from whatever it was to 2E56 and displays the next memory location as follows:

7D00 2E56 M1 DATA >2E56  
7D02 XXXX

Note that the next memory location is two higher than the previous one. This is because the assembler always displays two bytes at a time. The byte at 7D00 is now 2E and the byte at 7D01 is now 56. Later we will see that EASY BUG displays only one byte at a time.

I have done two things on this line:

1. I equated the name M1 with the address 7D00. The assembler also entered this information into the Symbol Table.
2. I used the DATA command to enter data into a specific memory location. Remember that the DATA command is not a TMS9900 assembly language instruction; it is an assembler directive. I could have achieved the same result another way.

```
7D00 XXXX M1 EQU >7D00
7D00 2E56 DATA >2E56
7D02 XXXX
```

Let's continue:

```
7D02 XXXX M2 BSS 2
7D04 XXXX
```

This step reserves two bytes of memory and equates address 7D02 with the name M2. Let's continue:

```
7D04 02E0 LWPI >70B8
7D06 70B8
7D08 XXXX
```

This is the first instruction in the program. This instruction loads the number 70B8 into the workspace pointer register. It tells the computer which 16 16-bit memory locations are to be used as workspace registers. Memory location 70B8 becomes register 0, 70BA becomes register 1, and so forth. Although TI has reserved a User's Workspace starting at address 70B8 when the Line-by-Line Assembler is loaded, it is necessary to load 70B8 into the workspace pointer register at the start of my program. Otherwise, I would be using the workspace starting at address 83E0—the workspace used by the EASY BUG program when it executes the program.

Note that this instruction uses two words of memory, one for the LWPI opcode (02E0) and one for the operand (70B8). Let's continue:

```
7D08 C820 MOV @M1,@M2
7D0A 7D00
7D0C 7D02
7D0E XXXX
```

This instruction moves the contents of memory location M1 (7D00) to memory location M2 (7D02). The symbolic addressing mode is specified by both operands. Consequently, one additional word of memory is required for each operand. The number C820 is the opcode for the MOV instruction when both operands specify the symbolic addressing mode.

This instruction illustrates the use of symbolic names as a shorthand method of referring to specific addresses or data values. The same result would have been achieved if I had entered `MOV @>7D00,@>7D02`.

Let's continue:

```
7D0E 045B B *R11
7D10 XXXX
```

This is the last instruction in the program. It says; branch to the address contained in workspace register 11.

When using the Line-by-Line assembler to create an assembly language program, workspace register 11 contains the number 609C which is the starting address of a routine which returns you to EASY BUG after the program executes. The number 609C is put into register 11 of the User's Workspace (70B8-70D6) by the assembler when the assembler is first started.

The program is complete. This program transfers a 16-bit number from one memory location to another. This is a very common operation in assembly language programs. This first program also illustrates the following basic steps in developing an assembly language program for the TI-99/4A when using the Mini Memory module and Line-by-Line Assembler:

☐ Assign symbolic names to addresses and data values as required.

☐ Enter initial data values into memory locations as required.

☐ Reserve memory locations for results as required.

☐ Set up workspace.

☐ Enter program instructions.

☐ Enter return instruction.

The next step is to exit the assembler and go to EASY BUG. To do this, use the following procedure:

1. Type `END` in the opcode field and press `ENTER`. The assembler displays `0000 UNRESOLVED REFERENCES` if you have correctly entered the program.

2. Press `ENTER`. (Any other key returns you to the assembler.) The assembler displays `PRESS ENTER TO CONTINUE`.

3. Press `ENTER`. The Mini Memory selection list is displayed.

4. Press `QUIT`. The `START` screen is displayed.

5. Press any key. The master selection list is displayed.

6. Press 2 for EASY BUG. The EASY BUG commands and special function keys are displayed.
7. Press any key. A question mark appears in the lower left hand corner of your screen.

You may now enter any one of the seven EASY BUG commands. To execute the assembly language program just created, use the E (Execute) command. Type E7D04 (7D04 is the starting address of our program) and press ENTER. The EASY BUG display looks like this:

```
?E7D04
?
```

The question mark is displayed to indicate that our program has finished and EASY BUG is ready for the next command. Since the purpose of the program was to move the number 2E56 from 7D00 to 7D02, let's look at CPU Memory location 7D02. Type M7D02 and press enter. EASY BUG displays the following:

```
?M7D02
M7D02 =2E ->
```

The M command is used to modify CPU Memory. Consequently, EASY BUG displays -> to prompt you to type new data in the space to the right. At the moment, you don't want to do that. Instead, you want EASY BUG to display the next byte to see if 56 was moved to 7D03. To do this, just press the space bar. EASY BUG displays the next byte as follows:

```
?M7D02
M7D02 =2E ->
M7D03 =56 ->
```

Recall that an even address may be used to refer to either a 16-bit word or an 8-bit byte. An odd address can refer only to an 8-bit byte. Thus, the word at address 7D02, for example, is composed of the byte at address 7D02 and the byte at address 7D03.

The program worked. A 16-bit data value was transferred from memory location 7D00 to memory location 7D02.

Let's try another number. Press the period to return to the EASY BUG command mode. Type M7D00 and press ENTER. EASY BUG displays the byte at 7D00:

```
?M7D00
M7D00 =2E ->
```

Now type FF to the right of the arrow and press ENTER. EASY BUG displays the byte at 7D01. Type FF again and press ENTER. The EASY BUG display should look like this:

```
?M7D00
M7D00 =2E -> FF
M7D01 =56 -> FF
M7D02 =2E ->
```

Now press the period, execute the program again, and verify that FFFF moved from memory location 7D00 to memory location 7D02.

### Program 9-1

```
7D00 2E56 M1 DATA >2E56
7D02 XXXX M2 BSS 2
7D04 02E0    LWPI >70B8
7D06 70B8
7D08 C820    MOV @M1,@M2
7D0A 7D00
7D0C 7D02
7D0E 045B    B *R11
7D10 XXXX    END
```

### 64-BIT DATA TRANSFER

Enter Program 9-2. The purpose of this program is to move four words (64 bits) of memory from one area of memory (7D10-7D17) to another area of memory (7D18-7D1F). This is accomplished as follows:

1. The AORG command is used to display address 7D10. This preserves the previous program.
2. 64 bits of data are entered 16 bits at a time using the DATA command. Also, address 7D10 is equated to M1 in the same step.
3. A 64-bit block of memory is reserved for storage using the BSS command. BSS 8 means reserve 8 bytes. Also, address 7D18 is equated to M2 in this step.

4. The workspace is set up.
5. Workspace registers 0 and 1 are loaded with the values 7D10 and 7D18, respectively. 7D10 is the starting address of the block of data to be moved and 7D18 is the starting address of the block of memory locations to which the data will be moved.
6. The data is transferred one word at a time. Notice the MOV instruction at address 7D2C. This instruction moves data. The source address is contained in workspace register 0 and the destination address is contained in workspace register 1. Then it increments both registers by two after the data has been transferred. Both operands specify the indirect addressing mode with autoincrement (asterisks and plus signs). After this instruction is executed, the data at 7D10 will be at 7D18 and the contents of registers 0 and 1 will be 7D12 and 7D1A, respectively. The instruction is repeated three times in order to transfer 64 bits. Note that the instruction at address 7D32 is different. Since all data will have been transferred after this instruction, it is not necessary to increment the registers any more.
7. The return instruction is entered.

Now go to EASY BUG and perform the following steps.

1. Execute the program. The starting address is 7D20.
2. Verify that the program worked by displaying locations 7D18-7D1F.
3. Change the input data at locations 7D10-7D17 as follows:

```

?M7D10
M7D10 =3E -> 01
M7D11 =2A -> 23
M7D12 =42 -> 45
M7D13 =A1 -> 67
M7D14 =21 -> 89
M7D15 =F2 -> AB
M7D16 =60 -> CD
M7D17 =A0 -> EF

```

4. Execute the program again.
5. Verify that the data was transferred correctly.

## Program 9-2

```

7D00 XXXX    AORG >7D10
7D10 3E2A M1  DATA >3E2A
7D12 42A1    DATA >42A1

```

```

7D14 21F2    DATA >21F2
7D16 60A0    DATA >60A0
7D18 XXXX M2 BSS 8
7D20 02E0    LWPI >70B8
7D22 70B8
7D24 0200    LI R0,M1
7D26 7D10
7D28 0201    LI R1,M2
7D2A 7D18
7D2C CC70    MOV *R0+,*R1+
7D2E CC70    MOV *R0+,*R1+
7D30 CC70    MOV *R0+,*R1+
7D32 C450    MOV *R0,*R1
7D34 045B    B *R11
7D36 XXXX    END

```

## 16-BIT ADDITION

Enter Program 9-3. The purpose of this program is to add two 16-bit numbers and store the result in memory. The first number is located at address 7D36, and the second number is located at address 7D38. The result will be stored at address 7D3A. This is accomplished in three instructions, not counting the instruction to set up the workspace and the return instruction.

1. The instruction at 7D40 moves the first number from M1 (7D36) to workspace register 0.
2. The next instruction adds the contents of memory location M2 (7D38) to register 0 and places the result in register 0.
3. The third instruction transfers the result to M3 (7D3A). Note that you can do a 16-bit addition in one instruction:

```

7D40 A820    A @M1,@M2
7D42 7D36
7D44 7D38
7D46 XXXX

```

This instruction adds the contents of M1 to the contents of M2 and stores the result at M2. You can see that when this instruction is

executed that the original contents of M2 will be lost. In some cases this is alright. The approach used in Program 9-3 preserves M2.

Now go to EASY BUG and verify that the result at 7D3A is 3716 (hexadecimal). Change M1 and M2 to 17F5 and 2182, respectively. Rerun the program and verify that the result at 7D3A has changed from 3716 to 3977 (hexadecimal).

### **Program 9-3**

```
7D00 XXXX AORG >7D36
7D36 10F5 M1 DATA >10F5
7D38 2621 M2 DATA >2621
7D3A XXXX M3 BSS 2
7D3C 02E0     LWPI >70B8
7D3E 70B8
7D40 C020     MOV @M1,R0
7D42 7D36
7D44 A020     A @M2,R0
7D46 7D38
7D48 C800     MOV R0,@M3
7D4A 7D3A
7D4C 045B     B *R11
7D4E XXXX     END
```

### **32-BIT ADDITION**

Enter Program 9-4. The purpose of this program is to add two 32-bit numbers and store the result in memory.

The first number is 12A2E641 and is stored in two adjacent 16-bit memory locations. The most significant word (12A2) is stored at location 7D4E and the least significant word (E641) is stored at location 7D50. The second number is 001019BF. The most significant word is stored at 7D52 and the least significant word is stored at 7D54. The most significant word of the result will be stored at 7D56 and the least significant word will be stored at 7D58.

Since the 9900 microprocessor has only a 16-bit ALU (arithmetic logic unit), it can only add 16 bits at a time. Therefore, to perform a 32-bit addition, two 16-bit additions must be performed. In the first 16-bit addition, the least significant words are added. If a



carry results, then 1 must be added to the most significant word of either of the addends. Next, the most significant words are added.

This is accomplished in Program 9-4 as follows:

1. Address 7D4E, the address of the most significant word of the first number, is loaded into register 0. Then indirect addressing with autoincrement is used to load the addends into registers 1 through 4. Two registers per addend are required. 12A2 is moved to register 1, E641 to register 2, 0010 to register 3, and 19BF to register 4. Note that after four operations, register 0 contains the number 7D56, the address where I will store the most significant word of the result.

2. The least significant words of the addends are added first. This is done at 7D6A. The instruction adds the contents of register 2 to the contents of register 4 and stores the result in register 4.

3. The next instruction (address 7D6C) jumps to J1 (7D70) if no carry was generated; otherwise go on to the next instruction. If a carry is generated you must add 1 to either register 1 or register 3. These registers contain the next 16 bits to be added. Therefore, the instruction at 7D6E increments register 1 (or adds 1 to the contents of register 1). This instruction is performed only if a carry was generated (carry bit of the status register set to 1), otherwise there is a jump over it to J1.

4. Next add the most significant words of the addends. This is done at 7D70.

5. Finally, the result is stored in memory. Conveniently, register 0 contains 7D56. Thus, the most significant word of the result is stored first and increment the contents of register 0 by two again. This is done, as before, by using the indirect addressing with autoincrement. Since you are performing a word operation (as opposed to a byte operation), then the contents of the specified register are incremented by two. For byte operations, the contents would be incremented by one.

Note that as you entered this program, the assembler displayed the following results starting at address 7D6C:

```
7D6C R17FF    JNC J1
7D6E 0581     INC R1
7D70 A0C1 J1 A R1,R3
7D6C*1701
7D72 XXXX
```

The first time 7D6C is displayed (after the instruction is

entered), J1 is an unresolved reference. Consequently, the assembler places an R between the address and data. Also, the FF in 17FF is a dummy displacement value. (The 17 is the code for JNC.) After the instruction at 7D70 is entered, however, J1 is resolved. Consequently, the assembler displays the address and data of all previous lines (only one in this program) in which J1 was unresolved. The new display shows an asterisk between the address and data, and the data has been corrected.

Recall that the displacement value in a jump instruction is a relative number. A displacement value of 1 does not mean that 1 is added to the current program counter value, but means that the program counter should be incremented by two 1 time, since the program counter is always an even address and can only be incremented or decremented by two or be replaced by an even number. Also, the displacement value times two is added to the program counter after the instruction is fetched. In Program 9-4, after the instruction at 7D6C has been fetched, the program counter equals 7D6E. Thus, the displacement value to jump to 7D70 is 1, since the program counter would have to be incremented by two 1 time to be equal to 7D70. (Now aren't you glad that you can use labels and just let the computer figure out the displacement for you?)

Now go to EASY BUG and verify that the result stored at 7D56-7D59 is equal to 12B30000. Using standard addition format, the problem and solution is as follows:

$$\begin{array}{r}
 1 \\
 12A2 \ E641 \\
 + \ 0010 \ 19BF \\
 \hline
 12B3 \ 0000
 \end{array}$$

Since the ALU adds 16 bits, or 4 hexadecimal digits, the numbers are shown with a space between the fourth and fifth digits.

Notice that this choice of addends produces a carry. To completely verify the program, choose a set of values that will not generate a carry. Therefore, change 001019BF to 001019BE. Rerun the program and verify that the result is 12B2FFFF.

#### Program 9-4

7D00 XXXX    AORG >7D4E

7D4E 12A2 M1 DATA >12A2

7D50 E641    DATA >E641

7D52	0010	DATA >0010
7D54	19BF	DATA >19BF
7D56	XXXX	BSS 4
7D5A	02E0	LWPI >70B8
7D5C	70B8	
7D5E	0200	LI R0,M1
7D60	7D4E	
7D62	C070	MOV *R0+,R1
7D64	C0B0	MOV *R0+,R2
7D66	C0F0	MOV *R0+,R3
7D68	C130	MOV *R0+,R4
7D6A	A102	A R2,R4
7D6C	1701	JNC J1
7D6E	0581	INC R1
7D70	A0C1	J1 A R1,R3
7D72	CC03	MOV R3,*R0+
7D74	C404	MOV R4,*R0
7D76	045B	B *R11
7D78	XXXX	END

## FIND THE LARGER OF TWO UNSIGNED NUMBERS

Enter Program 9-5. The purpose of this program is to find the larger of two unsigned numbers and to store the larger unsigned number in memory. This is accomplished as follows:

1. The instructions at 7D82 and 7D86 move the unsigned numbers to registers 0 and 1.
2. The instruction at 7D8A compares the contents of register 0 to the contents of register 1. After this instruction is executed, three bits of the status register are affected—LGT (logic greater than), AGT (arithmetic greater than), and EQ (equal).
3. In the context of the program, the instruction at 7D8C essentially says jump to J1 if the unsigned number in register 0 is greater than the unsigned number in register 1. This instruction tests the LGT and EQ status bits. If LGT equals 1 and EQ equals 0, then the jump condition is met.
4. If the number in register 0 is larger, then jump to J1 (7D90). The

instruction at that location then moves the number in register 0 to address M3 (7D7C).

5. If the number in register 0 is not larger, then perform the next instruction in sequence (7D8E). This instruction moves the number in register 1 to register 0. Then move the number in register 0 to M3, as in step 4.

The 9900 instruction set includes six jump instructions to be used when comparing unsigned numbers. These are listed in Table 9-1.

If I wanted to find the larger of two *signed* numbers, I would use the JGT instruction instead of the JH instruction. See Table 9-2 for the jump instructions to be used when comparing signed numbers. Note that for signed numbers, there is no jump instruction for the greater than or equal condition or for the less than or equal condition. To test for these conditions, the JEQ instruction would have to

**Table 9-1. Jump Instructions for Unsigned Numbers.**

Compare Condition	Jump Instruction
Greater Than	JH
Greater Than or Equal	JHE
Equal	JEQ
Not Equal	JNE
Less Than or Equal	JLE
Less Than	JL

be used before or after the JGT to test for greater than or equal and before or after the JLT to test for less than or equal.

Now go to EASY BUG and verify that you have entered the program correctly. The number 9125 should be in memory location 7D7C.

Change 9125 at 7D78 to 1000 (hexadecimal) and rerun the program. The number 102C should be in memory location 7D7C if the program was entered correctly.

**Table 9-2. Jump Instructions for Signed Numbers.**

Compare Condition	Jump Instruction
Greater Than	JGT
Equal	JEQ
Not Equal	JNE
Less Than	JLT

## Program 9-5

```
7D00 XXXX    AORG >7D78
7D78 9125 M1  DATA >9125
7D7A 102C M2  DATA >102C
7D7C XXXX M3  BSS  2
7D7E 02E0     LWPI >70B8
7D80 70B8
7D82 C020     MOV @M1,R0
7D84 7D78
7D86 C060     MOV @M2,R1
7D88 7D7A
7D8A 8040     C R0,R1
7D8C 1B01     JH J1
7D8E C001     MOV R1,R0
7D90 C800 J1  MOV R0,@M3
7D92 7D7C
7D94 045B     B *R11
7D96 XXXX     END
```

## SUM OF SQUARES

Enter Program 9-6. The purpose of this program is to perform the following arithmetic operation:

$$\begin{aligned} 7^2 + 50^2 &= 2549 && \text{(decimal)} \\ \text{or } 0007^2 + 0032^2 &= 000009F5 && \text{(hexadecimal)} \end{aligned}$$

This is the sum of squares operation. The numbers to be squared and added are stored in locations 7D96 and 7D98, respectively. The most significant 16 bits of the 32-bit result will be stored at location 7D9A, and the least significant 16 bits will be stored at location 7D9C. This is accomplished as follows:

1. The address 7D96 is loaded into register 0.
2. Using indirect addressing, the first number is moved to register 1.
3. Using indirect addressing with autoincrement, the first number is multiplied times the number in register 1. Since the number in

register 1 is the same, then the multiply operation has squared the number.

Note that the second operand of the MPY instruction must be a workspace register. In the instruction at 7DA8, the second operand is register 1. This instruction multiplies the number whose address is stored in register 0 times the number in register 1. Then it places the most significant 16 bits of the 32-bit register in register 1, and places the least significant 16 bits in register 2.

After this instruction the address stored in register 0 equals 7D98, the address of the next number that I want to square.

4. The instructions at 7DAA and 7DAC perform the same operation (that was done in step 3 above) on the second number. The 32-bit result is stored in registers 3 and 4. The address stored in register 0 now equals 7D9A, the address where I will store the 16 most significant bits of the sum of the squares.

5. Instructions at 7DAE-7DB8 add the 32-bit squares and move the result to memory locations 7D9A and 7D9C. These instructions are identical to those used in Program 9-4 to perform the same task.

Go to EASY BUG and verify that the answer in 000009F5.

### Program 9-6

```
7D00 XXXX    AORG >7D96
7D96 0007 M1  DATA 7
7D98 0032    DATA 50
7D9A XXXX    BSS 4
7D9E 02E0    LWPI >70B8
7DA0 70B8
7DA2 0200    LI R0,M1
7DA4 7D96
7DA6 C050    MOV *R0,R1
7DA8 3870    MPY *R0+,R1
7DAA C0D0    MOV *R0,R3
7DAC 38F0    MPY *R0+,R3
7DAE A102    A R2,R4
7DB0 1701    JNC J1
7DB2 0581    INC R1
7DB4 A0C1 J1  A R1,R3
```

```

7DB6 CC03    MOV R3,*R0+
7DB8 C404    MOV R4,*R0
7DBA 045B    B *R11
7DBC XXXX    END

```

### TABLE OF FACTORIALS

Enter Program 9-7. The purpose of this program is to determine the factorial of a number between 0 and 8. The factorial of a number is defined as follows:

$$N \text{ FACTORIAL} = N(N-1) (N-2) (N-3) . . . (N-M), \text{ where } M=N-1$$

The symbol for N FACTORIAL is N!. For example, 7! equals 7 times 6 times 5 times 4 times 3 times 2 times 1. 0! is defined as being equal to 1. N! may also be defined as N times (N-1)!. Table 9-3 shows the factorials of numbers 0 through 8. The table could easily be expanded, but has been limited to numbers whose factorials are less than 65,535 so that no more than 16 bits are needed to express the factorial in binary. (9! equals 362,880 and requires 17 bits.)

The approach taken in Program 9-7 is to have the computer determine the factorial of a number by looking it up in a table instead of computing the factorial according to the equation given above. The table (limited to 9 values in this program) is stored in the memory between locations 7DBC and 7DCC.

The number for which I want to find the factorial is located in memory location 7DCE, and the factorial, when found, will be placed in memory location 7DD0. This is accomplished in three steps:

Table 9-3. Table of Factorials.

N	N Factorial
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5,040
8	40,320

Left Shift Count	Multiplier
1	$2=2^1$
2	$4=2^2$
3	$8=2^3$
4	$16=2^4$
5	$32=2^5$
6	$64=2^6$
7	$128=2^7$
8	$256=2^8$
9	$512=2^9$
10	$1,024=2^{10}$
11	$2,048=2^{11}$
12	$4,096=2^{12}$
13	$8,192=2^{13}$
14	$16,384=2^{14}$
15	$32,768=2^{15}$

**Table 9-4. Left Shift  
Equivalent Multiplier Values.**

1. The number for which I want to find the factorial is loaded into register 1. See the instruction at 7DD6.
2. The number is then multiplied by two. The position of the factorial in the table is two times the number because each factorial takes two bytes of memory. To find 5! I must add 10 decimal (000A in hexadecimal) to the starting address of the table.

Notice the instruction at 7DDA. This instruction shifts the contents of register 1 to the left 1 bit. This is a special way to multiply by two. The SLA instruction may be used for multiplication in the following cases:

☐ The multiplier must be an integer that can be expressed as an integral power of 2 (see Table 9-4).

☐ The shift count must be less than the number of leading zeros in the binary expression of the number, otherwise the answer will be incorrect. For example, if the number 0000001101000111 is left-shifted 1 bit 8 times, then the 1s that were in bit positions 6 and 7 will be lost. (Remember that TI numbers their bit positions from left to right, from bit 0 to 15.)

3. The last step is to add twice the number to the starting address and move the number at this address to M3 (7DD0). This is done in a single instruction using the indexed addressing mode. The instruction at 7DDC moves data. The source address is the sum of M1 and the contents of register 1. The destination address is M3. 000A is added to 7DBC to get 7DC6. The number at 7DC6 is equal to 5! (0078 hexadecimal) and is moved to address 7DD0.

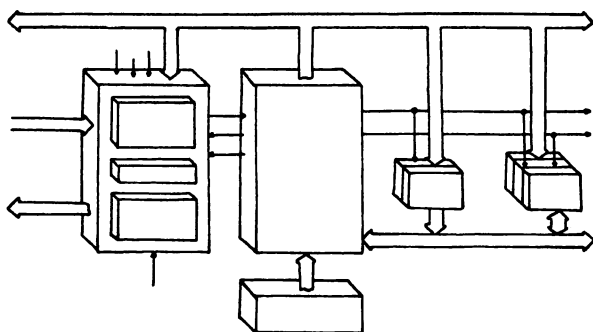


Go to EASY BUG and run the program. Verify that the number at 7DD0 equals 0078. Try one or two other values by changing the number at 7DCF from 05 to any number between 00 and 08.

### Program 9-7

```
7D00 XXXX    AORG >7DBC
7DBC 0001 M1  DATA 1
7DBE 0001    DATA 1
7DC0 0002    DATA 2
7DC2 0006    DATA 6
7DC4 0018    DATA 24
7DC6 0078    DATA 120
7DC8 02D0    DATA 720
7DCA 1380    DATA 5040
7DCC 9D80    DATA 40320
7DCE 0005 M2  DATA 5
7DD0 XXXX M3  BSS 2
7DD2 02E0    LWP1>70B8
7DD4 70B8
7DD6 C060    MOV @M2,R1
7DD8 7DCE
7DDA 0A11    SLA R1,1
7DDC C821    MOV @M1(R1),@M3
7DDE 7DBC
7DE0 7DD0
7DE2 045B    B *R11
7DE4 XXXX    END
```

## Chapter 10



### Simple Program Loops

In this chapter I present six example programs which have loops. The primary purpose of a program loop is to save memory space. Secondly, the use of program loops saves time in writing the program. For example, suppose you wanted to add a list of six numbers. Let's say that the address of the first number in the list is stored in register 1 and that the sum will be stored temporarily in register 0. Based on what you have learned so far, a program to add six numbers would look like this:

```
CLR R0  
A *R1+,R0  
A *R1+,R0  
A *R1+,R0  
A *R1+,R0  
A *R1+,R0  
A *R1+,R0
```

This program clears register 0 and then adds each number in the list to the contents of register 0. For six numbers this program is probably alright. But for a list of sixty numbers, it would be a pain to have to enter sixty addition instructions. Also, you would use sixty memory locations. By using a program loop, this number can be reduced.

A program that uses a loop will set aside a register to be what is

called a *loop counter*. Normally, the number of data items to be processed will be stored in the loop counter register. After each data item is processed, the loop counter is decremented by one and then tested to see if it equals zero. If the loop counter value is not zero, then the program jumps to the beginning of the processing section. If the loop counter value is zero, then the program exits the loop and executes the next instruction following the loop.

A program loop consists of the following steps:

1. Process data item. This step uses one or more instructions. In the above example, the processing section would be only one instruction, namely `A *R1+,R0`.
2. Set up conditions for next pass. In particular, the register which contains the address of the data to be processed is incremented by two. Using indirect register addressing with autoincrement, this step may be combined with one of the instructions in step 1.
3. Decrement the loop counter.
4. Test the loop counter value. Jump to the beginning of the loop if not zero, otherwise go on to the next instruction.

The program loop is preceded by one or more instructions to set up, or initialize, registers or memory locations used by the loop.

The following six programs contain simple program loops. You should enter each program into memory using the Line-by-Line Assembler and execute each program using EASY BUG. Remember to use the starting address of the program when you use the E command. The first instruction of all programs in this book is `LWPI >70B8`. Usually, there will be one or more assembler commands before this instruction. The assembler commands set up the memory locations used by the program and also equate one or more symbolic names to specific addresses used in the program.

## 16-BIT SUM OF DATA

The purpose of Program 10-1 is to add a series of 16-bit numbers. The numbers to be added are stored at locations 7DE4-7DE8. The result will be stored at 7DEA.

Memory location 7DEC contains the starting address of the first number to be added. This makes the program flexible. If I wanted to add a series of numbers somewhere else in memory, then I only need to change the address at 7DEC to be equal to the address of the first data item.

Memory location 7DEE contains the number of numbers to be added, three in this case. This also makes the program flexible. The program that follows operates on the number of data items at 7DEE.

This number may be changed. However, in this example, the number may not be greater than three because I have only allocated enough memory for three numbers. If, however, I had twelve numbers stored in memory locations 7F00-7F16, I could change the contents of 7DEC from 7DE4 to 7F00. Then the program that follows (7DF0-7E08) could add those twelve numbers.

Workspace registers are used as follows:

☐ Register 0 is used as temporary storage for the subtotals and for the total when all items have been added. Before any items are added, this register is cleared. See instruction at 7DF4.

☐ Register 1 stores the address of the next data item to be added. Initially, the address stored at 7DEC is moved to this register.

☐ Register 2 is used as the loop counter. Initially, the number at 7DEE is moved to this register.

The program loop consists of three instructions:

```
J1 A *R1+,R0
DEC R2
JNE J1
```

The first instruction fetches the contents of the address stored in register 1 and adds this number to the number in register 0. After the data is fetched, the address value in register 1 is incremented by two.

The second instruction decrements the number in register 2. After the number is decremented, it is compared to zero and the LGT, AGT, EQ, C, and OV status bits are either set to one or cleared to zero, depending on the new value in register 2.

The third instruction tests the EQ status bit. If EQ equals zero, then the jump condition is met and the program jumps to J1. The loop will be repeated until the loop counter equals zero, thereby causing the EQ bit to be set to one. Then the program exits the loop and goes on to the next instruction, which in this case moves the last subtotal in register 0 to memory location M1 (7DEA). The final result is 3EA4.

Note that a shorter program could be written if you always wanted to add only three numbers. However, to add ten or twenty numbers it is shorter (and usually better) to use a program loop.

## Program 10-1

```
7D00 XXXX    AORG >7DE4
7DE4 2040     DATA >2040
7DE6 1C22     DATA >1C22
7DE8 0242     DATA >0242
7DEA XXXX M1  BSS 2
7DEC 7DE4 M2  DATA >7DE4
7DEE 0003 M3  DATA 3
7DF0 02E0     LWPI >70B8
7DF2 70B8
7DF4 04C0     CLR R0
7DF6 C060     MOV @M2,R1
7DF8 7DEC
7DFA C0A0     MOV @M3,R2
7DFC 7DEE
7DFE A031 J1  A *R1+,R0
7E00 0602     DEC R2
7E02 16FD     JNE J1
7E04 C800     MOV R0,@M1
7E06 7DEA
7E08 045B     B *R11
7E0A XXXX     END
```

## 32-BIT SUM OF DATA

Program 10-1 has two problems: First, if the number at location 7DEE were equal to zero before the program was started, then the program would not work properly. The number in register 2 would be decremented from 0000 to FFFF after the first time through the loop. The program would have to add 65,536 numbers before the loop counter would be equal to zero again.

Second, only one 16-bit storage location has been allocated for the result. This is alright if you know ahead of time that the sum will never exceed sixteen bits. But if you are adding numbers which all could be near the maximum (65,535 decimal for a 16-bit micro-processor), then you should provide at least one extra bit per

number to be added. For example, if you are adding eight 16-bit numbers each which may be near the maximum, then you should reserve three bytes of memory for the result.

Program 10-2 solves both of these problems. The first problem is solved by comparing the loop counter value to zero before the program loop is entered. If the value is zero then the program jumps over the loop and moves a zero subtotal to the memory locations reserved for the total. The second problem is solved by simply allocating two 16-bit memory locations for the total. Looking at Program 10-2, you can easily see that two words of memory have been reserved at addresses 7E10 and 7E12. It is not, however, immediately obvious how the first problem is solved—that is, where exactly before the loop is the loop counter compared with zero. Let's look at the program one section at a time.

The instructions at 7E1C-7E2A set up the workspace registers as follows:

- ☐ Register 0 stores the destination address for the 32-bit total.

- ☐ Register 1 is the temporary storage for the sixteen most significant bits of the subtotal. This register is initially cleared and then in the loop is incremented by one each time a carry results from the addition of two 16-bit numbers.

- ☐ Register 2 is temporary storage for the sixteen least significant bits of the subtotal. This register is initially cleared. Each number in the list will be added to the contents of this register; and the result will be stored in this register.

- ☐ Register 3 contains the address of the number to be added to the contents of register 2. Initially, the address stored at memory location 7E14 is moved to this register. Each time through the loop the address in register 3 is incremented by two. This register is used as what is called a *data pointer* because the contents of the register always point to the next data item to be processed.

- ☐ Register 4 is used as the loop counter. Initially, the number stored at memory location 7E16 is moved to this register.

Note the instruction at 7E2C. This instruction tests the EQ status bit. The EQ status bit was either set or cleared by the previous MOV instruction at 7E2E. That instruction moved the loop counter value (the number of data items) to register 4. What is not obvious is that the MOV instruction always compares the source operand (the value to be moved) to zero. Consequently, a condi-

tional jump instruction can follow a MOV instruction—a savings of one instruction. (C R4, R1 would work, for example, since register 1 equals zero at this point.)

The program loop is stored at locations 7E2E-7E36. Two instructions (at 7E30 and 7E32) have been added to the program loop of Program 10-1. These two instructions essentially add one to the sixteen most significant bits of the subtotal each time a carry occurs when adding the sixteen least significant bits.

The remaining instructions move the final result to memory locations 7E10-7E12. The final result is 00011EA4.

### **Program 10-2**

```
7D00 XXXX    AORG >7E0A
7E0A 2040    DATA > 2040
7E0C 1C22    DATA > 1C22
7E0E E242    DATA >E242
7E10 XXXX M1 BSS 4
7E14 7E0A M2 DATA > 7E0A
7E16 0003 M3 DATA 3
7E18 02E0    LWPI > 70B8
7E1A 70B8
7E1C 0200    LI R0,M1
7E1E 7E10
7E20 04C1    CLR R1
7E22 04C2    CLR R2
7E24 C0E0    MOV @M2,R3
7E26 7E14
7E28 C120    MOV @M3, R4
7E2A 7E16
7E2C 1305    JEQ J3
7E2E A0B3 J1 A *R3+,R2
7E30 1701    JNC J2
7E32 0581    INC R1
7E34 0604 J2 DEC R4
7E36 16FB    JNE J1
```

```

7E38 CC01 J3 MOV R1,*R0+
7E34 C402     MOV R2,*R0
7E3C 045B     B *R11
7E3E XXXX     END

```

## NUMBER OF NEGATIVE NUMBERS

The purpose of Program 10-3 is to read a list of signed numbers and determine how many of them are negative. The list of numbers is stored in memory beginning at address 7E3E. The number of negative numbers will be stored at 7E44. Memory location 7E46 contains the starting address of the list and memory location 7E48 contains the number of numbers in the list. The actual program starts at address 7E4A.

The workspace registers are allocated as follows:

- ☐ Register 0 is used as temporary storage for the number of negative numbers in the list. This register is initially cleared to zero. It is incremented each time a negative number is found while the list is being read.

- ☐ Register 1 is used as the data pointer. Initially, the address stored at 7E46 is moved to this register. The register is incremented by two each time a number is read from the list.

- ☐ Register 2 is used as a loop counter. Initially, the number stored at 7E48 is moved to this register. The register is decremented each time a number is read from the list.

- ☐ Register 3 is used as a temporary storage location for each number as it is read from the list.

How is the list read? The list is read by moving a number in the list (designated by the address in register 1) to register 3. This is the first instruction in the program loop stored in memory locations 7E5A-7E64. Recall from our discussion of Program 10-2 that when a MOV instruction is executed, the data that is moved is compared to zero. If the number is greater than or equal to zero, then the program jumps to J2. Otherwise, the number is negative and register 0 is incremented by one.

