Heathkit
ZENITH
Educational
Systems

Macro-86

ASSEMBLY LANGUAGE PROGRAMMING

Written by: **Lawrence P. Larsen**
Senior Educational Media Designer

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

Model EC-1201
595-3173-01

Library of Congress Cataloging in Publication Data

Larsen, Lawrence P., 1944-
MACRO-86 assembly language programming.

At head of title: Heathkit Zenith educational systems.
Includes index.
1. Intel 8086 (Microprocessor)—Programming. 2. Intel 8088 (Microprocessor)—Programming. 3. Assembler language (Computer program language). 4. Microcomputers—Programming. I. Title. II. Series.
QA76.8.I292L37 1984 001.64'2 84-19295

ISBN 0-87119-100-8

CONTENTS

One last point about the 8088/8086 microprocessors. Because of the rapidly growing family of microprocessor support devices, Intel Corporation is now using a different method of identifying different versions of microprocessor "systems." The name of the basic 8088 is now the iAPX 88/10, while the basic 8086 is now the iAPX 86/10. However, because the general public is most familiar with the older designations 8088 and 8086, we will continue to use those model numbers.

Course Prerequisites

Because of the basic nature of this course, there are only a few prerequisites. Before you begin this course, you should:

1. Have access to a Zenith 100-series microcomputer or an IBM PC, and the appropriate operating system (MS-DOS, Z-DOS, or PC-DOS).

2. Be familiar with the basic operation of the operating system (e.g. directory usage, file transfer between disks, command line structure, etc.).

3. Have access to the system programs:

   A. MASM.EXE (assembler).
   B. LINK.EXE (object code linker).
   C. EXE2BIN.EXE (EXE to binary converter).
   D. CREF.EXE (cross reference facility).
   E. LIB.EXE (library facility).
   F. EDLIN.COM (file editor or equivalent).
   G. DEBUG.COM (program/file debugger).

4. Be familiar with the binary and hexadecimal number systems, and computer math. If you have any questions in this area, a quick overview is provided in Appendix A and Appendix C of this course.

COURSE OBJECTIVES

When you complete this course, you will be able to:

1. Describe the internal structure of the Intel 8088 microprocessor.
2. Describe how the physical address of instructions and data is determined.
3. Structure a COM-type program.
4. Structure an EXE-type program.
5. Develop a program flowchart to define a problem.
6. Design program arrays, records, and structures.
7. Add, subtract, multiply, and divide signed and unsigned binary numbers, and packed and unpacked decimal numbers.
8. Input and output data.
9. Develop interrupt service routines.
10. Design macro instructions.
11. Combine multiple program source listing files.
12. Link multiple program object files.
13. Structure the assembler listing format to match your printer.

COURSE OUTLINE

UNIT 1 — Introduction to the Microcomputer

A. Introduction

B. Unit Objectives

C. Microcomputer Overview

1. Microcomputer Hardware
2. Stored Program Concept
3. Bits, Bytes, Words, and Doublewords
4. Self-Review Questions

D. The 8088 Microprocessor

1. Stripped Down MPU

a. The Execution Unit
b. The Bus Interface Unit

2. Execution Unit Registers
3. Self-Review Questions

E. Interfacing To Memory

1. RAM vs. ROM
2. Memory Segmentation

a. Locating the Segments
b. Address Determination
c. Segment Contents
d. Determining the Physical Address of Each Segment

3. Self-Review Questions

F. Controlling The MPU

1. Accessing Memory (Fetch-Execute Sequence)
2. Machine Code Instruction
3. Self-Review Questions

UNIT 2 — Introduction To Assembly Language Programming

A. Introduction

B. Unit Objectives

C. Assembly Process Overview

1. The Editor
2. The Assembler
3. The Linker
4. EXE-To-Binary Conversion
5. Self-Review Questions

D. Elements Of Microsoft Assembly Language

1. The Assembly Language Instruction Set
2. The Assembler Directive Statement
3. Assembler Directives

a. ORG
b. EQU
c. DB
d. DW
e. END

4. Self-Review Questions

E. Operand Typing

1. Immediate Operands
2. Register Operands
3. Memory Operands
4. Self-Review Questions

F. Arithmetic Operators

1. Self-Review Questions

UNIT 3 — Program Transfer Instructions

UNIT 4 — Subroutines

UNIT 5 — New Addressing Modes

UNIT 6 — Expanding the Instruction Set

UNIT 7 — Segmented Memory And I/O

UNIT 8 — Interrupts and Strings

UNIT 9 — Code Macros and Other Interesting MACRO-86 Features

INSERT

Unit 1

INTRODUCTION TO THE MICROCOMPUTER

CONTENTS

INTRODUCTION

Assembly language, unlike the higher level programming languages, requires that the "programmer" understand the microcomputer "hardware" that operates under the control of the "software" program. This is necessary, since you as the programmer must control the step-by-step operation of the system, right down to the specific register or memory location that stores the data. Therefore, before you can learn to program in assembly language, you must learn how the microcomputer operates in a general fashion, and specifically the operation of the microprocessor unit (MPU), the brain of the microcomputer.

The purpose of this Unit is to introduce the microcomputer, and then describe the microprocessor in detail. You will learn the function of each register in the 8088 MPU. You will also learn how the MPU "talks" to memory (the microcomputer data storage area) and input/output (I/O) circuits. This will form the basis or foundation for the entire course.

If you are already familiar with an MPU outside of the 8088/8086 family, don't skip this unit, thinking there is nothing to be learned. While all MPUs operate in a similar manner, each is unique in how it interfaces to memory or I/O, and to its internal registers. Without a sound understanding of how the 8088 MPU operates, you will have a difficult time learning the MACRO-86 Assembler.

In order to get the most out of this and future units, it is necessary that you have a working knowledge of number systems. If you are not familiar with binary, decimal, and hexadecimal notation, refer to Appendix A at the end of the text. This Appendix contains a short discussion of number systems.

By the same token, an understanding of computer math is also desirable. Appendix B provides a quick review of binary addition, subtraction, multiplication, and division, as well as One's complement and Two's complement arithmetic, and the Boolean operations AND, OR, Exclusive OR (XOR), and Invert.

Because we will be using different number systems throughout the course, each will, when necessary, be identified by a letter suffix. For example, the number 10110101 binary will be written 10110101B. Generally, a number without a suffix is assumed to be a decimal number. Also, a hexadecimal number that begins with a "letter" will be preceded by a zero. Thus, F1A3H is properly written 0F1A3H.

The following are suffixes used in this course:

Binary ..... B
Decimal ..... blank or D
Hexadecimal ... H

Use the "Unit Objectives" that follow to evaluate your progress. When you can successfully accomplish all of the objectives, you will have completed this Unit. You can use the "Unit Activity Guide" to keep a record of those sections that you have completed.

UNIT OBJECTIVES

When you complete this Unit, you will be able to:

1. Define the following terms: Microprocessor, microcomputer, clock, memory device, bus, interface adapter, I/O device, input, output, I/O port, program, stored program concept, loading, bit, byte, word, doubleword, high byte, low byte, high word, low word, memory width, memory length, read only memory (ROM), random access memory (RAM), physical address, logical address, base address, segment, register, accumulator, and machine code.

2. Name the two processing units within the 8088 MPU.

3. Name and describe the function of the following sections of the 8088 MPU: The arithmetic logic unit (ALU), data/general registers, segment registers, flag registers, address generation and bus control section, instruction queue, and instruction pointer (IP).

4. Draw a block diagram of a basic microcomputer and identify the various components.

5. Draw a diagram of and identify the 8088 MPU registers.

6. Describe how the physical address of instructions and data is determined in various segments.

UNIT ACTIVITY GUIDE

		Completion Time
☐	Read the Section on "Microcomputer Overview."	_____
☐	Complete Self-Review Questions 1-22.	_____
☐	Read the Section on "The 8088 Microprocessor."	_____
☐	Complete Self-Review Questions 23-30.	_____
☐	Read the Section on "Interfacing To Memory."	_____
☐	Complete Self-Review Questions 31-46.	_____
☐	Read the Section on "Controlling The MPU."	_____
☐	Complete Self-Review Questions 47-55.	_____
☐	Perform the Experiment.	_____
☐	Complete the Unit 1 Examination.	_____
☐	Check the Examination Answers.	_____

MICROCOMPUTER OVERVIEW

In general terms, a **microcomputer** is a device capable of accepting data, applying prescribed processes to data, and supplying the results of these processes. It usually consists of input and output devices, data storage devices, and a process control unit. The data is processed in binary (digital) form, although the microcomputer can accommodate data using almost any number base.

This section will first examine the devices that make up a typical microcomputer. Then it will describe the process that makes the microcomputer such a powerful tool. Finally, it will present the language of the microcomputer.

Microcomputer Hardware

You have probably heard many terms for the process control unit in a microcomputer. The two most common are the **Central Processing Unit** (CPU) and the **Microprocessing Unit** (MPU). We'll use the term MPU throughout this course, since it more accurately describes the process control unit of a microcomputer.

As the name implies, the MPU controls the operation of the microcomputer. It is a complex logic element that is capable of performing arithmetic, logic, and control operations such as the sending and receiving of data from outside of the Microcomputer. Figure 1-1 is a simple block diagram of a microcomputer. Notice that all of the other devices interface with the MPU. We'll leave a detailed description of the MPU for a later section in this unit.

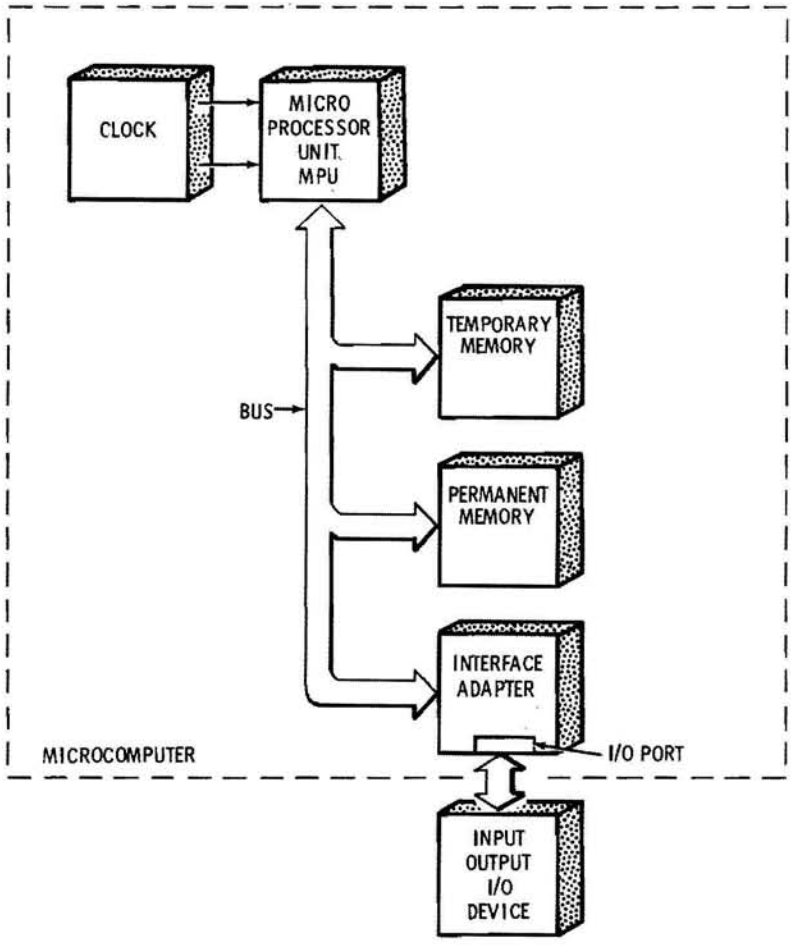


Figure 1-1
A basic microcomputer.

A feature of the microcomputer is the speed with which it processes data. This speed makes event timing critical. To control the timing, every microcomputer contains a master **clock** that produces a digital pulse train. Data is either received by, or sent from, the microcomputer at times dictated by the clock pulse. For instance, data may be brought into the MPU only on the falling edge of a clock pulse. The clock has no control over what occurs within the MPU; however, it does control when an action takes place.

Also included in the basic microcomputer are the **memory** devices. These devices, depending on their type, act as either temporary or permanent storage areas for information or data. The size of the memory, or the number of memory devices in the microcomputer, will vary greatly depending on the needs of the user.

Interface adapter is an all-encompassing term for a wide range of components. These components ensure that the electrical input signal is of an amplitude and type compatible with the rest of the circuitry in the microcomputer. By the same token, they ensure the compatibility of the output signal with the peripheral devices. Interface adapters can include items as simple as a bistable latch, or as complex as an analog-to-digital converter. To a great extent, the application dictates the type of interface adapter needed for a particular microcomputer and the associated input/output device.

Tying all of these devices together is the **bus**. It consists of a number of parallel conductors. These conductors serve as a multisignal path for efficient data transfer. Generally, the bus will contain from 8 to 20 conductors depending on the type of data that must be transferred.

Finally, there are the **I/O devices** themselves. These external devices transmit data to, or receive data from, a microcomputer. Quite often, these are referred to as peripherals since they are located on the "periphery" of the microcomputer. The type of device used for I/O is limited only by the user's imagination. Again, a great deal depends on the desired application. However, some of the more common devices are the line printer, CRT (video display), and keyboard.

Of course, all data received from an I/O device is referred to as **input**. Likewise, any data or direction from the microcomputer to the "outside world" is called **output**. In addition, the point at which the I/O device connects to the microcomputer is called the **I/O port**.

Stored Program Concept

For each operation that an MPU can perform, there is a corresponding instruction. You, as the programmer, will organize instructions into a sequence in order to perform specific tasks. These organized groups of instructions are called **programs**.

A simple task, such as adding a list of numbers, may consist of only a few instructions; while other tasks, like controlling an industrial robot, may require hundreds of instructions. If it were necessary for you to enter, or load, these programs into the microcomputer one instruction at a time each time the program was used, a great deal of time would be wasted. To overcome this drawback of microcomputer use, the **stored program concept** was introduced.

In the stored program concept, the list of instructions, or program, is permanently maintained on some type of storage medium. Floppy disks are a popular medium commonly used with the microcomputer. However, ROM, magnetic tape, punched paper tape, and rigid disks are also used as storage media.

When the time comes to use a program, it is loaded into the microcomputer. Essentially, the loading of a program is simply the copying of the program from the permanent storage medium into the temporary storage area in the microcomputer's memory. An extremely long program can be loaded in a few seconds, versus the hours it could take to manually load the same program.

The stored program eliminates the time delays and accuracy problems introduced by the human operator. Once a program is stored, it can be called on quickly and its accuracy is usually unquestioned.

Bits, Bytes, Words, and Doublewords

Our language is based on the alphabet. The individual letter is the smallest recognizable part of the human language. The smallest part of a microcomputer's language is the **bit**. A bit is a binary 1 or 0 that represents an electrical state. Usually, a 1 indicates the presence of voltage while a 0 indicates its absence. The Microcomputer's language, written in these electrical states, is commonly referred to as **machine code**.

The **byte** is the unit of microcomputer language that is smaller than a microcomputer word, but longer than a bit. A byte is roughly comparable to a syllable in human oriented language. By convention, a byte is a unit of information eight bits long.

Each **word** used in the 8088 MPU is 16 bits, or two bytes, in length. Figure 1-2 shows a word. The eight most significant bits of the word are referred to as the **high byte**, while the eight least significant bits are referred to as the **low byte**.

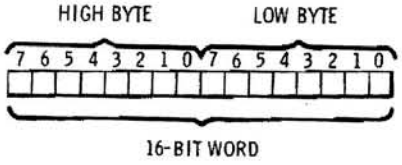


Figure 1-2
A 16-bit microcomputer word.

The 8088 MPU also makes limited use of a 32-bit quantity called a **doubleword**. An example of a doubleword is shown in Figure 1-3. The first 16 bits of the doubleword, shown on the left of the Figure, are called the **high word**; and the last 16 bits, shown on the right, are called the **low word**. As with the word, each part of the doubleword is further divided into a high and a low byte.

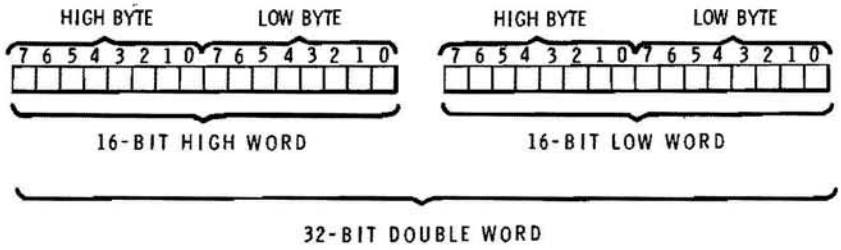


Figure 1-3
A 32-bit doubleword.

Longer computer word lengths allow us to work with larger numbers. A byte can specify, in binary, positive numbers between 0 and decimal 255. The word allows us to represent numbers up to 65,535. Doublewords can represent numbers up to 4,294,967,295. This doesn't mean that the microcomputer is restricted to working with numerical values. The contents of the byte, word, or doubleword can mean many things. Besides the numbers between 0 and 65,535, a 16-bit word can also represent a pair of coded characters. In fact, a bit pattern can represent any meaning you wish to assign to it. For example, a byte of information containing all ones could signify the decimal number 255. But if this same pattern is the output of a thermal sensor, it can just as easily mean that a hazardous situation exists in a piece of equipment.

At this point, it is not important to understand how an MPU differentiates between 11111111B, which means 255, and 11111111B, which means excessive temperature. It is important, however, to understand that identical bit patterns do not always mean the same thing.

Self-Review Questions

The self-review questions in this course are designed to reinforce your knowledge of the material you have studied. To obtain the maximum benefit from these exercises, read each question and answer it to the best of your ability. If you are unsure of the correct response, refer back to the appropriate section of the text. When you are satisfied with your answers to all of the questions, check them against the correct answers at the end of this unit.

1. The _____ is a complex logic element capable of performing arithmetic, logic, and control operations.

2. An MPU, along with all of the other components necessary to interface with the outside world, is called a _____.

3. The _____ acts as the master timer for the microcomputer.

4. _____ devices, depending on the type, act as either temporary or permanent storage areas for data.

5. The _____ consists of a number of parallel conductors that link together the various components that make up the microcomputer.

6. The _____ adapter ensures that the electrical input signals are of an amplitude and type compatible with the rest of the circuitry in the microcomputer.

7. External devices which transmit data to, or receive data from, a microcomputer are known as _____.

8. Information received from an I/O device is referred to as _____.

9. Any information or direction from the microcomputer to the outside world is called an _____.

10. The point at which the I/O device connects to the microcomputer is called the _____.

11. Instructions organized into a sequence in order to perform a specific task is called a/an _____.

12. In the _____ _____ concept, a program is permanently maintained on some type of storage medium.

13. Copying a program from a permanent storage medium into the temporary storage area in the microcomputer's memory is called _____ a program.

14. The smallest part of a microcomputer's language, the _____, is a binary 1 or 0 that represents an electrical state.

15. A microprocessor's language written in electrical states is referred to as _____ _____.

16. The _____ is a unit of information 8 bits long.

17. Each _____ used with the 8088 MPU is 16 bits long.

18. The eight most significant bits of a word are called the _____ _____.

19. The eight least significant bits of a word are called the _____ _____.

20. The 8088 MPU makes limited use of a 32-bit quantity called a _____.

21. The 16 most significant bits of a doubleword are called a _____ _____.

22. The 16 least significant bits of a doubleword are called a _____ _____.

NOTE: The Self-Review Answers are located at the end of this unit, beginning on Page 1-61.

THE 8088 MICROPROCESSOR

The 8088 MPU has considerable capabilities. Matching these capabilities is its complexity. Internally, there are almost two dozen 8-bit and 16-bit registers which are accessible to the programmer. These registers serve as storage locations for data that is used by the MPU during program execution. The programming instruction set (the code that controls the operation of the MPU) consists of over 100 basic instructions, with over a thousand possible variations of these instructions.

To begin our discussion of the 8088 MPU, let's start with a "stripped down" microprocessor. This will give you a chance to see how the microprocessor operates without cluttering the description with a lot of needless registers. Once we have the foundation laid, we will add a few more registers and show you how they are used in specific operations. The next section, "Memory," will complete the discussion of the MPU and show you how the 8088 MPU can access over one million bytes of memory.

Stripped Down MPU

The "stripped down" 8088 MPU, as shown in Figure 1-4, consists of two separate processing units: the **execution unit** (EU) and the **bus interface unit** (BIU). The EU performs arithmetic and logic operations, controls most of the internal registers, and manipulates data. The BIU performs bus operations, including data transfer, and controls the remaining registers for the MPU. Both of these processing units are capable of independent operation. That is, each unit can perform its assigned duties without assistance from the other unit. Although the EU is isolated from the bus by the BIU, for certain operations it can gain access to the bus by requesting the BIU temporarily suspend its activities so that it can either store or retrieve data for the EU.

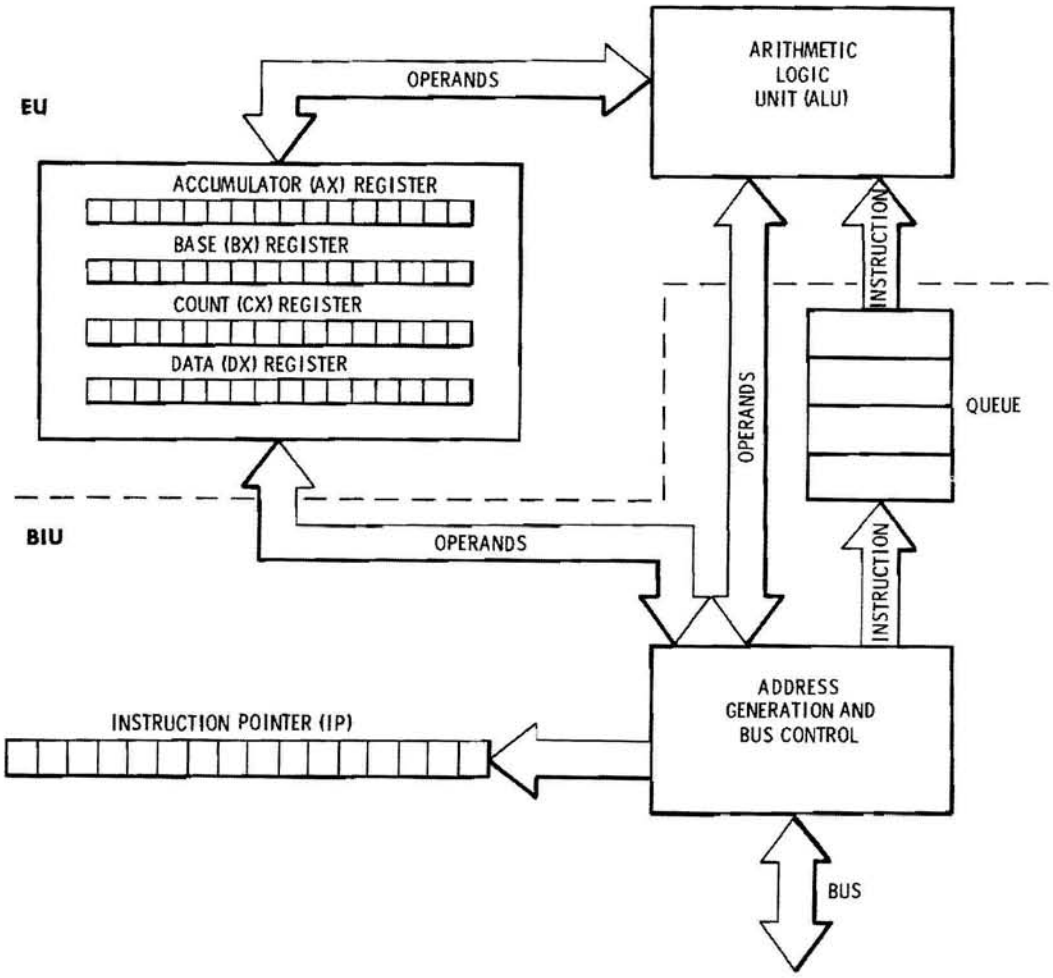


Figure 1-4
Simplified block diagram of an 8088 MPU.

THE EXECUTION UNIT

In our "stripped down" version of the 8088 MPU, the EU consists of two sections. These are the general, or data, registers and the arithmetic logic unit (ALU).

The **general**, or **data registers**, are the most useful registers in the microprocessor. They act as temporary storage areas for data that has been, or will be, used in computations performed by the MPU. Information can be brought into these registers from memory or peripherals via the BIU. Data contained in these registers can also be transmitted from the MPU to memory or peripherals; again, via the BIU. Finally, data can be transferred from one register to another within the EU.

All data registers are 16 bits long. Each of these registers can be further divided into a high byte, consisting of the eight most significant bits, and a low byte which contains the eight least significant bits. For example, Figure 1-5 shows the AX (Accumulator) register divided into the high byte (AH) and the low byte (AL). The other data registers can be similarly divided.

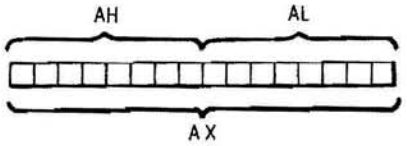


Figure 1-5
The AX register.

The 8088 MPU can use the data registers to hold 16 bits of data or it can manipulate data that is held in these registers. This is fine for 16-bit operations. However, the 8088 can also perform operations on single bytes of data. In this case, either the high or low byte of a register can be specified for the particular operation. In effect, each data register can act as a 16-bit or as two 8-bit registers.

All general registers are basically interchangeable. Most operations performed on data in one register may also be performed on data in the other registers. However, as the names accumulator, base, count, and data imply, the individual registers do have "special" uses. Some instructions require that specific operations be performed in certain registers. These will be discussed in detail when the particular instructions are introduced later in the course. For now, you may consider all general registers to be identical in structure and function.

The **arithmetic logic unit** is the "workhorse" of the EU. It receives instructions from the program through the BIU. The binary bit patterns from these instructions are applied to the internal circuitry of the ALU. The ALU then performs an arithmetic operation, such as add or subtract, or a logic operation, like AND or OR, on the data specified by the instruction.

At times, the information contained in the instruction and in the registers is not sufficient to complete an operation. When this occurs, the ALU, as part of the EU, can "ask" the BIU to obtain additional information from memory or to store a piece of data in memory. The ALU always interfaces with memory and other peripherals through the BIU. It has no contact with the "outside" world.

THE BUS INTERFACE UNIT

The bus interface unit consists of the address generation and control section, the instruction queue, and the instruction pointer.

The **address generation and bus control section** performs all bus operations for the MPU. It retrieves, or fetches, instructions from the program for the ALU. When necessary, it accesses locations, or addresses, in memory so that the EU can either retrieve data from or send data to those locations. This section also controls the direction in which information flows on the bus. If information is to be transmitted, the address generation and bus control section ensures that all proper control signals are set for transmit. The same is done when it is necessary to retrieve data. It could be said that this section performs the same function as a switchman in a rail yard; it ensures that all things arrive at the proper destination, at the proper time, and in the proper order.

The **instruction queue** in the 8088 MPU is made up of four 8-bit wide registers. These serve as a temporary storage area for instructions or data. The EU normally receives its instructions and their associated data through the queue. Thus, while the EU is processing an instruction, the BIU can continue to access additional instructions and data, maintaining a full queue for increased performance. While this may seem like useless information for a programmer, you will find later, that knowing the state of the queue can help you understand and determine program timing.

The **instruction pointer** is a 16-bit register that contains the address, or location of the next instruction to be executed. In effect, the instruction pointer controls the sequence in which the program will be executed by telling the EU which instruction it must execute next. Each time the EU accepts an instruction from the queue, the instruction pointer is updated to point to the next instruction in the program sequence.

Execution Unit Registers

In addition to the accumulator, base, count, and data registers, the EU contains four data, or general, index/pointer registers and a status flag register. These are shown in Figure 1-6. The stack pointer, base pointer, source index, and destination index registers, like the previously described registers, can participate in most arithmetic and logic operations. With the exception of the base pointer and status/control flag registers, these registers are also used implicitly in some instructions, as shown in Figure 1- 7.

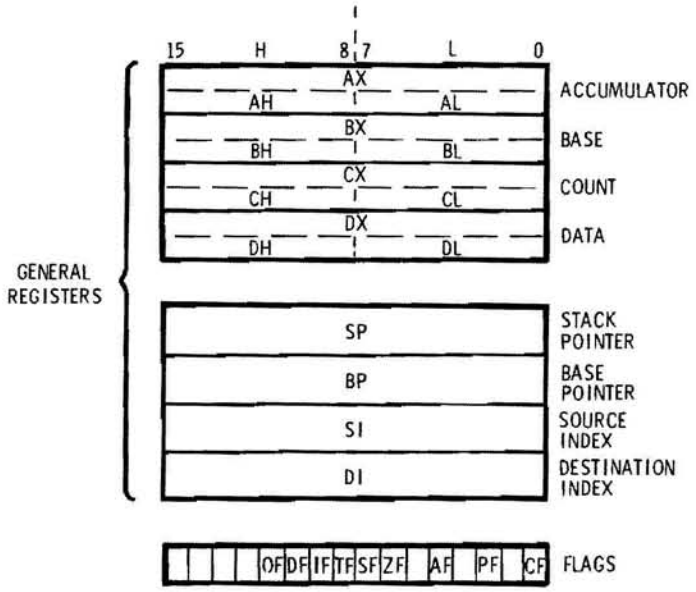


Figure 1-6
Registers contained in the execution unit of the 8088 MPU.

REGISTER	OPERATIONS
AX	Word Multiply, Word Divide, Word I/O
AL	Byte Multiply, Byte Divide, Byte I/O, Translate, Decimal Arithmetic
AH	Byte Multiply, Byte Divide
BX	Translate
CX	String Operations, Loops
CL	Variable Shift and Rotate
DX	Word Multiply, Word Divide, Indirect I/O
SP	Stack Operations
SI	String Operations
DI	String Operations

Figure 1-7
EU registers used implicitly with specific instruction types.

The **flag register** contains nine 1-bit flags. The other seven bits in the register are undefined (they could assume a logic 1 or 0 level when power is applied to the processor). Figure 1-8 lists the flag bits that are used. The three control flags can be set or cleared by the program to alter processor operation. For example, setting TF (the trap flag) puts the processor into single-step mode for debugging a program. The other six flag bits reflect the result of an arithmetic or logic operation. A group of instructions is available that allows a program to alter its execution depending on the state of these flags; that is, on the result of a prior operation. Trying to define the operation of each flag at this time is not practical, since you must first understand the instructions that affect the flags. Therefore, we will defer these descriptions to a later time.

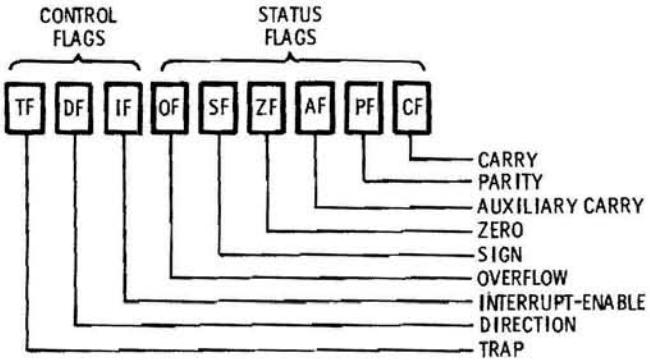


Figure 1-8
8088 MPU flag register bits.

Self-Review Questions

23. The 8088 MPU consists of two separate processing units. They are the _____ _____ and the _____ _____ _____.

24. The _____ or _____ registers are temporary storage areas where data can be held either before or after it is used in computations by the microprocessor.

25. The _____ _____ _____ performs arithmetic or logic operations on the data specified in the instructions.

26. The _____ _____ _____ _____ _____ _____ performs all bus operations for the 8088 MPU.

27. The _____ _____ is a 4-byte temporary storage area within the MPU that is used to hold instructions or data.

28. The _____ _____ is a 16-bit register that contains the address of the next instruction to be executed by the MPU.

29. The four data or general registers that can be used as either 8-bit or 16-bit registers are the _____, _____, _____, and _____ registers.

30. The _____ register contains nine 1-bit status and control registers.

INTERFACING TO MEMORY

As stated earlier, memory consists of a number of storage locations outside of the MPU. So that order is maintained when you use memory, each location in memory is assigned a number or address.

Figure 1-9 shows how memory is organized. Notice that each location has its own address and that these are numbered sequentially from 00H through 0FFH. Notice also that we have used the 0 prefix and H suffix to indicate hexadecimal notation. This particular memory has 256 separate addresses.

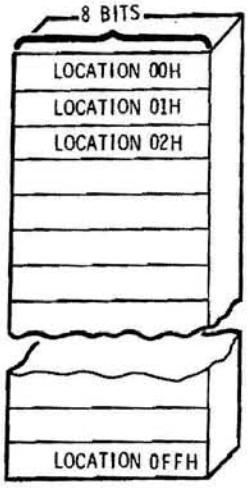


Figure 1-9
Typical memory organization.

The **width** of the memory refers to the amount of data, in bits, which can be stored in or retrieved from any given memory location at one time. The memory shown in Figure 1-9 is eight bits wide. Therefore, you can store or retrieve one byte of data at each memory address. A memory which is 256 bytes long and eight bits wide is commonly referred to as a 256 × 8 memory.

The width of memory used with an MPU is dictated by the amount of data that the MPU can access at one time. Since the 8088 MPU fetches one byte at a time, the memory used with the 8088 MPU must be eight bits wide. On the other hand, the **length** of memory is determined by the needs of the user. For some applications, a 256-byte memory is more than sufficient; but, for others, memory of a significantly larger size is required.

There is a limit, however, to the length of the memory that can be used with an MPU. Recall that the instruction pointer always points to the address of the next instruction to be executed. Also, remember that the IP is a 16-bit register. Because of this, the number of memory locations addressable by the 8088 MPU should be 0000H through 0FFFFH or 65,536 different addresses. Actually this isn't strictly true. You will find later in this section that the BIU uses four segment registers to extend the addressable memory range to over a million bytes.

Before we discuss the expanded addressing capabilities of the 8088 MPU, let's look at the different types of memory that are used in a microcomputer.

RAM vs ROM

Thus far, all storage areas for programs and data have been grouped under the general term memory. There are actually two classifications of memory which must be considered in order to understand the microcomputer. The first of these is **ROM**, or **read only memory**. ROM is a permanent storage area for programs or constants that are essential for the operation of the microcomputer. As the name implies, information can only be read from this type of memory. The contents of a read only memory are protected from any inadvertent write operation that may occur.

An example of the type of program that might be stored in ROM is the system startup routine, more commonly called the "bootstrap" routine, for a microcomputer. This routine sets up the MPU and its supporting circuits according to certain parameters. In fact, it's the "bootstrap" routine that makes it possible for you to enter and run your own programs.

There are other programs which are also stored in ROM, and they perform many functions. These programs, however, have two things in common: they are permanently stored in ROM and they are protected from changes due to a write operation or a power failure.

Generally speaking, you will not have an occasion to program a ROM. This is usually done at the factory at the time of manufacture, and each ROM is designed for a specific use with a specific system.

The second type of memory is **RAM**, or **random access memory**. In actuality, all memory is randomly accessible. That is, it does not have to be addressed in sequential order. But, through convention, only read/write memory is called RAM. In any case, RAM acts as a temporary storage location, within the microcomputer, for programs and data. Most programs you wish to run, and most data you wish to use, must first be loaded into RAM. Once in RAM, programs and data may be altered by the user. As you can see, both RAM and ROM have their own particular uses. You as the programmer, will deal primarily with RAM. Now let's see how the 8088 based microcomputer memory is structured.

Memory Segmentation

Up to this point, we have limited our 8088 based microcomputer to 65,536 bytes of memory. In actuality, the 8088 MPU can address a megabyte, or 1,048,576 bytes of memory. It does this by organizing the available memory into segments.

A **segment** is a logical unit of memory that is 64K bytes long. (1K byte is equal to 1024 bytes.) Each segment is made up of an uninterrupted block of memory locations. In addition, each segment is a separately addressable unit. Because it is separately addressable, each segment must be assigned a **base address**. This base address then becomes the starting address of the segment in memory.

The 8088 MPU has four 16-bit registers that can contain the starting, or base, address of a segment. They are the **CS** (code segment), **DS** (data segment), **SS** (stack segment), and **ES** (extra segment). Each of these registers serve a specific function: program instruction, or code, is addressed by the CS register, data is addressed by the DS register, the stack is addressed by the SS register, and additional data is addressed by the ES register. You will learn more about these register functions when you get into programming the MPU. For now, think of these registers as storage locations for the base address of different segments in memory.

Naturally, a megabyte of memory can contain more than four 64K segments. However, four uniquely defined memory segments is generally more than enough at any one time in a program. Should you need to identify another segment area, it's a simple matter to change the base address in one of the segment registers.

LOCATING THE SEGMENTS

Each of the four segments can be located anywhere within the available one-megabyte memory space. The segments may be adjacent to each other or separated by blocks of empty memory. In fact, the segments may overlap either partially or fully. Figure 1-10 shows three different segment arrangements. In part A, the segments begin at some place

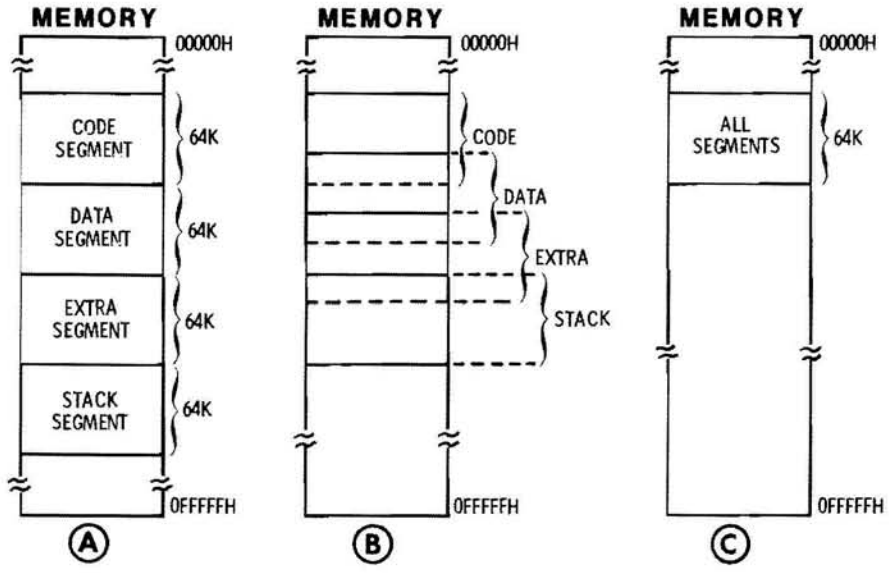


Figure 1-10
Examples of segmented memory.

within memory and are located adjacent to each other. In part B, the segments overlap each other. When two or more segments overlap, they share the overlapped portion of memory. In part C, all four segments share one 64K segment of memory. That may seem to be a problem. However, you will find when you begin writing programs that this type of segment arrangement is very common.

ADDRESS DETERMINATION

So far, we've established that the 8088 MPU can address up to one megabyte of memory. In addition, this memory is arranged in 64K-byte groups called segments. Finally, the base, or starting, address of a segment is identified by a 16-bit segment register within the MPU. Now if it takes a 20-bit address value to identify a location within a megabyte of memory, how can a 16-bit segment register produce that 20-bit value?

The answer lies within the address generation and bus control section of the 8088 MPU. To completely understand the process, however, we must introduce a couple of new terms: logical address and physical address.

The **logical address** is the address of an instruction or piece of data within a 64K byte segment of memory. This segment of memory can be at any location within the 1 megabyte of available memory. The logical address never exceeds 0FFFFH because this is the highest address available in any 64K segment.

The logical address is used as an offset. That is, it is combined with the contents of one of the four segment registers in order to produce a **physical address** that is a unique memory location in the 1 megabyte of addressable memory. To understand how the process works, let's look at the actual procedure as it occurs within the MPU.

For this example, we will have a program that begins at logical address 0100H and the contents of the CS register will be 0FF00H. This is shown in Figure 1-11 part A. The CS register is used here because the MPU "assumes" that program code resides in the code segment.

The address generation and bus control section performs two operations in order to produce the 20-bit physical address. First, it shifts the contents of the segment base register four bit positions to the left. This results in a new 20-bit **base** value in which the four least significant bits are zeros, as shown in Figure 1-11 part B.

Next, the address generation and bus control section adds the 20-bit base value to the 16-bit logical address as shown in Figure 1-11 part C. The result of this addition is the physical address of the beginning of the program in the code segment, 0FF100H. This 20-bit physical address is located in the segment whose base is pointed to by the CS register. Physical addresses within other segments are calculated in a similar manner.

You may come across a physical address specified as 0000:0000 in the hexadecimal number base. This gives the base address value and the logical, or offset, address value in the order BASE:OFFSET. To determine the actual physical address, the base value is shifted four bits left and then added to the offset value. This is a convenient method for specifying an address, since you can easily identify the actual base and offset components of the physical address.

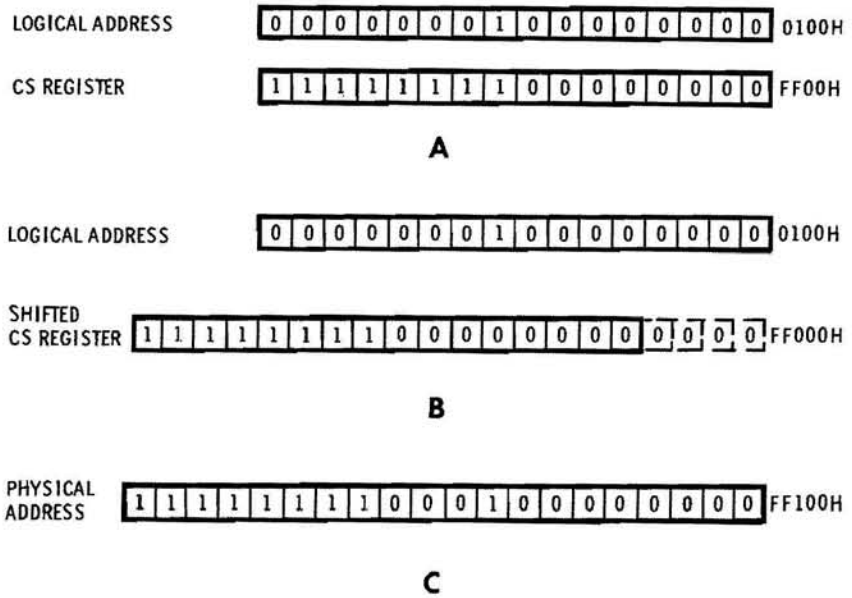


Figure 1-11
Calculating the physical address.

SEGMENT CONTENTS

It would be impossible to maintain any order in a program which used four segments unless the contents of these segments were defined in some way. Unless otherwise specified by the programmer, the MPU "assumes" that certain information will be in specific segments.

For instance, the **list of instructions** which make up the program, is "assumed" to reside in the **code segment**. The data, which consists of the **variables** used in the program, is "assumed" to reside in the **data segment**. Meanwhile, the **stack**, a temporary storage area in memory, is "assumed" to reside in the **stack segment**. The **extra segment** is considered a **secondary data segment**, however, there are certain **special operations** that "assume" that data is moved into or out of the extra segment.

DETERMINING THE PHYSICAL ADDRESS OF EACH SEGMENT

Physical addresses within each of the four separately addressable segments are determined in different ways. The address generation and bus control section uses a specific set of default registers and values to obtain the logical address which is used as an offset within the segment. It then combines these logical addresses with the contents of the various segment registers to obtain the physical addresses of the instructions and data in the different segments of memory. Let's see how this is done for each of the segments.

The **logical address**, at least in the case of the **code segment**, is obtained from the **instruction pointer**. To state this another way, the contents of the IP are the logical address. To obtain the 20-bit physical address which locates the instruction in memory, the contents of the IP are combined with the logically shifted contents of the CS register. That is, the CS register contents are shifted four bits to the left. Then the contents of the IP are added to the shifted CS register contents to produce the physical address. The MPU locates all instruction code in this manner.

The physical address of any memory location in the **data segment** is computed by shifting the contents of the DS register four bits to the left and adding to this quantity the effective address of a specified operand. The operand is part of the assembly language instruction and the **effective** address is a form of **logical** address that is derived from the computations performed by the MPU as prescribed for the addressing mode in which the instruction is written. You'll learn more about operands and effective addresses later in the course. The concept you should keep in mind is that a 16-bit address value is added to the shifted contents of the DS register to produce the physical address of data in memory.

Data within the **stack segment** can be addressed through two different registers. The first register is the **Stack Pointer** (SP). The Stack Pointer provides the **logical** address that is added to the shifted contents of the SS register to produce the physical address of the item within the stack. In this way, the SP register is similar in function to the IP register.

The other register that defaults to the **stack segment** is the **Base Pointer** (BP). Whenever the BP is used in indirect addressing, its contents form the **logical** address portion of the physical address within the stack segment. You'll learn about the various addressing modes in a later unit.

The final segment is the **extra segment**. It is used primarily for specific program functions called "string operations." In a string operation, the default register that contains the extra segment **logical** address is the **Destination Index** (DI) register. As with the other segments, the contents of the ES register are shifted four bits to the left and the logical address in the DI register is then added to this base to form the physical address. In operations that do not involve "strings," there is no default register associated with the extra segment. It is used in a manner similar to the DS register.

The 8088 MPU, depending on the particular instruction, will locate code and data, or variables, within the four segments according to the parameters discussed. This is a convenience to you as the programmer. For instance, if you wish to access some data within the data segment, the address generation and bus control section automatically uses the effective address of the data, as a logical address, along with the contents of the DS register to determine the physical address of the data. You don't have to specify the segment in the instruction.

Self-Review Questions

31. The _____ of memory refers to the amount of data, in bits, which can be stored in, or retrieved from, any given memory location at one time.

32. The _____ of memory is determined by the needs of the user, but limited to the addressing capability of the MPU.

33. _____ _____ _____ is a permanent storage area for programs or constants that are essential for microcomputer operation.

34. _____ _____ _____ acts as a temporary storage area, within the microcomputer, for programs or data.

35. A _____ is a logical unit of memory that is 64K bytes long.

36. Each segment is assigned a _____ _____, which is the starting address of the segment in memory.

37. Name the four segment registers in the 8088 MPU.

    A. _____
    B. _____
    C. _____
    D. _____

38. The _____ address is the address of an instruction or piece of data within a 64K block of memory.

39. Describe how the address generation and bus control section produces a 20-bit physical address. _____

_____

_____

_____

_____

40. The four segments can be located anywhere in memory. _____
True/False

41. The segments may be adjacent to each other, but they must not overlap. _____
True/False

42. The MPU "assumes" that instructions and data are randomly located throughout the four segments. _____
True/False

43. Where does the MPU obtain the logical address of an instruction in the code segment? _____

_____

_____

44. The logical address for determining the physical address in the data segment is not stored in a register. _____
True/False

45. State two ways that the physical address is determined in the stack segment. _____

_____

_____

_____

_____

46. What is the default register for the logical, or offset, address in the extra segment when it is used in a string operation? _____

CONTROLLING THE MPU

Now that you have a general understanding of physical characteristics of a microcomputer and the 8088 MPU, it's time to see how you control the microcomputer through the MPU. Recall that we said the MPU is controlled by a set of instructions called a program. The program tells the MPU exactly what operations to perform. When an 8088 based microcomputer executes a program, it goes through a fundamental sequence that is repeated over and over again. That sequence is called the "fetch-execute" sequence. What it means is that the MPU will fetch and execute each instruction in a manner unique to the 8088 MPU. We'll examine this fetch-execute sequence first, then we'll look at what makes up the machine code instruction.

Accessing Memory (Fetch-Execute Sequence)

Recall that the microcomputer system clock acts as a timer for the MPU — it determines when an action will occur. In the 8088 MPU, a fetch can occur every four clock pulses, or clock cycles. (Actually, the length of time between fetches can vary a slight amount. However, to avoid confusion at this time, we will assume four clock cycles.) A grouping of four clock cycles is called a **bus cycle**. Thus, a fetch can occur once during each bus cycle. The execution time for instructions, on the other hand, varies. Some instructions require as little as two clock cycles to perform, while others take in excess of 100 clock cycles. This can lead to some interesting timing situations.

The BIU and EU operate essentially independently of each other. While the BIU is fetching instructions and data, the EU is executing the instructions. If the BIU takes one bus cycle (four clock cycles) to fetch one byte of an instruction, and the EU is executing an instruction that takes only two clock cycles, the EU will have to wait for the BIU. However, an instruction that requires 100 clock cycles to execute will give the BIU enough time to fill the queue with data. Thus, when the EU is done with the first instruction, it can immediately execute the next instruction, since it is waiting in the queue. Naturally, a program will contain a mixture of instructions so that the EU and BIU will be able to operate on almost a continuous basis.

Later, when you begin writing programs, there will be times when you will have to take into consideration the fetch-execute timing. Appendix D, at the back of this text, contains a complete listing of the 8088 MPU instruction set. Each instruction has a table indicating the execution time for the instruction and the number of bytes needed to construct the instruction. When necessary, we'll show you how to determine the amount of time it takes to execute a group of instructions. Right now, let's see what constitutes a machine code instruction.

Machine Code Instruction

The 8088 MPU has a number of capabilities. Addition, subtraction, read an input, and write to a peripheral, as well as logical processes, are all part of the microprocessor's repertoire. The microcomputer, however, can do nothing on its own. It can perform only the operations that you tell it to perform.

In order to communicate with the microcomputer, you must learn its vocabulary. For each operation that a microcomputer can perform, there is a corresponding instruction. The instruction is used by the programmer to "tell" the microcomputer what operation it must perform. A microcomputer's vocabulary consists of the group of instructions called the **instruction set**.

The microcomputer, however, is a digital, or binary, device that only understands ones and zeros, or machine code. Therefore, instructions must be written for the computer in machine code. The binary pattern, or machine code, for each instruction is unique to that instruction. That pattern, when applied to the internal circuitry of the MPU, causes a particular operation to occur.

Depending on the instruction, the required machine code can be one to six bytes long. The first one or two bytes tell the MPU what operation is to be performed. The next one, two, or four bytes supply the MPU with the logical address for a memory location or data.

Figure 1-12 shows the typical machine instruction encoding format used with the 8088 MPU. The six most significant bits of the first byte of the instruction contain the **operation code** or **opcode**. The opcode tells the MPU precisely what type of operation it is to perform. Each type of operation has its own unique opcode.

The next bit of the first byte of the instruction is the **direction bit**. If the destination of the data used in the operation is a register, then this bit is set to 1. On the other hand, if the destination is in memory, then the bit is reset to 0.

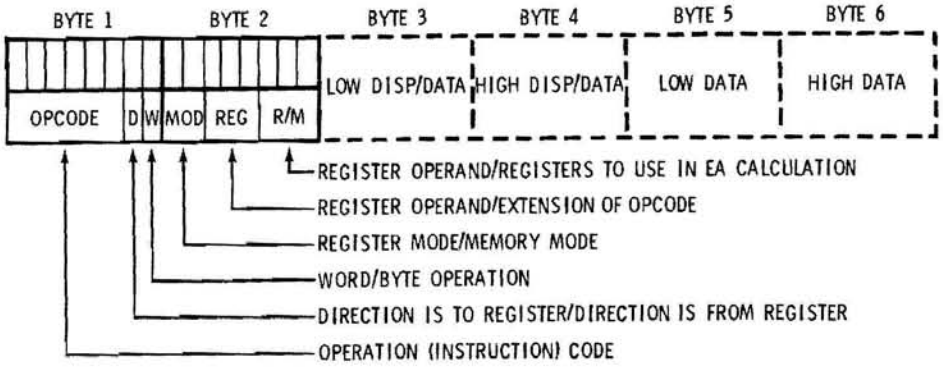


Figure 1-12
Typical machine instruction encoding format.

The last bit of the first byte of the instruction is the **word bit**. A 1 at this location tells the MPU that a 16-bit word is involved in the operation. A 0 in the same place indicates that an 8-bit byte operation is being performed.

The second byte of the instruction, when it is required, contains all of the information necessary to determine the source and destination operands for the instruction. In general, you can say that an operand is any quantity that is used in or results from an operation. However, this broad definition may result in some confusion in certain circumstances. Therefore, for instructional purposes, it is best to use three specific definitions rather than a single general description of an operand. When we are referring to operands, we will use the terms "source operand" and "destination operand."

The **destination operand** is a location, either in a register or in memory, which holds the results of a given operation.

The **source operand**, on the other hand, contains the data upon which the operation will be performed. Either a register or a memory location can be used as a source operand. Figure 1-13 illustrates the possible combinations of source and destination operands using registers and memory locations. Notice that a register can act as both the source and the destination operand in a single instruction. Unlike registers, in a single instruction, memory locations may act as either the source or the destination operand, but never as both.

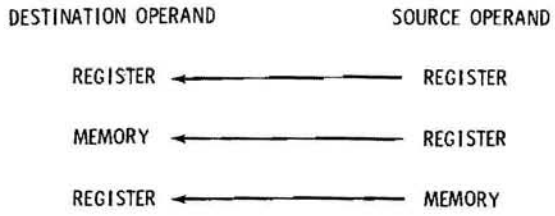


Figure 1-13
Register and memory as source and destination operands.

There is a third type of operand called an **immediate operand**. The immediate operand is not a location in memory, nor is it a register within the MPU. The immediate operand is an actual byte or word of data. This byte or word of data is encoded within an instruction and therefore resides within the program. Since this type of operand is not a place either in memory or within the MPU, it can only serve as a source operand. The possible combinations of immediate, and register or memory operands are shown in Figure 1-14.

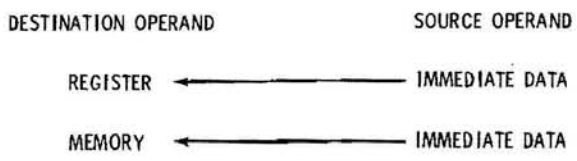


Figure 1-14
Use of immediate data as an operand.