When all the numbers have been read (register 2 equals zero), the number in register 0 (which contains the number of negative numbers) is moved to 7E44. In this example, two of the three numbers are negative. See Table 10-1 for a partial list of 4-digit hexadecimal signed numbers and decimal equivalents.



**Table 10-1. Partial List of Signed Number Equivalents.**

Hexadecimal	Decimal
7FFF	+32,767
7000	+28,672
6000	+24,576
5000	+20,480
4000	+16,384
3000	+12,288
2000	+8,192
1000	+4,096
0001	+96
0000	1
FFFF	-1
F000	-4,096
E000	-8,192
D000	-12,288
C000	-16,384
B000	-20,480
A000	-24,576
9000	-28,672
8000	-32,768

### Program 10-3

```

7D00 XXXX    AORG >7E3E
7E3E F1DC    DATA >F1DC
7E40 7E0A    DATA >7E0A
7E42 824B    DATA >824B
7E44 XXXX M1 BSS 2
7E46 7E3E M2 DATA >7E3E
7E48 0003 M3 DATA 3
7E4A 02E0    LWPI >70B8
7E4C 70B8
7E4E 04C0    CLR R0
7E50 C060    MOV @M2,R1
7E52 7E46
7E54 C0A0    MOV @M3,R2
7E56 7E48
7E58 1306    JEQ J3
7E5A C0F1 J1 MOV *R1+,R3

```

```

7E5C 1502    JGT J2
7E5E 1301    JEQ J2
7E60 0580    INC R0
7E62 0602 J2 DEC R2
7E64 16FA    JNE J1
7E66 C800 J3 MOV R0,@M1
7E68 7E44
7E6A 045B    B *R11
7E6C XXXX    END

```

## NUMBER OF ZERO, POSITIVE, AND NEGATIVE NUMBERS

The purpose of Program 10-4 is to read a list of signed numbers and to determine how many of them are negative, how many are equal to zero and how many are positive. The numbers and their addresses are as follows:

Number	Address
7602	7E6C
8D48	7E6E
2120	7E70
0000	7E72
E605	7E74
0004	7E76

The final storage locations for the results are as follows:

Result	Address
Number of negative numbers	7E78
Number of zero's	7E7A
Number of positive numbers	7E7C

The starting address of the list is stored in memory location 7E7E, and the number of data items to be processed is stored at 7E80. The program begins with the instruction at 7E82.

The registers are allocated as follows:

Register	Function
0	Contains final storage address for results.

Register	Function
1	N counter—number of negative numbers.
2	Z counter—number of zero's.
3	P counter—number of positive numbers.
4	Data pointer.
5	Loop counter.
6	Temporary storage - each number is moved here.

Program 10-4 may be divided into three sections: initialization (7E86-7E98), main processing loop (7E9A-7EAC), and save results and return (7EAE-7EB4).

The initialization section consists of eight instructions: 7E82 initializes the workspace. 7E86 loads the starting address of the block of memory that will be used as final storage for the results. 7E8A clears the N counter, 7E8C clears the Z counter, and 7E8E clears the P counter. 7E90 moves the address stored at 7E7E to the data pointer register. 7E94 moves the number stored at 7E80 to the loop counter. 7E98 tests the initial loop counter value. If zero, then the program jumps to the instructions which save the N, Z, and P counter values.

The main program loop consists of ten instructions: 7E9A reads the number and increments the data pointer. 7E9C tests the number to see if it equals zero. If so, then the program jumps to 7EA4 where the Z counter is incremented. 7E9E tests the number to see if it is positive. If so, then the program jumps to 7EA8 where the P counter is incremented. 7EAD increments the N counter. There is no need to test the number to see if it is negative. If it is not zero and not positive, then it must be negative. 7EA2 causes the program to jump to 7EAA, jumping over the instructions that increment the Z and P counters. 7EA4 increments the Z counter. 7EA6 causes the program to jump to 7EAA, jumping over the instruction to increment the P counter. 7EA8 decrements the loop counter. 7EAC tests the loop counter value. If not zero, then the program jumps to the beginning of the loop. You should be able to see that only one counter (N, Z, or P) is incremented each time through the loop.

The final section consists of four instructions, three of which store the final N, Z, and P counter values in memory. Using the numbers at 7E6C-7E76, the final values are 2, 0, and 3 respectively. The last instruction (7EB4) returns control to EASY BUG.

Although there are no new instructions or addressing modes in

this program, the program is a good example of how the use of a program loop can save instructions and programming time. Without using a loop, the set of eight instructions at address 7E9A-7EA8 would have to be used for each number in the list. Thus, the 10 loop instructions in Program 10-4 do the job of 48 nonloop instructions. Quite a savings in memory and programming time.

#### **Program 10-4**

```
7D00 XXXX    AORG >7E6C
7E6C 7602    DATA >7602
7E6E 8D48    DATA >8D48
7E70 2120    DATA >2120
7E72 0000    DATA 0
7E74 E605    DATA >E605
7E76 0004    DATA 4
7E78 XXXX M1 BSS 6
7E7E 7E6C M2 DATA >7E6C
7E80 0006 M3 DATA 6
7E82 02E0    LWPI >70B8
7E84 70B8
7E86 0200    LI R0,M1
7E88 7E78
7E8A 04C1    CLR R1
7E8C 04C2    CLR R2
7E8E 04C3    CLR R3
7E90 C120    MOV @M2,R4
7E92 7E7E
7E94 C160    MOV @M3,R5
7E96 7E80
7E98 130A    JEQ J5
7E9A C1B4 J1 MOV *R4+,R6
7E9C 1303    JEQ J2
7E9E 1504    JGT J3
7EA0 0581    INC R1
7EA2 1003    JMP J4
7EA4 0582 J2 INC R2
7EA6 1001    JMP J4
7EA8 0583 J3 INC R3
7EAA 0605 J4 DEC R5
7EAC 16F6    JNE J1
7EAE CC01 J5 MOV R1,*R0+
```

```

7EB0 CC02    MOV R2,*R0+
7EB2 C403    MOV R3,*R0
7EB4 045B    B *R11
7EB6 XXXX    END

```

## FIND MAXIMUM VALUE

The purpose of Program 10-5 is to read a list of unsigned numbers and determine which of them is the largest. Let me explain how this is accomplished.

The first number in the list is compared to zero, the initial value of register 0. If the number is larger than zero, then it is moved to register 0 and becomes the number to which the next number in the list is compared. The loop counter is decremented and the next number is read. If the first number is equal to zero, then zero is retained in register 0, the loop counter is decremented and the next number is read.

Each time through the loop, the new number is compared with the previous maximum that was found. When all items have been read, the number in register 0 is moved to memory location 7EBE. The maximum value in the list is E57A.

### Program 10-5

```

7D00 XXXX    AORG >7EB6
7EB6 A48E    DATA >A48E
7EB8 71AC    DATA >71AC
7EBA 34F1    DATA >34F1
7EBC E57A    DATA >E57A
7EBE XXXX M1 BSS 2
7EC0 7EB6 M2 DATA >7EB6
7EC2 0004 M3 DATA 4
7EC4 02E0    LWPI >70B8
7EC6 70B8
7EC8 04C0    CLR R0
7ECA C060    MOV @M2,R1
7ECC 7EC0
7ECE COA0    MOV @M3,R2
7ED0 7EC2
7ED2 1306    JEQ J3

```

```

7ED4 COF1 J1 MOV *R1+,R3
7ED6 8003   C R3,R0
7ED8 1A01   JL J2
7EDA C003   MOV R3,R0
7EDC 0602 J2 DEC R2
7EDE 16FA   JNE J1
7EE0 C800 J3 MOV R0,@M1
7EE2 7EBE
7EE4 045B   B *R11
7EE6 XXXX   END

```

## FIND MINIMUM BYTE VALUE

Program 10-6 illustrates how the 9900 microprocessor handles 8-bit, or byte, data values. The purpose of this particular program is to read a list of unsigned 8-bit numbers and determine which of them is the smallest.

The 8-bit data values and their addresses are given below:

Value	Address
65	7EE6
79	7EE7
15	7EE8
E3	7EE9
72	7EEA

Note that the values are hexadecimal numbers. Also, although 00 is stored at 7EEB, this number is not in the list. Since data values must be entered two bytes at a time using the DATA command, this byte was made equal to zero when the value 72 was entered.

Register 0 is used to temporarily store the minimum values as they are found. Initially, this register is set to FFFF (all 1's, the highest possible 16-bit unsigned number) using the SET0 (set to ones) instruction. Note, however, that only the upper byte, or eight most significant bits, of the register are used in the comparison operation.

Data values are read using the MOVB (move byte) instruction. The first time, for example, the byte at 7EE6 is moved to the upper byte half of register 3. Using indirect addressing with auto-increment, the data pointer is incremented by one. Recall that in the

auto-increment mode, the processor increments the specified register contents by one for byte operations and by two for word operations.

The CB (compare bytes) instruction at 7F04 compares the upper byte in register 3 with the upper byte in register 0. The MOV B instruction at 7F08 saves the lower value byte in register 0 (again, the upper byte half).

The MOV B instruction at 7F0E moves the final result (E3, in this list) to location 7EEC. Note that the byte at 7EED is unchanged by this program. In fact, I could save one word of memory by storing the result at 7EEB, the lower byte of word location 7EEA. To do this, just change the instruction of 7F0E to MOV B R0, @>7EEB, or go to EASY BUG and change the byte at 7F11 from EC to EB.

### Program 10-6

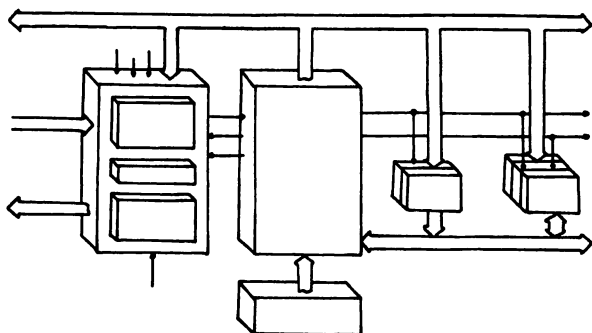
```
7D00 XXXX    AORG >7EE6
7EE6 6579     DATA >6579
7EE8 15E3     DATA >15E3
7EEA 7200     DATA >7200
7EEC XXXX M1 BSS 2
7EEE 7EE6 M2 DATA >7EE6
7EF0 0005 M3 DATA 5
7EF2 02E0     LWPI >70B8
7EF4 70B8
7EF6 0700     SET0 R0
7EF8 C060     MOV @M2,R1
7EFA 7EEE
7EFC C0A0     MOV @M3,R2
7EFE 7EF0
7F00 1306     JEQ J3
7F02 D0F1 J1 MOV B *R1+, R3
7F04 9003     CB R3,R0
7F06 1B01     JH J2
7F08 D003     MOV B R3,R0
7F0A 0602 J2 DEC R2
```

7FOC 16FA JNE J1  
7FOE D800 J3 MOVB R0,@M1  
7F10 7EEC  
7F12 045B B \*R11  
7F14 XXXX END





# Chapter 11



## Character-Coded Data

In this chapter, I will discuss seven short programs that process strings. A string is a series of one or more characters that have been translated into ASCII code. All data input via the keyboard, for example, is ASCII encoded before it is stored in memory and processed. Also, all information to be displayed on the screen must be converted to ASCII beforehand.

ASCII is in 7-bit code for letters, numbers, symbols (such as the comma, period, and so forth), and control characters (such as CONTROL P, a combination of the CONTROL key and the P key) all of which may or may not be on your keyboard. Since computers basically operate on 8- or 16-bit data, ASCII codes are stored and transferred as an 8-bit number with the leading bit always equal to zero.

Table 11-1 is a partial list of characters and their corresponding ASCII codes. For codes that are not included in the table, you should see the *TI-99/4A User's Reference Guide* supplied with your computer. Note that the TI-99/4A has more than one keyboard mode and some characters change code when the mode changes. The characters that change codes are ones that are not included in Table 11-1.

### LENGTH OF A STRING OF CHARACTERS

The purpose of Program 11-1 is to determine the length of a string of ASCII-encoded characters. The string "TI-99/4A" is

**Table 11-1. ASCII Character Codes.**

Character	ASCII	Character	ASCII	Character	ASCII
Carriage Return	0D	@	40		
Space	20	A	41	a	61
!	21	B	42	b	62
"	22	C	43	c	63
#	23	D	44	d	64
%	25	E	45	e	65
&	26	F	46	f	66
'	27	G	47	g	67
(	28	H	48	h	68
)	29	I	49	i	69
*	2A	J	4A	j	6A
+	2B	K	4B	k	6B
-	2D	L	4C	l	6C
.	2E	M	4D	m	6D
/	2F	N	4E	n	6E
0	30	O	4F	o	6F
1	31	P	50	p	70
2	32	Q	51	q	71
3	33	R	52	r	72
4	34	S	53	s	73
5	35	T	54	t	74
6	36	U	55	u	75
7	37	V	56	v	76
8	38	W	57	w	77
9	39	X	58	x	78
:	3A	Y	59	y	79
;	3B	Z	5A	z	7A
<	3C	[	5B	{	7B
=	3D	\	5C		7C
>	3E	]	5D	}	7D
?	3F	^	5E	~	7E
		_	5F		

stored at 7D00-7D07. 0D is the ASCII code for the ENTER key (or carriage return) and is stored at 7D08 immediately following the string to be processed. The 0D signals the program that the end of the string has been reached.

7D0A contains the number 7D00, the starting address of the string. 7D0C is reserved for the result. The length of the string, when determined, will be placed at this location.

7D0E is the starting address of the actual program. (Remember this when you go to EASY BUG to run the program. In this book, all programs start with the instruction LWPI >70B8 which initializes the workspace pointer.)

The instruction at 7D12 clears register 0 which is used as a string character counter. 7D14 loads the ASCII code for the ENTER key. Each character in the string is compared to 0D to see if the end of the string has been reached.

7D18 moves the address at 7D0A to register 2 which is used as the data pointer.

7D1C-7D22 contains the program loop. The first time through

the loop, 7D1C compares the first character in the string to 0D and increments the data pointer by one. 7D1E causes the program to exit the loop and go to 7D24 if the character equals 0D. Otherwise, 7D20 increments the character counter and 7D22 causes the program to go through the loop again.

7D24 moves the final count to 7D0C and 7D28 returns control to EASY BUG. You should go to EASY BUG and put 0D at an earlier point in the string and see if the count changes accordingly. If you want to try a different string, it will probably be easier if you go to the assembler and use the TEXT command. Be sure to make 0D your last character. Also, if you want to enter a string longer than ten characters (including the terminator 0D), you should put the string after 7D28 and then change the data in 7D0A to be equal to the starting address of your string.

### Program 11-1

```
7D00 5449    TEXT 'TI-99/4A'
7D02 2D39
7D04 392F
7D06 3441
7D08 0D00    DATA > 0D00
7D0A 7D00 M1 DATA > 7D00
7D0C XXXX M2 BSS 2
7D0E 02E0    LWPI > 70B8
7D10 70B8
7D12 04C0    CLR R0
7D14 0201    LI RI, > 0D00
7D16 0D00
7D18 COAO    MOV @M1, R2
7D1A 7D0A
7D1C 9072 J1 CB *R2+, R1
7D1E 1302    JEQ J2
7D20 0580    INC R0
7D22 10FC    JMP J1
7D24 C800 J2 MOV R0, @M2
7D26 7D0C
```

7D28 045B    B \*R 11

7D2A XXXX    END

## FIND FIRST NONBLANK CHARACTER

The purpose of Program 11-2 is to find the first nonblank character in an ASCII encoded string. The string "TI" is stored at 7D2A-7D2F. The starting address, 7D2A is stored at 7D30 and the location of the first nonblank character when found will be stored at 7D32.

The program starts at 7D34. The instruction at 7D38 loads the ASCII code for a blank in the upper byte half of register 0. 7D3C loads the starting address of the string into register 1.

7D40-7D42 contains the loop which tests succeeding characters until a nonblank character is found. The first time through the loop, the instruction at 7D40 compares the byte at 7D2A with the byte 20 and increments the data pointer by one—from 7D2A to 7D2B. 7D42 causes the program to go back to 7D40 if the character is a blank. When a nonblank character is found, the program exits the loop and goes on to 7D44.

7D44 decrements the data pointer. This is necessary because 7D40 automatically increments the data pointer. Thus, when the program gets to instruction 7D44 the data pointer is pointing at the second nonblank character.

7D46 transfers the address of the first nonblank character to address 7D32.

### Program 11-2

```
7D00 XXXX    AORG >7D2A
7D2A 2020    TEXT ' TI '
7D2C 5449
7D2E 2020
7D30 7D2A M1 DATA >7D2A
7D32 XXXX M2 BSS 2
7D34 02E0    LWPI > 70B8
7D36 70B8
7D38 0200    LI R0,>2000
7D3A 2000
7D3C C060    MOV @M1,R1
```

```

7D3E 7D30
7D40 9031 J1 CB *R1+,R0
7D42 13FE JEQ J1
7D44 0601 DEC R1
7D46 C801 MOV R1,@M2
7D48 7D32
7D4A 045B B *R11
7D4C XXXX END

```

## FIND LAST NONBLANK CHARACTER

The purpose of Program 11-3 is to read a string and determine which character is the last nonblank character. The string "TI-99/4A" is stored at 7D4C-7D57. For this string the program will determine that the last nonblank character is located at address 7D55.

The starting address of the string is stored at 7D58 and two bytes of memory have been reserved at 7D5A for the address of the last nonblank character when found. The instruction at 7D60 loads the ASCII code for a blank into the upper byte of register 0. 7D64 moves the starting address of the string to register 1.

The program contains two loops. The purpose of the first loop (7D68-7D6A) is to skip over any leading blanks. The second loop (7D6C-7D6E) reads characters until a blank is found.

7D70 decrements the data pointer (register 1) by two when the first blank after a nonblank character has been found. This is necessary because the autoincrement mode is used in the instruction of 7D6C and causes the data pointer to be pointing at the next character after the first blank. In this case, the data pointer value equals 7D57 before the instruction at 7D70 is executed. The nonblank character, A, is two locations back to 7D55. Thus, the data pointer must be decremented by two.

7D72 moves the address at the last nonblank character to memory location 7D5A.

### Program 11-3

```

7D00 XXXX AORG > 7D4C
7D4C 2020 TEXT ' TI-99/4A '
7D4E 5449

```

```

7D50 2D39
7D52 392F
7D54 3441
7D56 2020
7D58 7D4C M1 DATA > 7D4C
7D5A XXXX M2 BSS 2
7D5C 02E0     LWPI > 70B8
7D5E 70B8
7D60 0200     LI R0,>2000
7D62 2000
7D64 C060     MOV @M1,R1
7D66 7D58
7D68 9031 J1 CB *R1+,R0
7D6A 13FE     JEQ J1
7D6C 9031 J2 CB *R1+,R0
7D6E 16FE     JNE J2
7D70 0641     DECT R1
7D72 C801     MOV R1,@M2
7D74 7D5A
7D76 045B     B *R11
7D78 XXXX     END

```

## REPLACE LEADING ZEROS WITH BLANKS

The purpose of Program 11-4 is to read an ASCII encoded number and replace all leading zeros with blanks.

The number to be processed, 00005, is stored at 7D78-7D7C. The number is followed by a blank stored at 7D7D. The blank signals the program that the end of the string (the number) has been reached. The program has also been designed to look for the carriage return code (ENTER on the TI-99/4A), OD, to see if the string has ended.

Another common approach to determining when the string has ended is to put the length of the string first. Then the program uses the number as the initial value of a loop counter. For example, the input data for processing the four character string 0005 would look like this:

7D78 0004 DATA 4  
7D7A 3030 TEXT '0005'  
7D7C 3035

Of course, the program to process strings in this manner would be different than Program 11-4.

Also, part of the problem in writing this program is to make sure it can handle a string of all zeros. In this case, I want to replace all leading zeros (before a blank or carriage return) except the last one, which I want to preserve.

Now let's look at Program 11-4. Instructions at 7D84-7D8E load the ASCII codes for zero (30), blank (20) and carriage return (0D) into registers 0, 1, and 2, respectively. 7D90 moves the starting address of the string to register 3.

7D94-7D9A tests the first character to see if it is a blank or carriage return. If it is either one of these, then the program jumps to 7DB4, which ends the program.

7D9C-7DA2 is a loop which tests each character to see if it equals a zero. If so, the zero is replaced by a blank, the data pointer is incremented, and the program jumps to the start of the loop. If a nonzero character is discovered, the program exits the loop.

The remainder of the program determines whether or not the string was all zeros. If not, then the program ends, since no new characters need to be examined. If the string is all zeros, then the program must put the last one back.

This is accomplished in 7DA4-7DB0. 7DA4-7DAA compares the first nonzero character with the blank and the carriage return. If the first character (after replacing leading zeros) is either a blank or a carriage return, then the string must have been all zeros and the program jumps to 7DAE. 7DAE decrements the data pointer so that the last zero can be put back by the instruction at 7DB0.

If the code at 7DA4-7DAA determines that the first nonzero character is neither a blank or carriage return then the instruction at 7DAC is executed. 7DAC causes the program to jump to the end, bypassing the code that puts a zero back in the string.

#### **Program 11-4**

7D00 XXXX AORG >7D78  
7D78 3030 TEXT '0005 '  
7D7A 3030  
7D7C 3520



```

7D7E 7D78 M1 DATA >7D78
7D80 02E0      LWPI >70B8
7D82 70B8
7D84 0200      LI R0,>3000
7D86 3000
7D88 0201      LI R1,>2000
7D8A 2000
7D8C 0202      LI R2,>0D00
7D8E 0D00
7D90 C0E0      MOV @M1,R3
7D92 7D7E
7D94 9053      CB *R3,R1
7D96 130D      JEQ J4
7D98 9093      CB *R3,R2
7D9A 130B      JEQ J4
7D9C 9013 J1 CB *R3,R0
7D9E 1602      JNE J2
7DA0 DCC1      MOVB R1,*R3+
7DA2 10FC      JMP J1
7DA4 9053 J2 CB *R3,R1
7DA6 1303      JEQ J3
7DAB 9093      CB *R3,R2
7DAA 1301      JEQ J3
7DAC 1002      JMP J4
7DAE 0603 J3 DEC R3
7DB0 D4C0      MOVB R0,*R3
7DB2 045B J4 B *R11
7DB4 XXXX      END

```

## TRUNCATE DECIMAL STRING TO INTEGER FORM

The purpose of Program 11-5 is to read an ASCII encoded multidigit number and replace the decimal point and following digits with blanks. If no decimal point is found, then the number will be unchanged.

The string processed in this example program is the ASCII code for the number 3.1416. The string is stored at 7DB4-7DB8. 7DBA contains the ASCII code for carriage return. The starting address of the string is stored at 7DBC. The starting address of the actual program is 7DBE.

The instructions at 7DC2-7DD0 initialize the registers. The ASCII code for the decimal point (the period), the blank, and the carriage return are loaded into registers 0, 1, and 2, respectively. The starting address of the string is moved to register 3.

The instructions at 7DD2-7DDC test each character in the string until a blank, a carriage return, or a decimal point is found. If the character is either a blank or carriage return, then the program jumps to the end and the string is left unchanged. If the character is a decimal point, then the program proceeds to the next processing section.

After a decimal point is found, the instruction at 7DDE decrements the data pointer. Again, this is necessary because the autoincrement addressing mode was used in the instruction at 7DDA.

7DE0-7DE8 replaces the decimal point and subsequent characters with blanks until either a blank or carriage return is found, in which case the program ends.

### **Program 11-5**

```

7D00 XXXX    AORG >7DB4
7DB4 332E    TEXT '3.1416'
7DB6 3134
7DB8 3136
7DBA 0D00    DATA >0D00
7DBC 7DB4 M1 DATA >7DB4
7DBE 02E0    LWPI >70B8
7DC0 70B8
7DC2 0200    LI R0,>2E00
7DC4 2E00
7DC6 0201    LI R1,>2000
7DC8 2000
7DCA 0202    LI R2,>0D00
7DCC 0D00
7DCE C0E0    MOV @M1,R3

```

```

7DD0 7DBC
7DD2 9053 J1 CB *R3,R1
7DD4 130A JEQ J3
7DD6 9093 CB *R3,R2
7DD8 1308 JEQ J3
7DDA 9033 CB *R3+,R0
7DDC 16FA JNE J1
7DDE 0603 DEC R3
7DE0 DCC1 J2 MOVB R1,*R3+
7DE2 9053 CB *R3,R1
7DE4 1302 JEQ J3
7DE6 9093 CB *R3,R2
7DE8 16FB JNE J2
7DEA 045B J3 B *R11
7DEC XXXX END

```

## PATTERN MATCH

The purpose of Program 11-6 is to compare two strings and determine if they match.

The first string, "GLASS," is stored at 7DEC-7DF1. Note that GLASS has an odd number of characters. Thus, when the TEXT command is used to enter the string, the assembler adds 00 to the ASCII code to make the number of characters even. The program interprets the 00 as the string terminator. If the string had been an even number of characters, then it would have been necessary to add 00 or some other code for a terminator. A blank, carriage return, or FF are common terminators. The second string, "GRASS," is stored at 7DF2-7DF7.

The starting address of the first string is stored at 7DF8 and the starting address of the second string is stored at 7DFA. 7DFC will be used to store the code for either a match (0000) or a mismatch (FFFF).

The instructions at 7E02-7E08 initialize the registers. Register 0 is cleared. This register is used as temporary storage for the match/mismatch code. The register is initialized with the match code. If the strings are found to be different then the register contents will be set to all ones and moved to 7DFC.

The starting address of the first string is moved to register 1, and the starting address of the second string is moved to register 2. Both registers are used as data pointers and are incremented as the strings are read and compared character by character.

The instructions at 7E0C-7E14 form a program loop which compares each character in string 1 to the corresponding character in string 2. When a mismatch occurs, the program jumps to 7D16, where the match code in register 3 is changed to the mismatch code and then moved to 7DFC.

When a match occurs, the characters are tested to see if they are equal to zero, meaning that the string has ended. If so, the program jumps to 7E18 which moves the match code to 7DFC.

Note that the data pointers are incremented at different points in the loop 7E0C-7E14. The data pointer for string 2 is incremented at 7E0C, while the data pointer for string 1 is incremented at 7E10. Also note that after a match, it is necessary to compare only one character of one of the strings to zero. Finally, note that register 0 does double duty. Normally, it is used to store the match/mismatch code. Since, however, it always equals zero at the point when I want to see if a set of matched characters equal zero, then I may use it as the reference for comparison.

Now you should be able to see why the data pointers must be incremented at different times. If I incremented register 1 at 7E06, then I would have to decrement it to perform the comparison at 7E10, otherwise I would not be testing the same character to see if it equals zero. Then I would have to increment register 1 again before the end of the loop. A more straight forward method (which uses two more instructions) would look like this:

<b>J1 CB *R1,*R2</b>	<b>Compare characters.</b>
<b>JNE J2</b>	<b>If mismatch, go set code to mismatch.</b>
<b>CB *R1,R0</b>	<b>Characters match. Has string ended?</b>
<b>JEQ J3</b>	<b>If yes, to move match code to memory.</b>
<b>INC R1</b>	<b>Increment string 1 data pointer.</b>
<b>INC R2</b>	<b>Increment string 2 data pointer.</b>
<b>JMP J1</b>	<b>Start loop over.</b>

### **Program 11-6**

```
7D00 XXXX    AORG >7DEC
7DEC 474C    TEXT 'GLASS'
7DEE 4153
```

```

7DF0 5300
7DF2 4752    TEXT 'GRASS'
7DF4 4153
7DF6 5300
7DF8 7DEC M1 DATA>7DEC
7DFA 7DF2 M2 DATA>7DF2
7DFC XXXX M3 BSS 2
7DFE 02E0    LWPI>70B8
7E00 70B8
7E02 04C0    CLR R0
7E04 C060    MOV @M1,R1
7E06 7DF8
7E08 C0A0    MOV @M2,R2
7E0A 7DFA
7E0C 9C91 J1 CB *R1,*R2+
7E0E 1603    JNE J2
7E10 9031    CB *R1+,R0
7E12 1302    JEQ J3
7E14 10FB    JMP J1
7E16 0700 J2 SET0 R0
7E18 C800 J3 MOV R0,@M3
7E1A 7DFC
7E1C 045B    B *R11
7E1E XXXX    END

```

## STRING COMPARISON

The purpose of Program 11-7 is to compare two strings and determine which is greater. This kind of program is useful when you want to alphabetize string data. It is almost identical to Program 11-6. The only difference is in the program loop. Program 11-6 compares two corresponding characters of the strings and exits the loop if they are not equal.

```

J1 CB *R1,*R2+
JNE J2

```

Program 11-7 compares two corresponding characters and exits the loop if string 1 is greater than string 2.

J1 CB \*R1,\*R2+  
JH J2

The instruction at J2 sets the contents of register 0 to all ones, meaning that string 1 is greater than 2. If the JH condition is not met, then the program tests to see if string 1 is less than string 2.

JL J3

The instruction at J3 moves the contents of register 0 (all zeros) to 7E30, meaning that string 2 is less than or equal to string 2.

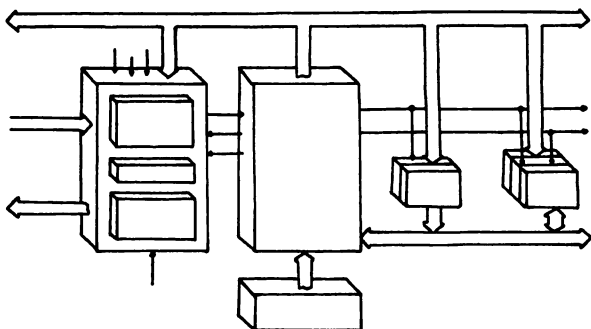
If the JL condition is not met, then the strings must be equal at this point. The program then tests to see if the end of the string has been reached. If not, the next pair of characters are compared.

### Program 11-7

```
7D00 XXXX    AORG >7E20
7E20 5445    TEXT 'TEXT '
7E22 5854
7E24 2000
7E26 5445    TEXT 'TENT '
7E28 4E54
7E2A 2000
7E2C 7E20 M1 DATA >7E20
7E2E 7E26 M2 DATA >7E26
7E30 XXXX M3 BSS 2
7E32 02E0    LWPI >70B8
7E34 70B8
7E36 0460    CLR R0
7E38 C060    MOV @M1,R1
7E3A 7E2C
7E3C C0A0    MOV @M2,R2
7E3E 7E2E
7E40 9C91 J1 CB *R1,*R2+
7E42 1B04    JH J2
```

7E44 1A04 JL J3  
7E46 9031 CB \*R1+,R0  
7E48 1302 JEQ J3  
7E4A 10FA JMP J1  
7E4C 0700 J2 SET0 R0  
7E4E C800 J3 MOV R0,@M3  
7E50 7E30  
7E52 045B B \*R11  
7E54 XXXX END

# Chapter 12



## Code Conversion

So far I have discussed 20 example programs which have collectively used the following 22 instructions:

A, Add Immediate

B, Branch

C, Compare Words

CB, Compare Bytes

CLR, Clear

DEC, Decrement

DECT, Decrement by Two

INC, Increment

JEQ, Jump if Equal

JGT, Jump if Greater Than

JH, Jump if Higher

JL, Jump if Lower

JMP, Unconditional Jump

JNC, Jump on No Carry

JNE, Jump if Not Equal

LI, Load Immediate

LWPI, Load Workspace  
Pointer Immediate

MOV, Move Word

MOVB, Move Byte

MPY, Multiply

SETO, Set to Ones

SLA, Shift Left Arithmetic

In this chapter, I will discuss 12 more example programs which will collectively use the following 10 new instructions:

AI, Add Immediate

ANDI, AND Immediate

CI, Compare Immediate

DIV, Divide

JLE, Jump if Less Than or Equal

SB, Subtract Bytes

SOC, Set Ones Corresponding

SRC, Shift Right Circular

SRL, Shift Right Logical

SWPB, Swap Bytes



The twelve programs in this chapter are all code conversion programs. The programs may be grouped in pairs. For example, Program 12-1 converts a single hexadecimal digit to ASCII and Program 12-2 converts an ASCII code to a single hexadecimal digit. I will begin with simple single digit conversion programs and work up to multidigit and ASCII string conversion programs.

The last two programs are especially important. One of my goals is to show you how to write a program that reads data input via the keyboard, processes that data, and then displays results on the screen. Program 12-12 converts ASCII-encoded input data (generated when you press keys on the keyboard) to binary numbers. Program 12-11 converts binary numbers to ASCII, a task that must be performed before results can be displayed on the screen. Later you will learn how to use some subroutines (already stored in the Mini Memory module ROM) which read the keyboard and display data on the screen.

## HEXADECIMAL TO ASCII

The purpose of Program 12-1 is to convert a single hexadecimal digit (0-9, A-F) to ASCII. Numbers 0 through 9 are converted simply by adding 30 to the number. ASCII for the number 0 is 30, ASCII for the number 1 is 31, and so forth.

ASCII for the hexadecimal number A, however, is 41, not 3A. If you go back and look at Table 11-1, you will see that codes 3A through 40 are used for symbols and that the capital letters begin with the ASCII code 41. Thus to convert the numbers A through F to ASCII you must add the hexadecimal number 37.

Program 12-1 does not convert any specific hexadecimal digit to ASCII. However, a one byte memory location (7E54) is reserved for the digit to be converted, and a one byte memory location (7E55) is reserved for the ASCII result. Before you run the program from EASY BUG, enter a hexadecimal digit in memory location 7E54. Since hexadecimal digits are only four bits wide, you must precede your candidate digit by zero. For example, if you want to convert the number C to ASCII, enter 0C into memory location 7E54. Run the program (type E7E56 and press ENTER), and read the result in memory location 7E55.

Let me explain how Program 12-1 converts hexadecimal digits to ASCII. The instruction at 7E5A moves the candidate digit to the upper byte half of register 0. 7E5E compares the digit to 0A to see if the digit is in the range of 0 through 9. If so, 7E62 causes the program to jump to 7E68 where 30 is added to the number.

If the number is not in the range 0 through 9, then 7E64 adds 07 to the number and 7E68 adds 30 to the number. This two-step process eliminates the need for a jump instruction. See the following alternate program.

```
LWPI >70B8
MOVB@M1,R0
CI R0,>0A00
JL J1
AI R0,>3700
JMP J2
J1 AI R0,>3000
J2 MOVB R0,@M2
B *R11
```

The instruction at 7E6C moves the result to 7E55. Note that this program assumes that input digits are in the valid range—0 through 9, and A through F. No provision has been made to detect out-of-range digits and store an error code in 7E55.

This program uses two new instructions—CI (compare immediate) and AI (add immediate). The instruction at 7E5E compares the contents of register 0 with the number 0A00. Note that there is no compare immediate instruction for byte values. Since I have moved the candidate digit to the upper byte half of register 0, I must compare it with 0A00, not 000A. Although the lower byte contents of register 0 are unknown, it doesn't matter since comparing with 0A00 is sufficient to determine if the number in the upper byte half of register 0 is in the range 0-9 no matter what is in the lower byte half of register 0. Care must be taken when mixing byte and word operations. Sometimes it is necessary to clear the register before moving a byte value into the upper byte half of a register.

The instructions at 7E64 and 7E68 use the add immediate instruction. 7E64 adds the contents of register 0 to the number 0700 and stores the result in register 0. Note that there is no add immediate instruction for byte values. Also, as in the case of the compare immediate instruction, be careful when mixing byte and word operations.

### **Program 12-1**

```
7D00 XXXX M1 EQU >7E54
7D00 XXXX M2 EQU >7E55
7D00 XXXX AORG >7E56
```

```

7E56 02E0    LWPI > 70B8
7E58 70B8
7E5A D020    MOVB @M1,R0
7E5C 7E54
7E5E 0280    CI R0,>0A00
7E60 0A00
7E62 1A02    JL J1
7E64 0200    AI R0,>0700
7E66 0700
7E68 0200 J1 AI R0,>3000
7E6A 3000
7E6C D800    MOVB R0,@M2
7E6E 7E55
7E70 045B    B *R11
7E72 XXXX    END

```

## ASCII TO HEXADECIMAL

Program 12-2 performs the function of Program 12-1 in reverse. ASCII codes are converted to single hexadecimal digits. The program assumes that the ASCII codes represent the digits 0 through 9 and A through F.

As you might expect, conversion from ASCII to hexadecimal is performed by subtracting 30 from codes 30 through 39 to obtain digits 0-9 and by subtracting 37 from codes 41 through 46 to obtain digits A-F.

The instructions at 7E78 and 7E7C load the values 30 and 07 into the upper byte halves of registers 0 and 1, respectively. This is necessary because there is no subtract immediate instruction.

7E80 moves the ASCII digit from byte location 7E72 to the upper byte half of register 2.

7E84 subtracts 30 from the ASCII digit and 7E86 compares the result with 0A. If less than 0A (and hence in the range of 0-9), then 7E8A causes the program to jump to 7E8E. The instruction at 7E8E then moves the result to byte memory location 7E73.

If the result is not in the range 0-9, then the program assumes that it is in the range A-F. The instruction at 7E8C subtracts 07 from the previous result. The new result is now in the range A-F and is moved to 7E73 by the instruction at 7E8E.

As in Program 12-1, no specific input number is processed. Before you run the program in EASY BUG, enter a valid ASCII code (see Table 11-1) in memory location 7E72.

## Program 12-2

```

7D00 XXXX M1 EQU >7E72
7D00 XXXX M2 EQU >7E73
7D00 XXXX AORG 1 74
7E74 02E0 LWPI >70B8
7E76 70B8
7E78 0200 LI R0,>3000
7E7A 3000
7E7C 0201 LI R1,>0700
7E7E 0700
7E80 D0A0 MOVB @M1,R2
7E82 7E72
7E84 7080 SB R0,R2
7E86 0282 CI R2,>0A00
7E88 0A00
7E8A 1A01 JL J1
7E8C 7081 SB R1,R2
7E8E D802 J1 MOVB R2,@M2
7E90 7E73
7E92 045B B *R11
7E94 XXXX END

```

## ASCII TO DECIMAL

Program 12-3 converts an ASCII code value at byte memory location 7E94 to a decimal number between 0 and 9 and stores the result at byte memory location 7E95. Because the conversion is a simple process (less code than ASCII to hexadecimal conversion which is also simple), I have added one feature: the program checks to make sure the result is a valid decimal number between 0 and 9. If not the error code FF is stored at 7E95.

The instruction at 7E9A sets the contents of register 0 to FFFF. The upper FF will be moved to 7E95 if the conversion result

is not valid. 7E9C loads the value 30 into the upper byte half of register 1. 7EA0 moves the ASCII code value to the upper byte half of register 2. 7EA4 subtracts 30 from the ASCII code and stores the result in the upper byte half of register 2.

7EA6 compares the result with 09FF. The FF is necessary because the lower byte contents of the register are not known. Note that I could have compared the result with 0A00 and used the JL instruction at 7EAA.

The instruction at 7EAA jumps to 7EAE if the contents of register 2 is less than or equal to 09FF. If the result is valid, then this instruction causes the program to jump over the instruction at 7EAC which moves the error code FF to the upper byte half of register 2.

7EAE moves the contents of register 2, either a valid decimal number (00 to 09) or the error code (FF) to memory location 7E95.

### Program 12-3

```
7D00 XXXX M1 EQU >7E94
7D00 XXXX M2 EQU >7E95
7D00 XXXX    AORG >7E96
7E96 02E0    LWPI >70B8
7E98 70B8
7E9A 0700    SET0 R0
7E9C 0201    LI R1,>3000
7E9E 3000
7EA0 D0A0    MOVB @M1,R2
7EA2 7E94
7EA4 7081    SB R1,R2
7EA6 0282    CI R2,>09FF
7EA8 09FF
7EAA 1201    JLE J1
7EAC D080    MOVB R0,R2
7EAE D802 J1 MOVB R2,@M2
7EB0 7E95
7EB2 045B    B *R11
7EB4 XXXX    END
```

## DECIMAL TO ASCII

Program 12-4 is the reverse of Program 12-3. The program converts a decimal value stored at 7EB4 to its ASCII equivalent and stores the result at 7EB5. Additionally, input decimal values are checked to make sure they are valid (0-9) before they are converted. If input values are not valid, then error code 20 (ASCII code for a space) is stored in memory location 7EB5.

### Program 12-4

```
7D00 XXXX M1 EQU > 7EB4
7D00 XXXX M2 EQU > 7EB5
7D00 XXXX    AORG > 7EB6
7EB6 02E0    LWPI > 70B8
7EB8 70B8
7EBA 0200    LI R0,>2000
7EBC 2000
7EBE D060    MOVB @M1,R1
7EC0 7EB4
7EC2 0281    CI R1,> 0A00
7EC4 0A00
7EC6 1A02    JL J1
7EC8 D040    MOVB R0,R1
7ECA 1002    JMP J2
7ECC 0221 J1 AI R1,>3000
7ECE 3000
7ED0 D801 J2 MOVB R1,@M2
7ED2 7EB5
7ED4 045B    B *R11
7ED6 XXXX    END
```

## BINARY-CODED DECIMAL TO BINARY

Program 12-5 converts a 4-digit binary-coded decimal (BCD) to binary. BCD is a convenient form in which to encode decimal numbers. For example, the binary equivalent of 2,971 is 0000101010011010, or 0B9B in hexadecimal. The BCD form is twice as long:

00000010000010010000011100000001

This BCD number is 02090701 in hexadecimal. In other words, each digit in a decimal number is stored as a single byte in memory in its binary equivalent. A 4-digit decimal number requires 4 bytes of memory.

If BCD requires twice the memory, why use it? There are two basic reasons. First, many instruments, such as a digital voltmeter, output digital data in BCD format. Thus, if a digital voltmeter or a decade counter integrated circuit (such as Motorola MC14553) is connected to the computer, then it is necessary to have a program to convert the BCD value to binary.

Second, converting an ASCII-decimal string to binary is a two-step process. The first step converts ASCII to BCD by subtracting 30 from each digit. The second step converts the BCD number to binary. An ASCII-decimal string is the usual form in which numbers are entered into memory via the keyboard.

The conversion of the number 02090701 is performed as follows:

1. Multiply the most significant digit by 10.

Decimal	Hexadecimal
2	0002
$\times 10$	$\times 000A$
<u>20</u>	<u>0014</u>

2. Add the next digit to the result of Step 1.

Decimal	Hexadecimal
20	0014
$+ 9$	$+ 0009$
<u>29</u>	<u>001D</u>

3. Multiply the result of Step 2 times 10.

Decimal	Hexadecimal
29	001D
$\times 10$	$\times 000A$
<u>290</u>	<u>0122</u>

4. Add the next digit to the result of Step 3.

Decimal	Hexadecimal
290	0122
<u>+ 7</u>	<u>+ 0007</u>
297	0129

5. Multiply the result of Step 3 times 10.

Decimal	Hexadecimal
297	0129
<u>× 10</u>	<u>000A</u>
2970	0B9A

6. Add the last digit to the result of Step 5.

Decimal	Hexadecimal
2970	0B9A
<u>+ 1</u>	<u>+ 0001</u>
2971	0B9B

Conversion is complete. The process is simple: multiply the result by 10 and add the next digit. In step 1, the result is just the first digit. Thereafter, however, the result is the sum of the previous product and the next digit. For a 4-digit BCD number, a total of three multiplications and four additions are performed.

Now let's look at Program 12-5 line-by-line. The instructions at 7EE0 through 7EEC initialize the registers. Register 0 is used as a loop counter. The initial value is 4 because the program processes a 4-digit number. Register 1 contains the constant multiplier value, 10. Register 2 is the data pointer. It contains the address of the BCD digit to be processed. The initial digit address is 7ED6. Register 3 is used as a subtotal register. All products are moved to this register. Also, BCD digits are moved to register 4, and then added to register 3. Initially, this register is cleared to zero.

The instruction at 7EEE causes the program to jump over the multiplication portion of the loop. 7EF4 moves the first digit to the upper byte half of register 4. 7EF6 shifts the digit to the lower byte half and replaces the upper byte with zero. Now the digit is in the proper position to be added to previous results by the instruction at 7E48. The first time through the loop the result in register 3 is zero.

7EFA decrements the loop counter and the instruction at 7EFC causes the program to repeat the entire multiply-add routine if the counter value is not zero.



7EF0 multiplies the sum in register 3 by 10. The 32-bit result is stored in registers 3 and 4. For 4-digit BCD numbers, register 3 will always be zero after this multiplication. Since it is known ahead of time that the product is less than sixteen bits, it is safe to move the nonzero result in register 4 to register 3. This is done by the instruction at 7EF2.

The instructions at 7EF4-7EFC bring in the next digit, shift it to the proper position, and add it to the previous result stored in register 3. This process (7EF0-7EFC) continues until the loop counter equals zero. Then the final result is moved to memory location 7EDA.

To aid in understanding the conversion process and the use of registers in this program, the intermediate register results are shown in Table 12-1. Blanks in the table indicate that the result for that register has not changed since the last entry.

**Table 12-1. Intermediate Results of Program 12-5.**

Instruction Address	Register Results			
	R0	R2	R3	R4
7EE0	0004			
7EE8		7ED6		
7EEC			0000	
7EF4		7ED7		0200
7EF6				0002
7EF8			0002	
7EFA	0003			
7EF0			0000	0014
7EF2			0014	
7EF4		7ED8		0900
7EF6				0009
7EF8			001D	
7EFA	0002			
7EF0			0000	0122
7EF2			0122	
7EF4		7ED9		0700
7EF6				0007
7EF8			0129	
7EFA	0001			
7EF0			0000	0B9A
7EF2			0B9A	
7EF4		7EDA		0100
7EF6				0001
7EF8			0B9B	
7EFA	0000			

## Program 12-5

```
7D00 XXXX    AORG >7ED6
7ED6 0209 M1  DATA >0209
7ED8 0701    DATA >0701
7EDA XXXX M2  BSS 2
7EDC 02E0    LWPI >70B8
7EDE 70B8
7EE0 0200    LI R0,4
7EE2 0004
7EE4 0201    LI R1,10
7EE6 000A
7EE8 0202    LI R2,M1
7EEA 7ED6
7EEC 04C3    CLR R3
7EEE 1002    JMP J2
7EF0 38C1 J1  MPY R1,R3
7EF2 C0C4    MOV R4,R3
7EF4 D132 J2  MOVB *R2+,R4
7EF6 0984    SRL R4,8
7EF8 A0C4    A R4,R3
7EFA 0600    DEC R0
7EFC 16F9    JNE J1
7EFE C803    MOV R3,@M2
7F00 7EDA
7F02 045B    B *R11
7F04 XXXX    END
```

## BINARY TO BCD

Program 12-6 converts a 16-bit binary number (4-digit hexadecimal) to BCD. The example number processed by the program is 1C53. The method of conversion is as follows:

1. Divide the number by 1000. Store the lower byte (07) of the result in memory.

Decimal	Hexadecimal
$\begin{array}{r} 7 \\ 1000 \overline{)7251} \\ \underline{7000} \\ 251 \end{array}$	$\begin{array}{r} 0007 \\ 03E8 \overline{)1C53} \\ \underline{1B58} \\ 00FB \end{array}$

2. Divide the remainder by 100. Store the lower byte (02) of the result in memory.

Decimal	Hexadecimal
$\begin{array}{r} 2 \\ 100 \overline{)251} \\ \underline{200} \\ 51 \end{array}$	$\begin{array}{r} 0002 \\ 0064 \overline{)00FB} \\ \underline{00C8} \\ 0033 \end{array}$

3. Divide the remainder by 10. Store the lower byte (05) of the result in memory.

Decimal	Hexadecimal
$\begin{array}{r} 5 \\ 10 \overline{)51} \\ \underline{50} \\ 1 \end{array}$	$\begin{array}{r} 0005 \\ 000A \overline{)0033} \\ \underline{0032} \\ 0001 \end{array}$

4. Store the lower byte (01) of the remainder in memory. Conversion is now complete. (Note that each divisor is one tenth of the previous divisor. However, in Step 4, dividing by one is an unnecessary operation.)

Program 12-6 stores the 8-bit BCD digits in memory locations 7F06-7F09. The divisors 1000, 100 and 10 are stored in locations 7F0A-7F0E.

Four registers are used in the conversion. Register 0 is used as a loop counter. This register is initially set to 3, because the division operation is performed three times, one for each of the first three digits to be determined. The fourth digit is simply the remainder from the previous division.

Register 1 is one of two pointers. This register points to, or contains the value of, the next divisor to be used. The first divisor is 1000. Register 2 is the second pointer. This register contains the address to which the next BCD digit (when determined) will be moved. The first digit will be moved to address 7F06.

Registers 3 and 4 are used in the division operation. In a divide

operation, the number to be divided (the dividend) must be a 32-bit number and must be stored in two adjacent registers. The instruction at 7F26 divides the 32-bit number contained in register 3 and 4 by the number whose address is stored in register 1. The quotient is stored in register 3, and the remainder is stored in register 4. Then it increments the address in register 1 by two.

Initially, register 3 is cleared and the number to be converted (1C53) is moved from 7F04 to register 4. After the first divide operation register 3 contains 0007 and register 4 contains 00FB.

**Table 12-2. Intermediate Results of Program 12-6.**

Instruction Address	Register Results				
	R0	R1	R2	R3	R4
7F14	0003	7F0A	7F06	0000 0007 0700	1C53 1C53 00FB
7F18					
7F1C					
7F20					
7F24					
7F26	0002	7F0C	7F07	0000 0002 0200	00FB 0033
7F28					
7F2A					
7F2C					
7F24					
7F26	0001	7F0E	7F08	0000 0005 0500	0033 0001
7F28					
7F2A					
7F2C					
7F24					
7F26	0000	7F10	7F09		0100
7F28					
7F2A					
7F2C					
7F30					

Before the next divide operation, the lower byte of register 3 must be saved and the register cleared, otherwise the next divide operation will be performed on the 32-bit number 000700FB.

The instruction at 72FA shifts the 8-bit BCD from the lower byte half of register 3 to the upper byte half. This is necessary because the MOVB (move byte) instruction at 7F2A moves the upper byte of register 3 to the address stored in register 2.

The loop 7F24-7F2E is executed three times, one time for each of the first three BCD digits computed. The fourth digit is the

last remainder. 7F30-7F32 moves the fourth digit to memory location 7F09. This operation completes the conversion.

Table 12-2 shows the intermediate register results of Program 12-6.

### **Program 12-6**

```
7D00 XXXX    AORG >7F04
7F04 1C52 M1  DATA >1C53
7F06 XXXX M2  BSS  4
7F0A 03E8 M3  DATA 1000
7F0C 0064     DATA 100
7F0E 000A     DATA 10
7F10 02E0     LWPI >70B8
7F12 70B8
7F14 0200     LI  R0,3
7F16 0003
7F18 0201     LI  R1,M3
7F1A 7F0A
7F1C 0202     LI  R2,M2
7F1E 7F06
7F20 C120     MOV @M1,R4
7F22 7F04
7F24 04C3 J1  CLR R3
7F26 3CF1     DIV *R1+,R3
7F28 0A83     SLA R3,8
7F2A DC83     MOVB R3,*R2+
7F2C 0600     DEC R0
7F2E 16FA     JNE J1
7F30 0A84     SLA R4,8
7F32 D484     MOVB R4,*R2
7F34 045B     B  *R11
7F36 XXXX     END
```

### **BINARY NUMBER TO ASCII-BINARY STRING**

Program 12-7 converts a binary number to an ASCII string.

This type of program is necessary if you want to display a binary number on the screen as a string of ones and zeros. This program converts a 16-bit binary number to a 16-character string. For example, in order to display 0011000111010010 (31D2, hexadecimal), each bit must be encoded in ASCII—30 for 0 and 31 for 1. When encoded in ASCII, 31D2 equals

30303131303030313131303130303130

Program 12-7 encodes the example number, 31D2, and stores the ASCII string in memory locations 7F38-7F47. The method of conversion begins by storing the number in a register. Shift left one bit one time. If the carry status bit equals 1, then store 31 at memory location 7F38. If the carry status bit equals 0, then store 30 at memory location 7F38. After a one bit shift, the carry status bit and register contents are as follows:

Carry	Register Contents
0	0110001110100100

Thus, 30 is stored at memory location 7F38.

Shift the register contents left one bit again. The result is:

Carry	Register Contents
0	1100011101001000

Thus, 30 is stored at memory location 7F39.

Shift the register contents left one bit again. The result is:

Carry	Register Contents
1	1000111010010000

Thus, 31 is stored at memory location 7F3A.

Repeat this operation until all sixteen bits have been shifted to the carry status bit and either a 30 or 31 for each bit has been stored consecutively in memory.

Program 12-7 implements this conversion very simply. 7F4C loads the number 16 into register 0, the loop counter. 7F50 moves the example number from location 7F36 to register 1. This register will be left-shifted one bit a total of sixteen times. The carry status bit will be tested after each shift operation.

7F54 loads the address of the first memory location where the

ASCII string will be stored. 7F58 loads 30 (ASCII for 0) into the upper byte half of register 3. 7F5C shifts the number one bit to the left. 7F5E causes the program to jump to 7F64 if the carry status bit is 0. (7F64 moves 30 to memory location 7F38 the first time the loop is executed.)

If the carry bit equals 1, then 0100 is added to register 3. Thus, the upper byte half of register 3 equals 31. Then 7F64 moves 31 to memory. (7F64 moves 31 to memory location 7F3A the third time the loop is executed.)

7F66-7F68 decrements the loop counter and causes the loop to be reexecuted if not equal to zero. If zero, the conversion is complete.

Note that each time through the loop, register 3 is reset to 3000. An alternative program follows. This program has one more instruction, but it is more straightforward than Program 12-7.

```

LI R0,16
MOV @M1,R1
LI R2,M1
LI R3,>3000
LI R4,>3100
J1 SLA R1,1
JNC J2
MOVB R4,*R2+
J2 MOVB R3,*R2+
DEC R0
JNE J1
B *R11

```

### Program 12-7

```

7D00 XXXX    AORG >7F36
7F36 31D2 M1  DATA >31D2
7F38 XXXX M2  BSS  16
7F48 02E0    LWPI >70B8
7F4A 70B8
7F4C 0200    LI  R0,16
7F4E 0010
7F50 C060    MOV @M1,R1
7F52 7F36
7F54 0202    LI  R2,M2

```

```

7F56 7F38
7F58 0203 J1 LI R3, > 3000
7F5A 3000
7F5C 0A11 SLA R1,1
7F5E 1702 JNC J2
7F60 0223 AI R3,>0100
7F62 0100
7F64 DC83 J2 MOVB R3,*R2+
7F66 0600 DEC R0
7F68 16F7 JNE J1
7F6A 045B B *R11
7F6C XXXX END

```

## ASCII-BINARY STRING TO BINARY NUMBER

Program 12-8 converts an ASCII string of ones and zeros to a binary number. This type of program is necessary in order to process a binary number entered into memory via the keyboard. Program 12-8 converts the ASCII code for 0001110001010010 (1C52, hexadecimal) to binary. The ASCII code for 1C52 is as follows:

30303031313130303031303130303130

The conversion begins by clearing a register to all zeros. Now for every 31 in the ASCII string, set the corresponding bit in the register to 1.

To set individual bits to 1 I must use the SOC, set ones corresponding, instruction. To use this instruction I must set up a reference, or *mask*, register. Program 12-8 uses register 3 as a mask register. The initial value of register 3 is 8000, or 1000000000000000 binary. In other words, register 3 has a 1 in bit position 0 and 0s in bit positions 1-15.

Let's start with the first byte in the ASCII string, which happens to be 30, or 0 binary.

1. Clear register 6.
2. Move the first byte to register 5.
3. Subtract 30 from register 5. The result is zero. Therefore, do not set bit 0 of register 6.



4. Shift register 3 one bit to the right.
5. Move the next byte to register 5.
6. Subtract 30 from register 5. The result is zero. Therefore, do not set bit 1 of register 6. Note that bit 1 of register 3 is a 1.

Register 6	Register 3
0000000000000000	0100000000000000

7. Perform steps 4-6 again. Register 5 equals 0. Therefore, do not set bit 2 of register 6. Note that bit 2 of register 3 is a 1.

Register 6	Register 3
0000000000000000	0010000000000000

8. Perform steps 4-6 again. Register 5 equals 1. Therefore, set bit 3 of register 6 to a 1. Note that bit 3 of register 3 is a 1.

Register 6	Register 3
0001000000000000	0001000000000000

Finally, a 31 is detected. The SOC instruction at 7FA6 in Program 12-8 sets the bits in register 6 for which there are corresponding 1s in the register 3. There is only 1 in register 3—in bit position 3. Thus, bit 3 in register 6 is set to 1.

Table 12-3 shows the intermediate results for registers 3 and 6 after each execution of the instruction at 7FA6. Note that bits in register 6 that have been set by prior SOC operations are unaffected by succeeding SOC operations.

Program 12-8 has one other important feature. That is, it will convert strings of variable length. The program looks for a 20 which is ASCII for a space. Once a space is found, it is necessary to right-justify the result. The number of bit positions that the result must be shifted is contained in register 0.

The SRC (shift right circular) instruction is used instead of SRL (shift right logical). This is necessary for the one case when register 0 equals 0, meaning that a 16-character string has been converted. In this case, the SRL would perform a 16-bit shift, causing the result to be wiped out. All sixteen bits shifted would be replaced by zeros.

The SRC instruction circulates the bits. This satisfies all cases. If register 0 equals 0, then 16-bit circulate shifts the previous result back into the register. If register zero is greater than zero,

**Table 12-3. Intermediate Register Results of Program 12-8.**

Register 3	Register 6
0000000000000000	1000000000000000
0000000000000000	0100000000000000
0000000000000000	0010000000000000
0001000000000000	0001000000000000
0001100000000000	0000100000000000
0001110000000000	0000010000000000
0001110000000000	0000001000000000
0001110000000000	0000000100000000
0001110000000000	0000000010000000
0001110001000000	0000000001000000
0001110001000000	0000000000100000
0001110001010000	0000000000010000
0001110001010000	0000000000001000
0001110001010000	0000000000000100
0001110001010010	0000000000000010
0001110001010010	0000000000000001

then that number of zeros are circulated back into register 6, causing the number to be right-justified.

When the conversion is complete, the result is moved to memory location 7F7E. For this example, the result is 1C52, hexadecimal.

### **Program 12-8**

```
7D00 XXXX    AORG >7F6C
```

```
7F6C 3030 M1 TEXT '0001110001010010
```

```
7F6E 3031
```

```
7F70 3131
```

7F72 3030  
 7F74 3031  
 7F76 3031  
 7F78 3030  
 7F7A 3130  
 7F7C 2000  
 7F7E XXXX M2 BSS 2  
 7F80 02E0     LWPI >70B8  
 7F82 70B8  
 7F84 0200     LI R0,16  
 7F86 0010  
 7F88 0201     LI R1,>3000  
 7F8A 3000  
 7F8C 0202     LI R2,>2000  
 7F8E 2000  
 7F90 0203     LI R3,>8000  
 7F92 8000  
 7F94 0204     LI R4,M1  
 7F96 7F6C  
 7F98 04C5     CLR R5  
 7F9A 04C6     CLR R6  
 7F9C D174 J1 MOVB \*R4+,R5  
 7F9E 9085     CB R5,R2  
 7FA0 1306     JEQ J3  
 7FA2 7141     SB R1,R5  
 7FA4 1301     JEQ J2  
 7FA6 E183     SOC R3,R6  
 7FA8 0913 J2 SRL R3,1  
 7FAA 0600     DEC R0  
 7FAC 10F7     JMP J1  
 7FAE 0B06     SRC R6,0  
 7FB0 C806 J3 MOV R6,@M2

7FB2 7F7E

7FB4 045B    B \*R11

7FB6 XXXX    END

## BINARY NUMBER TO ASCII-HEXADECIMAL STRING

Program 12-9 converts a 16-bit binary number to 4-character ASCII string. This type of program is necessary when you want to display a binary number on the screen as a 4-digit hexadecimal number. For example, the assembler displays the memory address and updated data and next address and current data each time you enter an assembly language instruction. In order for the assembler to display 7D00 31D2, the assembler must first convert the binary equivalent of 7D00 and 31D2 to the ASCII codes 37443030 and 33314432, respectively.

Program 12-9 converts the example number, 31D2, to the ASCII code 33314432. This is accomplished in two steps. Each step is a small program in itself.

The first step (7D0A-7D2C) disassembles the 16-bit number into four eight bit numbers and stores them in memory locations 7D02-7D05. The first four bits are converted to 03, the second four bits to 01, the third four bits to 0D, and the last four bits to 02.

The second step (7D2E-7D4A) converts the 8-bit hexadecimal digits to ASCII and stores results back in locations 7D02-7D05, copying over the results of the first step. This second step is essentially the same as Program 12-1. The main difference is that Program 12-9 processes four digits and uses indirect addressing.

Now let's look at how the digits are disassembled in the first step. 7D0A moves the number 31D2 to register 0. 7D0E loads the starting address of the four bytes where the results will be stored. 7D12 copies the number 31D2 into register 2. 7D14 shifts the number to the right four bit positions. Positions vacated are replaced by zeros. Register 2 now equals 031D. 7D16 moves the byte 03 to 7D02. 7D18 copies the number 31D2 into register 2 again.

7D1A replaces the first four bits with zeros. The ANDI (AND Immediate) instruction logically ANDs 31D2 with OFFF (called the *mask*) and stores the result back in register 2. The AND operation is performed bit by bit:

31D2 = 0011000111010010  
0FFF = 0000111111111111  
31D2 AND 0FFF = 0000000111010010 = 01D2

Only in bit positions where both 31D2 and 0FFF have a 1 will a 1 occur in the result. Looking at the hexadecimal representations, you can see that a 0 in the mask number clears corresponding bits in the other number and that an F in the mask number leaves the corresponding bits in the other number unchanged.

Next, 7D1E moves the byte 01 to memory location 7D03. 7D20 swaps the bytes in register 2. 01D2 becomes D201. 7D22 shifts the D201 to the right four bit positions. Register 2 now equals 0D02. 7D24 moves the byte 0D to memory location 7D04. 7D26 shifts 7D20 to the left four bit positions. Register 2 now equals D200. 7D26 replaces the first four bits with zeros. The result is 0200. 7D28 moves the last byte, 02, to memory location 7D05.

### **Program 12-9**

```
7D00 31D2 M1 DATA >31D2
7D02 XXXX M2 BSS 4
7D06 02E0     LWPI >70B8
7D08 70B8
7D0A C020     MOV @M1,R0
7D0C 7D00
7D0E 0201     LI R1,M2
7D10 7D02
7D12 C080     MOV R0,R2
7D14 0942     SRL R2,4
7D16 DC42     MOVB R2,*R1+
7D18 C080     MOV R0,R2
7D1A 0242     ANDI R2,>0FFF
7D1C 0FFF
7D1E DC42     MOVB R2,*R1+
7D20 06C2     SWPB R2
7D22 0942     SRL R2,4
7D24 DC42     MOVB R2,*R1+
7D26 0A42     SLA R2,4
7D28 0242     ANDI R2,>0FFF
7D2A 0FFF
7D2C D442     MOVB R2,*R1
```

```

7D2E 0200    LI R0,4
7D30 0004
7D32 0201    LI R1,M2
7D34 7D02
7D36 0091 J1  MOVB *R1,R2
7D38 0282    CI R2,>0A00
7D3A 0A00
7D3C 1A02    JL J2
7D3E 0222    AI R2,>0700
7D40 0700
7D42 0222 J2  AI R2,>3000
7D44 3000
7D46 DC42    MOVB R2,*R1+
7D48 0600    DEC R0
7D4A 16F5    JNE J1
7D4C 045B    B *R11
7D4E XXXX    END

```

## ASCII-HEXADECIMAL STRING TO BINARY NUMBER

Program 12-10 converts an ASCII string of four hexadecimal digits to a binary number. This type of program is required to process a 4-digit hexadecimal number which has been input via the keyboard. For example, when you use the assembler and type **AORG >7D4E**, the assembler must convert the ASCII code for 7D4E to a binary number as part of the processing required to display address 7D4E and the current data, and gets ready for you to type the next command or assembly language instruction.

Program 12-10 converts the example ASCII code 33314432 to the binary number 31D2 (hexadecimal form). First 7D58 loads register 0 with the number 4. Register 0 is the loop counter. Four bytes of ASCII will be processed.

7D5C loads the address of the first ASCII byte into register 1. 7D5E adds three to that address. The address in register 1 is now 7D51, the address of the last ASCII byte. Program 12-10 will process the ASCII string backwards, starting with the last byte and proceeding to the first byte.

7D64 and 7D66 load the values 30 and 07 into the upper byte halves of registers 2 and 3, respectively. These constants will be

used to convert each ASCII byte to an 8-bit hexadecimal digit.

7D6C and 7D6E clear registers 4 and 5, which will be used to assemble the 16-bit number 31D2 from the individual bytes 03, 01, 0D, and 02.

7D70 moves the last ASCII byte, 32, to register 5. 7D72 subtracts 30 from 32. The result is 02. 7D74 compares 02 with 0A. 7D76 causes the program to jump to 7D7C if the number is in the range 0-9, otherwise the next instruction 7D7A will subtract 07. (Review program 12-2, if necessary.)

7D7C adds 0200 to 0000 (the initial value of register 4). The result is 0200. 7D7E shifts 0200 to the right four bit positions and circulates the right-most four bits into the left-most bit positions. The result is 0020.

7D80 decrements the address in register 1. The address is now 7D50, pointing at ASCII byte 44, the next byte to be processed.

7D82 decrements the loop counter and causes the program to jump to 7D70 if not zero. The second time through the loop, 7D70-7D7A converts the byte 44 to 0D. Then the instruction at 7D7C adds 0D00 to 0020 to get 0D20. 7D7E circulates 0D20 four bit positions. The result is now 00D2.

The third time through the loop, 7D70-7D7A converts the byte 31 to 01. Then the instruction at 7D7C adds 0100 to 00D2 to get 01D2. 7D7E circulates 01D2 four bit positions. The result is now 201D.

The fourth time through the loop, 7D70-7D7A converts the byte 33 to 03. Then the instruction at 7D7C adds 03 to 201D to get 231D. 7D7E circulates 231D four bit positions. The result is now D231.

The loop counter now equals zero. The loop is exited and the next instruction at 7D86 swaps the bytes in register 4. The result is 31D2. 7D88 moves the final result to memory location 7D52.

## Program 12-10

```
7D00 XXXX    AORG >7D4E
7D4E 3331 M1 TEXT '31D2'
7D50 4432
7D52 XXXX M2 BSS 2
7D54 02E0    LWPI >70B8
7D56 70B8
7D58 0200    LI R0,4
```

```

7D5A 0004
7D5C 0201    LI R1,M1
7D5E 7D4E
7D60 0221    AI R1,3
7D62 0003
7D64 0202    LI R2,>3000
7D66 3000
7D68 0203    LI R3,>0700
7D6A 0700
7D6C 04C4    CLR R4
7D6E 04C5    CLR R5
7D70 D151 J1 MOVB *R1,R5
7D72 7142    SB R2,R5
7D74 0285    CI R5,>0A00
7D76 0A00
7D78 1A01    JL J2
7D7A 7143    SB R3,R5
7D7C A105 J2 A R5,R4
7D7E 0B44    SRC R4,4
7D80 0601    DEC R1
7D82 0600    DEC R0
7D84 16F5    JNE J1
7D86 06C4    SWPB R4
7D88 C804    MOV R4,@M2
7D8A 7D52
7D8C 045B    B *R11
7D8E XXXX    END

```

## **BINARY NUMBER TO ASCII-DECIMAL STRING**

Program 12-11 converts a 16-bit binary number to an ASCII string in decimal form. This type of program is used to display a program result on the screen as a decimal number. Program 12-11 is a combination of three previous programs: 12-6, Binary to BCD, 12-4, Decimal to ASCII, and 11-4, Replace leading zeros with blanks.



The instructions at 7DA2-7DCE convert 31D2 to 313237353420 (the decimal number 12754 followed by a space). 7DB4 divides the number by 10,000 (2710 hexadecimal). 7DB6 shifts the quotient 0001 to the left eight bit positions. The result in register 3 is 0100. 7DB8 adds 3000 to 0100 to get 3100. 7DBC stores 31 at 7D90.

The second time through the loop, the remainder from the first division is divided by 1000 (03E8 hexadecimal). The result is 0002. 7DBC stores 32 at 7D91.

The third time through the loop, the remainder from the second division is divided by 100 (0064 hexadecimal). The result is 0007. 7DBC stores 37 at 7D92.

The fourth time through the loop, the remainder from the third division is divided by 10 (000A hexadecimal). The result is 0005. 7DBC stores 35 at 7D93. The loop is exited and 7DC2 shifts the remainder 0004 to the upper byte position. 7DC4 adds 3000 to get 3400 and 7DC8 moves 34 to 7D94. 7DCA-7DCE loads 20 (ASCII for a blank) into memory location 7D95. At this point conversion is complete.

7DD0-7DE8 replaces leading zeros with blanks. To test this code you should try a different example number that is less than 10,000 and see if the leading zeros (30 in ASCII) are replaced with blanks (20). Try 0100 and see if the result at 7D90-7D95 is 202032353620 (256 decimal). Try 0000 and see if the result is 202020203020.