Getting back to the example in Figure 1-12, the first two bits of the second byte of the instruction contain the **MOD field**. This field tells the MPU whether register or memory operands are used in the instruction. If either the source or destination operand is in memory, the instruction is said to be in the memory mode. If this is the case, the MOD field also indicates how the address of the operand is calculated. If neither of the operands is a memory operand, the instruction is in the register mode.

The next three bits of byte two are the **REG** or **register field**. In most instructions, this field contains the identity of a register used as an operand. If the instruction is in the register mode, the indicated register is the destination operand for the instruction. When the instruction is in the memory mode, the contents of this field can specify either the source or the destination operand for the instruction. In some cases, the contents of this field are an extension of the opcode presented in the first byte of the instruction.

The final three bits of this byte are the **R/M** or **register/memory** field. Again, as in the register field, the contents of this field depend on the mode in which the instruction is written. If the instruction is in the register mode, this field specifies the register which acts as the source operand for the instruction. In an instruction written in the memory mode, however, the contents of this field tell how the effective address, or actual address, of either the source or destination operand is calculated.

The last four bytes shown in Figure 1-12 vary a great deal depending on the contents of the first two bytes of the instruction. In fact, in many instructions, these bytes are not used. However, if the **displacement bytes** are used, they will contain numerical data used in the calculation of the effective address of an operand. If the **data bytes** are used in an instruction, they will contain the immediate operand for the instruction.

At this point we must point out that not all instructions follow the same format. Our example was presented as a limited learning tool to familiarize you with the general instruction structure and terminology. This will be useful, however, if you need to analyze a group of instruction code. If you wish to know more about machine encoding and the code structure for the complete instruction set of the 8088 MPU, refer to Appendix C at the back of this text.

Self-Review Questions

47. A group of four clock cycles is called a _____ _____.

48. The 8088 MPU can fetch and execute instructions at the same time. _____
True/False

49. If the MPU clock is operating at 4 MHz, the MPU can effectively fetch _____ instructions per second.

50. The _____ is used by the programmer to "tell" the computer what operation it is to perform.

51. A microcomputer's vocabulary consists of a group of instructions called an _____ _____.

52. The _____ portion of the instruction tells the microcomputer what type of operation to perform.

53. The _____ operand is a location, either in a register or in memory, which holds the results of a given operation.

54. The _____ operand contains the data upon which an operation is performed.

55. An _____ operand is an actual word or byte of data.

EXPERIMENT

The Microcomputer's MPU and Memory Structure

OBJECTIVES:

1. *Demonstrate the 8088 MPU registers and show how they can be manipulated by machine code.*
2. *Introduce a few simple 8088 MPU instructions.*
3. *Demonstrate the difference between RAM and ROM.*

Introduction

Recall from the Course Introduction that we stipulated a number of prerequisites to completing this course. Directly related to this and the later experiments are the following:

- You must have access to a microcomputer that uses a form of Microsoft's Disk Operating System (Zenith's Z-DOS, IBM's PC-DOS, etc.).

- In addition to the DOS and its standard supporting files, you will need a copy of Microsoft's Macro Assembler and its supporting files:

  MASM.EXE
  LINK.EXE
  LIB.EXE
  CREF.EXE
  EXE2BIN.EXE

- You must be familiar with the microcomputer and the DOS to the extent that you can read the disk directory, run a program, format a disk, copy a file, and delete a file.

- Finally, you must be able to use Microsoft's editor, EDLIN.COM, or a similar text editor. We will use EDLIN in all of our experiment examples. However, as you become more proficient in your programming, you will find that a sophisticated text editor is very helpful.

In this experiment, we will use the program debugger, DEBUG.COM, to examine the 8088 MPU registers and memory. One of the fundamental uses of DEBUG is to troubleshoot assembly language programs. While we won't be using it for that purpose in this experiment, it will prove helpful in later experiments.

For now, the debugger will allow you to examine all of the MPU registers discussed in this unit. You will also move data into and out of the registers and observe the effect program execution has on the Instruction Pointer register. Finally, the debugger will give you the opportunity to manipulate data in memory.

While we could have you load the necessary program machine code into memory, one byte at a time, there is an easier way; write the program in assembly language, and then let the microcomputer create the machine code and load it into memory. Therefore, in the first part of the experiment, you will write an assembly language program, assemble the program, and convert the resulting object code to executable machine code.

Procedure

Refer to Figure 1-15 as you perform the following steps. The figure represents the data displayed on the console from the time you call up EDLIN until you end the editing process. Depending on the microcomputer and the version of EDLIN that you are using, you may not see the title line or version number shown in the figure. Note that the asterisk (*) preceding each line of program code was produced by the editor, it is not part of the code.

```
A:EDLIN TEST.ASM

EDLIN  version 1.02
New file
*I
     1:*TEST    SEGMENT
     2:*        ASSUME  CS:TEST,DS:TEST,SS:TEST,ES:TEST
     3:*        ORG     100H
     4:*START:  MOV     AX,1111H
     5:*        MOV     BX,5555H
     6:*        ADD     AX,BX
     7:*        MOV     CX,AX
     8:*TEST    ENDS
     9:*        END     START
    10:*^C

*E

A:
```

Figure 1-15
Using EDLIN to write the program TEST.ASM.

1. Call up the editor and specify the file, or program, you are about to create as TEST.ASM. Remember, you must always specify a file name when you run EDLIN.

   Note that we will use the terms "program" and "file" throughout the course to mean the same thing. While the term file normally relates to data on a mass storage device such as a floppy disk, common usage has made its definition generally equivalent to

2. Enter the Insert mode and type the program beginning on line 1 and ending on line 9. The space between each column is made with the "TAB" key. The "C" on line 10 represents the editor code "CONTROL C" (IBM's editor code is called "CONTROL BREAK") that causes the editor to leave the Insert mode.

   The first three lines and the last two lines of the program contain assembler directives. These directives provide the assembler with specific information about the program. The remaining four lines, 4 through 7, contain the actual program instructions and data in assembly language form.

3. After you have entered the program, end the edit process by typing "E" and RETURN. This allows you to exit the editor and save the program TEST.ASM.

Refer to Figure 1-16 as you convert the assembly language program TEST.ASM into machine executable code in the following steps. Again, the figure represents the console display as you perform the steps. The title blocks for the assembler, the linker, and the EXE2BIN converter programs may not match those in the figure. This information is determined by the type of microcomputer and the version of the software you are using.

```
A:MASM TEST;
The Microsoft MACRO Assembler
Version 1.07, Copyright (C) Microsoft Inc. 1981,82


Warning Severe
Errors  Errors
0       0

A:LINK TEST;

   Microsoft Object Linker V1.10
(C) Copyright 1981 by Microsoft Inc.

Warning: No STACK segment

There was 1 error detected.

A:EXE2BIN TEST.EXE TEST.COM

Exe2bin  version 1.5


A:
```

Figure 1-16
Converting TEST.ASM to machine executable code.

4. At the "A:" prompt, type "MASM TEST;" and RETURN, as shown. This calls the assembler program MASM.EXE and specifies that the program TEST.ASM is to be assembled, producing the appropriate files.

   The title block for the assembler and two error messages will be displayed. The error messages take the form:

Warning Errors	Severe Errors
0	0

   If any value other than 0 is in the error message, you have an error in your program. The assembler will identify the error type, and print the line that contains the error. Often, an error in one line will create an apparent error in another line. For example, if you left the colon out of line 4, the assembler would flag line 4 in error, but it would also flag line 9 in error because of its reference to line 4. If an error is indicated, return to the editor and make any necessary changes. If there are more than one, fix the obvious ones first, then reassemble the program following the description given at the beginning of step 4. When the obvious errors are fixed, the not too obvious errors will probably disappear. If all else fails, erase the file TEST.ASM from the disk and write it over again. Once you get a good assembly, proceed with step 5.

5. At the "A:" prompt, type "LINK TEST;" and RETURN. This calls the object file linker program, LINK, and specifies that the object file TEST.OBJ should be converted to a machine executable file.

   As the linker processes the data, it will print a title block, a warning message, and an error message. This is normal and does not indicate a problem. The warning message indicates that there is no identifiable stack area. The error message relates to the missing stack segment. The program you wrote doesn't need a stack. Therefore, the warning is not important. Later in the course, it will be significant. Any other warning or error message would indicate a problem with the file stored on the disk. If that should happen, try reassembling and linking the program TEST.ASM one more time.

6. At the "A:" prompt, type "EXE2BIN TEST.EXE TEST.COM" and RETURN. This calls the EXE to binary conversion program that translates the TEST.EXE file to TEST.COM.

   Depending on the microcomputer and software you are using, you may not see the title line or version number shown in Figure 1-16.

Discussion

You have just written a short program in assembly language and then converted that program into machine executable code. While you may not fully understand what you did, that is not important at this time. The point of the exercise is to create a machine code program that you can use to manipulate a few of the MPU registers. Later in the course, we will explain all of these processes in great detail. Right now, let's call up the debugger and examine the MPU registers and memory.

Procedure Continued

7. Call up the debugger, by typing "DEBUG" and RETURN. You may or may not see a title line. The debugger command prompt is either a "right arrow" or a hyphen, depending on the debugger.

8. Type "D100" and RETURN. You will see a display similar to the one in Figure 1-17. You have just executed the **dump** command. While the command can take many forms, the three most common are:

Dyyyy:xxxx	(dump segment:offset address)
Dxxxx	(dump offset, default segment is DS register value)
D	(default segment is DS register value, offset is IP register value)

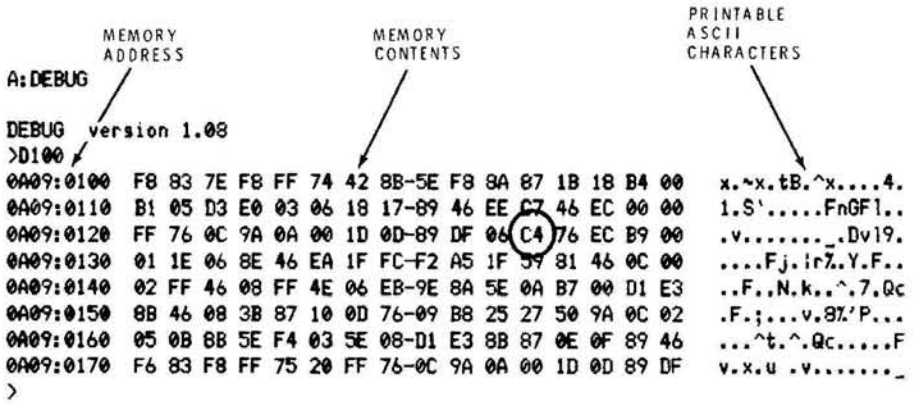


Figure 1-17
Debugger memory dump.

Since you used the second form, your segment value in the memory address may not match the value in Figure 1-17. Notice in the figure that the dump display contains three columns. The large center column contains the hex contents of 128 consecutive memory locations, starting at the address specified by the dump command. These are the actual current contents of memory, in our microcomputer. Your display probably won't match the contents in the figure.

The left column contains the physical address (segment:offset) for the first byte of memory in each row. Naturally, the address of the first byte is the address specified in the dump command. To determine the address of a memory byte within a row, begin counting from the left end of the row with the number zero hex. When you reach your byte, add the count to the row address, and you have the physical address of the byte. For example, the byte circled in the third row of the figure is at address 0A1BBH.

SEGMENT	=	0A090H
OFFSET	=	0120H
COUNT	=	0BH
ADDRESS	=	0A1BBH

The column on the right displays the ASCII character associated with the code at each memory address. This lets you read any text that may be stored in memory. If there is no printable character for that code, the ASCII listing uses a period. The term ASCII stands for American Standard Code for Information Interchange. Most microcomputers use ASCII for data transfer. We'll expand on the concept when we get into assembly language programming.

9. Now that you understand the dump display, let's load some data into memory that you can recognize. To do this, we'll use the **fill** command. Refer to Figure 1-18 and type the command "F100 17F FF" and RETURN. This tells the debugger to load the byte value 0FFH into 128 consecutive bytes of memory, beginning at offset 0100H and ending at offset 017FH. The segment part of the address is found in the DS register. However, at this time, the DS register contents are the same as those in the CS register. Therefore, you have just loaded 0FFH into the area you examined earlier, D100. To verify this, reexamine the area; type "D100" and RETURN. The memory area from offset 100H to 17FH should be filled with FF.

```
A:DEBUG

DEBUG  version 1.08
>D100
0A09:0100  F8 83 7E F8 FF 74 42 8B-5E F8 8A 87 1B 18 B4 00   x.~x.tB.^x....4.
0A09:0110  B1 05 D3 E0 03 06 18 17-89 46 EE C7 46 EC 00 00   1.S`.....FnGF1..
0A09:0120  FF 76 0C 9A 0A 00 1D 0D-89 DF 06 C4 76 EC B9 00   .v........_.Dv19.
0A09:0130  01 1E 06 8E 46 EA 1F FC-F2 A5 1F 59 81 46 0C 00   ....Fj.|r%.Y.F..
0A09:0140  02 FF 46 08 FF 4E 06 EB-9E 8A 5E 0A B7 00 D1 E3   ..F..N.k..^.7.Qc
0A09:0150  8B 46 06 3B 87 10 0D 76-09 B8 25 27 50 9A 0C 02   .F.;...v.8%'P...
0A09:0160  05 0B 8B 5E F4 03 5E 08-D1 E3 8B 87 0E 0F 89 46   ...^t.^.Qc.....F
0A09:0170  F6 83 F8 FF 75 20 FF 76-0C 9A 0A 00 1D 0D 89 DF   v.x.u .v.......
>F100 17F FF
>D100
0A09:0100  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0110  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0120  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0130  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0140  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0150  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0160  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0170  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
>
```

Figure 1-18
Result of filling section of memory with 0FFH.

10. Now it's time to load the program you wrote earlier. The first step is to identify the program. Refer to Figure 1-19 and type "NTEST.COM" and RETURN. This tells the debugger the **name** of the program you wish to load is TEST.COM. **Load** the program by typing "L" and RETURN. Since all COM type programs are always loaded into memory at offset 0100H, in the current code segment of memory, you can verify the program was loaded. Type "D100" and RETURN. As Figure 1-19 and your display shows, the first ten bytes of displayed memory no longer contain the value 0FFH. Instead, they contain the program machine code.

```
>NTEST.COM
>L
>D100
0A09:0100  B8 11 11 BB 55 55 03 C3-8B C8 FF FF FF FF FF FF   8..;UU.C.H......
0A09:0110  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0120  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0130  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0140  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0150  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0160  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
0A09:0170  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
>
```

Figure 1-19
Naming, loading, and verifying the program was loaded.

Recall that the program you wrote first loads the value 1111H into the AX register. Then it loads the value 5555H into the BX register. Next it adds the contents of the BX register to the contents of the AX register. Finally, it moves the contents of the AX register into the CX register. In the following steps, you will cause the MPU to execute each of these instructions. The debugger will display the results after each execution.

11. Before you execute the first instruction, you should determine the current contents of the MPU registers. Type "R" (for register) and RETURN. Figure 1-20 shows the debugger's response to the **register** command. The first line identifies the eight general registers. The second line shows the four segment registers, the Instruction Pointer register, and the status of eight of the nine flags. The Trap flag isn't shown, since the debugger considers its value insignificant for debugging purposes. Rather than show each flag and its state (for example, ZF=0), the debugger uses a simple code to conserve space. The code is given in Figure 1-21. The last line in the register display shows the address of the current instruction (code segment:offset), the instruction contents in hex machine code, and the assembly language translation of the machine code. Notice that the address of the current instruction matches the contents of the Code Segment register and the contents of the Instruction Pointer. These registers always point to the next instruction to be executed. Since no instructions have been executed, they point to the first instruction in the program. Keep in mind that the contents of your segment registers may not match the values shown in Figure 1-20.

```
>R
AX=0000  BX=0000  CX=000A  DX=0000  SP=FFF0  BP=0000  SI=0000  DI=0000
DS=0A09  ES=0A09  SS=0A09  CS=0A09  IP=0100   NV UP DI PL NZ NA PO NC
0A09:0100 B81111        MOV     AX,1111
>
```

Figure 1-20
Examining the MPU register contents with the debugger.

FLAG NAME	SET	CLEAR
Overflow	OV	NV
Direction	DN Decrementing	UP Incrementing
Interrupt	EI Enabled	DI Disabled
Sign	NG Negative	PL Plus
Zero	ZR	NZ
Auxiliary Carry	AC	NA
Parity	PE Even	PO Odd
Carry	CY	NC

Figure 1-21
Flag set and flag clear codes.

12. Now that you know the current status of the MPU, let's **trace** (single-step) through each instruction and observe the results. Figure 1-22 shows the results of each trace operation as well as the original status of the MPU registers. Type "T" and RETURN. The first instruction, MOV AX,1111H, is executed. The value 1111H is moved into the AX register. The Instruction Pointer is incremented to point to the next instruction. The third line in the display is updated to show the address and contents of the next instruction to be executed.

```
AX=0000  BX=0000  CX=000A  DX=0000  SP=FFF0  BP=0000  SI=0000  DI=0000
DS=0A09  ES=0A09  SS=0A09  CS=0A09  IP=0100   NV UP DI PL NZ NA PO NC
0A09:0100 B81111        MOV     AX,1111
>T

AX=1111  BX=0000  CX=000A  DX=0000  SP=FFF0  BP=0000  SI=0000  DI=0000
DS=0A09  ES=0A09  SS=0A09  CS=0A09  IP=0103   NV UP DI PL NZ NA PO NC
0A09:0103 BB5555        MOV     BX,5555
>T

AX=1111  BX=5555  CX=000A  DX=0000  SP=FFF0  BP=0000  SI=0000  DI=0000
DS=0A09  ES=0A09  SS=0A09  CS=0A09  IP=0106   NV UP DI PL NZ NA PO NC
0A09:0106 03C3          ADD     AX,BX
>T

AX=6666  BX=5555  CX=000A  DX=0000  SP=FFF0  BP=0000  SI=0000  DI=0000
DS=0A09  ES=0A09  SS=0A09  CS=0A09  IP=0108   NV UP DI PL NZ NA PE NC
0A09:0108 8BC8          MOV     CX,AX
>T

AX=6666  BX=5555  CX=6666  DX=0000  SP=FFF0  BP=0000  SI=0000  DI=0000
DS=0A09  ES=0A09  SS=0A09  CS=0A09  IP=010A   NV UP DI PL NZ NA PE NC
0A09:010A FFFF          ???     DI
>
```

Figure 1-22
Single-stepping through the program TEST.COM.

13. Type "T" and RETURN. The second instruction, MOV BX,5555H, is executed. Again, only two registers are affected by the operation. The value 5555H is moved into the BX register and the Instruction Pointer is incremented to point to the next instruction.

    The first two instructions moved what we call **immediate** values into the AX and BX registers of the MPU. An immediate value is a constant that forms part of the instruction. Thus, you could say the first two instructions were immediate addressing type instructions. The next instruction is called a register addressing instruction, because the value being manipulated is contained in a register. Immediate and register addressing instructions are two of seven addressing modes you will learn about in this course.

14. Type "T" and RETURN. The third instruction, ADD AX,BX, is executed. The contents of the BX register is added to the contents of the AX register (5555H plus 1111H equals 6666H) and the Instruction Pointer is incremented to point to the next instruction. Although the contents of the BX register was added to the contents of the AX register, the BX register retains its original value. This is true of any MPU register or memory location. When the contents of a register or memory location is moved, added to, or subtracted from another register or memory location, the original value is not lost or reduced to zero.

15. Type "T" and RETURN. The fourth instruction, MOV CX,AX, is executed. A copy of the contents of the AX register is moved into the CX register and the Instruction Pointer is incremented to point to the next instruction. As usual, the third line of the display shows the address and contents of the next instruction.

    Question: If the program TEST.COM has only four instructions, where did the debugger come up with a fifth?

Answer: When the debugger executes the trace command, the last step is to decode the next byte or bytes pointed to by the Instruction Pointer and determine the instruction. In most cases, it will be able to come up with some valid instruction even though the data it is examining may be garbage. In this case, the next 118 bytes of memory contain the value 0FFH. In its quest for a valid instruction, the debugger has determined that 0FFFFH could be an instruction that uses the DI register, but it isn't sure of the instruction type, hence the three question marks.

Discussion

The first part of the experiment gave you a quick example of how a program is created and stored on a magnetic disk (the stored program concept). The next part of the experiment had you move the program into memory, and then single- step through the four instructions. This gave you an opportunity to see how the MPU registers could be manipulated by a few simple instructions. We didn't use the flag registers, since they are instruction dependent and it wouldn't have been beneficial to you at this time. The last part of the experiment will show you the difference between RAM and ROM.

Procedure Continued

16. Before we play with RAM and ROM, there is one important point you should know about memory. Because of the way that memory is addressed, it forms a closed loop with itself. That is, when you get to the end of memory, the next byte you will see when you increment the address is the first byte of memory. For example, examine the last 128-byte segment of memory. Type "DFFF8:0000" and RETURN. Figure 1-23 shows the output from our microcomputer. The last byte in memory contains the value 0FEH. The value in your last byte may be different. Now if you examine the next 128 bytes, you should be looking at the first 128 bytes in memory. Type "DFFF8:0080" and RETURN. Now type "D0000:0000" and RETURN. Compare the last two displays — they are identical. The reason they are identical is because they both decode to the same address. Segment address 0FFF8H plus offset 0080H equals physical address 100000H. Since the address bus on the 8088 MPU is only 20 bits wide, the 21st bit from our addition is dropped, and the decoded address is 00000H — the first byte in memory.

```
>DFFF8:0000
FFF8:0000  20 20 00 00 00 00 1C 20-1C 02 1C 00 00 10 10 38    ..... .......8
FFF8:0010  10 10 12 0C 00 00 00 00-22 22 22 26 1A 00 00 00  ........"""&....
FFF8:0020  00 22 22 22 14 08 00 00-00 00 22 22 2A 2A 14 00  ."""......""**..
FFF8:0030  00 00 00 22 14 08 14 22-00 00 00 00 22 22 22 1E  ..."..."....""".
FFF8:0040  02 1C 00 00 00 3E 04 08-10 3E 00 00 0C 10 10 20  .....>...>.....
FFF8:0050  10 10 0C 00 00 08 08 08-00 08 08 08 00 00 18 04  ................
FFF8:0060  04 02 04 04 18 00 00 30-49 06 00 00 00 00 00 00  .......0I.......
FFF8:0070  EA 00 00 01 FE 5A EF EE-3E 07 D3 F5 3E 80 D3 FE  j...~Zon>.Su>.S~
>
```

Figure 1-23
Last 128 bytes of memory.

To see what we mean, type "D0000:FF00" and RETURN. The debugger will display memory locations 0000:FF00 through 0000:FF7F. Now type "D" and RETURN. The debugger automatically displays the **next consecutive** 128 bytes of memory (0000:FF80 through 0000:FFFF). What you now see on your display is the last 256 bytes of the memory segment that begins at segment base addresss 0000H. Type "D" and RETURN one more time. Now you can see the first 128 bytes of the segment. Once the debugger reached the end of the segment (logical address 0FFFFH), the next **logical** address was 0000H. Thus, the last debugger display began at logical address 0000H. In display-

ing the memory within the segment, the debugger effectively "wrapped-around" the end of the segment, back to the beginning. While this may not seem very significant right now, it could affect the way you handle the programs you write in the future.

17. There are no special rules concerning the placement of RAM or ROM in memory, except for this — the 8088 MPU expects to see the system "boot" address at the top of memory (FFFF:0000). Thus, whenever power is applied to the microcomputer, or after a system "reset," the MPU automatically goes to address 0FFFF0H and uses the data stored at that and the following locations to begin system start-up. For that reason, there must be at least 16 bytes of ROM at the top of memory. Normally, all of the system ROM is placed at the top of memory. Let's look at those last 16 bytes of ROM — type "DFFFF:0000" and RETURN. The first 16 bytes in the display are values stored in ROM. The next 112 bytes are values stored in RAM.

18. To demonstrate that ROM does indeed reside at the top of memory and that it is impossible to write data into ROM, while RAM is easily written to, try to modify the **last** 16 bytes of memory and the **first** 16 bytes of memory. The fill command isn't reliable in this example, so use the **examine** command instead. Type "EFFFF:0000" and RETURN. This will cause the value at address FFFF:0000 to be displayed. Type the characters "FF" and then tap the "space bar." This will cause the debugger to write FF to address FFFF:0000 and then display the value stored at the next address, FFFF:0001. Continue entering the characters "FF" and tapping the space bar, until you have made 31 entries. Now enter "FF" one more time and then tap the RETURN key instead of the space bar. Theoretically, you have entered the value 0FFH at 32 consecutive locations in memory. Examine the end and beginning of memory — type "DFFFF:0000" and RETURN. What do you see? The byte values at the end of memory have not changed; however, the byte values at the beginning of memory are now 0FFH.

The debugger doesn't know the difference between RAM and ROM. Therefore, when you attempted to change those memory locations, the debugger followed your commands. Since ROM can't be written to, the debugger had no effect on the end of memory. On the other hand, 0FFH was written to the beginning of memory, since it is composed of RAM, which is easily written to.

19. Exit the debugger — type "Q" and RETURN. This completes the Experiment for Unit 1. If you plan on using the microcomputer to run another program, "reset" the system first. You modified a number of memory locations that should contain certain preset values for program support. Resetting the system will restore those values. Proceed to the Unit 1 Examination.

UNIT 1 EXAMINATION

The purpose of this exam is to help you review the key facts in this unit. The problems are designed to test your retention and understanding by making you apply what you have learned. This exam is not so much a test as it is another learning method. Be fair to yourself and work every problem first before checking the answers.

1. Draw a block diagram of a basic microcomputer and identify the various components.

2. Name the two separate processing units within the 8088 MPU. _____ _____.

3. Draw a diagram and identify each of the 8088 MPU registers.

4. Define the term "segment." _____

_____

_____

_____

5. Segments may be adjacent to each other, separated by blocks of memory, or they may overlap. _____
True/False

6. The Code Segment register contains the value 0101H and the Instruction Pointer contains the value 0100H. What is the physical address of the next instruction to be executed? _____

EXAMINATION ANSWERS

1. Figure 1-24 is a block diagram of a basic microcomputer.

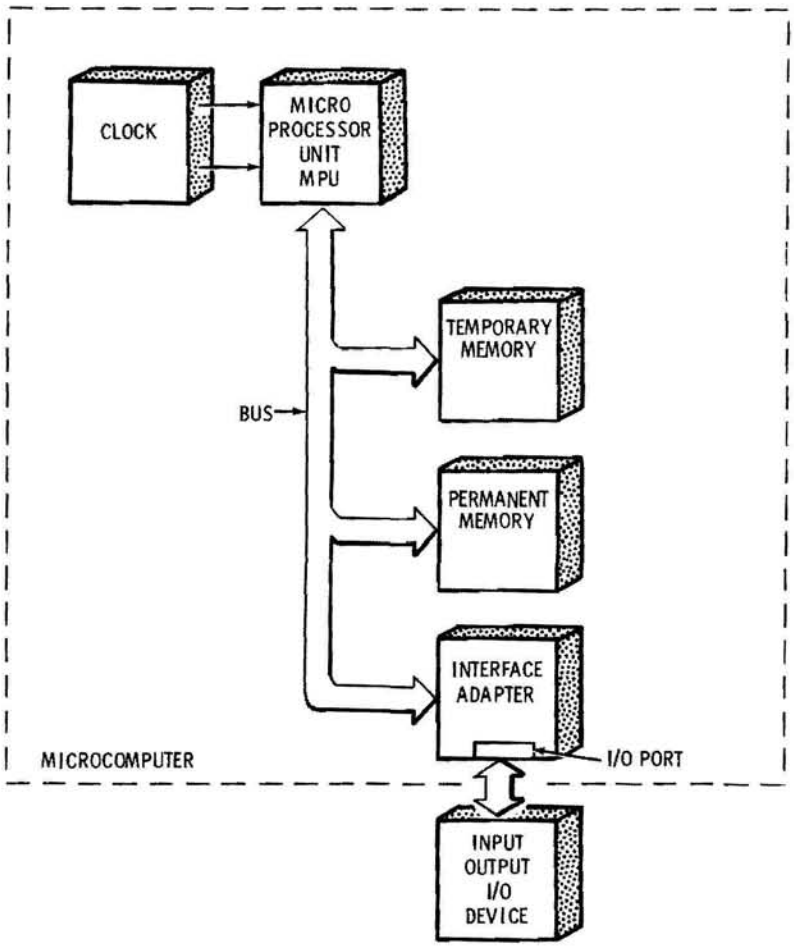


Figure 1-24
Diagram of a basic microcomputer.

2. The two separate processing units within the 8088 MPU are the **execution unit** and the **bus interface unit.**

3. Figure 1-25 is a diagram of the 8088 MPU registers.

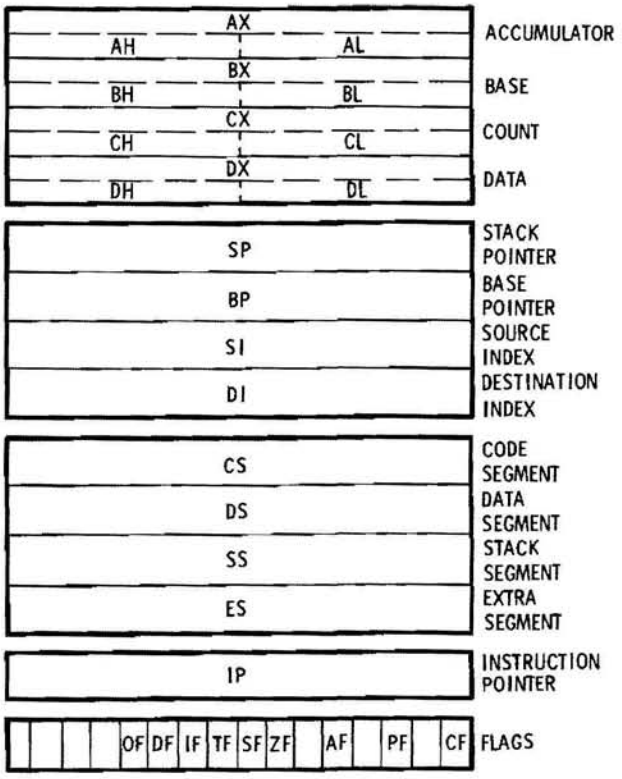


Figure 1-25
8088 MPU register set.

4. A segment is a logical unit of memory that can be up to 64K bytes long. Each segment consists of an uninterrupted block of memory locations.

5. **True**. Segments may be adjacent to each other or separated by blocks of memory. They may also overlap either partially or fully.

6. If the Code Segment register contains the value 0101H and the Instruction Pointer contains the value 0100H, then the physical address of the next instruction to be executed is **01110H**.

Segment	=	01010H
Offset	=	0100H
Total	=	01110H

SELF-REVIEW ANSWERS

1. The **microprocessor** is a complex logic element capable of performing arithmetic, logic, and control operations.

2. An MPU, along with all of the other components necessary to interface with the outside world, is called a **microcomputer**.

3. The **clock** acts as the master timer for the microcomputer.

4. **Memory** devices, depending on the type, act as either temporary or permanent storage areas for data.

5. The **bus** consists of a number of parallel conductors that link together the various components that make up the microcomputer.

6. The **interface** adapter ensures that the electrical input signals are of an amplitude and type compatible with the rest of the circuitry in the microcomputer.

7. External devices which transmit data to, or receive data from, a microcomputer are known as **I/O devices** or **peripherals**.

8. Information received from an I/O device is referred to as **input**.

9. Any information or direction from the microcomputer to the outside world is called an **output**.

10. The point at which the I/O device connects to the microcomputer is called the **I/O port**.

11. Instructions organized into a sequence in order to perform a specific task is called a **program**.

12. In the **stored program** concept, a program is permanently maintained on some type of storage medium.

13. Copying a program from a permanent storage medium into the temporary storage area in the microcomputer's memory is called **loading** a program.

14. The smallest part of a microcomputer's language, the **bit**, is a binary 1 or 0 that represents an electrical state.

15. A microprocessor's language written in electrical states is referred to as **machine code**.

16. The **byte** is a unit of information 8 bits long.

17. Each **word** used with the 8088 MPU is 16 bits long.

18. The eight most significant bits of a word are called the **high byte**.

19. The eight least significant bits of a word are called the **low byte**.

20. The 8088 MPU makes limited use of a 32-bit quantity called a **doubleword**.

21. The 16 most significant bits of a doubleword are called a **high word**.

22. The 16 least significant bits of a doubleword are called a **low word**.

23. The 8088 MPU consists of two separate processing units. They are the **execution unit (EU)** and the **bus interface unit (BIU)**.

24. The **data** or **general** registers are temporary storage areas where data can be held either before or after it is used in computations by the microprocessor.

25. The **arithmetic logic unit (ALU)** performs arithmetic or logic operations on the data specified in the instructions.

26. The **address generation and bus control section** performs all bus operations for the 8088 MPU.

27. The **instruction queue** is a 4-byte temporary storage area within the MPU that is used to hold instructions or data.

28. The **instruction pointer (IP)** is a 16-bit register that contains the address of the next instruction to be executed by the MPU.

29. The four data or general registers that can be used as either 8-bit or 16-bit registers are the **accumulator**, **base**, **count**, and **data** registers.

30. The **flag** register contains nine 1-bit status and control registers.

31. The **width** of memory refers to the amount of data, in bits, which can be stored in, or retrieved from, any given memory location at one time.

32. The **length** of memory is determined by the needs of the user, but limited to the addressing capability of the MPU.

33. **Read only memory (ROM)** is a permanent storage area for programs or constants that are essential for microcomputer operation.

34. **Random access memory (RAM)** acts as a temporary storage area, within the microcomputer, for programs or data.

35. A **segment** is a logical unit of memory that is 64K bytes long.

36. Each segment is assigned a **base address**, which is the starting address of the segment in memory.

37. The 8088 MPU has four segment registers. They are:

    A. CS or code segment register.
    B. DS or data segment register.
    C. SS or stack segment register.
    D. ES or extra segment register.

38. The **logical**, or **offset**, address is the address of an instruction or piece of data within a 64K block of memory.

39. The address generation and bus control section shifts the segment base register contents four bit positions to the left. This results in a 20-bit segment base value in which the four least significant bits are zeros. The logical, or offset, address is then added to this value to form the physical address.

40. **True**. The four segments can be located anywhere in memory.

41. **False**. The segments may be adjacent to each other, and they may overlap.

42. **False**. The MPU "assumes" that instruction code is located in the code segment, variables are located in the data segment, the stack is located in the stack segment, and the extra segment is used for "special operations."

43. In the case of the code segment, the logical address is obtained from the instruction pointer.

44. **True**. The logical address for determining the physical address in the data segment is not stored in a register. It is obtained from the effective address determined by the operand of an instruction.

45. In the stack segment, the physical address can be obtained by adding the offset value in the SP register or the value in the BP register to the shifted contents of the SS register.

46. The default register for the logical, or offset, address in the extra segment is the **DI** register.

47. A group of four clock cycles is called a **bus cycle**.

48. **True**. The 8088 MPU can fetch and execute instructions at the same time, because the 8088 MPU consists of two independently operating units: the EU and the BIU.

49. If the MPU clock is operating at 4 MHz, the MPU can effectively fetch **1,000,000** instructions per second.

50. The **instruction** is used by the programmer to "tell" the computer what operation it is to perform.

51. A microcomputer's vocabulary consists of a group of instructions called an **instruction set.**

52. The **opcode** portion of the instruction tells the microcomputer what type of operation to perform.

53. The **destination** operand is a location, either in a register or in memory, which holds the results of a given operation.

54. The **source** operand contains the data upon which an operation is performed.

55. An **immediate** operand is an actual word or byte of data.

INSERT

Unit 2

INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

CONTENTS

INTRODUCTION

The first Unit introduced you to the 8088 MPU and the microcomputer environment in which it operates. While the major concepts were presented in that unit, a number of items were covered in a general fashion and a few were completely ignored. It was done this way because these items were strictly hardware related or more easily understood from a programming point of view. As you progress through the course, the items that are programming related will be introduced or expanded in concept.

This unit introduces you to assembly language programming. First, it will give you a general idea of what you are creating when you write and process an assembly language program. Next, it describes what makes up the assembly language program: instruction format, data allocation, assembler directives and expressions, and operand typing. These items cover the fundamentals only — enough to let you start writing programs. The third section in this unit describes program segmentation and its relation to microcomputer memory. Finally, you will be taken, in a step-by-step manner, through the process of assembling and linking a program.

Remember that assembly language programming demands that you be precise in the manner in which you structure the language. The smallest mistake can cause the assembler to misinterpret or reject the program. (You may have experienced that problem in the last experiment.) Therefore, it is important that you learn the material in this unit. If the Self-Review at the end of a section indicates that you are weak in a certain area, review that subject.

NOTE: If you have taken the Heath/Zenith Educational Systems course Advanced Microprocessors, EE-8088, you will notice a similarity between that course and this course. That is to be expected, since both courses deal with the same microprocessor, and by default, the same basic instruction set. However, the assembly language used in Advanced Microprocessors is very different in concept from MACRO-86.

Use the "Unit Objectives" that follow to evaluate your progress. When you successfully accomplish all of the objectives, you will have completed this Unit. You can use the "Unit Activity Guide" to help you maintain a record of the sections that you complete.

UNIT OBJECTIVES

When you complete this Unit, you will be able to:

1. Define the following terms: assembly language, assembler, object code, linker, COM, EXE, delimiter, symbol, label, name, expression, operator, and directive.

2. Define an instruction statement and a directive statement, name the fields and their delimiters that make up the statements, and state the possible contents of these fields.

3. State the four steps in creating a COM program.

4. State the characters that can be used in an assembly language label or name.

5. State the purpose and use of the ORG, EQU, SEGMENT, ENDS, ASSUME, END, DB, DW, TITLE, +, −, *, /, MOD, SHL, SHR, DUP, ?, and RADIX assembler directives and operators.

6. Use the ADD, SUB, INC, DEC, and MOV instructions in a program.

UNIT ACTIVITY GUIDE

	Completion Time
☐ Read the Section on "Assembly Process Overview."	_____
☐ Complete Self-Review Questions 1-8.	_____
☐ Read the Section on "Elements of Microsoft Assembly Language."	_____
☐ Complete Self-Review Questions 9-26.	_____
☐ Read the Section on "Operand Typing."	_____
☐ Complete Self-Review Questions 27-40.	_____
☐ Read the Section on "Arithmetic Operators."	_____
☐ Complete Self-Review Questions 41-49.	_____
☐ Read the Section on "Program Segmentation."	_____
☐ Complete Self-Review Questions 50-60.	_____
☐ Perform the Experiment.	_____
☐ Complete the Unit 2 Examination.	_____
☐ Check the Examination Answers.	_____

ASSEMBLY PROCESS OVERVIEW

In Unit 1, you learned that the MPU only understands machine language (logical ones and zeros). However, writing a useful program in machine language can be a very tedious operation, and if you make a mistake, finding the error is nearly impossible. The Experiment had you write a simple program that could be used by the debugger to illustrate the operation of the MPU and memory. The program was written in **assembly language** — the next step up from machine language.

A more human oriented method of communicating with the microcomputer, assembly language uses **symbolic notation**, rather than ones and zeros, to structure the program code. Basically, this notation consists of two types of statements. These are the instruction statement and the assembler directive statement. Figure 2-1 shows the program you wrote in the first Experiment. Lines 1, 2, 3, 8, and 9 contain assembler directive statements, while lines 4, 5, 6, and 7 contain instruction statements.

```
A:EDLIN TEST.ASM

EDLIN  version 1.02
New file
*I
     1:*TEST    SEGMENT
     2:*        ASSUME  CS:TEST,DS:TEST,SS:TEST,ES:TEST
     3:*        ORG     100H
     4:*START:  MOV     AX,1111H
     5:*        MOV     BX,5555H
     6:*        ADD     AX,BX
     7:*        MOV     CX,AX
     8:*TEST    ENDS
     9:*        END     START
    10:*^C

*E

A:
```

Figure 2-1
Program from Unit 1 experiment.

As the name implies, the **assembler directive statement** tells the assembler what to do with the instruction statements. These directives do not become part of the assembled program. Only the instruction statements and any supporting data comprise the final program. Let's quickly review how a program is created. Later, we'll get into the nuts and bolts of the process.

The Editor

It's a common misconception that you write assembly language programs with the assembler. That is not the case, as you learned in the experiment in Unit 1. The **editor program** is used to generate the assembler directive and instruction statements called a **source program**, **source listing**, or **source code**. It is the source program that is processed by the assembler program.

The program EDLIN.COM is a very simple editor that is supplied with MS-DOS. Since that is the only editor we can be sure you have, all of our example programs will be written with EDLIN. If you decide to use a more sophisticated editor, make sure it is compatible with the MACRO-86 assembler. Some editors add hidden control characters to the source code that cannot be translated by the assembler, and thus, will generate an assembly error.

The Assembler

Once the source program is written, the next step is to translate it into machine executable code. This is actually a two-step process. The first step is handled by the **assembly language program** MASM.EXE. It takes the alphanumeric code in the source program and converts it into the binary code described in Unit 1. This produces what we call the **object file**.

Recall from the experiment that the source program that you wrote was called TEST.ASM. The name of the program is TEST. The three letters following the name are called the **file extension**. They help you and the microcomputer identify the program type. The file extension for your source program is ASM. This identifies the program as a source listing. Another common term for this type of program is **assembly language listing**, that is, a program that can be processed by an assembler.

After the assembler translated the program TEST.ASM, in the experiment, it produced a second program called TEST.OBJ. The file extension OBJ identifies the program as an **object file**. The object file is essentially a machine executable program. However, there are a few "housekeeping" chores that must be handled before the MPU can run the program. This forms the second part of the two-step source program translation.

The Linker

The second step in translating a source program into a machine executable program is the linking process. The **linker program**, LINK.EXE, takes the object file and joins the several parts and routines into an **executable file** with an EXE extension. During the linking operation, all program segmentation is resolved, and the various parts of a segment that were specified in separate places in the source code are joined into consecutive locations in the EXE file.

While linking is considered the final step for producing a **run file**, a program that can be executed by a microprocessor, not all object files produce a usable run file after linking. An EXE file contains, by definition, a specifically identifiable **stack segment** and completely **relocatable code**. This means that a specific data storage area called the stack must be established through an assembler directive, and the program instructions and data must be structured so that they can be loaded anywhere within memory. The program you wrote didn't meet these requirements. Therefore, it was necessary to perform one more processing step — conversion to a command file form.

EXE-To-Binary Conversion

The EXE2BIN.COM program is an **EXE-to-binary conversion program** that is used to convert some programs with an EXE extension to an equivalent program with a COM extension. The COM extension stands for **command file code**.

Historically, programs for microcomputers were relatively simple constructions and made to reside in and use only 64K bytes of memory. This is the basis for the COM program. The microcomputers that use MS-DOS are capable of handling much more memory. However, the structure of the code is different from that of the COM program. The EXE2BIN program makes it possible to convert a simpler COM-type program into a form that can be used with MS-DOS.

Every program is first converted into the EXE format by the linker. However, unless the original ASM file was written to produce an EXE file, this EXE file cannot be executed as such. When you write an assembly language program that will become an EXE file, you must incorporate several characteristics. Likewise, if the program is going to become a COM file, there are other characteristics you must include. Figure 2-2 is a listing of the most common considerations made when preparing EXE and COM programs. These are not all of the considerations, but they are the significant differences between the two types of programs. While some of the terms may not be familiar, you will learn what they mean.

.COM	Program can occupy only a single segment, and that must be less than or equal to 64 K bytes.
	You cannot specify a stack segment and pointer.
	The program must be ORGed at 100H.
	You must provide an END <label> statement at the end of the program. The label will specify the programs starting point.
.EXE	The program will be multisegment, normally less than or equal to 384 K bytes.
	The stack pointer will be set to the stack segment automatically.
	You do not need an ORG statement.
	You use an END <label> statement only for the main module's starting segment address.

Figure 2-2
EXE and COM source code considerations.

Self-Review Questions

1. Assembler notation consists of two types of statements: assembler _____ and _____.

2. Assembly language programs are written using _____ or some other text editor.

3. The program in text form is called the _____ code.

4. MASM.EXE is used to translate the program into _____ code.

5. The file extension for an assembly language program is _____.

6. The _____ is used to convert object code to executable code.

7. The program _____ is used to convert an EXE file to a COM file.

8. The _____ file is limited in size to 64K bytes.

NOTE: The Self-Review Answers are located at the end of this unit, beginning on Page 2-77.

ELEMENTS OF MICROSOFT ASSEMBLY LANGUAGE

When you write a microcomputer program, you prepare a list of instructions the microcomputer must perform in order to do the required work. Before the microcomputer can do the work, it must read and translate these instructions, as we described earlier. Now the lowest level of microcomputer control is provided by **binary codes** that actually turn on and off the electrical circuits in the MPU. This lowest level of code is called **machine code**, or **machine language**. One level above that, and closely related to it is **assembly language**. This language defines each machine code with a **mnemonic**, which is the text abbreviation for that specific machine operation. In addition to the MPU controlling instructions, your assembly language program contains a number of assembler directive statements. These tell the assembler how to structure the instructions and supporting data. This section teaches you the proper format for the MACRO-86 assembler instruction and directive statements.

The Assembly Language Instruction Statement

Figure 2-3 shows a typical assembly language instruction statement. The instruction is written in a specific format so that it can be properly translated by the assembler. The instruction statement can be broken down into smaller sections called fields. Each of these **fields** contains some information about the instruction.

```
LABEL   OPCODE   OPERANDS   COMMENTS

START:  MOV      AX, 0A3DH  ; Store immediate value in accumulator
```

Figure 2-3
The instruction statement.

The first field of the instruction is called the label. **Labels** are symbolic addresses that are used by MACRO-86 to identify specific areas in a program. They can be used to reference procedures within or outside the 64K segment where they are defined. While a label can be assigned to any instruction, it is not necessary to put a label on every instruction.

The second field in the instruction, called the **opcode field**, is occupied by the opcode's **mnemonic**. Simply stated, a mnemonic is an abbreviated version of the stated MPU operation. For instance, the mnemonic for halt is HLT; the one for the move operation is MOV; and the one for subtract with borrow is SBB. Every MPU operation has a specific mnemonic. A complete list of the 8088 MPU mnemonics, called an **Instruction Set**, is provided in Appendix D.

The next field of the instruction is the operand, or operands, field. Recall that the **operands** contain the data, or indicate the location of the data, to be operated upon. The number of operands in an instruction depends on the type of operation being performed. A move (MOV) instruction would contain two operands: the destination operand followed by the source operand. An increment (INC) instruction would contain just one operand: the item being incremented. This could be the contents of a register or the contents of a memory location. Some instructions require no operand. Clear Carry flag (CLC) is such an instruction.

The last field in the instruction is the **comments field**. Here, you can state any needed information about the instruction. The **comments** are used to document the program as you write it. In a very short program, the need for documentation is not immediately evident. Once you start writing longer programs, however, you will find that it is very easy to forget why you did a certain procedure in a certain way. Comments also help others to understand your programs.

Of course, you can't just run all these fields together on a program line and expect the assembler to make sense of them. There are certain characters that are used to indicate the beginning and end of the different fields. The word **delimiter** is used to describe a character that marks the **limits** of a field. Let's look at the fields of the sample instruction shown in Figure 2-3 and see exactly how delimiters are used.

The beginning of the label field is marked by the first letter or character of the assigned label. The end of the label field is marked by a colon. The label can be of any length up to the limit of the physical line. However, the assembler will only use the first 31 characters of the label to identify the symbolic address.

The legal characters for a label are:

A-Z 0-9 ? @ _ $

Only the numerals 0-9 cannot appear as the first character of a label. Lower case letters can also be used, but the assembler will treat them as upper case letters. Therefore, you must never try to distinguish between two otherwise identical labels by using upper and lower case letters.

Because a "space" is not a legal character, the "_" character is often used in its place. For instance, the label NUMONE can be written NUM_ONE. This makes it easier to read. However, be sure you don't use a reserved word for a label. A **reserved word** is any word used by the assembler to identify registers, directives, instructions, or operations. For example, the words SEGMENT, ADD, INC, and AX are considered reserved words.

Normally, the label is on the same line as the instruction it is identifying. However, it can be placed on the line preceding the instruction. This is especially useful if you are using a long label. If a label is not used, the field can be left blank.

The beginning of the next field, the opcode field, uses the end delimiter of the label field as its beginning delimiter. The end of this field is indicated by a space character. The space character, which indicates the end of the opcode field, also marks the beginning of the operands field.

If an instruction uses two operands, they are separated by a comma. The comma denotes the end of the destination operand and the beginning of the source operand. The end delimiter of the operand field is a carriage return if no comments are to be used on the program line.

If you wish to use a comment on a particular program line, the beginning of the comment is indicated by a semicolon. All information after the semicolon is considered as a comment. If a comment runs longer than the space to the end of the line allows, it may be continued on the next line as long as you begin the line with a semicolon. The end of the comment field, as well as the end of the program line, is marked by the carriage return.

In order to make your programs more readable, you can insert multiple spaces or tabs anywhere that a space, colon, or semicolon is used as a delimiter. A few program lines that look like this:

```
NUMBER:MOV AX,CX;MOVE REGISTER CONTENTS
ADD BX,0FF3AH;ADD IMMEDIATE TO REGISTER
```

can be rewritten to look like this:

```
NUMBER:     MOV AX,CX          ;MOVE REGISTER CONTENTS
            ADD BX,0FF3AH      ;ADD IMMEDIATE TO REGISTER
```

As you can see, this greatly improves the appearance and readability of the program.

The Assembler Directive Statement

Thus far, we have talked about the instructions that the MPU executes. However, there are times when you want to pass on some information to the assembler such as where to store certain information in memory. When this happens, you must use an "instruction" that is intended for the assembler rather than the MPU. These "instructions" to the assembler are called **assembler directive statements**, or **assembler directives**.

Although assembler directives are included in the source program which you write, they are never translated into machine language. After all, assembler directives are meant strictly for the assembler. The MPU could not understand them anyway.

Assembler directives are constructed in much the same way as instructions. Each assembler directive consists of a number of fields; however, these fields differ somewhat in their content and purpose from the fields of the instruction.

Figure 2-4 shows the fields of a typical assembler directive. The first field is the **name field**. Some assembler directives require that a **name** be specified in this field. With other directives, a name field is optional.

```
NAME    DIRECTIVE    ARGUMENT    COMMENTS

COUNT   EQU          0F3H        ; Set COUNT value
```

Figure 2-4
The assembler directive statement.

A name can assume one of two attributes: **variable** or symbol. A variable represents an address offset where a specified value may be found. As the name implies, the value stored at the variable "address" can be changed during program run-time. **Symbols**, on the other hand, are used to represent constants that are established at the time the program is assembled.

The next field of the assembler directive contains the **directive**, or **pseudo-opcode**. Like the mnemonics used in instructions, the directives are shortened versions of a stated operation. For instance, the form of the equate directive is EQU.

The third field of the assembler directive contains the argument. The **argument** can be either a constant expression, a variable expression, an address expression, or a character. The contents of this field are, for a large part, dictated by the directive statement.

The last field in an assembler directive is the **comments field**. This field is used for exactly the same purpose as the comments field in the instruction statement.

The delimiters used in an assembler directive are slightly different from those used with an instruction. The name field uses the same characters and follows the same rules as the label field. However, the end delimiter is not a colon. The end delimiter is a space character. The end delimiter for the directive field is also a space character.

The argument field ends with a carriage return if there are no comments, or a semicolon, if the comments field is used. Naturally the comments field is ended with a carriage return.

Assembler Directives

MACRO-86 uses many assembler directives to control the translation of instruction code. There are so many, in fact, that it isn't practical to cover them all at once. Therefore, we will introduce you to a core of directives at this time, and then later, present the rest when their use is required or more easily understood.

ORG

The **origin directive** allows you to specify an **offset address value** in a program. For instance,

```
ORG  100H
```

dictates that the next instruction is located at memory offset address 100H.

Generally, it is considered poor programming practice to use the ORG directive in a program. However, there is one time when you are compelled to use the ORG directive. Command, or COM, programs must begin at offset address 100H. MS-DOS uses those first 100H bytes for "housekeeping" and program handling. Therefore, before you write any code in a program that will be translated into a COM file, you must use the ORG 100H assembler directive statement.

EQU

The **equate directive** allows you to associate a symbolic name to a constant value or another symbolic name. The symbol itself is a word or alphnumeric representation that you want to use in your program. While it may not seem too important in a short program, consider what could happen if you used a value ten times in a 1000-line program. If, at some later time, you had to change the value, you might have a hard time finding all of its occurrences. However, had you equated that value to a symbolic name, you would have to make only one change – the equate directive value.

Let's see how that works. If you wanted to have the symbol FREEZE equated to the value 32, you would write the directive as:

```
FREEZE    EQU     32
```

Now if you use the symbol FREEZE in your program, the assembler converts it to the value 32.

We also said you could associate one symbol with another. If for some reason you wish to equate the symbol MELT with the symbol FREEZE, you simply write the directive as:

```
MELT      EQU     FREEZE
```

Now both the MELT and the FREEZE symbols are associated with the value 32. Thus, the instruction

```
ADD       CX,FREEZE
```

or

```
ADD       CX,MELT
```

would give the same results as the instruction

```
ADD       CX,32
```

DB

The **define byte directive** is used to allocate memory in byte-sized units. The general format for this directive is

```
<var-name>    DB      <exp>[,<exp>,...]
```

where <var-name> is a legal variable name assigned to the define byte statement, DB is the define byte directive, and <exp> is the value expression for the defined byte of memory. Notice that the define byte directive can associate one or more consecutive memory locations to a variable name, by separating each expression with a comma. Naturally, you are limited by the physical length of the statement line as to the number of value expressions associated with the variable name. However, you can append additional expressions by repeating the define byte directive without the variable name. For example,

```
PRODUCT     DB      85,0FEH,34H,55,45,0FFH,0A4H,22H,44
            DB      0ABH,0E4H,35H,147,23,100,POUND
```

defines 16 consecutive bytes in memory and associates these bytes to the variable name PRODUCT. We call this a table of byte-sized values. The variable name PRODUCT is used to identify the offset address of the first byte in the define byte statement. Thus, PRODUCT points to the beginning of the table.