## Program 12-11

```
7D00 XXXX    AORG >7D8E
7D8E 31D2 M1  DATA >31D2
7D90 XXXX M2  BSS  6
7D96 2710 M3  DATA 10000
7D98 03E8     DATA 1000
7D9A 0064     DATA 100
7D9C 000A     DATA 10
7D9E 02E0     LWPI >70B8
7DA0 70B8
7DA2 0200     LI  R0,4
7DA4 0004
7DA6 0201     LI  R1,M3
```

7DA8 7D96  
 7DAA 0202 LI R2,M2  
 7DAC 7D90  
 7DAE C120 MOV @M1,R4  
 7DB0 7D8E  
 7DB2 04C3 J1 CLR R3  
 7DB4 3CF1 DIV \*R1+,R3  
 7DB6 0A83 SLA R3,8  
 7DB8 0223 AI R3,>3000  
 7DBA 3000  
 7DBC DC83 MOVB R3,\*R2+  
 7DBE 0600 DEC R0  
 7DC0 16F8 JNE J1  
 7DC2 0A84 SLA R4,8  
 7DC4 0224 AI R4,>3000  
 7DC6 3000  
 7DC8 DC84 MOVB R4,\*R2+  
 7DCA 0201 LI R1,>2000  
 7DCC 2000  
 7DCE D481 MOVB R1,\*R2  
 7DD0 0200 LI R0,>3000  
 7DD2 3000  
 7DD4 0203 LI R3,M2  
 7DD6 7D90  
 7DD8 9013 J2 CB \*R3,R0  
 7DDA 1602 JNE J3  
 7DDC DCC1 MOVB R1,\*R3+  
 7DDE 10FC JMP J2  
 7DE0 9053 J3 CB \*R3,R1  
 7DE2 1301 JEQ J4  
 7DE4 1002 JMP J5  
 7DE6 0603 J4 DEC R3

7DE8 DC40     MOVB R0,\*R3

7DEA 045B J5 B \*R11

7DEC XXXX     END

## ASCII-DECIMAL STRING TO BINARY NUMBER

Program 12-12 converts a 5-character ASCII code for the decimal number 12754 to its binary equivalent. This type of program is used to convert decimal numbers that are entered via the keyboard.

This program is a composite of Program 12-3 (ASCII to decimal) and Program 12-5 (Binary-coded decimal to binary).

7DFE-7E1E converts the ASCII code 3132373534 to 0102070504 and stores the result at 7DF2-7DF7. Each time a digit is converted, register 5 is incremented. Conversion stops when a 20 is encountered. Register 5 then contains the number of digits processed and will be used as a loop counter in the next section (7D20-7E3C), which converts the BCD to binary and stores the result 31D2 at memory location 7DF8.

### Program 12-12

7D00 XXXX     AORG >7DEC

7DEC 3132 M1 TEXT '12754 '

7DEE 3735

7DF0 3420

7DF2 XXXX M2 BSS 6

7DF8 XXXX M3 BSS 2

7DFA 02E0     LWPI >70B8

7DFC 70B8

7DFE 0200     LI R0,>2000

7E00 2000

7E02 0201     LI R1,>3000

7E04 3000

7E06 0202     LI R2,M2

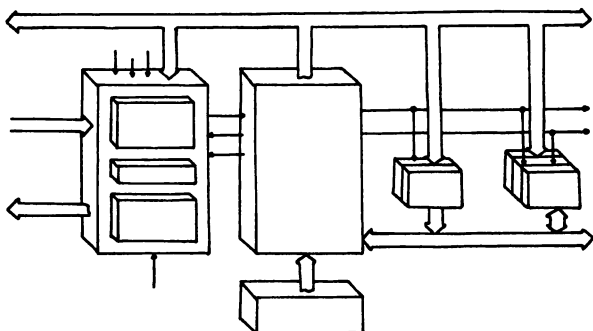
7E08 7DF2

7E0A 0203     LI R3,M1

7E0C 7DEC

7EDE 04C4	CLR R4
7E10 04C5	CLR R5
7E12 D133 J1	MOVB *R3+,R4
7E14 9004	CB R4,R0
7E16 1304	JEQ J2
7E18 7101	SB R1,R4
7E1A DC84	MOVB R4,*R2+
7E1C 0585	INC R5
7E1E 10F9	JMP J1
7E20 0201 J2	LI R1,10
7E22 000A	
7E24 0202	LI R2,M2
7E26 7DF2	
7E28 04C3	CLR R3
7E2A 1002	JMP J4
7E2C 38C1 J3	MPY R1,R3
7E2E C0C4	MOV R4,R3
7E30 D132 J4	MOVB *R2+,R4
7E32 0984	SRL R4,8
7E34 A0C4	A R4,R3
7E36 0605	DEC R5
7E38 16F9	JNE J3
7E3A C803	MOV R3,@M3
7E3C 7DF8	
7E3E 045B	B *R11
7E40 XXXX	END

## Chapter 13



### Arithmetic Problems

In this chapter, I will discuss the following five arithmetic problems: 32-bit by 32-bit multiply, 64-bit division, Square root, Reciprocal of a number, and Sine of an angle. In addition, the sine problem is repeated in programs 13-6 and 13-7 in order to demonstrate simple subroutine techniques using the BL and BLWP instructions.

#### 32-BIT BY 32-BIT MULTIPLY

Program 13-1 multiplies the 32-bit number 002468AC times the 32-bit number 03281088. The result is the 64-bit number 000072ECB8C25B60. The multiplication is accomplished by performing a total of four 16-bit by 16-bit multiplications. Partial products are added to form a final 64-bit product. Register and memory usage is shown in Fig. 13-1.

The instructions at 7E5C-7E5E move the multiplier to registers 2 and 3, the most significant sixteen bits 0024 to register 2 and the least significant sixteen bits 68AC to register 3.

The instructions at 7E60-7E62 move the multiplicand to registers 4 and 5, 0328 to register 4 and 1088 to register 5.

7E64 multiplies 68AC times 1088. 7E66 saves the least significant sixteen bits 5B60 in register 7. 7E68 saves the most significant sixteen bits 06C2 in memory location 7E48. See Fig. 13-1. You should be able to see that the contents of registers 5 and 6 must be saved. Register 5 will be overwritten in the next three

REGISTERS					MEMORY		
		R4	0328	R5	1088	7E40	0024
		R2	0024	R3	68AC	7E42	68AC
		R5	06C2	R6	5B60	7E44	0328
	R4	014A	R5	5EE0		7E46	1088
	R5	0002	R6	5320		7E48	06C2
R4	0000	R5	71A0			7E4A	5EE0
R4	0000	R5	72EC	R6	B8C2	7E4C	014A
				R7	5B60	7E4E	0002

Fig. 13-1. Register and memory usage of Program 13-1.

multiplications. Register 6 will be written over in the multiplication at 7E76.

7E6A multiplies 68AC times 0328. 7E6C saves 5EE0 in memory location 7E4A and the instruction at 7E6E saves 014A in memory location 7E4C. The two previous multiplications have wiped out the multiplicand. Therefore, the instructions at 7E70-7E74 are necessary to restore the multiplicand 03281088 to registers 4 and 5.

7E76 multiplies 0024 times 1088. 7E78 saves 0002 in memory location 7E4E. It is not necessary to save 5320 (the contents of register 6) because the next multiplication will not write over this number. See Fig. 13-1.

7E7A multiplies 0024 times 0328. This completes the generation of partial products.

The instructions at 7E7C-7E9E add the partial products. The contents of memory locations 7E48 and 7E4A are added to the contents of register 6. Register 8 is used to keep track of carries. Then the contents of memory locations 7E4C and 7E4E are added to the contents of register 5. Any carries in register 8 are also added. Register 4 is incremented if any carries result.

The final 64-bit result is now contained in registers 4 through 7 as you can see in Fig. 13-1. The instructions at 7EA0-7EAA move the final number to memory locations 7E48-7E4E, overwriting partial products previously generated.

After you run the program in EASY BUG, run the program again using the multiplier A02468AC and the multiplicand E328C088. This combination generates a carry in register 8 and a carry which is added to register 4. The result should be 8E19C6F140B89B60.

	E328	C088	
	<u>A024</u>	<u>68AC</u>	
	4EB8	9B60	
5CE0	DEE0		
7870	1320		
8E18	F1A0		
8E19	C6F1	40B8	9B60

### Program 13-1

```

7D00 XXXX    AORG >7E40
7E40 0024 M1  DATA >0024
7E42 68AC    DATA >68AC
7E44 0328    DATA >0328
7E46 1088    DATA >1088
7E48 XXXX M2 BSS 8
7E50 02E0    LWPI >70B8
7E52 70B8
7E54 0200    LI  R0,M1
7E56 7E40
7E58 0201    LI  R1,M2
7E5A 7E48
7E5C C0B0    MOV *R0+,R2
7E5E C0F0    MOV *R0+,R3
7E60 C130    MOV *R0+,R4
7E62 C150    MOV *R0+,R5
7E64 3943    MPY R3,R5
7E66 C1C6    MOV R6,R7
7E68 CC45    MOV R5,*R1+
7E6A 3903    MPY R3,R4
7E6C CC45    MOV R5,*R1+

```

7E6E	CC44	MOV R4,*R1+
7E70	0640	DECT R0
7E72	C130	MOV *R0+,R4
7E74	C150	MOV *R0,R5
7E76	3942	MPY R2,R5
7E78	C445	MOV R5,*R1
7E7A	3902	MPY R2,R4
7E7C	0201	LI R1,M2
7E7E	7E48	
7E80	04C8	CLR P8
7E82	A1B1	A *R1+,R6
7E84	1701	JNC J1
7E86	0588	INC R8
7E88	A1B1 J1	A *R1+,R6
7E8A	1701	JNC J2
7E8C	0588	INC R8
7E8E	A171 J2	A *R1+,R5
7E90	1701	JNC J3
7E92	0584	INC R4
7E94	A171 J3	A *R1+,R5
7E96	1701	JNC J4
7E98	0584	INC R4
7E9A	A148 J4	A R8,R5
7E9C	1701	JNC J5
7E9E	0584	INC R4
7EA0	0201 J5	LI R1,M2
7EA2	7E48	
7EA4	CC44	MOV R4,*R1+
7EA6	CC45	MOV R5,*R1+
7EA8	CC46	MOV R6,*R1+
7EAA	C447	MOV R7,*R1
7EAC	045B	B *R11
7EAE	XXXX	END



## 64-BIT DIVISION

Program 13-2 divides the 32-bit number A02468AC into the 64-bit number 8E19C6F140B89B60 to get the 32-bit quotient E328C088 and a remainder of zero. The problem and solution in hexadecimal notation and standard division representation looks like this:

A024 68AC		8E19 C6F1 40B8 9B60	E328 C088
		8E19 48E0 DEE0	
		7870 61D8 9B60	
		7870 61D8 9B60	
		0000 0000	

Let's look at Program 13-2 line-by-line to see how this is accomplished. 7EC6 loads the number 2 into register 12, which will be used as a loop counter. The loop 7EE4-7F16 is executed twice, once to determine the sixteen most significant bits of the 32-bit quotient and once to determine the sixteen least significant bits of the 32-bit quotient.

7ECA loads the address 7EBA into register 13. 7EBA is the address at the first memory location in an 8-byte block used to store the 32-bit quotient and 32-bit remainder.

7ECE loads the dividend address 7EAE into register 0. 7DD2 loads the divisor address 7EB6 into register 1. 7ED6-7ED8 loads 8E19C6F1 into the register 2-3 combination. 7EDA-7EDE loads 8E19C6F140B8 into the register 4-5-6 combination. 7EE0 loads A024 into register 7.

I am now ready to perform a trial division. I will divide A024 into 8E19C6F1 and get a trial quotient. Then I will multiply the trial quotient times the entire divisor A02468AC to get a 48-bit result which I will compare with 8E19C6F140B8, which was stored in the register 4-5-6 combination. If the 48-bit product is smaller than 8E19C6F140B8, then I will subtract that 48-bit product from 8E19C6F140B8. If the 48-bit product is larger than 8E19C6F140B8, then I must decrement the trial quotient by one and multiply it times the divisor A02468AC again. The new 48-bit product will be compared with 8E19C6F140B8. If smaller (which it should be at this point), then the 48-bit product is subtracted from 8E19C6F140B8. The difference is combined with 9B60 (the last sixteen bits of the 64-bit dividend) to form the next dividend to be divided into.

Let's see what actually occurs. 7EE2 divides A024 into 8E19C6F1.

$$\begin{array}{r} \text{E329} \\ \text{A024 } \overline{)8\text{E19C6F1}} \end{array}$$

I don't care what the remainder is. E329 is the trial quotient. 7EE4 loads 68AC into register 9. 7EE6-7EEE multiplies E329 times A02468AC and adds the partial products to get 8E19EEA5478C which is contained in the register 7-8-10 combination.

$$\begin{array}{r} \begin{array}{cc} \text{R7 } \boxed{\text{A024}} & \text{R9 } \boxed{\text{68AC}} \\ & \text{R2 } \boxed{\text{E329}} \\ \hline & \text{R9 } \boxed{\text{5CE1}} & \text{R10 } \boxed{\text{478C}} \\ \text{R7 } \boxed{\text{8E19}} & \text{R8 } \boxed{\text{91C4}} & \\ \hline \text{R7 } \boxed{\text{8E19}} & \text{R8 } \boxed{\text{EEA5}} & \text{R10 } \boxed{\text{478C}} \end{array} \end{array}$$

7EF0-7EFE compares 8E19C6F140B8 (the first forty-eight bits of our 64-bit dividend) to 8E19CEEA5478C sixteen bits at a time. 8E19 (register 4) is compared to 8E19 (register 7). If the number in register 4 had been lower than the number in register 7, then the comparison process would have stopped. The program would go to the next step, which is to subtract the 48-bit product (generated by 7EE6-7EEE) from 8E19C6F140B8. If the number in register 4 had been higher than register 7, then the trial quotient is too high. The program would jump to 7F00-7F08 which decrements the trial quotient, restores register 7 (register 9 is restored at the first instruction in the loop), and causes the program to jump to the beginning of the loop (7EE4).

Since however, the number in register 4 equals the number in register 7, the next sixteen bits are compared. C6F1 (register 5) is compared to EEA5 (register 8). C6F1 is lower. Therefore, the program jumps to 7F00. 7F00 decrements the trial quotient from E329 to E328. 7F02-7F04 restores A024 to register 7. 7F08 causes the program to jump to 7EE4.

7EE6-7EEE now multiplies E328 times A02468AC to get 8E1948E0DEE0. 7EF0-7EEE compares 8E19C6F140B8 to 8E1948E0DEE0 sixteen bits at a time. 8E19 equals 8E19. But now C6F1 is higher than the number to which it is compared—48E0. Therefore, the program jumps 7F0A. 7F0A saves E328 in memory location 7EBA.

7F0C subtracts DEE0 from 40B8. A borrow occurs and C6F1 is decremented to C6F0. 7F12 subtracts 48E0 from C6F0.

8E19	C6F1	40B8
8E19	48E0	DEE0
<hr/>		
	7870	61D8

The problem and partial solution are as follows:

				E328
A024	68AC	<hr/>	8E19	C6F1
			40B8	9B60
			8E19	48E0
			DEE0	
			<hr/>	
			7870	61D8
				9B60

7F14 decrements the loop counter. Since it is not zero yet, the program executes instructions 7F18-7F2A, which prepare the registers for the next pass through the loop.

7F18-7F1A loads 787061D8 into the register 2-3 combination. 7F1C-7F20 loads 787061D89B60 into the register 4-5-6 combination. 7F26 performs the trial division to get the next quotient, the least sixteen significant bits of the final 32-bit quotient.

7F28 causes the program to jump to 7EE4, the beginning of the loop. The second time through the loop the second half of the quotient is generated—C088. 7F0A stores C088 at memory location 7EBC. 7F14 decrements the loop counter. Since it is now equal to zero, the program jumps to 7F2C.

7F2C-7F2E moves the remainder to memory locations 7EBE-7EC0. The remainder is zero. To get a nonzero remainder, divide A02468AC into 8E19C6F15AB99BAF. The remainder will be 1A01004F. The quotient will be the same—E328C088.

## Program 13-2

```

7D00 XXXX    AORG > 7EAE
7EAE 8E19 M1  DATA > 8E19
7EB0 C6F1    DATA > C6F1
7EB2 40B8    DATA > 40B8
7EB4 9B60    DATA > 9B60
7EB6 A024 M2  DATA > A024
7EB8 68AC    DATA > 68AC
7EBA XXXX M3  BSS  8
7EC2 02E0    LWPI > 70B8
7EC4 70B8

```

7EC6 020C	LI R12,2
7EC8 0002	
7ECA 020D	LI R13,M3
7ECC 7EBA	
7ECE 0200	LI R0,M1
7ED0 7EAE	
7ED2 0201	LI R1,M2
7ED4 7EB6	
7ED6 C0B0	MOV *R0+,R2
7ED8 C0F0	MOV *R0+,R3
7EDA C102	MOV R2,R4
7EDC C143	MOV R3,R5
7EDE C1B0	MOV *R0+,R6
7EE0 C1F1	MOV *R1+,R7
7EE2 3C87	DIV R7,R2
7EE4 C251 J1	MOV *R1,R9
7EE6 3A42	MPY R2,R9
7EE8 39C2	MPY R2,R7
7EEA A209	A R9,R8
7EEC 1701	JNC J2
7EEE 0587	INC R7
7EF0 81C4 J2	C R4,R7
7EF2 1A06	JL J3
7EF4 1B0A	JH J4
7EF6 8205	C R5,R8
7EF8 1A03	JL J3
7EFA 1B07	JH J4
7EFC 8286	C R6,R10
7EFE 1405	JHE J4
7F00 0602 J3	DEC R2
7F02 0201	LI R1,M2
7F04 7EB6	
7F06 C1F1	MOV *R1+,R7

```

7F08 10ED    JMP J1
7F0A CF42 J4 MOV R2,*R13+
7F0C 618A    S R10,R6
7F0E 1801    JOC J5
7F10 0605    DEC R5
7F12 6148 J5 S R8,R5
7F14 060C    DEC R12
7F16 130A    JEQ J6
7F18 C085    MOV R5,R2
7F1A C0C6    MOV R6,R3
7F1C C105    MOV R5,R4
7F1E C146    MOV R6,R5
7F20 C190    MOV *R0,R6
7F22 0201    LI R0,M2
7F24 7EB6
7F26 C1F1    MOV *R1+,R7
7F28 3C87    DIV R7,R2
7F2A 10DC    JMP J1
7F2C CF45 J6 MOV R5,*R13+
7F2E C746    MOV R6,*R13
7F30 045B    B *R11
7F32 XXXX    END

```

## SQUARE ROOT

Program 13-3 takes the square root of the unsigned number 10,000 (2710 hexadecimal), using a successive approximation method.

The first approximation is equal to the number (10,000) divided by 200, plus 2:

$$\text{1st approx.} = (10,000/200) + 2 = 52$$

The second and succeeding approximations are as follows:

$$\text{2nd approx.} = ((10,000/52) + 52)/2 = (192 + 52)/2 = 122$$

3rd approx. =  $((10,000/122) + 122)/2 = (81 + 122)/2 = 101$   
 4th approx. =  $((10,000/101) + 101)/2 = (99 + 101)/2 = 100$   
 5th approx. =  $((10,000/100) + 100)/2 = (100 + 100)/2 = 100$

In this case, the fourth approximation is the correct number. A program to implement the above process can go on indefinitely unless some criteria for ending the program is devised. One way is to stop the program when two successive approximations are equal. This will work for numbers with integer square roots only. It would not work, for example, if the number to be processed was 10,003.

A second method is to stop the program when two successive approximations are within plus or minus one. Program 13-3 uses a combination of both methods.

7F3A loads the number 200 into register 0. 7F3E clears register 1. The register 1-2 combination is used for the 32-bit dividend which varies as the program is executed. The initial dividend is 10,000. 7F40 moves the number 10,000 (the candidate number) to register 2. 7F44 divides 200 into 10,000 to get 50. 7F46 adds 2 to 50 to get 52, the first approximation. 7F48 saves 52 in register 3. 7F4A-7F4C loads 10,000 into the register 1-2 combination again. 7F50 divides 52 into 10,000 to get 192. 7F52 adds 52 to 192 to get 244.

7F54 divides 244 by 2 to get 122, the second approximation. (Recall that a one-bit left shift is equivalent to division by 2.) 7F56 compares 52 to 192. If they were equal, the program would jump to 7F68 which saves the result in memory location 7F34. 7F5A subtracts 192 from 52 to get -140. 7F4C compares -140 to 1. If equal, the program jumps to 7F68. 7F62 compares -140 to -1. If not equal, the program jumps to 7F48 to derive the next approximation.

The next approximation is 101. The one after that is 100. The program ends.

### Program 13-3

```
7D00 XXXX    AORG >7F32
7F32 2710 M1  DATA 10000
7F34 XXXX M2  BSS 2
7F36 02E0    LWPI >70B8
7F38 70B8
7F3A 0200    LI R0,200
7F3C 00C8
```

```

7F3E 04C1    CLR R1
7F40 COA0    MOV @M1,R2
7F42 7F32
7F44 3C40    DIV R0,R1
7F46 05C1    INCT R1
7F48 COC1 J1 MOV R1,R3
7F4A 04C1    CLR R1
7F4C COA0    MOV @M1,R2
7F4E 7F32
7F50 3C43    DIV R3,R1
7F52 A043    A R3,R1
7F54 0911    SRL R1,1
7F56 8043    C R3,R1
7F58 1307    JEQ J2
7F5A 60C1    S R1,R3
7F5C 0283    CI R3,1
7F5E 0001
7F60 1303    JEQ J2
7F62 0283    CI R3,-1
7F64 FFFF
7F66 16F0    JNE J1
7F68 C801 J2 MOV R1,@M2
7F6A 7F34
7F6C 045B    B *R11
7F6E XXXX    END

```

## RECIPROCAL OF A NUMBER

Up to this point, I have been working with integers, or numbers with no fractional components. Now you will see how the computer handles fractions. The purpose of Program 13-4 is to take the reciprocal of the number 20. The reciprocal of 20 is  $1/20$  or 0.05 decimal. In binary, the fraction 0.05 may be expressed as

.0000110011001100

Each bit position to the right of the decimal point has a place value. The place value of bit 0 (the first bit to the right of the decimal point) is  $\frac{1}{2}$  or .5. The place value of bit 1 is  $\frac{1}{4}$  or .25. Each succeeding bit position has half the value of the preceding bit position. The place values of bits 0-15 are given in Table 13-1.

Using Table 13-1, I can compute the binary equivalent of  $\frac{1}{20}$  to sixteen bits. For each 1 in the above expression, I look up the corresponding place value in Table 13-1 and compute the sum:

Bit 4 = .3125  
 Bit 5 = .015625  
 Bit 8 = .001953125  
 Bit 9 = .0009765625  
 Bit 12 = .0001220703125  
 Bit 13 = .00006103515625  
 Total = .04998779296875

Not exactly 0.05. One way to achieve more accuracy is to extend the number of places from sixteen to thirty-two bits. However, even more bits will be required to accurately express the reciprocal of a large number simply because the first several bits following the decimal point will be zeros. (In the case of the above example, there are only four zeros before we get to the first bit position with a one in it.)

The solution to this problem is to use scientific notation. For example, the decimal number 0.00001234 may be expressed as a number times a power of ten:

**Table 13-1. Place Values for Binary Fractions.**

Bit Position	Place Value
0	.5
1	.25
2	.125
3	.0625
4	.03125
5	.015625
6	.0078125
7	.00390625
8	.001953125
9	.0009765625
10	.00048828125
11	.000244140625
12	.0001220703125
13	.00006103515625
14	.000030517578125
15	.0000152587890625



$$1.234 \times 10^{-5}, \text{ or}$$

$$1234 \times 10^{-8}, \text{ or}$$

$$0.1234 \times 10^{-4}$$

In the same way, binary numbers may be expressed as a number times a power of two. The decimal number 0.05 in a binary may be expressed as

$$.CCCCCCCC \times 2^{FFFC}$$

In this form, three words of memory are required to store the number—two words for the fraction and one word for the exponent, which is stored as a signed number (FFFC equals  $-4$ ). It is not necessary to store the decimal point (or binary point, to be precise), the multiplication sign, or the number 2.

The greatest accuracy for a fixed number of bits (32 in this case) is achieved when the number is expressed as the largest possible fraction times a power of two. For example, the number  $1/20$  expressed as a binary 32-bit fraction

$$.00001100110011001100110011001100$$

This pattern of bits continues infinitely. Shifting the decimal point four positions to the right allows room in the 32-bit fraction for four more bits. The 4-bit shift right is equivalent to multiplying by  $2^4$ . To keep the number the same value the shifted fraction must be multiplied by  $2^{-4}$ .

$$.11001100110011001100110011001100 \times 2^{1111111111111100}, \text{ or}$$

$$.CCCCCCCC \times 2^{FFFC} \text{ hexadecimal}$$

This is the result generated by Program 13-4. Let's look at this program line-by-line to see how this is accomplished. 7D0C loads the candidate number, 20, into register 0. 7D10 loads the number 1, the number to be divided into, into register 1. 7D14-7D16 clears registers 3 and 4, which will be used as a temporary storage for the 32-bit fraction as it is derived. 7D18 sets register 5 equal to 16, the initial value of the exponent. Thus the initial value of the result is  $.00000000 \times 2^{0010}$  hexadecimal. The exponent will be decremented during the J1 loop based on the results of successive division operations. This will be explained shortly.

7D1C loads the number 16 into register 7, which will be used

as a loop counter by the J2 loop. The J2 loop determines the value of the last sixteen bits of the fraction. 7D20 loads the address where the final results will be stored.

The J1 loop (7D24-7D32) determines the value of the first sixteen bits of the fraction and the value of the exponent. 7D24 clears register 1, the first sixteen bits of the dividend. 7D26 divides 0014 into 00000001 (20 into 1, decimal) on the first pass. The quotient is zero, and the remainder is 0001. This is to be expected. 20 won't go into 1 a whole number of times.

For the moment, let's skip over the instructions at 7D28-7D2C. These instructions don't affect the results yet. 7D28 adds zero to zero, no jump occurs at 7D2A, and 7D2C shifts a zero value by one.

The next step is to multiply the remainder times 2 and try to divide 20 into 2. This is fair as long as the remainder is divided by 2, which can be done by decrementing the exponent value in register 5. The result so far is

$$.00000000 \times 2^{000F}$$

The second time through the J1 loop 0014 is divided into 00000002. The quotient is again zero. The remainder is 0002. Therefore, multiply by 2 and decrement the exponent. The result is

$$.00000000 \times 2^{000E}$$

The third time through the J1 loop 0014 is divided into 00000004. The quotient is again zero. The remainder is 0004. Therefore, multiply by 2 and decrement the exponent. The result is

$$.00000000 \times 2^{000D}$$

The fourth time through the J1 loop 0014 is divided into 00000008. The quotient is again zero. The remainder is 0008. Therefore multiply by 2 and decrement the exponent. The result is

$$.00000000 \times 2^{000C}$$

The fifth time through the J1 loop 0014 is divided into 00000010. The quotient is again zero. The remainder is 0010. Therefore multiply by 2 and decrement the exponent. The result is as follows:

$$.00000000 \times 2^{000B}$$

The sixth time through the J1 loop, 0014 is divided into 00000020. The quotient is 0001 and the remainder is 000C. The result is:

$$\begin{aligned} &.00010000 \times 2^{000B}, \text{ or} \\ &.0000152587890625 \times 2^{11} = .03125 \end{aligned}$$

To get more bits of accuracy, the division process must continue. I already know that 0014 won't divide into the remainder 000C. Therefore, I must multiply it by 2. (I have been doing this in the previous iterations, but the result was always zero.)

Now let's look at the instructions at 7D28-7D2C. 7D28 adds the quotient to register 3. Before I divide into the remainder I must add the quotient computed by 7D26 to previous results. 7D2A tests the quotient in register 3 to see if there is a 1 in bit position 0, the first bit position to the right of the decimal point. 7D2C multiplies the quotient in register 3 by 2. Each time through the J1 loop, the quotient in register 3 is multiplied by 2, the remainder is multiplied by 2 and the exponent is decremented. This process will continue until a 1 occurs in bit position 0. When this occurs, the largest possible 16-bit fraction has been computed. Then the J1 loop is exited, and the J2 loop is entered. The J2 loop will determine the last sixteen bits of the fraction.

Let's go through the J1 loop a few more times to see how the reciprocal is built bit-by-bit. To repeat, the sixth time through the J1 loop, 0014 is divided into 00000020. The quotient is 0001 and the remainder is 000C. The result after the instruction at 7D28 is executed as:

$$.00010000 \times 2^{000B}$$

7D2A tests to see if a 1 is in bit position 0. There is not, therefore 7D2C-7D2E multiplies the quotient in register 3 and the remainder in register 2 by 2 and decrements the exponent. The result is:

$$.00020000 \times 2^{000A}$$

Note that the value of the quotient has not changed, only the form.

The seventh time through the J1 loop, 0014 is divided into 00000018 (0000000C times 2). The quotient is 0001 and the re-

mainder is 0004. 0001 is added to 0002 in register 3. The reciprocal value so far is:

$$\begin{aligned} &.00030000 \times 2^{000A}, \text{ or} \\ &.0000457763671875 \times 2^{10} = .046875 \end{aligned}$$

The J1 loop is repeated until the result is:

$$.CCCC0000 \times 2^{FFFC} = .04999924$$

The J2 loop determines the last sixteen bits in a similar manner. Register 4 is used instead of register 3. The exponent was determined and, therefore, is not changed. Register 6 keeps track of the number of bits which remain to be determined.

7D44-7D48 moves the result to memory.

### Program 13-4

```

7D00 0014 M1 DATA 20
7D02 XXXX M2 BSS 6
7D08 02E0 LWPI >70B8
7D0A 70B8
7DOC C020 MOV @M1,R0
7D0E 7D00
7D10 0202 LI R2,1
7D12 0001
7D14 04C3 CLR R3
7D16 04C4 CLR R4
7D18 0205 LI R5,16
7D1A 0010
7D1C 0206 LI R6,16
7D1E 0010
7D20 0207 LI R7,M2
7D22 7D02
7D24 04C1 J1 CLR R1
7D26 3C40 DIV R0,R1
7D28 A0C1 A R1,R3
7D2A 1104 JLT J2

```

```

7D2C 0A13    SLA R3,1
7D2E 0A12    SLA R2,1
7D30 0605    DEC R5
7D32 10F8    JMP J1
7D34 0A12 J2  SLA R2,1

7D36 04C1    CLR R1
7D38 3C40    DIV R0,R1
7D3A A101    A R1,R4
7D3C 0606    DEC R6
7D3E 1302    JEQ J3
7D40 0A14    SLA R4,1
7D42 10F8    JMP J2
7D44 CDC3 J3  MOV R3,*R7+
7D46 CDC4    MOV R4,*R7+
7D48 C5C5    MOV R5,*R7
7D4A 045B    B *R11
7D4C XXXX    END

```

## SINE OF AN ANGLE

Program 13-5 finds the sine of an angle between 0 and 360 degrees. It does this by looking up the value in a table. Memory locations 7D4C-7E00 contain the sine values for angles 0 through 90. The numbers are stored as integers in order to save memory. For example, the sine of 1 degree is 0.0175. The hex equivalent for 175 is stored at 7D4E. To get the actual value, I divide by 10,000. This is done after the sine value is looked up.

Note that it is only necessary to store values of angles 0 through 90. The sine of an angle between 91 and 180 is determined by subtracting the angle from 180 and then looking up the value in the 0-90 table. The sine of 91 exactly equals the sine of 89.

The sines of angles between 181 and 270 are equivalent to the sines of 0-90 except that the *sign* is negative. The sines of angles 271-360 are equivalent to the sines of 91-180 except that the sign is negative.

Memory location 7E02 contains the number of the angle for which I want to find the sine. This number must be between 0 and

360 degrees. The example number is 45, or 002D in hex. Memory locations 7E04-7E0A are reserved for the result. 7E04 will contain the sign, 0000 for positive sine values, 0001 for negative sine values. 7E06-7E08 will contain the binary fraction. 7E0A will contain the value of the exponent.

The program has two main parts. 7E10-7E38 looks up the integer value in memory. 7E3A-7E70 divides the integer by 10,000 and stores the result in memory. The second part is identical to Program 13-5 (Reciprocal of a number) except that in Program 13-5 the dividend is a constant and the divisor is a variable. In Program 13-6 the divisor is a constant and the dividend is the variable. The loop up procedure begins at 7E1C. 7E1C clears register 3, which is temporarily used to store the sign of the sine. A positive sign is assumed until determined otherwise.

7E22 compares the angle value with 180. If greater than 180, then register 3 is incremented to 1 and 180 is subtracted from the angle. If the angle is less than 180, then the program jumps to 7E2A. 7E2A stores the sign in memory.

7E2C compares the angle with 90. If greater than 90, then the angle value is subtracted from 180. 7E32 moves the difference back to register 4. This is necessary because the subtraction instruction stores the result in register 1. The subtraction may or may not occur, depending on the value of the angle. If the subtraction is bypassed, then the angle is in register 4. If the subtraction is carried out, then the result must be moved to register 4.

7E34 doubles the value of the angle. This is necessary because indexed addressing is used to look up the sine and because each sine value (16-bit integer form) uses two bytes of memory.

7E36 moves the integer sine value to register 2. The sine of 45 is .7071. Thus, 1B9F (hex for 7071) is moved from 7DA0 to register 2. 7E3A-7E70 divides 1B9F by 2710 (10,000 decimal) to get

$$.B504816F \times 2^{0000}$$

or just .B504816F since  $2^{0000}$  equals 1. Of course, the decimal point, the times sign, and the 2 are not stored in memory. The sign is 0000, stored in memory earlier in the program.

### Program 13-5

```
7D00 XXXX    AORG >7D4C
```

```
7D4C 0000 M1 DATA 0,175,349,523,698,872
```

```
7D4E 00AF
```

7D50 015D  
7D52 020B  
7D54 02BA  
7D56 0368  
7D58 0415 DATA 1045,1219,1392,1564,1736  
7D5A 04C3  
7D5C 0570  
7D5E 061C  
7D60 06C8  
7D62 0774 DATA 1908,2079,2250,2419,2588  
7D64 081F  
7D66 08CA  
7D68 0973  
7D6A 0A1C  
7D6C 0AC4 DATA 2756,2924,3090,3256,3420  
7D6E 0B6C  
7D70 0C12  
7D72 0CB8  
7D74 0D5C  
7D76 0E00 DATA 3584,3746,3907,4067,4226  
7D78 0EA2  
7D7A 0F43  
7D7C 0FE3  
7D7E 1082  
7D80 1120 DATA 4384,4540,4695,4848,5000  
7D82 11BC  
7D84 1257  
7D86 12F0  
7D88 1388  
7D8A 141E DATA 5150,5299,5446,5592,5736  
7D8C 14B3  
7D8E 1546  
7D90 15D8

7D92 1668  
7D94 16F6 DATA 5878,6018,6157,6293,6428  
7D96 1782  
7D98 180D  
7D9A 1895  
7D9C 191C  
7D9E 19A1 DATA 6561,6691,6820,6947,7071  
7DA0 1A23  
7DA2 1AA4  
7DA4 1B23  
7DA6 1B9F  
7DAB 1C19 DATA 7193,7313,7431,7547,7660  
7DAA 1C91  
7DAC 1D07  
7DAE 1D7B  
7DB0 1DEC  
7DB2 1E5B DATA 7771,7880,7986,8090,8191  
7DB4 1EC8  
7DB6 1F32  
7DB8 1F9A  
7DBA 1FFF  
7DBC 2062 DATA 8290,8387,8480,8572,8660  
7DBE 20C3  
7DC0 2120  
7DC2 217C  
7DC4 21D4  
7DC6 222A DATA 8746,8829,8910,8988,9063  
7DC8 227D  
7DCA 22CE  
7DCC 231C  
7DCE 2367  
7DD0 23AF DATA 9135,9205,9272,9336,9397



7DD2 23F5  
 7DD4 2438  
 7DD6 2478  
 7DD8 24B5  
 7DDA 24EF DATA 9455,9511,9563,9613,9659  
 7DDC 2527  
 7DDE 255B  
 7DE0 258D  
 7DE2 25BB  
 7DE4 25E7 DATA 9703,9744,9781,9816,9848  
 7DE6 2610  
 7DE8 2635  
 7DEA 2658  
 7DEC 2678  
 7DEE 2695 DATA 9877,9903,9926,9945,9962  
 7DF0 26AF  
 7DF2 26C6  
 7DF4 26D9  
 7DF6 26EA  
 7DF8 26F8 DATA 9976,9986,9994,9998,10000  
 7DFA 2702  
 7DFC 270A  
 7DFE 270E  
 7E00 2710  
 7E02 002D M2 DATA 45  
 7E04 XXXX M3 BSS 8  
 7E0C 02E0 LWPI >70B8  
 7E0E 70B8  
 7E10 C120 MOV @M2,R4  
 7E12 7E02  
 7E14 0201 LI R1,180  
 7E16 00B4

7E18	0202	LI R2,90
7E1A	005A	
7E1C	04C3	CLR R3
7E1E	0207	LI R7,M3
7E20	7E04	
7E22	8044	C R4,R1
7E24	1202	JLE J1
7E26	0583	INC R3
7E28	6101	S R1,R4
7E2A	CDC3 J1	MOV R3,*R7+
7E2C	8084	C R4,R2
7E2E	1202	JLE J2
7E30	6044	S R4,R1
7E32	C101	MOV R1,R4
7E34	0A14 J2	SLA R4,1
7E36	COA4	MOV @M1(R4),R2
7E38	7D4C	
7E3A	0200	LI R0,10000
7E3C	2710	
7E3E	04C3	CLR R3
7E40	04C4	CLR R4
7E42	0205	LI R5,16
7E44	0010	
7E46	0206	LI R6,16
7E48	0010	
7E4A	04C1 J3	CLR R1
7E4C	3C40	DIV R0,R1
7E4E	A0C1	A R1,R3
7E50	1104	JLT J4
7E52	0A13	SLA R3,1
7E54	0A12	SLA R2,1
7E56	0605	DEC R5

```

7E58 10F8    JMP J3
7E5A 0A12 J4 SLA R2,1
7E5C 04C1    CLR R1
7E5E 3C40    DIV R0,R1
7E60 A101    A R1,R4
7E62 0606    DEC R6
7E64 1302    JEQ J5
7E66 0A14    SLA R4,1
7E68 10F8    JMP J4
7E6A CDC3 J5 MOV R3,*R7+
7E6C CDC4    MOV R4,*R7+
7E6E C5C5    MOV R5,*R7
7E70 045B    B *R11
7E72 XXXX    END

```

## Branch and Link

Program 13-6 performs the same function as Program 13-5 except that Program 13-6 uses the BL (Branch and Link) instruction.

The code at 7E3E-7E70 of Program 13-5 is used as a subroutine of Program 13-6. This code performs a generalized function (integer division with results in sign plus fraction plus exponent, or so-called *floating point* format) and, hence, is an excellent candidate for a subroutine.

The BL instruction is one of three instructions used to call subroutines: BL, branch and link, BLWP, branch and load workspace pointer, and XOP, extended operation. The BL is the simplest subroutine call. In Program 13-6, the instruction at 7EA6 branches to address M4 (7E3E) and saves the old program counter (PC) value (7EAA) in register 11.

Looking at Program 13-5, you can see that the instruction at 7E70 will cause the program to branch back to 7EAA. It is very important to note that before executing a BL instruction that the current contents of register 11 must be saved. Also, after executing the subroutine, the old contents of register 11 must be restored. In Program 13-6 this is done by the instructions at 7EA4 and 7EAA. If I was going to execute Program 13-6 once and once only, then instruction at 7EAA would not be necessary and I could end the

program with a B \*R10 instruction.

Recall that the assembler puts the address 609C in the Users' Workspace (WP = 70B8) register 11 for us. Assume that I did not restore the old PC value to register 11 and instead put B \*R10 at 7EAA. Then, after executing Program 13-6 from EASY BUG, address 609C would be in register 10, and 7EAA would be in register 11. The program would continue to execute 7EAA forever because 7EAA branches to the address in register 10—which is 7EAA! (Guess how I learned this?)

### Program 13-6

```
7D00 XXXX    AORG >7E72
7E72 XXXX M1 EQU >7D4C
7E72 XXXX M2 EQU >7E02
7E72 XXXX M3 EQU >7E04
7E72 XXXX M4 EQU >7E3E
7E72 02E0    LWPI >70B8
7E74 70B8
7E76 C120    MOV @M2,R4
7E78 7E02
7E7A 0201    LI R1,180
7E7C 00B4
7E7E 0202    LI R2,90
7E80 005A
7E82 04C3    CLR R3
7E84 0207    LI R7,M3
7E86 7E04
7E88 8044    C R4,R1
7E8A 1202    JLE J1
7E8C 0583    INC R3
7E8E 6101    S R1,R4
7E90 CDC3 J1 MOV R3,*R7+
7E92 8084    C R4,R2
7E94 1202    JLE J2
7E96 6044    S R4,P1
```

```

7E98 C101    MOV R1,R4
7E9A 0A14 J2 SLA R4,1
7E9C C0A4    MOV @M1(R4),R2
7E9E 7D4C
7EA0 0200    LI R0,10000
7EA2 2710
7EA4 C28B    MOV R11,R10
7EA6 06A0    BL @M4
7EA8 7E3E
7EAA C2CA    MOV R10,R11
7EAC 045B    B *R11
7EAE XXXX    END

```

## Branch and Load Workspace Pointer

Program 13-7 performs the sine look-up again, but this time using the BLWP subroutine call. In order to use BLWP, two things are necessary. First, the subroutine's WP and PC values must be stored somewhere in memory. 7EAE is used for the WP value (7092, the Utility Workspace), and 7EB0 is used for the PC value (7EB2), which is the starting address of the subroutine.

Second, the subroutine code needs to be modified (and, hence, rewritten and relocated) so that I can tell the subroutine the values of the divisor, the dividend, and the address where the result is to be stored. This is called *parameter passing*. This is not necessary when using the BL instruction which uses the same workspace (unless explicitly changed; then you will probably want to use BLWP).

The BLWP instruction saves the old WP, PC and ST (status) values in register 13, 14, 15, respectively. The instruction at 7EB2 moves the old R0 value to the new workspace R0. 7EB4 uses indexed addressing to move the old R2 value to the new workspace R2. The number 4 is added to 70B8 (stored in register 13) to get the memory address of the old workspace R2 because each register uses 2 bytes of memory. 7EB8 moves the old workspace R7 value (address 70B8 plus 000E) to the new workspace R7. It just so happens (for simplicity) that the register numbers stayed the same, but this is not necessary.

Also note that the subroutine ends with a RTWP (Return with

Workspace Pointer) instruction.

The third subroutine instruction, XOP, is not demonstrated in this book because the XOP WP and PC values are not stored in RAM. ROM addresses 0040-007E are designed for the XOP WP and PC values, or XOP software trap vectors, as TI calls them. Unless you plan to develop a microprocessor-based machine (both hardware and software), it is not necessary for you to learn how to use the XOP instruction.

### Program 13-7

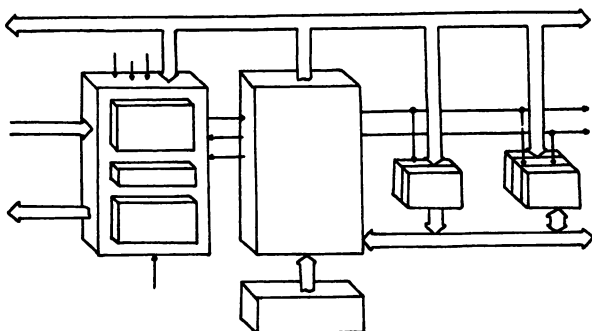
```
7D00 XXXX    AORG >7EAE
7EAE XXXX M1 EQU >7D4C
7EAE XXXX M2 EQU >7E02
7EAE XXXX M3 EQU >7E04
7EAE 7092 M4 DATA >7092
7EB0 7EB2    DATA >7EB2
7EB2 C01D    MOV *R13,R0
7EB4 C0AD    MOV @4(R13),R2
7EB6 0004
7EB8 C1ED    MOV @14(R13),R7
7EBA 000E
7EBC 04C3    CLR R3
7EBE 04C4    CLR R4
7EC0 0205    LI R5,16
7EC2 0010
7EC4 0206    LI R6,16
7EC6 0010
7EC8 04C1 J3 CLR R1
7ECA 3C40    DIV R0,R1
7ECC A0C1    A R1,R3
7ECE 1104    JLT J4
7ED0 0A13    SLA R3,1
7ED2 0A12    SLA R2,1
7ED4 0605    DEC R5
7ED6 10F8    JMP J3
```

7ED8 0A12 J4 SLA R2,1  
 7EDA 04C1 CLR R1  
 7EDC 3C40 DIV R0,R1  
 7EDE A101 A R1,R4  
 7EE0 0606 DEC R6  
 7EE2 1302 JEQ J5  
 7EE4 0A14 SLA R4,1  
 7EE6 10F8 JMP J4  
 7EE8 CDC3 J5 MOV R3,\*R7+  
 7EEA CDC4 MOV R4,\*R7+  
 7EEC C5C5 MOV R5,\*R7  
 7EEE 0380 RTWP  
 7EF0 02E0 LWPI >70B8  
 7EF2 70B8  
 7EF4 C120 MOV @M2,R4  
 7EF6 7E02  
 7EF8 0201 LI R1,180  
 7EFA 00B4  
 7EFC 0202 LI R2,90  
 7EFE 005A  
 7F00 04C3 CLR R3  
 7F02 0207 LI R7,M3  
 7F04 7E04  
 7F06 8044 C R4,R1  
 7F08 1202 JLE J1  
 7F0A 0583 INC R3  
 7F0C 6101 S R1,R4  
 7F0E CDC3 J1 MOV R3,\*R7+  
 7F10 8084 C R4,R2  
 7F12 1202 JLE J2  
 7F14 6044 S R4,R1  
 7F16 C101 MOV R1,R4

7F18 0A14 J2 SLA R4,1  
7F1A 0A4 MOV @M1(R4),R2  
7F1C 7D4C  
7F1E 0200 LI R0,10000  
7F20 2710  
7F22 0420 BLWP @M4  
7F24 7EAE  
7F26 045B B \*R11  
7F28 XXXX END



## Chapter 14



### Using the System Utilities

The primary aim of the preceding chapters was to teach you through the use of examples how to write assembly-language programs on the TI-99/4A. Not all of the 69 instructions were illustrated, but at this point you should be able to write programs in 9900 assembly language and be able to learn the remaining instructions (when needed) as well as read and understand both the *Mini Memory Owner's Manual* (which comes with the command module) and the *Editor/Assembler Manual* (which you may purchase directly from Texas Instruments).

The aim of this chapter is to show you how to use the *utilities* (built-in subroutines) described in pages 34-50 of the *Mini Memory Owner's Manual*. Seven programs will be listed and discussed. Program 14-1 will illustrate the VDP Single Byte Write routine which is accessed by the instruction BLWP@>6024. (Note that the *Mini Memory Owner's Manual* uses the 4-character mnemonic VSBW for the memory address 6024 and other 4-character mnemonics for other memory addresses. When using the Line-by-Line Assembler, however, you are restricted to using either 2-character address mnemonics or explicitly using the 4-digit hexadecimal address.)

Program 14-2 will illustrate the VDP Multiple Byte Write routine which is accessed by the instruction BLWP@>6028. Program 14-3 will use both of the above routines and show you how to generate a cursor. Program 14-4 will illustrate the Keyboard Scan

routine which is accessed by the instruction `BLWP@>6020`. Program 14-5 will show you how to access ROM-resident routines, the so-called XML routines, using the `BLWP@>601C` instruction. Program 14-6 will show you how to access GROM-resident routines (the so-called graphics programming language routines, or GPL Routines) using the `BLWP@>6018` instruction. This program will also show you how to name your program and execute it from the Mini Memory RUN option. Program 14-7, will show you how to change the screen color using the VDP Write to Register routine, accessed by the instruction `BLWP@>6034`.

## CLEARING THE SCREEN

Program 14-1 shows you how to clear the screen using the VDP Single Byte Write (VSBW) routine. Clearing the screen is such a common operation that I decided to make this program into a subroutine and use it in the programs that follow.

Memory locations 7D00 and 7D02 contain the subroutine workspace pointer and program counter values. The 32-byte memory block starting at 7FD0 is used as a workspace when executing the subroutine and address 7D04 is the address of the first instruction in the subroutine. The subroutine at 7D04-7D16 is called by the main program at 7D18 which consists of three instructions: load workspace pointer, call subroutine, and branch back to EASY BUG.

Note that the VSBW subroutine is called by the subroutine which clears the screen. This is called *nesting*. The only limitation on the number of nesting levels is the amount of RAM available for workspaces. In order to use the VSBW routine, I must place a VDP RAM address in register 0 and the 8-bit data in the upper byte half of register 1. The screen data is contained in VDP RAM address 0000-02FF, or 0-767 decimal. This corresponds to the 768 screen positions, each screen position itself being an 8 by 8 matrix.

The data to be written into the first 768 bytes of VDP RAM is the names, or codes, of characters or patterns defined elsewhere in the VDP RAM. For example, to clear the screen I want to write a 20 into VDP RAM locations 0000-02FF. 20 is the ASCII code for a blank, or a space. The pattern data for the space and all other keyboard characters are stored in VDP RAM locations 0800-0FFF when the console is first turned on. (Programs 14-3 and 14-6 will show you how to create, store and name other patterns.)

The instruction at 7D04 clears register 0 to 0000, which will be the first VDP RAM location that I will transfer a 20 to. 7D06 loads 20 into the upper byte half of register 1. 7D0A calls the VSBW

subroutine. 7D0E increments the VDP RAM address pointer. 7D10 compares the address to 0300 to see if all positions have been written to. 7D16 causes the computer to return to the calling program.

### **Program 14-1**

```
7D00 7FD0 M1 DATA >7FD0
7D02 7D04 DATA >7D04
7D04 04C0 CLR R0
7D06 0201 LI R1,>2000
7D08 2000
7D0A 0420 J1 BLWP @>6024
7D0C 6024
7D0E 0580 INC R0
7D10 0280 CI R0,>300
7D12 0300
7D14 16FA JNE J1
7D16 0380 RTWP
7D18 02E0 LWPI >70B8
7D1A 70B8
7D1C 0420 BLWP @M1
7D1E 7D00
7D20 045B B *R11
7D22 XXXX END
```

### **DISPLAY TEXT**

Program 14-2 uses the VDP Multiple Byte Write (VMBW) subroutine to display two strings of text on the screen at specific locations. To use the VMBW subroutine you must: load the VDP RAM destination address in register 0, load the source address of the text in register 1, and load the number of bytes (the number of characters, including blanks) to be transferred.

The only difficulty in using the VMBW routine is calculating the VDP RAM address. One way is to draw up a grid 32 squares across by 24 squares down and number them 0 through 767, starting with 0 in the upper lefthand corner and number across. The first row designates VDP RAM locations 0 through 31, the second row 32



Another way (Fig. 14-2) is to make a 32 by 24 grid and number the columns 1 through 32 across the top or bottom (not in the boxes) and the rows 1 through 24 down the left or right sides. Exact VDP RAM locations can be computed in terms of row and column as follows:

$$\text{VDP RAM ADDRESS} = 32(\text{ROW} - 1) + (\text{COLUMN} - 1)$$

With either method you will probably want to draw a 32 by 24 grid and make copies to use as work sheets. This will save programming time, although it is still very likely that you will make changes to your program in order to shift the text a few positions left or right, up or down, until you are satisfied with your display.

### Program 14-2

```

7D00 XXXX    AORG >7D22
7D22 5052 M1 TEXT 'PROGRAM 14-2'
7D24 4F47
7D26 5241
7D28 4D20
7D2A 3134
7D2C 2D32
7D2E 4449 M2 TEXT 'DISPLAY TEXT'
7D30 5350
7D32 4C41
7D34 5920
7D36 5445
7D38 5854
7D3A 02E0    LWPI >70B8
7D3C 70B8
7D3E 0420    BLWP @>7D00
7D40 7D00
7D42 0200    LI R0,362
7D44 016A
7D46 0201    LI R1,M1
7D48 7D22
7D4A 0202    LI R2,12

```

```

7D4C 000C
7D4E 0420    BLWP @>6028
7D50 6028
7D52 0200    LI R0,426
7D54 01AA
7D56 0201    LI R1,M2
7D58 7D2E
7D5A 0202    LI R2,12
7D5C 000C
7D5E 0420    BLWP @>6028
7D60 6028
7D62 045B    B *R11
7D64 XXXX    END

```

## GENERATE CURSOR

Program 14-3 generates a cursor similar to the one you see when you run BASIC. Since the cursor is not a standard ASCII character, it is necessary to create it and store it in the Pattern Generator Table in the VDP RAM.

Figure 14-3 shows an 8 by 8 matrix of boxes with X's in those boxes that define the cursor, a simple 5 by 7 rectangle. It takes 64 bits to define this pattern, a zero for each empty box and a one for each filled box, as follows:

```

00000000
01111100
01111100
01111100
01111100
01111100
01111100
01111100

```

In hexadecimal, the code for this pattern, starting at the top and reading across one line at a time, is

```
007C7C7C7C7C7C
```

It requires 8 bytes or 4 words of memory. The code at 7D70-7D80

	x	x	x	x	x		
	x	x	x	x	x		
	x	x	x	x	x		
	x	x	x	x	x		
	x	x	x	x	x		
	x	x	x	x	x		
	x	x	x	x	x		

Fig. 14-3. Cursor pattern.

stores this pattern in the Pattern Generator Table in the VDP RAM. The Pattern Generator Table starts at location 0800 in the VDP RAM. Character number 00 is stored in the 8 bytes 0800-0807, character number 01 is stored in 0808-080F, and so forth.

In creating a new pattern, the first step is to give the pattern a number. In this case, I decided to number the cursor 1E, or 30 in decimal (the same number used by TI BASIC). The next step is to calculate the VDP RAM location where the pattern will be stored. In hexadecimal the equation is as follows:

$$\text{VDP RAM ADDRESS} = 0800 + 08(\text{PATTERN NUMBER})$$

For the cursor, the VDP RAM address is

$$0800 + 08(1E) = 0800 + 00F0 = 08F0$$

If you prefer to calculate the VDP RAM address in decimal, use the following formula:

$$\text{VDP RAM ADDRESS} = 2048 + 8(\text{PATTERN NUMBER})$$

In Program 14-3, the instruction at 7D70 loads 08F0 into register 0, 7D74 loads 7D64 (the address of the pattern data in CPU RAM) into register 1, 7D78 loads the number of bytes (to be transferred) into register 2, and 7D7C calls the VMBW subroutine.

Note that 7D70-7D80 is itself a subroutine. The subroutine workspace pointer value is stored at 7D6C and the starting address is stored at 7D6E. This subroutine will also be called by Program 14-4 (keyboard input and display).

The cursor pattern subroutine is called by the instruction at 7D86 of the main program. 7D8A clears the screen. 7D8E loads VDP RAM address 0190 into register 0. This is approximately the center of the screen. This is where we will display the cursor, alternately turning it on and turning it off. 7D92 loads the character number 1E into register 1. 7D96 loads the number 0400 into register 2. This register will be used as a loop counter. 7D9A writes the

cursor pattern on the screen. 7D9E decrements the loop counter and 7DA0 causes the program to continue to write the cursor pattern on the screen until register 2 equals zero.

After writing the cursor pattern 0400 times (1024 decimal), the program exits that loop and enters another loop which writes a blank (character number 20 in hexadecimal) to the screen 0400 times. You may adjust the flash rate by changing the numbers at locations 7D98 and 7DA8. The present numbers cause the cursor to flash approximately 80 times per minute.

You also may move the cursor to another location by changing the number at 7D90. In the next program, the cursor will move across the screen automatically as you enter the text.

### **Program 14-3**

```
7D00 XXXX    AORG >7D64
7D64 007C M1  DATA >007C,>7C7C,>7C7C,>7C7C
7D66 7C7C
7D68 7C7C
7D6A 7C7C
7D6C 7FD0 M2  DATA >7FD0
7D6E 7D70    DATA >7D70
7D70 0200    LI R0,>08F0
7D72 08F0
7D74 0201    LI R1,M1
7D76 7D64
7D78 0202    LI R2,8
7D7A 0008
7D7C 0420    BLWP @ >6028
7D7E 6028
7D80 0380    RTWP
7D82 02E0    LWPI >70B8
7D84 70B8
7D86 0420    BLWP @M2
7D88 7D6C
7D8A 0420    BLWP @ >7D00
7D8C 7D00
```



```

7D8E 0200    LI R0,> 0190
7D90 0190
7D92 0201 J1 LI R1,> 1E00
7D94 1E00
7D96 0202    LI R2,> 0400
7D98 0400
7D9A 0420 J2 BLWP @>6024
7D9C 6024
7D9E 0602    DEC R2
7DA0 16FC    JNE J2
7DA2 0201    LI R1,> 2000
7DA4 2000
7DA6 0202    LI R2,>0400
7DA8 0400
7DAA 0420 J3 BLWP @> 6024
7DAC 6024
7DAE 0602    DEC R2
7DB0 16FC    JNE J3
7DB2 10EF    JMP J1
7DB4 XXXX    END

```

## KEYBOARD INPUT AND DISPLAY

Program 14-4 clears the screen, flashes the cursor in the upper lefthand corner of the screen, and waits for you to enter text via the keyboard. Each time you press a key, that key is displayed on the screen and then the flashing cursor moves over one position. Other features of the program are:

- ☐ The program stops when you press QUIT (FCTN and =).
- ☐ The FCTN and S combination causes the cursor to back up one space, thus erasing previous keystrokes.
- ☐ After the screen is full, an additional keystroke causes the program to start over. The screen is cleared and the program waits for more entries.

By itself, the program is not very useful but it does show you

how to enter text or data via the keyboard using the keyboard scan (KSCAN) routine. However, this program and the others in this book can be used as modules or subroutines in programs that you design yourself.

To use the KSCAN routine, first tell the computer the keyboard device number used by your program. In most cases, this will be 0. (See the *User's Reference Manual* for the meaning of other keyboard device numbers in the section which discusses the KEY subprogram.) This number must be stored at 8374 before you call KSCAN. In Program 14-4, the instruction at 7DB8 clears register 0 to 0000 and the instruction at 7DCC moves the byte 00 to memory location 8374.

The next step is to call KSCAN repeatedly until a key has been pressed. If a key has been pressed since the last call to KSCAN, then bit 2 of the GPL status byte will have been set to one. Note that the *Mini-Memory Owner's Manual* says bit 5. This is not correct. The manual incorrectly labels the bits from left to right as bits 7 through 0. This is not the standard TI designation. The *Editor/Assembler Manual* uses the standard TI bit numbering and is as follows:

H	GT	Cond	Carry	OVF	0	0	0
0	1	2	3	4	5	6	7

The GPL status byte is located at 837C. To determine if bit 2 equals a one, Program 14-4 uses the COC (compare ones corresponding) instruction at 7DEC. The upper byte half of register 3 contains the value 00100000 (binary). The upper byte half of register 6 contains a copy of the GPL status byte (moved there by the instruction at 7DE8). The COC instruction tests only those bits in register 6 for which there is a corresponding one in register 3. In this case, only bit 2 is tested. If bit 2 in register 6 equals a one (meaning that a key has been pressed since the last call to KSCAN), the EQU status register is set. (Don't confuse the status register inside the 9900 microprocessor with the GPL status byte at memory location 837C.)

In Program 14-4, the KSCAN routine is called approximately 2000 times per minute. The program alternately displays the cursor pattern and a blank as was shown in Program 14-3.

7DD8-7DDC loads the character number for the cursor and the loop counter value which determines how many times (and hence, how long) the cursor pattern will be displayed before switching to the blank.

7DE0-7DF2 is the combination display-cursor and call-KSCAN loop. The program stays in the loop either until the loop counter is zero or a key has been pressed. If a key has been pressed, then the program jumps to 7E12, where the program determines which key was pressed and decides what to do next. If a key is not pressed before the loop counter reaches zero, then when the counter is zero the program loads a blank, restores the loop counter, and then enters the combination display-blank and call-KSCAN loop at 7DFC-7E0E.

If no key is detected after 128 calls, then the program jumps back to the display-cursor and the call-KSCAN loop again. Once a key is detected, the program proceeds.

7E12-7E16 writes a blank to the screen. This is necessary for the case when the backspace key was detected after the cursor was displayed. I don't want to leave a black rectangle on the screen.

Next, 7E1A moves the ASCII byte value of the key to register 1 from memory location 8375 where the KSCAN routine stored the key value it detected. (8375 contains an FF if no key was pressed since the last KSCAN call.)

7E1E compares the value to 05, the TI code for the QUIT key. If the key equals 05, then the program ends.

7E22 compares the value to 08, the TI code for the backspace (FCTN and S) or left arrow (←). If the key is a backspace register 0 is decremented and the program jumps to 7DD8 and waits for another key input. Register 0 always contains the screen location where the cursor is flashing.

If the key is neither QUIT nor backspace, then the key is displayed by the instruction at 7E2A.

7E2E increments the screen location pointer (VDP RAM address). 7E30 compares the VDP RAM address with 0300 (the end of the screen). If the address equals 0300 then the program starts over, clearing the screen. If the address is less than 0300 then the program jumps to 7DD8, starts flashing the cursor and waits for the next key input.

#### **Program 14-4**

```
7D00 XXXX    AORG > 7DB4
7DB4 02E0    LWPI > 70B8
7DB6 70B8
7DB8 04C0 J1 CLR R0
7DBA 0202    LI R2, > 0500
```

7DBC 0500  
 7DBE 0203     LI R3,>2000  
 7DC0 2000  
 7DC2 0204     LI R4,>0300  
 7DC4 0300  
 7DC6 0205     LI R5,>0800  
 7DC8 0800  
 7DCA 04C6     CLR R6  
 7DCC D800     MOVB R0,@ >8374  
 7DCE 8374  
 7DD0 0420     BLWP @ > 7D00  
 7DD2 7D00  
 7DD4 0420     BLWP @ > 7D6C  
 7DD6 7D6C  
 7DD8 0201 J2 LI R1,> 1E00  
 7DDA 1E00  
 7DDC 0207     LI R7,>80  
 7DDE 0080  
 7DE0 0420 J3 BLWP @ >6024  
 7DE2 6024  
 7DE4 0420     BLWP @> 6020  
 7DE6 6020  
 7DE8 D1A0     MOVB @>837C,R6  
 7DEA 837C  
 7DEC 2183     COC R3,R6  
 7DEE 1311     JEQ J5  
 7DF0 0607     DEC R7  
 7DF2 16F6     JNE J3  
 7DF4 0201     LI R1,>2000  
 7DF6 2000  
 7DF8 0207     LI R7,>80  
 7DFA 0080  
 7DFC 0420 J4 BLWP @> 6024

```

7DFE 6024
7E00 0420    BLWP @>6020
7E02 6020
7E04 D1A0    MOVB @>837C,R6
7E06 837C
7E08 2183    COC R3,R6
7E0A 1303    JEQ J5
7E0C 0607    DEC R7
7E0E 16F6    JNE J4
7E10 10E3    JMP J2
7E12 0201 J5 LI R1,> 2000
7E14 2000
7E16 0420    BLWP @>6024
7E18 6024
7E1A D060    MOVB @>8375,R1
7E1C 8375
7E1E 9081    CB R1,R2
7E20 130A    JEQ J7
7E22 9141    CB R1,R5
7E24 1602    JNE J6
7E26 0600    DEC R0
7E28 10D7    JMP J2
7E2A 0420 J6 BLWP @>6024
7E2C 6024
7E2E 0580    INC R0
7E30 8100    C R0,R4
7E32 13C2    JEQ J1
7E34 10D1    JMP J2
7E36 045B J7 B *R11
7E38 XXXX    END

```

## **CONVERT STRING TO NUMBER**

Program 14-5 converts a number (entered via the keyboard) to

TI's 8-byte floating-point format and then displays the TI floating-point version as a 16-digit hexadecimal number. The aim of the program is to illustrate the use of ROM-resident routines which are accessed by the instruction BLWP@>601C followed by a 16-bit data value which references the desired ROM subroutine. In this case, the Convert String to Number routine is accessed. The data value to access this routine is 1000 hexadecimal.

Table 14-1 is a partial list of TI floating-point number equivalents. This table was compiled by running Program 14-5 for each entry in the table. TI's format is a variation on the standard binary-coded decimal (BCD) format. Instead of using one byte per decimal digit, TI uses one byte per two decimal digits. Also, the first byte indicates both the sign and the place value of the next byte. For example, the TI equivalent of 9012.3456789 is 415A0C22384E5A00. This may be broken down as follows:

- 41 - The next byte place value is 100
- 5A - 90 hundreds
- 0C - 12 ones
- 22 - 34 hundredths
- 38 - 56 ten thousandths
- 4E - 78 millionths
- 5A - 90 ten millionths

**Table 14-1. Partial List of TI Floating-Point Number Equivalents.**

Decimal Number	TI Hexadecimal Floating-Point Notation
100,000	420A000000000000
10,000	4201000000000000
1,000	410A000000000000
100	4101000000000000
10	400A000000000000
1	4001000000000000
.1	3F0A000000000000
.01	3F01000000000000
.001	3E0A000000000000
.0001	3E01000000000000
11.1111	400B0B0B00000000
-11.1111	BFF50B0B00000000
-1	BFFF000000000000
-100,000	BDF6000000000000
3.1415926	40030E0F5C3C0000
9,012.3456789	415A0C22384E5A00
100,000.00000001	420A000000000001

First Byte Value (Hex)	Next Byte Place Value (Decimal)
45	10,000,000,000
44	100,000,000
43	1,000,000
42	10,000
41	100
40	1
3F	.01
3E	.0001
3D	.000001
3C	.00000001
3B	.0000000001

**Table 14-2. Partial List of Next-Byte Place Values.**

Table 14-2 is a partial list of next-byte place values used by TI. It appears that TI chose 40 to indicate that the next byte is the ones digit because 40 is in the middle of the allowable range for positive 8-bit numbers. Negative numbers begin at 80 (hex). Thus, this format can represent the same number of digits to the left and right of the decimal point. Also, it appears from Table 14-1 that only the first two bytes of negative numbers are in two's complement form. See, for example, the TI equivalents of +11.1111 and -11.1111.

Now let's look at Program 14-5, which is long, but has a fairly simple structure. 7E38-7E80 is the storage area for the text displayed by the program. 7E82 loads the workspace pointer. 7E86 clears the screen. 7E8A-7EB6 displays the first three strings of text—PROGRAM 14-5, CONVERT STRING TO NUMBER, and INPUT STRING:

7EBA-7EFE displays the number as it is entered. This code is similar to the code of Program 14-4. The main difference is that there is no flashing cursor, which was intentionally left out to keep the program as simple as possible. The backspace feature, however, has been left in. The program will accept positive and negative numbers, both integers and floating-point numbers. Pressing the ENTER key (0D in ASCII) signals the program to go on to the conversion portion of the program. (By the way, don't let the term *floating-point number* scare you. The term applies to all nonintegers, or all decimal fractions in which the number of digits following the decimal point is not fixed. 1.3, 0.006, 1674.9, and 3.14159 are floating-point numbers.)

7F00-7F04 calls the Convert String to Number ROM subroutine. To use this subroutine, it is necessary to first store the address of the string at location 8356. This was done back at 7ED2.

The address of the string is the VDP RAM address corresponding to the position on the screen of the first digit (or sign, if a negative number was entered). In this case, the number to be converted is stored at VDP RAM location 432 (decimal), the number in register 0 when the input-text routine began. The 8-byte result is stored at 834A-8351.

7F06-7F12 displays the text **FLOATING-POINT NUMBER:** at screen location 514 (decimal). 7F16 loads the screen location 616 into register 0. 616 will be the location of the first digit of the 16-digit hexadecimal result when it is displayed on the screen. 7F1A-7F56 converts the 8-byte floating-point number located at 834A-8351 (the FAC, or floating-point accumulator) to ASCII and displays the result on the screen. The conversion and display is done four bits at a time.