Notice that all of the define byte value expressions are byte-sized. That is, each value is 255 or less. When a symbol is used as a value expression, it must be equated to a byte-sized value. For that reason, the symbol POUND, the last value expression in our example, must have been equated earlier to a byte-sized value.

All of the values in the table PRODUCT are defined at assembly time. Therefore, when the program is executed, the values are stored in memory beginning at the physical address identified by the base value in the Data Segment register and the offset value assigned to the variable name PRODUCT. During program execution, these values can then be read or modified by the program instructions.

Often, there is a need to initialize a block of predefined byte-sized values. A variation of the define byte directive statement lets you reserve and identify such an area in memory. A general form of the statement is

```
<var-name>    DB      <exp> DUP (<exp>)
```

where <var-name> is again the statement offset address, and DB is the directive. The argument in this statement is defined in this manner: The first <exp> is a constant or symbol other than zero that specifies the number of bytes in the block of memory. The operator **DUP** tells the assembler that the expression enclosed in parentheses is to be duplicated in each memory location. The expression enclosed in parentheses can be a byte-sized constant or symbol. For example,

```
PRODUCT         DB      20 DUP (33)
```

tells the assembler that a 20-byte block of memory is to be initialized with the value 33 in each byte.

If you wish to initialize a block of memory, but you don't want to predefine the contents of that memory, then you would use the expression?. For example,

```
PRODUCT         DB      20 DUP (?)
```

sets aside a 20-byte block of memory, but does not change the contents of that memory.

One other type of data can be stored in memory using the define byte directive — text characters. Have you ever wondered how the microcomputer is able to display text? The text is stored in memory in a coded form, and when needed, translated and displayed. The code used to store the text, and many other characters and code for that matter, is called **ASCII** (American Standard Code for Information Interchange). Figures 2-5A and 2-5B show the code for the "control" commands, alphanumeric characters, and special characters supported by ASCII; and describe the commands supported by the code.

ROW	COLUMN	0 (3)	1 (3)	2 (3)	3	4	5	6	7 (3)
	BITS 765 / 4321	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
10	1010	LF	SUB	*	:	J	Z	j	z
11	1011	VT	ESC	+	;	K	[	k	{
12	1100	FF	FS	,	<	L	\	l	|
13	1101	CR	GS	-	=	M	]	m	}
14	1110	SO	RS	.	>	N	^ (1)	n	~
15	1111	SI	US	/	?	O	_ (2)	o	DEL

Figure 2-5A
Table of 7-bit American Standard Code for Information Interchange.

NOTES:

(1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.

(2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.

(3) Explanation of special control functions in columns 0, 1, 2, and 7.

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell (audible signal)	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation (punched card skip)	EM	End of Medium
		SUB	Substitute
LF	Line Feed	ESC	Escape
VT	Vertical Tabulation	FS	File Separator
FF	Form Feed	GS	Group Separator
CR	Carriage Return	RS	Record Separator
SO	Shift Out	US	Unit Separator
SI	Shift In	DEL	Delete
SP	Space (blank)		

Figure 2-5B
Continuation of the table of 7-bit American Standard Code for Information Interchange.

Notice that it takes seven bits to define one of the characters. For example, the ASCII code for the letter F is 1000110B. As a general rule of thumb, the eighth bit is considered zero. Therefore, to store the ASCII code for the letter F in memory, you would use the define byte directive:

```
LETTER      DB      01000110B
```

or

```
LETTER      DB      46H
```

Looking-up the ASCII code for every letter in a message can be tedious. To save time, you can have the assembler look up the proper code. This is accomplished by enclosing the ASCII character within apostrophes or quotation marks. Thus, the define byte directive can be rewritten:

```
LETTER      DB      'F'
```

or

```
LETTER      DB      "F"
```

Although you can use either apostrophes or quotation marks, you cannot mix them in a directive statement. The statement

```
LETTER      DB      "F'
```

will generate an assembly error.

Another feature of the assembler's handling of ASCII characters, is its ability to generate the ASCII code from a character string. A **character string** is a group of characters enclosed within apostrophes or quotation marks. The define byte directive allows you to write a string of characters up to the length of the physical line. To continue the string, simply write another statement on the next line as follows:

```
COMMENT     DB      'This is an example of a character'
            DB      'string.'
```

When assembled, each character is assigned a consecutive byte in memory, beginning at the address offset assigned to the variable name COMMENT.

DW

The **define word** assembler directive is used to allocate memory in word-sized units. The general format for the directive is

```
<var-name>    DW      <exp>[,<exp>,...]
```

where <var-name> is again a legal variable name assigned to the define word statement, DW is the define word directive, and <exp> is the word-sized value expression for the defined word of memory. One or more expressions can be assigned to a variable name, with each expression separated by a comma.

The primary difference between the define byte and the define word directives is the way the assembler treats the memory variable. Since all memory locations are byte-sized, each defined byte value occupies one memory location. Each defined word obviously can't occupy one memory location. Therefore, the assembler assigns two consecutive memory locations to each defined word. For example, the instruction

```
NUM_ONE         DW      45F3H
```

places the value 45F3H into the two memory locations assigned to the variable name NUM_ONE — F3 in the low byte and 45 in the high byte. This is an important concept to remember. The low byte of the word is always stored in the low memory location, while the high byte is always stored in the high memory location. Another way to look at this is, the address pointed to by the variable NUM_ONE contains the value 0F3H, while the address pointed to by the variable NUM_ONE + 1 contains the value 45H.

You can also store the code for ASCII characters with the define word directive using any of the following methods:

```
DW      'T'
DW      'TE'
DW      'TE','ST','T','HE','W','AT','ER'
```

The first statement stores the ASCII code for the character T in the first memory location and zero in the next memory location. The second statement stores the ASCII code for the character T in the first memory location and the ASCII code for the character E in the second memory location. The third statement stores the ASCII code for each character in consecutive memory locations, with the low character byte in each

expression preceding the high character byte. You cannot use the define word directive to store a character string. Character strings can only be stored using the define byte directive. The only way to store a string of characters is by the method shown in the third statement. The characters must be stored a word-sized expression at a time.

You can also initialize a block of word-sized memory locations using the DUP operator. For example,

```
STORE          DW       20 DUP (34D7H)
```

tells the assembler that a 20-word block of memory is to be initialized with the value 34D7H in each word. As explained earlier, D7 will be stored in the first, or low, byte of the memory word and 34 in the second, or high, byte.

Keep in mind, there is a limit to the size value that can be stored in a word of memory. That limit is 0FFFFH, 65535, or two ASCII character codes.

END

The last directive used in a program is the **END** directive. It tells the assembler that the program is complete. A typical END directive statement would be

```
END       <exp>
```

where <exp> identifies the starting address of the program. The first instruction in the program must contain a label that matches the END directive expression. For example:

```
START:      ADD     AX,58E3H      ;First program instruction
              .
              .
              .
            END     START         ;Program ends here
```

The END directive expression START tells the assembler that the program begins at the address offset assigned to the label START.

Self-Review Questions

9. The assembly language instruction statement consists of four fields. These are:

   A. _____
   B. _____
   C. _____
   D. _____

10. Characters that are used to indicate the beginning and end of the different fields in an instruction are called _____.

11. Another term for opcode in an instruction is _____.

12. An instruction statement can contain one, two, or three operands.

    _____
    True/False

13. In addition to the letters A through Z and the numerals 0 through 9, the legal characters in a label or name include:

    A. _____
    B. _____
    C. _____
    D. _____

14. The assembler will recognize labels and names up to _____ characters long.

15. The assembler directive statement consists of four fields. They are:

    A. _____
    B. _____
    C. _____
    D. _____

16. Assembler directive statements are never translated into object code. _____
    True/False

17. A name can assume one of two attributes: _____ or symbol.

18. Symbols represent _____ .

19. The end delimiter for a name is a colon. _____
True/False

20. The _____ directive allows you to specify an address offset value in a program.

21. The _____ directive allows you to associate a symbolic name to a constant or another symbol.

22. The _____ directive is used to allocate memory units to data storage.

23. The physical address of a variable is determined by adding the variable offset to the shifted contents of the _____ _____ register.

24. The operator _____ is used to initialize a block of predefined memory.

25. If the address offset assigned to the variable PRODUCT is 0100H, the directive statement

```
PRODUCT    DW      0E5A8H
```

will load the value _____ at address offset 0100H and the value _____ at address offset 0101H.

26. The _____ directive is used to tell the assembler that the program is complete.

OPERAND TYPING

Recall that operand types fall into three general categories: immediate, register, and memory. These categories determine the method of addressing used by the instruction. With few exceptions, every 8088 MPU instruction uses some form of addressing. We call those exceptions **inherent**, or **processor control** instructions; that is, instructions that affect the basic operation of the MPU. Examples of inherent instructions are: halt (HLT), stop the MPU; and no operation (NOP), execute a dummy instruction to waste time. Inherent instructions, because they are MPU operation related, require no operands. Let's examine the instructions that use operands, or at least, imply an operand in the instruction.

Immediate Operands

Within the instruction, operands indicate the source and the destination of the specified operation. From that perspective, an immediate operand is always a source operand. As the **source**, it supplies a constant value that is entered when the instruction is written. The value may be either a data item or a symbol.

The default input value **radix** (number base) is decimal. Thus, any numeric value entered without a base notation will be treated as a decimal value. Naturally MACRO-86 also recognizes values in forms other than decimal when the base notation is appended to the value. For example,

11011010B	Binary value
0F4EH	Hexadecimal value
3509	Decimal value (default)
4690D	Decimal value
473Q	Octal value (letter Q)
372O	Octal value (letter O)

are the different value types that can be used with MACRO-86. Notice that when the default value radix is not decimal, decimal values should be identified with the base notation D.

If you wish to change the default radix, use the assembler directive

```
.RADIX    <exp>
```

where <exp> is the decimal value 2, 8, 10, or 16. The two move instructions

```
MOV       BX,0FFH
.RADIX    16
MOV       BX,0FF
```

are acceptable to the assembler, since the first was written when the default radix was ten, and the second was written after the radix was changed to sixteen. Remember, the radix directive only affects the constant values specified after the directive.

There is, however, one exception to this rule. The default radix in a define directive is **always** decimal, regardless of the radix specified for the program code.

Another acceptable immediate data item is an ASCII character for a byte-sized value or two ASCII characters for a word-sized value. For example, '#' is translated by the assembler to 00100011B (23H), while '##' is translated to 0010001100100011B (2323H). The apostrophes or quotation marks enclosing the ASCII character(s) forces the assembler to generate the appropriate ASCII code, one character per byte.

Symbols, as you will recall, are simply names equated to a numeric constant. Using a symbol as an immediate value is the same as using a numeric constant. Just remember to define the symbol before you use it as an immediate value.

Since an immediate operand always assumes the position of the **source operand** in the instruction, the **destination operand**, by default, must be either a register or memory operand. Therefore, the instructions

```
NUM1:     MOV       AX,DATA
          and
          MOV       PRODUCT,0F3E5H
```

are both considered immediate operand instructions even though the destination for the operation is a register or a memory address. The first is combined with a register operand, while the second is combined with a memory operand. In the first example, DATA is a symbol name for a constant. In the second example, PRODUCT is a variable name that represents the logical, or offset, memory address where the constant is to be stored.

Register Operands

An instruction is written in the register operand addressing mode if **both** the source and destination operands of the instruction are **registers**. For example, the instruction

```
DATA1:      ADD       DX,SI
```

is a register addressing instruction. The contents of the source operand, the SI register, is added to the destination, DX register, operand. If prior to the operation, DX equaled 1111H and SI equaled 2222H, then after the operation, DX will equal 3333H and SI will remain 2222H. Remember, the destination operand is the only operand that is modified by the instruction.

The 8088 MPU contains 14 registers. The Instruction Pointer is not directly accessible through an instruction. The rest of the registers can be operated upon by a number of different instructions. However, the segment registers and the Flag register cannot be used as an operand in arithmetic and logical instructions.

The four general registers, AX, BX, CX, and DX, are both 8-bit and 16-bit registers. Actually, the 16-bit general registers are composed of a pair of 8-bit registers, one for the low byte (bits 0-7) and one for the high byte (bits 8-15). Each of the 8-bit registers can be used independently from its mate. In this case, each 8-bit register contains bits 0-7. For example, the 16-bit accumulator register is called the AX register, while its associated 8-bit registers are called the AL (for low byte) and AH (for high byte) registers.

Although it is possible to use 8- or 16-bit registers in an instruction, you must not mix the register sizes. It's obvious that the contents of a 16-bit register can't be moved into an 8-bit register. On the other hand, you would think that it is possible to move the contents of an 8-bit register into a 16- bit register. The assembler, however, will reject such an operation. You must keep the register sizes the same.

The register addressing mode is used when it is necessary to manipulate operands within the MPU. By its very definition, this is the only possible time in which it could be used. Because the operation is performed entirely within the MPU, the execution time is extremely fast; typically 2 to 3 clock cycles. Speed of execution is the primary advantage of the register operand addressing mode.

Memory Operands

Thus far, you have learned that it is possible to retrieve immediate data and manipulate information within the MPU using register addressing. However, a program that uses only constants, and moves them about in the MPU, will probably have a very limited value. In order to accomplish any useful task with the microcomputer, you must have a way of retrieving and storing information in memory locations other than those occupied by the program. The 8088 MPU can do this in a number of ways. The one we are going to discuss at this time is direct memory addressing.

Direct memory addressing allows you to access specific areas in memory outside of the instruction stream. Our discussion of immediate addressing gave an example of direct memory addressing. The instruction

```
MOV       PRODUCT,0F3E5H
```

moves the immediate value 0F3E5H into the memory address assigned to the variable name PRODUCT. By the same token, the instruction

```
MOV       BX,PRODUCT
```

moves the contents of the memory address assigned to the variable name PRODUCT into the BX register.

Recall that the physical address of PRODUCT is composed of two address values. The first value is the logical, or offset, address for the variable PRODUCT. This is the distance in bytes of memory from the beginning of data segment. The assembler and linker determine this offset. The second address value is determined by the contents of the DS register. Thus, the physical address of PRODUCT is the contents of the DS register, shifted left four bits, added to the address offset calculated by the assembler and linker.

Just as register sizes must match in register addressing, so too must the memory location and the register or immediate value size match. You can't write a word-sized value into a byte-sized memory location. Although all memory is byte-sized, the assembler still determines how the values in memory are treated. If you assign a word value to a memory location with a define word assembler directive, the assembler treats the two addresses containing that word value as a word. It won't let you write a byte value to that location. Likewise, it won't let you move a word-sized variable into a byte- sized register.

Register sizes are fixed; either they are byte-sized or they are word-sized. Data moved into a register must match the register size. This is controlled by the MPU. Memory is physically byte-sized, but its **atributes** (byte or word size) are controlled by the assembler. If the assembler says an address variable is word-sized, the MPU will move two bytes of memory if instructed. Since the atributes of memory are controlled by the assembler, there should be a way to temporarily modify those atributes in an instruction. There is, and it's handled by an assembler operator called a **pointer** (PTR). Assume the following conditions:

```
BYTE_TAB   DB       "Now is the time for all good people"
WORD_TAB   DW       '^C',0FF3FH,25
```

The first statement defines a byte-sized table in memory containing a string of characters. The second statement defines a word-sized table in memory containing three constants. Now if you try to assemble the instructions

```
MOV        AX,BYTE_TAB
ADD        WORD_TAB,BL
```

the assembler will generate an error message stating that the operand types must match. However, if you add the operator PTR and specify the operand type, the instructions will assemble properly. For example:

```
MOV        AX,WORD PTR BYTE_TAB
ADD        BYTE PTR WORD_TAB,BL
```

The first instruction moves the contents of the first two memory locations in BYTE_TAB into the AX register. The ASCII code for the letter N (4EH) is stored in the low byte of the AX register, and the ASCII code for the letter o (6FH) is stored in the high byte. The second instruction adds the contents of the BL register to the ASCII code for the letter C (43H) stored in the first memory location in WORD_TAB. (Remember, the low byte of a word is always stored in memory before the high byte. Thus, the ASCII code for the letter C is added to the BL register rather than the ASCII code for the character ^.)

It's always a good idea to define data in the form that it will be used by the program. Where this isn't possible, use the assembler operator PTR as described. Be sure to combine the operator with the operand that must be temporarily overridden.

So far, we have described immediate data/memory operations and register/memory operations. We haven't discussed memory/memory operations. There is a good reason for that; you can't transfer data from one memory location to another. If you must perform a memory-to-memory operation, then you have to temporarily store the data in an MPU register. Thus you wind up with a memory-to-register- to-memory operation using three separate instructions.

Self-Review Questions

27. Instructions that do not use some form of addressing are called _____ instructions.

28. An immediate operand is always a _____ operand.

29. The default radix for an immediate operand constant is _____.

30. You can change the default radix with the assembler directive _____.

31. How many ASCII characters can be coded into a byte-sized memory location? _____

32. Can the register operand addressing mode contain an immediate operand? _____
Yes/No

33. Of the 14 8088 MPU registers, the _____ _____ register is not directly addressable through an instruction.

34. The four general registers that are composed of two 8-bit registers are:

    A. _____
    B. _____
    C. _____
    D. _____

35. Is the instruction

```
DATA1:      ADD       SI,BL
```

a legal operation? _____
Yes/No

36. The instruction

```
MOV     AX,PRODUCT
```

will move the offset address of the variable PRODUCT into the AX register. _____
True/False

37. If the offset address of the variable SUM is 0105H and SUM is defined as a word-sized memory location, the instruction

```
MOV     SUM,3456H
```

will move the value 34H into the memory location at offset address _____.

38. If the DS register contains the value 0D20H, then the instruction from question 37 will load the value 56H into the memory location at physical address _____.

39. The assembler operator _____ will allow the contents of two consecutive byte-sized memory locations to be loaded into a 16-bit general register.

40. Data in one memory location can be transferred directly to another memory location. _____
True/False

ARITHMETIC OPERATORS

Arithmetic operators are used to combine elements of an expression to produce a single expression. For example, the instruction

```
MOV     AL, PRODUCT+2    ;Source operand is offset plus 2
```

is a direct memory addressing instruction that moves the contents of the third byte in the data block, identified by the variable PRODUCT, into the AL register. The operator in this example is the plus sign. It tells the assembler to add the value 2 to the offset address assigned to the variable PRODUCT, to create a new offset address.

The arithmetic operators for MACRO-86 are:

+	Add
–	Subtract or Negate
*	Multiply
/	Divide
MOD	Modulo
SHR	Shift Right
SHL	Shift Left

The first four operators act like their corresponding math functions. However, the divide operator returns only the quotient. To determine the remainder from a division operation, use the MOD operator. The two shift operators are used to shift the binary value right or left the specified number of bits. Let's see how each operator can be used. In the following discussion, the symbol "FREEZE" represents an **equated** value, while the symbols "PRODUCT" and "SUM" represent the **name** of a data address offset.

```
MOV     AX, 16+34
MOV     AX, 16+FREEZE
MOV     AX, PRODUCT+5
```

The **add** (+) operator in the first example causes decimal 50 to be moved into the AX register. The second example moves the sum of the constant 16 and the equated value of the symbol FREEZE into the AX register. The last example moves the data stored in the memory location five bytes past the offset address assigned to PRODUCT into the AX register. If the offset is 0103H, then the memory location is 0103H + 5H = 0108H. The add (+) operator lets you combine constants and/or symbols to produce a constant. It also lets you combine a constant or symbol with a variable to produce a new variable. It will not let you combine two variables. One final point — be sure that the sum from your add operation doesn't exceed the capacity of the destination operand.

```
MOV     AX,-16
MOV     AX,FREEZE-32
MOV     AX,PRODUCT-3
MOV     AX,PRODUCT-SUM
```

The **subtract** (-) operator in the first example causes the 2's complement of the constant 16 (0FFF0H) to be moved into the AX register. You can also use the subtract operator to negate symbols. The second example moves the difference between the equated value of the symbol FREEZE and the constant 32 into the AX register. As with all subtraction operations, the right value is subtracted from the left. The third example moves the data stored in the memory location three bytes in front of the offset address assigned to PRODUCT into the AX register. If the offset is 0105H, then the memory location is 0105H - 3H = 0102H. The last example subtracts the offset address value assigned to SUM from the offset address value assigned to PRODUCT and stores the difference in the AX register as a constant. As with the add (+) operator, the subtract (-) operator treats constants and symbols as constants. If the subtrahend is larger than the minuend, the difference will be stored in its 2's complement form. The subtract operator can't subtract the offset address assigned to a variable from a constant or symbol. It also can't subtract one variable from another if they aren't in the same memory segment.

```
MOV     AX,32*5
MOV     AX,FREEZE*5
```

The **multiply** (*) operator can only multiply constants and/or symbols. If FREEZE is equated to 32, then both of the above examples are identical; the value 160 is moved into the AX register. Should the product exceed the capacity of the destination, the assembler will generate an error message.

```
MOV     AX,32/5
MOV     AX,FREEZE/5
```

Just like multiply (*), the **divide** (/) operator can only operate on constants and/or symbols. Assuming FREEZE is equated to 32, then both of the above examples will move the value 6 into the AX register. The divide (/) operator ignores any remainder.

```
MOV     AX,32 MOD 5
MOV     AX,FREEZE MOD 5
```

If you are concerned about the remainder after a division operation, then use the **MOD** operator. Both of the above examples move the remainder from the division operation (2) into the AX register. Notice that there is a space preceding and following the MOD operator. These spaces must be used to make sure the assembler isn't fooled into thinking the MOD operator is a symbol. While it isn't necessary, you can precede and follow the other four operators with spaces. Quite often, that makes the operation easier to read.

The last two arithmetic operators, **shift right** (SHR) and **shift left** (SHL), give you the opportunity to rearrange the bit pattern in a byte or word of data. Again, you must use constants or symbols to identify the data and/or shift value. Following are examples of the shift operators:

```
MOV     AL,00001100B SHR 2
MOV     AX,11111100B SHR 4
MOV     AL,00001100B SHL 2
MOV     AX,11111100B SHL 10
```

Although we used binary values to show the value being shifted, you can use any numeric base; the assembler automatically converts the number to its equivalent binary bit pattern, and then shifts the pattern the specified count. The operation is evaluated: <expression> shift right or left <count>. Thus, in the first example, 00000011B (03H)is moved into the AL register. The second example causes the number 0000000000001111B (000FH) to be moved into the AX register. Two of the six one-bits are lost as the value is shifted right. In the third example, the value 00110000B (30H)is moved into the AL register. The last example moves the value 1111000000000000B (0F000H) into the AX register. Again, two one-bits are lost when they are shifted out of the usable range of the register.

Remember, 16-bit registers have a maximum capacity of 65,535 (0FFFFH) and every bit shift is equal to multiplying the value being shifted by two. 8-bit registers are a different story. If you exceed the capacity of an 8-bit register, with a shift-left operation, the assembler will first assume you are creating a 16-bit value, then it will realize its error and generate an error message.

Early versions of the IBM-PC MACRO-86 assembler handle the shift operations a little differently than specified in the "Macro Assembler" owner's manual. First, the shift-right operation always returns the value zero, whether an 8-bit or 16-bit register is involved. Second, the shift-left operation is evaluated the reverse of the expected sequence. It is evaluated: <count> shift-left <expression>. Thus, the instruction

```
MOV     AX,4 SHL 11111111B
```

moves the value 0000111111110000B (0FF0H) into the AX register. Before you use the shift arithmetic operators with IBM's MACRO-86 assembler, carefully test their response to determine whether or not they follow standard MACRO-86 convention.

The arithmetic operator expressions evaluate left to right and in a specific level of precedence. The operators: *, /, MOD, SHR, and SHL are equal in level of precedence and higher than the operators + and −. Therefore, *, /, MOD, SHR, and SHL will be evaluated before + or −. Where the level of precedence is equal, they evaluate left to right. Entries within parentheses have the highest level of precedence. For example:

```
MOV     AX,10+70 MOD FREEZE  ;FREEZE = 32, AX = 16
MOV     AX,101B SHL 2*2      ;AX = 0028H
MOV     AX,101B SHL (2*2)    ;AX = 0050H
MOV     AX,6+5*2-8/4         ;AX = 14
MOV     AX,(6+5*2-8)/4       ;AX = 2
MOV     AX,(6+5)*2-8/4       ;AX = 20
```

Self-Review Questions

41. Name the seven arithmetic operators.

    A. _____
    B. _____
    C. _____
    D. _____
    E. _____
    F. _____
    G. _____

42. The add (+) and subtract (−) operators can be used with constants, symbols, and variables. _____
    True/False

43. Subtracting a constant from a variable produces a _____.

44. Subtracting a variable from a variable produces a _____.

45. The 2's complement is produced using the _____ operator.

46. The multiply (*) and divide (/) operators can be used with constants, symbols, and variables. _____
    True/False

47. The _____ operator returns the quotient.

48. The assembler allows the shift-left operator to generate a value that exceeds the capacity of a register. _____
    True/False

49. The instruction

```
VALUE:    MOV     AL,11B SHL ((5*2-20/3) MOD 5)
```

    moves the value _____ into the AL register.

PROGRAM SEGMENTATION

One of the primary features of the 8088/8086 family of MPUs is their use of memory segments to partition a program into specific groups of program code and data. This section will show you how the assembler is directed to create these program segments.

Logical Program Segmentation

With MACRO-86, a program is organized into a series of named segments. Recall that segmentation of memory was described in the first Unit. However, in the program context, the segments are "logical" segments; that is, they are not assigned physical addresses in the assembly language program. For that reason, programs written in MACRO-86 contain what we call **relocatable code**. This means that after these programs are assembled, they can be loaded practically anywhere in memory and executed.

Naturally, once the program is loaded into memory, the segment registers must be loaded with specific values in order to produce the physical addresses used by the program. That is primarily a function of the **program loader** in MS-DOS.

Segmentation Assembler Directives

Three assembler directives are used to set up the logical segments in an assembly language program. A description of each follows:

SEGMENT — At program run-time, every instruction and data item must reside within a segment. The segment directive defines the segment beginning. A name is used to identify the segment. When the program is assembled, linked, and loaded into memory, that name is used to assign a value to one or more segment registers.

Each SEGMENT name must be unique to the program, contain legal characters, and not be a reserved word. **Reserved words** are words that are used by the assembler to identify registers, directives, or operations. For example, the words SEGMENT, AX, MOV, and ORG are considered reserved words. If you should use a reserved word by mistake, the assembler will tell you with an error message.

A program can contain one or more segments. You can group segments together before the program is assembled, or you can "link" a number of preassembled segments together with the linker.

ENDS — This directive identifies the end of the segment. The ENDS directive name must match the beginning SEGMENT name.

All data and instructions written between SEGMENT and ENDS are part of the named segment. In small programs, variables often are defined in one or two segments, stack space is allocated in another segment, and instructions are written in a third or fourth segment. It is perfectly possible, however, to write a complete program in one segment. If this is done, all the segment registers will contain the same base address. Command, or COM, programs use this method of segment arrangement. Large programs may be divided into dozens of segments.

The segment directives SEGMENT and ENDS take the form

```
<seg-name> SEGMENT
           .
           .
      ;program segment
           .
           .
<seg-name> ENDS
```

where <seg-name> is a name unique to that segment.

ASSUME — Recall that at program run-time, every memory reference requires two components in order to be physically addressed by hardware:

A. A 16-bit segment base value that must be contained in one of the four segment registers (Code, Data, Stack, or Extra), and

B. a 16-bit logical, effective, or offset address giving the offset to the memory reference from the segment base value.

The ASSUME directive builds a **symbolic link** between your assembly-time placement of instructions and data in logical segments (between SEGMENT/ENDS pairs) and the run-time event of physically addressing instructions and data in memory through segment registers. In other words, ASSUME is a "promise" to the assembler that instructions and data are run-time addressable through certain segment registers.

The assembler checks each memory instruction operand, determines which segment it is in, and which segment register contains the address of that segment. If the assumed register is the register expected by the MPU for that instruction type, then the assembler generates the instruction code in the normal manner. If, however, the MPU expects one segment register to be used, and the operand is not in the segment pointed to by that register, then the assembler automatically precedes the machine instruction with a **segment override prefix byte**. This prefix tells the MPU which segment register should be used for the base address in its address computation. (If the segment cannot be overridden, or determined, the assembler generates an error message.)

The format for the ASSUME directive is:

```
ASSUME    <seg-reg>:<seg-name>[,...]
```

ASSUME requires two arguments separated by a colon to identify a segment. The first argument selects a segment register: CS, DS, SS, or ES. The second argument specifies the segment name associated with that segment register. The argument pair can be repeated up to four times in an ASSUME directive — once for each segment register. For example:

```
TEST    SEGMENT
        ASSUME    CS:TEST,DS:TEST,SS:TEST,ES:TEST
          .
          .
    ;program segment
          .
          .
TEST    ENDS
```

The program segment is named TEST, and all four segment registers are referenced to TEST. Thus, each segment register is assumed to contain the same segment base address. Because of this, the offset address value for every instruction or data item is referenced to the same segment base value. This is the arrangement commonly found in command, or COM, programs.

At this time, you should remember that every program must contain at least one SEGMENT, ASSUME, and ENDS directive. Naturally, where there are multiple program segments, there must be at least one SEGMENT and one ENDS directive for each segment. You only need one ASSUME directive, but you can add more if you must change a segment register reference in the program.

It isn't necessary to identify a segment register if it isn't used in a program. If you don't, the assembler will assume the register doesn't exist. However, for COM programs, we recommend that you make it a habit to always identify at least the CS, DS, and SS registers rather than take a chance on forgetting one when it is needed. The ES register is seldom used in COM programs.

Initializing The Segment Registers

Earlier, we said that the segment register values are determined by the MS-DOS program loader, when the program is loaded into memory. The program loader determines what area of memory is available for program use, and then loads the program into the bottom, or low address area, of that memory. The program loader then loads the CS and IP registers with the base and offset address for the first instruction in the program.

In COM-type programs, the program loader also stores the value in the CS register into the DS, SS, and ES registers. Naturally, only those registers that are identified in the ASSUME directive are loaded. That, however, isn't the case in EXE-type programs. You, as the programmer, must load the appropriate base addresses into the DS, SS, and ES registers. Figure 2-6 is a partial listing of a program to show you how the registers should be loaded.

```
PROG_STK        SEGMENT STACK
                DW      100 DUP (?)     ;100 word stack area
PROG_STK        ENDS
PROG_DATA       SEGMENT
DATA1           DB      200 DUP (?)     ;Reserve 200 bytes memory
DATA2           DW      50 DUP (?)      ;Reserve 50 words memory
PROG_DATA       ENDS
PROG            SEGMENT
                ASSUME  CS:PROG,DS:PROG_DATA,SS:PROG_STK,ES:PROG_DATA
START:          MOV     AX,PROG_STK     ;Get stack base address
                MOV     SS,AX           ;Load in Stack register
                MOV     AX,PROG_DATA    ;Get data base address
                MOV     DS,AX           ;Load in Data register
                MOV     ES,AX           ;Load in Extra register
                .
                .
;Program instructions here
                .
                .
PROG            ENDS
                END     START
```

Figure 2-6
Partial listing of a typical EXE-type program.

The ASSUME directive indicates that all Code Segment references will be made to the segment named PROG. By the same token, Data and Extra Segment references will be made to the segment named PROG_DATA, while Stack Segment references will be made to the segment named PROG_STK. These references are established by the assembler. The assembler also calculates the length of each segment. The linker then combines the segments so that when they are loaded, they will occupy consecutive bytes in memory: code segment first, followed by the data and extra segments, and finally, the stack segment. With the program loaded into memory, each segment name is assigned a base value, referenced to the Code Segment register contents. Therefore, the first instruction in the program,

```
START:      MOV      AX, PROG_STK      ;Get stack base address
```

takes the value assigned to the name PROG_STK and moves it into the AX register. This is similar to the "move symbol name" instruction described earlier. PROG_STK is a symbol representing the base address of a program segment.

Notice that the value is not moved directly into the SS register. The process takes two instructions. First, the value is moved into the AX register; and then, the value is moved from the AX register into the SS register. This is necessary since there is no instruction that allows a direct move into a segment register.

The process is repeated for the Data and Extra Segment registers. First the base address for the segment assigned to data is moved into the AX register. Then, the contents of the AX register is moved into the Data and Extra Segment registers, since they both will reference the same program segment.

Now you may be wondering why the loader can't load the segment registers when the program is moved into memory, as it did with the COM program, since it knows the base value for each segment. To answer that question, you have to remember the one important difference between COM and EXE programs: COM programs have only one program segment, therefore the base values can't change; EXE programs, on the other hand, can have any number of program segments. There may be three or four data segments, stack segments, or even code segments. The loader has no way of determining which program segments you wish to use first. Therefore, it leaves that decision up to you. You must supply the program instructions to load the registers. Just remember, the only segment registers you have control over are the DS, SS, and ES registers.

If you have no control over the CS register contents, and you link two or more code segments into a program, how is the segment containing the first instruction identified? Remember the program END directive? You can only have one END directive in a program. The argument to that directive points to the first instruction in the appropriate code segment. This information is used by the linker to arrange the code segment placement.

Did you notice that we introduced a variation in the SEGMENT directive? In the directive statement

```
PROG_STK    SEGMENT     STACK
```

the argument STACK identifies the segment as the stack segment. You must identify the stack segment in an EXE program, or the program may not run properly. The linker will tell you if you forget to use the argument STACK in one of the SEGMENT directives in an EXE program. On the other hand, never use the argument STACK in a COM program. The EXE to COM conversion won't work.

Self-Review Questions

50. The assembler assigns physical addresses to the program segments. _____
True/False

51. The assembler directive _____ defines the beginning of a program segment.

52. The assembler directive _____ defines the end of a program segment.

53. The name SEGMENT_DATA is considered a reserved word. _____
True/False

54. You are limited to four segments in a COM program. _____
True/False

55. The ASSUME directive requires two arguments separated by a colon: they are _____ _____ and _____ _____.

56. If you use three SEGMENT directives in a program, the minimum number of ENDS directives is _____, while the minimum number of ASSUME directives is _____.

57. You can use as many SEGMENT and ENDS directives as you feel is necessary in a COM program. _____
True/False

58. When an EXE program is loaded into memory, all of the segment registers are automatically loaded with the appropriate base address values _____
True/False

59. If you have assigned the name DATA1 to the program segment that will contain the program variables, what are the two instructions that you should use to load the DS register with the appropriate base address value?

_____

_____

60. If you forget to add the argument STACK to the program segment in a COM program, will the EXE to COM translation program work? _____
Yes/No

EXPERIMENT

An Introduction to Assembly Language Programming

OBJECTIVES:

1. *Demonstrate a typical COM program structure.*
2. *Review the assembler directives introduced in the Unit.*
3. *Demonstrate three MPU addressing modes.*
4. *Review the files generated in the assembly and linking process.*
5. *Demonstrate the assembler arithmetic operators.*

Introduction

We've introduced you to a great number of new concepts in this Unit. While you should be able to define or describe each of these concepts, you may not have a firm understanding of what they should or can do for you. That is the point of this experiment. It will review the material presented in the unit, giving you the opportunity to write and exercise many assembly language programs.

We will, however, limit the examples in this experiment to COM programs. EXE programs will be demonstrated at a later time.

Procedure

1. Call up your editor and enter the program listed in Figure 2-7. We suggest you use a file name that relates to what you are writing, such as PROG1.ASM, to represent the first assembly language program in the experiment. All of our references to the program will use the name PROG1.

   You can include the program comments, or leave them out. We listed them to help you follow the program steps. Right now comments aren't that important; however, when you begin writing longer programs, comments are indispensable for reminding you, and others, why you used a particular set of instructions.

```
TITLE EXPERIMENT 1 -- PROGRAM 1 -- COM PROGRAM FUNDAMENTALS
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
VALUE   EQU     33               ;Equate a value to a symbol
        ORG     100H             ;COM programs always start here
START:  MOV     AX,VALUE         ;Store the immediate value in AX,
        ADD     AX,33            ;then, add the immediate value to AX
        MOV     BX,VALUE         ;Store the immediate value in BX,
        ADD     BX,33            ;then add the immediate value to BX,
        SUB     AX,BX            ;finally, subtract BX from AX
        MOV     AX,DATA1         ;Store the data in memory into AX
        ADD     AX,DATA2         ;Add more data to AX
        MOV     SUM,AX           ;Store the sum in memory
        SUB     AX,AX            ;Clear AX
        INC     AX               ;Add 1 to AX
        DEC     AX               ;Subtract 1 from AX
;
DATA1   DB      33,0             ;Place 2 byte-sized values in memory
DATA2   DW      VALUE            ;Place word-sized value in memory
SUM     DB      2 DUP (?)        ;Reserve but don't initialize 2 memory bytes
COM_PROG ENDS                    ;End of program segment
        END     START            ;End of program, point to first instruction
```

Figure 2-7
Listing of first assembly language program.

Notice in the program listing that the first line contains a new assembler directive. The directive is TITLE. It allows you to identify a program by giving it a title. The assembler uses the title in its assembled program source listing as an identification header on each page of the listing. You can use up to 60 printable characters in a title. However, we suggest that you always make sure you use the legal characters, described for labels and names, in the first six character positions in the title. This is to ensure that the linker will always have valid characters to work with when it performs some of its multiple segment linking duties.

2. Refer to Figure 2-8 and assemble the program. The following steps will allow you to create the program object file, listing file, and cross- reference file.

```
A:MASM PROG1
The Microsoft MACRO Assembler
Version 1.07, Copyright (C) Microsoft Inc. 1981,82

Object filename [PROG1.OBJ]:
Source listing  [NUL.LST]: PROG1
Cross reference [NUL.CRF]: PROG1
 010E  A1 011C R                        MOV     AX,DATA1        ;Store the dataX
 E r r o r  ---.       31:Operand types must match
 0115  A3 0120 R                        MOV     SUM,AX          ;Store the sum y
 E r r o r  ---        31:Operand types must match

Warning Severe
Errors  Errors
0       2

A:
```

Figure 2-8
Assembling the first assembly language program.

A. Type the command "MASM PROG1" (assuming PROG1 is the file name) and RETURN.

B. The assembler will print a program heading and version number, and then the statement "Object filename [PROG1.OBJ]:". It is asking if you want the object file to be called PROG1.OBJ. You do, so press RETURN.

C. The assembler will print the statement "Source listing [NUL.LST]:". NUL means there will be no listing if you press RETURN. You want a listing, so type "PROG1" and RETURN. (The assembler will automatically add the extension LST.)

D. The assembler will print the statement "Cross reference [NUL.CRF]:". Again, NUL indicates there will be no cross reference file if you press RETURN. You want the file, so type "PROG1" and RETURN. The assembler will process the program, generate the files you requested, and display any error messages. If you wrote the program as it is listed in Figure 2-7, you will have two errors in your program.

```
 The Microsoft MACRO Assembler            01-20-84     PAGE    1-1
EXPERIMENT 1 -- PROGRAM 1 -- COM PROGRAM FUNDAMENTALS


1                                       TITLE EXPERIMENT 1 -- PROGRAM 1 -- COM
                                      PROGRAM FUNDAMENTALS
2                                       ;
3       0000                            COM_PROG  SEGMENT               ;Beginn
                                      ing of program segment
4                                               ASSUME  CS:COM_PROG,DS:COM_PROG
                                      ,SS:COM_PROG
5                                       ;
6       = 0021                          VALUE   EQU     33              ;Equate
                                       a value to a symbol
7       0100                                    ORG     100H            ;COM pr
                                      ograms always start here
8       0100  B8 0021                   START:  MOV     AX,VALUE        ;Store
                                      the immediate value in AX,
9       0103  05 0021                           ADD     AX,33           ;then,
                                      add the immediate value to AX
10      0106  BB 0021                           MOV     BX,VALUE        ;Store
                                      the immediate value in BX,
11      0109  83 C3 21                          ADD     BX,33           ;then a
                                      dd the immediate value to BX,
12      010C  2B C3                             SUB     AX,BX           ;finall
                                      y, subtract BX from AX
13      010E  A1 011C R                         MOV     AX,DATA1        ;Store
                                      the data in memory into AX
 E r r o r   ---        31:Operand types must match
14      0111  03 06 011E R                      ADD     AX,DATA2        ;Add mo
                                      re data to AX
15      0115  A3 0120 R                         MOV     SUM,AX          ;Store
                                      the sum in memory
 E r r o r   ---        31:Operand types must match
16      0118  2B C0                             SUB     AX,AX           ;Clear
                                      AX
17      011A  40                                INC     AX              ;Add 1
                                      to AX
18      011B  48                                DEC     AX              ;Subtra
                                      ct 1 from AX
19                                      ;
20      011C  21 00                     DATA1   DB      33,0            ;Place
                                      2 byte-sized values in memory
21      011E  0021                      DATA2   DW      VALUE           ;Place
                                      word-sized value in memory
22      0120     02 [                   SUM     DB      2 DUP (?)       ;Reserv
                                      e but don't initialize 2 memory bytes
23                     ??
24                        ]
25
26      0122                            COM_PROG ENDS                   ;End of
                                       program segment
27                                              END     START           ;End of
                                      program, point to first instruction
```

Figure 2-9
Source listing of the first program.

3. Figure 2-9 shows part of the listing for the program you just assembled. Note that the errors involve the sixth (line 13) and eighth (line 15) instructions. In both cases, the operand types don't match. Examine your program listing to see if the errors are the same. Type "TYPE PROG1.LST" and RETURN. The program listing will scroll rapidly up the screen. To stop the scrolling action, hold the CTRL key down and press the S key. To restart the scrolling action, press one of the character keys or the space bar. Stop the scrolling action when both errors are visible. Why do you think those instructions didn't assemble properly?

4. If you said that the instructions were mixing byte-sized memory with word sized registers, you are right. What can be done to correct those two instructions? Write the corrected instructions below.

5. Figure 2-10 shows the assembly language program with the sixth and eighth instructions corrected so they will assemble properly. In both cases, the addition of the assembler operator WORD PTR resolved the conflict. Notice that the operator is used with the operand that must be redefined. Naturally, it's always best to assign a variable size to match the data that will be used with that variable. Where that isn't possible, use the appropriate PTR operator in the instruction.

```
TITLE EXPERIMENT 1 -- PROGRAM 1 -- COM PROGRAM FUNDAMENTALS
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
VALUE   EQU     33               ;Equate a value to a symbol
        ORG     100H             ;COM programs always start here
START:  MOV     AX,VALUE         ;Store the immediate value in AX,
        ADD     AX,33            ;then, add the immediate value to AX
        MOV     BX,VALUE         ;Store the immediate value in BX,
        ADD     BX,33            ;then add the immediate value to BX,
        SUB     AX,BX            ;finally, subtract BX from AX
        MOV     AX,WORD PTR DATA1 ;Store the data in memory into AX
        ADD     AX,DATA2         ;Add more data to AX
        MOV     WORD PTR SUM,AX ;Store the sum in memory
        SUB     AX,AX            ;Clear AX
        INC     AX               ;Add 1 to AX
        DEC     AX               ;Subtract 1 from AX
;
DATA1   DB      33,0             ;Place 2 byte-sized values in memory
DATA2   DW      VALUE            ;Place word-sized value in memory
SUM     DB      2 DUP (?)        ;Reserve but don't initialize 2 memory bytes
COM_PROG ENDS                    ;End of program segment
        END     START            ;End of program, point to first instruction
```

Figure 2-10
Corrected listing of the first assembly language program.

6. Call up your editor and modify the sixth and eighth instructions to match those in Figure 2-10.

7. Assemble the program following the sequence described in the second step of this experiment. Don't worry about deleting the files generated in the first assembly, each old file will be deleted as the new file is stored on the disk.

   The one file that isn't deleted immediately, is the ASM file. When you use the Editor to modify a file, the original file is saved, just in case you make a mistake. The saved file is identified by the file extension BAK. If you need to use the BAK file, simply change the file extension back to ASM, using the MS-DOS RE-Name command.

8. The first file produced by the assembler is the object file PROG1.OBJ. This is the file that contains the machine level code. However, the code is not complete; there are a few references that must be resolved by the linker. Because the object file contains machine code and not ASCII code, you can't display the code with the MS-DOS TYPE command.

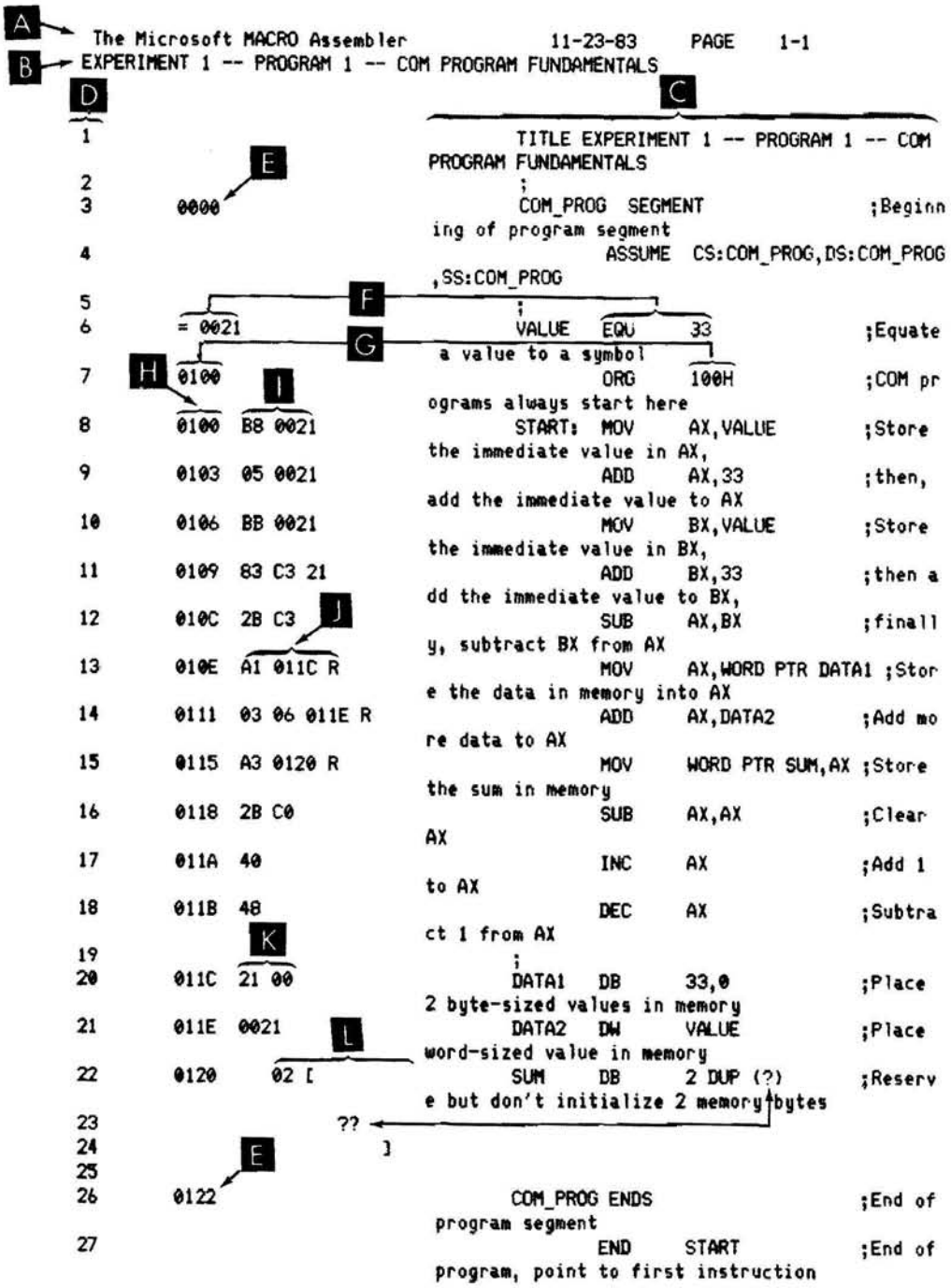


Figure 2-11
Source listing of the corrected first program.

9. The next file produced by the assembler is the source listing PROG1.LST. This file can be displayed. Figure 2-11 shows the source listing of your program. In addition to supplying a copy of the assembly language code, it also gives the machine code representing each instruction, and the relative address for each byte of code and data within the program. You can use the TYPE command described earlier to view your program listing. The various areas of the listing in Figure 2-11 are identified with letters to match the following description.

   A — The first line of the file identifies the assembler, prints the date assigned to the file, and a double page number. The number to the right advances with each new page of code. The number on the left advances only when told to do so, with the assembler directive PAGE +. When the assembler encounters the directive PAGE +, a new listing page is started, the number on the left is incremented, and the number on the right is reset to 1. In this manner, you can identify a new section or segment when you print the listing, since each PAGE + directive will also start a new page.

   B — The second line in the file is the title line. This is repeated on every page of your program. If there is no TITLE directive in your program, the line is left blank.

   C — This area of the listing is a copy of the original assembly language program. Because the column is only 47 characters wide, any line that is longer than 47 characters is "wrapped around" to the next line. That can make the code tricky to read, but everything is there.

   D — The far left column in the listing contains the program line numbers. Each line is assigned a number to make future debugging easier. This will make more sense when we describe the cross-reference file. Note that the line numbers are not displayed if the CRF file is not generated. Also, the line number feature (in some assemblers) is "turned-off" if the first line of the program is left blank.

E — The offset value 0000H serves as a reminder that this is the beginning of a segment. Every SEGMENT directive has that value assigned to it. The ENDS directive is assigned a number that represents the offset from the beginning of the segment to the last byte in the segment. The offset in this example is 122H; there are 22H bytes of code and data, and the program begins at offset 100H.

Notice that we used hexadecimal notation for all of the values in the preceding paragraph. This is because all numbers on the left side of the listing (except for the line numbers) are hexadecimal by default. Thus, there is no reason for the assembler to append the letter H to each number. The numbers used on the right side of the listing assume the default radix of the program.

F — The number on the left is the equated value of the number on the right. Every equated value is easy to spot, because it is preceded by an equals sign.

G — Recall that every COM program must begin at offset address 100H. The number on the left identifies the offset value specified by the ORG directive in the program. This is emphasized by the offset address of the following instruction.

H — This column lists the offset address of the first byte of every instruction or data definition in the program. For example, the first instruction begins at offset 0100H, the beginning of the program. The next instruction begins at offset 0103H, four bytes from the beginning of the program. This is because the first instruction is three bytes long. Since all code and data is referenced from the same segment base address, the first byte of data occupies the next consecutive offset address following the last byte of instruction code. In this case, the last instruction is a single-byte instruction at offset 011BH (line 18); the first byte of data is located at offset 011CH (line 20). Line 19 contains no code or data; it contains a semicolon, which is ignored by the assembler. You may have noticed that the program listing includes three lines that contain only semicolons. This is a common practice to help identify, or segregate, different areas of a program.

I — Instructions and data machine code are located in this column. For example, the first instruction moves the immediate value 33 into the AX register. The instruction code is hexadecimal B8, while the word-sized immediate value is hexadecimal 0021.

Keep in mind that the machine code in this column is in hexadecimal. A two-digit number group equals a byte-sized value, while a four-digit number group equals a word-sized value.

J — A memory operation is represented a little differently. In the example indicated, data located at the address offset identified by the variable DATA1, is moved into the AX register. The first byte-sized value is the machine code for the move instruction. The next word-sized value represents the address offset to the byte of data to be moved. The R following the offset is an indicator, or flag, to the linker that it must verify and/or modify the offset value to accomplish the memory move operation. In this simple COM program, the offset won't be changed by the linker. However, in a program with multiple segments, the offset from the segment register base address value may change when the segments are linked.

K — Data is represented as it is defined. In the first define byte statement (line 20), two bytes of data, 33 and 0, are defined. The machine code column shows the data as hexadecimal 21 and 00. When the symbol VALUE is defined as a word (line 21), it is shown as hexadecimal 0021.

L — Adding the DUP operator to a define statement changes the way the data is represented in the machine code column. The number of bytes or words being defined is specified first. Then, the expression being duped is shown enclosed within brackets. The expression is vertically isolated from the brackets to make it easier to identify. When the expression is undefined, as in this example, question marks are displayed — two for a byte-sized value and four for a word-sized value. Defined values are shown in their hexadecimal equivalent.

10. There is another part to the program source listing; it is called the Symbol Table. It serves as a programming aid for determining how the program is structured with regard to the various labels and names assigned to that program. Figure 2-12 shows the symbol table for your program.

```
A → The Microsoft MACRO Assembler            11-23-83     PAGE    Symbols
                                      -1
B → EXPERIMENT 1 -- PROGRAM 1 -- COM PROGRAM FUNDAMENTALS

C → Segments and groups:

                N a m e                 Size    align   combine class

COM_PROG . . . . . .                    0122    PARA    NONE

D → Symbols:

                N a m e                 Type    Value   Attr

DATA1. . . . . .                        L BYTE  011C    COM_PROG
DATA2. . . . . .                        L WORD  011E    COM_PROG
START. . . . . .                        L NEAR  0100    COM_PROG
SUM. . . . . .                          L BYTE  0120    COM_PROG        Length
                                =0002
VALUE. . . . . .                        Number  0021

E → Warning Severe
    Errors  Errors
    0       0
```

Figure 2-12
Symbol Table listing for the first program.

A — The first line of the display is similar to the program listing. Instead of numbering individual program sections, the PAGE is called "Symbols." The page count is maintained on the next line.

B — The program title is placed on this line, if the program is titled.

C — This section identifies the program segments and groups. Your program contains one segment. The segment is called COM_PROG.

Size indicates the size of the segment. This is not necessarily the number of program bytes, but rather the number of bytes from the beginning to the end of the segment. While your program contained only 23H bytes, it is originated at address offset 100H. Therefore the segment is 122H bytes long.

Align (alignment-type) is a term that specifies how the program is to be loaded into memory. PARA is the default alignment-type that stands for paragraph alignment. This specifies that the segment begins on a paragraph boundary — the address is divisible by 16. That is, the least significant hexadecimal digit of the address equals 0H. There are three other alignment-types:

BYTE	—	Specifies that the segment can begin anywhere.
WORD	—	Specifies that the segment can begin at any even address.
PAGE	—	Specifies that the segment can begin at any address that is divisible by 256. That is, the two least significant hexadecimal digits of the address equal 00H.

Nonparagraph alignment types should only be used with multiple-segment programs. This tells the assembler how to sequence the code and data. An example of a SEGMENT directive that uses an alignment-type argument is:

```
COM_PROG  SEGMENT   PAGE
```

This tells the assembler that the segment COM_PROG must begin at an address divisible by 256. Thus, if the preceding segment only occupies the first 36H address locations, COM_PROG will still begin at offset address 100H. We will use the default alignment-type paragraph in our COM programs.

Combine and class also assign specific characteristics to a segment during the linking process. Since these characteristics are unique to multisegment programs we will describe their functions at a later time.

D — This section lists all of the labels, variables, and symbols used in your program. Each column following the label or name describes specific characteristics for that label or name.

Type identifies the name. The letter L indicates it is a label (label, symbol, variable). The terms BYTE and WORD indicate the length of the variable, while the terms NEAR or FAR indicate the relative distance from the label to the instruction or directive that references the label. Distance is an important characteristic with the 8088 MPU. Near operations occur within the boundary of a segment. Far operations occur over distances that cross the boundary of a segment. The last type is number. This is always used in conjunction with symbols.

Value can assume two characteristics: number or offset. Numbers relate to symbols. Address offset values relate to labels and variables.

Attr (attribute) always shows the segment assigned to the label or variable. If a DUP operator is used with a define directive, the DUP "count" is shown as a length. In your program, the variable SUM contained a DUP statement with a count of two. No attribute is assigned to a symbol, since a symbol is simply a value not related to a segment.

E — The last section is a copy of the error message displayed at the end of the assembly operation. Warning Errors usually indicate that you made a minor mistake that the assembler "thinks" it resolved. Theoretically, you can run a program that contains Warning Errors. Severe Errors are program stoppers. These errors must be resolved before the program will run.

11. The last file generated when you assembled your program is PROG1.CRF. This is a cross-reference file that identifies the location of all of the labels, symbols, and variables in the program. However, it cannot be displayed. To make use of this file, you must translate the code using the Cross Reference program called CREF.COM. Type "CREF PROG1" and RETURN. The program will identify itself, and then ask if you want the cross-reference file to be called PROG1.REF. You do, so press RETURN. The file that's generated contains the necessary code to display a cross-reference listing of your program.

12. Type "TYPE PROG1.REF" and RETURN. You should see a display identical to Figure 2-13. Every label or name used in your program is listed. Following each is one or more decimal numbers. These numbers represent the line number in the program where you will find the label or name used. The pound sign (#) identifies the line where the name is defined. For example, the name COM_PROG is used once on line 3, three times on line 4, and once on line 26. It is defined on line 3.

```
EXPERIMENT 1 -- PROGRAM 1 -- COM PROGRAM FUNDAMENTALS

Symbol Cross Reference                (# is definition)      Cref-1

COM_PROG . . . . . .    3#    4     4     4     26

DATA1. . . . . . . .    13    20#
DATA2. . . . . . . .    14    21#

START. . . . . . . .    8#    27
SUM. . . . . . . . .    15    22#

VALUE. . . . . . . .    6#    8     10    21
```

Figure 2-13
Cross-reference file listing for
the first assembly language program.

The cross-reference file is used as a debugging tool. In a short program, it is of little value. However, in a large program, it can be very helpful when you must locate every application of a particular name.

13. Now that you have assembled your program, it's time to link it and resolve any addressing problems. Type "LINK PROG1;" and RETURN. The linker will display its identification header, a warning that there is no STACK segment in the program, and a statement that there was one error detected. A missing stack segment is normal for a COM program. The error message related to the missing stack. Thus, the linking operation was completed with no problems.

    The linker performs a number of special operations, and if requested, generates two other files in addition to the EXE file. Since these are related to multisegment programs, we'll ignore them for now. Placing the semicolon after the object file name causes the linker to generate only the EXE file.

14. You now have an EXE file. But this file is of no value, since your program was written in a COM file format. Thus, you have one more operation to perform — EXE to COM conversion. Type "EXE2BIN PROG1.EXE PROG1.COM" and RETURN. The program header will be displayed, and then the MS-DOS prompt (A: or A>) will be displayed. You now have a machine executable COM program.

    The command line for this conversion program is quite different from the command lines you typed earlier. After you specify the name of the program you wish to run (EXE2BIN), you must indicate the name of the program to be converted, along with its EXE file extension. Then you must indicate the name of the converted program along with its COM file extension. If you wish, you can change the name of the converted program. Just don't forget the COM file extension. Should you forget to include the converted program name, the converted program will receive, by default, the unconverted program name with the file extension BIN. Programs with the BIN extension will not run under MS-DOS. This means that you will have to rename the program after it is converted.

15. Examine the disk directory; type "DIR" and RETURN. Record the program size for each of the following programs:

    PROG1 OBJ _____
    PROG1 EXE _____
    PROG1 COM _____

The program PROG1.OBJ is the object program generated by the assembler. It contains 113 bytes of program machine code, relative memory addresses, and specific commands to be used by the linker for converting the code to an EXE program. After the linker processes the code, the file PROG1.EXE is generated. It contains 896 bytes of machine code and program support data. This code would be machine executable if it had been originally constructed to fit the EXE program format. Since it wasn't formatted to be an EXE program, it had to be processed one more time to remove any EXE program support data. This produced the program PROG1.COM. The 34 bytes of code remaining in the program after the last conversion is the actual machine code for the instructions and data that was generated by the assembler. You can verify this by counting the bytes of code in Figure 2-11.

Comparing the EXE and COM program sizes shows that a great amount of control data is stored within the EXE program. This is necessary to handle any possible multiple-segment format. On the other hand, COM programs are arranged in a specific fashion; additional support is not needed. Therefore, COM programs contain only the actual program code. Obviously, creating and storing small programs in COM form is a more efficient way of using disk storage than the EXE program form. You, of course, have no choice if your program requires more than one segment of code or data. You must then use the EXE form.

Discussion

The first part of the experiment gave you an opportunity to review the program assembly and conversion process. You should now understand what each of the "output" programs provide in terms of program documentation. The next part of the experiment will have you examine your program after it has been loaded into memory. You will also execute the program code, one instruction at a time, and observe the effects on the MPU and memory. Note that all of the addressing modes, except inherent addressing, are illustrated in the program.

Procedure Continued

16. Figure 2-14 is a copy of the source listing of your program. Use it as a reference in this and the following steps. Call up the debugger and load your program into memory. Type "DEBUG PROG1.COM" and RETURN. Now type "R" (for register) and RETURN. You can see that the program loader has assigned an address base value to all of the segment registers. When the program is executed, every memory operation will use that value to deter-

mine the physical address. Notice that the IP register contains the hexadecimal value 0100. This is automatically loaded into the IP register when a COM program is loaded into memory.

```
 The Microsoft MACRO Assembler            01-20-84    PAGE    1-1
EXPERIMENT 1 -- PROGRAM 1 -- COM PROGRAM FUNDAMENTALS


1
2                                       TITLE EXPERIMENT 1 -- PROGRAM 1 -- COM
                                      PROGRAM FUNDAMENTALS
3                                       ;
4       0000                            COM_PROG  SEGMENT               ;Beginn
                                      ing of program segment
5                                               ASSUME  CS:COM_PROG,DS:COM_PROG
                                      ,SS:COM_PROG
6                                       ;
7       = 0021                          VALUE   EQU     33              ;Equate
                                       a value to a symbol
8       0100                                    ORG     100H            ;COM pr
                                      ograms always start here
9       0100  B8 0021                   START:  MOV     AX,VALUE        ;Store
                                      the immediate value in AX,
10      0103  05 0021                           ADD     AX,33           ;then,
                                      add the immediate value to AX
11      0106  BB 0021                           MOV     BX,VALUE        ;Store
                                      the immediate value in BX,
12      0109  83 C3 21                          ADD     BX,33           ;then a
                                      dd the immediate value to BX,
13      010C  2B C3                             SUB     AX,BX           ;finall
                                      y, subtract BX from AX
14      010E  A1 011C R                         MOV     AX,WORD PTR DATA1 ;Stor
                                      e the data in memory into AX
15      0111  03 06 011E R                      ADD     AX,DATA2        ;Add mo
                                      re data to AX
16      0115  A3 0120 R                         MOV     WORD PTR SUM,AX ;Store
                                      the sum in memory
17      0118  2B C0                             SUB     AX,AX           ;Clear
                                      AX
18      011A  40                                INC     AX              ;Add 1
                                      to AX
19      011B  48                                DEC     AX              ;Subtra
                                      ct 1 from AX
20                                      ;
21      011C  21 00                     DATA1   DB      33,0            ;Place
                                      2 byte-sized values in memory
22      011E  0021                      DATA2   DW      VALUE           ;Place
                                      word-sized value in memory
23      0120     02 [                   SUM     DB      2 DUP (?)       ;Reserv
                                      e but don't initialize 2 memory bytes
24                       ??
25                          ]
26
27      0122                            COM_PROG ENDS                   ;End of
                                       program segment
28                                              END     START           ;End of
                                       program, point to first instruction
```

Figure 2-14
Copy of the source listing of the first program.

Examine the last line of the display. The first eight numerals are the base and offset address values, separated by a colon, for the first instruction in your program. The rest of the line contains the machine language code and the assembly language code for that instruction.

17. Compare the code in your display to the code in Figure 2-14, the program source listing. What are the differences in the assembly language code?

What are the differences in the machine code, and why is there a difference?

The difference in the assembly language code lies in the handling of the label and symbol. The label is missing and the symbol is replaced by its actual value. The reason for these differences is that the debugger doesn't have access to a program listing. Rather, the debugger contains a disassembler that converts the machine code back into assembly language. Labels aren't reproduced, since they only have significance when a program is being assembled for the first time, when address offsets must be calculated. Symbols are shown as actual values, since that is the data stored in the machine code.

The difference in machine code lies in the manner that it is presented; the actual code hasn't changed. The source listing presents the code in a logical arrangement. In this case, move (B8) a value (0021) into the AX register. The debugger presents the code as it is physically stored in memory. The instruction (B8) is still first, but the word-sized value is rearranged so the low byte (21) precedes the high byte (00). Recall that when the MPU processes word-sized values in memory, it expects to find the low-byte in the first (lowest) memory location and the high-byte in the next (higher) memory location.

18. To get a better idea of how this low-byte/high-byte arrangement affects the loading of a program in memory, let's examine the section of memory containing your program. Type "D" (for dump memory) and RETURN. The debugger displays 128 bytes of memory beginning at the address pointed to by the DS register with an offset of 0100H. That is the default address for the first dump after entering the debugger with a COM program. Figure 2-15 represents the display you have produced since entering the debugger. Your display should be similar. The segment register values may be different. Remember, the program loader determines those values.

    Compare the code in your display with the source listing machine code in Figure 2-14. Notice that the code is in sequence, except for the word-sized values. After offset address 0121H, your display may not match the figure. This area is beyond your program and could contain any value. It all depends on what your system did with this memory prior to your loading the debugger.

    Record the values stored at the following offset address locations.

    0120H _____
    0121H _____

    Recall that these are the two reserved locations that were not initialized by the program. Because these memory locations were not initialized, there is no way to determine what values you will find.

```
A:DEBUG PROG1.COM

DEBUG  version 1.08
>R
AX=0000  BX=0000  CX=0022  DX=0000  SP=FFF0  BP=0000  SI=0000  DI=0000
DS=0A09  ES=0A09  SS=0A09  CS=0A09  IP=0100   NV UP DI PL NZ NA PO NC
0A09:0100 B82100        MOV     AX,0021
>D
0A09:0100  B8 21 00 05 21 00 BB 21-00 83 C3 21 2B C3 A1 1C   8!..!.;!..C!+C!.
0A09:0110  01 03 06 1E 01 A3 20 01-2B C0 40 48 21 00 21 00   .....# .+@@H!.!.
0A09:0120  00 00 0D 0A 20 20 20 20-20 20 20 20 20 20 20 20   ....
0A09:0130  20 20 20 20 20 20 20 20-20 20 20 20 72 61 74 65               rate
0A09:0140  2E 0D 0A 0D 0A 20 20 20-20 20 20 20 20 20 20 20   .....
0A09:0150  20 20 20 20 20 46 39 20-2D 20 53 61 76 65 20 70        F9 - Save p
0A09:0160  72 6F 67 72 61 6D 20 69-6D 61 67 65 20 20 2D 20   rogram image  -
0A09:0170  20 54 68 69 73 20 6F 70-74 69 6F 6E 20 77 69 6C    This option wil
>
```

Figure 2-15
Debugger display.

19. Now it's time to evaluate the program instructions. You will execute each instruction using the "trace" feature of the debugger. Every time you use the trace command, the current instruction will be executed, updating the MPU registers and any affected memory. After the instruction is executed, the next instruction will be displayed.

    Before you begin "single-stepping" through the program, type "R" and RETURN. This will show you the current status of the MPU registers, and it will show you the first instruction in the program. Notice that the IP register contains the value 0100H, the address offset of the next instruction to be executed.

20. Type "T" and RETURN. The trace command executes the instruction pointed to by the CS and IP registers. In this case, move the value 33 into the AX register. The AX register now contains the value _____H.

21. Type "T" and RETURN. The value 33 is added to the contents of the AX register. The AX register now contains the value _____H.

22. Type "T" and RETURN. This instruction moves the value 33 into the BX register. The BX register now contains the value _____H.

    Notice that the next instruction to be executed adds the immediate value 33 to the BX register. However, the instruction is structured a little differently from the previous add instruction. The actual instruction code requires two bytes of code, whereas the add AX instruction required only one byte of code. This is because the 8088 MPU, like most other MPUs, is structured to conserve code and execution time when performing arithmetic operations with the AX (accumulator) register.

    It's interesting to note that the value 33 occupies two bytes in the add AX instruction while it occupies only one byte in the add BX instruction. That is a function of the 8088 MPU instruction code. Had the value exceeded 255, the add BX instruction would have also required two data bytes. The term "+21" in the instruction, is the disassembler's way of showing that the immediate value 21H is being added to the specified register.

23. Type "T" and RETURN. The value 33 is added to the contents of the BX register. The BX register now contains the value _____H.

    These last four instructions are examples of immediate operand or immediate addressing mode instructions. The immediate value occupies the source operand part of the instruction.

24. Type "T" and RETURN. The contents of the BX register are subtracted from the contents of the AX register, and the difference is stored in the AX register. Notice that the contents of the BX register are not affected by the operation. The AX register now contains the value _____H, while the BX register contains the value _____H. This was an example of a register addressing mode instruction. Remember that any general register pair can be used to add or subtract one register from another. Likewise, an immediate value can be added to or subtracted from any general register.

25. Type "T" and RETURN. Since this is a word move operation, the contents of address offset 011CH are moved into the low-byte of the AX register, while the contents of address offset 011DH are moved into the high-byte of the AX register. The AX register now contains the value _____H. Notice that the disassembler identified the memory location by placing the address offset within square brackets. It also indicated the segment register that will supply the base value for determining the physical address of the memory location, and the data that is stored at that location. This information is on the right end of the instruction code line (DS:011C=0021).

    Remember, because this instruction is moving data from memory, the MPU obtains the base address value from the DS register. The offset address is then added to the base address to form the physical address of the memory location. For example, using the DS register contents in Figure 2-15 as the base address, then the first byte of data is found at physical address 0A1ACH (0A090H + 011CH).

26. Type "T" and RETURN. This time, we are adding a word of data from memory to the AX register. Again, the first byte addressed in memory is moved into the low-byte of the AX register, while the next byte in memory is moved into the high-byte of the AX register. The AX register now contains the value _____H.

27. Type "T" and RETURN. The data in the AX register is moved into offset memory locations 0120H and 0121H. What value, do you think, is moved into each of the locations?

    0120H _____H
    0121H _____H

28. Type "D 100" and RETURN. Typing a hexadecimal value after the "D" causes the memory dump to begin at the offset address value specified. If the value is not included, the dump will begin at the next memory location after the previous dump. In your case, the next memory location would have been offset address 0180H.

    Examine offset address locations 0120H and 0121H. What value was moved into each location?

    0120H _____H
    0121H _____H

    You should have found the values 42H (offset 0120H) and 00H (offset 0121H).

29. Type "T" and RETURN. This instruction subtracted the AX register from the AX register. The reason for performing such an operation is to clear, or zero, the register contents. You could have moved the value 0000H into the register, but it would have taken more bytes of machine code and more time to execute. You will find this a handy process when you need to clear any of the general registers. The AX register now contains the value _____H.

30. Type "T" and RETURN. The AX register is incremented by one. The AX register now contains the value _____H.

31. Type "T" and RETURN. The AX register is decremented by one. Both the increment and the decrement instructions are single-byte arithmetic operations that let you increase or decrease the value of a general register. The AX register now contains the value ____H.

    The decrement instruction is the last instruction in your program. Yet, if you look at your display, you see that the debugger's disassembler has decoded the program's data and come up with another instruction. Naturally, there are no more instructions, but the disassembler has no way of determining that. This points out the fact that you must be careful when using the disassembler to examine code. It always assumes that the IP register is pointing at a valid instruction.

Discussion

The few instructions that you have examined should give you an idea of how the 8088 MPU can be made to perform many useful tasks. While the range of control is limited, a practical program is not impossible. Therefore, your next task in this experiment is to write a simple COM program to multiply one number by another. Use the editor to write the program, the assembler to convert it to object code, the linker to process the code, and the EXE2BIN program to convert the EXE file into a COM file. Don't forget to use the appropriate assembler directives.

Procedure Continued

32. Exit the debugger in preparation for writing the program. Type "Q" and RETURN. The system prompt (A: or A>) will appear.

33. Write a program to multiply the number five times the number seven. To help conserve memory space, use byte-sized values. Have the program store the product in memory. Assemble, link, and convert the program to a COM file. Load the program into memory with the debugger and single-step through the instructions to verify that the program works.

Discussion

Even with a few simple instructions to choose from, there can be many variations in program design, or structure. Figure 2-16 is one possibility. If your program is different, don't worry. The point of this exercise is to use the knowledge you have gained to create a program that works. Since we can't review your program, let's take a look at the program in Figure 2-16.

```
TITLE EXPERIMENT 1 -- PROGRAM 2 -- SIMPLE MULTIPLICATION THROUGH ADDITION
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
MULTI   EQU     7                ;Equate program multiplicand value
        ORG     100H             ;COM programs always start here
START:  SUB     AL,AL            ;Clear the PRODUCT register
        MOV     BL,MULTI         ;Get the multiplicand,
        ADD     AL,BL            ;then perform the
        ADD     AL,BL            ;multiplication operation
        ADD     AL,BL            ;by adding the multiplicand
        ADD     AL,BL            ;to the PRODUCT register
        ADD     AL,BL            ;5 (multiplier) times
        MOV     PRODUCT,AL       ;Store the product
;
PRODUCT DB      0                ;Reserve one byte in memory
                                 ;and initialize the byte to 0
COM_PROG  ENDS                   ;End of program segment
        END     START            ;End of program, point to first instruction
```

Figure 2-16
Program to multiply two numbers through repeated addition.

We decided to use an equate statement to define the multiplicand. You could have just as easily moved the immediate value seven into a register. However, you will find that it is a good idea to make it a habit to use equate statements to define constants.

The first instruction in the program clears the AL register. You may have noticed that when you examined your program with the debugger, the AX register was already at zero. So why clear it again? The reason is, you can never be sure that any particular register will be cleared prior to executing a program. Therefore, it's always best to clear a register if that register must be empty for an operation.

The next instruction loads the BL register with the multiplicand, in preparation for the multiplication through repeated addition operation. You could have just as easily added the immediate value seven to the AL register five times. However, the process we chose uses less instruction code and execution time, even though there are more instructions involved in the process. The next five instructions perform the actual multiplication operation.

Finally, the last instruction stores the product in memory.

As you can see, multiplication through repeated addition is a tedious process. In fact, it is almost impossible to use when large numbers are involved. In addition, every time you change the multiplier, you must change the number of instructions in the program. The next unit will show you how a simple program loop can resolve these problems. However, before you leave this experiment, we wish to show you how the arithmetic operators are used.

Procedure Continued

34. Enter and convert the assembly language program shown in Figure 2-17 into a COM program.

35. Load the program into memory with the debugger, and single-step through the program. As you execute each instruction, compare the display with the instructions and comments in Figure 2-17, and with the following comments.

```
TITLE EXPERIMENT 1 -- PROGRAM 3 -- USING ARITHMETIC OPERATORS
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
NUM0    EQU     0                ;First constant
NUM1    EQU     NUM0 + 16        ;Reference second constant to first
NUM2    EQU     NUM0 + 33        ;Reference third constant to first
        ORG     100H             ;COM programs always start here
START:  MOV     AX,NUM0 + 15     ;Load AX immediate with first
                                 ;constant plus 15
        MOV     BX,NUM1 - 0FH    ;Load BX immediate with second
                                 ;constant minus 15
        MOV     CX,15 * 3        ;Load CX immediate with 45
        MOV     DX,NUM2 / 0FH    ;Load DX immediate with quotient
                                 ;of third constant divided by 15
        MOV     BP,NUM2 MOD 0FH  ;Load BP immediate with remainder
                                 ;of third constant divided by 15
        MOV     SI,VAR0 + 2      ;Load the data word located at
                                 ;address offset VAR0 + 2 into SI
        MOV     DI,VAR2 - 4      ;Load the data word located at
                                 ;address offset VAR2 - 4 into DI
        MOV     AX,VAR1 - VAR0   ;Load the constant, produced by
                                 ;subtracting variable
                                 ;VAR0 from VAR1, into AX
        MOV     BX,- (NUM0 + 1)  ;Load the 2s complement of the
                                 ;first constant + 1 into BX
        MOV     CX,0F0F0H SHL 4  ;Shift the immediate value 4 bits
                                 ;left and load it into CX
        MOV     DX,0F0F0H SHR 4  ;Shift the immediate value 4 bits
                                 ;right and load it into DX
;
VAR0    DW      0F0FH            ;Initialize a word of data
VAR1    DW      1010H            ;Initialize another word of data
VAR2    DW      2 DUP (?)        ;Reserve space for 2 words of data
COM_PROG  ENDS                   ;End of program segment
        END     START            ;End of program, point to beginning
```

Figure 2-17
Program to show the operation of the arithmetic operators.

Before we discuss the instructions, look at how the equate statements are handled. The symbol NUM0 is equated to the value zero. The next two symbols are equated to the symbol NUM0 plus a constant, using an arithmetic operator. This is a handy way to establish related symbols in a program.

The first five instructions produce obvious results, since they are dealing with immediate values. The next three instructions show how the operators can be used with variables. There are no other possible combinations of variables and constants (symbols).

In the first example, VAR0 has an offset of 0123H. When the constant 2 is added to the offset, the new offset is 0125H. Thus, the word-sized data that is loaded into the SI register is located at offset addresses 0125H and 0126H.

The second example performs the same operation, only this time, the constant 4 is subtracted from the offset of VAR2. This produces the new offset address 0123H. The word-sized data that is located at this address and address 0124H is loaded into the DI register.

The last example using a variable, shows that when one variable is subtracted from another, the difference is a constant. In this case, VAR1 minus VAR0 equals 2.

The next instruction illustrates the hierarchy aspect of arithmetic operators. Without the parentheses, NUM0 would be negated, and then one would be added to the result to produce the value 1. With the parentheses, one is added to NUM0, and then the result is negated to produce the value 0FFFFH.

The last two instructions show the operation of the shift operators. In the first example, the value 0F0F0H is shifted left four bits to produce the value 0F00H. Remember that in some early versions of the IBM assembler, the operation is reversed. It assumes that the instruction reads, "shift left the value 4, 0F0F0H counts." Since the shift count exceeds the capacity of the register, the shift is not executed. Thus, the unshifted value four is moved into the CX register.

The second example shifts the value 0F0F0H four bits right to produce the value 0F0FH. Again, we must remind you that some versions of the IBM assembler always return the value 0000H for a shift right operation.

This completes the Experiment for Unit 2. Now would be a good time to write a few programs on your own, to reinforce your understanding of the material presented in this unit. When you are finished, proceed to the Unit 2 Examination.

UNIT 2 EXAMINATION

1. State the four steps in the generation of a COM program.

   A. _____
   B. _____
   C. _____
   D. _____

2. Which characters are allowed to be used in labels and names?

   _____
   _____

3. In an assembler directive statement, the name field uses a colon as an end delimiter. _____
   True/False

4. In an instruction statement, if two operands are used they are always separated by a:

   A. Comma.
   B. Semicolon.
   C. Space.
   D. Colon.

5. In an instruction statement or assembler directive statement, the comments field always uses a semicolon as a beginning delimiter.

   _____
   True/False

6. A label or name can begin with a letter or a numeral. _____
   True/False

7. Which of the following operand types is part of a processor control instruction?

   A. Register.
   B. Inherent.
   C. Memory.
   D. Immediate.

8. The default radix for MACRO-86 is _____.

9. The instruction

```
INC   BP
```

uses _____ addressing.

10. The arithmetic operator in the following instruction that will be evaluated last is:

```
ADD  AX,(NUM1 + 5) * 5 MOD (NUM1 SHL 2)
```

A. +
B. *
C. MOD
D. SHL

11. The ASSUME directive must precede the SEGMENT directive in a COM program. _____
True/False

12. The assembler directive END identifies:

A. The beginning of an EXE program segment.
B. The end of an EXE program segment.
C. The beginning of an EXE program.
D. The end of an EXE program.

13. The highest number of SEGMENT directives that can be used in a COM program is:

A. One.
B. Two.
C. Four.
D. No limit.

14. COM programs must have a separate stack segment. _____
True/False

EXAMINATION ANSWERS

1. The four steps in the generation of a COM program are:

   A. Write the assembly language code with an editor.
   B. Assemble the program code to produce the object code.
   C. Link the object code.
   D. Convert the EXE file to a COM file.

2. The characters **A** through **Z,, 0** through **9**, **?**, **@**, **_**, and **$** are allowed to be used in labels and names.

3. **False**. In an assembler directive statement, the name field uses a space, not a colon, as an end delimiter.

4. A — In an instruction statement, if two operands are used they are always separated by a comma.

5. **True**. In an instruction statement or assembler directive statement the comments field always uses a semicolon as a beginning delimiter.

6. **False**. A label or name cannot begin with a numeral.

7. B — The **inherent** operand type is part of a processor control instruction.

8. The default radix for MACRO-86 is **decimal**, or **10**.

9. The instruction

```
INC  BP
```

uses **register** addressing.

10. C — The arithmetic operator in the following instruction that will be evaluated last is MOD.

```
ADD AX,(NUM1 + 5) * 5 MOD (NUM1 SHL 2)
```

11. **False**. The ASSUME directive does not precede the SEGMENT directive in a COM program.

12. D — The assembler directive END identifies the end of an EXE program.

13. A — The highest number of SEGMENT directives that can be used in a COM program is one.

14. **False**. EXE programs must have a separate stack segment, COM programs contain one common segment.

SELF-REVIEW ANSWERS

1. Assembler notation consists of two types of statements: assembler **directives** and **instructions**.

2. Assembly language programs are written using **EDLIN.COM** or some other text editor.

3. The program in text form is called the **source** code.

4. MASM.EXE is used to translate the program into **object** code.

5. The file extension for an assembly language program is **ASM**.

6. The **linker** is used to convert object code to executable code.

7. The program **EXE2BIN.EXE** is used to convert an EXE file to a COM file.

8. The **COM** file is limited in size to 64K bytes.

9. The assembly language instruction statement consists of four fields. These are:

   A. Label
   B. Opcode
   C. Operands
   D. Comments

10. Characters that are used to indicate the beginning and end of the different fields in an instruction are called **delimiters**.

11. Another term for opcode in an instruction is **mnemonic**.

12. **False**. An instruction statement can contain zero, one, or two operands, but not three.

13. In addition to the letters A through Z and the numerals 0 through 9, the legal characters in a label or name include:

   A. ?
   B. @
   C. _
   D. $

14. The assembler will recognize labels and names up to **31** characters long.

15. The assembler directive statement consists of four fields. They are:

    A. Name.
    B. Directive.
    C. Argument.
    D. Comments.

16. **True**. Assembler directive statements are never translated into object code.

17. A name can assume one of two attributes: **variable** or **symbol**.

18. Symbols represent **constants**.

19. **False**. The end delimiter for a name is a space, not a colon.

20. The **ORG (origin)** directive allows you to specify an address offset value in a program.

21. The **EQU (equate)** directive allows you to associate a symbolic name to a constant or another symbol.

22. The **define (byte or word)** directive is used to allocate memory units to data storage.

23. The physical address of a variable is determined by adding the variable offset to the shifted contents of the **Data Segment** register.

24. The operator **DUP** is used to initialize a block of predefined memory.

25. If the address offset assigned to the variable PRODUCT is 0100H, the directive statement

```
PRODUCT     DW      0E5A8H
```

    will load the value **0A8H** at address offset 0100H and the value **0E5H** at address offset 0101H.

26. The **END** directive is used to tell the assembler that the program is complete.

27. Instructions that do not use some form of addressing are called **inherent** instructions.

28. An immediate operand is always a **source** operand.

29. The default radix for an immediate operand constant is **decimal**.

30. You can change the default radix with the assembler directive **.RADIX**.

31. Only **one** ASCII character can be coded into a byte-sized memory location.

32. **No**. The register operand addressing mode cannot contain an immediate operand. By definition, it contains two register operands.

33. Of the 14 8088 MPU registers, the **Instruction Pointer** register is not directly addressable through an instruction.

34. The four general registers that are composed of two 8-bit registers are:

    A. AX
    B. BX
    C. CX
    D. DX

35. **No**. The instruction

```
DATA1:    ADD    SI,BL
```

    is not a legal operation. The register sizes do not match.

36. **False**. The instruction

```
          MOV    AX,PRODUCT
```

    will move the contents of the memory location found at the offset address assigned to the variable PRODUCT into the AX register.

37. If the offset address of the variable SUM is 0105H and SUM is defined as a word-sized memory location, the instruction

```
          MOV    SUM,3456H
```

    will move the value 34H into the memory location at offset address **0106H**.

38. If the DS register contains the value 0D20H, then the instruction from question 37 will load the value 56H into the memory location at physical address **0D305H**.

39. The assembler operator **pointer (PTR)** will allow the contents of two consecutive byte-sized memory locations to be loaded into a 16-bit general register.

40. **False**. Data in one memory location cannot be transferred directly to another memory location. It must be transferred to a register first.

41. Name the seven arithmetic operators.

    A. + Add
    B. − Subtract or Negate
    C. * Multiply
    D. / Divide
    E. MOD Modulo
    F. SHR Shift Right
    G. SHL Shift Left

42. **True**. The add (+) and subtract (−) operators can be used with constants, symbols, and variables.

43. Subtracting a constant from a variable produces a **variable**.

44. Subtracting a variable from a variable produces a **constant**.

45. The 2's complement is produced using the **subtract** or **negate** operator.

46. **False**. The multiply (*) and divide (/) operators can only be used with constants and symbols. Variables can only be used with the add (+) and subtract (−) operators.

47. The **divide (/)** operator returns the quotient.

48. **False**. The assembler allows the shift left operator to generate a value that exceeds the capacity of a register only if the register is a 16-bit register. 8-bit registers will cause the assembler to generate an error message.

49. The instruction

```
VALUE:    MOV     AL,11B SHL ((5*2-20/3) MOD 5)
```

moves the value **00110000B** or **30H** into the AL register. There are two levels of parentheses in the source operand. The inner level is evaluated first, then the outer level. Thus, the expression (5*2-20/3) is evaluated first, giving the value four. Then, this is divided by five, leaving a remainder of four. Finally, the value 11B is shifted left four bits by the remainder four.

50. **False**. The assembler assigns logical addresses to the program segments.

51. The assembler directive **SEGMENT** defines the beginning of a program segment.

52. The assembler directive **ENDS** defines the end of a program segment.

53. **False**. The name SEGMENT_DATA is not considered a reserved word. However, the name SEGMENT by itself would be considered a reserved word.

54. **False**. You are limited to one segment in a COM program. Only EXE programs can have more than one segment.

55. The ASSUME directive requires two arguments separated by a colon: they are **segment register** and **segment name**.

56. If you use three SEGMENT directives in a program, then you must have an equal number of ENDS directives, **three**. The minimum number of ASSUME directives is **one**, there is no maximum number.

57. **False**. You can only use one SEGMENT and one ENDS directive in a COM program.

58. **False**. When an EXE program is loaded into memory, only the CS register is automatically loaded with the appropriate base address value.

59. If you have assigned the name DATA1 to the program segment that will contain the program variables, the two instructions that you should use to load the DS register with the appropriate base address value are:

```
MOV     AX,DATA1        ;Get the base address value
MOV     DS,AX           ;Store the value
```

60. **Yes**. A COM program will translate properly if the argument STACK is left out of the program, since you don't use the argument STACK in a COM program. It is only used in an EXE program.

INSERT

Unit 3

PROGRAM TRANSFER INSTRUCTIONS

CONTENTS

INTRODUCTION

The last unit introduced you to the basic concepts of assembly language. You learned how to structure the code, establish program segments, and implement the three basic instruction addressing modes. The programs you wrote are what we call "straight-line" programs. That is, all of the instructions are executed sequentially.

While straight-line programs serve a function, they do present certain limitations; the biggest of these was illustrated in the multiplication program you wrote in the last experiment. If you wish to repeat an operation 25 times, then you must write the instruction 25 times. Ideally, you would write the instruction one time, and then repeat the instruction by "branching" back to it the required number of times.

The 8088 MPU instruction set has a number of instructions that let you control the direction of program flow. They can be used to force the program to repeat a number of instructions, or they can cause the program to bypass a number of instructions under certain operating conditions. These program transfer instructions fall under two categories: jump instructions and loop instructions. The main purpose of this unit is to show you how to use these instructions to break out of the straight-line program.

Since you will be writing more complex programs in this and the following units, you should first learn how to approach the task of writing a program in an organized fashion. For that reason, the first part of this unit will show you one of the more popular methods for planning the structure of a program — flowcharting. After we show you how to organize and construct a flowchart, we will review conditional and unconditional jump instructions, and finally, we will review the conditional and unconditional loop instructions.

Use the "Unit Objectives" that follow to evaluate your progress. When you can successfully accomplish all of the objectives, you will have completed this Unit. You can use the "Unit Activity Guide" to keep a record of those sections that you have completed.

UNIT OBJECTIVES

When you complete this Unit, you will be able to:

1. Construct a flowchart for a stated problem.

2. Name the five most commonly used flowchart symbols.

3. Explain the difference between direct and indirect jumps, and name a use for each.

4. Explain the difference between conditional and unconditional jumps and loops.

5. Name the six flags in the Flag register that are used by the conditional jump and loop instructions and state the conditions that cause these flags to be set.

6. State how to use, or use the following instructions in a program: CMP, CLC, CMC, STC, JA, JNBE, JNB, JAE, JB, JNAE, JC, JBE, JNA, JE, JZ, JNLE, JG, JLE, JNG, JNL, JGE, JL, JNGE, JNC, JNE, JNZ, JNO, JNP, JPO, JNS, JO, JP, JPE, JS, JMP, LOOP, LOOPE, LOOPZ, LOOPNE, and LOOPNZ.

7. Name the conditions that cause the various conditional jumps and loops to be taken.

8. Write a simple program to form a program loop.

UNIT ACTIVITY GUIDE

		Completion Time
☐	Read the Section on "Flowcharting."	_____
☐	Complete Self-Review Questions 1-5.	_____
☐	Read the Section on "Jumps."	_____
☐	Complete Self-Review Questions 6-18.	_____
☐	Read the Section on "Loops."	_____
☐	Complete Self-Review Questions 19-27.	_____
☐	Perform the Experiment.	_____
☐	Complete the Unit 3 Examination.	_____
☐	Check the Examination Answers.	_____

FLOWCHARTING

As the programs you write increase in complexity and length, the need for some type of organization for problem solving becomes quite evident.

The exact method of problem solving will vary from person to person. Some may have an intuitive "feel" for program development and may write programs "off the top of their head." If you can do this, you are fortunate indeed. Most people, however, need a formal "plan of attack" when solving problems and writing programs.

Program Organization

Essentially, there are three steps to writing a program. The first is to **define the problem**. This may seem like a relatively easy task, but it sometimes proves to be the most difficult. Unfortunately, there is no set procedure for defining problems. Some problems will lend themselves to mathematical definitions, while others may require a logical or graphical approach. We can only advise you to analyze each situation carefully before you attempt to define the problem.

The last step in creating a program is the **writing of that program**. You already know a little bit about writing programs, and you will be learning much more as you progress through the course. What we are primarily concerned with here is the second step in creating a program: mapping or **flowcharting** the solution.

Flowchart Symbols

Given a well structured blueprint, a carpenter can construct even the most complex project with little or no difficulty. The same is true of a programmer with a good flowchart. As a matter of fact, some flowcharts are so detailed that little remains for the programmer to do other than fill in the appropriate instructions. For our purposes, we'll use a simple flowcharting technique using five of the most common symbols. These are shown in Figure 3-1.

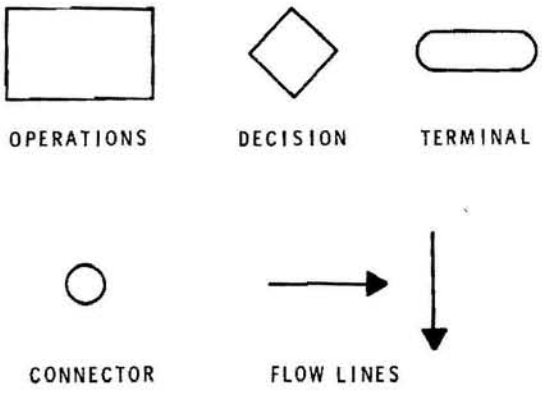


Figure 3-1
Flowchart symbols.

All flowcharts have a beginning, and most will have an end. These are referred to as the "terminal" points of the flowchart, and they are represented by the **terminal** symbol. The terminal symbol seen in Figure 3-2 is simply labeled "start" or "stop," to denote its meaning.

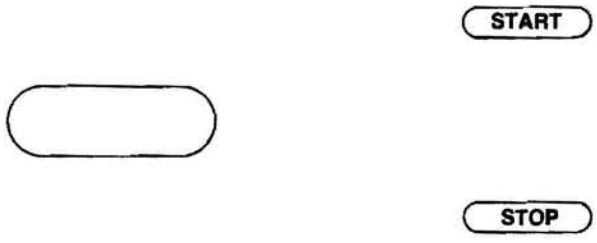


Figure 3-2
Terminal symbol.

The **operations box** is probably the most frequently used flowchart symbol. It may represent a transfer process such as moving or loading data, or an algebraic process like addition or multiplication. It can also be used for such statements as "print" or "set." If you have a process that is not specified by one of the other symbols, use the operations box. Some examples of the operations box are shown in Figure 3-3.

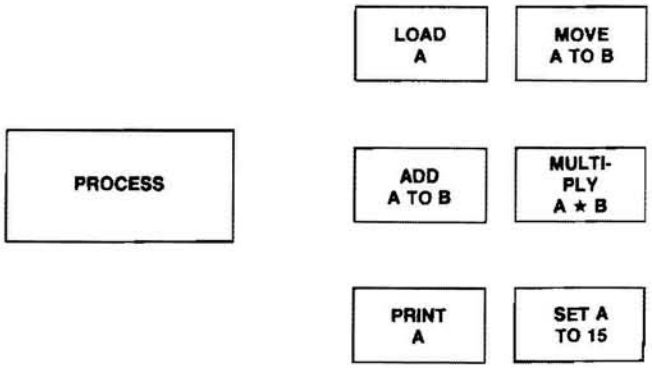


Figure 3-3
The operation box.

The diamond-shaped **decision box** is really the "heart" of the flowchart. It indicates a logical choice between two conditions and, therefore, it controls the direction of program flow. If a condition is satisfied, the "yes" route is taken. If the condition is not satisfied, the "no" route is selected. Typical examples for using the decision box are shown in Figure 3-4. The yes and no routes can originate in any corner of the diamond.

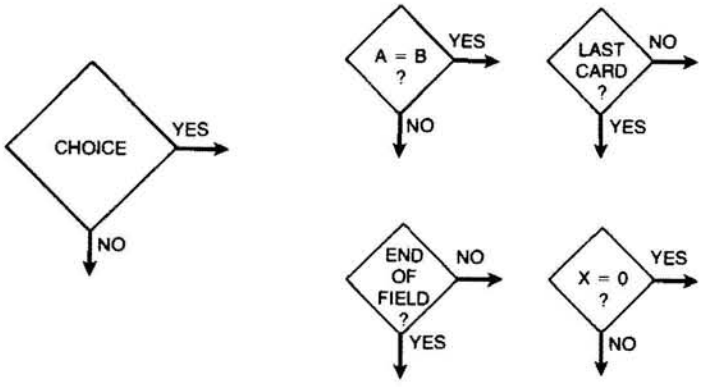


Figure 3-4
Decision box.

Flow lines and **connectors** are used to tie the symbols and sections of the flowchart together. Normally, the chart will be arranged to flow from top to bottom and from left to right. However, there is no set rule in regard to this, and the flowchart could just as well flow in the opposite direction. The use of flow lines as well as connectors is demonstrated in Figure 3-5. The connector symbols direct you from one section of the program to another. The connectors are labeled with values to indicate where the connection takes place; 1A connects with 1A, 2A connects with 2A, etc.

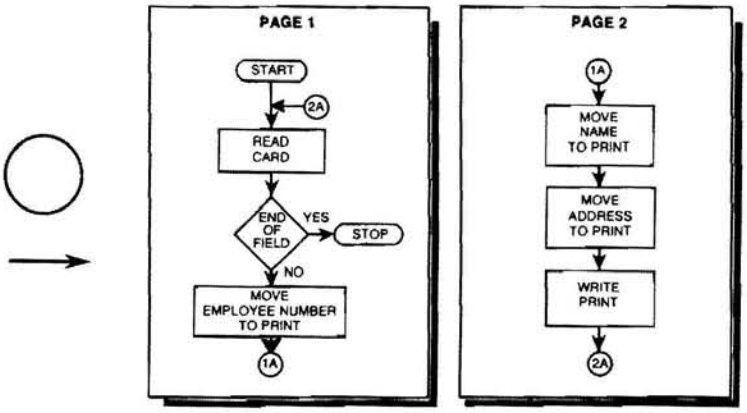


Figure 3-5
Connectors and flow lines.

Mathematical Symbols

Usually, sophisticated flowcharts use mathematical symbols to represent the decision-making process. These symbols allow us to illustrate our decision quickly and concisely. In Figure 3-6A, you will see the five most frequently used symbols and their meanings. Figure 3-6B shows some examples of their use.

A

	SYMBOL	EXAMPLE	DESCRIPTION
1.	=	A = B	A is equal to B.
2.	>	A > B	A is greater than B.
3.	<	A < B	A is less than B.
4.	≧	A ≧ B	A is greater than or equal to B.
5.	≦	A ≦ B	A is less than or equal to B.

B

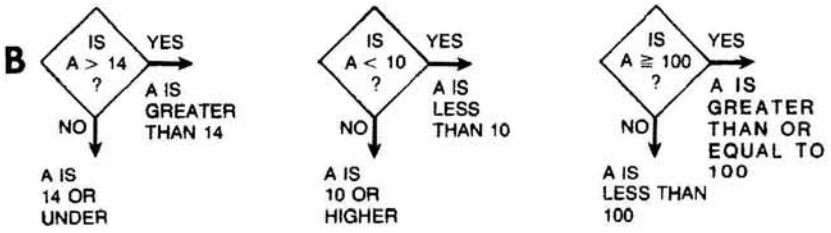


Figure 3-6
Mathematical decision making symbols and their use.

Constructing The Flowchart

Now that you can identify flowchart symbols, it's time to solve a few problems using flowcharts. The first, to add three numbers and produce a sum, is shown in Figure 3-7. The problem and the mathematical expression of the problem are shown on the left. The mathematical expression serves as a definition of the problem. Our flowchart is on the right.

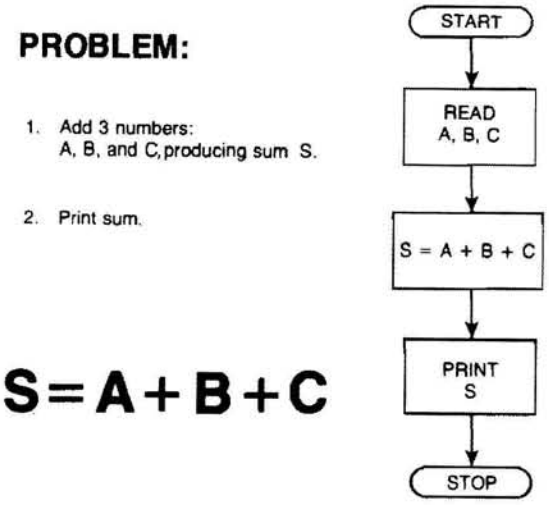


Figure 3-7
Flowchart construction.

It begins with the start terminal symbol. In a flowchart of this size, you might think that this symbol is unnecessary. But, no matter how simple you think a flowchart is, always use the start terminal symbol. You will see later on in this course that flowcharts can get quite lengthy, and it is often necessary to return to the beginning of your flowchart. The start terminal symbol allows you to do this with ease. If you use it all the time, even when it seems unnecessary, then you'll never get into difficulty locating the beginning of the program.

Next you find an operation box that tells the computer to read, or identify, the variables A, B, and C. This process would actually move the numbers from memory into the microprocessor. Again, you might think that this box is unnecessary or that it could be combined with the next operation box. But, remember, the more detailed the flowchart, the easier it is to write your program from it.

In the second operation box, the microprocessor is instructed to add A, B, and C, giving the total S. The final operation box tells the microcomputer to print the solution, S. Finally, the stop terminal symbol is used to end the process.

Now that you have seen how it is done, it's time to look at a more complex problem. In Figure 3-8 you see a drawing of a traffic intersection. Main Street is a busy thoroughfare, while Side Street handles only a moderate amount of traffic. A traffic study shows that if traffic on Main Street is allowed to move for two minutes, while traffic on Side Street is allowed to move for one minute, the intersection controls traffic very efficiently. Light A has been assigned for control of traffic on Main Street and light B for control of traffic on Side Street.

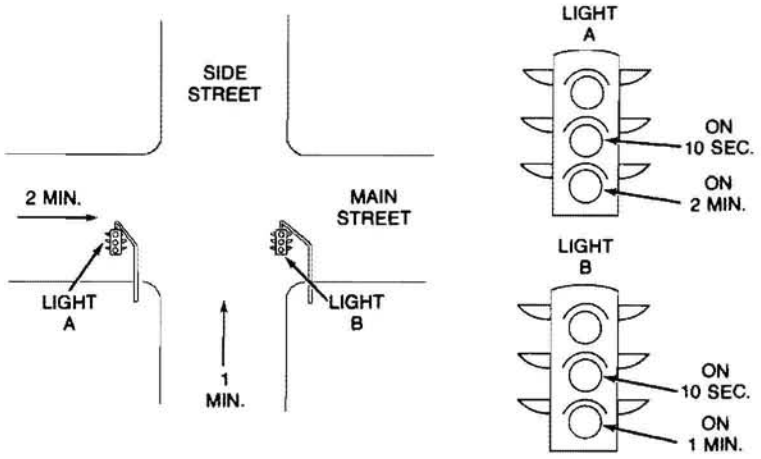


Figure 3-8
The traffic problem.

It is not necessary at this point, but you might like to try to draw this flowchart yourself. If not, just read along while we work our way through the problem. Before you can attempt to draw a flowchart, you must carefully examine the problem. That is, you must carefully determine exactly what each light must do.

Light A must remain green for two minutes. Naturally, you want a caution light between read and green. We suggest a yellow light duration of 10 seconds. The duration of the red light will be controlled by light B. Light B, on the other hand, will be green for one minute and yellow for ten seconds. When light B turns red, light A turns green and the cycle is repeated.

With this information, you should encounter little difficulty drawing the flowchart for this problem. Defining the problem in detail will always simplify problem solving. If you care to try your hand at this flowchart, grab a pencil and some paper and give it a try. If not, just continue reading. You'll find our solution in Figure 3-9.

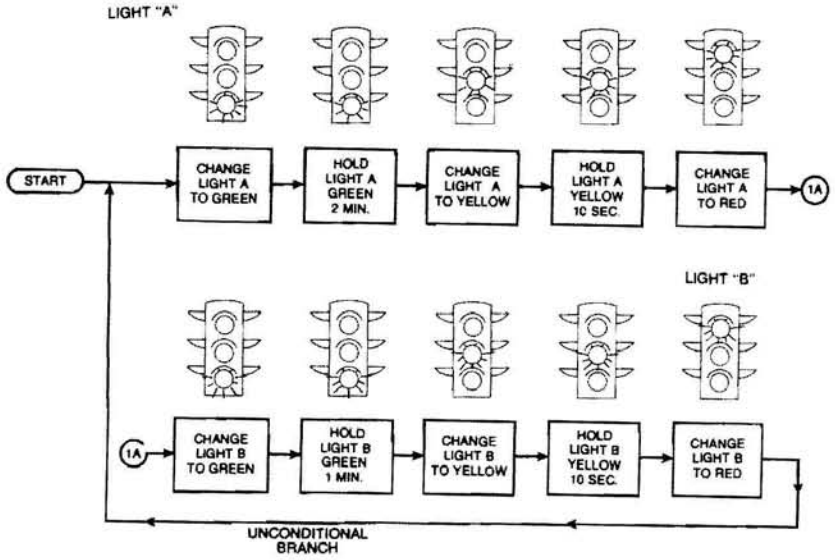


Figure 3-9
The traffic solution.

The flowchart begins in the top left-hand corner of Figure 3-9. The first five operation blocks control the operation of light A. You begin the program at the box that changes light A to green. The second operation holds the light green for two minutes. After this, light A must change to yellow and hold there for ten seconds as the next two blocks indicate. Finally, light A is changed to red by the fifth operation.

Connector 1A tells you that the flow of the program picks up at the corresponding 1A connector in the bottom left-hand corner of the illustration. Here, you enter the light B routine that controls the traffic on Side Street. Now that the traffic on Main Street has a red light, it is safe to change light B to green. Light B is held green for one minute, then it is changed to yellow and held for ten seconds. Finally, light B changes to red. Of course, you want this process to repeat, so you draw a flow line from the end of the light B cycle back to the beginning of the light A cycle.

This type of a flow line is called an unconditional branch, because it must always return the program to a particular point in the flowchart. The unconditional branch would correspond to an unconditional jump in the actual program code. Keep this in mind when we discuss program jumps in the next section.

There are other flow lines that originate at a decision box rather than at an operation box. These are called conditional branches because the flow of the program along these lines is based on the conditions stated in the box. Conditional branches in a flowchart are related to conditional jumps in an actual program.

Figure 3-10 demonstrates the use of the decision box and the conditional branch. The problem is stated on the left and the flowchart solution is shown on the right.

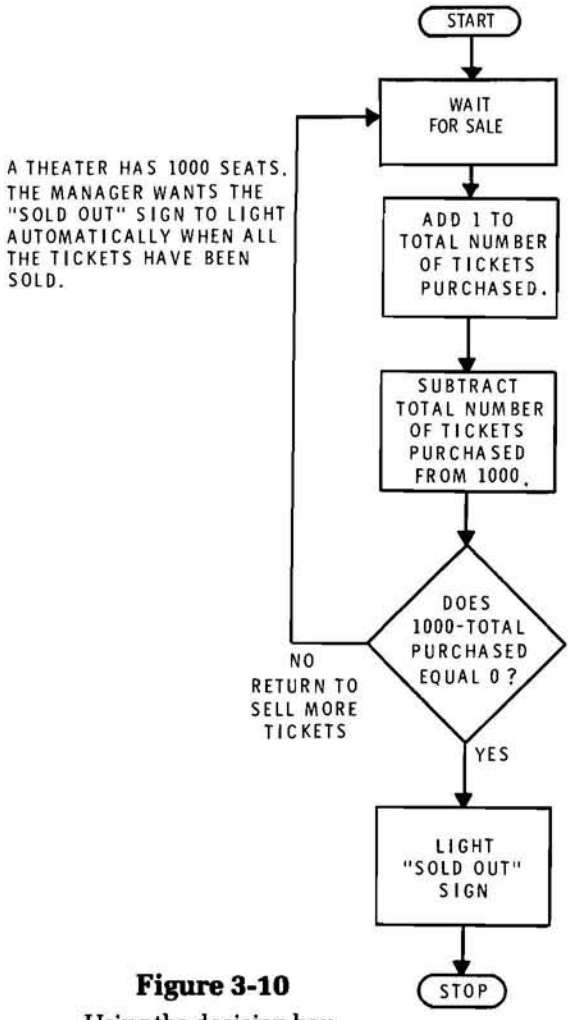


Figure 3-10
Using the decision box.

Again, the chart begins with the terminal start symbol. As each ticket is purchased, it is added to the total number of tickets purchased. Then the total number of tickets sold is subtracted from the number of seats available.

Now, the decision must be made. If all tickets have not been sold, you must branch to the beginning of the flowchart and resume the count. If all tickets are sold (if the result of the subtraction is zero), then ignore the branch and light the "SOLD OUT" sign. Once the sign is lit, you complete the flowchart with the stop terminal symbol. Although the problem is very simple, there is no way to solve it without using a decision box and conditional branch in the flowchart.