A sample run looks like this:

**PROGRAM 14-5**  
**CONVERT STRING TO NUMBER**

INPUT STRING: 3.1415926  
FLOATING-POINT NUMBER:  
40030E0F5C3C0000

**Program 14-5**

```
7D00 XXXX      AORG  7E38
7E38 5052 M1 TEXT 'PROGRAM 14-5'
7E3A 4F47
7E3C 5241
7E3E 4D20
7E40 3134
7E42 2D35
7E44 434F M2 TEXT 'CONVERT STRING TO NUMBER'
7E46 4E56
7E48 4552
7E4A 5420
7E4C 5354
7E4E 5249
7E50 4E47
7E52 2054
```



7E54 4F20  
 7E56 4E55  
 7E58 4D42  
 7E5A 4552  
 7E5C 494E M3 TEXT 'INPUT STRING:  
 7E5E 5055  
 7E60 5420  
 7E62 5354  
 7E64 5249  
 7E66 4E47  
 7E68 3A20  
 7E6A 464C M4 TEXT 'FLOATING-POINT NUMBER:  
 7E6C 4F41  
 7E6E 5449  
 7E70 4E47  
 7E72 2D50  
 7E74 4F49  
 7E76 4E54  
 7E78 204E  
 7E7A 554D  
 7E7C 4245  
 7E7E 523A  
 7E80 2020  
 7E82 02E0      LWPI >70B8  
 7E84 70B8  
 7E86 0420      BLWP @>7D00  
 7E88 7D00  
 7E8A 0200      LI R0,138  
 7E8C 008A  
 7E8E 0201      LI R1,M1  
 7E90 7E38  
 7E92 0202      LI R2,12

7E94	000C	
7E96	0420	BLWP @ > 6028
7E98	6028	
7E9A	0200	LI R0,196
7E9C	00C4	
7E9E	0201	LI R1,M2
7EA0	7E44	
7EA2	0202	LI R2,24
7EA4	0018	
7EA6	0420	BLWP @ > 6028
7EA8	6028	
7EAA	0200	LI R0,418
7EAC	01A2	
7EAE	0201	LI R1,M3
7EB0	7E5C	
7EB2	0202	LI R2,14
7EB4	000E	
7EB6	0420	BLWP @ > 6028
7EB8	6028	
7EBA	04C0	CLR R0
7EBC	0202	LI R2,> 0D00
7EBE	0D00	
7EC0	0203	LI R3,> 2000
7EC2	2000	
7EC4	0205	LI R5,> 0800
7EC6	0800	
7EC8	04C6	CLR R6
7ECA	D800	MOVB R0,@ > 8374
7ECC	8374	
7ECE	0200	LI R0,432
7ED0	01B0	
7ED2	C800	MOV R0,@ > 8356
7ED4	8356	

```

7ED6 0420 J1 BLWP @>6020
7ED8 6020
7EDA D1A0      MOVB @>837C,R6
7EDC 837C
7EDE 2183      COC R3,R6
7EE0 16FA      JNE J1
7EE2 D060      MOVB @>8375,R1
7EE4 8375
7EE6 9081      CB R1,R2
7EE8 130B      JEQ J3
7EEA 9141      CB R1,R5
7EEC 1605      JNE J2
7EEE 0600      DEC R0
7EF0 C043      MOV R3,R1
7EF2 0420      BLWP @> 6024
7EF4 6024
7EF6 10EF      JMP J1
7EF8 0420 J2 BLWP @> 6024
7EFA 6024
7EFC 0580      INC R0
7EFE 10EB      JMP J1
7F00 0420 J3 BLWP @> 601C
7F02 601C
7F04 1000      DATA > 1000
7F06 0200      LI R0,514
7F08 0202
7FOA 0201      LI R1,M4
7FOC 7E6A
7FOE 0202      LI R2,24
7F10 0018
7F12 0420      BLWP @>6028
7F14 6028
7F16 0200      LI R0,616

```

7F18 0268  
 7F1A 0202   LI R2,> 834A  
 7F1C 834A  
 7F1E 0203   LI R3,8  
 7F20 0008  
 7F22 04C1 J4 CLR R1  
 7F24 D072   MOVB \*R2+,R1  
 7F26 0941   SRL R1,4  
 7F28 0281   CI R1,> 0A00  
 7F2A 0A00  
 7F2C 1A02   JL J5  
 7F2E 0221   AI R1,> 0700  
 7F30 0700  
 7F32 0221 J5 AI R1,> 3000  
 7F34 3000  
 7F36 0420   BLWP @> 6024  
 7F38 6024  
 7F3A 0580   INC R0  
 7F3C 0A81   SLA R1,8  
 7F3E 0941   SRL R1,4  
 7F40 0281   CI R1,> 0A00  
 7F42 0A00  
 7F44 1A02   JL J6  
 7F46 0221   AI R1,> 0700  
 7F48 0700  
 7F4A 0221 J6 AI R1,> 3000  
 7F4C 3000  
 7F4E 0420   BLWP @> 6024  
 7F50 6024  
 7F52 0580   INC R0  
 7F54 0603   DEC R3  
 7F56 16E5   JNE J4

7F58 045B     B \*R11

7F5A XXXX     END

## RAISE NUMBER TO A POWER

Program 14-6 raises a number to a power. The program prompts you to input two numbers—X and Y. The result is X raised to the Y power. A sample run looks like this:

INPUT X: 10

INPUT Y: 2

$X \uparrow Y = 100$

Also, this program is different from all previous programs in that it is executed from Mini Memory, not from EASY BUG.

The program illustrates the use of the GROM-resident routines, or the GPL routines. A breakdown of the program goes like this. 7D00-7D16 is the code for the subroutine that clears the screen. 7D18-7D64 is the code for the subroutine that inputs data from the keyboard and converts that data to the TI 8-byte floating-point format. The instruction at 7D1C moves the contents of the old workspace register 0 to the new workspace register 0. The number in register 0 is the VDP RAM address of the string (string, because the number is ASCII-encoded as it is entered) to be converted to TI floating-point, the form required by resident math routines. Also, 7D5A moves the VDP RAM address to location 8356. This is necessary before calling the Convert String to Number routine.

7D66-7D6C is the code data for the exponentiation symbol placed between the X and the Y in the display. See Fig. 14-4. 7D6E-7D82 is the code for the subroutine which stores the exponentiation symbol in the VDP RAM Pattern Generator Table. 0C00-0C08 is the VDP RAM location for character 128 (decimal),

Fig. 14-4. Exponentiation pattern.

			X				
		X	X	X			
	X		X		X		
			X				
			X				
			X				
			X				

the first code number available after the standard ASCII codes. (Note that any number from 128 to 255 is valid.) 7D84-7D98 contains the text data to be displayed. 7D9A loads the workspace pointer. 7D9E clears the screen. 7DA2 loads the exponentiation symbol into the VDP RAM. 7DA6-7DB4 displays the text INPUT X: 7DB6-7DBC inputs X and converts it to the 8-byte floating-point number which is stored at 834A-8351.

7DBE-7DCA moves the floating-point number to VDP RAM locations 1000-1007. This is done for two reasons. First, I need to input Y. If I don't move X from 834A-8351 (CPU RAM), it will be lost when I input Y. Second, the exponentiation routine (or Involution Routine, page 43, *Mini Memory Owner's Manual*) requires that the base number (the number to be raised to a power) be located somewhere in VDP RAM. Later, I will put the VDP RAM address of the base number at location 836E. This must be done before the exponentiation routine is called.

7DCE-7DDC displays the text INPUT Y:. 7DDE-7DE4 inputs Y and converts it to the 8-byte floating-point number which is stored at 834A-8351. Note that the exponent value must be at this location before the exponentiation routine is called. Thus, there is no need to move this number.

7DE6-7DF4 displays  $X Y =$ . 7DF6-7E00 puts the exponentiation symbol between the X and the Y of the previously displayed text.

7E02-7E08 moves the VDP RAM address (1000, hexadecimal) of the base number to location 836E.

7E0A-7E0E clears the GPL status byte at 837C. This step is absolutely necessary. The GPL status byte must be cleared before calling GROM-resident routines. Although this step is clearly shown in the *Editor/Assembler* manual, it is not shown in the *Mini Memory Owner's Manual*.

7E10-7E14 calls the exponentiation routine. The result is in the FAC (floating-point accumulator, 834A-8351).

7E16 clears location 8355. This step is done in preparation to use the Convert Number to String routine, another GPL routine. (See page 42 of the *Mini Memory Owner's Manual*.) When 8355 is set to zero, the output of this routine will be in TI BASIC format. 7E1A clears the GPL status byte. 7E1E-7E22 calls the Convert Number to String routine. The ASCII-encoded output string is located in CPU RAM at address 8300 plus the byte value stored at 8355. Location 8356 contains the length of the string.

7E24-7E3A displays the result on the screen. 7E28-7E30

computes the string address. 7E32-7E36 retrieves the string length and shifts it from the upper byte position to the lower byte position of register 2. 7E38 calls the VMBW routine.

7E3C-7E50 maintains the display until any key is pressed. 7E52-7E56 clears the GPL status byte. This procedural step is required, otherwise when you branch back to the calling program (Mini Memory), you will get a meaningless error message. (Thank you, *Editor/Assembler* manual.)

7FE8-7FED contains the name of the program—any name that you want to give to it, up to six characters—and the starting address. In this case, I named the program PWR. The starting address is 7D9A. 7FE8-7FED previously contained the name and address of the LINES program, which I have since overwritten many times.

To run this program, select the RUN option after you have selected MINI MEMORY from the master selection list. PROGRAM NAME? will be displayed. Type PWR and press ENTER. The screen will be cleared and the program will display

**INPUT X:**

in the upper left hand corner of the screen in black text on a light green background.

Type the base number 10 and press ENTER. The screen will display

**INPUT Y:**

just below the INPUT X: 10. Type 2 and press ENTER. The screen will display

**X↑Y = 100**

Nothing further will happen until you press any key. When you press any key, the computer will display PRESS ENTER TO CONTINUE in white text on a dark blue background.

Press ENTER. The screen will display the Mini Memory selection list. You may quit or rerun the program by selecting RUN and then entering PWR again.

Note that this program (and any other program using GPL routines) will not execute from EASY BUG. I have notified TI of this anomaly, but I have not received a reply as to why and/or what to do

so that programs using GPL routines may be run from EASY BUG. This is probably not, however, a serious problem. After you learn TI-99/4A assembly language, it is more likely that you will run your programs from Mini Memory than from EASY BUG anyway. Or you will create assembly language routines that you will link to your BASIC language programs.

### Program 14-6

```

7D00 7FC0    DATA >7FC0
7D02 7D04    DATA >7D04
7D04 04C0    CLR R0
7D06 0201    LI R1,>2000
7D08 2000
7D0A 0420 J1 BLWP @>6024
7D0C 6024
7D0E 0580    INC R0
7D10 0280    CI R0,>300
7D12 0300
7D14 16FA    JNE J1
7D16 0380    RTWP
7D18 7FC0    DATA >7FC0
7D1A 7D1C    DATA >7D1C
7D1C C01D    MOV *R13,R0
7D1E 0202    LI R2,>0D00
7D20 0D00
7D22 0203    LI R3,>2000
7D24 2000
7D26 0205    LI R5,>0800
7D28 0800
7D2A 04C6    CLR R6
7D2C D806    MOVB R6,@>8374
7D2E 8374
7D30 0420 J2 BLWP @>6020
7D32 6020

```



7D34 D1A0	MOVB @>837C,R6
7D36 837C	
7D38 2183	COC R3,R6
7D3A 16FA	JNE J2
7D3C D060	MOVB @>8375,R1
7D3E 8375	
7D40 9081	CB R1,R2
7D42 130B	JEQ J4
7D44 9141	CB R1,R5
7D46 1605	JNE J3
7D48 0600	DEC R0
7D4A C043	MOV R3,R1
7D4C 0420	BLWP @> 6024
7D4E 6024	
7D50 10EF	JMP J2
7D52 0420 J3	BLWP @> 6024
7D54 6024	
7D56 0580	INC R0
7D58 10EB	JMP J2
7D5A C81D J4	MOV *R13,@> 8356
7D5C 8356	
7D5E 0420	BLWP @> 601C
7D60 601C	
7D62 1000	DATA > 1000
7D64 0380	RTWP
7D66 0010	DATA >0010, > 3854,>1010,> 1010
7D68 3854	
7D6A 1010	
7D6C 1010	
7D6E 7FC0	DATA > 7FC0
7D70 7D72	DATA > 7D72
7D72 0200	LI R0,> 0C00
7D74 0C00	

7D76 0201	LI R1,>7D66
7D78 7D66	
7D7A 0202	LI R2,8
7D7C 0008	
7D7E 0420	BLWP @>6028
7D80 6028	
7D82 0380	RTWP
7D84 494E	TEXT 'INPUT X:'
7D86 5055	
7D88 5420	
7D8A 583A	
7D8C 494E	TEXT 'INPUT Y:'
7D8E 5055	
7D90 5420	
7D92 593A	
7D94 5820	TEXT 'X Y = '
7D96 5920	
7D98 3D20	
7D9A 02E0	LWPI >70B8
7D9C 70B8	
7D9E 0420	BLWP @>7D00
7DA0 7D00	
7DA2 0420	BLWP @>7D6E
7DA4 7D6E	
7DA6 0200	LI R0,34
7DA8 0022	
7DAA 0201	LI R1,>7D84
7DAC 7D84	
7DAE 0202	LI R2,8
7DB0 0008	
7DB2 0420	BLWP @>6028
7DB4 6028	

7DB6 0200	LI R0,43
7DB8 002B	
7DBA 0420	BLWP @> 7D18
7DBC 7D18	
7DBE 0200	LI R0,>1000
7DC0 1000	
7DC2 0201	LI R1,> 834A
7DC4 834A	
7DC6 0202	LI R2,8
7DC8 0008	
7DCA 0420	BLWP @> 6028
7DCC 6028	
7DCE 0200	LI R0,66
7DD0 0042	
7DD2 0201	LI R1,> 7D8C
7DD4 7D8C	
7DD6 0202	LI R2,8
7DD8 0008	
7DDA 0420	BLWP @> 6028
7DDC 6028	
7DDE 0200	LI R0,75
7DE0 004B	
7DE2 0420	BLWP @> 7D18
7DE4 7D18	
7DE6 0200	LI R0,130
7DE8 0082	
7DEA 0201	LI R1,> 7D94
7DEC 7D94	
7DEE 0202	LI R2,5
7DF0 0005	
7DF2 0420	BLWP @> 6028
7DF4 6028	

7DF6 0200	LI R0,131
7DF8 0083	
7DFA 0201	LI R1,> 8000
7DFC 8000	
7DFE 0420	BLWP @>6024
7E00 6024	
7E02 0200	LI R0,> 1000
7E04 1000	
7E06 C800	MOV R0,@> 836E
7E08 836E	
7E0A 04C1	CLR R1
7E0C D801	MOVB R1,@> 837C
7E0E 837C	
7E10 0420	BLWP @>6018
7E12 6018	
7E14 0024	DATA >24
7E16 D801	MOVB R1,@>8355
7E18 8355	
7E1A D801	MOVB R1,@>837C
7E1C 837C	
7E1E 0420	BLWP @>6018
7E20 6018	
7E22 0014	DATA >14
7E24 0200	LI R0,137
7E26 0089	
7E28 D060	MOVB @> 8355,R1
7E2A 8355	
7E2C 0981	SRL R1,8
7E2E 0221	AI R1,> 8300
7E30 8300	
7E32 D0A0	MOVB @> 8356,R2
7E34 8356	

```

7E36 0982    SRL R2,8
7E38 0420    BLWP @> 6028
7E3A 6028
7E3C 0203    LI R3,> 2000
7E3E 2000
7E40 04C6    CLR R6
7E42 D806    MOVB R6,@> 8374
7E44 8374
7E46 0420 J5 BLWP @> 6020
7E48 6020
7E4A D1A0    MOVB @> 837C,R6
7E4C 837C
7E4E 2183    COC R3,R6
7E50 16FA    JNE J5
7E52 04C0    CLR R0
7E54 D800    MOVB R0,@> 837C
7E56 837C
7E58 045B    B *R11
7E5A XXXX    AORG > 7FE8
7FE8 5057    TEXT 'PWR
7FEA 5220
7FEC 2020
7FEE 7D9A    DATA > 7D9A
7F00 XXXX    END

```

## CHANGE SCREEN COLOR

I personally don't care too much for a light green (or is it medium green?) screen background color. Consequently, I wrote Program 14-7, which essentially changes the screen color, then branches to Program 14-6. The text color is also changed and the result is white text on a dark blue background, the same color combination used by the Line-by-Line Assembler.

In order to change the background color (and text, the foreground), I modified the Color Table in the VDP RAM. The Color

Table starts at VDP RAM address 0380 (hex). Each entry in the table specifies the background and foreground colors of a group of eight characters. The entry at 0380 specifies the colors of characters 00 through 07, the entry at 0381 specifies the colors of characters 08 through 0F, and so forth.

There are 32 groups of 8 characters. To make sure that the background and foreground colors of all characters are changed, Program 14-7 changes all 32 entries in the Color Table.

The color codes are listed in Table 14-3. To change the entry in the Color Table, I must specify both the foreground and background codes. White characters on a dark blue background is specified as F4 in the Color Table.

The instructions at 7E5E-7E72 write an F4 to all 32 entries in the Color Table. This alone will change only the 32 by 24 central rectangular area of the screen. There is still some background area between this rectangle and the edge of your TV screen.

To change this border area, I must write a set of color codes to one of the eight write-only registers inside the VDP chip. These write-only registers are described on pages 326-328 of the *Editor/Assembler* manual. They are not described in the *Mini Memory Owner's Manual*.

Bits 0-3 of VDP register 7 contain the foreground color in the text mode. (I have been using the graphics mode, the default mode. See the *Editor/Assembler* manual for an explanation of the different modes.) Bits 4-7 of VDP register 7 contain the color code of the background color in all modes. These are the bits we want to change.

Color	Hexadecimal Code
Transparent	0
Black	1
Medium green	2
Light green	3
Dark blue	4
Light blue	5
Dark red	6
Cyan	7
Medium red	8
Light red	9
Dark yellow	A
Light yellow	B
Dark green	C
Magenta	D
Gray	E
White	F

**Table 14-3.**  
**Color Codes.**

To change the data in VDP register 7, Program 14-7 uses the VDP Write to Register (VWTR) routine mentioned on page 36 of the *Mini Memory Owner's Manual*.

The instruction at 7E74 loads the value 07F4 into register 0. The 07 corresponds to register 7, the VDP write-only register that I want to change. The F4 corresponds to the foreground and background color codes. 7E78 calls the VWTR routine. 7E7C branches to the beginning of the PWR program.

Finally, the address at 7FEE is changed to 7E5A so that the program can be run from Mini Memory.

### **Program 14-7**

```
7D00 XXXX    AORG >7E5A
7E5A 02E0    LWPI >70B8
7E5C 70B8
7E5E 0200    LI R0,>0380
7E60 0380
7E62 0201    LI R1,>F400
7E64 F400
7E66 0202    LI R2,32
7E68 0020
7E6A 0420 J1 BLWP @>6024
7E6C 6024
7E6E 0580    INC R0
7E70 0602    DEC R2
7E72 16FB    JNE J1
7E74 0200    LI R0,>07F4
7E76 07F4
7E78 0420    BLWP @>6034
7E7A 6034
7E7C 0460    B @>7D9E
7E7E 7D9E
7E80 XXXX    AORG >7FEE
7FEE 7E5A    DATA >7E5A
7F00 XXXX    END
```

## CONCLUSION

If you have faithfully entered all the programs and carefully read and understood the program descriptions, you should be ready to do some serious assembly language programming on your TI-99/4A Home Computer.

At this point, I recommend that you buy the *Editor/Assembler* manual if you haven't already done so, especially if you plan to write assembly language programs involving, sound, color, graphics, and moving graphics called *sprites*. It would take another book to illustrate these advanced features of the TI Home Computer.





# **Appendix**

## **TMS9900 Instruction Set**

---

(Courtesy of Texas Instruments Incorporated)

## ASSEMBLY LANGUAGE PROGRAMMING INFORMATION †

In order to understand the instruction descriptions and applications the assembly language nomenclature must be understood. Assembly language is a readily understood language in which the 9900 instructions can be written. The machine code that results from the assembly of programs written in this language is called object code. Such object code may be absolute or relocatable, depending on the assembly language coding. Relocatable code is that which can be loaded into any block of memory desired, without reassembling or without changing program operation. Such code has its address information relative to the first instruction of the assembly language program so that once a loader program specifies the location of this first instruction, the address of all instructions are adjusted to be consistent with this location. Absolute code contains absolute addresses which cannot be changed by the loader or any operation other than reassembling the program. Generally, relocatable code is preferable since it allows the program modules to be located anywhere in memory of the final system.

### ASSEMBLY LANGUAGE FORMATS

The general assembly language source statements consist of four fields as follows:

LABEL	MNEMONIC	OPERANDS	COMMENT
-------	----------	----------	---------

The first three fields must occur within the first 60 character positions of the source record. At least one blank must be inserted between fields.

#### Label Field

The label consists of from one to six characters, beginning with an alphabetic character in character position one of the source record. The label field is terminated by at least one blank. When the assembler encounters a label in an instruction it assigns the current value of the location counter to the label symbol. This is the value associated with the label symbol and is the address of the instruction in memory. If a label is not used, character position 1 may be a blank, or an asterisk.

#### Mnemonic or Opcode Field

This field contains the mnemonic code of one of the instructions, one of the assembly language directives, or a symbol representing one of the program defined operations. This field begins after the last blank following the label field. Examples of instruction mnemonics include A for addition and MOV for data movement. The mnemonic field is required since it identifies which operation is to be performed.

#### Operands Field

The operands specify the memory locations of the data to be used by the instruction. This field begins following the last blank that follows the mnemonic field. The memory locations can be specified by using constants, symbols, or expressions, to describe one of several addressing modes available. These are summarized in *Figure 6-5*.

†Excerpts from Model 9900 computer TMS 9900 Microprocessor Assembly Language Programmer's Guide.

<i>Type of Addressing</i>	<i>Operand Format</i>	<i>Memory Location Specified</i>	<i>MOV Instruction Example Coding</i>	<i>Result</i>	<i>T<sub>d</sub> or T<sub>s</sub> Field Code</i>
Workspace Register	n	Workspace Register n R <sub>n</sub>	MOV 3,5	R3 → R5	00
Workspace Register Indirect	*n	Address given by the contents of workspace register n M(R <sub>n</sub> )	MOV *3,*5	M(R3) → M(R5)	01
Workspace Register Indirect, Autoincrement	*n +	As in register Indirect; address register R <sub>n</sub> is incremented after the operation (by one for byte operations, by two for word operations)	MOV *3+,*5 +	M(R3) → M(R5) R3 + 2 → R3 R5 + 2 → R5	11

Symbolic Memory	@exp	Address is given by value of exp. $M(\text{exp})$	MOV	@ONE, @10	$M(\text{ONE}) \longrightarrow M(10)$	10
Indexed Memory	@exp(n)	Address is the sum of the contents of Rn and the value of exp $M(Rn + \text{exp})$	MOV	@2(3), @DP(5)	$M(R3 + 2) \longrightarrow M(R5 + DP)$	10

**Notes:**

n is the number of the workspace register:  $0 \leq n \leq 15$ ; n may not be 0 for indexed addressing.

exp is a symbol, number, or expression

The  $T_s$  and  $T_d$  fields are two bit portions of the instruction machine code. There are also S and D four bit fields, which are filled in with the four bit code for n. n is 0 for symbolic or direct addressing.

**Figure 6-5. Addressing Modes**

### Comments Field

Comments can be entered after the last blank that follows the operands field. If the first character position of the source statement contains an asterisk (\*), the entire source statement is a comment. Comments are listed in the source portion of the assembler listing, but have no affect on the object code.

### TERMS AND SYMBOLS

Symbols are used in the label field, the operator field, and the operand field. A symbol is a string of alphanumeric characters, beginning with an alphabetic character.

Terms are used in the operand fields of instructions and assembler directives. A term is a decimal or hexadecimal constant, an absolute assembly-time constant, or a label having an absolute value. Expressions can also be used in the operand fields of instructions and assembler directives.

### Constants

Constants can be decimal integers (written as a string of numerals) in the range of -32,768 to +65,535. For example:

257

Constants can also be hexadecimal integers (a string of hexadecimal digits preceded by

>). For example:

> 09AF

ASCII character constants can be used by enclosing the desired character string in single quotes. For example:

'DX' = 4458<sub>16</sub>      'R' + 0052<sub>16</sub>

Throughout this book the subscript 16 is used to denote base 16 numbers. For example, the hexadecimal number 09AF will be written 09AF<sub>16</sub>.

### Symbols

Symbols must begin with an alphabetic character and contain no blanks. Only the first six characters of a symbol are processed by the assembler.

The assembler predefines the dollar sign (\$) to represent the current location in the program.

A given symbol can be used as a label only once, since it is the symbolic name of the address of the instruction. Symbols defined with the DXOP directive are used in the OPCODE field. Any symbol in the OPERANDS field must have been used as a label or defined by a REF directive.



Expressions

Expressions are used in the OPERANDS fields of assembly language statements. An expression is a term or a series of terms separated by the following arithmetic operations:

- + addition
- subtraction
- \* multiplication
- / division

The operator precedence is +, –, \*, / (left to right).

The expression must not contain any imbedded blanks or extended operation defined (DXOP directive defined) symbols. Unary minus (a minus sign in front of a number or symbol) is performed first and then the expression is evaluated from left to right. An example of the use of the unary minus in an expression is:

LABEL + TABLE + ( – INC)

which has the effect of the expression:

LABEL + TABLE – INC

The relocatability of an expression is a function of the relocatability of the symbols and constants that make up the expression. An expression is relocatable when the number of relocatable symbols or constants added to the expression is one greater than the number of relocatable symbols or constants subtracted from the expressions. All other expressions are absolute. The expression given earlier would be relocatable if the three symbols in the expression are all relocatable.

The following are examples of valid expressions.

BLUE + 1

2\*16 + RED

440/2 -- RED

#### SURVEY OF THE 9900 INSTRUCTION SET

The 9900 instructions can be grouped into the following general categories: data transfer, arithmetic, comparison, logical, shift, branch, and CRU input/output operations. The list of all instructions and their effect on status bits is given in *Figure 6-6*.

Mnemonic	L>	A>	EQ	C	OV	OP	X	Mnemonic	L>	A>	EQ	C	OV	OP	X
A	X	X	X	X	X	-	-	DIV	-	-	-	-	X	-	-
AB	X	X	X	X	X	X	-	IDLE	-	-	-	-	-	-	-
ABS	X	X	X	X	X	-	-	INC	X	X	X	X	X	-	-
AI	X	X	X	X	X	-	-	INCT	X	X	X	X	X	-	-
ANDI	X	X	X	-	-	-	-	INV	X	X	X	-	-	-	-
B	-	-	-	-	-	-	-	JEQ	-	-	-	-	-	-	-
BL	-	-	-	-	-	-	-	JGT	-	-	-	-	-	-	-
BLWP	-	-	-	-	-	-	-	JH	-	-	-	-	-	-	-
C	X	X	X	-	-	-	-	JHE	-	-	-	-	-	-	-
CB	X	X	X	-	X	-	-	JL	-	-	-	-	-	-	-
CI	X	X	X	-	-	-	-	JLE	-	-	-	-	-	-	-
CKOF	-	-	-	-	-	-	-	JLT	-	-	-	-	-	-	-
CKON	-	-	-	-	-	-	-	JMP	-	-	-	-	-	-	-
CLR	-	-	-	-	-	-	-	JNC	-	-	-	-	-	-	-
COC	-	-	X	-	-	-	-	JNE	-	-	-	-	-	-	-
CZC	-	-	X	-	-	-	-	JNO	-	-	-	-	-	-	-
DEC	X	X	X	X	X	-	-	JOC	-	-	-	-	-	-	-
DECT	X	X	X	X	X	-	-	JOP	-	-	-	-	-	-	-
LDCR	X	X	X	-	-	1	-	SBZ	-	-	-	-	-	-	-
LI	X	X	X	-	-	-	-	SETO	-	-	-	-	-	-	-



### Data Transfer Instructions

*Load*— used to initialize processor or workspace registers to a desired value.

*Move*— used to move words or bytes from one memory location to another.

*Store*— used to store the status or workspace pointer registers in a workspace register.

### Arithmetic Instructions

*Addition and Subtraction*—perform addition or subtraction of signed or unsigned binary words or bytes stored in memory.

*Negate and Absolute Value*—changes the sign or takes the absolute value of data words in memory.

*Increment and Decrement*—Adds or subtracts 1 or 2 from the specified data words in memory.

*Multiply*—Performs unsigned integer multiplication of a word in memory with a workspace register word to form a 32 bit product stored in two successive workspace register locations.

*Divide*—Divides a 32 bit unsigned integer dividend (contained in two successive workspace registers) by a memory word with the 16 bit quotient and 16 bit remainder stored in place of the dividend.

#### Compare Instructions

These instructions provide for masked or unmasked comparison of one memory word or byte to another or a workspace register word to a 16 bit constant.

#### Logical Instructions

*OR and AND*—masked or unmasked OR and AND operations on corresponding bits of two memory words. A workspace register word can be ORed or ANDed with a 16 bit constant.

*Complement and Clear* — The bits of a selected memory word can be complemented, or cleared or set to ones.

*Exclusive OR*—A workspace register word can be exclusive ORed with another memory word on a bit by bit basis.

*Set Bits Corresponding*—Set bits to one (SOC) or to zero (SZC) whose positions correspond to one positions in a reference word.

### Shift Instructions

A workspace register can be shifted arithmetically or logically to the right. The registers can be shifted to the left (filling in vacated positions with zeroes) or circulated to the right. The shifts and circulates can be from 1 to 16 bit positions.

### Branch Instructions

The branch instructions and the JMP (jump) instruction unconditionally branch to different parts of the program memory. If a branch occurs, the PC register will be changed to the value specified by the operand of the branch instruction. In subroutine branching the old value of the PC is saved when the branch occurs and then is restored when the return instruction is executed. The conditional jump instructions test certain status bits to determine if jump is to occur. When a jump is made the PC is loaded with the sum of its previous value and a displacement value specified in the operand portion of the instruction.

### Control/CRU Instructions

These instructions provide for transferring data to and from the communications register input/output unit (CRU) using the CRUIN, CRUOUT and CRUCLK pins of the 9900.

### INSTRUCTION DESCRIPTIONS

The information provided for each instruction in the next section of this chapter is as follows:

Name of the instruction.

Mnemonic for the instruction.

Assembly language and machine code formats.

Description of the operation of the instruction.

Effect of the instruction on the Status Bits.

Examples.

Applications.

The format descriptions and examples are written without the label or comment fields for simplicity. Labels and comments fields can be used in any instruction if desired.



Each instruction involves one or two operand fields which are written with the following symbols:

G—Any addressing mode is permitted except I (Immediate).

R—Workspace register addressing.

exp—A symbol or expression used to indicate a location.

value—a value to be used in immediate addressing.

cnt—A count value for shifts and CRU instructions.

CRU—CRU (Communications Register Unit) bit addressing.

The instruction operation is described in written and equation form. In the equation form, an arrow( $\longrightarrow$ ) is used to indicate a transfer of data and a colon ( $:$ ) is used to indicate a comparison. In comparisons, the operands are not changed. In transfers, the source operand (indicated with the subscript s) is not changed while the destination operand (indicated with the subscript d) is changed. For operands specified by the symbol G, the M(G) nomenclature is used to denote the memory word specified by G. MB(G) is used to denote the memory byte specified by G. Thus, transferring the memory word contents addressed by  $G_s$  to the memory word location specified by  $G_d$  and comparing the source ( $G_s$ ) data to zero during the transfer, can be described as:

$$M(G_s) \longrightarrow M(G_d)$$

$$M(G_s):0$$

which is the operation performed by the MOV instruction:

$$\text{MOV} \quad G_s, G_d$$

A specific example of this instruction could be:

$$\text{MOV} \quad @ONE, 3$$

which moves the contents of the memory word addressed by the value of the symbol ONE to the contents of workspace register 3:

$$M(ONE) \longrightarrow R3$$

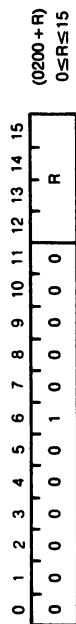
$$M(ONE) : 0$$

## DATA TRANSFER INSTRUCTIONS

The MOV instructions are used to transfer data from one part of the system to another part. The LOAD instructions are used to initialize registers to desired values. The STORE instructions provide for saving the status register (ST) or the workspace pointer (WP) in a specified workspace register.

### LOAD IMMEDIATE

**Format: LI R,value**



**Operation:** The 16 bit data value in the word immediately following the instruction is loaded into the specified workspace register R.

value  $\longrightarrow$  R

immediate operand: 0

**Affect on Status: LGT,AGT,EQ**

**Examples:**    **LI    7,5                      5    $\longrightarrow$  R7**

**LI    8,>FF                      00FF<sub>16</sub>  $\longrightarrow$  R8**

# LI

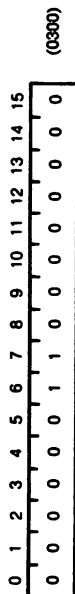
*Applications:* The LI instruction is used to initialize a workspace register with a program constant such as a counter value or data mask.

---

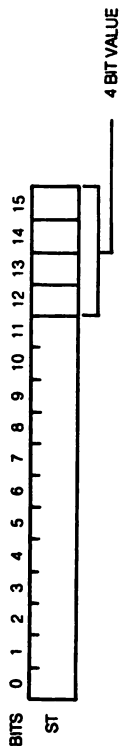
LOAD INTERRUPT MASK IMMEDIATE

**LIMI**

*Format:* **LIMI**      **value**



*Operation* The low order 4 bit value (bits 12-15) in the word immediately following the instruction is loaded into the interrupt mask portion of the status register:



*Affect on Status:* Interrupt mask code only

*Example:* **LIMI 5**

Enables interrupt levels 0 through 5

*Application:* The LIMi instruction is used to initialize the interrupt mask to control which system interrupts will be recognized.

# LWPI

## LOAD WORKSPACE POINTER IMMEDIATE

*Format:* **LWPI value**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	0

(02E0)

*Operation:* The 16 bit value contained in the word immediately following the instruction is loaded into the workspace pointer (WP):

value  $\longrightarrow$  WP

*Affect on Status:* None

*Example:* **LWPI > 0500**

Causes 0500<sub>16</sub> to be loaded into the WP.

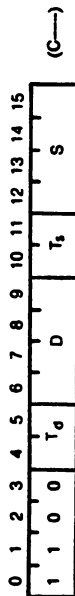
*Application:* LWPI is used to establish the workspace memory area for a section of the program.

---

## MOVE WORD

# MOV

**Format: MOV G<sub>a</sub>,G<sub>d</sub>**



**Operation:** The word in the location specified by G<sub>s</sub> is transferred to the location specified by G<sub>a</sub>, without affecting the data stored in the G<sub>s</sub> location. During the transfer, the word (G<sub>s</sub> data) is compared to 0 with the result of the comparison stored in the status register:

$$M(G_s) \longrightarrow M(G_a)$$

$$M(G_s):0$$

**Status Bits Affected: LGT, AGT, and EQ**

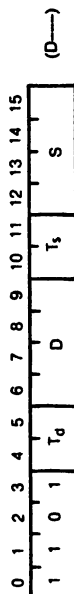
**Examples:**

<b>MOV</b>	<b>R1,R3</b>	<b>R1</b>	<b>→ R3,</b>	<b>R1:0</b>
<b>MOV</b>	<b>*R1,R3</b>	<b>M(R1)</b>	<b>→ R3,</b>	<b>M(R1):0</b>
<b>MOV</b>	<b>@ONES,*1</b>	<b>M(ONES)</b>	<b>→ M(R1),</b>	<b>M(ONES):0</b>
<b>MOV</b>	<b>@2(5),3</b>	<b>M(R5 + 2)</b>	<b>→ R3,</b>	<b>M(R5 + 2):0</b>
<b>MOV</b>	<b>*R1 +,*R2 +</b>	<b>M(R1)</b>	<b>→ M(R2),</b>	<b>M(R1):0,</b>
		<b>(R1) + 2</b>	<b>→ R1,</b>	<b>(R2) + 2</b>
				<b>→ R2</b>

**Application:** MOV is used to transfer data from one part of the system to another part.

# MOVE BYTE

**Format:** **MOV B G<sub>s</sub>G<sub>d</sub>**



**Operation:** The Byte addressed by G<sub>s</sub> is transferred to the byte location specified by G<sub>d</sub>. If G is workspace register addressing, the most significant byte is selected. Otherwise, even addresses select the most significant byte; odd addresses select the least significant byte. During the transfer, the source byte is compared to zero and the results of the comparison are stored in the status register.