Self-Review Questions

1. The five most common flowcharting symbols are the:

   A. _____
   B. _____
   C. _____
   D. _____
   E. _____

2. You are permitted to use mathematical symbols in a flowchart.

   _____
   True/False

3. The flowchart should always begin with a start terminal symbol.

   _____
   True/False

4. The _____ _____ in a flowchart corresponds to the unconditional jump in the actual program.

5. The _____ _____ in a flowchart corresponds to the conditional jump in the actual program.

NOTE: The Self-Review Answers are located at the end of this unit, beginning on Page 3-67.

JUMPS

The programs in Unit 2 used the "straight line" method of processing code. That is, the instructions in those programs were executed one after another. As you found out, programs of this type can be limited in scope. As an example, let's take another look at the multiply-by-repeated-addition program from Experiment 2. Refer to Figure 3-11.

```
TITLE EXPERIMENT 1 -- PROGRAM 2 -- SIMPLE MULTIPLICATION
;
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
COM_PROG  SEGMENT               ;Beginning of program segment
;
MULTI   EQU     7               ;Equate program multiplicand value
        ORG     100H            ;COM programs always start here
START:  SUB     AL,AL           ;Clear the PRODUCT register
        MOV     BL,MULTI        ;Get the multiplicand,
        ADD     AL,BL           ;then perform the
        ADD     AL,BL           ;multiplication operation
        ADD     AL,BL           ;by adding the multiplicand
        ADD     AL,BL           ;to the PRODUCT register
        ADD     AL,BL           ;5 (multiplier) times
        MOV     PRODUCT,AL      ;Store the product
;
PRODUCT DB      0               ;Reserve one byte in memory
                                ;and initialize the byte to 0
COM_PROG  ENDS                  ;End of program segment
        END     START           ;End of program, point to beginning
```

Figure 3-11

Multiplication program from Experiment 2.

The program multiplies the number 7 by the number 5. This works well enough as long as the multiplier is a relatively small number. But, if the multiplier is 50, then the length of the program becomes prohibitive both in time invested to write the program and memory used to store the program. Obviously, some better technique must be employed.

Virtually every program uses a programming method called the loop. The loop allows a section of the program to be run as often as needed. For instance, to multiply 7 by 50 in our multiply program, you simply delete all but one of the ADD instructions, and then repeat that instruction 50 times. The 8088 MPU has a number of instructions that allow you to create a loop. Among these are the jump instructions, the subject of this section.

Unconditional Jump

Although there are a number of jump instructions, for now, we will only consider the **unconditional jump** instruction. The unconditional jump instruction has one mnemonic, JMP, that can assume three different characteristics depending on how it is used in a program. The three characteristics are jump short, jump normal, and jump far. Each allows the microprocessor to escape the straight line instruction sequence by altering the direction of program flow. That is to say, these instructions alter the contents of the IP register, thereby changing the sequence in which the instructions in a program are executed. For now, we will limit our discussion to short and normal jumps.

The **jump short** instruction uses a signed, 8-bit offset value that allows a "jump" range of 256 bytes of memory. It does this by causing a forward jump of up to +127 bytes or a backward jump of up to −128 bytes. The ability to span 256 bytes of memory constitutes a "short" jump.

The **jump normal** instruction is similar to the jump short instruction, only it uses an unsigned, 16-bit offset value. Thus, it can cause a forward jump of up to 65,536 bytes. This is the default jump used by the MACRO-86 assembler.

Both the short and normal jumps are what we call **intrasegment** jumps; they occur within the 64K boundary of a program segment.

Each time you use a JMP instruction, you must specify the instruction's target. The **target** is the place, or memory location, in the program to which you wish to jump. However, before you can understand how to specify the target, you should understand how the address of the target is calculated.

DIRECT JUMPS

A JMP instruction is generally considered to be **direct**, when its operand contains a relative displacement value. When you write a direct JMP instruction, you identify the memory location, or **target**, of the jump with a label in the operand field. The assembler then calculates the relative displacement by identifying the offset address of the target. Let's see how this process works.

Figure 3-12 is our simple multiplication program that has been modified by the deletion of a few ADD instructions and the addition of a JMP instruction and a target label. The JMP instruction alters the sequence of program execution and, as a result, reduces the number of ADD instructions necessary in the program. The label TIMES serves as the target of the JMP instruction.

```
TITLE UNIT 3 -- PROGRAM 1 -- MULTIPLICATION THROUGH A LOOP
;
COM_PROG  SEGMENT               ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
MULTI   EQU     7               ;Equate program multiplicand value
        ORG     100H            ;COM programs always start here
START:  SUB     AL,AL           ;Clear the PRODUCT register
        MOV     BL,MULTI        ;Get the multiplicand, then perform
TIMES:  ADD     AL,BL           ;the multiplication operation by
        JMP     TIMES           ;repeating the addition operation
        MOV     PRODUCT,AL      ;Store the product
;
PRODUCT DB      0               ;Reserve one byte in memory
                                ;and initialize the byte to 0
COM_PROG  ENDS                  ;End of program segment
        END     START           ;End of program, point to beginning
```

Figure 3-12

Multiplication using repeated addition and a direct jump.

When the assembler "sees" the JMP instruction with the specified target in its operand field, the assembler calculates a displacement relative to the present contents of the Instruction Pointer and the instruction containing the target label. It then stores this displacement as the operand of the JMP instruction. When the program is run, and the JMP instruction executed, the MPU **adds** the displacement value to the contents of the IP. As you know, the IP always points to the next instruction to be executed. Adding the displacement to the IP causes the IP to now point to the target specified in the JMP instruction.

The relative displacement is a number which indicates either a forward jump, further into the program, or a backward jump, to some point already past. This is handled in one of two ways. In a short jump, the displacement can be either a positive or a negative 8-bit value as described earlier. If the jump is forward, the displacement is a normal 8-bit value. However, if the jump is backward, the displacement is a 2s complement of the 8-bit negative value.

In a normal jump, the displacement is an unsigned 16-bit value that is added to the contents of the IP register. Backward jumps are handled by the address "wraparound" process described earlier. For example, suppose you wanted the program to jump backward, from address offset 0112H to address offset 010CH. The assembler would calculate a displacement value of 0FFFAH and place it in the operand. Thus, when the jump is executed, the jump target is determined by adding the jump displacement to the contents of the IP to produce a new IP value that points to the specified target. The calculation follows:

DISPLACEMENT	0FFFAH
IP REGISTER	+ 0112H
TARGET	1010CH

Notice that the calculation exceeded the 16-bit capacity of the IP register. The overflow of one is ignored, producing the new IP value 010CH.

By now, you may be wondering what determines the type of jump (short or normal) used. The assembler makes that determination by identifying the offset of the target. If it is within the 256-byte boundary of the JMP instruction, an 8-bit displacement is automatically generated; otherwise, a 16-bit displacement is generated.

Direct jumps are by far the most common type of unconditional jump. There is, however, another form called the indirect jump.

INDIRECT JUMP

A direct jump contains a relative displacement value in its operand to point to the jump target. The **indirect jump**, on the other hand, uses one of the MPU general registers to hold the "actual address" of its jump target. When an indirect jump instruction is executed, the contents of its "address" register are used to **replace** the contents of the IP register. For example, after the instructions

```
MOV BX,0FF3EH
JMP BX
```

are executed, the contents of the IP register are replaced with the value 0FF3EH. Thus, the next instruction to be executed is located at address offset 0FF3EH.

Conditional Jumps

All JMP instructions are unconditional. That is, when the MPU encounters a JMP instruction, it will always execute that instruction. Look at Figure 3-12 once more. You should discover a problem. The program loops back to the ADD instruction well enough, but it will continue to perform this loop forever. There is no provision in the program to get out of, or exit, the loop.

In order to exit the loop, the MPU must make a decision; namely, when should the loop be terminated? The ability to make a decision is, after all, the real power of the microprocessor. Before the MPU can make that decision, it must have information on which to base the decision.

THE FLAG REGISTER

The 8088 MPU bases its "decisions" on the contents of the Flag register. The Flag register is a 16-bit register that contains, among other things, information about the last mathematical or logical operation performed. (From now on, we'll call these operations **arithmetic operations**.) Six bits, or flags, within the Flag register actually contain information about these arithmetic operations. They are called **condition code** flags. Figure 3-13 shows the location of the six condition code flags in the Flag register.

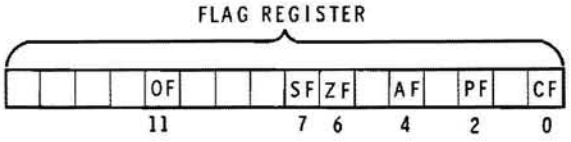


Figure 3-13
The condition code bits in the Flag register.

Like all other registers within the 8088 MPU, the contents of the condition code flags can be either ones or zeros. If the contents of a particular flag is one, we say that flag is **set**. On the other hand, if the flag contains a zero, then we say that flag is **clear**.

The first flag we will describe is the CF, or **carry flag**. This flag is set if there is a carry out of, or a borrow into the most significant bit in either 8-bit or 16-bit arithmetic operations, as shown in Figure 3-14. If the last arithmetic operation did not cause a carry or borrow, the CF will be clear.

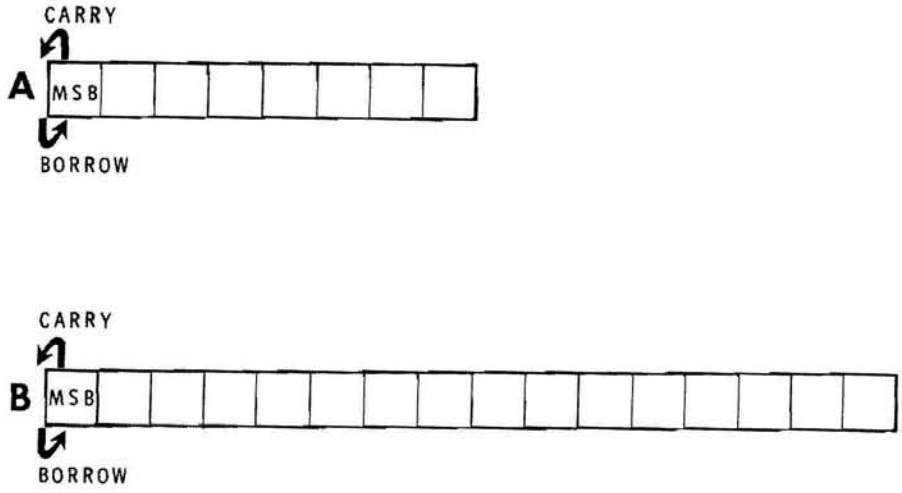


Figure 3-14
Conditions that set the Carry flag.

The next flag is the PF, or **parity flag**. The 8088 MPU sets the parity flag if the result of a byte-sized arithmetic operation has an even number of 1-bits. That is, if the result of an add operation is 11000011B, the parity flag is set because the result contains an even number of 1-bits. Likewise, if the result of the last math operation is 00000001B, the PF is clear, indicating an odd parity for the result. If you perform an arithmetic operation using a 16-bit register, the parity flag will only test the low byte for parity.

The third flag is the AF, or **auxiliary carry flag**. Two conditions will cause this flag to be set. The first is a borrow from bit 4 of a byte during an arithmetic operation involving an 8-bit quantity. The second condition that will cause this flag to be set, is a carry from bit 3 during an 8-bit arithmetic operation. This is shown in Part A of Figure 3-15. As Part B shows, a carry out of bit 3 to bit 4, or a borrow from bit 4 to bit 3 during a 16-bit arithmetic operation will also set the auxiliary flag. Therefore, whether you are dealing with an 8-bit or a 16-bit arithmetic operation, bits 3 and 4 affect the status of the auxiliary flag.

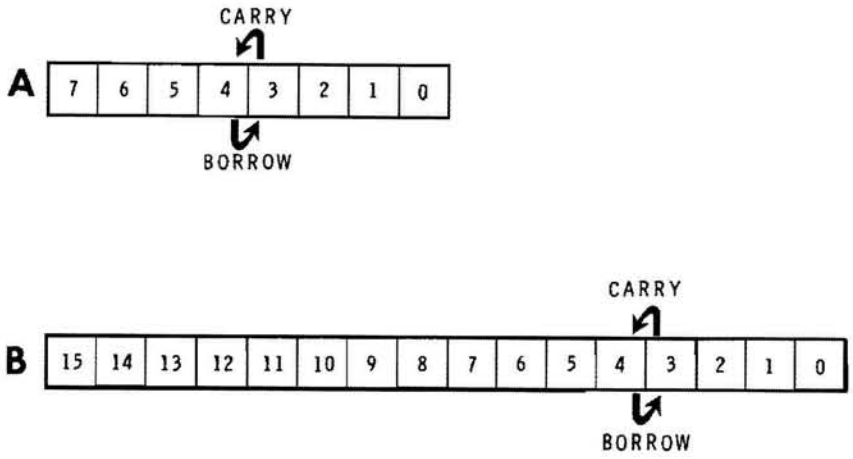


Figure 3-15
Conditions that set the Auxiliary Carry flag.

The fourth flag shown is the ZF, or **zero flag**. This flag bit is set only if the result of the last arithmetic operation is zero. If the result of the last arithmetic operation is a number other than zero, this bit is clear.

The fifth flag shown in the register is the SF, or **sign flag**. This flag is set if the result of the last arithmetic operation performed is less than zero. That is, it is set if the last arithmetic operation resulted in a negative number. If the result of the last arithmetic operation was zero, or a positive number, then this flag is clear. During an arithmetic operation, the MPU treats the most significant bit as the sign bit. If the bit is zero, the MPU assumes the result was positive, and it clears the sign flag. By the same token, if the most significant bit is one, the MPU assumes the result was negative, and it sets the sign flag.

The last flag we will discuss is the OF, or **overflow flag**. This flag is set if the result of the last arithmetic operation is larger than the destination location. Recall that we said the MPU considers the most significant bit the sign bit. Thus, the largest value that the MPU will recognize is a 7-bit or 15-bit value. Naturally, all 8 or 16 register bits are still used, but the MPU indicates an overflow into the most significant bit by setting the overflow flag. Because the result of an arithmetic operation could fool the MPU with regard to an overflow condition, the MPU makes two tests. First, it checks for a carry into, or a borrow from, the most significant bit. Next, it checks for a carry out of, or a borrow into, the most significant bit. These conditions are shown in Figure 3-16. If there was no carry, or if there was a carry into **and** out of the most significant bit, the OF is clear. If, on the other hand, there was a carry into **or** out of the most significant bit, the OF is set. As before, a borrow is treated the same as a carry.

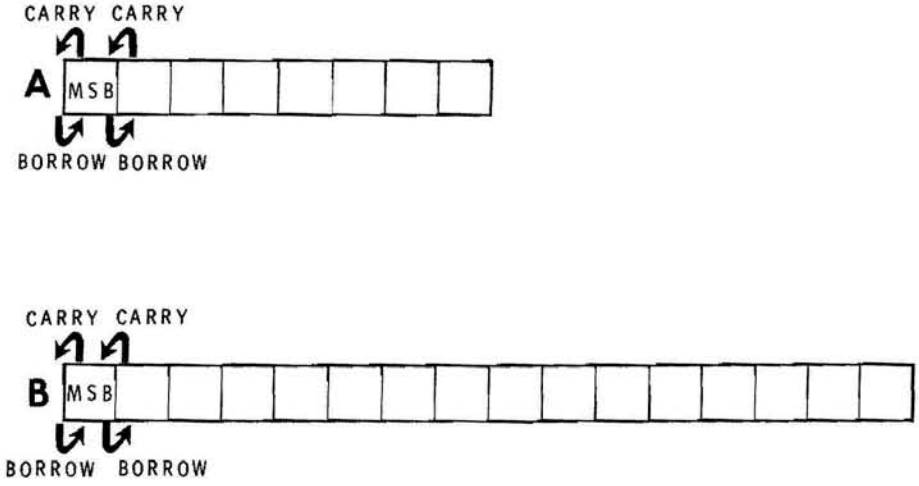


Figure 3-16
Conditions that affect the Overflow flag.

THE CONDITIONAL JUMP INSTRUCTIONS

The 8088 MPU instruction set has a number of conditional jump instructions. These instructions are used to make decisions concerning the flow of the program. That is, if a certain set of conditions exist, the program will continue in a straight line. However, if another set of conditions exist, which are unique for each instruction, the MPU makes the decision to take the conditional jump. During the execution of a conditional jump instruction, the MPU tests the contents of the Flag register or, to be more specific, tests the contents of certain bits in the Flag register.

The conditional jump instructions are listed in Figure 3-17. In this figure, the first column contains the mnemonics for the various conditional jump instructions.

CONDITIONAL JUMP INSTRUCTION	JUMP IF:	JUMP IS TAKEN IF:
JA	ABOVE	(CF OR ZF) = 0
JNBE	NOT BELOW NOR EQUAL	(CF OR ZF) = 0
JNB	NOT BELOW	CF=0
JAE	ABOVE OR EQUAL	CF=0
JB	BELOW	CF=1
JNAE	NOT ABOVE NOR EQUAL	CF=1
JC	CARRY	CF=1
JBE	BELOW OR EQUAL	(CF OR ZF) =1
JNA	NOT ABOVE	(CF OR ZF) =1
JE	EQUAL	ZF=1
JZ	ZERO	ZF=1
JNLE	NOT LESS NOR EQUAL	[(SF XOR OF) OR ZF]=0
JG	GREATER	[(SF XOR OF) OR ZF]=0
JLE	LESS OR EQUAL	[(SF XOR OF) OR ZF]=1
JNG	NOT GREATER	[(SF XOR OF) OR ZF]=1
JNL	NOT LESS	(SF XOR OF) = 0
JGE	GREATER OR EQUAL	(SF XOR OF) = 0
JL	LOWER THAN	(SF XOR OF) = 1
JNGE	NOT GREATER NOR EQUAL	(SF XOR OF) = 1
JNC	NO CARRY	CF=0
JNE	NOT EQUAL	ZF=0
JNZ	NOT ZERO	ZF=0
JNO	NOT OVERFLOW	OF=0
JNP	NO PARITY	PF=0
JPO	PARITY ODD	PF=0
JNS	POSITIVE	SF=0
JO	OVERFLOW	OF=1
JP	PARITY	PF=1
JPE	PARITY EVEN	PF=1
JS	SIGN	SF=1

Figure 3-17
Conditional jump instructions.

The second column states the conditions that must be present if the jump is to occur. For example, the conditions for the JNZ instruction are jump if the result of the last arithmetic operation is not zero, or jump if not zero. As you can see, the mnemonic JNZ is an abbreviated form of this statement.

The third column shows the flag, or flags, that are tested and any logic combination that occurs during the testing. An OR in this column indicates that the MPU performs a logical inclusive OR operation on the contents of the specified flags. Likewise, an XOR in this column indicates a logical exclusive OR operation. These logic operations are done automatically by the MPU when it executes the conditional jump instructions.

USING THE CONDITIONAL JUMP

The conditional jump instructions have two things in common with the unconditional jump short instruction. First, all conditional jump instructions are direct jumps; there are no indirect conditional jumps. Second, the conditional jump is limited to a range of 256 bytes. It can cause a forward jump of up to +127 bytes or a backward jump of up to −128 bytes.

Now that you are familiar with the Flag register and some of the mathematical operations that the MPU can perform, let's see exactly how a conditional jump instruction works.

Most programs make some type of decision. Some frequently encountered decisions are:

"Which number is larger?"

"Are these two numbers equal?"

"Is this an even number of bits?"

"Has the program loop been repeated enough times?"

Conditional jumps are used to make these decisions. To see how this is done, let's look at our multiplication program one more time. This time, however, the unconditional jump is replaced with a conditional jump. The conditional jump is used to decide when the program is finished. To do this, the MPU must decide if a number is or is not zero. Figure 3-18 shows the program.

```
TITLE UNIT 3 -- PROGRAM 2 -- CONDITIONAL LOOP MULTIPLICATION
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
MULTIPLICAND    EQU     7
MULTIPLIER      EQU     5
;
        ORG     100H             ;COM programs always start here
START:  SUB     AL,AL            ;Clear the PRODUCT register
        MOV     CL,MULTIPLIER    ;Set multiplier as operation counter
        MOV     BL,MULTIPLICAND ;Get the multiplicand, then perform
TIMES:  ADD     AL,BL            ;the multiplication operation by
                                 ;adding the multiplicand to itself
                                 ;"count" times
        DEC     CL               ;Decrement the "count" after the add
        JNZ     TIMES            ;and repeat if "count" not zero
        MOV     PRODUCT,AL       ;Store the product if "count" zero
;
PRODUCT DB      0                ;Reserve one byte in memory
                                 ;and initialize the byte to 0
COM_PROG  ENDS                   ;End of program segment
        END     START            ;End of program, point to beginning
```

Figure 3-18

Multiplication using repeated addition and a conditional jump to form a program loop.

Notice that we have enhanced a few areas of the program and added a couple of instructions. First of all, both the multiplier and the multiplicand are identified with equate statements. By using the full name for each symbol, we've eliminated the need for comments; and at the same time, made the program easier to follow.

The first version of this program added the multiplicand value seven to the AL register five times, to simulate the multiplication operation 7 times 5. In this version, a loop is used to perform the add operation. The instruction JNZ (jump while not zero — ZF clear) is used to monitor the loop.

As before, the multiplicand is loaded into the BL register. The multiplier is loaded into the CL register to provide a "count" for the multiplication by repeated addition operation. After each addition operation, the count is decremented. As long as the count is not zero, the zero flag is clear.

When the JNZ instruction is executed, the ZF is tested. If the flag is clear, the condition for a jump is true, and the jump is taken; the program loops back to the target label TIMES. As long as the count is not zero, the loop will continue. When the count reaches zero, the zero flag will be set. This time, the condition for a jump is false, and the jump is **not** taken. Instead, the next sequential instruction is executed. When a conditional jump is not taken, we say that the program "falls through" to the next instruction.

If you follow the program through, you can see that the CL register will not be zero until the program loop is accomplished five times. Since the addition occurs five times, you have, in effect, multiplied the number 7 times the number 5.

It would be nice if the microprocessor could make the proper decisions without any help from us, but this is not possible. As a programmer, you must set up the desired parameters for your program and use the correct instructions in the proper sequence. There are, no doubt, a number of ways to write this same program that will work just as well as the example program. One of the most important things about programming is to understand exactly how each instruction works before using it in a program.

Before we look at any more examples of conditional jump instructions, there is one more arithmetic instruction you should learn. That is the CMP, or **compare** instruction. When this instruction is executed, the source operand is subtracted from the destination operand. The mathematical subtraction alters Condition flags, but the operands are unaffected. Examples of this instruction are:

```
CMP AX,DX
CMP AX,SYMBOL
CMP AX,0F0H
```

In the first example, the contents of the DX register are subtracted from the AX register. This operation alters the flags, but does not change the contents of either register. As the name compare implies, the operation is simply a comparison.

In the next two examples, the constant is subtracted from the contents of the AX register. Again, only the flags are affected; the contents of the AX register do not change.

Now let's take a closer look at a couple of conditional jumps to give you an idea of the "considerations" that go into MPU decision making.

First look at the jump if above (JA) instruction in Figure 3-17. You can see that this instruction tests both the carry flag and the zero flag. To test these conditions for the conditional jump, the MPU performs a logical OR operation on the contents of the two flags. The following program segment demonstrates the use of the JA instruction.

```
MOV AX,0002H
CMP AX,0001H
JA  THERE
```

The results of the CMP (compare) operation are quite obvious. Since the CMP subtracts the source operand from the destination operand, it is apparent that the result of this operation is a number above zero. The jump is taken. But, how does the MPU actually make the jump decision?

First, look at the individual flags that are "considered" during the execution of the JA instruction. The first flag tested is the carry flag. As you know, subtracting the number 1 from the number 2 does not generate either a carry or a borrow. Therefore, the CF, after the execution of the CMP instruction, is cleared.

The zero flag is also tested when the JA instruction is implemented. Since the mathematical result of the CMP instruction is not zero, this flag is also cleared.

The MPU now does a logical inclusive OR using the contents of the two flags. If you check the conditions listed in column three of Figure 3-17, you will see that the results of the calculations satisfy these conditions and the jump is taken. If either the CF or the ZF had been set, the OR operation would have produced a one, and the jump would not be taken.

As long as the mathematical result in the AX register for this program is above 1, the jump will be taken.

There is some difficulty, however, if you are dealing with signed numbers. The 2s complement of a negative number may be, at least in magnitude, greater than a positive number. For example, if the 2s complement for −1, 0FFFFH, is in the AX register, then the JA is taken even though this number is, in reality, less than one. This occurs because the JA instruction does not test the sign flag. Therefore, the JA instruction should only be used when dealing with unsigned numbers.

If you must work with signed numbers, you will have to use an instruction that tests the sign flag. An example of this type of instruction is the jump if greater (JG) instruction.

A glance at Figure 3-17 will confirm that the JG instruction does indeed test the sign flag. However, it also tests the overflow and zero flags. Let's look at an example where JG could be used:

```
MOV AL,0FH
CMP AL,0FFH
JG  THERE
```

In this program segment, the immediate value 0FFH is compared to the contents of the AL register, 0FH. The result of the compare is a signed number less than zero. The compare operation does not generate an overflow, since the result of the operation is not larger than eight bits. Therefore, the overflow flag is zero.

The compare operation, however, did result in a negative number. Hence, the sign flag is set to indicate that the result is less than zero.

The MPU now accomplishes the first part of a 2-part decision. An exclusive OR operation is performed on the contents of the sign and overflow flags. This operation produces a one.

Next, the MPU checks the contents of the zero flag. The result of the compare operation is not zero, so the ZF is clear, or zero. Now the contents of the ZF is ORed with the result of the previous exclusive OR operation. Again, the result is a one.

However, the conditions stated in Figure 3-17 indicate that the jump if greater operation will be taken only if the result of the logical processes on the contents of the flag register is zero. Since this is not the case, the jump is not taken, and the program falls through to the next instruction.

All of this may seem pretty complex. Setting flags, comparing results, and making decisions can get confusing, to say the least. One consolation, though, is the fact that all of the computations take place within the MPU. You will never see the actual computations occur; only the effect that they have on your program. You must always be extremely careful, however, that you present the correct instructions in the correct format, so that the MPU will act as you want it to in your program.

Self-Review Questions

6. Virtually every program uses a technique called a _____.

7. State the purpose of the loop. _____
_____
_____

8. The _____ instruction allows the microprocessor to escape the straight line program sequence.

9. Each time you use a jump instruction, you must specify the instruction's _____.

10. The _____ is the place, or memory location, in the program to which you wish the program to jump.

11. When the address of the target is calculated using a relative value, the instruction is said to be a _____ jump.

12. In a/an _____ jump, the address of the target is contained in a register. This address replaces the contents of the IP register when the instruction is executed.

13. What is the real power of the microprocessor? _____
_____

14. The 8088 MPU bases its decisions on the contents of the _____ register.

15. The CMP instruction subtracts the source operand from the destination operand and stores the result in the AX register. _____
True/False

16. The _____ jump instruction removes the MPU from the decision making process.

17. The JNZ instruction will cause the program to jump to the target if the zero flag is _____.

18. The _____ instruction should not be used with signed arithmetic operations.
JA/JG

LOOPS

The jump instructions are powerful programming tools, in that they can force an unconditional change in what would otherwise be a straight-line program. They can also provide a form of artificial intelligence and alter the sequence, given specific conditions. The "conditional jump" often takes the form of a program loop, or set of instructions that repeat in a cycle. However, once in a loop, it is often necessary to manipulate data in the general registers or test the jump "condition flags" in order to get out of the loop. This is shown in Figure 3-18, our multiplication-through-repeated-addition program.

```
TITLE UNIT 3 -- PROGRAM 2 -- CONDITIONAL LOOP MULTIPLICATION
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
MULTIPLICAND    EQU     7
MULTIPLIER      EQU     5
;
        ORG     100H             ;COM programs always start here
START:  SUB     AL,AL            ;Clear the PRODUCT register
        MOV     CL,MULTIPLIER    ;Set multiplier as operation counter
        MOV     BL,MULTIPLICAND ;Get the multiplicand, then perform
TIMES:  ADD     AL,BL            ;the multiplication operation by
                                 ;adding the multiplicand to itself
                                 ;"count" times
        DEC     CL               ;Decrement the "count" after the add
        JNZ     TIMES            ;and repeat if "count" not zero
        MOV     PRODUCT,AL       ;Store the product if "count" zero
;
PRODUCT DB      0                ;Reserve one byte in memory
                                 ;and initialize the byte to 0
COM_PROG  ENDS                   ;End of program segment
        END     START            ;End of program, point to beginning
```

Figure 3-18

Multiplication using repeated addition and a conditional jump to form a program loop.

The DEC instruction reduces the loop "count" after every ADD operation. The JNZ instruction causes the loop to continue until the count is zero. At that time, the program falls through to the next instruction.

Although the jump instructions as a group are a handy way to create a program loop, there is another set of instructions that, in most circumstances, do a better job. These are, as you might have guessed, called **loop** instructions. Like the jump instructions, loop instructions can take two forms: conditional and unconditional loops. Let's take a look at the unconditional loop instruction first.

Unconditional Loop

The instruction **LOOP** is considered an unconditional instruction. It transfers control to the instruction indicated by the target operand, regardless of Flag register condition. So how, you might ask, does the program get out of the loop?

This is controlled by the CX register. Earlier, we said the CX register was simply a general register that could handle high and low bytes (8 bits per byte), as well as words (16 bits) of data. It also serves as the **Count** register for loop instructions. Hence its common name, "Count register."

The "count down" process works like this: First, a value is stored in the Count register. Now every time the LOOP instruction is executed, the Count register is automatically decremented, and then it is tested to see if it is zero. When the register value becomes zero, the "loop" is ignored, and the program falls through to the next instruction. Naturally, you shouldn't try to store data in the CX register during a loop. You could wind up looping forever. To see how the LOOP instruction works, let's examine a modified version of our multiplication-through-repeated-addition program. Refer to Figure 3-19.

```
TITLE UNIT 3 -- PROGRAM 3 -- UNCONDITIONAL LOOP MULTIPLICATION
;
COM_PROG  SEGMENT               ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
MULTIPLICAND    EQU     7
MULTIPLIER      EQU     5
;
        ORG     100H            ;COM programs always start here
START:  SUB     AL,AL           ;Clear the PRODUCT register
        MOV     CX,MULTIPLIER   ;Set multiplier as operation counter
        MOV     BL,MULTIPLICAND ;Get the multiplicand, then perform
TIMES:  ADD     AL,BL           ;the multiplication operation by
                                ;adding the multiplicand to itself
                                ;"count" times
        LOOP    TIMES           ;Decrement the "count" after the add
                                ;and repeat if "count" not zero
        MOV     PRODUCT,AL      ;Store the product if "count" zero
;
PRODUCT DB      0               ;Reserve one byte in memory
                                ;and initialize the byte to 0
COM_PROG  ENDS                  ;End of program segment
        END     START           ;End of program, point to beginning
```

Figure 3-19

Using the unconditional LOOP instruction.

Compared to the previous program, the LOOP instruction has reduced the physical length by only one instruction. That would seem hardly worth the effort, until you analyze where the instruction was deleted. Notice that DEC was removed from the "loop." Now, if you run the program using a multiplier of 1000, you will find the program has been effectively reduced from a length of 3004 instruction executions to 2004 instruction executions. This is a substantial savings.

One other change in the program occurred in the instruction that loads the multiplier. The register name was changed from CL to CX to meet the requirement of a count register.

Related to the unconditional LOOP instruction is the conditional, jump if count register zero (JCXZ) instruction. Essentially, it directs the program to jump to the address specified by its operand if the contents of the CX register is zero. This may appear insignificant, until you consider the consequences. Often, the data loaded into the Count register is a variable. Depending on the program application, the data could be any value between 0 and 0FFFFH. Now when a LOOP instruction is executed, the first step is to **decrement** the Count register. If the count was zero before the register was decremented, it will be 0FFFFH after the register is decremented. Thus, when the register is examined for zero, the second step in the LOOP instruction, zero will not be found and the program will loop to the indicated address.

Because 0FFFFH is now in the Count register, the program will have to loop that many more times before it can escape; normally an undesirable condition. By placing the JCXZ instruction just **after** the instruction that loads the Count register and **before** the program loop, you make sure the program jumps around the loop when the Count register is zero. Figure 3-20 is an example of how JCXZ could be implemented.

```
TITLE UNIT 3 -- PROGRAM 4 -- UNCONDITIONAL LOOP MULTIPLICATION
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
MULTIPLICAND    EQU     7
MULTIPLIER      EQU     5
;
        ORG     100H             ;COM programs always start here
START:  SUB     AL,AL            ;Clear the PRODUCT register
        MOV     CX,MULTIPLIER    ;Set multiplier as operation counter
        JCXZ    DONE             ;Multiplier zero, can't multiply
        MOV     BL,MULTIPLICAND ;Get the multiplicand, then perform
TIMES:  ADD     AL,BL            ;the multiplication operation by
                                 ;adding the multiplicand to itself
                                 ;"count" times
        LOOP    TIMES            ;Decrement the "count" after the add
                                 ;and repeat if "count" not zero
DONE:   MOV     PRODUCT,AL       ;Store the product if "count" zero
;
PRODUCT DB      0                ;Reserve one byte in memory
                                 ;and initialize the byte to 0
COM_PROG  ENDS                   ;End of program segment
        END     START            ;End of program, point to beginning
```

Figure 3-20

Testing the count register for zero, before a loop.

Again, we've used our multiply-through-repeated-addition program. The JCXZ instruction is placed so that it can check the value of the multiplier before the multiplication loop. If CX is zero, the program is forced to jump to the target label DONE. Any other value allows the program to fall through the JCXZ instruction to the next MOV instruction. While you are not required to use the JCXZ with the LOOP instruction, it is a good practice to do so when you have no control over the data that will be moved into the Count register.

Conditional Loop

Recall from our discussion on jumps, that a conditional instruction, by its nature, relies on a **flag** to determine whether the instruction should be executed or ignored. This is also true of the conditional loop instructions. However, in addition to responding to a flag, the conditional loop will also respond to the status of the Count register. Remember, **every** loop instruction relies on the Count register to determine the number of "loops" that should be executed. The flag simply determines whether the "conditions" are right for another loop. Thus, if the Count register content is zero, the loop will not be executed, regardless of flag condition. On the other hand, if the required flag condition occurs before the Count register reaches zero, the program will stop "looping" and proceed to the next instruction.

Each conditional loop instruction uses the zero flag to test program conditions. Depending on the instruction, execution will stop if the zero flag is set or clear. The four conditional loop instructions are:

LOOPE — Loop while equal
LOOPZ — Loop while zero
LOOPNE — Loop while not equal
LOOPNZ — Loop while not zero

However, there are only **two** conditional loop **opcodes**. This is because there are only two assigned conditions to be met. Either the zero flag (ZF) is set, meaning the previous arithmetic operation produced a zero result; or the zero flag is clear, meaning the previous arithmetic operation produced a value other than zero. Both LOOPE and LOOPZ will cause the program to loop if ZF = 1. More precisely, the loop will occur if ZF = 1 and CX $\neq$ 0 (CX does not equal 0). By the same token, both LOOPNE and LOOPNZ will cause the program to loop is ZF = 0 and CX $\neq$ 0.

Now this may seem an attempt to add complexity to an already complex subject. However, these instruction pairs are not truly redundant from a programming point of view. They help maintain a continuity in program concept. For Example, consider this section of a program:

```
CMP       AX,DATA    ;AX equal DATA?
LOOPE     COUNT      ;Yes, repeat count
MOV       ANS,BX     ;No, store value
```

Assuming the Count register contains some value greater than one at this time, what's this section of program doing? The first line compares the **word** value found at address DATA and the accumulator. If the values are equal, the zero flag is set; if not, the ZF is clear. Then the "loop if equal" instruction decrements the Count register and checks for zero. Since CX does not equal zero, the zero flag is checked. This indicates whether the values in the previous operation were equal (ZF = 1) or not (ZF = 0). If the contents of the accumulator do not equal the contents at DATA, no loop is performed, and the next instruction is executed, storing the contents of the BX register at address ANS.

The key word in this example is **equal**. Does the value in the accumulator **equal** the value at DATA? Loop if the result of the previous arithmetic operation is **equal**. LOOPZ would have given the same results, but it would not have "read" the same. Likewise, you would not perform a subtraction and then "loop if equal." You would use the "loop if **zero**" instruction.

Thus, you have two loop instructions that give the same result, but are used for different reasons. LOOPNE and LOOPNZ are another example of one opcode with two different mnemonics or meanings.

Loop Addressing

All loop instructions are **Instruction Pointer-relative**. That is, they can only transfer to target addresses that are within −128 to +127 bytes of the Instruction Pointer. Another way of stating this is: they are **SHORT** transfer instructions. Thus, the effective address of the transfer is calculated in the same manner as described for conditional jumps.

Self-Review Questions

19. An _____ operation will affect the condition of the Flag register.

20. The _____ flag is tested when a LOOPE instruction is executed.

21. The LOOP instruction is considered a conditional instruction.

_____
True/False

22. What is the first step performed when the LOOP instruction is executed? _____

23. The _____ instruction is used to test the CX register for zero.

24. The LOOPE instruction is considered a conditional instruction.
_____
True/False

25. If the Zero flag is clear, the LOOPE instruction will execute a program loop. _____
True/False

26. What other instruction has the same opcode as the LOOPNE instruction? _____

27. The maximum distance a program can loop backward with the LOOP instruction is _____ bytes.

EXPERIMENT

Program Transfer

OBJECTIVES:

1. *Demonstrate program branching through the use of conditional and unconditional jumps and loops.*
2. *Demonstrate the effect arithmetic operations have on the Flag register.*
3. *Demonstrate the flowcharting process.*

Introduction

The last experiment got you started on assembly language programming, but you were limited to straight-line programs. Sequential execution of the program instructions is a useful and important part of almost every program. However, that form of programming doesn't allow the microprocessor the opportunity to make processing decisions. This experiment resolves that problem by covering all of the aspects of program transfer using the jump and loop instructions.

We will begin by examining the effect arithmetic instructions have on the six condition code flags in the Flag register. Knowing how these flags respond is a prerequisite for understanding the conditional jumps.

Procedure

1. The program in Figure 3-21 exercises each flag to give you an opportunity to see how a flag responds to a particular operation. At the same time, it introduces three flag manipulation instructions. You will find that they can be useful for presetting a flag prior to an operation.

   Call up your editor and enter the program listed in Figure 3-21. (We will refer to this program as PROG1.ASM.) Notice that we have introduced a new type of comment at the beginning and near the middle of the program. Programmers often use this type of comment to highlight or give a short overview of a program section. This can be very handy when you have a multisection program and you are looking for a particular operation. Naturally, the semicolons preceding each line identify these overviews as comments to the assembler.

2. Assemble the program. Type the command "MASM PROG1,PROG1,PROG1,PROG1;" and RETURN. This is a quick method for getting the assembler to generate all of the appropriate files. Notice that you only have to identify the program name, without the file extension, for each file to be generated. The first name identifies the ASM file; the next, the OBJ file; the next, the LST file; and the last, the CRF file.

3. Now finish generating the COM file. Type "LINK PROG1" and RETURN. When the linker is finished, type "EXE2BIN PROG1.EXE PROG1.COM" and RETURN.

4. Call up the debugger and load your program into memory. Type "DEBUG PROG1.COM" and RETURN. Now type "R" and RETURN. The debugger has loaded your program into memory, displayed the register contents, and listed the first instruction of your program.

```
TITLE EXPERIMENT 3 -- PROGRAM 1 -- FLAG MANIPULATION
;
COM_PROG  SEGMENT               ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
;
;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
;This part of the program uses arithmetic instructions to indirectly modify x
;the contents of the Condition Code bits in the Flag register.              x
;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
;
;
        ORG     100H            ;COM programs always start here
START:  MOV     AX,0            ;1  Clear the AX register
                                ;without affecting the flags
        ADD     AL,1            ;2  Add 1 to 8-bit register
        ADD     AL,0FH          ;3  4-bit auxiliary carry
        SUB     AL,1            ;4  4-bit auxiliary borrow
        ADD     AL,71H          ;5  Clear AL and set MSB=1
        SUB     AL,1            ;6  Borrow from MSB
        ADD     AL,1            ;7  Carry into MSB
        ADD     AL,7FH          ;8  Fill AL with 1's
        CMP     AL,0FFH         ;9  Is AL all 1's?
        CMP     AL,0            ;10 Is AL zero?
        ADD     AL,1            ;11 Overflow AL register
        SUB     AL,1            ;12 Borrow into AL register
        ADD     AX,1            ;13 8-bit carry into AH?
        ADD     AX,0FEFFH       ;14 Fill AX with 1's
        ADD     AX,1            ;15 Overflow AX register
        CMP     AX,0            ;16 Is AX zero?
        CMP     AX,0FFFFH       ;17 Is AX all 1's?
        SUB     AX,1            ;18 Borrow into AX register
        SUB     AX,7FFFH        ;19 Clear AX and set MSB=1
        SUB     AX,1            ;20 Borrow from MSB
        ADD     AX,1            ;21 Carry into MSB
        ADD     AX,7FFFH        ;22 Fill AX with 1's
        INC     AX              ;23 Add 1, no carry
        DEC     AX              ;24 Subtract 1, no borrow
;
;
;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
;Direct manipulation of the Carry flag is possible with the Carry flag      x
;control instructions.  These instructions follow.                          x
;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
;
;
        STC                     ;25 Set Carry flag
        CLC                     ;26 Clear Carry flag
        CMC                     ;27 Complement Carry flag
;
COM_PROG  ENDS                  ;End of program segment
        END     START           ;End of program, point to beginning
```

Figure 3-21

Program to manipulate the contents of the Flag register.

As you single-step through the program instructions, keep your eye on the AX register and the eight flag status indicators. Figure 3-22 shows the debugger symbol used for each flag bit in its set and clear condition. For this experiment, however, you are interested in the six arithmetic operation condition code flags. These are identified by the letters NV, PL, NZ, NA, PO, and NC, which indicate that all of the flags are clear at this time.

FLAG NAME	SET		CLEAR	
Overflow	OV		NV	
Direction	DN	Decrementing	UP	Incrementing
Interrupt	EI	Enabled	DI	Disabled
Sign	NG	Negative	PL	Plus
Zero	ZR		NZ	
Auxiliary Carry	AC		NA	
Parity	PE	Even	PO	Odd
Carry	CY		NC	

Figure 3-22
Debugger Flag register bit status symbols.

5. Now single-step though the program, using the "T" (trace) debugger command, and observe the effect each instruction has on the AX and Flag registers. The comments in the program are numbered to help you follow the program.

   Step 1, the first instruction, may seem like a waste of time. The register is already zero. Remember, however, you should never rely on a register being zero. If it must start out zero, then make it zero. We used the MOV AX,0 instruction instead of SUB AX,AX because we didn't want to affect the flags. Recall that only arithmetic operations affect the flags.

   Step 2 adds one to the AL register in preparation for the next step. The flags are affected by this operation, but none of the values changed.

   Step 3 adds 15 to the AL register. Because there is a carry from the fourth bit to the fifth bit (assuming the first bit is bit 1 and the last bit is bit 8), the Auxiliary Carry flag is set (AC).

Step 4 subtracts one from the AL register. This time, there is an auxiliary borrow from the fifth bit. An auxiliary borrow is the same as an auxiliary carry, as far as the Auxiliary Carry flag is concerned, so the flag is again set. In addition, the AL register now contains an even number of ones, so the Parity flag is set (PE).

Step 5 zeros the lower seven bits in the AL register, leaving the eighth bit, the most significant bit (MSB) set. Again, there was an auxiliary carry, setting the flag (AC). In addition, the Sign flag (NG) and Overflow flag (OV) are set. This may be difficult to understand, since the program appears to be processing positive numbers, and there was no overflow out of the register. You know that the MPU can add and subtract 8- and 16-bit numbers. It can also process signed (positive and negative) numbers. A positive number is indicated by a zero in the most significant bit of the register, while a negative number is indicated by a one in the MSB. Because the MPU can't tell if you are processing signed or unsigned numbers, it must assume you could be processing either type. Therefore, the Sign flag will always echo the condition of the MSB, after an arithmetic operation.

The Overflow flag is a little more tricky. It looks for a carry into the MSB and a carry out of the MSB. It then uses an exclusive OR operation to determine the condition of the flag bit. If there was no carry, or if there was a carry into and out of the MSB, the Overflow flag is cleared (NV). If, on the other hand, there was a carry into the MSB, but no carry out of the MSB, the Overflow flag is set. In step five, there was a carry into the MSB, but no carry out of the MSB. Therefore, the Overflow flag is set (OV). Now you may wonder how you can determine whether the carry that set the Overflow flag was into or out of the MSB. It's really quite simple. Check the Carry flag. If it is set, then you know the carry was out of the MSB. By the same token, if the Carry flag is cleared (NC), the carry was into the MSB.

One last flag to be affected by the addition operation is the Parity flag. With only one 1-bit in AL, the parity is odd, and the Parity flag is clear (PO).

Step 6 subtracts one from the AL register. During the subtraction process, bit 4 borrows from bit 5 (auxiliary carry). This causes the Auxiliary Carry flag to be set (AC). The Overflow flag is set (OV) for the same reason. A borrow from the MSB is same as carry into the MSB. Because there was no borrow into the MSB, the exclusive OR operation causes the Overflow flag to be set (OV).

Step 7 yields the same results as step 5: the Overflow flag is set (OV), the Sign flag is set (NG), the Auxiliary Carry flag is set (AC), and the remaining flags are clear (NZ, PO, and NC).

Step 8 adds 7FH to the AL register, to fill the register with ones. The MSB is still one, so the Sign flag remains set (NG). The Parity flag is set (PE) because there are an even number of ones. No other flags are set by the operation.

Step 9 uses the compare operation to determine if the AL register contains all ones. Recall that the Compare instruction subtracts the immediate data in the instruction from the contents of the specified register, and uses the result of the operation to set the flags. However, the result of the operation is not returned to the register, as in a normal subtraction operation. The result of this compare sets the Zero flag (ZR) and the Parity flag (PE). The Zero flag was set because the result was zero. Remember, the Parity flag was also set (PE) because the result was zero. Remember, the Parity flag can only indicate an even or odd number of ones. Since there were no ones in the result, the count can't be odd. Therefore, the count must be even, and the PF is set.

Step 10 makes the same compare, only this time, the immediate data is zero. The result is 0FFH, so the Sign and Parity flags are set (NG and PE). Had there been any carries generated by the compare, either the Overflow, Auxiliary Carry, or Carry flags would have been set (OV, AC, or CY). Generally, when you make a compare, you are looking for a match between data. Therefore, the only flag of interest is the Zero flag.

Step 11 adds one to the AL register, causing it to overflow into the Carry flag. Thus, the Carry flag is set (CY). However, in this operation, the Overflow flag is clear (NV). This is because there was a carry into as well as out of the MSB. The exclusive OR result of these two carries is zero, hence the zero in the Overflow flag. Since the AL register now contains zeros, the Zero and Parity flags are set (ZR and PE), and the Sign flag is clear (PL). Finally, the add operation caused an auxiliary carry, setting the Auxiliary Carry flag (AC).

Step 12 subtracts one from a register containing zero. This causes a borrow into the MSB. Recall that a borrow is treated like a carry, so the Carry flag is set (CY). Since there was also a borrow out of the MSB, the Overflow flag is clear (NV). Finally, there was a borrow from bit 5, so the Auxiliary carry flag is set (AC). With the register now full of ones, the Sign and Parity flags are set (NG and PE).

Step 13 changes the focus of the program from the AL register to the AX register. Now when there is a carry from the eighth bit to the ninth bit of the register, the Carry flag is not set. This is because the Carry flag only looks at the MSB of the register. Since AX is 16 bits long, the Carry flag is only affected by a carry out of the sixteenth bit. The Auxiliary carry flag is set (AC) because of the carry from bit 4 to bit 5 in the low byte of the register. Notice that the Parity flag is also set (PE). You might think that it would be clear, to indicate odd parity. Remember, however, that the Parity flag only checks the low byte of a 16-bit register. With no ones in the low byte, the Parity flag is set (PE).

Step 14 fills the AX register with ones. With no carries, the only flags set are Sign and Parity (NG and PE).

Step 15 is identical to step 11, only now the AX register has overflowed. This sets the Zero, Auxiliary Carry, Parity, and Carry flags (ZR, AC, PE, and CY), and clears the Sign flag (PL).

Step 16 compares the AX register with zero, and gets a match; the Zero flag is set (ZR). Because the result is zero, the Parity flag is also set (PE).

Step 17 repeats the compare, only this time for 0FFFFH. Subtract 0FFFFH from 0000H and you get 0001H, with a borrow. The borrow sets the Carry flag (CY), and the one in the result clears the Parity and Zero flags (PO and NZ). Because there was an auxiliary borrow in the operation, the Auxiliary Carry flag is set (AC). Finally, the contents of the register remain zero, because the result is not stored in the register for a compare operation.

Step 18 is identical to step 12, only now the AX register is involved. The register is filled with ones, setting the Sign and Parity flags (NG and PE). The borrow, caused by the subtraction, sets the Auxiliary Carry and Carry flags (AC and CY). The Overflow flag is not set (NV) because there was a borrow into and out of the MSB.

Step 19 subtracts 7FFFH from the AX register. This clears the lower 15 bits of the register, and sets the MSB. Since there were no carries, only the Sign and Parity flags are set (NG and PE). Again, the Parity flag is set because the low-byte of the AX register has even parity.

Step 20 subtracts one more from the AX register, and as a result, reverses all of the bits in the register. A borrow out of the MSB sets the Overflow flag (OV). An auxiliary carry during the operation sets the Auxiliary Carry flag (AC). Because there was no carry into the MSB, the Carry flag is clear (NC). Even parity in the low byte sets the Parity flag (PE).

Step 21 adds one to the AX register and again reverses all of the bits in the register. A carry into the MSB sets the Overflow and Sign flags (OV and NG). An auxiliary carry sets the Auxiliary Carry flag (AC). Finally, even parity in the low byte sets the Parity flag (PE).

Step 22 fills the AX register with ones in preparation for the next step. The Sign and Parity flags are set (NG and PE). The Overflow and Auxiliary Carry flags are cleared (NV and NA).

Step 23 increments the contents of the AX register. This is equivalent to adding one to the register, as we did in step 15. However, in this case, the Carry flag is not affected; it remains in the state determined by the previous step. The INC instruction affects all of the flags except the Carry flag. Thus, in this step, only the Zero, Auxiliary Carry, and Parity flags are set (ZR, AC, and PE). The Overflow and Sign flags are clear (NV and PL).

Step 24 decrements the contents of the AX register. This is equivalent to subtracting one from the register. Like the increment instruction, the decrement instruction will have no effect on the status of the Carry flag. Of the flags it does affect, the Sign, Auxiliary Carry, and Parity flags are set (NG, AC, and PE).

Step 25 is the first of the three Carry flag control instructions. These instructions have no operand, since they operate directly on the Flag register. The set carry flag (STC) instruction sets the Carry flag (CY). It has no effect on any other flag or register in the MPU.

Step 26 clears the Carry flag (NC), with the clear carry flag (CLC) instruction.

Step 27 sets the Carry flag (CY), with the complement carry flag (CMC) instruction. Whatever the status of the Carry flag before this instruction is executed, the status will be reversed.

Discussion

After stepping through this program, you should be very familiar with the way each flag will respond to an arithmetic or flag control instruction. If you are still a little fuzzy on a particular operation, go back and write a short program that tests that operation. When you feel confident, proceed with the experiment.

Procedure Continued

6. Now that you understand the arithmetic condition code flag responses, let's apply what you learned. The next program you will load is our multiply-through-repeated-addition program using a conditional jump instruction. Call up the editor and enter the program listed in Figure 3-23. Type "EDLIN PROG2.ASM" and RETURN.

```
TITLE EXPERIMENT 3 -- PROGRAM 2 -- CONDITIONAL LOOP MULTIPLICATION
;
COM_PROG  SEGMENT               ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
MULTIPLICAND    EQU     7
MULTIPLIER      EQU     5
;
        ORG     100H            ;COM programs always start here
START:  SUB     AL,AL           ;Clear the PRODUCT register
        MOV     CL,MULTIPLIER   ;Set multiplier as operation counter
        MOV     BL,MULTIPLICAND ;Get the multiplicand, then perform
TIMES:  ADD     AL,BL           ;the multiplication operation by
                                ;adding the multiplicand to itself
                                ;"count" times
        DEC     CL              ;Decrement the "count" after the add
        JNZ     TIMES           ;and repeat if "count" not zero
        MOV     PRODUCT,AL      ;Store the product if "count" zero
;
PRODUCT DB      0               ;Reserve one byte in memory
                                ;and initialize the byte to 0
COM_PROG  ENDS                  ;End of program segment
        END     START           ;End of program, point to beginning
```

Figure 3-23

Multiplication using repeated addition and a conditional jump to form a program loop.

7. After you exit the editor, assemble, link, and convert PROG2.ASM into PROG2.COM. Use the following commands to initiate each program:

```
MASM PROG2,PROG2,PROG2,PROG2;
LINK PROG2;
EXE2BIN PROG2.EXE PROG2.COM
```

8. Figure 3-24 is the assembled program listing. Most of the data presented in the listing was described earlier. One area not covered earlier is the conditional jump instruction. Because the jump is relative, a target address is not specified in the instruction. Rather, the relative offset to the target is calculated by the assembler. Line 17 contains the jump instruction. The first byte, 75H, is the JNZ opcode. The second byte, 0FAH, is the 2's complement offset to the target of the jump. A 2's complement value is given because the jump is backward.

   Assuming the jump is executed, 0FAH is added to the current value of the IP register, 010CH, to produce the new IP register value 0106H, the address of the target. (Adding 0FAH is equivalent to subtracting 6H.) Thus, the next instruction to be executed is located at address 0106H. To see how the process works, let's use the debugger to single-step through the program.

9. Load the program into memory with the debugger. Type "DEBUG PROG2.COM" and RETURN. Then, display the MPU register contents. Type "R" and RETURN. Notice that the Flag register bits are all clear, or reset to a zero logic level.

10. Using the debugger Trace command, single-step through the first four instructions.

    The first instruction clears the AL register. Because it is an arithmetic instruction it affects the condition of the Flag register. In this case, the zero and parity flags are set.

    The next two instructions load the multiplier and the multiplicand, but they have no effect on the Flag register, since they are not arithmetic instructions.

    The fourth instruction added the multiplicand to the AL register. This operation clears the zero and parity flags.

```
The Microsoft MACRO Assembler          01-20-84     PAGE    1-1


1                                      TITLE EXPERIMENT 3 -- PROGRAM 2 -- COND
                                    ITIONAL LOOP MULTIPLICATION
2                                      ;
3       0000                           COM_PROG  SEGMENT               ;Beginn
                                    ing of program segment
4                                              ASSUME  CS:COM_PROG,DS:COM_PROG
                                    ,SS:COM_PROG
5                                      ;
6       = 0007                         MULTIPLICAND    EQU     7
7       = 0005                         MULTIPLIER      EQU     5
8                                      ;
9       0100                                   ORG     100H             ;COM pr
                                    ograms always start here
10      0100  2A C0                    START:  SUB     AL,AL            ;Clear
                                    the PRODUCT register
11      0102  B1 05                            MOV     CL,MULTIPLIER    ;Set mu
                                    ltiplier as operation counter
12      0104  B3 07                            MOV     BL,MULTIPLICAND ;Get th
                                    e multiplicand, then perform
13      0106  02 C3                    TIMES:  ADD     AL,BL            ;the mu
                                    ltiplication operation by
14                                                                      ;adding
                                     the multiplicand to itself
15                                                                      ;"count
                                    " times
16      0108  FE C9                            DEC     CL               ;Decrem
                                    ent the "count" after the add
17      010A  75 FA                            JNZ     TIMES            ;and re
                                    peat if "count" not zero
18      010C  A2 010F R                        MOV     PRODUCT,AL       ;Store
                                    the product if "count" zero
19                                     ;
20      010F  00                       PRODUCT DB      0                ;Reserv
                                    e one byte in memory
21                                                                      ;and in
                                    itialize the byte to 0
22      0110                           COM_PROG  ENDS                   ;End of
                                     program segment
23                                             END     START            ;End of
                                     program, point to beginning
```

Figure 3-24

Assembled listing of multiplication program.

11. Single-step through the decrement instruction. The "count" is reduced to four, causing the zero flag to remain clear. This would suggest that the next instruction will force the program to branch back to the jump target. Notice that the debugger has decoded the target address and placed the offset value in the instruction operand. This is simply a programming aid to show where the program will branch.

12. Single-step through the JNZ instruction. The IP register now points to the jump target at address 0106H. Thus, the next instruction to be executed is the ADD instruction.

13. Continue to single-step through the program until the CL register is decremented to zero. At this point, the AL register contains the final product of the multiplication process, 23H (35). The zero flag is clear, indicating the last decrement zeroed the CL register.

14. Single-step through the jump instruction. With the zero flag clear, the jump if zero instruction is ignored, and the program falls through to the next instruction — the IP register contains the offset 010CH. Notice that the next instruction will save the contents of the AL register at offset 010FH. This is the address of the defined byte PRODUCT.

15. To verify the save operation, examine offset address 010FH. Type "D" and RETURN. Offset address 010FH contains the value _____.

16. Now execute the last instruction. Type "T" and RETURN. Examine memory again. Type "D100" and RETURN. Offset address 010FH contains the value _____. Is it the correct product?

Discussion

Our multiplication program is a good example of using a conditional jump instruction. It also provides a convenient method for multiplying one number times another. But, the program is limited. It can only accommodate values that fit within an 8-bit register and answers that fit within an 8-bit register. Capacity is easily increased by using 16-bit registers. Now you can multiply numbers as large as 65,535. However, the product still can't exceed 65,535. The most significant bits of numbers exceeding 16 bits will be carried out of the register and lost. What we need is a routine that will recognize and save any carries during the multiplication process.

The best way to approach the problem is to analyze the program with a flowchart, and then modify the chart to accommodate the additional program steps. Figure 3-25 shows the original program on the left; the boxed-in steps on the right are the solution to the problem.

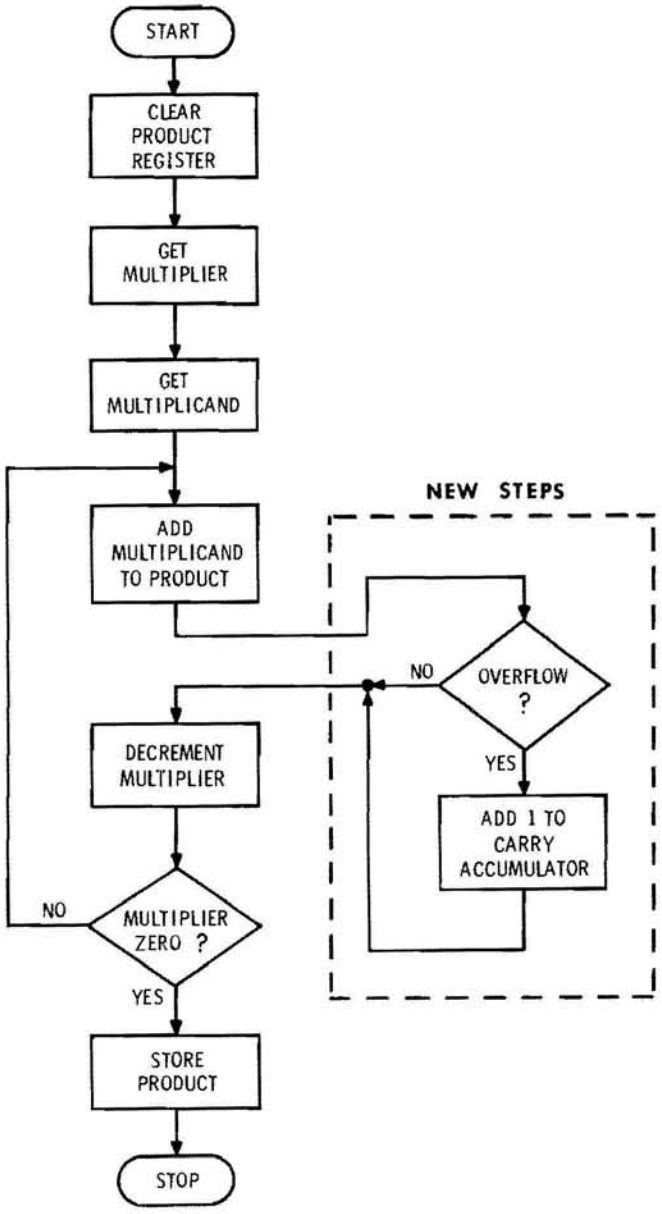


Figure 3-25

Flowchart of program to multiply two numbers up to one word in size.

The original program used a process of repeated addition to determine the product of two numbers. The multiplicand was added to itself the number of times specified by the multiplier. Now if you want to check for a product that is too large for its register, the best time is just after the multiplicand is added to the register, as we've shown in Figure 3-25. Since any carry out of the register is never larger than one bit, a conditional jump that tests the Carry flag is an ideal method. If the Carry flag is set, then the program can add one to the "carry accumulator," a register reserved by you to hold the carry count. If the Carry flag isn't set, the program can continue with the multiplication process without adding one to the carry accumulator. The solution should be easy to implement.

Procedure Continued

17. If you think you understand the problem and the solution, rewrite PROG2.ASM to accommodate large, 16-bit numbers. Two 16-bit registers are large enough to hold the product. Try not to look at Figure 3-26 as you rewrite the program. Test the program by multiplying 65535 (0FFFFH) times 3. Use the debugger to single-step through the instructions after you convert your ASM file into a COM file. Keep track of the Flag register, the multiplier register, the multiplicand register, and the two product registers as you step though the program.

18. At the end of the program, record the following data:

CARRY REGISTER = _____ H
PRODUCT REGISTER = _____ H

```
TITLE EXPERIMENT 3 -- PROGRAM 3 -- LARGE NUMBER MULTIPLICATION
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
MULTIPLICAND    EQU     0FFFFH
MULTIPLIER      EQU     3
;
        ORG     100H             ;COM programs always start here
START:  SUB     AX,AX            ;Clear the PRODUCT register
        SUB     DX,DX            ;Clear the CARRY register
        MOV     CX,MULTIPLIER    ;Set multiplier as operation counter
        MOV     BX,MULTIPLICAND ;Get the multiplicand, then perform
TIMES:  ADD     AX,BX            ;the multiplication operation by
                                 ;adding the multiplicand to itself
                                 ;"count" times
        JNC     CONTINUE         ;Is product more than 65535?
        INC     DX               ;Yes, add 1 to CARRY register
CONTINUE:
        DEC     CX               ;Decrement the "count" after the add
        JNZ     TIMES            ;and repeat if "count" not zero
        MOV     PRODUCT,AX       ;Store the product if "count" zero
        MOV     PRODUCT+2,DX     ;Store the carry
        INT     3                ;Halt the program and
                                 ;return to the debugger
;
PRODUCT DW      0,0              ;Reserve two words in memory
                                 ;and initialize the words to 0
COM_PROG  ENDS                   ;End of program segment
        END     START            ;End of program, point to beginning
```

Figure 3-26

Program to multiply two word-sized numbers.

Discussion

Figure 3-26 should be pretty close to the program you wrote. We changed all of the 8-bit registers to 16-bit registers, and the defined memory to a word-sized location. Then, we added a second memory location to store the "carry" register contents. Three new instructions allowed us to check the Carry flag, increment the carry register if there was a carry, and store the contents of the carry register. If there was no carry, as in the first add operation, the JNC instruction routed the program around the increment instruction.

There is one more instruction at the end of the program that is new to you. It is called a **program interrupt**. We won't be covering program interrupts until later in the course, but this one is very useful for program debugging, so we will discuss it briefly here.

An interrupt is a command to the MPU to stop executing the current program and begin executing a special routine. In the case of INT 3, the MPU is instructed to stop executing the current program, while in the debugger, and return to the debugger command prompt. This can be very handy if you wish to test a few lines of code, and it isn't practical to single-step through the program.

For example, a good test for the validity of your multiplication program is to reverse the multiplier and the multiplicand. Naturally, you wouldn't try to single-step through the program 65,535 times. You can, however, place the INT 3 instruction in the program where you wish to stop execution and return to the debugger. In Figure 3-26, we placed the interrupt after the product and carry are stored in memory. Why don't you give it a try.

Procedure Continued

19. Call up the editor and reverse the multiplier and multiplicand in your program. Then add the INT 3 instruction to the program. Convert the ASM file to a COM file and load the program into memory with the debugger. Now run the program. Type "G=100" and RETURN. After a slight pause, the program has completed execution, and the register contents are displayed. To give you an idea how fast the MPU executes program code, keep in mind that between the time you hit the RETURN key and the registers were displayed, the program completed 65,535 multiplication loops.

    The command "G=100" translates to "Go," or run the program starting at address offset 100H, the beginning of the program.

    Are the product and carry values produced by this program the same as before? There should be no difference between adding 3 to itself 65,535 times and adding 65,535 to itself 3 times.

20. Your program can multiply one number by another, using numbers up to 16 bits in length, with one exception. What do think that exception is? The answer is a zero in the multiplier. A zero in the multiplicand will return an answer of zero, but a zero in the multiplier will return an answer 65,536 times the value in the multiplicand. The reason lies in how the multiplier is handled by the program. After the multiplicand is added to the AX

(product) register, the multiplier register (CX) is decremented and tested for zero. If the multiplier was originally zero, decrementing the register will change the contents to 0FFFFH. That means the program must go through 65,535 more loops before the register is again zero.

Since you want the program to handle every number including zero, you must modify the program one more time. So let's see what you've learned. Refer back to the flowchart in Figure 3-25 and determine where the program can be modified. Then, modify your program, and call it PROG4.ASM. Save the ASM file, but don't convert it to a COM file yet. We want to show you how a BAT (batch) file can be used to generate the COM file.

21. Call up the editor and create the file ASM.BAT. Following are the three lines of code that make up the file.

```
MASM %1,%1,%1,%1;
LINK %1;
EXE2BIN %1.EXE %1.COM
```

Now to convert your ASM file to a COM file, type "ASM PROG4" and RETURN. The BAT file will process all of the necessary commands to generate the four MASM files, the EXE file, and the COM file. As these files are being processed, the normal program statements are displayed. At the end, two monitor prompts will be displayed, rather than the standard single prompt, with the flashing cursor after the second prompt.

Save the BAT file and use it whenever you need to convert an ASM file to a COM file. Just remember when you execute a BAT file, never give the file extensions (BAT or ASM), just type the BAT file name, and then, type the name of the ASM file to be processed.

22. Figure 3-27 shows our modified program. Two instructions have been added. The first compares the contents of the CX register with zero. The second is a conditional jump. If CX equals zero, the jump is taken, and the program branches to the target labeled STOP. A value other than zero causes the jump to be ignored.

```
TITLE EXPERIMENT 3 -- PROGRAM 4 -- LARGE NUMBER MULTIPLICATION
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
MULTIPLICAND    EQU     0FFFFH
MULTIPLIER      EQU     3
;
        ORG     100H             ;COM programs always start here
START:  SUB     AX,AX            ;Clear the PRODUCT register
        SUB     DX,DX            ;Clear the CARRY register
        MOV     CX,MULTIPLIER    ;Set multiplier as operation counter
        CMP     CX,0             ;Is multiplier zero?
        JZ      STOP             ;Yes, don't try to multiply
        MOV     BX,MULTIPLICAND ;Get the multiplicand, then perform
TIMES:  ADD     AX,BX            ;the multiplication operation by
                                 ;adding the multiplicand to itself
                                 ;"count" times
        JNC     CONTINUE         ;Is product more than 65535?
        INC     DX               ;Yes, add 1 to CARRY register
CONTINUE:
        DEC     CX               ;Decrement the "count" after the add
        JNZ     TIMES            ;and repeat if "count" not zero
STOP:   MOV     PRODUCT,AX       ;Store the product if "count" zero
        MOV     PRODUCT+2,DX     ;Store the carry
        INT     3                ;Halt the program and
                                 ;return to the debugger
;
PRODUCT DW      0,0              ;Reserve two words in memory
                                 ;and initialize the words to 0
COM_PROG  ENDS                   ;End of program segment
        END     START            ;End of program, point to beginning
```

Figure 3-27

Program to multiply two numbers and check for a zero multiplier.

Discussion

Conditional jump instructions give you an opportunity to test for a specific condition, and depending on the condition, either jump to a different program area or proceed with the next instruction. Unconditional jump instructions are used to force a change in program execution regardless of the condition of the Flag register. For example, at the end of a program, rather than stop execution with an INT 3 instruction, you could use an unconditional jump instruction to shift program execution to a ROM Monitor routine. Since that is a little too complex for now, let's try a simple counting routine.

Procedure Continued

23. Call up the editor and enter the program in Figure 3-28. Use the ASM.BAT program to convert the ASM file to a COM file. Then, load the program into memory with the debugger.

```
TITLE EXPERIMENT 3 -- PROGRAM 5 -- UNCONDITONAL COUNTING LOOP
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
COUNT_VALUE     EQU     0FFH     ;Count defined
;
        ORG     100H             ;COM programs always start here
START:  SUB     AL,AL            ;Clear the COUNT register
COUNT:  INC     AL               ;Count one
        MOV     SUM,AL           ;Store count
        CMP     AL,COUNT_VALUE   ;Count long enough?
        JZ      STOP             ;Yes, halt the program
        JMP     COUNT            ;No, count one more time
STOP:   INT     3                ;Halt the program and
                                 ;return to the debugger
;
SUM     DB      0                ;Memory reserved for count
COM_PROG  ENDS                   ;End of program segment
        END     START            ;End of program, point to beginning
```

Figure 3-28

Counting program to demonstrate the unconditional jump.

The program is designed to count up from zero, using the unconditional jump instruction to complete the loop. To add a little more substance to the program, it also stores each new count in memory. Finally, because a program loop of this nature will never end, unless you reset the MPU, the program has a conditional jump to escape the loop after the count reaches a predetermined value.

24. Single-step through the program loop a few times. First type "R" and RETURN, then use the "T" command for each single-step operation. Notice that each time through the loop, one is added to the AL register, the register contents are stored in memory, the register contents are compared to an immediate value, the Zero flag is tested, and the loop is completed with an unconditional jump. If you are patient, you can single-step through the loop 255 times, or you can save time, and use the "Go" command to run the program. In this case, you don't type "G=100", you just type "G" and RETURN. The program will run from the point where you stopped single-stepping. When the program finally stops, the AL register and the memory storage area will contain the value 0FFH.

Discussion

The assembler treated the unconditional jump instruction in your program as a short jump. That is, it calculated the relative distance to the target as an 8-bit value rather than a 16-bit value. It did this because it knew the location of the target, and the target was within the 256-byte jump range. The assembler knew the location of the target because the target label preceded the jump instruction. If the target follows the jump instruction, the assembler will assume a normal jump (16-bit target address).

The reason for this curious situation relates to the way the assembler translates the source code into object code. MACRO-86 is a two-pass assembler. This means that the source file is read twice by the assembler. During the first pass, the assembler evaluates the statements and determines the amount of code it will generate. From that information, it builds a symbol table where all symbols, variables, and labels are assigned values. During the second pass, the assembler fills in the symbols, variables, and labels from the symbol table.

Therefore, on the first pass, if the target precedes the jump, the assembler can determine if it must reserve code space for an 8-bit address or a 16-bit address. However, if the target follows the jump, a "forward jump," the assembler doesn't know how much space it must reserve for the target address. In this case, it automatically reserves two bytes of code for the target address. On the second pass, the assembler fills in the target address value. If the forward jump is "short," the address value is one byte long. Since the assembler reserved two bytes for the address value, it is left with an empty byte. Empty bytes aren't legal, so the assembler fills the byte with the code, 90H, for a NOP (no operation) instruction. Recall that a NOP instruction does nothing. It simply takes up space in the program and uses three clock cycles of MPU execution time.

Now that you have had a chance to see how jump instructions can be used to redirect program flow, let's look at a more refined form of program branch, the loop instruction.

Procedure Continued

25. The program in Figure 3-27 used a couple of jump instructions to perform a multiplication operation. Figure 3-29 is the same basic program, only here, loop related instructions are used. Call up the editor and enter the program in Figure 3-29. Use the ASM.BAT program to convert the ASM file to a COM file. Then, load the program into memory with the debugger.

```
TITLE EXPERIMENT 3 -- PROGRAM 6 -- LARGE NUMBER MULTIPLICATION LOOP
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
MULTIPLICAND    EQU     0FFFFH
MULTIPLIER      EQU     3
;
        ORG     100H             ;COM programs always start here
START:  SUB     AX,AX            ;Clear the PRODUCT register
        SUB     DX,DX            ;Clear the CARRY register
        MOV     CX,MULTIPLIER    ;Set multiplier as operation counter
        JCXZ    STOP             ;Is multiplier zero?
                                 ;If so, don't try to multiply
        MOV     BX,MULTIPLICAND  ;Get the multiplicand, then perform
TIMES:  ADD     AX,BX            ;the multiplication operation by
                                 ;adding the multiplicand to itself
                                 ;"count" times
        JNC     CONTINUE         ;Is product more than 65535?
        INC     DX               ;Yes, add 1 to CARRY register
CONTINUE:
        LOOP    TIMES            ;Decrement the "count" after the add
                                 ;and repeat if "count" not zero
STOP:   MOV     PRODUCT,AX       ;Store the product if "count" zero
        MOV     PRODUCT+2,DX     ;Store the carry
        INT     3                ;Halt the program and
                                 ;return to the debugger
;
PRODUCT DW      0,0              ;Reserve two words in memory
                                 ;and initialize the words to 0
COM_PROG  ENDS                   ;End of program segment
        END     START            ;End of program, point to beginning
```

Figure 3-29

Multiplication program modified with a LOOP instruction.

26. Single-step through the program and observe the registers. The end result should be no different from the earlier program; it's just accomplished in a slightly different fashion.

Discussion

The program is changed in two areas. First, the check for a zero multiplier is now handled by the jump if CX zero instruction, instead of the compare and jump if zero instructions. The change is insignificant in this program. It simply reduces the number of instructions by one. Where it could be important is in a program loop. Here, a savings of one instruction could make a big difference in program execution time. The second area of program change is a good example.

The decrement and jump if not zero instructions are replaced by the LOOP instruction. Since the LOOP instruction is part of the program loop, the savings of one instruction reduces the execution time of the program; The bigger the multiplier, the greater the savings.

That was an example of an unconditional loop instruction. Now let's look at an example of a conditional loop instruction.

Procedure Continued

27. Call up the editor and enter the program in Figure 3-30. Use the ASM.BAT program to convert the ASM file to a COM file. Then load the program into memory with the debugger.

```
TITLE EXPERIMENT 3 -- PROGRAM 7 -- COUNTING ROUTINE
;
COM_PROG  SEGMENT                ;Beginning of program segment
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
MAX_COUNT       EQU     10       ;Maximum number of counts
BREAK           EQU     3        ;Stop count number
;
        ORG     100H             ;COM programs always start here
START:  SUB     AX,AX            ;Zero the register
        MOV     CX,MAX_COUNT     ;Get the maximum count value
        MOV     BX,BREAK         ;Get the stop count value
COUNT:  INC     AX               ;Add one to count register
        DEC     BX               ;Decrement the break value
        LOOPNZ  COUNT            ;Loop if BX or CX not zero
        INT     3                ;Halt the program and
                                 ;return to the debugger
;
COM_PROG  ENDS                   ;End of program segment
        END     START            ;End of program, point to beginning
```

Figure 3-30

Program to show the operation of the conditional loop instruction.

28. Single-step through the program. The loop count register (CX) is loaded with the value 10, yet the program stopped counting after 3. Why?

Discussion

The conditional loop instruction makes two tests prior to each loop. If either test fails, the loop is ignored. In the case of the LOOPNZ instruction in this program, the zero flag and the CX register contents are tested. If the zero flag is clear, the last arithmetic operation produced a non-zero, and if the CX register is not zero, the loop is taken. Since the BX register was zeroed during the third loop, the program never executed a fourth loop, leaving the count in the AX register at 3. Had the MAX_COUNT and BREAK values been reversed, the result would have been the same.

This completes the Experiment for Unit 3. Proceed to the Unit 3 Examination.

UNIT 3 EXAMINATION

1. What are the six flags discussed in this Unit?

   A. _____.
   B. _____.
   C. _____.
   D. _____.
   E. _____.
   F. _____.

2. What are the contents of the six condition flags after the following instructions are executed?

```
MOV AL,0BH
SUB AL,0FH
```

   A. _____.
   B. _____.
   C. _____.
   D. _____.
   E. _____.
   F. _____.

3. List the flags that affect the following conditional jumps, and state the condition of each flag or group of flags.

   JZ _____.
   JNC _____.
   JPE _____.
   JO _____.
   JNS _____.
   JNE _____.
   JNL _____.
   JNBE _____.
   JE _____.
   JA _____.

4. The last mathematical operation set the flags in this manner: OF = 1, SF = 1, ZF = 0, AF = 1, PF = 1, CF = 1. Based on this, which of the following conditional jumps would be taken? Answer YES or NO for each jump.

JNG _____.
JNO _____.
JLE _____.
JNGE _____.
JBE _____.
JL _____.
JNLE _____.
JBE _____.
JS _____.

5. Identify the flowchart symbols in Figure 3-31.

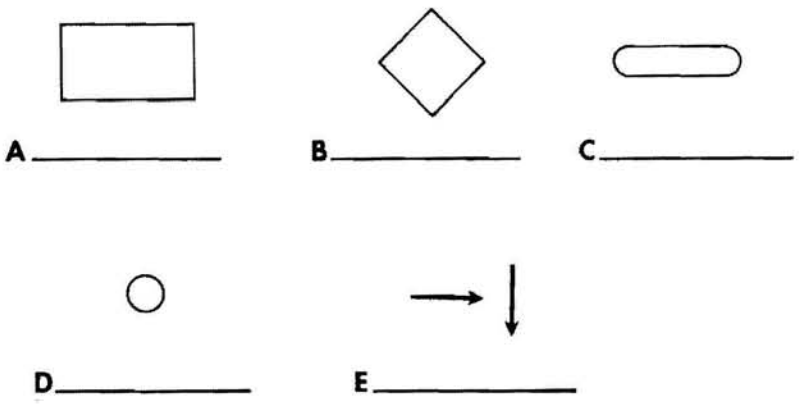


Figure 3-31
Figure for question 5.

6. List the five decision-making mathematical symbols most commonly used in flowcharts.

   A. _____.
   B. _____.
   C. _____.
   D. _____.
   E. _____.

7. The loop count for a LOOP instruction is contained in the _____ register.

8. The status of the CX register is the only test performed by the LOOPE instruction prior to executing the loop. _____
   True/False

EXAMINATION ANSWERS

1. The six flags discussed in this Unit are:

   A. Overflow flag or OF.
   B. Sign flag or SF.
   C. Zero flag or ZF.
   D. Auxiliary carry flag or AF.
   E. Parity flag or PF.
   F. Carry flag or CF.

2. The contents of the six condition flags are

   A. OF = 0
   B. SF = 1
   C. ZF = 0
   D. AF = 1
   E. PF = 1
   F. CF = 1

   after the instructions

```
MOV  AL,0BH
SUB  AL,0FH
```

   are executed.

3. Following are the flags and their condition that control the execution of the conditional jump instructions:

```
JZ    ZF = 1.
JNC   CF = 0.
JPE   PF = 1.
JO    OF = 1.
JNS   SF = 0.
JNE   ZF = 0.
JNL   (SF XOR OF) = 0.
JNBE  (CF OR ZF) = 0.
JE    ZF = 1.
JA    (CF OR ZF) = 0.
```

4. The last mathematical operation set the flags in this manner: OF = 1, SF = 1, ZF = 0, AF = 1, PF = 1, CF = 1. Based on this, the jumps that would be taken are indicated below.

   JNG NO.
   JNO NO.
   JLE NO.
   JNGE NO.
   JBE YES.
   JL NO.
   JNLE YES.
   JBE YES.
   JS YES.

5. The flowchart symbols shown in Figure 3-31 are the:

   A. Operations box.
   B. Decision box.
   C. Terminal box.
   D. Connector.
   E. Flow lines.

6. The five decision-making mathematical symbols most commonly used in flowcharts are:

   A. =, is equal to.
   B. >, is greater than.
   C. <, is less than.
   D. $\geq$, is greater than or equal to.
   E. $\leq$, is less than or equal to.

7. The loop count for a LOOP instruction is contained in the **Count**, or **CX** register.

8. **False**. The status of the CX register is only one of two tests performed by the LOOPE instruction prior to executing the loop. The other test is whether the zero flag is set.

SELF-REVIEW ANSWERS

1. The five most common flowcharting symbols are the:

   A. Terminal Symbol
   B. Operation Box
   C. Decision Box
   D. Flow Lines
   E. Connectors

2. **True**. You are permitted to use mathematical symbols in a flowchart.

3. **True**. The flowchart should always begin with a start terminal symbol.

4. The **unconditional branch** in a flowchart corresponds to the unconditional jump in the actual program.

5. The **conditional branch** in a flowchart corresponds to the conditional jump in the actual program.

6. Virtually every program uses a technique called a **loop**.

7. The loop allows a section of the program to be repeated as often as needed to perform an operation.

8. The **jump** instruction allows the microprocessor to escape the straight line program sequence.

9. Each time you use a jump instruction, you must specify the instruction's **target**.

10. The **target** is the place, or memory location, in the program to which you wish the program to jump.

11. When the address of the target is calculated using a relative value, the instruction is said to be a **direct** jump.

12. In an **indirect** jump, the address of the target is contained in a register. This address replaces the contents of the IP register when the instruction is executed.

13. The ability to make a decision is the real power of the microprocessor.

14. The 8088 MPU bases its decisions on the contents of the **Flag** register.

15. **False**. The CMP instruction does not subtract the source operand from the destination operand and store the result in the AX register. Neither operand is affected by the compare operation. Only the Flag register is affected.

16. The **unconditional** jump instruction removes the MPU from the decision making process.

17. The JNZ instruction will cause the program to jump to the target if the zero flag is **clear**.

18. The **JA** instruction should not be used with signed arithmetic operations. It doesn't test the sign flag.

19. An **arithmetic** operation will affect the condition of the Flag register.

20. The **zero** flag is tested when a LOOPE instruction is executed.

21. **False**. The LOOP instruction is considered an **unconditional** instruction.

22. The the first step performed when the LOOP instruction is executed is decrement the CX register and test for zero.

23. The **JCXZ** instruction is used to test the CX register for zero.

24. **True**. The LOOPE instruction is considered a conditional instruction.

25. **False**. The Zero flag must be set before the LOOPE instruction will execute a program loop.

26. The **LOOPNZ** instruction has the same opcode as the LOOPNE instruction.

27. The maximum distance a program can loop backward with the LOOP instruction is **−128** bytes.

INSERT

Unit 4

SUBROUTINES

CONTENTS

INTRODUCTION

By now, you should be a little more comfortable with assembly language. You can probably see great programming possibilities with the various program transfer instructions. But there is one major problem with branching; you have no easy way to jump from different areas of the program to a specific area of the program and then back again.

This Unit will resolve that problem. Here, you will learn how to write and address program sections called subroutines. Then you will learn how to use two new instructions, call and return, that allow you to access and return from these subroutines. Next, you will learn how the MPU uses a "memory stack" to save vital data, such as the subroutine return address. Finally, you will learn how to create file libraries of subroutines, and "include" them in a program.

Use the "Unit Objectives" that follow to evaluate your progress. When you successfully accomplish all of the objectives, you will have completed this Unit. You can use the "Unit Activity Guide" to keep a record of the sections that you complete.

UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. Define stack, subroutine, and Stack Pointer.
2. Explain the operations performed by each of the following instructions: PUSH, PUSHF, POP, POPF, CALL, and RET.
3. Write simple programs that use subroutines.
4. Explain the use of the Stack Pointer register.
5. Write simple programs that use the memory stack.
6. Define the LABEL and INCLUDE directives.
7. Define the operator OFFSET.
8. Write a library file subroutine, and include that file in another program.

UNIT ACTIVITY GUIDE

		Completion Time
☐	Read the Section on "Subroutine Calls."	_____
☐	Complete Self-Review Questions 1-6.	_____
☐	Read the Section on "The Stack."	_____
☐	Complete Self-Review Questions 7-20.	_____
☐	Read the Section on "The Call and Return Instructions."	_____
☐	Complete Self-Review Questions 21-26.	_____
☐	Read the Section on "Including Files."	_____
☐	Complete Self-Review Questions 27-30.	_____
☐	Perform the Experiment.	_____
☐	Complete the Unit 4 Examination.	_____
☐	Check the Examination Answers.	_____

SUBROUTINE CALLS

A Subroutine is a step-by-step procedure for doing a particular job. As the name implies, it is a subsection or part of a larger routine or program. By convention, a subroutine is a series of instructions used to perform a task that may occur many times throughout a program. You might think of it as a specialized loop in an otherwise linear program. Its real power, however, lies in the fact that it can be used over and over again from any part of the program. To get an idea how a subroutine could be used, let's look at a program that operates in an endless loop.

The Problem

Suppose you had a microprocessor-controlled sound generator and you wanted to create the sound of a clock "tick-tock." The program flowchart for such a program would probably look like the one in Figure 4-1.

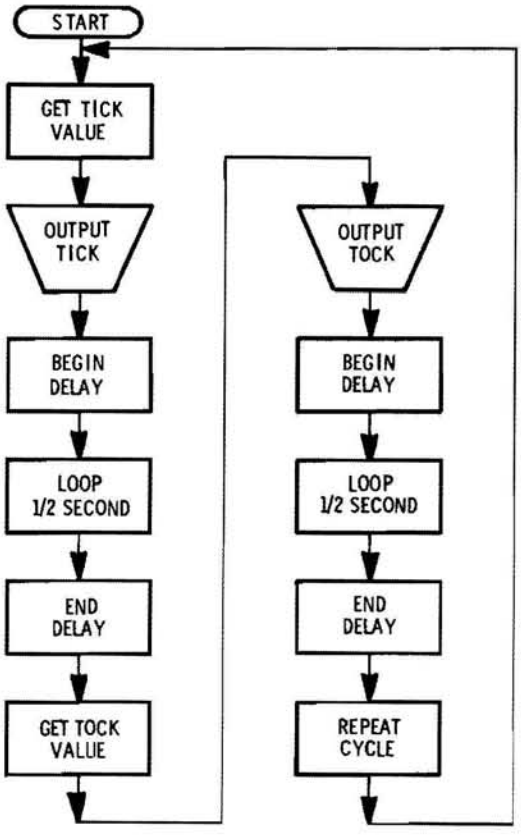


Figure 4-1
Flowchart for a "tick-tock" program.

The START symbol, of course, indicates the beginning of the program. The program commences by loading the "tick value." This is some number that, when translated by the generator, would produce the sound of a "tick." Next, the program outputs the tick value to the generator. (The trapezoid-shaped box is a flowchart symbol for input/output.) Finally, the tick portion of the program ends with a delay routine that loads a value into the Count (CX) register that will produce one-half second worth of looping. You will find, when we examine the program later, that each instruction requires a specific amount of time to execute. By arranging a unique string of instructions in the loop, you can make the program delay any amount of time.

After the delay, the "tock" portion of the program is executed. First the "tock value" is loaded. Then this value is sent to the generator. Finally, the program delays for one-half second. After the second delay, the program cycle repeats. Since the program is meant to continuously cycle, there is no STOP symbol.

Computer memory costs money, and the time it takes to enter a program costs even more money. Therefore, every time you can save program space and your time to write the program, it's to your advantage. The "tick-tock" program is a prime candidate.

The Subroutine Solution

Notice that the program has a one-half second delay that is repeated twice each cycle. If you were to make it a separate subroutine, and then jump to it every time you needed a delay, you could shorten the program length. Figure 4-2 shows how this is done. The small circles in the flowchart indicate a change in flow direction, usually to a subroutine. Simply match the number or letter in the circle in the main program to the number or letter of the appropriate subroutine. In this case there is only one subroutine. The number or letter at the end of the subroutine directs you back to the appropriate point in the main program. We've added dotted lines to help you follow the flow of the program. These are normally left out. Other than the jumps, or as they are normally labeled, "**subroutine calls**," the program executes as described earlier.

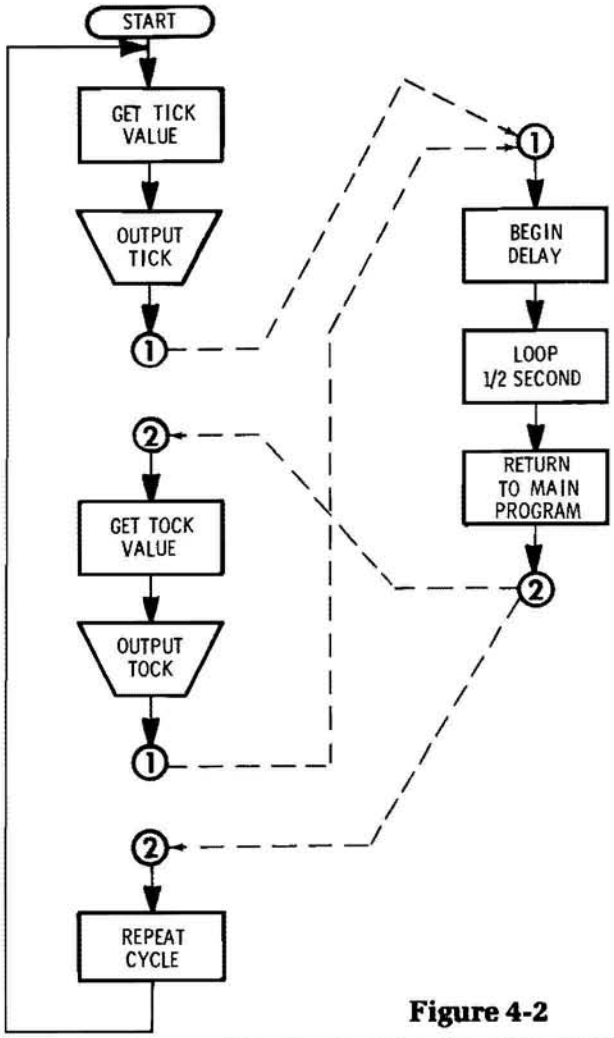


Figure 4-2
Modified flowchart for a "tick-tock" program.

Examine the flowchart in Figure 4-2. Can you see where there might be a problem if you use a JMP instruction? After the program jumps to the subroutine, how does it know at which point to re-enter the main program? Before the jump-to-subroutine occurs, the address of the next instruction in the main program must somehow be saved. And then after the subroutine is completed, the address must be recalled and placed in the Instruction Pointer (IP) register of the MPU.

The instruction that lets you save the IP before jumping is **CALL** (call subroutine). After the subroutine is completed, the final instruction is **RET** (return from subroutine). This instruction "gets" the address that the CALL instruction saved earlier and returns it to the IP. Now the MPU can execute the next instruction in the main program.

Before we can examine the mechanics of the CALL and RET instructions, we must introduce another area of the microcomputer, the **stack**. This is where the IP value is stored before a subroutine call.

Self-Review Questions

1. A subroutine is a step-by-step _____ for doing a particular job.

2. A subroutine can only be used once._____
   True/False

3. A subroutine is called with the _____ instruction.

4. The last instruction in a subroutine is the _____ instruction.

5. The Trapezoid-shaped box in a flowchart represents an _____ / _____ process.

6. The small circle in a flowchart represents a _____ in program direction.

NOTE: The Self-Review Answers are located at the end of this unit, beginning on Page 4-61.

THE STACK

In computer jargon, a **stack** is a group of temporary storage locations in which data can be stored and later retrieved. In this regard, a stack is somewhat like memory. In fact, most microprocessors, including the 8088, use a section of memory as a stack.

The stack can occupy up to 64K of memory space (a complete segment) in an EXE program. In a COM program, on the other hand, the stack is more limited in size, since COM programs are limited to a single 64K segment of memory. The stack size you choose depends on your needs, and will vary from program to program. Let's see what makes up a stack.

Structure

The stack occupies an area in memory. This area is located at the end of the program, following any defined data. Figure 4-3 shows the general structure of a typical COM program containing instruction code, defined data, and a stack. Each boxed-in area represents a byte in memory.

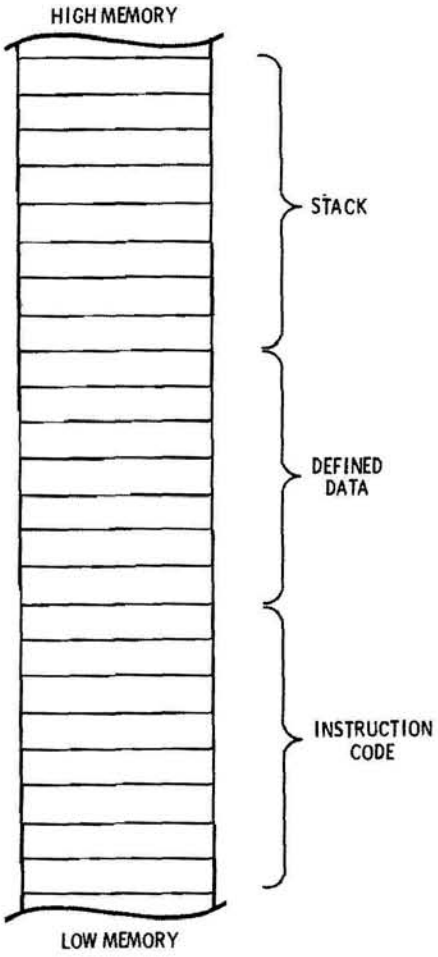


Figure 4-3
Relationship of instruction code, defined data, and the stack in a program.

The stack area in a program is not treated in the same manner as the defined data area. The stack area has a specific top and bottom, and the data is considered **not** randomly accessible. When data is moved into the stack, it is stored from the top down, with the high address location considered the "top of stack."

The MPU addresses the "top of stack" through the Stack Pointer (SP) register. That is, the SP register contains the offset address of the top of the stack. As data is stored in the stack, the SP register is decremented to point to the next free memory location in the stack, the new "top of stack." By the same token, when data is read from the stack, the SP register is incremented to point to the preceding data location. The process operates on the LIFO principle, Last data In is First data Out.

In a COM program, the stack area is not specifically identified with a SEGMENT directive, as would be the case with an EXE program. However, the Stack Segment register must still be identified in the ASSUME directive, so that the assembler knows the stack segment exists, for addressing purposes. This is no different from what you have been doing in all of your earlier programs.

The actual assignment of a stack is shown in Figure 4-4. The Figure illustrates the basic structure of a COM program that contains a stack. Notice that defining the stack area is only one part of the process of establishing the stack. Let's look at the process.

```
1   TITLE UNIT4 -- PROGRAM 1 -- PROGRAM STRUCTURE
    ;
2   COM_PROG  SEGMENT
3           ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
    ;
4           ORG     100H
5   START:  MOV     SP,OFFSET TOP_OF_STACK ;Get STACK address
            .
            .
            .
6           DW      10 DUP (?)      ;STACK size defined
7   TOP_OF_STACK    LABEL   WORD    ;Identify top of STACK
    ;
8   COM_PROG ENDS
9           END     START
```

Figure 4-4

Basic structure of a COM program with a stack.

In line 6, the stack area is defined. A "DUP (?)" expression is normally used, since you don't care what is stored in the stack prior to program execution. The Define Word assembler directive must be used, because **all stack related instructions operate on word-sized values.**

The number of bytes reserved for the stack depends on the amount of data that will be stored during program execution. Keep in mind, however, the data stored by your program may not be the only data stored in the stack. Other system programs may share the same stack. The debugger program, for example, may store up to eight words of data in the stack while it is being used to debug a program. As a general rule of thumb, if you anticipate debugging your program, always define at least eight additional words in your stack to accommodate the debugger.

Once the stack is defined, the "top of stack" must be identified. Line 7 of the program in Figure 4-4 shows the preferred method. The term "TOP_OF_STACK" is simply a descriptive name assigned to the assembler directive **LABEL**. This directive tells the assembler to associate the current address offset with the name. The operator WORD completes the identification by telling the assembler the name relates to word-sized values. Thus, line 7 has identified the next word of memory following the memory area assigned to the stack. This point is considered the top of stack. Notice that the directive LABEL does not reserve a memory location; it simply points to it.

The stack area is identified and the top of stack is identified. The last step is to load the address offset of the top of stack into the Stack Pointer register. The assembler operator **OFFSET** handles that operation. When the operator OFFSET precedes a name in the source operand of an instruction, the assembler calculates the effective address, or offset, of that name. That value becomes the source operand. Therefore, line 5 of the program in Figure 4-4 moves the offset address of the name TOP_OF_STACK into the SP register. All stack operations use the contents of the Stack Pointer along with the contents of the Stack Segment register to calculate the physical address of the top of stack.

Addressing The Stack

Data within the stack can be accessed directly or indirectly. The CALL and RET instructions use the indirect form. That is, they use the stack to support a program transfer operation. The PUSH and POP instructions directly manipulate data within the stack.

PUSH INSTRUCTIONS

The 8088 MPU has two basic push instructions. **PUSH** lets you store the contents of a 16-bit register or memory word in the stack. **PUSHF** is more specific; it lets you store the contents of the Flag register in the stack. The following is an example of how each instruction-type could be written.

```
PUSH BX          ;Save the BX register
PUSH COUNT       ;Save data at offset
                 ;COUNT and COUNT+1
PUSHF            ;Save the Flag register
```

The first example is quite straightfoward, the contents of the BX register are saved in the stack. You could not "PUSH BL" or "PUSH BH," since these are only 8-bit registers. The second example requires a little more planning if you wish to save memory data. "PUSH COUNT" indicates that you are saving the data located at the address in memory identified by the label COUNT. However, because the stack stores **words** and not **bytes**, the data at address COUNT+1 is also saved. Therefore, you must make sure the data you wish to save has been defined as word-sized data. The third example has no operand, since the opcode specifies the register to be saved.

Figure 4-5 shows the effects of the "PUSH BX" instruction. Before the instruction is executed, the Stack Pointer contains the address 01F9H. This is the top of stack address. The BX register contains the data word 0F00DH. Any data in the stack at this time is not important. Figure 4-5A shows the SP and BX register contents, and a portion of memory prior to execution.

When the PUSH BX instruction is executed, the MPU first **decrements** the Stack Pointer register by two. Then the contents of the BX register are stored, "pushed," into memory locations 01F7H and 01F8H. The SP is first decremented by two because the MPU always stores the low byte of a 2-byte word first. Since the low byte is always stored in the low memory address, the SP must be pointing at the lower of the two address locations. This is shown in Figure 4-5B.

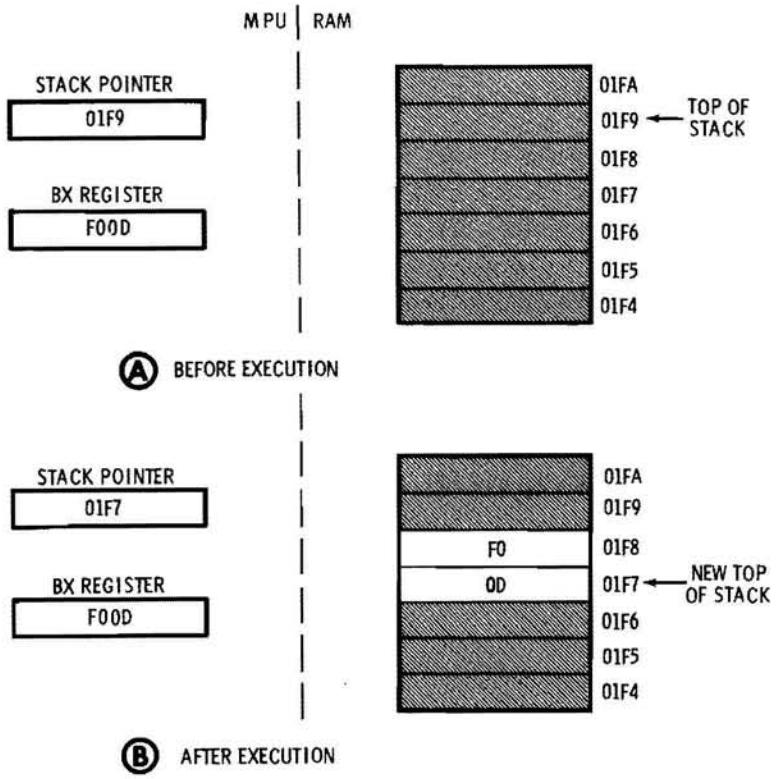


Figure 4-5
Executing the PUSH BX instruction.

The 8088 Instruction Set Summary in Appendix D uses an abbreviated flow diagram to describe the operation of each instruction. The one for the PUSH instruction is:

1. (SP)←(SP)-2

2. ((SP + 1):(SP))←(SRC)

This means that there are two steps to the PUSH operation. The first, reading from right-to-left, says subtract two from the contents of the Stack Pointer and store the result in the Stack Pointer. The second step says the contents of the specified register or memory source operand (SRC) are stored in the stack at the address indicated by the Stack Pointer value (low byte) and the Stack Pointer value plus one (high byte). Thus, if you have any question as to how an instruction is implemented, you can refer to its abbreviated operation flow diagram.

Because the Stack Pointer is always decremented-by-two before any data is stored, the first memory location at the top of stack is always considered empty or undefined. Figure 4-5B illustrates this point. After the PUSH instruction is executed, address 01F9H has no value assigned to it. The new top of stack address is 01F7H, as indicated by the Stack Pointer.

Now if you were to execute the PUSH instruction a second time, you would obtain the results shown in Figure 4-6. Part A is the MPU and RAM status before execution; part B is the MPU and RAM status after execution. Data in the BX register was changed to make the operation easier to follow. As before, the Stack Pointer is first decremented by two. Then the low byte of the BX register is moved to address 01F5H, followed by the high byte to address 01F6H.

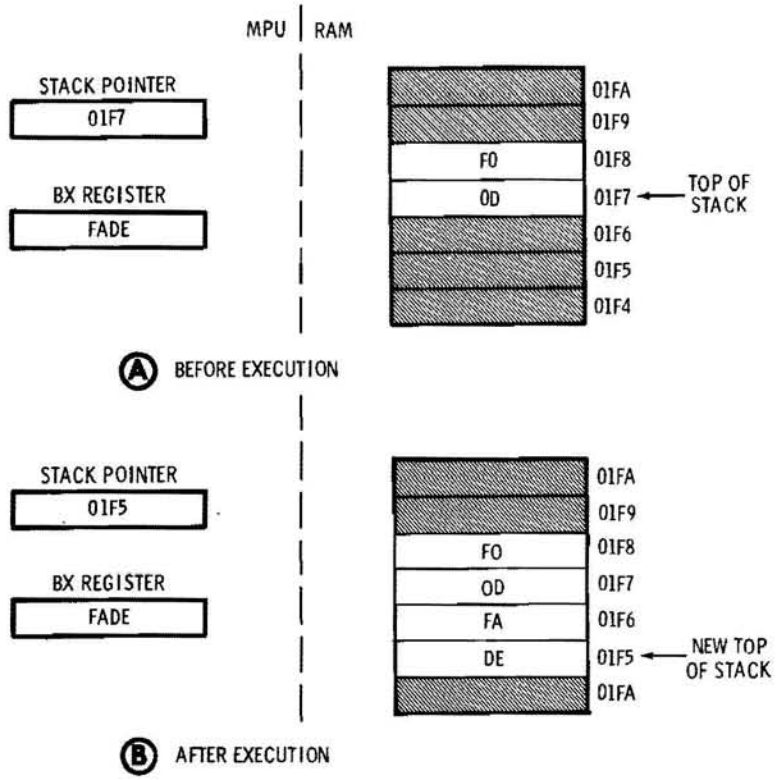


Figure 4-6

Executing a second PUSH BX instruction.

POP INSTRUCTIONS

Once you've moved data into the stack, you can remove it with a POP instruction. As with the PUSH instructions, the 8088 MPU has two basic POP instructions. **POP** lets you remove a data **word** from the stack and place it in a register or memory location. **POPF** is the opposite of PUSHF, in that it moves data from the stack to the Flags register. This, by the way, is a handy instruction if you want to preset all of the flags to some unique value. Simply load the value in a register, PUSH the register value into the stack, and then POPF the value into the Flags register. Later in the course, you will learn other flag manipulation instructions, but this is the only way you can modify all the flags at once.

Following is an example of how each instruction-type could be written.

```
POP BX          ;Load BX register from STACK
POP COUNT       ;Move STACK word to offset
                ;address COUNT and COUNT+1
POPF            ;Get Flags from STACK
```

Each of these instructions is just the opposite of the examples we gave for the various PUSH instructions. In the case of the POP to memory instruction, the low byte is moved to the address indicated by the label COUNT, and the high byte is moved to address COUNT+1. Again, you must make sure the memory location is defined as a **word**.

Figure 4-7 shows the "POP BX" operation. Notice that the stack is in the same condition as we left it in Figure 4-6, except the BX register has been cleared. When the instruction is executed, the data byte pointed to by the Stack Pointer is "popped" and stored as the low byte in the BX register. Then the data pointed to by the Stack Pointer "plus one" is popped and stored as the high byte in the BX register. Finally, the Stack Pointer value is **incremented** by two. As Figure 4-7B shows, the new top of stack address is 01F7H.

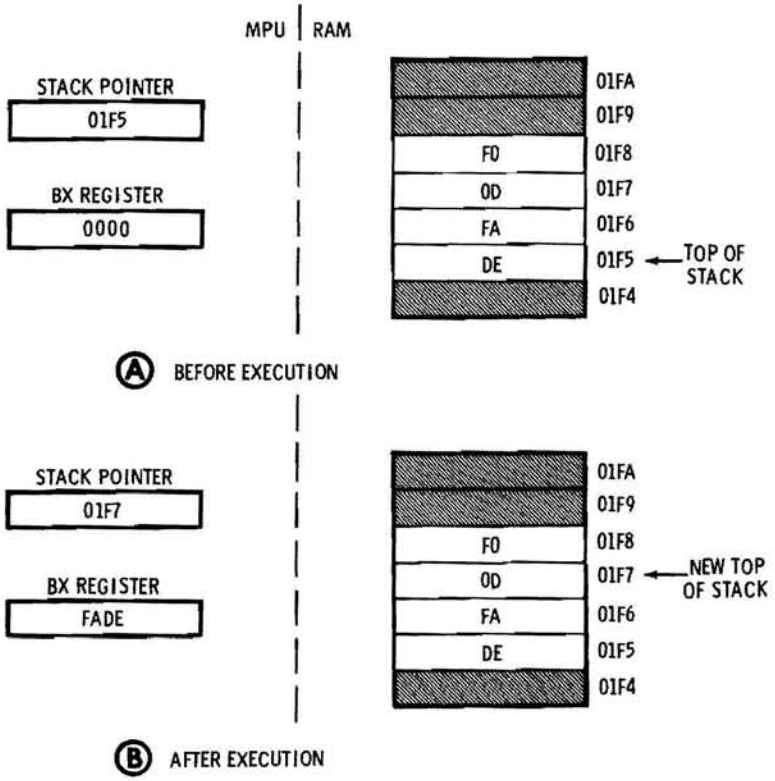


Figure 4-7
Executing the POP BX instruction.

Even though the top of stack has changed, the data stored at the old top of stack has not changed. It will remain in memory until new data is pushed into the stack, or the microcomputer is switched off.

One thing to keep in mind when using the PUSH and POP instructions, is that other than the values stored in the Stack Pointer and the Stack Segment registers, the MPU **doesn't know** where the "stack" is located in memory. Push too many data words and you may wind up writing over your program data or code. Conversely, popping more data than you pushed will move you into an area where the data is unknown. It may be reserved system control data, random RAM data, or ROM code, any of which will probably cause a program failure. **Always determine where your "stack" will reside, and then stay within those boundaries!**

The four instructions, PUSH, POP, PUSHF, and POPF are the only instructions that let you directly manipulate the stack. However, as we mentioned earlier, there are two other instructions that use the stack to save data during their operation. These, you may recall, are CALL and RET. We will describe their operation in the next section.