MB(G<sub>s</sub>) → MB(G<sub>d</sub>)  
 MB(G<sub>s</sub>):0

**Status Bits Affected:** **LGT, AGT, EQ, OP**

# MOV B

*Examples:*    **MOVB @>1C14,3**  
                  **MOVB \*8,4**

These instructions would have the following example affects:

<i>Memory Location</i>	<i>Contents</i>	
	<i>Initially</i>	<i>After Transfer</i>
<b>1C14</b>	<b>2016</b>	<b>2016</b>
<b>R3</b>	<b><u>542B</u></b>	<b><u>202B</u></b>
<b>R8</b>	<b>2123</b>	<b>2123</b>
<b>2123</b>	<b><u>1040</u></b>	<b><u>1040</u></b>
<b>R4</b>	<b><u>0A0C</u></b>	<b><u>400C</u></b>

The underlined data are the bytes selected.

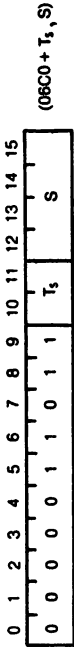
*Application:* MOVB is used to transfer 8 bit bytes from one byte location to another.



SWAP BYTES

**SWPB**

**Format: SWPB G**



**Operation:** The most significant byte and the least significant bytes of the word at the memory location specified by G are exchanged.

**Affect on Status:** None

	Before	After
--	--------	-------

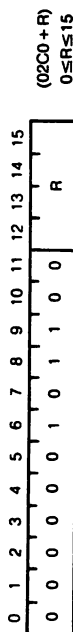
**Example: SWPB 3** R3 Contents: F302 02F3

**Application:** Used to interchange bytes if needed for subsequent byte operations.

---

## STORE STATUS

*Format:* **STST R**



*Operation:* The contents of the status register are stored in the workspace register specified:

ST → R

*Affect on Status:* None

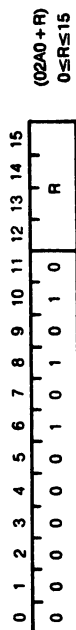
*Example:* **STST 3** ST is transferred to R3

*Application:* STST is used to save the status for later reference.

# STST

STORE WORKSPACE POINTER

STWP

*Format:* **STWP R**

*Operation:* The contents of the workspace pointer are stored in the workspace register specified:

WP → R

*Affect on Status:* None

*Example:* **STWP 3** WP is transferred into R3

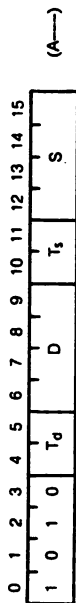
*Application:* STWP is used to save the workspace pointer for later reference.

## ARITHMETIC INSTRUCTIONS

These instructions perform the following basic arithmetic operations: addition (byte or word), subtraction (byte or word), multiplication, division, negation, and absolute value. More complicated mathematical functions must be developed using these basic operations. The basic instruction set will be adequate for many system requirements.

### ADD WORDS

**Format:** A       $G_s, G_d$



**Operation:** The data located at the address specified by  $G_s$  is added to the data located at the address specified by  $G_d$ . The resulting sum is placed in the  $G_d$  location and is compared to zero:

$$\begin{aligned} M(G_s) + M(G_d) &\longrightarrow M(G_d) \\ M(G_s) + M(G_d) &: 0 \end{aligned}$$

**Status Bits Affected:** LGT, AGT, EQ, C, OV

A

*Examples:*      **A**      **5,@TABLE**     $R5 + M(TABLE) \longrightarrow M(TABLE)$   
                   **A**      **3,\*2**         $R3 + M(R2) \longrightarrow M(R2)$

with the sums compared to 0 in each case. Binary addition affects on status bits can be understood by studying the following examples:

$M(G_d)$	$M(G_d)$	Sum	LGT	AGT**	EQ	C	OV*
1000	0001	1001	1	1	0	0	0
F000	1000	0000	0	0	1	1	0
F000	8000	7000	1	1	0	1	1
4000	4000	8000	1	0	0	0	1

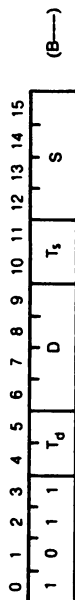
\*OV (overflow) is set if the most significant bit of the sum is different from the most significant bit of  $M(G_d)$  and the most significant bit of both operands are equal.

\*\*AGT (arithmetic greater than) is set if the most significant bit of the sum is zero and if EQ (equal) is 0.

*Application:* Binary addition is the basic arithmetic operation required to generate many mathematical functions. This instruction can be used to develop programs to do multiword addition, decimal addition, code conversion, and so on.

### ADD BYTES

**Format:** AB    **G<sub>s</sub>**, **G<sub>d</sub>**



AB

**Operation:** The source byte addressed by  $G_s$  is added to the destination byte addressed by  $G_d$  and the sum byte is placed in the  $G_d$  byte location. Recall that even addresses select the most significant byte and odd addresses select the least significant byte. The sum byte is compared to 0.

$$\begin{aligned} MB(G_s) + MB(G_d) &\longrightarrow MB(G_d) \\ MB(G_s) + MB(G_d) &: 0 \end{aligned}$$

**Status Bits Affected:** LGT, AGT, EQ, C, OV, OP

*Example:* AB 3,\*4+ R3 + MB(R4) → MB(R4), R4 + 2 → R4  
AB @TAB,5 MB(TAB) + R5 → R5

To see how the AB works, the following example should be studied:  
AB @>2120,@>2123

<i>Memory Location</i>	<i>Data Before Addition</i>	<i>Data After Addition</i>
2120	F320	F320
2123	<u>2106</u>	<u>21F9</u>

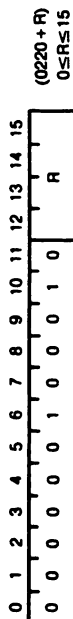
The underlined entries are the addressed and changed bytes.

*Application:* AB is one of the byte operations available on the 9900. These can be useful when dealing with subsystems or data that use 8 bit units, such as ASCII codes.

# AI

## ADD IMMEDIATE

**Format:** AI      R, Value



**Operation:** The 16 bit value contained in the word immediately following the instruction is added to the contents of the workspace register specified.

R + Value → R,      R + Value:0

**Status Bits Affected:** LGT, AGT, EQ, C, OV

**Example:** AI      6, >C

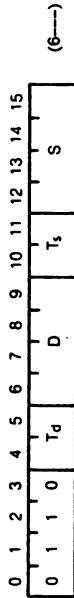
Adds C<sub>16</sub> to the contents of workspace register 6. If R6 contains 1000<sub>16</sub>, then the instruction will change its contents to 100C<sub>16</sub>, and the LGT and AGT status bits will be set.

**Application:** This instruction is used to add a constant to a workspace register. Such an operation is useful for adding a constant displacement to an address contained in the workspace register.



SUBTRACT WORDS

Format: S G<sub>s</sub>G<sub>d</sub>



Operation: The source 16 bit data (location specified by G<sub>s</sub>) is subtracted from the destination data (location specified by G<sub>d</sub>) with the result placed in the destination location G<sub>d</sub>. The result is compared to 0.

$$\begin{aligned} M(G_s) - M(G_s) &\longrightarrow M(G_d) \\ M(G_d) - M(G_s) &: 0 \end{aligned}$$

Status Bits Affected: LGT, AGT, EQ, C, OV

Examples: S @OLDVAL,@NEWVAL  
would yield the following example results:

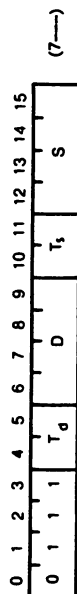
Memory Location	Before Subtraction Contents	After Subtraction Contents
OLDVAL	1225	1225
NEWVAL	8223	6FFE (8223-1225)

All status bits affected would be set to 1 except equal which would be reset to 0.

Application: Provides 16 bit binary subtraction.

# SUBTRACT BYTES

**Format:** **SB**      **G<sub>s</sub>G<sub>d</sub>**



**Operation:** The source byte addressed by G<sub>s</sub> is subtracted from the destination byte addressed by G<sub>d</sub> with the result placed in byte location G<sub>d</sub>. The result is compared to 0. Even addresses select the most significant byte and odd addresses select the least significant byte. If workspace register addressing is used, the most significant byte of the register is used.

MB(G<sub>d</sub>) - MB(G<sub>s</sub>) → MB(G<sub>s</sub>)  
 MB(G<sub>d</sub>) - MB(G<sub>s</sub>):0

**Status Bits Affected:** **LGT, AGT, C, EQ, OV, OP**

**Format:** **SB**      \*6+,1      R1 - MB(R6) → R1  
    R1 - MB(R6):0  
    R6 + 1 → R6

**SB**

This operation would have the following example result:

<i>Memory Location</i>	<i>Contents Before Instruction</i>	<i>Contents After Instruction</i>
<b>R6</b>	<b>121D</b>	<b>121E</b>
<b>121D</b>	<b>3123</b>	<b>4123</b>
<b>R1</b>	<b><u>1344</u></b>	<b><u>F044</u></b>

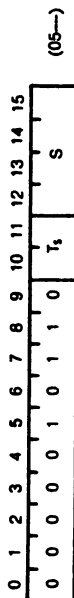
The underlined entries indicated the addressed and changed bytes. The LGT (logical greater than) status bit would be set to 1 while the other status bits affected would be 0.

*Application:* SB provides byte subtraction when 8 bit operations are required by the system.

## INCREMENT

# INC

**Format: INC G**



**Operation:** The data located at the address indicated by G is incremented and the result is placed in the G location and compared to 0.

$$M(G) + 1 \longrightarrow M(G)$$

$$M(G) + 1 : 0$$

**Status Bits Affected: LGT, AGT, EQ, C, OV**

**Examples:** **INC @TABL**  $M(TABL) + 1 \longrightarrow M(TABL)$   
**INC 1**  $(R1) + 1 \longrightarrow R1$

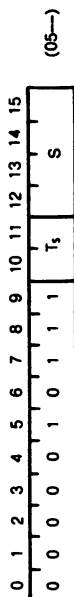
**Application:** INC is used to increment byte addresses and to increment byte counters. Autoincrementing addressing on byte instructions automatically includes this operation.

---

## INCREMENT BY TWO

# INCT

**Format:** INCT G



**Operation:** Two is added to the data at the location specified by G and the result is stored at the G location and is compared to 0:

$$M(G) + 2 \longrightarrow M(G)$$

$$M(G) + 2 : 0$$

**Status Bits Affected:** LGT, AGT, EQ, C, OV

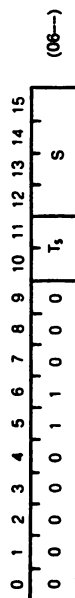
**Example:** INCT 5 (R5) + 2  $\longrightarrow$  R5

**Application:** This can be used to increment word addresses, though autoincrementing on word instructions does this automatically.

## DECREMENT

# DEC

*Format:* **DEC G**



*Operation:* One is subtracted from the data at the location specified by G, the result is stored at that location and is compared to 0:

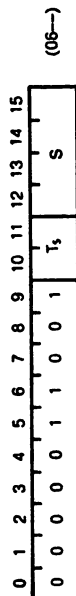
$$\begin{aligned} M(G) - 1 &\longrightarrow M(G) \\ M(G) - 1 : 0 \end{aligned}$$

*Status Bits Affected:* **LGT, AGT, EQ, C, OV**

*Example:* **DEC @TABL**      $M(TABL) - 1 \longrightarrow M(TABL)$

*Application:* This instruction is most often used to decrement byte counters or to work through byte addresses in descending order.

---

DECREMENT BY TWO**Format: DECT G**

**Operation:** Two is subtracted from the data at the location specified by G and the result is stored at that location and is compared to 0:

$$M(G) - 2 \longrightarrow M(G)$$

$$M(G) - 2 : 0$$

**Status Bits Affected:** LGT, AGT, EQ, C, OV

**Example:** DECT 3 (R3) - 2 → R3

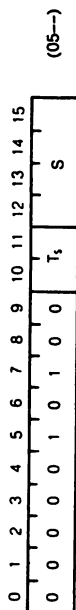
**Application:** This instruction is used to decrement word counters and to work through word addresses in descending order.

DECT

## NEGATE

*Format:*    **NEG    G**

**NEG**



*Operation:* The data at the address specified by G is replaced by its two's complement.  
The result is compared to 0:

— M(G)  $\longrightarrow$  M(G)  
— M(G) : 0

*Status Bits Affected:*    **LGT, AGT, EQ, OV** (OV set only when operand = 8000<sub>16</sub>)

*Example:*    **NEG    5**    — (R5)  $\longrightarrow$  R5

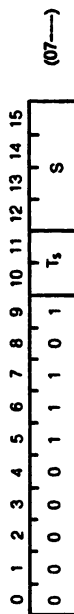
If R5 contained A342<sub>16</sub>, this instruction would cause the R5 contents to be changed to 5CBE<sub>16</sub> and will cause the LGT and AGT status bits to be set to 1.

*Application:* NEG is used to form the 2's complement of 16 bit numbers.



ABSOLUTE VALUE

## ABS

**Format: ABS G**

**Operation:** The data at the address specified by G is compared to 0. Then the absolute value of this data is placed in the G location:

$$\begin{array}{l} M(G) : 0 \\ |M(G)| \longrightarrow M(G) \end{array}$$

**Status Bits Affected:** **LGT, AGT, EQ, OV** (OV set only when operand = 8000<sub>16</sub>)

**Example:** **ABS @LIST(7)**  $|M(R7 + LIST)| \longrightarrow M(R7 + LIST)$

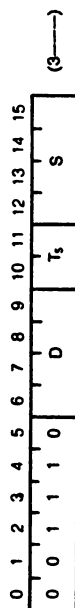
If the data at R7 + LIST is FF3C<sub>16</sub>, it will be changed to 00C4<sub>16</sub> and LGT will be set to 1.

**Application:** This instruction is used to test the data in location G and then replace the data by its absolute value. This could be used for unsigned arithmetic algorithms such as multiplication.

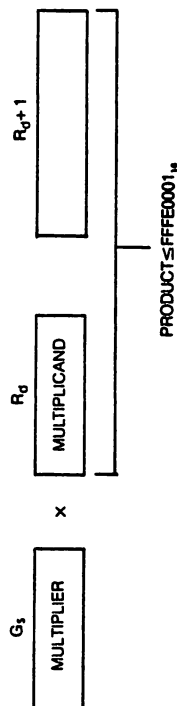
# MULTIPLY

## MPY

**Format:** MPY     $G_s, R_d$



**Operation:** The 16 bit data at the address designated by  $G_s$  is multiplied by the 16 bit data contained in the specified workspace register  $R$ . The unsigned binary product (32 bits) is placed in workspace registers  $R$  and  $R + 1$ :



*Affect on Status:* None

*Example:* **MPY @NEW,5**

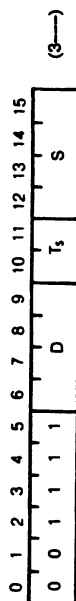
If the data at location NEW is 0005<sub>16</sub> and R5 contains 0012<sub>16</sub>, this instruction will cause R5 to contain 0000<sub>16</sub> and R6 to contain 005A<sub>16</sub>.

*Application:* MPY can be used to perform 16 bit by 16 bit binary multiplication. Several such 32 bit subproducts can be combined in such a way to perform multiplication involving larger multipliers and multiplicands such as a 32 bit by 32 bit multiplication.

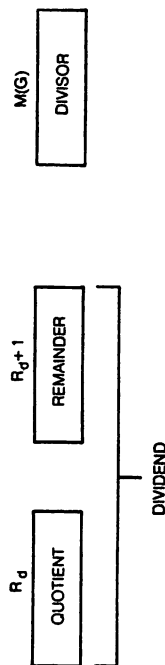
# DIVIDE

# DIV

**Format:** **DIV** **G<sub>s</sub>**, **R<sub>d</sub>**



**Operation:** The 32 bit number contained in workspace registers  $R_d$  and  $R_d + 1$  is divided by the 16 bit data contained at the address specified by  $G_s$ . The workspace register  $R_d$  then contains the quotient and workspace  $R_d + 1$  contains the 16 bit remainder. The division will occur only if the divisor at  $G$  is greater than the data contained in  $R_d$ :



*Affect on Status:* Overflow (OV) is set if the divisor is less than the data contained in  $R_d$ . If OV is set,  $R_d$  and  $R_d + 1$  are not changed.

**Example: DIV @LOC,2**

If R2 contains 0 and R3 contains 000D<sub>16</sub> and the data at address LOC is 0005<sub>16</sub>, this instruction will cause R2 to contain 0002<sub>16</sub> and R3 to contain 0003<sub>16</sub>. OV would be 0.

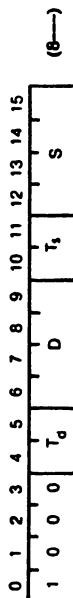
*Application:* DIV provides basic binary division of a 32 bit number by a 16 bit number.

## COMPARISON INSTRUCTIONS

These instructions are used to test words or bytes by comparing them with a reference constant or with another word or byte. Such operations are used in certain types of division algorithms, number conversion, and in recognition of input command or limit conditions.

### COMPARE WORDS

**Format:** C      G<sub>s</sub>,G<sub>d</sub>



**Operation:** The 2's complement 16 bit data addressed by G<sub>s</sub> is compared to the 2's complement 16 bit data addressed by G<sub>d</sub>. The contents of both locations remain unchanged.

M(G<sub>s</sub>) : M(G<sub>d</sub>)

**Status Bits Affected:** LGT, AGT, EQ

C

*Example:*    **C        @T1,2**

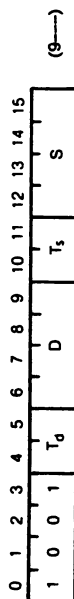
This instruction has the following example results:

<i>Data at</i>		<i>R2</i>	<i>Results of Comparison</i>		
<i>Location</i>	<i>T1</i>		<i>LGT</i>	<i>AGT</i>	<i>EQ</i>
	<b>FFFF</b>	<b>0000</b>	<b>1</b>	<b>0</b>	<b>0</b>
	<b>7FFF</b>	<b>0000</b>	<b>1</b>	<b>1</b>	<b>0</b>
	<b>8000</b>	<b>0000</b>	<b>1</b>	<b>0</b>	<b>0</b>
	<b>8000</b>	<b>7FFF</b>	<b>1</b>	<b>0</b>	<b>0</b>
	<b>7FFF</b>	<b>7FFF</b>	<b>0</b>	<b>0</b>	<b>1</b>
	<b>7FFF</b>	<b>8000</b>	<b>0</b>	<b>1</b>	<b>0</b>

*Application:* The need to compare two words occurs in such system functions as division, number conversion, and pattern recognition.

## COMPARE BYTES

*Format:*    **CB**    **G<sub>s</sub>**, **G<sub>d</sub>**



*Operation:* The 2's complement 8 bit byte addressed by G<sub>s</sub> is compared to the 2's complement 8 bit byte addressed by G<sub>d</sub>:

MB(G<sub>s</sub>) : MB(G<sub>d</sub>)

*Status Bits Affected:*    **LGT, AGT, EQ, OP**

OP (odd parity) is based on the number of bits in the source byte.

*Example:*    **CB**    **1,\*2**

with the typical results of (assuming R2 addresses an odd byte):

**CB**



<i>RI data</i>	<i>M(R2) Data</i>	<i>LGT</i>	<i>Results of Comparison</i>		
			<i>AGT</i>	<i>EQ</i>	<i>OP</i>
<u>FFFF</u>	<u>FF00</u>	1	0	0	0
<u>7F00</u>	<u>FF00</u>	1	1	0	1
<u>8000</u>	<u>FF00</u>	1	0	0	1
<u>8000</u>	<u>FF7F</u>	1	0	0	1
<u>7F00</u>	<u>007F</u>	0	0	1	1

The underlined entries indicate the byte addressed.

*Application:* In cases where 8 bit operations are required, CB provides a means of performing byte comparisons for special conversion and recognition problems.

## COMPARE IMMEDIATE

# CI

**Format: CI R,Value**



**Operation:** CI compares the specified workspace register contents to the value contained word immediately following the instruction:

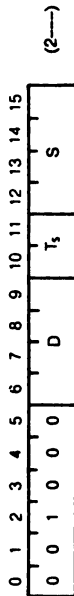
R : Value

**Status Bits Affected: LGT, AGT, EQ**

**Example: CI 9,>F330**

If R9 contains 2183<sub>16</sub>, the equal (EQ) and logical greater than (LGT) bits will be 0 and arithmetic greater than (AGT) will be set to 1.

**Application:** CI is used to test data to see if system or program limits have been met or exceeded or to recognize command words.

COMPARE ONES CORRESPONDING**COC****Format: COC G<sub>s</sub>,R**

**Operation:** The data in the location addressed by G<sub>s</sub> act as a mask for the bits to be tested in workspace register R. That is, only the bit position that contain ones in the G<sub>s</sub> data will be checked in R. Then, if R contains ones in all the bit positions selected by the G<sub>s</sub> data, the equal (EQ) status bit will be set to 1.

**Status Bits Affected: EQ**

**Example: COC @TESTBIT, 8**

If R8 contains E306<sub>16</sub> and location TESTBIT contains C102<sub>16</sub>,

**TESTBIT Mask = 1100 0001 0000 0010**

**R8 = 1110 0011 0000 0110**

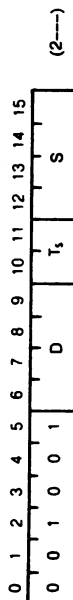
equal (EQ) would be set to 1 since everywhere the test mask data contains a 1 (underlined positions), R8 also contains a 1.

**Application:** COC is used to selectively test groups of bits to check the status of certain sub-systems or to examine certain aspects of data words.

## COMPARE ZEROES CORRESPONDING

# CZC

**Format:** CZC G<sub>s</sub>,R



**Operation:** The data located in the address specified by G<sub>s</sub> act as a mask for the bits to be tested in the specified workspace register R. That is, only the bit positions that contain ones in the G<sub>s</sub> data are the bit positions to be checked in R. Then if R contains zeroes in all the selected bit positions, the equal (EQ) status bit will be set to 1.

**Status Bits Affected:** EQ

**Examples:** CZC @TESTBIT,8

If the TESTBIT location contains the value C102<sub>16</sub> and the R8 location contains 2301<sub>16</sub>,

**TESTBIT Data = 1100 0001 0000 0010**  
**R8 = 0010 0011 0000 0001**  
X

the equal status bit would be reset to zero since not all the bits of R8 (note the X position) are zero in the positions that the TESTBIT data contains ones.

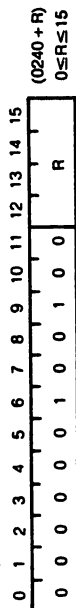
**Application:** Similar to the COC instruction.

## LOGIC INSTRUCTIONS

The logic instructions allow the processor to perform boolean logic for the system. Since AND, OR, INVERT, and Exclusive OR (XOR) are available, any boolean function can be performed on system data.

### AND IMMEDIATE

**Format: ANDI R, Value**



## ANDI

**Operation:** The bits of the specified workspace register R are logically ANDed with the corresponding bits of the 16 bit binary constant value contained in the word immediately following the instruction. The 16 bit result is compared to zero and is placed in the register R:

R AND Value → R  
R AND Value : 0

Recall that the AND operation results in 1 only if *both* inputs are 1.

*Status Bits Affected:*    **LGT, AGT, EQ**

*Example:*    **ANDI 0,>6D03**

If workspace register 0 contains D2AB<sub>16</sub>, then (D2AB) AND (6D03) is 4003<sub>16</sub>:

Value =	0110	1101	0000	0011
R0 =	1101	0010	1010	1011
R0 AND Value =	0100	0000	0000	0011 = 4003 <sub>16</sub>

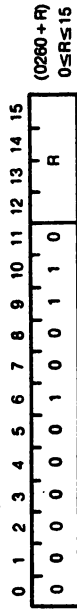
This value is placed in R0. The LGT and AGT status bits are set to 1.

*Application:* ANDI is used to zero all bits that are not of interest and leave the selected bits (those with ones in Value) unchanged. This can be used to test single bits or isolate portions of the word, such as a four bit group.

OR IMMEDIATE

ORI

Format: ORI R, Value



*Operation:* The bits of the specified workspace register R are ORed with the corresponding bits of the 16 bit binary constant contained in the word immediately following instruction. The 16 bit result is placed in R and is compared to zero:

R OR Value → R  
R OR Value : 0

Recall that the OR operation results in a 1 if *either* of the inputs is a 1.

*Status Bits Affected:* LGT, AGT, EQ

*Example:* ORI 5, >6D03

If R5 contained D2AB<sub>16</sub>, then R5 will be changed to FFAB<sub>16</sub>:

R5 = 1101	0010	1010	1011
Value = 0110	1101	0000	0011
1111	1111	1010	1011

1011 = FFAB<sub>16</sub> = R5 OR Value

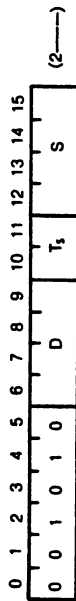
with LGT being set to 1.

*Application:* Used to implement the OR logic in the system.

## EXCLUSIVE OR

# XOR

**Format: XOR     G<sub>s</sub>R<sub>d</sub>**



**Operation:** The exclusive OR is performed between corresponding bits of the data addressed by G<sub>s</sub> and the contents of workspace register R<sub>d</sub>. The result is placed in workspace register R<sub>d</sub> and is compared to 0:

$$\begin{array}{l} M(G_s) \text{ XOR } R_d \longrightarrow R_d \\ M(G_s) \text{ XOR } R_d : 0 \end{array}$$

**Status Bits Affected: LGT, AGT, EQ**

**Example: XOR     @CHANGE,2**

If location CHANGE contains 6D03<sub>16</sub> and R2 contains D2AA<sub>16</sub>, R2 will be changed to BFA9<sub>16</sub>:

$$\begin{array}{rcl} \text{CHANGE Data} & = & 0110 \quad 1101 \quad 0000 \quad 0011 \\ R2 & = & 1101 \quad 0010 \quad 1010 \quad 1010 \\ M(\text{CHANGE}) \text{ XOR } R2 & = & 1011 \quad 1111 \quad 1010 \quad 1001 = \text{BFA9}_{16} \end{array}$$

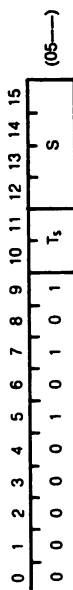
and the LGT status bit will be set to 1. Note that the exclusive OR operation will result in a 1 if *only one* of the inputs is a 1.

**Application:** XOR is used to implement the exclusive OR logic for the system.



INVERT

INV

**Format: INV G**

**Operation:** The bits of the data addressed by G are replaced by their complement. The result is compared to 0 and is stored at the G location:

$$\begin{array}{l} \overline{M(G)} \longrightarrow M(G) \\ M(G) : 0 \end{array}$$

**Status Bits Affected: LGT, AGT, EQ**

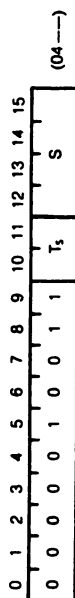
**Example: INV 11**

If R11 contains 00FF<sub>16</sub>, the instruction would change the contents to FF00<sub>16</sub>, causing the LGT status bit to set to 1.

**Application:** INV is used to form the 1's complement of 16 bit binary numbers, or to invert system data.

## CLR

*Format:* **CLR    G**



*Operation:* 0000<sub>16</sub> is placed in the memory location specified by G.

0000<sub>16</sub> → M(G)

*Affect on Status:* None

*Example:* **CLR    \*11**

would clear the contents of the location addressed by the contents of R11, that is:

0000<sub>16</sub> → M(R11)

*Application:* CLR is used to set problem arguments to 0 and to initialize memory locations to zero during system start-up operations.

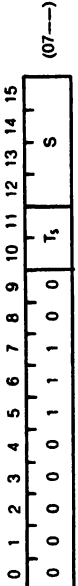
---

# CLR

SET TO ONE

Format: SETO G

SETO



Operation: FFFF<sub>16</sub> is placed in the memory location specified by G: FFFF<sub>16</sub> → M(G)

Affect on Status: None

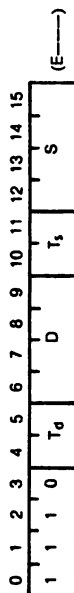
Example: SETO 11  
would cause all bits of R11 to be 1.

Application: Similar to CLR

# SET ONES CORRESPONDING

# SOC

**Format:** SOC     $G_s, G_d$



**Operation:** This instruction performs the OR operation between corresponding bits of the data addressed by  $G_s$  and the data addressed by  $G_d$ . The result is compared to 0 and is placed in the  $G_d$  location:

$$\begin{array}{l} M(G_s) \text{ OR } M(G_d) \longrightarrow M(G_d) \\ M(G_s) \text{ OR } M(G_d) : 0 \end{array}$$

**Status Bits Affected:** LGT, AGT, EQ

**Example:** SOC    3,@NEW

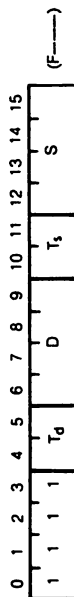
If location NEW contains AAAA<sub>16</sub> and R<sub>3</sub> contains FF00<sub>16</sub>, the contents at location NEW will be changed to FF00<sub>16</sub> and the LGT status bit will be set to 1.

**Application:** Provides the OR function between any two words in memory.

# SET ONES CORRESPONDING, BYTE

## SOCB

*Format:* **SOCB**  $G_s, G_d$



*Operation:* The logical OR is performed between corresponding bits of the byte addressed by  $G_s$  and the byte addressed by  $G_d$  with the result compared to 0 and placed in location  $G_d$ :

$MB(G_s) \text{ OR } MB(G_d) \longrightarrow MB(G_d)$   
 $MB(G_s) \text{ OR } MB(G_d) : 0$

*Status Bits Affected:* **LGT, AGT, EQ, OP**

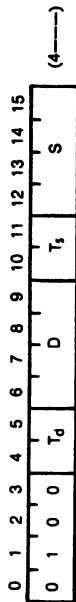
*Example:* **SOCB 5,8**

If R5 contains F013<sub>16</sub> and R8 contains AA24<sub>16</sub>, the most significant byte of R8 will be changed to FA<sub>16</sub> so that R8 will contain FA24<sub>16</sub> and the LGT status bit will be set to 1.

*Application:* The SOCB provides the logical OR function on system bytes.

# SET TO ZEROES CORRESPONDING

**Format: SZC     $G_s, G_d$**



**Operation:** The data addressed by  $G_s$  forms a mask for this operation. The bits in the destination data (addressed by  $G_d$ ) that correspond to the one bits of the source data (addressed by  $G_s$ ) are cleared. The result is compared to zero and is stored in the  $G_d$  location.

$$\begin{array}{l} \overline{M(G_s)} \text{ AND } M(G_d) \longrightarrow M(G_d) \\ \overline{M(G_s)} \text{ AND } M(G_d) : 0 \end{array}$$

**Status Bits Affected: LGT, AGT, EQ**

**SZC**

**Example: SZC 5,3**

If R5 contains 6D03<sub>16</sub> and R3 contains D2A8<sub>16</sub>, this instruction will cause the R3 contents to change to 92A8<sub>16</sub>:

R5 (Mask) = 0110 1101 0000 0011

R3 = 1101 0010 1010 1010

Result = 1001 0010 1010 1000 = 92A8<sub>16</sub>

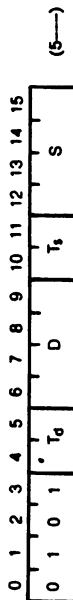
with the LGT status bit set. The underlined entries indicate which bits are to be cleared.

*Application:* SZC allows the programmer to selectively clear bits of data words. For example, when an interrupt has been serviced, the interrupt request bit can be cleared by using the SZC instruction.

## SET TO ZEROES CORRESPONDING, BYTES

# SZCB

**Format: SZCB  $G_s G_d$**



**Operation:** The byte addressed by  $G_s$  will provide a mask for clearing certain bits of the byte addressed by  $G_d$ . The bits in the  $G_d$  byte that will be cleared are the bits that are one in the  $G_s$  byte. The result is compared to zero and is placed in the  $G_d$  byte:

$$\frac{\overline{MB(G_s)}}{MB(G_s)} \text{ AND } MB(G_d) \longrightarrow MB(G_d)$$

$$\frac{\overline{MB(G_s)}}{MB(G_s)} \text{ AND } MB(G_d) : 0$$

**Status Bits Affected:** LGT, AGT, EQ, OP

**Example:** SZCB @BITS,@TEST

If location BITS is an *odd* address which locates the data 18F0<sub>16</sub>, and location TEST contains an *even* address which locates the data AA24<sub>16</sub>, the instruction will clear the first four bits of TEST data changing it to 0A24<sub>16</sub>.

**Application:** Provides selective clearing of bits of system bytes.

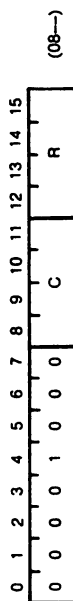


## SHIFT INSTRUCTIONS

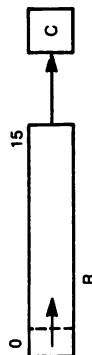
These instructions are used to perform simple binary multiplication and division on words in memory and to rearrange the location of bits in the word in order to examine a given bit with the carry (C) status bit.

### SHIFT RIGHT ARITHMETIC

**Format:** SRA R,Cnt



**Operation:** The contents of the specified workspace register R are shifted right Cnt times, filling the vacated bit position with the sign (most significant bit) bit: The shifted number is compared to zero:



**Status Bits Affected:** LGT, AGT, EQ, C

# SRA

*Number of Shifts:* Cnt (number contained in the instruction from 0 to 15) specifies the number of bits shifted unless Cnt is zero in which case the shift count is taken from the four least significant bits of workspace register 0. If both Cnt and these four bits are 0, a 16 bit position shift is performed.

*Example:*    **SRA 5,2**    Shift R5 2 bit positions right  
                  **SRA 7,0**

If R0 least four bits contain  $6_{16}$ , then the second instruction will cause register 7 to be shifted 6 bit positions (Cnt in that instruction is 0):

If R7 Before Shift = 1011 1010 1010 1010 = BAAA<sub>16</sub>  
   R7 After Shift    = 1111 1110 1110 1010 = FEEA<sub>16</sub>  
   If R5 Before Shift = 0101 0101 0101 0101 = 5555<sub>16</sub>  
   R5 After Shift    = 0001 0101 0101 0101 = 1555<sub>16</sub>

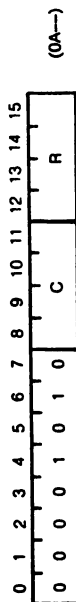
After the R7 shift the LGT would be set, and Carry = 1

After the R5 shift LGT and AGT would be set and Carry = 0

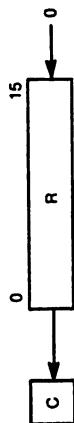
*Application:* SRA provides binary division by  $2^{Cnt}$ .