Self-Review Questions

7. The 8088 MPU stack uses the _____ principle of data storage.
   LIFO/FIFO

8. For easier data transfer, the stack is always located directly after the program code section. _____
   True/False

9. The _____ _____ register contains the stack base address.

10. The _____ _____ register contains the stack offset address.

11. The _____ instruction is used to move data directly into the stack.

12. To remove data from the stack, you would use the _____ instruction.

13. The _____ instruction is used to store the contents of the Flag register in the stack.

14. The _____ instruction is used to move a 16-bit value from the stack into the Flag register.

15. Depending on the instruction, you can save both words and bytes in the stack. _____
    True/False

16. The PUSH instruction decrements the SP by two and then stores the data at addresses SP and SP + 1. _____
    True/False

17. The POPF instruction increments the SP by two and then retrieves the data at addresses SP and SP + 1. _____
    True/False

18. You can pop more data from the stack than you pushed. _____
    True/False

19. The assembler directive _____ identifies an offset address location with a name.

20. The assembler operator _____ causes the assembler to calculate the offset address of a name.

THE CALL AND RETURN INSTRUCTIONS

A program jump to a subroutine is a permanent operation. That is, the microprocessor no longer knows where the main program is located. To get back to the main program, you need another "permanent" jump instruction. This is fine if you always jump back to the same point in the program, but what do you do if you want to use the same subroutine in two or more different areas of the program? What you need is a system to remember where you left the main program, so you can return to the same point.

Recall that the Instruction Pointer (IP) is used to tell the MPU the address of the next instruction to be executed. As soon as that instruction is read into the MPU, the value in the IP is changed to indicate the address of the **next** instruction. Therefore, if the IP value of the instruction after the jump is saved, the MPU will know where to return in the main program, **after** the subroutine is completed. You could, with a little manipulation, save the IP value on the stack and then use that value to jump back. But there is a better way, the CALL and RET instructions. This section will describe how each instruction works, and then present an example program where they are used to access and return from a subroutine.

The CALL Instruction

The **CALL** instruction is similar to the PUSH instruction in the way it moves data into the stack. But rather than store data, the CALL instruction stores the offset address for the next instruction in the main program. Figure 4-8 illustrates the process involved in executing a CALL instruction. Part A shows segments of the MPU and RAM prior to execution. The Instruction Pointer points to the CALL instruction at address 011AH in the main program. The Stack Pointer points to the top of stack at address 0214H. At this time you can consider the stack empty.

The CALL instruction occupies three bytes in memory. Hex code 0E8H is the instruction code. The next two bytes indicate the offset address of the destination or **target** operand calculated by the assembler. This is the starting address of the subroutine being "called." Since the CALL instruction occupies three bytes in memory, the fourth byte, at address 011DH, must contain the **next** instruction in the main program.

Figure 4-8B shows the result of executing the CALL instruction. To get to this position:

1. The CALL instruction is first read into the MPU.

2. The CALL is decoded.

3. The IP is updated to point to the address of the next **sequential** instruction.

4. The SP is decremented twice and the IP value is pushed into the stack location pointed to by the SP and SP+1. Thus, the stack now contains the values 1DH at address 0212H and 01H at address 0213H. This is the **return address** in the main program.

5. The IP value is then replaced with the new offset address value found in the CALL instruction destination operand. Thus, the next instruction that will be loaded into the MPU is the first instruction of the subroutine, 0B2H.

Once the Instruction Pointer has been shifted to the subroutine area of the program, the MPU begins executing the instruction code. However, there comes a time when the subroutine is finished and the MPU must return to the main program. This is accomplished with the **RET** (return from subroutine) instruction.

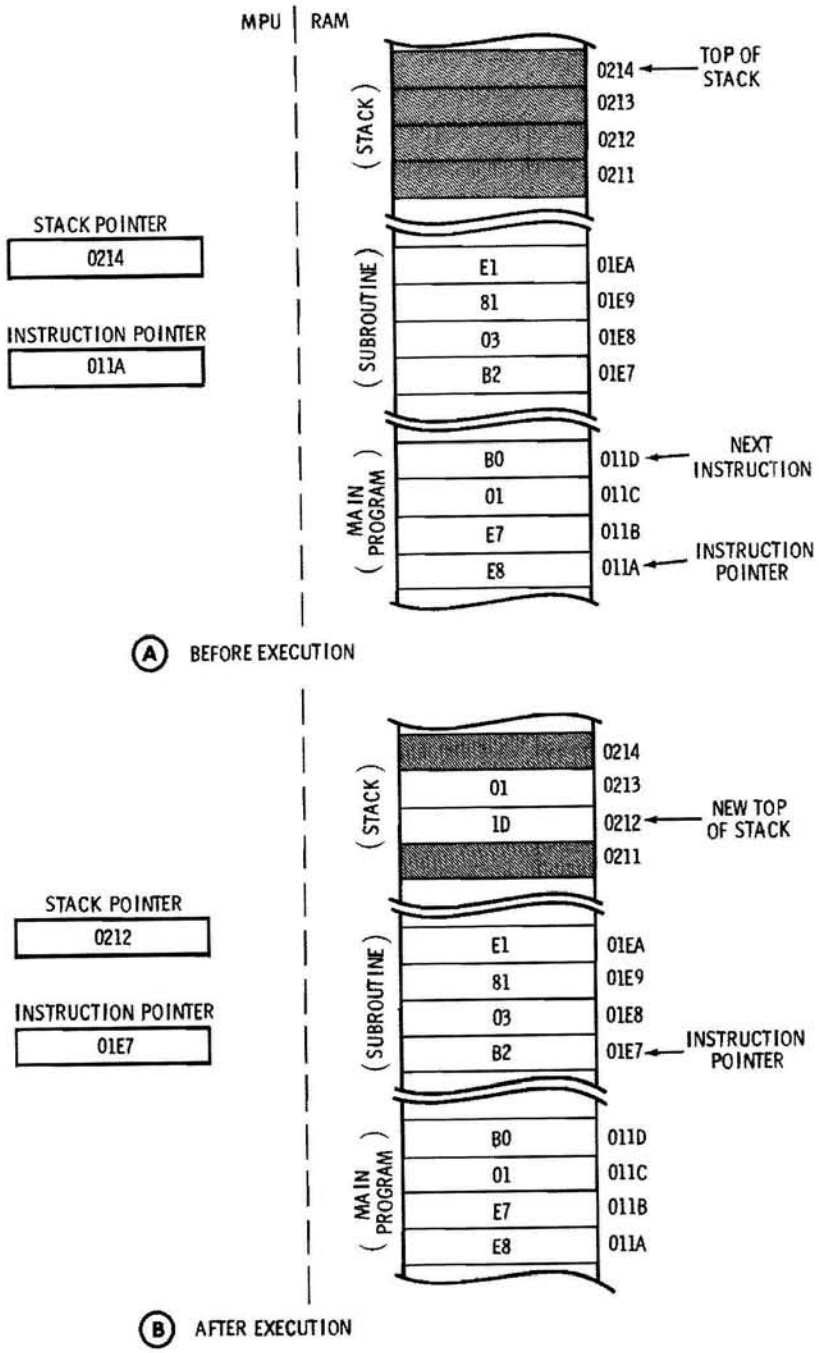


Figure 4-8

Executing a CALL to subroutine instruction.

The RET Instruction

Essentially, RET pops the data word from the top of the stack and places it in the Instruction Pointer. The MPU then uses the address in the IP to fetch the next instruction. Figure 4-9 illustrates the process involved in executing a RET instruction. Part A shows segments of the MPU and RAM prior to execution. The Instruction Pointer points to the RET instruction at address 01FCH in the subroutine. The Stack Pointer points to the current "top of Stack" at address 0212H. The data in the stack at this time is the value pushed there by the earlier CALL instruction.

Figure 4-9B shows the result of executing the RET instruction. To get to this position:

1. The RET instruction is first loaded into the MPU.

2. The RET instruction is then decoded, causing the data word at the top of the stack to be popped into the Instruction Pointer.

3. Finally, the Stack Pointer is incremented by two, to address 0214H.

4. The IP now contains the address of the next instruction to be fetched. This is the instruction immediately following the original CALL instruction described in Figure 4-8.

Once the Instruction Pointer has been shifted back to the main program, the MPU again begins executing code.

In addition to storing the return address for a called subroutine, the stack is often used as temporary storage for data that is to be passed to a subroutine. For example, the main program may gather data from a peripheral and then call a subroutine to process the data. Once the subroutine has processed the data, it's no longer of any value to the main program. However, the subroutine, can't change the Stack Pointer contents to point to a location in the stack above the useless data. The reason is quite simple. The current top of stack contains the return address to the main program. There is, however, a return instruction that will allow a return to the main program, and then adjust the contents of the Stack Pointer so that it points to a stack location above the useless data. This is the **return and immediate add** instruction.

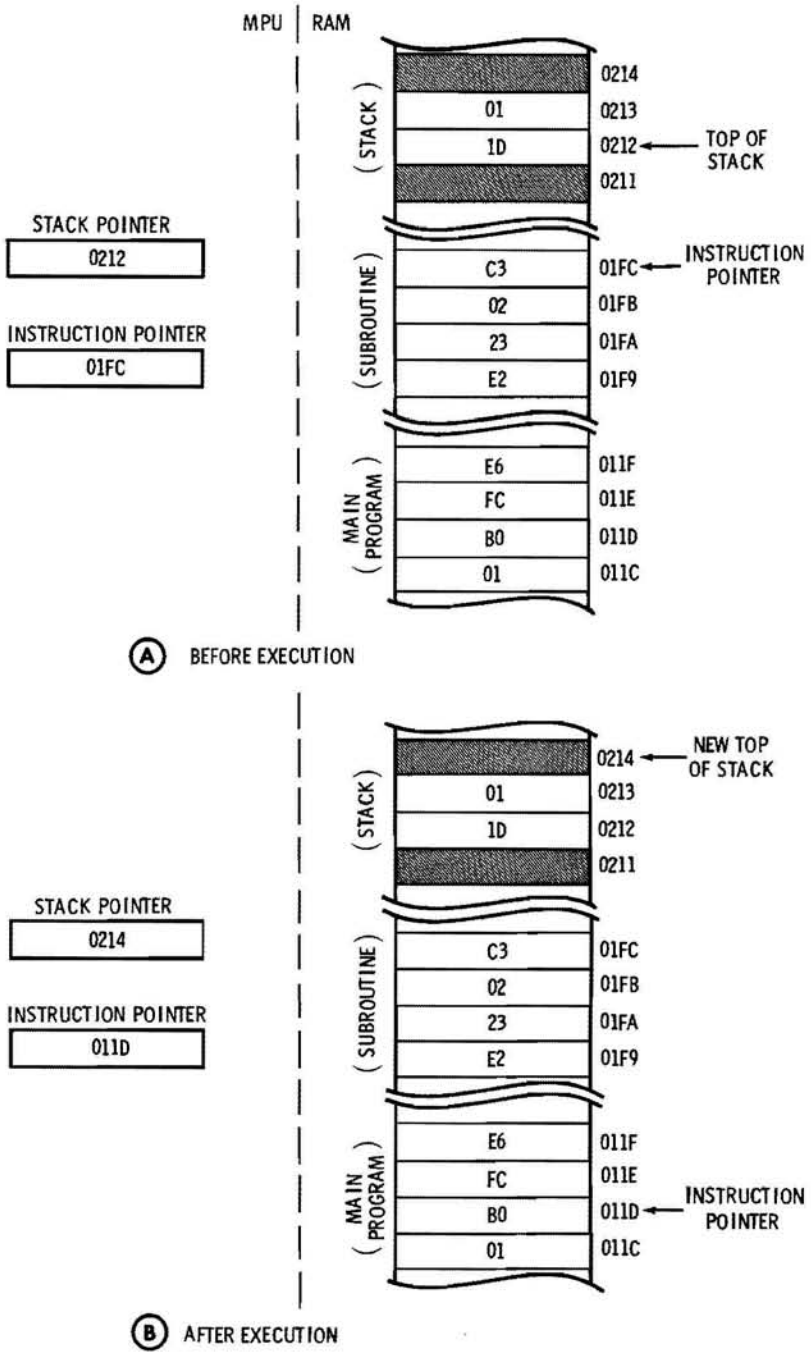


Figure 4-9

Executing a RET from subroutine instruction.

The mnemonic for this instruction is also RET. Therefore, to set it apart from a normal return instruction, the mnemonic is followed by the **immediate value** that is to be added to the Stack Pointer. As an example, "RET 4H" means place the address of the next instruction in the IP, then add **0004H** to the Stack Pointer. Figure 4-10 shows the effect of the RET 4H instruction. You may have noticed that it is very similar to the RET instruction in Figure 4-9. However, there are now three stages to the instruction.

In Part A, the instruction has not been executed. Thus, the Instruction Pointer contains the address of the RET instruction in the subroutine, 0127H. The Stack Pointer contains an address that points to the subroutine return address.

In Part B of Figure 4-10, the RET 4H instruction is fetched and decoded in the MPU. Two immediate results are shown in the figure. First, the IP is loaded with the address, 011DH, of the next instruction to be executed. Second, the Stack Pointer is incremented by two. Thus, the SP now points to address 01F6H in the stack.

Part C of the figure completes the RET 4H instruction by adding 4 to the Stack Pointer. Thus, the new top of stack is at address 01FAH. The four bytes of useless data have been bypassed.

Remember to always add the correct value to the Stack Pointer. Each digit represents **one** byte in memory. Therefore, if you pushed three **words** into the stack prior to the CALL, you would use the immediate value 6H with your RET instruction to bypass those values. Because the stack will only accommodate word values, the immediate value following the RET instruction is always an even number.

You may have noticed that even though we only added "4" to the Stack Pointer in Figure 4-10, the instruction had two bytes of immediate data, 00H and 04H. This is because the RET instruction is performing a word-sized add operation. Therefore, whenever you use this form of return instruction, always expect it to occupy three bytes in memory, even if the immediate value is 0FFH or less.

One last consideration. You should always end a "called subroutine" with a "return (or return and immediate add) from subroutine" instruction. The CALL and RET instructions are designed to complement each other. To get a better handle on how the CALL and RET instructions would be used to access a subroutine, let's examine the "Tick-Tock" program we flowcharted earlier in this unit.

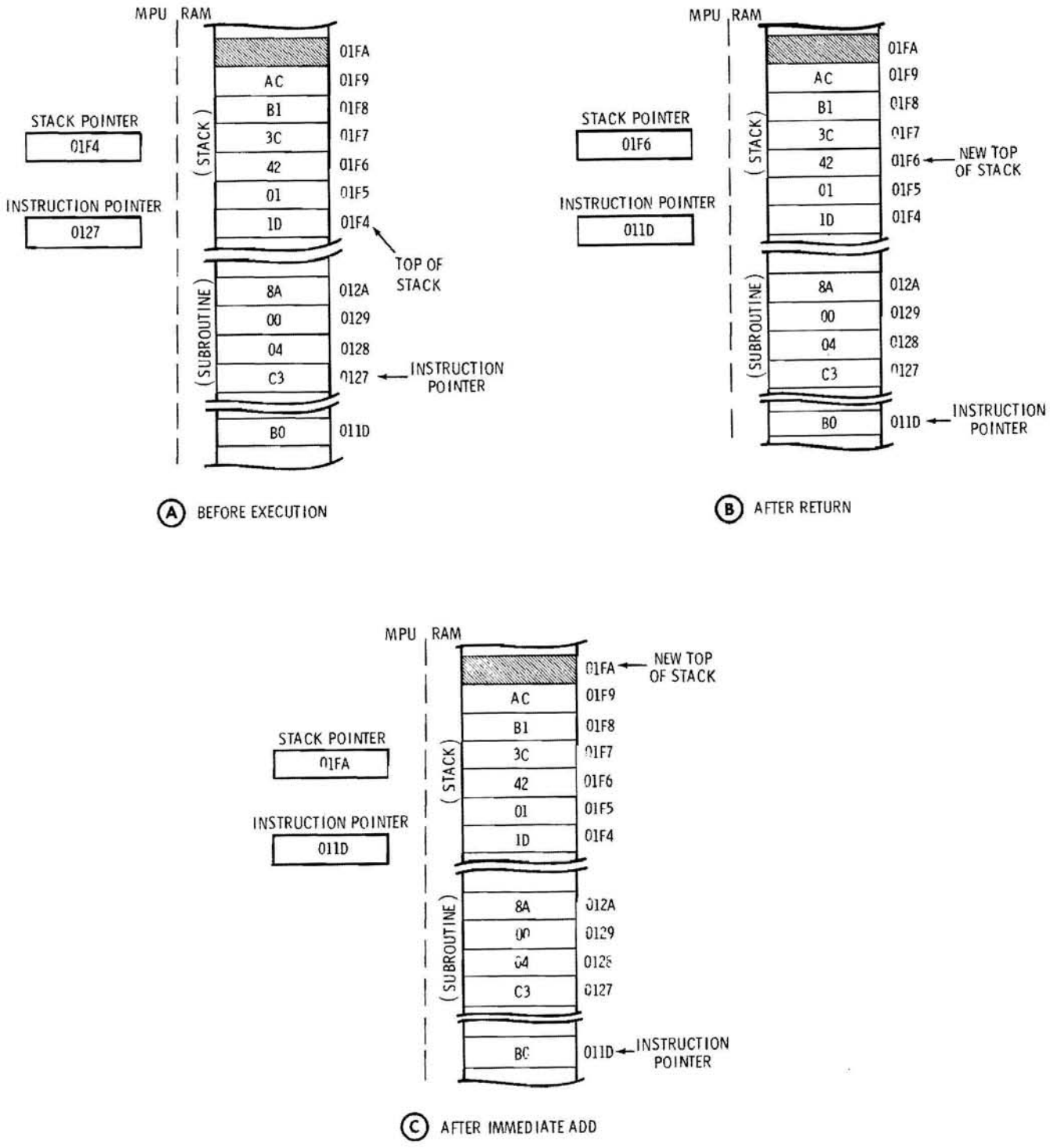


Figure 4-10
Executing a "return and immediate add" instruction.

Using CALL and RET

Recall that the "Tick-Tock" program is being used to control a hypothetical sound generator outside the microcomputer. Every half-second the microcomputer tells the generator to produce a sound of a "tick" or a "tock." The main program controls the transfer of data to the generator, while the subroutine provides the necessary half-second delay between data transfers. Figure 4-11 shows the program.

To begin, data byte 01H is loaded into the AL register. This value will be used to tell the generator to make the "tick." Next, the value in the AL register is transferred to the generator. The mnemonic OUT is the output command, while 25H is the address of the generator input/output port. We are introducing this instruction now to help illustrate our subroutine. Later in the course, we will fully explain how the IN and OUT instructions are used to interface the microcomputer to its peripherals. As soon as the generator gets its "tick" signal, the MPU calls the delay subroutine. The operand DELAY serves as the target address for the first instruction in the subroutine.

```
TITLE UNIT4 -- PROGRAM 2 -- TICK-TOCK GENERATOR
;
COM_PROG  SEGMENT
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  MOV     SP,OFFSET TOP_OF_STACK ;Get STACK address
GENERATE_SIGNAL:
        MOV     AL,01H          ;Data for tick
        OUT     25H,AL          ;Output to generator
        CALL    DELAY           ;Wait for half-second
        MOV     AL,0FCH         ;Data for tock
        OUT     25H,AL          ;Output to generator
        CALL    DELAY           ;Wait for half-second
        JMP     GENERATE_SIGNAL ;Repeat tick-tock
;
DELAY:  MOV     DL,02H          ;Start half-second delay
        MOV     CX,0AAAAH       ;Load count register
CYCLE:  NOP                     ;Waste time
        LOOP    CYCLE           ;Dec count, is it zero?
        DEC     DL              ;Count one loop
        JNZ     CYCLE           ;Main loop done? If not,
                                ;repeat inner loop
        RET                     ;Done, return from CALL
;
        DW      10 DUP (?)      ;STACK size defined
TOP_OF_STACK    LABEL   WORD    ;Identify top of STACK
;
COM_PROG ENDS
        END     START
```

Figure 4-11
"Tick-tock" program.

This particular subroutine relies on the fact that every instruction executed by the MPU takes a specific amount of time. This time is determined by the MPU **clock rate** and the number of **clock cycles** required to complete the instruction. Once you know the time for each instruction, it's a simple matter of adding up the number of instructions needed to create a specific delay. The instructions in the subroutine were selected to give that half-second delay required by the program. Let's see how the instructions are used to create that delay.

Data byte 02H is loaded into the DL register to serve as the main delay loop counter in the routine. Next, the Count register is loaded with 0AAAAH, to set up a second delay loop within the routine. Then a NOP instruction is executed. This provides a convenient method of wasting time without affecting any Flags or registers. In the case of the 8088 MPU, a NOP instruction takes three clock cycles to execute. (The Instruction Set in Appendix D lists the number of clock cycles used by the MPU to execute every instruction.) The following LOOP instruction decrements the Count register and then checks its contents for zero. Since the contents are certainly not zero now, the routine loops back to the target address CYCLE. The NOP-LOOP cycle continues until the Count register is zero, at which time, the program "falls-through" the loop to the next instruction. The DL register, our main loop counter, is decremented and then its contents are checked for zero. If DL is not zero, CYCLE is repeated 0FFFFH more times. This occurs because, the first time the LOOP instruction is executed, the CX register is decremented from zero to 0FFFFH.

When DL is finally zeroed, the half-second delay period is over. The next instruction, RET, sends the MPU back to the main program memory address found at the top of the stack. That address points to the MOV AL,0FCH instruction. Thus, the MPU begins execution of the main program by moving 0FCH into the AL register.

The AL register now contains the value needed to make the generator produce the "tock" sound. This value is transferred to the generator through the same output port, 25H, as was used for the "tick" value. After transfer is complete, the **delay** subroutine is again called. Only this time there is a new address pushed into the stack. Thus, when the MPU returns from the subroutine, the IP will have the memory address for the instruction JMP GENERATE_SIGNAL rather than the earlier instruction MOV AL,0FCH.

When the jump instruction is executed, the MPU will jump back to the beginning of the main program and load the "tick" value into the AL register. In this manner, the program is recycled. And, it will continue to cycle until the microcomputer is reset. You should not get the impression that you must return from a subroutine before calling another; subroutines can be nested.

Subroutine Nesting

Because the CALL instruction uses the stack to save the return address of the next instruction, a subroutine call can be made from within a subroutine. The only limiting factor is the capacity of the stack. Let's look at an example.

Figure 4-12 shows a situation in which the main program calls subroutine A. In turn, subroutine A calls subroutine B. Thus, subroutine B would be considered a nested subroutine. That is, a **nested subroutine** is a program segment that is called by another subroutine. If control is to be eventually returned to the main program, two return addresses must be saved and recalled in the proper order. Figure 4-12 is a very simple example of how this could be done.

Initially, assume the stack is empty and the main program instructions are being executed. When the CALL instruction is read and executed, three things happen. First, the SP register is decremented by two. Next, the address of the next instruction, IP value 0122H, is pushed into the stack. Then, the address offset in the CALL instruction is loaded into the IP register. This directs the MPU to **subroutine A**, which starts at offset address 0131H.

Notice that halfway through subroutine A another CALL instruction is encountered. When it is read and executed, the process is repeated. First the SP register is decremented by two. Next, the address of the next instruction, IP value 0138H, is pushed into the stack. Then the address of the address offset in the CALL instruction is loaded into the IP register. This directs the MPU to **subroutine B**, which starts at offset address 01D3H.

Subroutine B has no nested subroutines of its own, so the program flow is through the subroutine as shown. The last instruction in the subroutine is the RET instruction. When this instruction is read and executed, the return address is popped from the stack and placed in the IP register, and the SP register is incremented by two. The MPU then begins executing code at offset address 0138H in subroutine A, pointed to by the IP register.

When the RET instruction is read and executed, the process is repeated one more time. The return address at the top of the stack is moved into the IP register and the SP register is incremented by two. The MPU then begins executing code at address 0122H in the main program.

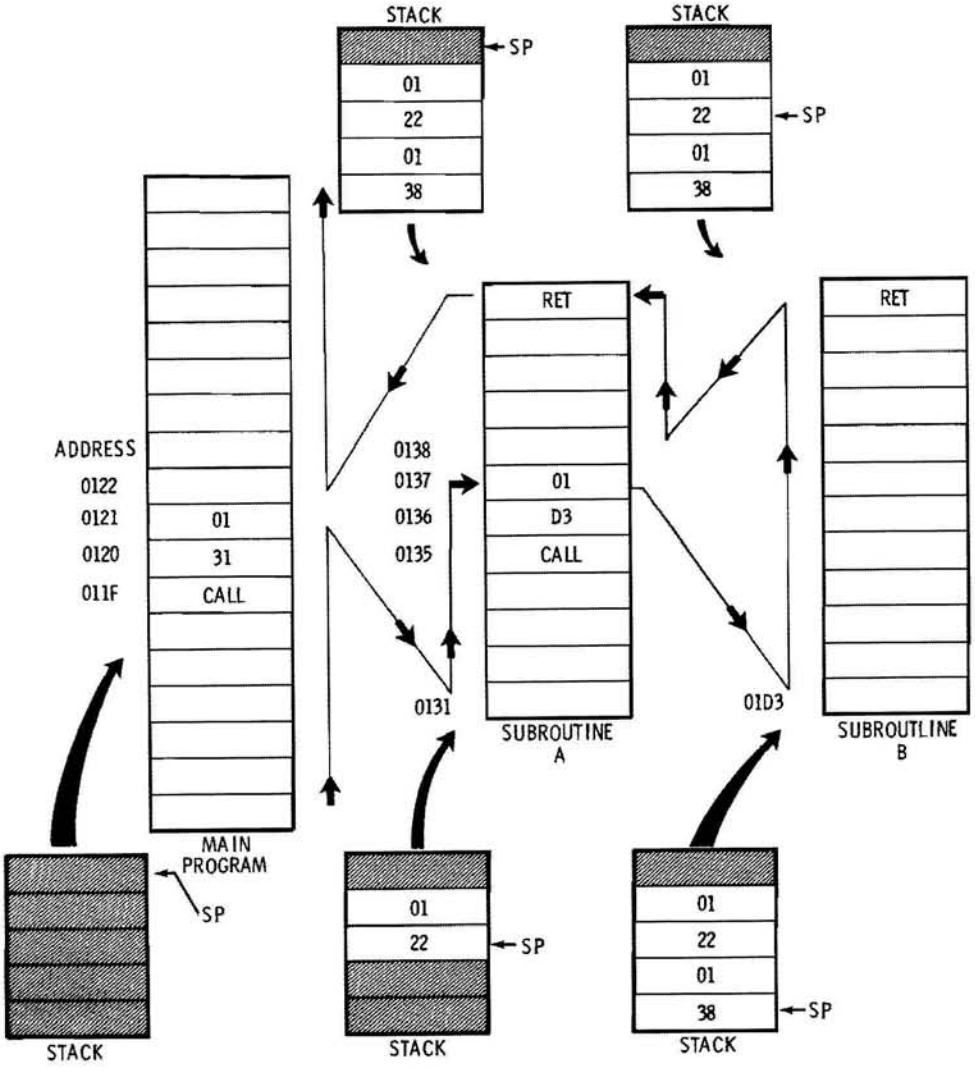


Figure 4-12
Handling nested subroutines.

Here are a few points to keep in mind when nesting subroutines:

1. Always use a return instruction to return from a subroutine call. This is especially important in nested subroutines.

2. Whenever you push data into the stack, always pop an equal amount of data from the stack. If you forget, the program may wind up returning to an unknown area of memory.

3. Avoid calling a previously nested subroutine. The subroutine return may not send the MPU to the correct (desired) program location.

4. Physically separate the code for a subroutine from the code for the main program or another subroutine. This will help reduce confusion if you must modify the code at some later date.

Self-Review Questions

21. The CALL instruction decrements the Stack Pointer by two and then stores the return address at addresses SP and SP+1.

_____
True/False

22. The RET instruction causes a change in program direction by moving an address into the _____ _____ register.

23. The target address for a return instruction is always at the top of the stack. _____
True/False

24. In order to bypass four words of data in a stack, the return immediate instruction must contain the immediate value _____.

25. A CALL instruction can be used to access more than one subroutine in a program. _____
True/False

26. A RET instruction at the end of a subroutine can cause the MPU to jump to any part of the program and begin execution.

_____
True/False

INCLUDING FILES

As you gain programming experience, you will find there are a number of subroutines that are repeated in nearly every program. Some of these routines, such as a disk file reader, contain over a page of code. Entering this code every time you write a program is just not practical. MACRO-86 has an assembler directive that solves the problem, it's called the INCLUDE directive.

The **INCLUDE** directive tells the assembler to read the specified ASM file, combine the code with the current ASM file, and assemble all of the code as one unit. The code being included must be valid source code, and it must be compatible with the current program being assembled. Code compatibility refers to the use of names, labels, and symbols. Improper references will be flagged as assembly errors.

The directive takes the form

```
INCLUDE    <file-name>
```

where <file-name> is the valid name for an ASM source code file. Examples of INCLUDE directives are:

```
INCLUDE    DELAY.ASM
INCLUDE    B:GEN_OUT.ASM
```

In the first example, DELAY.ASM is a subroutine that is located on the current, or active, disk drive. The second example, B:GEN_OUT.ASM, indicates that subroutine GEN_OUT.ASM is located on disk drive B. It is important that you indicate the location (disk drive) of the file if the file is not located on the current drive. Assuming the program is being assembled on drive A, any file that is included from drive B must have its file name preceded by a B: drive indicator.

Although we used the ASM file extension in both examples, it isn't necessary. You can use almost any extension, such as FIL, INC, or DAT; or you can leave off the extension. However, we recommend you don't use the predefined extensions we discussed earlier (EXE, COM, OBJ, CRF, LIB, and MAP).

File Format

Creating a file that will be included with another file is quite simple. You don't have to use any SEGMENT, ASSUME, or ENDS directives. All you need are code, data, and any comments that might make the file easier to identify. Any labels, symbols, or names should be unique to the file to prevent duplication in the main program.

Figure 4-13 is an example of a simple include routine. As a matter of fact, it's the half-second delay subroutine from the program in Figure 4-11. The code hasn't been changed; however, we have added a number of comments. These comments:

1. Identify the routine and its function.

2. Identify any data, from the main program, that is needed for the routine to function properly. In this example, none is required. The data can reside in a register or memory, although data is usually transferred through a register. The location must be identified.

3. Identify any data that must be returned to the main program. In this example, no data is returned. If data is returned, the location of that data must be specified.

4. Identify any registers that are modified by the routine. In this example, the CX and DL registers are modified. This means you either don't use CX and DL in the main program, or you save the contents of CX and DL in the stack prior to calling the subroutine or after the subroutine is called.

```
;INCLUDE file DELAY.INC is half-second delay routine.
;INPUT to routine:  None
;OUTPUT from routine:  None
;Registers modified:  DL and CX
;
DELAY:  MOV     DL,02H          ;Start half-second delay
        MOV     CX,0AAAAH       ;Load count register
CYCLE:  NOP                     ;Waste time
        LOOP    CYCLE           ;Dec count, is it zero?
        DEC     DL              ;Count one loop
        JNZ     CYCLE           ;Main loop done?  If not,
                                ;repeat inner loop
        RET                     ;Done, return from CALL
```

Figure 4-13
Example of an include file.

While there is no fixed format for comments, you should make it a habit to identify these four subjects. It could resolve a lot of questions when you try to include the file in another program.

Figure 4-14 is an example of a program using the assembler directive INCLUDE. It is the "tick-tock" program without the delay subroutine. We've replaced the delay subroutine with the directive:

```
INCLUDE    DELAY.INC
```

DELAY.INC is the name of the file in Figure 4-13. When the assembler encounters the directive INCLUDE, it will stop assembly and read the file into the main program source code. Then it will continue assembling the program from that point.

```
TITLE UNIT4 -- PROGRAM 3 -- USING AN INCLUDE FILE
;
COM_PROG  SEGMENT
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  MOV     SP,OFFSET TOP_OF_STACK ;Get STACK address
GENERATE_SIGNAL:
        MOV     AL,01H          ;Data for tick
        OUT     25H,AL          ;Output to generator
        CALL    DELAY           ;Wait for half-second
        MOV     AL,0FCH         ;Data for tock
        OUT     25H,AL          ;Output to generator
        CALL    DELAY           ;Wait for half-second
        JMP     GENERATE_SIGNAL ;Repeat tick-tock
;
        INCLUDE DELAY.INC
;
        DW      10 DUP (?)      ;STACK size defined
TOP_OF_STACK    LABEL   WORD    ;Identify top of STACK
;
COM_PROG ENDS
        END     START
```

Figure 4-14
Program using included file.

Figure 4-15 is the program listing generated by the assembler. Notice that the assembler treated the code from the main program, as well as the code from the included file, as one complete program. To help you identify which part of the code came from an include file, the letter C precedes each line of the included file beginning with the INCLUDE directive.

```
 The Microsoft MACRO Assembler            01-27-84     PAGE    1-1
UNIT4 -- PROGRAM 3 -- USING AN INCLUDE FILE


1                                       TITLE UNIT4 -- PROGRAM 3 -- USING AN IN
                                      CLUDE FILE
2                                       ;
3       0000                            COM_PROG  SEGMENT
4                                               ASSUME  CS:COM_PROG,DS:COM_PROG
                                      ,SS:COM_PROG
5                                       ;
6       0100                                    ORG     100H
7       0100  BC 0134 R                 START:  MOV     SP,OFFSET TOP_OF_STACK
                                      ;Get STACK address
8       0103                            GENERATE_SIGNAL:
9       0103  B0 01                             MOV     AL,01H          ;Data f
                                      or tick
10      0105  E6 25                             OUT     25H,AL          ;Output
                                       to generator
11      0107  E8 0113 R                         CALL    DELAY           ;Wait f
                                      or half-second
12      010A  B0 FC                             MOV     AL,0FCH         ;Data f
                                      or tock
13      010C  E6 25                             OUT     25H,AL          ;Output
                                       to generator
14      010E  E8 0113 R                         CALL    DELAY           ;Wait f
                                      or half-second
15      0111  EB F0                             JMP     GENERATE_SIGNAL ;Repeat
                                       tick-tock
16                                      ;
17                                      C       INCLUDE DELAY.INC
18                                      C  ;INCLUDE file DELAY.INC is half-second
                                      delay routine.
19                                      C  ;INPUT to routine:  None
20                                      C  ;OUTPUT from routine:  None
21                                      C  ;Registers modified:  DL and CX
22                                      C  ;
23      0113  B2 02                     C  DELAY:  MOV     DL,02H          ;Start
                                      half-second delay
24      0115  B9 AAAA                   C          MOV     CX,0AAAAH       ;Load c
                                      ount register
25      0118  90                        C  CYCLE:  NOP                     ;Waste
                                      time
26      0119  E2 FD                     C          LOOP    CYCLE           ;Dec co
                                      unt, is it zero?
27      011B  FE CA                     C          DEC     DL              ;Count
                                      one loop
28      011D  75 F9                     C          JNZ     CYCLE           ;Main l
                                      oop done?  If not,
29                                      C                                  ;repeat
                                       inner loop
30      011F  C3                        C          RET                     ;Done,
                                      return from CALL
31                                      C
32                                      ;
33      0120     0A [                           DW      10 DUP (?)      ;STACK
                                      size defined
```

Figure 4-15A

Program listing generated by the assembler.

```
 The Microsoft MACRO Assembler              01-27-84     PAGE    1-2
UNIT4 -- PROGRAM 3 -- USING AN INCLUDE FILE


34                      ????
35                           ]
36
37      0134                          TOP_OF_STACK    LABEL   WORD    ;Identi
                                    fy top of STACK
38                                    ;
39      0134                          COM_PROG ENDS
40                                          END     START
```

Figure 4-15B
Continuation of the program listing generated by the assembler.

Self-Review Questions

27. The assembler directive _____ is used to combine source code files to produce one file.

28. The file name TIME.OBJ is considered a valid name for a routine to be combined with another file or program. _____ True/False

29. When combining files, each file must contain a SEGMENT and an ENDS assembler directive. _____ True/False

30. Every include file should contain comments covering four subject areas. These subject areas are:

    A. _____

    _____

    B. _____

    _____

    C. _____

    _____

    D. _____

    _____

EXPERIMENT

Using Subroutines

OBJECTIVES:

1. *Demonstrate the 8088 MPU memory stack.*
2. *Demonstrate ways you can use the stack.*
3. *Demonstrate the subroutine instructions.*
4. *Demonstrate the INCLUDE assembler directive.*

Introduction

The jump and loop instructions introduced in Unit 3 opened the door to a number of new programming possibilities. This unit carried the process one step further by showing you how to jump to and return from specialized program areas called subroutines. To make the transition to the CALL and RET instructions, you had to learn about a new form of data storage used by the MPU called the stack. This experiment will give you an opportunity to play with the stack. It will also explore the programming possibilities using program subroutines both within the main program and in separate files that can be "included" when the main program is assembled.

Procedure

1. Although you probably won't use the stack manipulation instructions very often in your early attempts at programming, you should know how they affect the MPU and memory. The first program in this experiment is a "do nothing" that simply moves data around in the MPU registers and memory using the various stack instructions. Call up the editor and enter the program in Figure 4-16. When you finish, assemble and convert the source code into a COM file. Then load the COM file into memory with the debugger.

```
TITLE EXPERIMENT 4 -- PROGRAM 1 -- STACK MANIPULATION
;
COM_PROG  SEGMENT
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  MOV     SP,OFFSET TOP_OF_STACK ;Get STACK address
        MOV     AX,0EE55H       ;Load Accumulator
        MOV     BP,9999H        ;Load Base Pointer
        PUSH    AX              ;Store Accumulator in stack
        PUSH    BP              ;Store Base Pointer in stack
        PUSHF                   ;Store Flags in stack
        POP     CX              ;Move Flags into Count reg.
        POP     DATA2           ;Store Base Pointer in memory
        MOV     DATA1,0AAAAH    ;Store constant in memory
        POP     DATA1 + 2       ;Store Accumulator in memory
        INT     3               ;Return to Debugger
;
DATA1   DW      4 DUP (?)       ;Reserve 4 words in memory
DATA2   DW      0               ;Set memory word to zero
;
        DW      20H DUP (?)     ;STACK size defined
TOP_OF_STACK    LABEL   WORD    ;Identify top of STACK
;
COM_PROG ENDS
        END     START
```

Figure 4-16
Program to show how data is moved into and out of the stack.

2. Type "R" and RETURN. The Stack Pointer contains the value ____H.

When a program is loaded by the debugger, the debugger assigns the Stack Pointer value 0FFF0H. If you don't change that value with program code, the debugger will assume that is the top of stack. During debugger operations, address and control data are temporarily stored in the stack by the debugger. You will see the result of this temporary storage as we examine your program.

When a COM program is executed normally, the **system program loader** assigns a different value to the Stack Pointer. If there is sufficient memory, 0FFFFH is loaded into the Stack Pointer. If, on the other hand, the program memory segment does not contain 64K of memory, then the Stack Pointer is loaded with the offset address of the last byte in memory. Finally, the Stack Pointer is decremented-by-two and a word of zeros are loaded into the stack. These values are, of course, meaningless if the program loads a different value in the Stack Pointer, as is the case with the program you are examining.

3. Type "D100" and RETURN. Notice that the value 00 is contained within memory locations 011CH through 0165H.

   These are the memory locations identified by the program. As you single-step through the program, data will be moved into and out of this memory by the program and the debugger. The memory was initially zeroed by the debugger when it recognized the "DUP (?)" statement. The statement itself did not cause the zeros to be loaded.

4. Single-step through the program and answer the following questions. When you are asked to identify a value at a specific address, use the command "D100" to display the block of memory containing your program and its stack.

   A. After the first instruction is executed, the Stack Pointer contains the value _ _ _ _ H. Examine the stack. Notice that the debugger used the first three word locations at the top of the stack to temporarily store data. This data will change as you single-step through the program.

      The value 0166H is the offset address loaded into the Stack Pointer by the program. It points to the top of the program stack. The debugger uses the same stack for temporary storage of data.

   B. After the third instruction is executed, the Accumulator contains the value _ _ _ _ H and the Base Pointer contains _ _ _ _H.

      These values were established to give the MPU something to PUSH into the stack. The numbers have no significance.

C. After the fourth instruction is executed, address 0165H contains the value _ _ H and address 0164H contains the value _ _ H. The Stack Pointer contains the value ____H.

The PUSH AX instruction caused the Stack Pointer to decrement by two, to 0164H. Then the high byte in the Accumulator, 0EEH, was moved into location 0165H and the low byte, 55H, was moved into location 0164H. The debugger again used the stack to hold temporary data, but it made sure that the data pushed by the program was saved at the top of the stack.

D. After the fifth instruction is executed, address 0163H contains the value _ _H and address 0162H contains the value _ _H. The Stack Pointer contains the value ____ H.

This instruction pushed the contents of the Base Pointer into the stack. The Stack Pointer was decremented by two and now contains the value 0162H.

E. After the sixth instruction is executed, address 0161H contains the value _ _ H and address 0160H contains the value _ _H. The Stack Pointer contains the value ____H.

The PUSHF instruction moved the contents of the Flag register into the stack. Address 0160H contains the low-byte value of the register, while address 0161H contains the high-byte value of the register. Experience with many different systems using the 8088 MPU suggests that your Flag register contained 0F102H or a similar value. This might seem odd, since the debugger display indicates that the Flag register bits are zero.

Recall, however, that even though the Flag register contains only nine flag bits, it is treated as a 16-bit register during a push operation. Therefore, the MPU must somehow account for the remaining seven bits. It does this by treating these bits as blank, or "don't care", bits. Because the MPU doesn't care what value these bits contain, the value stored by the transfer operation is essentially random. For that reason, the third word in your stack may not be 0F102H.

Figure 4-17 shows how the bits are arranged in the Flag register. It also shows the binary equivalent for the value 0F102H that is probably stored in your stack. Notice that the eight flag bits recognized by the debugger are zero. The ninth bit, T (trap) flag, is set to one. This flag bit is used by the debugger, through the MPU, to control the single-step operation. It is always set when you are using the debugger; that's why the debugger doesn't display its condition. The remaining flag bits are those don't care bits. Because they have no guaranteed logic state, you should never rely on their condition to control an operation.

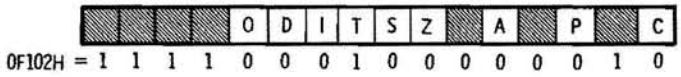


Figure 4-17

Arrangement of the Flag register bits and their 16-bit value. The shaded areas are "don't care" bits.

F. After the seventh instruction is executed, the Count register contains the value ____H and the Stack Pointer contains the value ____H.

The Count register now contains the same value as the Flag register. The process of popping that data from the stack also incremented the Stack Pointer by two. Something else happened that probably surprised you — the Flag register data disappeared from the stack. Under normal circumstances, that data would still be in the stack. However, the debugger is sharing the stack with the program. Thus, when data is popped from the stack, the debugger assumes it is free to use that area to store temporary data. It's always a good idea, when running two or more programs simultaneously, to know how each program affects the operation of the other.

G. After the eighth instruction is executed, address 0124H contains the value _ _ H and address 0125H contains the value _ _ H. The Stack Pointer contains the value ____H.

Until now, all of the stack operations have involved the MPU registers. The eighth instruction moved the value 9999H into a location in memory. Thus, you can see that you are not limited to register moves when using the stack. This is true for both pushing and popping data.

H. After the tenth instruction is executed, address 011EH contains the value _ _ H and address 011FH contains the value _ _ H. The Stack Pointer contains the value ____H.

The ninth instruction simply moved 0AAAAH into memory to provide you with a reference for the tenth instruction. That instruction popped the last word in the stack into a memory location that was indexed one word from the name DATA1 using the assembler operator add (+). Again, the Stack Pointer is incremented by two, returning it to the original top of stack value. Finally, the data in the stack is overwritten by the debugger. Thus, you have no record of the data you moved into the stack because of the debugger's manipulation of the program stack. Had the program been executed in a normal manner, the three words the program pushed into its stack would still be there.

Discussion

Working with the stack is not much different from working with the MPU registers or memory. You just have to remember where the data is located within the stack. Probably the most important rule for working with the stack is to always pop the data from the stack in reverse order. For instance, suppose you need to save the contents of three registers before you execute a subroutine. Before running the subroutine, you PUSH AX, BX, and DX. When the subroutine is completed, you POP DX, BX, and AX, in that order. If you pop the data in the same order that it was pushed, the contents of AX and DX will be exchanged.

The next program is a counting routine that illustrates the call and return from call instructions. So that you can indirectly watch the program use the CALL and RET instructions (they access a 1-second delay routine), we will also introduce a new system interrupt, INT 21H. Depending on how it is used, INT 21H lets your program control many functions of the computer including character display. Thus, when you run the program, you will be able to see the program counting and displaying a string of 80 decimal numbers.

Procedure Continued

5. Call up the editor and enter the program in Figure 4-18. Assemble and convert the source code to a COM file.

```
TITLE EXPERIMENT 4 -- PROGRAM 2 -- CALLING A SUBROUTINE
;
COM_PROG  SEGMENT
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  MOV     SP,OFFSET TOP_OF_STACK ;Get STACK address
        MOV     CX,80           ;Set number of counts
RESTART:MOV     DL,'0'-1        ;Set-up number register
                                ;with ASCII 0 minus 1
COUNT:  INC     DL              ;Make ASCII code for number
        CMP     DL,'9'          ;Is number above decimal range?
        JA      RESTART         ;Yes, restore number register
        MOV     AH,2            ;Output character interrupt function
        INT     21H             ;Function requesting interrupt
        CALL    DELAY           ;Delay one second between numbers
        LOOP    COUNT           ;Repeat if necessary
        MOV     AH,0            ;End program interrupt function
        INT     21H             ;Function requesting interrupt
;
DELAY:  MOV     BL,4            ;Start 1 second delay subroutine
        MOV     DI,04568H       ;Load count
DELAY1: DEC     DI              ;Count down for delay
        JNZ     DELAY1          ;Is inner delay loop done?
        DEC     BL              ;Count one inner delay loop
        JNZ     DELAY1          ;Main delay loop done?
        RET
;
        DW      100H DUP (?)    ;STACK size defined
TOP_OF_STACK    LABEL   WORD    ;Identify top of STACK
;
COM_PROG ENDS
        END     START
```

Figure 4-18

Program to demonstrate subroutine calling by displaying a string of numbers.

6. To run a program under the MS-DOS disk operating system, the program name must have the file extension COM, EXE, or BAT. However, when you command MS-DOS to run a program, you should only identify the program name — leave off the file extension. Thus, to run the program P4-6.COM, you would simply type "P4-6" and RETURN. MS-DOS will locate the program on the disk; load it into memory; and finally, tell the MPU to execute the code.

   Run your program — type the program name and RETURN. The disk drive will run for a few seconds, indicating that the program is being read into memory. Then the character 0 will appear on the next line of the display. After a 1-second delay, the character 1 will appear. This process will continue until the count reaches nine, then the count will repeat. One second after the eighth nine appears, the program will stop and control will be returned to MS-DOS.

Discussion

The program you just ran is designed to sequentially display the decimal numbers zero through nine, and repeat the sequence eight times. After each number is displayed, the program waits for one second. This waiting period is controlled by a 1-second delay subroutine that is called by the program.

The display process is controlled by a subroutine in MS-DOS. The subroutine is "called" by using an interrupt instruction; in this case, INT 21H. Interrupt 21H actually comprises a group of computer-controlling subroutines. To access one of these subroutines, you identify the routine number; then you execute the interrupt. The character display subroutine is routine number 2.

At the end of the program, system control is returned to MS-DOS through another INT 21H subroutine. In this case, the routine is number 0. Later in the course, we will be using a number of different INT 21H subroutines. If you are interested in examining the other INT 21H operations, refer to the Appendix of your "DOS" manual, under Interrupt Function Calls.

Now let's examine the program in detail.

Procedure Continued

7. Load your program COM file into memory with the debugger. Using Figure 4-19 as a reference, single-step to the first interrupt instruction at address 0111H.

   By now you should be familiar with the code in these program steps, so we'll limit the description to the instructions. To begin, the value 80 is loaded into the Count register to set the number of numbers that will be displayed. Then the ASCII value for the character 0 minus one is loaded into the DX register. Two important concepts are introduced in this step.

   First, whenever you use the ASCII code for a printable character, it's a good idea to use the character enclosed within quotes, and let the assembler determine the numeric code. It's not that you might make a mistake entering the code; rather, it makes the instruction easier to understand. The character 0 means more than the value 30H.

   The second concept is tied to the first. We could have used the value 2FH and accomplished the same goal, but subtracting one helps to show the purpose of the instruction. It is setting up a register to hold the ASCII value of the numbers that will be displayed. To generate each number, one is added to the value in the register. Since this instruction isn't part of the number generating (COUNT) loop, it must store an ASCII value one less than the first character to be displayed.

   The first instruction in the COUNT loop adds one to the ASCII value in the DL register. This produces the ASCII code for the character 0. Each time the loop is repeated, one is added to produce the next sequential decimal number character.

   Since the program is designed to display only decimal numbers, DL must be tested to determine when it increments beyond the decimal number range. This is accomplished by comparing DL to ASCII 9. The next instruction, jump if above, then makes the decision to continue the loop or start a new count loop.

```
 The Microsoft MACRO Assembler            02-10-84     PAGE    1-1
EXPERIMENT 4 -- PROGRAM 2 -- CALLING A SUBROUTINE


1                                       TITLE EXPERIMENT 4 -- PROGRAM 2 -- CALL
                                      ING A SUBROUTINE
2                                       ;
3       0000                            COM_PROG  SEGMENT
4                                               ASSUME  CS:COM_PROG,DS:COM_PROG
                                      ,SS:COM_PROG
5                                       ;
6       0100                                    ORG     100H
7       0100  BC 0329 R                 START:  MOV     SP,OFFSET TOP_OF_STACK
                                      ;Get STACK address
8       0103  B9 0050                           MOV     CX,80           ;Set nu
                                      mber of counts
9       0106  B2 2F                     RESTART:MOV     DL,'0'-1        ;Set-up
                                       number register
10                                                                      ;with A
                                      SCII 0 minus 1
11      0108  FE C2                     COUNT:  INC     DL              ;Make A
                                      SCII code for number
12      010A  80 FA 39                          CMP     DL,'9'          ;Is num
                                      ber above decimal range?
13      010D  77 F7                             JA      RESTART         ;Yes, r
                                      estore number register
14      010F  B4 02                             MOV     AH,2            ;Output
                                       character interrupt function
15      0111  CD 21                             INT     21H             ;Functi
                                      on requesting interrupt
16      0113  E8 011C R                         CALL    DELAY           ;Delay
                                      one second between numbers
17      0116  E2 F0                             LOOP    COUNT           ;Repeat
                                       if necessary
18      0118  B4 00                             MOV     AH,0            ;End pr
                                      ogram interrupt function
19      011A  CD 21                             INT     21H             ;Functi
                                      on requesting interrupt
20                                      ;
21      011C  B3 04                     DELAY:  MOV     BL,4            ;Start
                                      1 second delay subroutine
22      011E  BF 4568                           MOV     DI,04568H       ;Load c
                                      ount
23      0121  4F                        DELAY1: DEC     DI              ;Count
                                      down for delay
24      0122  75 FD                             JNZ     DELAY1          ;Is inn
                                      er delay loop done?
25      0124  FE CB                             DEC     BL              ;Count
                                      one inner delay loop
26      0126  75 F9                             JNZ     DELAY1          ;Main d
                                      elay loop done?
27      0128  C3                                RET
28                                      ;
29      0129  0100 [                            DW      100H DUP (?)    ;STACK
                                      size defined
30                    ????
31                        ]
```

Figure 4-19A

Assembler listing of the subroutine call program.

```
 The Microsoft MACRO Assembler              02-10-84     PAGE    1-2
EXPERIMENT 4 -- PROGRAM 2 -- CALLING A SUBROUTINE


32
33      0329                            TOP_OF_STACK    LABEL   WORD    ;Identi
                                    fy top of STACK
34                                      ;
35      0329                            COM_PROG ENDS
36                                              END     START
```

Figure 4-19B

Continuation of the assembler listing of the subroutine call program.

Assuming the count is continued, the next instruction loads the interrupt 21H function select number into the AH register. When the interrupt is executed, the MPU examines the AH register to determine what system function is to be performed. The number two indicates a "display character" function. It is assumed by the display character subroutine that the ASCII code for the character to be displayed is in the DL register.

8. The next instruction to be executed is INT 21H. Single-stepping through an interrupt is practically not feasible. The best way to handle interrupts while in the debugger is to use the debugger Go command, with a break point, and let the MPU execute the interrupt. To do this, you would type G and the offset address in the program where execution should stop. For your program, that offset address is the address of the next instruction. Execute the interrupt, type "G113" and RETURN. Notice that when the debugger executed the interrupt, the character 0 appeared on the next line. Then the normal debugger register display appeared on the following lines.

9. Single-step through the next instruction, the CALL DELAY instruction. Notice that the Stack Pointer register decremented by two, and the Instruction Pointer now contains the offset address of the first instruction in the DELAY subroutine. Now examine the top of stack, type "D327" and RETURN. Offset address 0327H contains the value _ _ H and address 0328H contains the value _ _ H. These contain the low-byte and high-byte of the offset address of the next instruction.

10. The DELAY subroutine is similar to the one described earlier in this unit, only here the delay is for one second (IBM approximately 1.1 seconds) rather than one-half second. Also, the NOP and LOOP instructions have been replaced with DEC and JNZ instructions because the Count register is being used in the main program. Since it will take too long to single-step through the DELAY subroutine, use the GO with break point command to jump to its end. Type "G128" and RETURN. The Instruction Pointer should be pointing at the return from subroutine instruction.

11. Single-step through the RET instruction. The Stack Pointer is incremented by two, and the Instruction Pointer contains the offset address value that was previously stored in the stack by the CALL instruction. The next instruction to be executed is the LOOP COUNT instruction that follows the CALL DELAY instruction. You've now had the opportunity to observe the results of the operation of the call and return from call instructions. Rather than repeat the loop, let's exit the program gracefully.