SHIFT LEFT ARITHMETIC

## SLA

**Format: SLA R,Cnt**

**Operation:** The contents of workspace register R are shifted left Cnt times (or if Cnt = 0, the number of times specified by the least four bits of R0) filling the vacated positions with zeroes. The carry contains the value of the last bit shifted out to the left and the shifted number is compared to zero:



**Status Bits Affected: LGT, AGT, EQ, C, OV**

**Example: SLA 10,5**

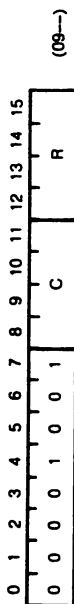
If workspace register 10 contains 1357<sub>16</sub> the instruction would change its contents to 6AE0<sub>16</sub>, causing the arithmetic greater than (AGT), logical greater than (LGT), and overflow (OV) bits to set. Carry would be zero, the value of the last bit shifted.

**Application:** SLA performs binary multiplication by 2<sup>Cnt</sup>

## SHIFT RIGHT LOGICAL

# SRL

**Format:** **SRL R,Cnt**



**Operation:** The contents of the workspace register specified by R are shifted right Cnt times (or if Cnt = 0, the number of times specified by the least four bits or R0) filling in the vacated positions with zeroes. The carry contains the value of the last bit shifted out to the right and the shifted number is compared to zero:



**Status Bits Affected:** **LGT, AGT, EQ, C**

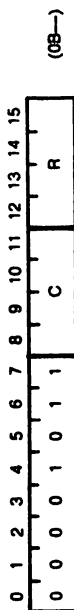
**Example:** **SRL 0,3**

If R0 contained FFEF<sub>16</sub>, the contents would become 1FFD<sub>16</sub> with the AGT, LGT, and C bits set to 1:

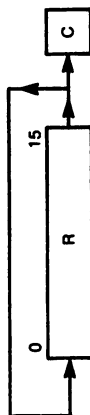
R0 Before Shift = 1111 1111 1110 1111 = FFEF<sub>16</sub>

R0 After Shift = 0001 1111 1111 1101 = 1FFD<sub>16</sub>

**Application:** Performs binary division by 2<sup>Cnt</sup>

SHIFT RIGHT CIRCULAR**SRC****Format: SRC R,Cnt**

**Operation:** On each shift the bit shifted out of bit 15 is shifted back into bit 0. Carry contains the value of the last bit shifted and the shifted number is compared to 0. The number of shifts to be performed is the number Cnt, or if Cnt = 0, the number contained in the least significant four bits of R0:



**Status Bits Affected: LGT, AGT, EQ, C**

**Example: SRC 2,7**

If R2 initially contains FFEF<sub>16</sub>, then after the shift it will contain DFFF<sub>16</sub> with LGT and C set to 1.

R2 Before Shift = 1111 1111 1110 1111 = FFEF<sub>16</sub>

R2 After Shift = 1101 1111 1111 1111 = DFFF<sub>16</sub>

**Application:** SRC can be used to examine a certain bit in the data word, change the location of 4 bit groups, or swap bytes.

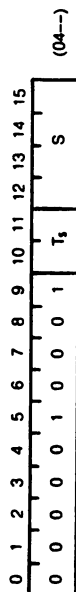
## UNCONDITIONAL BRANCH INSTRUCTIONS

These instructions give the programmer the capability of choosing to perform the next instruction in sequence or to go to some other part of the memory to get the next instruction to be executed. The branch can be a subroutine type of branch, in which case the programmer can return to the point from which the branch occurred.

### BRANCH

**B**

*Format:* **B**      **G<sub>s</sub>**



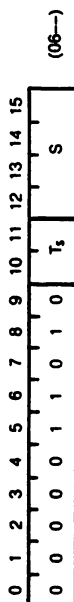
*Operation:* The G<sub>s</sub> address is placed in the program counter, causing the next instruction to be obtained from the location specified by G<sub>s</sub>.

*Affect on Status:* None

*Example:* **B**      \*3

If R3 contains 21CC<sub>16</sub>, then the next instruction will be obtained from location 21CC<sub>16</sub>.

*Application:* This instruction is used to jump to another part of the program when the current task has been completed.

BRANCH AND LINK**BL****Format: BL    G<sub>s</sub>**

**Operation:** The source address G<sub>s</sub> is placed in the program counter and the address of the instruction following the BL instruction is saved in workspace register 11.

G<sub>s</sub> → PC  
(Old PC) → R11

**Affect on Status:** None

**Example: BL    @TRAN**

Assume the BL instruction is located at 3200<sub>16</sub> and the value assigned to TRAN is 2000<sub>16</sub>. PC will be loaded with the value 2000<sub>16</sub> (TRAN) and R11 will be loaded with the value 3202<sub>16</sub> (old PC value).

**Application:** This is a shared workspace subroutine jump. Both the main program and the subroutine use the same workspace registers. To get back to the main program at the branch point, the following branch instruction can be used at the end of the subroutine:

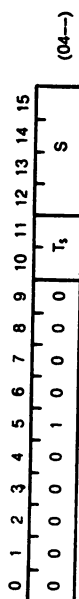
B    \*11

which causes the R11 contents (old PC value) to be loaded into the program counter.

## BRANCH AND LOAD WORKSPACE POINTER

# BLWP

**Format: BLWP G<sub>s</sub>**



**Operation:** The word specified by the source  $G_s$  is loaded into the workspace pointer (WP) and the next word in memory ( $G_s + 2$ ) is loaded into the program counter (PC) to cause the branch. The old workspace pointer is stored in the new workspace register 13, the old PC value is stored in the new workspace register 14, and the status register is stored in new workspace register 15:

$M(G_s) \longrightarrow \text{WP}$   
 $M(G_s + 2) \longrightarrow \text{PC}$   
 $(\text{Old WP}) \longrightarrow \text{New R13}$   
 $(\text{Old PC}) \longrightarrow \text{New R14}$   
 $(\text{Old ST}) \longrightarrow \text{New R15}$

*Affect on Status:* None



**Example: BLWP \*3**

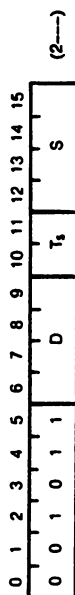
Assuming that R3 contains 2100<sub>16</sub> and location 2100<sub>16</sub> contains 0500<sub>16</sub> and location 2102<sub>16</sub> contains 0100<sub>16</sub>, this instruction causes WP to be loaded with 0500<sub>16</sub> and PC to be loaded with 0100<sub>16</sub>. Then, location 051A<sub>16</sub> will be loaded with the old WP value, the old PC value will be saved in location 051C<sub>16</sub>, and the status (ST) will be saved in location 051E<sub>16</sub>. The next instruction will be taken from address 0100<sub>16</sub> and the subroutine workspace will begin at 0500<sub>16</sub> (R0). BLWP and XOP do not test IREQ at the end of instruction execution.

*Application:* This is a context switch subroutine jump with the transfer vector location specified by G<sub>s</sub>. It uses a new workspace to save the old values of WP, PC, and ST (in the last three registers). The advantage of this subroutine jump over the BL jump is that the subroutine gets its own workspace and the main program workspace contents are not disturbed by subroutine operations.

# XOP

## EXTENDED OPERATION

**Format:** XOP G<sub>s</sub>,n



**Operation:** n specifies which extended operation transfer vector is to be used in the context switch branch from XOP to the corresponding subprogram. The effective address G<sub>s</sub> is placed in R11 of the subprogram workspace in order to pass an argument or data location to the subprogram:

$M(n \times 4 + 0040_{16}) \longrightarrow WP$   
 $M(n \times 4 + 0042_{16}) \longrightarrow PC$   
 (Old WP)  $\longrightarrow$  New R13  
 (Old PC)  $\longrightarrow$  New R14  
 (Old ST)  $\longrightarrow$  New R15  
 $G_s \longrightarrow$  New R11

*Affect on Status:* Extended Operation (X) bit is set.

**Example: XOP \*1,2**

Assume R1 contains  $0750_{16}$ . WP is loaded with the word at address  $48_{16}$  (first part of transfer vector for extended operation 2) and PC is loaded with the word at address  $4A_{16}$ . If location  $48_{16}$  contains  $0200_{16}$ , this will be the address of R0 of the subprogram workspace. Thus, location  $0236_{16}$  (new R11) will be loaded with  $0750_{16}$  (contents of R1 in main program), location  $023A_{16}$  (new R13) will be loaded with the old WP value, location  $023C_{16}$  will be loaded with the old PC value, and location  $023E_{16}$  (new R15) will be loaded with the old status value:

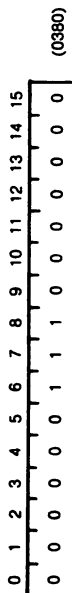
$M(48_{16})$	$\longrightarrow$	WP	
$M(4A_{16})$	$\longrightarrow$	PC	
(Old WP)	$\longrightarrow$	$M(023A_{16})$	New R13
(Old PC)	$\longrightarrow$	$M(023C_{16})$	New R14
(Old ST)	$\longrightarrow$	$M(023E_{16})$	New R15
$0750_{16}$	$\longrightarrow$	$M(0236_{16})$	New R11

*Application:* This can be used to define a subprogram that can be called by a single instruction. As a result, the programmer can define special purpose instructions to augment the standard 9900 instruction set.

## RETURN WITH WORKSPACE POINTER

# RTWP

**Format:** RTWP



**Operation:** This is a return from a context switch subroutine. It occurs by restoring the WP, PC, and ST register contents by transferring the contents of subroutine workspace registers R13, R14, and R15, into the WP, PC, and ST registers, respectively.

R13 → WP  
R14 → PC  
R15 → ST

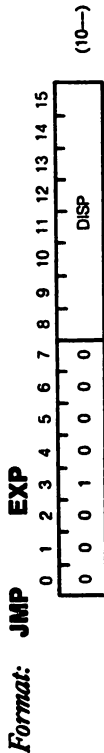
**Status Bits Affected:** All (ST receives the contents of R15)

**Application:** This is used to return from subprograms that were reached by a transfer vector operation such as an interrupt, extended operation, or BLWP instruction.

---

# JMP

## UNCONDITIONAL JMP



**Operation:** The signed displacement defined by EXP is added to the current contents of the program counter to generate the new value of the program counter. The location jumped to must be within  $-128$  to  $+127$  words of the present location.

**Affect on Status:** None

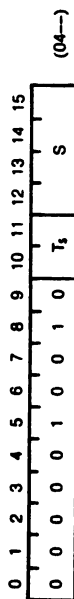
**Example:** JMP THERE

If this instruction is located at  $0018_{16}$  and THERE is the label of the instruction located at  $0010_{16}$ , then the Exp value placed in the object code would be FB (for  $-5$ ). Since the Assembler makes this computation, the programmer only needs to place the appropriate label or expression in the operand field of the instruction.

**Application:** If the subprogram to be jumped to is within 128 words of the JMP instruction location, the unconditional JMP is preferred over the unconditional branch since only one memory word (and one memory reference) is required for the JMP while two memory words and two memory cycles are required for the B instruction. Thus, the JMP instruction can be implemented faster and with less memory cost than can the B instruction.

## EXECUTE

**Format:** X      G<sub>s</sub>



**Operation:** The instruction located at the address specified by G<sub>s</sub> is executed.

**Status Bits Affected:** Depends on the instruction executed

**Example:** X      \*11

If R11 contains 2000<sub>16</sub> and location 2000<sub>16</sub> contains the instruction for CLR 2 then this execute instruction would clear the contents of register 2 to zero.

**Application:** X is useful when the instruction to be executed is dependent on a variable factor.

X

## CONDITIONAL JUMP INSTRUCTIONS

These instructions perform a branching operation only if certain status bits meet the conditions required by the jump. These instructions allow decision making to be incorporated into the program. The conditional jump instruction mnemonics are summarized in *Table 6-1* along with the status bit conditions that are tested by these instructions.

**Format: Mnemonic Exp**



**Operation:** If the condition indicated by the branch mnemonic is true, the jump will occur using relative addressing as was used in the unconditional JMP instruction. That is, the Exp defines a displacement that is added to the current value of the program counter to determine the location of the next instruction, which must be within 128 words of the jump instruction.

**Effect on Status Bits:** None

**Example:** C R1, R2  
JNE LOOP

JH  
JL  
JHE  
JLE  
JGT  
JLT  
JEQ  
JNE  
JOC  
JNC  
JNO  
JOP

The first instruction compares the contents of registers one and two. If they are not equal, EQ = 0 and the JNE instruction causes the branch to LOOP to be taken. If R1 and R2 are equal, EQ = 1 and the branch is not taken.

**Table 6-1. Status Bits Tested by Instructions**

<i>Mnemonic</i>	<i>L&gt;</i>	<i>A&gt;</i>	<i>EQ</i>	<i>C</i>	<i>OV</i>	<i>OP</i>	<i>Jump if:</i>	<i>CODE*</i>
JH	X	—	X	—	—	—	$L > \bullet \overline{EQ} = 1$	B
JL	X	—	X	—	—	—	$L > +EQ = 0$	A
JHE	X	—	X	—	—	—	$L > +EQ = 1$	4
JLE	X	—	X	—	—	—	$\overline{L} > +EQ = 1$	2
JGT	—	X	—	—	—	—	$A > = 1$	5
JLT	—	X	X	—	—	—	$A > +EQ = 0$	1
JEQ	—	—	X	—	—	—	$EQ = 1$	3
JNE	—	—	X	—	—	—	$EQ = 0$	6
JOC	—	—	—	X	—	—	$C = 1$	8
JNC	—	—	—	X	—	—	$C = 0$	7
JNO	—	—	—	—	X	—	$OV = 0$	9
JOP	—	—	—	—	—	X	$OP = 1$	C

*Note:* In the Jump if column, a logical equation is shown in which  $\bullet$  means the AND operation, + means the OR operation, and a line over a term means negation or inversion.

\*CODE is entered in the CODE field of the OPCODE to generate the machine code for the instruction.

**Application:** Most algorithms and programs with loop counters require these instructions to decide which sequence of instructions to do next.

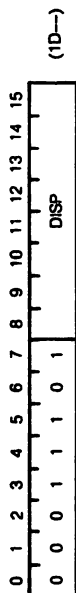


## CRU INSTRUCTIONS

The communications register unit (CRU) performs single and multiple bit programmed input/output for the microcomputer. All input consists of reading CRU line logic levels into memory, and all output consists of setting CRU output lines to bit values from a word or byte of memory. The CRU provides a maximum of 4096 input and 4096 output lines that may be individually selected by a 12 bit address which is located in bits 3 through 14 of workspace register 12. This address is the hardware base address for all CRU communications.

### SET BIT TO LOGIC ONE

*Format:*    **SBO**    **disp**



# SBO

*Operation:* The CRU bit at disp plus the hardware base address is set to one. The hardware base address is bits 3 through 14 of workspace register 12. The value disp is a signed displacement.

1  $\longrightarrow$  Bit (disp + base address)

*Affect on Status:* None

*Example:*    **SBO**    **15**

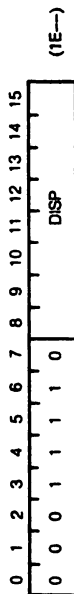
If R12 contains a software base address of  $0200_{16}$  so that the hardware base address is  $0010_{16}$  (the hardware base address is one-half the value of the contents of R12 excluding bits 0, 1 and 2), the above instruction would set CRU line  $010F_{16}$  to a 1.

*Application:* Output a one on a single bit CRU line.

# SET BIT TO LOGIC ZERO

## SBZ

**Format:** SBZ      disp



**Operation:** The CRU bit at disp plus the base address is reset to zero. The hardware base address is bits 3 through 14 of workspace register 12. The value disp is a signed displacement.

0 —> Bit (disp + hardware base address)

**Affect on Status:** None

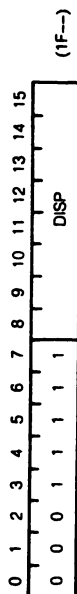
**Example:** SBZ    2

If R12 contains 0000<sub>16</sub>, the hardware base address is 0 so that the instruction would reset CRU line 0002<sub>16</sub> to zero.

**Application:** Output a zero on a single bit CRU line.

## Test Bit

**Format:** TB disp



**Operation:** The CRU bit at disp plus the base address is read by setting the value of the equal (EQ) status bit to the value of the bit on the CRU line. The hardware base address is bits 3 through 14 of workspace register 12. The value disp is a signed displacement.

Bit (disp + hardware base address)  $\longrightarrow$  EQ

**Status Bits Affected:** EQ

**Example:** TB 4

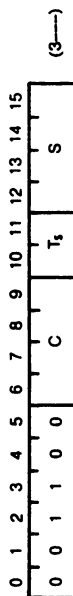
If R12 contains 0140<sub>16</sub>, the hardware base address is A0<sub>16</sub> (which is one-half of 0140<sub>16</sub>):

R12 Contents = 0000 0001 0100 0000

Note that the underlined hardware base address is 0A0<sub>16</sub>. Equal (EQ) would be made equal to the logic level on CRU line 0A0<sub>16</sub> + 4 = CRU line 0A4<sub>16</sub>.

**Application:** Input the CRU bit selected.

TB

LOAD CRU**LDCR****Format: LDCR G<sub>s</sub>,Cnt**

**Operation:** Cnt specifies the number of bits to be transferred from the data located at the address specified by G<sub>s</sub>, with the first bit transferred from the least significant bit of this data, the next bit from the next least significant bit and so on. If Cnt = 0, the number of bits transferred is 16. If the number of bits to be transferred is one to eight, the source address is a byte address. If the number of bits to be transferred is 9 to 16, the source address is a word address. The source data is compared to zero before the transfer. The destination of the first bit is the CRU line specified by the hardware base address, the second bit is transferred to the CRU line specified by the hardware base address + 1, and so on.

**Status Bits Affected: LGT, AGT, EQ****OP** (odd parity) with transfer of 8 or less bits.**Example: LDCR @TOM,8**

Since 8 bits are transferred, TOM is a byte address. If TOM is an even number, the most significant byte is addressed. If R12 contains 0080<sub>16</sub>, the hardware base address is 0040<sub>16</sub> which is the CRU line that will receive the first bit transferred. 0041<sub>16</sub> will be the address of the next bit transferred, and so on to the last (8th) bit transferred to CRU line 0047<sub>16</sub>. This transfer is shown in Figure 6-7.

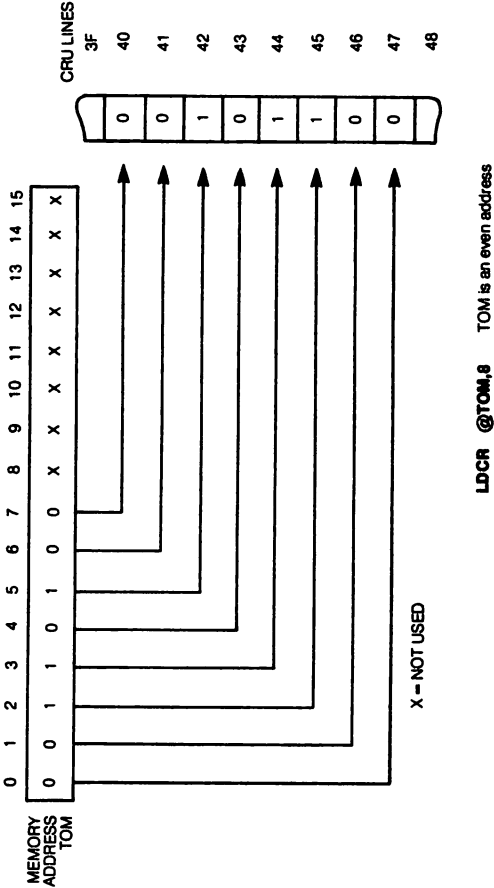


Figure 6-7. LDCR byte transfer

*Application:* The LDCR provides a number of bits (from 1 to 16) to be transferred from a memory word or byte to successive CRU lines, starting at the hardware base address line; the transfer begins with the least significant bit of the source field and continues to successively more significant bits. A further example of word versus byte transfers is given in Figure 6-8, in which a 9 bit (word addressed source) transfer is shown.

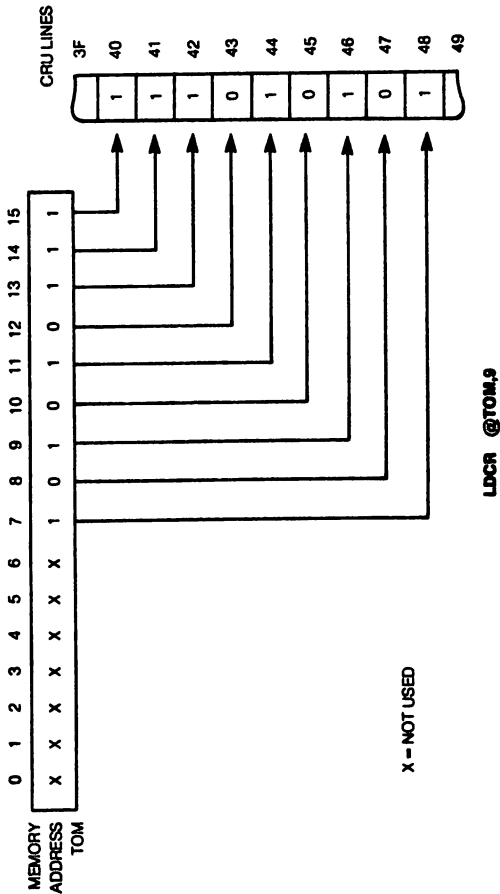
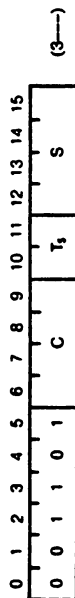


Figure 6-8. LDCR Word transfer

## STORE CRU

# STCR

**Format:** STCR G<sub>s</sub>Cnt



**Operation:** Cnt specifies the number of bits to be transferred from successive CRU lines (starting at the hardware base address) to the location specified by G<sub>s</sub>, beginning with the least significant bit position and transferring successive bits to successively more significant bits. If the number of bits transferred is 8 or less, G<sub>s</sub> is a byte address. Otherwise, G<sub>s</sub> is a word address. If Cnt = 0, 16 bits are transferred. The bits transferred are compared to zero. If the transfer does not fill the entire memory word, the unfilled bits are reset to zero.

**Status Bits Affected:** LGT, AGT, EQ

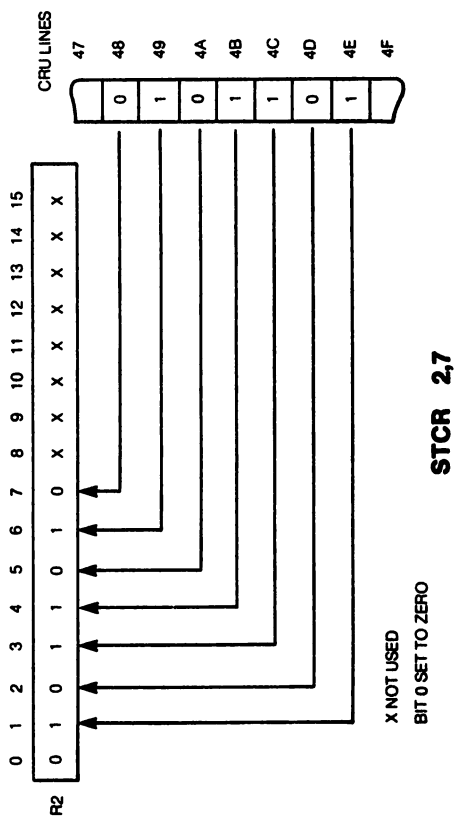
OP for transfers of 8 bits or less

**Example:** STCR 2,7

Since 7 bits are to be transferred this is a byte transfer so that the bits will be transferred to the most significant byte of R2. Figure 6-9 illustrates this transfer assuming that R12 contains 90<sub>16</sub> so that the hardware base address is 48<sub>16</sub> for the first bit to be transferred.



*Note:* Bits 8-15 are unchanged if transfer is less than 8 bits.



*Figure 6-9. STCR Example*

## CONTROL INSTRUCTIONS

The control instructions are primarily applicable to the Model 990 Computer. These instructions are RSET (Reset), IDLE, CKOF (Clock off), CKON (Clock on), LREX (restart). The Model 990/10 also supports the long distance addressing instructions: LDS (Load long distance source) and LDD (Long distance destination). The use of these instructions are covered in the appropriate Model 990 computer programmer's manuals.

The control instructions have an affect on the 9900 signals on the address lines during the CRU Clock as shown below:

Instruction	A <sub>0</sub>	A <sub>1</sub>	A <sub>2</sub>	OP CODE
LREX	H	H	H	<div> <div>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</div> <div>0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0</div> </div> <div>(03E0)</div>
				<div> <div>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</div> <div>0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0</div> </div> <div>(03C0)</div>
CKOF	H	H	L	<div> <div>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</div> <div>0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0</div> </div> <div>(03C0)</div>
				<div> <div>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</div> <div>0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0</div> </div> <div>(03A0)</div>
CKON	H	L	H	<div> <div>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</div> <div>0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0</div> </div> <div>(03A0)</div>
				<div> <div>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</div> <div>0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0</div> </div> <div>(03A0)</div>

Instruction	A <sub>0</sub>	A <sub>1</sub>	A <sub>2</sub>	OP CODE																																
RSET	L	H	H	<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table> (0360)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15																					
0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0																					
IDLE	L	H	L	<table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table> (0340)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15																					
0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0																					
CRU	L	L	L																																	

The IDLE instruction puts the 9900 in the idle condition and causes a CRUCLK output every six clock cycles to indicate this state. The processor can be removed from the idle state by 1) a  $\overline{\text{RESET}}$  signal, 2) any interrupt that is enabled, or 3) a  $\overline{\text{LOAD}}$  signal.

For the 9900 the above instructions are referred to as external instructions, since external hardware can be designed to respond to these signals. The address signals A<sub>0</sub>, A<sub>1</sub>, and A<sub>2</sub> can be decoded and the instructions used to control external hardware.

# **Index**



# Index

---

## A

A, 39, 242  
AB, 39, 244  
ABS, 40, 255  
Addresses, 6, 18  
Addressing, CRU, 216  
Addressing, immediate, 29, 46  
Addressing, indexed, 30, 48  
Addressing, register direct, 29, 47  
Addressing, register indirect, 29, 47  
Addressing, relative, 30, 49  
Addressing, symbolic, 29, 48  
Addressing modes, 28, 46  
AI, 39, 246  
AND, 24  
ANDI, 41, 267  
AORG, 68  
Architecture, 13, 33  
Architecture, memory-to-memory, 35  
Arithmetic instructions, 39, 227  
Arithmetic logic unit, 15  
Assembler, 7  
Assembler directives, 68  
Assembler syntax, 65  
Assembly-language formats, 217

## B

B, 43, 284  
BASIC, 3  
Binary addition, 21  
Binary number system, 5  
Binary subtraction, 23

Bit, 9  
BL, 43, 285  
BLWP, 43, 286  
Branch instructions, 229  
Branch instructions, unconditional, 43  
BSS, 68  
Bus, 9  
Bus, bidirectional, 10  
Bus width, 9  
Byte, 11

## C

C, 40, 260  
Carry bit, 16  
CB, 41, 262  
CI, 41, 264  
CLR, 42, 272  
COC, 41, 265  
Communications register unit, 35  
Compare instructions, 228  
Comparison instructions, 40  
Condition code, 15  
Control instructions, 46  
Control line, read, 11  
Control line, write, 11  
Control transfer, 26  
Control unit, 18  
CRU, 35, 57  
CRU instructions, 45  
CZC, 41, 266

## D

DATA, 69

Data pointer, 999  
Data transfer, 20  
Data transfer instruction, 38, 227, 233  
DEC, 40, 252  
DECT, 40, 253  
Direct addressing, 29, 47  
DIV, 40, 258

## E

EASY BUG, 59, 70, 73  
END, 72  
Entry point, 65  
EQU, 70  
Exclusive-OR, 24  
Extended operation vectors, 37

## F

Field, label, 7  
Field, operand, 7  
Fields, 65, 218-224  
Function select, 15

## G

GROM, 54, 56

## H

Hexadecimal, 6

## I

Immediate addressing, 29, 46  
INC, 40, 250  
INCT, 40, 251  
Indexed addressing, 30, 48  
Index register, 48  
Indirect addressing, 29, 47  
Instruction, 6  
Instruction descriptions, 230  
Instruction set, 38-49, 224  
Interrupt mask, 36  
Interrupt priority code, 36  
Interrupts, 36  
Interrupt vector, 37  
INV, 42, 271

## J

JEQ, 45  
JGT, 45  
JH, 45  
JHE, 45  
JL, 45  
JLE, 45  
JLT, 45  
JNC, 45  
JMP, 44, 291  
JNE, 45  
JNO, 45

JOC, 45  
JOP, 45  
JUMP, 26  
Jump instructions, 293  
Jump instructions, conditional, 44

## L

Label field, 7  
Language, high-level, 5  
Language, machine, 5  
LDCR, 46, 304  
LI, 38, 233  
LIMI, 39, 234  
Line-by line assembler, 59, 60  
Line numbers, 4  
LOAD, 20  
Logical instructions, 228  
Logic instructions, 41  
LWPI, 39, 235

## M

Memory map, 63  
Microprocessor, block diagram of, 14  
Microprocessor, program, 3  
Mini Memory module, 53, 59  
Mnemonics, 7, 29  
MOV, 39, 236  
MOVB, 39, 237  
MOVE, 20  
MPY, 40, 256

## N

NEG, 40, 254  
Negative bit, 16  
Nesting, 27

## O

Object code, 7, 66  
Opcode field, 7  
Operand field, 7  
Operands, 15, 28  
OR, 24  
ORI, 41, 269  
Overflow bit, 16

## P

Program, ASCII-binary string to binary number, 140  
Program, ASCII-decimal string to binary number, 151  
Program, ASCII-hex string to binary number, 146  
Program, ASCII to decimal, 128  
Program, BCD to binary, 130  
Program, binary number to ASCII-decimal string, 148

Program, binary to ASCII-binary string, 137  
 Program, binary to ASCII-hex string, 144  
 Program, binary to BCD, 134  
 Program, branch and link, 176  
 Program, branch and load workspace pointer, 178  
 Program, change screen color, 210  
 Program, clearing the screen, 183  
 Program, convert string to number, 194  
 Program, decimal to ASCII, 130  
 Program, determine numbers, 103  
 Program, display the text, 184  
 Program, find first nonblank character, 113  
 Program, find last nonblank character, 114  
 Program, find maximum value, 106  
 Program, find minimum byte value, 107  
 Program, find the larger number, 88  
 Program, generate cursor, 187  
 Program, hex to ASCII, 125  
 Program, how many negative numbers, 101  
 Program, keyboard input and display, 190  
 Program, length of string, 110  
 Program, make an integer, 117  
 Program, pattern match, 119  
 Program, raise number to a power, 202  
 Program, reciprocal of a number, 164  
 Program, replace zeros, 115  
 Program, sine of an angle, 170  
 Program, 16-bit addition, 83  
 Program, 16-bit data transfer, 77  
 Program, 16-bit sum of data, 96  
 Program, 64-bit data transfer, 82  
 Program, 64-bit division, 158  
 Program, square root, 162  
 Program, string comparison, 121  
 Program, sum of squares, 90  
 Program, table of factorials, 92  
 Program, 32-bit addition, 85  
 Program, 32-bit by 32-bit multiply, 154  
 Program, 32-bit sum of data, 98  
 Program counter, 18, 34, 215  
 Programming, 3

## R

RAM, 12  
 Random access, 12  
 References, 64, 72

Register, 16  
 Relative addressing, 30, 49  
 ROM, 11  
 ROTATE, 25, 26  
 RTWP, 44, 290  
 RWM, 11

## S

S, 40, 247  
 SB, 40, 248  
 SBC, 297  
 SBO, 45, 295  
 SBZ, 45  
 Scratch pad, 53  
 SCZ, 42  
 Service routine, 37  
 SETO, 42, 273  
 SHIFT, 25  
 Shift instructions, 42, 229  
 SLA, 42, 281  
 SOC, 42, 274  
 SOCB, 275  
 Sound generator, 54  
 Source code, 66  
 Source program 7  
 SR, 42, 279  
 SRC, 43, 283  
 SRL, 43, 282  
 Starting address, 64  
 Statements, 4  
 Status, 15  
 Status register, 16  
 STCR, 46, 300  
 STORE, 20  
 STST, 39, 240  
 STWP, 39, 241  
 Subroutine, 5, 27  
 SWPB, 39, 239  
 Symbolic addressing, 29, 48  
 SZC, 276  
 SZCB, 42, 278  
 SYM, 70

## T

TB, 298  
 TEXT, 72  
 TI-99/4A, block diagram of the, 52  
 TMS9900, 13, 33, 45  
 TMS9901, 57  
 TMS9918A, 54  
 Two's complement, 23

## U

Utilities, 182



<b>V</b>		<b>X</b>
Video data processor RAM, 54	X, 44, 292	
	XOP, 43, 288	
<b>W</b>		
Workspace, 34	XOR, 41, 270	
Workspace, changing the, 35		<b>Z</b>
Workspace pointer, 34	Zero bit, 16	

# Fundamentals of TI-99/4A Assembly Language

by M. S. Morley

Whether you want to get more action from arcade-type games or get more efficient performance from business or financial programs, this book shows the way. It's a must-have handbook for every TI owner or user who wants to tap the potentials offered by Assembly Language . . . faster, more efficient program execution and better use of the machine's memory space!

You'll learn about the architecture of the TI-99/4A and its 16-bit TMS 9900 microprocessor—information most manuals "assume" you already know! And you'll appreciate this easy-to-follow approach to a language that is far less difficult than you've imagined!

Part I provides you with the fundamental concepts of programming languages and microprocessor systems. You'll discover the differences and similarities of BASIC, machine language, and assembly language. You'll also examine the internal organizational features common to most microprocessors and read about the five basic types of microprocessor operations, and the addressing mode concept.

In Part II, you'll learn how the TI Mini Memory Module software cartridge enhances and expands your assembly-language programming efforts . . . learn the internal organization of data file structures used in the TI-99/4A . . . and find out the specifics of assembler syntax, codes, and more.

Plus, there are 47 worked out program modules developed by the author to illustrate the potentials offered by Assembly Language . . . programs that you can use to do the jobs you want done!

If you are a technician, engineer, or hobbyist who wants to learn 9900 assembly language programming . . . a TI-99/4A owner who wants more out of your computer . . . or a programmer who wants to learn 16-bit assembly language . . . this book is for YOU!

M. S. Morley is a project manager for Rockwell International Missile Systems Division, where he is primarily a hardware designer and an assembly language programmer.

## OTHER POPULAR TAB BOOKS OF INTEREST

Assembly Language Programming for the  
TRS-80™ Model 16 (No. 1649—\$9.95 paper;  
FPT \$10.25; \$15.95 hard)

Machine and Assembly Language Programming  
(No. 1389—\$9.95 paper; FPT \$10.25; \$15.95  
hard)

**TAB TAB BOOKS Inc.**

Blue Ridge Summit, Pa. 17214

Send for FREE TAB Catalog describing over 750 current titles in print.

FPT > \$11.50

ISBN 0-8306-1722-1