12. First, you must reduce the Count register contents to one (not zero). Type "RCX" and RETURN. The debugger will display the contents of the Count register and prompt you for a new value. Type "1" and RETURN. The Count register now contains the value one.

13. Single-step through the LOOP COUNT instruction. The Counter is decremented to zero and the loop is ignored.

14. Single-step through the next instruction. This loads the value zero into the AH register. We could have used the SUB AH,AH instruction and saved execution time, but it wouldn't have conveyed the same meaning. This instruction is loading the function number for the next INT 21H instruction. The act of loading a number is more significant. The number zero identifies the interrupt subroutine that performs the necessary system housekeeping (file and memory handling) to properly exit a program.

15. Again, you shouldn't try to single-step through an interrupt; use the Go command instead. Since this is the last instruction in the program, you won't need a break point. Type "G" and RETURN. The debugger will display the message "Program Terminated Normally." Notice that, because you executed the program while you were running the debugger, you are still in the debugger. To exit, you will have to type "Q" and RETURN.

Discussion

This program gave you a chance to see how a subroutine is called and exited. It also introduced two new interrupt instructions: Interrupt 21H, function 2 is used to display a character; while interrupt 21H, function 0 is used to exit a program and return control to the system or another program.

You may have also noticed another change in the program from previous programs. The stack size was increased from 20H to 100H. This change was necessary to support the interrupt instructions. MS-DOS suggests that you add 80H words of memory to any program stack requirements when using an interrupt 21H instruction.

Procedure Continued

16. Modify your program so that it will count and display all of the hexadecimal number characters; leave the total count at 80. Run the program to verify that it works.

Discussion

Figure 4-20 is one example of how the problem of displaying hexadecimal characters can be solved. It is by no means the only way. You could have used a number of subroutine calls to generate the count. The important point is that you were able to make your program generate the hexadecimal number set.

```
TITLE EXPERIMENT 4 -- PROGRAM 3 -- HEXADECIMAL NUMBER DISPLAY
;
COM_PROG  SEGMENT
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  MOV     SP,OFFSET TOP_OF_STACK ;Get STACK address
        MOV     CX,80           ;Set number of counts
RESTART:MOV     DL,'0'-1        ;Set-up number register
                                ;with ASCII 0 minus 1
COUNT:  INC     DL              ;Make ASCII code for number
        CMP     DL,'9'          ;Is number above decimal range?
        JA      HEX_CHAR        ;Yes, start hex character output
        MOV     AH,2            ;Output character interrupt function
        INT     21H             ;Function requesting interrupt
        CALL    DELAY           ;Delay one second between numbers
        LOOP    COUNT           ;Repeat if necessary
;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
HEX_CHAR:
        MOV     DL,'A'-1        ;Set-up number register for hex values
COUNT_HEX:
        INC     DL              ;Make ASCII code for number
        CMP     DL,'F'          ;Is number above hex range?
        JA      RESTART         ;Yes, restore decimal number register
        MOV     AH,2            ;Output character interrupt function
        INT     21H             ;Function requesting interrupt
        CALL    DELAY           ;Delay one second between numbers
        LOOP    COUNT_HEX       ;Repeat hex count if necessary
;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
        MOV     AH,0            ;End program interrupt function
        INT     21H             ;Function requesting interrupt
;
DELAY:  MOV     BL,4            ;Start 1 second delay subroutine
        MOV     DI,04568H       ;Load count
DELAY1: DEC     DI              ;Count down for delay
        JNZ     DELAY1          ;Is inner delay loop done?
        DEC     BL              ;Count one inner delay loop
        JNZ     DELAY1          ;Main delay loop done?
        RET
;
        DW      100H DUP (?)    ;STACK size defined
TOP_OF_STACK    LABEL   WORD    ;Identify top of STACK
;
COM_PROG ENDS
        END     START
```

Figure 4-20

Program to display the hexadecimal character set.

Since we can't evaluate your program, let's look at the program in Figure 4-20. The code to generate the hexadecimal characters is positioned between the two rows of x's to make it easier for you to locate. It is identical to the original decimal display code; only here, the DL register is loaded with the ASCII code for the character A minus one. To get to this section of code, the previous jump if above (JA) instruction's target address is changed to point to the code.

Because the program loop count ended with the character F being displayed, the "end program" interrupt function location didn't have to be changed. However, if you have no control over the loop count, there is a possibility the program could end after a decimal number is displayed. In this case, you should make some provision for executing the interrupt regardless of where the program ended. One option is to place an unconditional jump instruction after each loop instruction, and have the jump target the "end program" interrupt function.

Before we move on to the next section, there is one more concept that should be covered concerning interrupts. They are simply subroutines that are part of the disk operating system (MS-DOS). Like all programs, they use registers within the MPU. Normally, the interrupt preserves the contents of any register it may use on the stack. Then, before returning to the calling program, it pops the data back into the registers. There are, however, exceptions to the rule; some registers may not be preserved. These instances are identified in the DOS Manual, in the description of the interrupt. For example, interrupt 21H function 2 does not preserve the contents of the AX register.

Procedure Continued

17. The last section of Unit 4 described how common subroutines could be saved as files on a disk, and then "included" when needed in a program. Let's see if you still remember the process. Remove the 1-second delay subroutine from your counting program and make it an include file with the name DELAY.INC. Then rewrite the counting program so that it will "include" the file DELAY.INC. Run the program to verify that it works.

Discussion

Figure 4-21 is an example of what your include file could look like. Did you remember to preface the file with the appropriate comments? It may not seem important now; but when you start writing larger subroutines, those comments can save you time.

```
;INCLUDE file DELAY.INC is a 1-second time delay subroutine.
;INPUT to subroutine:  NONE
;OUTPUT from subroutine:  NONE
;Registers modified: BL and DI
;
DELAY:  MOV     BL,4            ;Start 1 second delay subroutine
        MOV     DI,04568H       ;Load count
DELAY1: DEC     DI              ;Count down for delay
        JNZ     DELAY1          ;Is inner delay loop done?
        DEC     BL              ;Count one inner delay loop
        JNZ     DELAY1          ;Main delay loop done?
        RET
```

Figure 4-21
Include file example.

Figure 4-22 shows the main program with the INCLUDE assembler directive. When this program is assembled, the source code in file DELAY.INC will be combined with the program source code to produce a whole program.

This completes the Experiment for Unit 4. Now is a good time to experiment with the programs you wrote, and possibly write a few more experiments to test your understanding of the material presented. For instance, what would happen if you deleted the DELAY subroutine and the two CALL DELAY instructions in the hexadecimal counting program? When you have finished experimenting, proceed to the Unit 4 Examination.

```
TITLE EXPERIMENT 4 -- PROGRAM 4 -- INCLUDING A SUBROUTINE
;
COM_PROG  SEGMENT
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  MOV     SP,OFFSET TOP_OF_STACK ;Get STACK address
        MOV     CX,00           ;Set number of counts
RESTART:MOV     DL,'0'-1        ;Set-up number register
                                ;with ASCII 0 minus 1
COUNT:  INC     DL              ;Make ASCII code for number
        CMP     DL,'9'          ;Is number above decimal range?
        JA      HEX_CHAR        ;Yes, start hex character output
        MOV     AH,2            ;Output character interrupt function
        INT     21H             ;Function requesting interrupt
        CALL    DELAY           ;Delay one second between numbers
        LOOP    COUNT           ;Repeat if necessary
HEX_CHAR:
        MOV     DL,'A'-1        ;Set-up number register for hex values
COUNT_HEX:
        INC     DL              ;Make ASCII code for number
        CMP     DL,'F'          ;Is number above hex range?
        JA      RESTART         ;Yes, restore decimal number register
        MOV     AH,2            ;Output character interrupt function
        INT     21H             ;Function requesting interrupt
        CALL    DELAY           ;Delay one second between numbers
        LOOP    COUNT_HEX       ;Repeat hex count if necessary
        MOV     AH,0            ;End program interrupt function
        INT     21H             ;Function requesting interrupt
;
        INCLUDE DELAY.INC       ;1-second time delay subroutine
;
        DW      100H DUP (?)    ;STACK size defined
TOP_OF_STACK    LABEL   WORD    ;Identify top of STACK
;
COM_PROG ENDS
        END     START
```

Figure 4-22
Program using an include file.

UNIT 4 EXAMINATION

1. The stack used in an 8088-based microcomputer is: _____

   _____

   _____

2. The Stack Pointer is decremented twice after a word of data is pushed into the stack. _____
   True/False

3. The contents of the _____ _____ register are pushed into the stack when a subroutine is called.

4. The PUSHF instruction has the MPU store:

   A. Bytes of data.
   B. Words of data.
   C. Flag register data.
   D. General register data.

5. Execution of the return instruction at the end of a subroutine causes the MPU to: _____

   _____

   _____

6. The top of stack location in memory is usually identified with the assembler directive _____.

7. The INCLUDE assembler directive lets you combine two COM files during program assembly. _____
   True/False

8. When the instruction MOV AX,OFFSET STORAGE is executed:

   _____

   _____

   _____

EXAMINATION ANSWERS

1. The stack used in an 8088-based microcomputer is a group of temporary storage locations in memory, usually located after the program code and data.

2. **False**. The Stack Pointer is decremented twice **before** a word of data is pushed into the stack.

3. The contents of the **Instruction Pointer** register are pushed into the stack when a subroutine is called.

4. **C** — The PUSHF instruction has the MPU store Flag register data.

5. Execution of the return instruction at the end of a subroutine causes the MPU to pop the word of data at the top of stack into the Instruction Pointer, increment the Stack Pointer by two, and execute the instruction pointed to by the contents of the Instruction Pointer.

6. The top of stack location in memory is usually identified with the assembler directive **LABEL**.

7. **False**. The INCLUDE assembler directive lets you combine two ASM files during program assembly.

8. When the instruction MOV AX,OFFSET STORAGE is executed, the offset address value of the name STORAGE is moved into the AX register.

SELF-REVIEW ANSWERS

1. A subroutine is a step-by-step **procedure** for doing a particular job.

2. **False**. A subroutine can be used an unlimited number of times.

3. A subroutine is called with the **CALL** instruction.

4. The last instruction in a subroutine is the **RET (return)** instruction.

5. The Trapezoid-shaped box in a flowchart represents an **input/output** process.

6. The small circle in a flowchart represents a **change** in program direction.

7. The 8088 MPU stack uses the **LIFO** principle of data storage.

8. **False**. The location of the stack has no bearing on the ease of data transfer.

9. The **Stack Segment** register contains the stack base address.

10. The **Stack Pointer** register contains the stack offset address.

11. The **PUSH** instruction is used to move data directly into the stack.

12. To remove data from the stack, you would use the **POP** instruction.

13. The **PUSHF** instruction is used to store the contents of the Flag register in the stack.

14. The **POPF** instruction is used to move a 16-bit value from the stack into the Flag register.

15. **False**. Only word-sized values can be saved in the stack.

16. **True**. The PUSH instruction decrements the SP by two and then stores the data at addresses SP and SP + 1.

17. **False**. The POPF instruction first retrieves the data at addresses SP and SP + 1; then it increments the SP by two.

18. **True**. You can pop more data from the stack than you pushed. However, that additional data will be of an unknown quantity.

19. The assembler directive **LABEL** identifies an offset address location with a name.

20. The assembler operator **OFFSET** causes the assembler to calculate the offset address of a name.

21. **True**. The CALL instruction decrements the Stack Pointer by two and then stores the return address at addresses SP and SP + 1.

22. The RET instruction causes a change in program direction by moving an address into the **Instruction Pointer** register.

23. **True**. The target address for a return instruction is always at the top of the stack. If it is not, the MPU will not return to the correct location in the main program.

24. In order to bypass four words of data in a stack, the return immediate instruction must contain the immediate value **eight**.

25. **False**. A CALL instruction can not be used to access more than one subroutine in a program, since its destination operand contains the address of the subroutine being called.

26. **True**. A RET instruction at the end of a subroutine can cause the MPU to jump to any part of the program and begin execution, since the return address is located in the program stack.

27. The assembler directive **INCLUDE** is used to combine source code files to produce one file.

28. **False**. The file name TIME.OBJ is **not** considered a valid name for a routine to be combined with another file or program. The file extension OBJ should be used only with "object files."

29. **False**. When combining files, each include file must not contain a SEGMENT and an ENDS assembler directive. Only the main program should contain these directives.

30. Every include file should contain comments covering four subject areas. These subject areas are:

    A. Identify the routine and its function.

    B. Identify any data from the main program that is needed for the routine to function properly.

    C. Identify any data that must be returned to the main program.

    D. Identify any registers that are modified by the routine.

INSERT

Unit 5

NEW ADDRESSING MODES

CONTENTS

INTRODUCTION

The first four units used three basic forms of addressing: immediate, register, and direct. While these were adequate for our programming examples, they didn't allow you to use the full potential of the MACRO-86 instruction set. This unit will complete the addressing mode family by introducing you to the four variations of register indirect addressing: register indirect, based, indexed, and based index. Keep in mind that these will be treated as **intrasegment** modes of addressing. That is, addressing that is confined to a single 64K segment of memory. Unit 7 shows you how these addressing modes can be used for **intersegment** addressing (accessing any location in memory).

Indirect addressing is used primarily for accessing data. This data can take the form of display character sets, look-up tables, records, and structures. Display character sets and look-up tables are simply strings of ASCII codes or numeric values. Records and structures, on the other hand, are arranged in a specific manner to allow direct access to unique groups or elements of data. MACRO-86 has a number of assembler directives to aid you in handling this data.

Before we can describe records and structures, we must first present a number of new methods for initializing data. This information is covered in the section "Data Storage." The indirect method for addressing data is the final topic in this unit.

Use the "Unit Objectives" that follow to evaluate your progress. When you can successfully accomplish all of the objectives, you will have completed this unit. You can use the "Unit Activity Guide" to keep a record of those sections that you have completed.

UNIT OBJECTIVES

When you complete this Unit, you will be able to:

1. Explain the operation of the LEA instruction.

2. Define the terms: array, record, structure, and field.

3. Define the assembler directives: define doubleword (DD), define quadword (DQ), define ten-byte (DT), STRUC, ENDS, and RECORD.

4. Define the assembler operator MASK.

5. Define a register displacement value.

6. Explain the use of the assembler directive brackets.

7. Define the four indirect addressing modes: register indirect, based, indexed, and based index.

8. Write simple programs that use the indirect forms of addressing.

UNIT ACTIVITY GUIDE

	Completion Time
☐ Read the Section on "Data Storage."	_____
☐ Complete Self-Review Questions 1-8.	_____
☐ Read the Section on "Structures."	_____
☐ Complete Self-Review Questions 9-17.	_____
☐ Read the Section on "Records."	_____
☐ Complete Self-Review Questions 18-25.	_____
☐ Read the Section on "Register Indirect Addressing."	_____
☐ Complete Self-Review Questions 26-35.	_____
☐ Perform the Experiment.	_____
☐ Complete the Unit 5 Examination.	_____
☐ Check the Examination Answers.	_____

DATA STORAGE

Until now, you have assigned data storage in memory in terms of bytes and words. To help locate the data, you gave it a name. There has been no attempt to arrange the data in any specific manner. This unit will show how MACRO-86 allows you to group data into two unique forms — the RECORD and the STRUCTURE. However, before we describe these two methods of arranging data, we need to define a common term for "groups of data." We'll call these groups "arrays."

Arrays

Every time you define more than one byte or word of data in a program, you create an array of data. An **array** is a sequence of common data elements. Two or more defined bytes or words can constitute an array. For example:

```
BYTE_ARRAY1  DB  'A CHARACTER STRING' ;18-byte array
BYTE_ARRAY2  DB  18 DUP (?)  ;Uninitialized 18-byte array
```

Each of these assembler directive statements produce an array of data bytes; one initialized with a string of ASCII character codes and the other uninitialized. In both cases, a single directive established the array. You can also use a group of directives to establish the array. For example:

```
DATA_TABLE    DW   1   ;Initialized word of data
              DW   2   ;Initialized word of data
              DW   3   ;Initialized word of data
              DW   4   ;Initialized word of data
```

Although only one directive in this example is identified by a name, each directive can be addressed through that name. Simply add 2, 4, or 6 to the target address value to address the second, third, or fourth word of data. The important point to remember is that the first byte in a byte-sized array, or word in a word-sized array, is located at offset zero with respect to the array's target address. The remaining bytes, or words, are located at offset multiples of one, or two, into the array. For example, to retrieve the first word in word-array DATA_TABLE, you could use the instruction:

```
MOV  AX,DATA_TABLE
```

Then, to retrieve the third word in word-array DATA_TABLE, you could use the instruction:

```
MOV   AX,DATA_TABLE+4
```

If the first word is located at offset zero, then the second word is located at offset two and the third word must be located at offset four.

The arrays we've been discussing contain bytes and words of data. MACRO-86 provides three other methods for defining data in a program. These are the doubleword, the quadword, and the ten-byte word assembler directives.

More Data Definition

As you begin writing more complex programs, you will discover the need to assign large values to a specific memory location. While the DB and DW assembler directives will satisfy most of your needs, there will be times when they just won't work. MACRO-86 provides three define data directives that will satisfy those specific needs. They are: define doubleword (DD), define quadword (DQ), and define ten-byte (DT). Each lets you define and initialize different data sizes. As their names imply, DD defines values up to two words in length, DQ defines values up to four words in length, and DT defines values up to ten bytes in length. When you try to access this data, both the DD and DQ memory locations default to word-size values. Although DT is defined as a 10-byte memory location, it also defaults to word-sized values when accessed for a move data operation. Both the DQ and DT directives are specifically designed to handle large numbers for the 8087 Numeric Data Processor when it is used with the 8088/8086 MPU. You can use these directives to assign data even if you aren't using the 8087. However, we recommend that you stick with DB, DW, and DD.

Figure 5-1 is a portion of an assembler listing showing how these define data assembler directives can be used. The DB and DW directives are included to let you compare each of the define directives.

```
45      0193  01                        DEFINE_BYTE     DB      1
46      0194  12                        DEFINE_BYTE1    DB      12H
47      0195  0001                      DEFINE_WORD     DW      1
48      0197  1234                      DEFINE_WORD1    DW      1234H
49      0199  01 00 00 00               DOUBLE_WORD     DD      1
50      019D  78 56 34 12               DOUBLE_WORD1    DD      12345678H
51      01A1  01 00 00 00 00 00         QUAD_WORD       DQ      1
52            00 00
53      01A9  EF CD AB 90 78 56         QUAD_WORD1      DQ      1234567890ABCDE
                                  FH
54            34 12
55      01B1  00 00 00 00 00 00         TEN_BYTE        DT      1
56            00 00 00 01
57      01BB  00 00 00 00 00 11         TEN_BYTE1       DT      1112131415
58            12 13 14 15
59      01C5  00 11 12 13 14 15         TEN_BYTE2       DT      111213141516171
                                  819
60            16 17 18 19
61      01CF  02 12 13 14 15 16         TEN_BYTE3       DT      111213141516171
                                  81910
 Error  ---        29:Division by 0 or overflow
62            17 18 19 10
```

Figure 5-1
Examples of the five define directives.

Earlier, you discovered that the listing for data defined by the DW directive did not match the way the data was stored in memory. It really didn't matter since you were dealing with only two bytes of data. However, you should realize most major programs, like MACRO-86, contain a few minor problems, or bugs. This is one of those bugs; the data should be listed in the order it will be stored in memory.

On the other hand, the MACRO-86 listing for a doubleword and a quadword properly present the data in the order in which it is stored in memory, with the least significant byte first. Where the defined data doesn't fill all of the reserved memory space, the most significant memory locations are filled with zeros. This is illustrated on lines 49 and 51 of the figure.

The define ten-byte directive is designed to allow you to load decimal values directly into memory. As you know, the assembler automatically converts any decimal value into hexadecimal to conserve memory space. However, when a decimal value is used with the DT directive, it is not converted into hexadecimal, it remains a decimal. As a result, decimal data is loaded into memory two digits per byte. Naturally, the define ten-byte directive will store ten byte-sized, decimal values. Lines 55 through 60 show that for values less than ten bytes long, the unused byte locations are always filled with zeros. The lines also show that the least significant digit of the decimal value is stored in the last byte. Line 61 shows that there is a bug in MACRO-86. It will not store a decimal value that contains 19 or 20 decimal digits. To attempt such an operation will generate an assembly error. This is true for all known versions of MACRO-86. You should be aware of one more problem. After MACRO-86 version 1.10, the data is stored in reverse of the earlier versions. For example, with version 1.10 the value 1112131415 is arranged and later stored in memory in the order:

```
00 00 00 00 00 11 12 13 14 15
```

With version 1.25 the value 1112131415 is arranged and later stored in memory in the order:

```
15 14 13 12 11 00 00 00 00 00
```

Because of this difference between versions, we suggest you use the DT directive cautiously.

These five data directives give you the power to construct data arrays to fit any programming need. You will find, however, that the DB, DW, and DD directives are used more often than DQ and DT. In fact, the primary purpose for DQ and DT is data transfer between the MPU and the 8087 Numeric Data Processor. Since this course isn't covering the interface between the MPU and the 8087, we won't use the DQ and DT directives in our programming examples.

This section has completed the description of all of the MACRO-86 data definition instructions. It has also laid the ground work for grouping data, in this case, in the form of arrays of common data elements. The next section will carry this concept a step further and show you how MACRO-86 can be used to structure data into arrays of uncommon data elements. But first, let's review the concepts presented in this section.

Self-Review Questions

1. The _____ _____ assembler directive allows you to assign two words of data to memory.

2. A/An _____ is a group of two or more common data elements.

3. The _____ _____ assembler directive creates what could be considered a 10-byte array of data.

4. The _____ _____ assembler directive allows you to assign four words of data to memory.

5. Indicate, after each of the following assembler directives, the data width (byte or word) assumed by MACRO-86.

   DB _____
   DW _____
   DD _____
   DQ _____
   DT _____

6. Write the instruction that will move the contents of the AL register into the third byte of the array named DATA_BYTE.

   _____

7. Write the instruction that will move the contents of the BX register into the fourth word of the array named DATA_WORD.

   _____

8. Write the instruction that will move the first byte of data in memory, initialized by the assembler directive statement:

```
TEN_BYTE  DT  1234567890H
```

   into the AL register.

   _____

NOTE: The Self-Review Answers are located at the end of this unit, beginning on Page 5-74.

STRUCTURES

A **structure** is a group of data bytes and/or words that are arranged in a specific manner. The structure is further broken down into data elements called **fields**. By assigning unique data elements to each field, that data is easily accessed during program run-time. The real power of this type of data handling in MACRO-86 is that it is designed to work with an array of common data structures. To initialize a data structure or an array of data structures, you must first design the structure template.

Structure Template

The structure template is a form of "blueprint." It shows the assembler how the related structures in a program are to be generated when the program is assembled. Figure 5-2 is an example of a typical structure template. The STRUC (structure) and ENDS (end structure) directive statements define the beginning and end of the structure template. The structure must have a unique name. In this example, we've called the structure STRUCTURE_NAME.

Each field in the structure must also be identified by a unique name. To keep things simple, we've called the five fields FIELD_1 through FIELD_5. The size of each field can be defined with either the DB, DW, or DD directives. The data within each field is defined in the structure template. Later, when the program structure is initialized, the data for that particular structure can be changed, or overridden, to meet specific requirements in the program. There is one constraint; if the field contains more than one data element, the data can only be defined in the template. This means that the data in FIELD_3 and in FIELD_4 must be specified in the template; it cannot be changed when a structure is initialized. Naturally, during program run-time, data can be moved into and out of any structure field.

```
STRUCTURE_NAME  STRUC
FIELD_1         DW      ?
FIELD_2         DB      0
FIELD_3         DB      7,3,1
FIELD_4         DD      3 DUP (12345678H)
FIELD_5         DB      '2/28/84'
STRUCTURE_NAME  ENDS
```

Figure 5-2
Typical structure template.

Structure Initialization

Once the structure template is constructed, any number of structures can be initialized from that template. The initialization command takes the form:

[name] [structure-name] <[exp][,...]>

where:

[name]

is the name you have assigned to the structure being initialized,

[structure-name]

identifies the structure template being used to initialize the structure, and

<[exp][,...]>

specifies a (possibly empty) list of field-initialization or optional field-override values. The angle brackets identify the structure fields, while commas are used to separate the fields. The fields default to the order they were defined. For example:

<>	means retain the originally defined values.
<5>	means override the first field value with 5.
<5,5>	means override the first and second field values with 5.
<,5>	means override the second field value with 5.
<,,5>	means override the third field value with 5.
<5,,5>	means override the first and third field values with 5.
<,5,5>	means override the second and third field values with 5.

Each of these examples suggest the existence of one, two, or three fields in the structure. However, there could have been more. Any field following those enclosed by the angle brackets are assumed to be unchanged. Thus, each of the examples could have referred to a structure containing four or more fields.

The term:

[exp]

represents a constant, a string, or the indeterminate character "?", used to define a field in the structure template.

Only simple fields can be overridden. A simple field can be defined with a DB, DW, or DD directive, but it cannot be a multiple expression (a list or a DUP clause). The one exception to this restriction is the DB character string. For example, in Figure 5-2, FIELD_1, FIELD_2, and FIELD_5 can be overridden because they contain single expressions or a character string; FIELD_3 and FIELD_4 cannot be overridden because they contain multiple expressions.

Strings should only be used to override strings. If a shorter string is used to override a longer string, any unconverted bytes are replaced with the ASCII code for a **space** character (20H). Strings that are longer than the original string are automatically truncated to the correct number of characters.

Thus far, we've only talked about initializing a single structure. MACRO-86 has the capability to duplicate a large number of identical structures from a single template. The initialization command takes the form:

[name] [structure-name] [exp] DUP (<[exp][,...]>)

where the familiar DUP clause creates as many copies of the template as specified. Thus, the term:

[exp] DUP

indicates that the structure is duplicated [exp] number of times. As with every DUP clause, parentheses enclose the expression being duplicated.

```
TITLE UNIT 5 -- PROGRAM 1 -- DEFINING STRUCTURES
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
        ORG     100H
START:
;
STRUCTURE_NAME  STRUC
FIELD_1         DW      ?
FIELD_2         DB      0
FIELD_3         DB      7,3,1
FIELD_4         DD      3 DUP (12345678H)
FIELD_5         DB      '2/28/84'
STRUCTURE_NAME  ENDS
;
LARRY   STRUCTURE_NAME  <0ABCDH,'A',,,'123456'>
HARRY   STRUCTURE_NAME  2 DUP (<,,,,'123456789'>)
;
COM_PROG  ENDS
        END     START
```

Figure 5-3
Initializing a structure.

Figure 5-3 is a simple program that contains a structure template and two structure initialization statements. Program code was left out, since our primary concern here is the structure. When the program is assembled, a structure named LARRY will be created from the template. Two of the fields are duplicated, while the other three are changed as indicated in the initialization statement. In this example: FIELD_1 will contain the word 0ABCDH, FIELD_2 will contain the ASCII code for the character "A", and FIELD_5 will contain the ASCII code for the character string "123456".

The second initialization statement contains a DUP clause. Therefore, in this example, two identical structures will be created by the assembler. As you can see, only FIELD5 is changed; it is loaded with the ASCII code for the character string "123456789".

Figure 5-4 shows the assembler source listing of our simple structure program. Lines 7 through 17 contain the structure template. Notice that the address offset of the template begins at 0000H and ends at 0019H. This data is used by the assembler for reference purposes only; the template does not become part of the program code.

The structure named LARRY begins at address offset 0100H. This structure is part of the program. Because there is no instruction code, it resides at the beginning of the program. Notice that FIELD_5, at address offset 0112H, contains the new character string 31H, 32H, 33H, 34H, 35H, and 36H. Because the string is one byte shorter than the template, the last byte contains the ASCII code for the "space" character, 20H.

The structure named HARRY is duplicated twice. Because the structure is duplicated, the assembler listing uses the standard duplication format. In this example the dupe count, 02, precedes the duplicated data enclosed within square brackets. The only field changed in the initialization statement is FIELD_5. Again, the character string is replaced by another character string. Only this time, the new character string is longer than the original. As a result, the excess bytes are truncated, beginning with the last byte entered. The string is shown on lines 37 and 38.

(CAUTION: Assembler versions 1.00 through 1.07 do not properly duplicate the structure. That is, the first field is duplicated count times, but then the remaining fields are duplicated only once.)

```
 The Microsoft MACRO Assembler            03-15-84     PAGE    1-1
UNIT 5 -- PROGRAM 1 -- DEFINING STRUCTURES


1                                       TITLE UNIT 5 -- PROGRAM 1 -- DEFINING S
                                    TRUCTURES
2       0000                        COM_PROG  SEGMENT
3                                           ASSUME CS:COM_PROG,DS:COM_PROG,
                                    SS:COM_PROG
4       0100                                ORG     100H
5       0100                        START:
6                                   ;
7                                   STRUCTURE_NAME  STRUC
8       0000  ????                  FIELD_1         DW      ?

9       0002  00                    FIELD_2         DB      0

10      0003  07 03 01              FIELD_3         DB      7,3,1

11      0006     03 [               FIELD_4         DD      3 DUP (12345678
                                    H)
12                  78 56 34 12
13                               ]
14
15      0012  32 2F 32 38 2F 38    FIELD_5         DB      '2/28/84'

16            34
17      0019                        STRUCTURE_NAME  ENDS
18                                  ;
19      0100  ABCD                  LARRY   STRUCTURE_NAME  <0ABCDH,'A',,,'
                                    123456'>
20      0102  41
21      0103  07 03 01
22      0106     03 [
23                  78 56 34 12
24                               ]
25      0112  31 32 33 34 35 36
26            20
27
28      0119     02 [               HARRY   STRUCTURE_NAME  2 DUP (<,,,,'12
                                    3456789'>)
29                    ????
30                    00
31                    07
32                    03
33                    01
34                 03 [
35                    78 56 34 12
36                               ]
37                    31 32 33 34
38                    35 36 37
39                             ]
40                             ]
41
42                                  ;
43      014B                        COM_PROG  ENDS
44                                          END     START
```

Figure 5-4
Assembler source listing of the structure program.

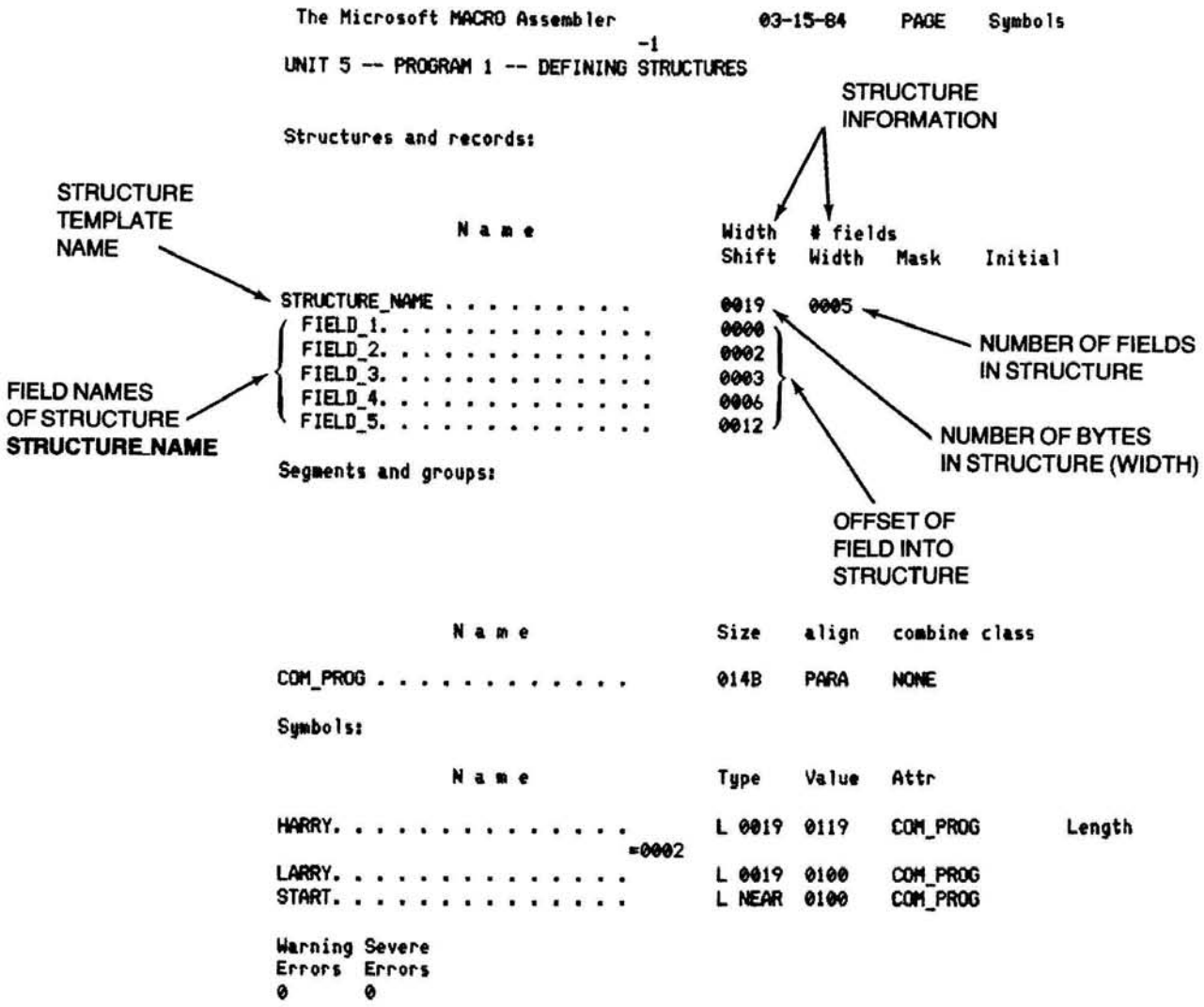


Figure 5-5
Last portion of the assembler source listing shown in Figure 5-4.

Figure 5-5 is the last page of the assembler listing. It shows that along with segments and symbols, data concerning structures and their fields are also listed. Each **structure template** is identified in the "Structures and Records" table. Following each structure template name is the name of each field in that template. The field names are indented to make them easier to locate.

The two values following the structure template name are the **width**, or number of bytes used by the template, and the number of **fields** in the template. After each field name is a value that indicates the number of bytes each field is offset into the template. Remember, the values used in this and in the other tables are hexadecimal, not decimal.

The actual program structures are identified in the "Symbols" table. Reading from left to right, the structure HARRY is 19H bytes long, it begins at address offset 0119H, it is located in the segment COM_PROG, and it is composed of two structures (Length =0002). The structure LARRY is also 19H bytes long, it begins at address offset 0100H, and it is also located in the segment COM_PROG. Since there is no "Length" specified, you can assume there is a single structure associated with the name LARRY.

After defining the structure template and initializing the program structures, the last step is to reference, or access, the individual fields of those structures.

Structure Access

Normally, when you access a variable, you identify the location through its **name**. That isn't practical with a structure, since the location is actually referenced by two unique names — the structure name and the field name. To resolve that problem, MACRO-86 has you combine the two names to produce a single "structure reference" name, with the form:

[structure name].[field name]

where [structure name] is the name of the initialized structure rather than the name of the structure template. The [field name] is the name of the field being accessed within that structure. A period (.) is used to separate the names. Thus, to access the data in FIELD_2 of the structure LARRY, you could use the instruction:

```
MOV    AL,LARRY.FIELD_2
```

Using the data from the structure program in Figure 5-4, you can see that this instruction will move the ASCII code for the character "A" into the AL register. When you access a structure field, make sure the source and destination operand sizes match.

Accessing data in a multiple element field requires a little more care. If it is the first item, you can use the previous example as a guide. For example, the instruction:

```
MOV     AL,LARRY.FIELD_3
```

moves the value 7 into the AL register. To access the second or third item in FIELD_3 you must use an arithmetic operator to provide the additional offset into the field. For example, to move the third item in FIELD_3 into the AL register, you would use the instruction:

```
MOV     AL,LARRY.FIELD_3+2
```

With the first item in FIELD_3 located at offset zero of the field, the third item must be located at offset two.

Accessing doubleword data also requires care on your part. Referring back to Figure 5-4, the instruction

```
MOV     AX,LARRY.FIELD_4
```

will move the value 5678H into the AX register. This is because the DD statement stores the low-byte of the doubleword expression first, followed by the next lowest byte, and so on. Thus, when you access the first word, you are actually accessing the **low-word** in the expression. Naturally, the instruction

```
MOV     AX,LARRY.FIELD_4+2
```

will move the high-word (1234H) into the AX register.

This concludes the discussion on structures. Remember, the structure template is used to establish the basic format for the structures you will use in the program. All of the fields can be undefined, or they can contain specific data. When you initialize a structure — allocate memory space to that structure — you can leave the fields unchanged, or you can modify their contents. Often, the final contents of a structure are determined at program run-time.

Self-Review Questions

9. A _____ is a group of data bytes and/or words that are arranged in a specific manner.

10. The data elements of a structure are called _____.

11. The beginning of a structure template is identified by the directive statement _____.

12. The end of a structure template is identified by the directive statement _____.

Refer to Figure 5-6 for the following questions.

```
TEMP    STRUC
F1      DB      '03/05/84'
F2      DD      1290H
F3      DW      ?
F4      DB      24,33,85,12
F5      DW      5 DUP (?)
TEMP    ENDS
```

Figure 5-6
Figure for questions 13 through 17.

13. The structure template contains _____ fields.

14. List the name of each field that can be overridden.

    _____
    _____
    _____
    _____
    _____

15. Write a structure initialization statement named APPLE that duplicates the template.

    _____

16. Write a structure initialization statement named PEAR that duplicates the template three times and changes the contents of the third field to 0ABCDH.

17. Write the instructions that will move the ASCII code for the day from the structure PEAR into the AX register, assuming the first field contains the character string for the month, day, and year.

RECORDS

The last section showed you how to create structures of data bytes and words. Records are similar in concept to structures. However, rather than being an arrangement, or pattern, of bytes and words, a **record** is an arrangement, or pattern, of bits **within** a byte or word. Like the structure, a record is made up of fields. Each field contains one or more data bits. The total number of field bits cannot exceed eight for a byte-sized record or 16 for a word-sized record. To initialize a record or an array of similar records, you must first design the record template.

Record Template

The record template shows the assembler how the related records in a program are to be generated when the program is assembled. A record template takes the form:

[record-name] RECORD [[field-name]:[width][= exp]][,...]

where:

[record-name]

is the name assigned to the record template,

RECORD

is the assembler directive that establishes the template,

[field-name]

is a unique name that identifies a field in the record, and

:[width]

indicates the number of bits in the field. It must be a constant in the range of 1 to 16. The sum of the field-name widths must fall within the range of 1 to 16. If the sum is 8 or less, a byte-sized record is defined. If the sum is between 9 and 16, a word-sized record is defined. It's possible to have a record that contains up to 16 single-bit fields. The colon must be used to separate the field-name from its width.

Again, like the structure template, the record template can be used to define the contents of a field. This is indicated by the term:

[=exp]

where exp is a default value with the following characteristics:

1. It can be retained or overridden when the record is initialized.

2. If no default value is specified, zero is used.

3. If specified, the default value evaluates to a positive integer expressible in the number of bits defined by the field. For example, a field three bits wide can hold the value 7 (111B), but not 8 (1000B).

4. If the field is exactly eight bits wide, it may be initialized to a **single** ASCII character as in:

```
FIELD_C:8='A'
```

Now let's look at a simple record template containing no defined data.

```
CHIPS RECORD RAM:7,EPROM:4,ROM:5
```

The record template CHIPS contains three fields: RAM, EPROM, and ROM. When the template is used to initialize a record, the fields will have the following characteristics:

Field Name	Field Width	Bit Positions	Maximum Value
RAM	7	15-9	$2^7-1=127$
EPROM	4	8-5	$2^4-1=15$
ROM	5	4-0	$2^5-1=31$

The "Bit Positions" column indicates where the field resides within the record, with 0 the least significant bit. The "Maximum Value" column indicates the largest value that can be stored within a field. Since "Field Width" indicates the number of binary bits in the field, that value becomes the exponent in the formula to determine the maximum field value. One is subtracted from the total because the maximum value is always one less than the maximum binary count.

The record template CHIPS contained no defined values. Adding defined values to the template will produce a record template similar to CHIPS2.

```
CHIPS2   RECORD   RAM2:7=4,EPROM2:4=2,ROM2:5=25
```

In this template, the field RAM2 is seven bits wide and it contains the value 4; the field EPROM2 is four bits wide and it contains the value 2; and the field ROM2 is five bits wide and it contains the value 25.

Earlier, we said the record could contain anywhere from 1 to 16 bits. While it is always a good idea to use records that are 8 or 16 bits long, they can be shorter. For example, the record template:

```
CHIPS3   RECORD   RAM3:7=4,EPROM3:4=2
```

contains two fields with a total bit count of 11. The five unused bits are considered undefined. Thus, CHIPS3 is considered an 11-bit record template. Fields are always right justified in the template; therefore, EPROM3 occupies bit positions 0 through 3, RAM3 occupies bit positions 4 through 10, and the undefined bits occupy positions 11 through 15.

Field bits that are not assigned a value in the record template are automatically given the value zero when the program is assembled. Unused bits are also zeroed by the assembler.

Record Initialization

Once the record template is constructed, any number of records can be initialized from that template. The initialization command takes the form:

[name] [record-name] <[exp][,...]>

where:

[name]

is the name you have assigned to the record being initialized,

[record-name]

identifies the record template being used to initialize the structure, and

<[exp][,...]>

specifies a (possibly empty) list of field-initialization or optional field-override values. The angle brackets identify the record fields, while commas are used to separate the fields. The fields default to the order they were defined in the template. This is handled in exactly the same manner as described for structure fields. Just make sure the expression values will fit into the specified field bit patterns.

Initializing a large number of identical records from a single template also follows the method described for multiple structures. The initialization command takes the form:

[name] [record-name] [exp] DUP (<[exp][,...]>)

where the DUP clause creates as many copies of the template as specified. Again, parentheses enclose the field expression being duplicated.

```
TITLE UNIT 5 -- PROGRAM 2 -- DEFINING RECORDS
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
        ORG     100H
START:
;
RECORD_NAME     RECORD  FIELD_1:5=25,FIELD_2:4=7,FIELD_3:5=10,FIELD_4:2
;
TEMPLATE1       RECORD_NAME     <,,,3>
DATA_WORD       DW              0CBABH
TEMPLATE2       RECORD_NAME     3 DUP (<31,0,25>)
;
COM_PROG  ENDS
        END     START
```

Figure 5-8
Initializing a record.

Figure 5-8 is a simple program that contains a record template and two record initialization statements. Program code was left out, since our primary concern here is the record. The record template contains four fields. The first three contain defined values; the fourth is undefined. When the program is assembled, two records are created. The first is called TEMPLATE1. Its first three fields are a duplicate of the template RECORD_NAME, the fourth is initialized with the value three. When the record TEMPLATE1 is created, you can expect the word of data to contain the value 0CBABH. This is determined by combining the individual field bit values as shown in Figure 5-9.

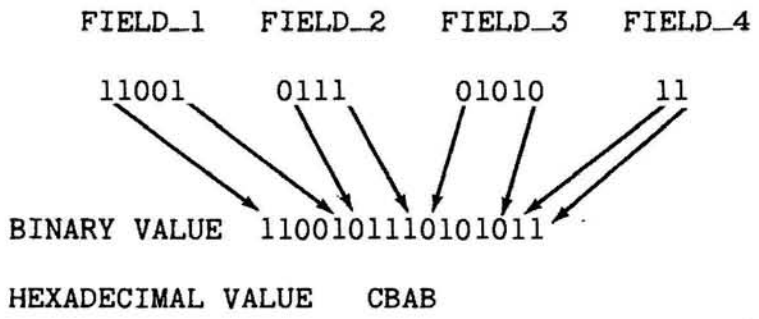


Figure 5-9
Determining the value of record TEMPLATE1.

The second record is called TEMPLATE2. It contains a DUP clause. Therefore, in this example, three identical records will be created by the assembler. Notice that the first three fields are overridden with new values; the fourth field is left out of the expression, and by default is not changed. This record should contain the value 0F864H when it is created.

```
 The Microsoft MACRO Assembler              03-16-84     PAGE    1-1
UNIT 5 -- PROGRAM 2 -- DEFINING RECORDS


1                                       TITLE UNIT 5 -- PROGRAM 2 -- DEFINING R
                                    ECORDS
2       0000                        COM_PROG  SEGMENT
3                                           ASSUME CS:COM_PROG,DS:COM_PROG,
                                    SS:COM_PROG
4       0100                                ORG     100H
5       0100                        START:
6                                   ;
7                                   RECORD_NAME     RECORD  FIELD_1:5=25,FI
                                    ELD_2:4=7,FIELD_3:5=10,FIELD_4:2
8                                   ;
9       0100  AB CB                 TEMPLATE1       RECORD_NAME     <,,,3>
10      0102  CBAB                  DATA_WORD       DW              0CBABH
11      0104     03 [               TEMPLATE2       RECORD_NAME     3 DUP (
                                    <31,0,25>)
12                 64 F8
13                       ]
14
15                                  ;
16      010A                        COM_PROG  ENDS
17                                          END     START
```

Figure 5-10
Assembler source listing of the record program.

Figure 5-10 shows the first part of the assembler source listing of our simple record program. Line 7 is the record template. Unlike the structure template, the record template has no offset address reference, and the field contents are not identified in the data column of the listing.

Record TEMPLATE1 contains the value 0CBABH. Notice that the value is listed in the data column just as it will be loaded into memory, low-byte first. We included a DW directive with the same value to show you that the assembler listing for a word-sized record is not presented in the same manner as a define word directive. This is to remind you that you should not interpret the listed data for the two word values in the same manner. The record named TEMPLATE2 is duplicated three times. Again, the assembler listing uses the standard duplication format. In this example the dup count, 03, precedes the duplicated data enclosed by square brackets. The value of the record is 0F864H.

Accessing Record Data

Accessing a record is no different than accessing any other named variable in a program. Data can be moved into a record from any 8- or 16-bit general register as long as the register and record sizes (byte or word) match. By the same token, record data can be moved into any 8- or 16-bit general register. However, we recommend that you do not use the CL or CX registers in this operation. You'll see why shortly.

If record data is treated no differently than any other variable, why bother creating the record in the first place? Records let you pack a lot of information into a small area. Many microcomputer operations require 1- or 2-bit codes to control the process. Rather than waste memory space and add to the complexity of the program with bytes of data containing these simple codes, MACRO-86 gives you the record. With the record, many codes can be stored within a byte or word of data. Accessing that code is the trick!

Suppose you wanted to access the code in record TEMPLATE1, field FIELD_3 of our simple record program. The first step is to move the contents of the record into a register. We'll use the instruction:

```
MOV    DX,TEMPLATE1
```

The DX register now contains the value 0CBABH. In binary, the individual fields contain the value:

```
11001 0111 01010 11
```

The next step is to mask the unwanted fields; that is, zero all the field bits except those in FIELD_3. Normally, this is accomplished by ANDing the register contents with a value that contains zeros in all of the mask-bit positions, and ones in all of the bit positions that must be preserved. Now you could calculate the value each time you wished to read a record field, or you could let MACRO-86 do the work for you.

When MACRO-86 assembles a program that contains a record, it uses the template to determine the location of every field in that record. With that data, the assembler sets-up a table that shows the bit-count from the least significant bit of the record to the beginning of each field. The table also contains the value that can be used to mask any field in the record. Now, when the assembler encounters an instruction that contains the operator MASK followed by a field name, the assembler substitutes the appropriate mask value from the table.

Therefore, to mask the unwanted fields from the record in the DX register, you would use the instruction:

```
AND    DX,MASK FIELD_3
```

The mnemonic AND performs the same operation as the assembler operator AND, only it does it during program run-time rather than when the program is assembled. This is important, since we are dealing with a variable that can change while the program is running. Thus, the contents of register DX are ANDed with the mask for FIELD_3, and the result is stored in DX. The mask for FIELD_3 is 007CH. In binary, the bit field arrangement is:

```
00000 0000 11111 00
```

After the AND operation, the DX register contains the value 0028H. Arranging the binary value by field, the register contains:

```
00000 0000 01010 00
```

The last step in the operation is to shift the contents of the field into the low-order bits of the register. Here you need to know the offset of the field into the record. Recall that the field offset is one of the items calculated by the assembler and stored in its record table. You can retrieve that value with the instruction:

```
MOV    CL,FIELD_3
```

The assembler knows that when it is asked to move a **field name** into the CL register, it is supposed to move the **field offset value** into the CL register. That's why we said earlier that you shouldn't use the CL or CX register to handle record data. Since the field offset value is the number of bits the field is shifted left of the least significant bit in the record, we should be able to use the value as a **shift count**.

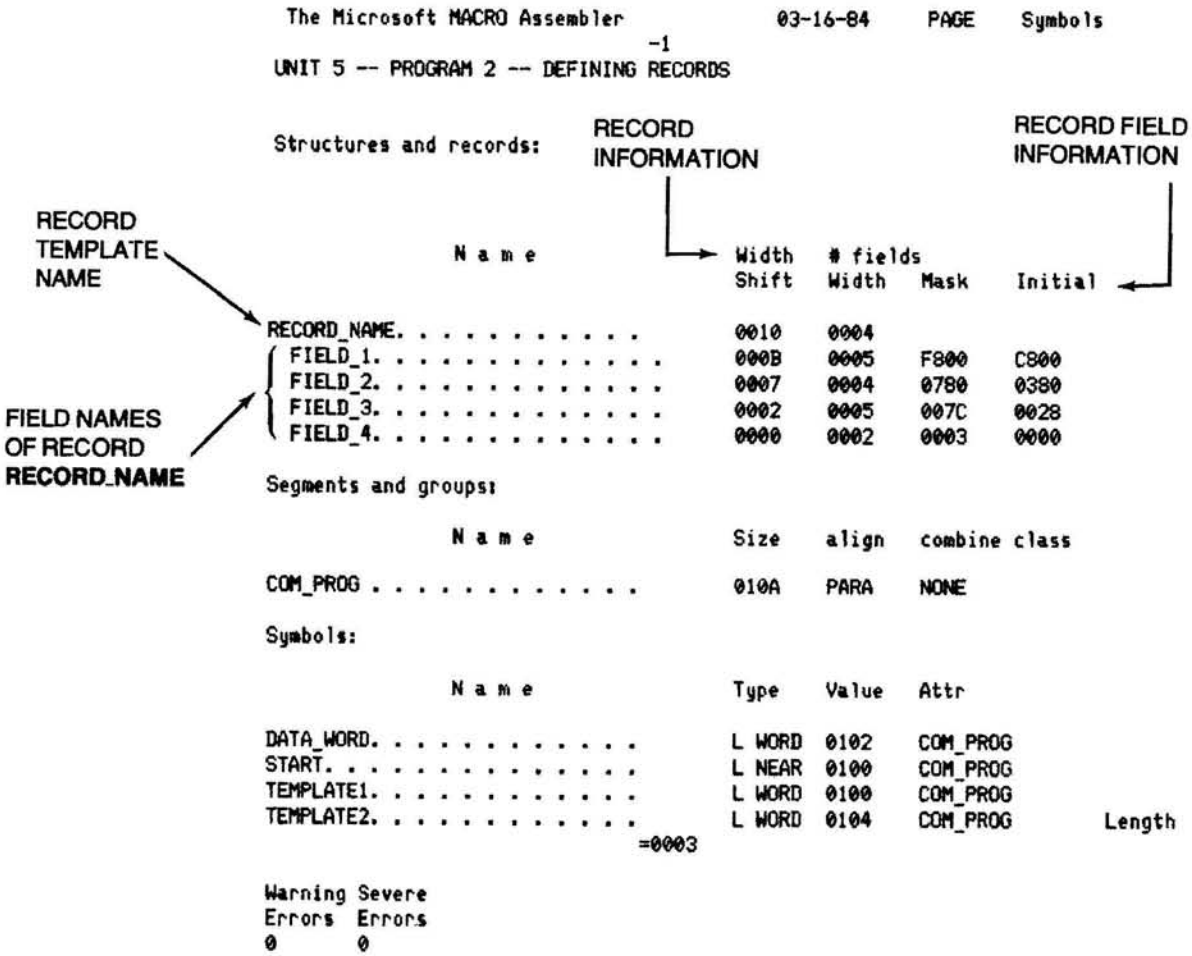
(Caution: IBM MACRO-86 version 1.00 will cause an assembly phase error if you use the "MOV CL,[field-name]" instruction.)

Naturally, you can't use the assembler operator SHR (shift right), since the shift operation must be performed during program run-time. There is, however, an instruction mnemonic called Shift Logical Right (SHR). It shifts the contents of the indicated register the number of bits right specified by the CL register. As the low-order bits are shifted out of the register, they are lost. At the same time, the empty high-order bits are filled with zeros. Therefore, to shift the contents of field FIELD_3 into the low-order bits of the DX register, use the instruction:

```
SHR    DX,CL
```

The DX register ends-up with the value 000AH — the original contents of field FIELD_3.

We made a number of references to a record table in the previous discussion. While the assembler has direct access to the table during the assembly process, you can also see the table in the assembler source listing. Figure 5-11 is the last part of the assembler listing for the record program. Each **record template** is identified in the "Structures and Records" table. Following each record template name is the name of each field in that template. The field names are indented to make them easier to locate.




```
 The Microsoft MACRO Assembler          03-16-84     PAGE    Symbols
                                        -1
UNIT 5 -- PROGRAM 2 -- DEFINING RECORDS

Structures and records:

                    N a m e                 Width   # fields
                                            Shift   Width   Mask    Initial
RECORD_NAME. . . . . . . . . .              0010    0004
  FIELD_1. . . . . . . . . . .              000B    0005    F800    C800
  FIELD_2. . . . . . . . . . .              0007    0004    0780    0380
  FIELD_3. . . . . . . . . . .              0002    0005    007C    0028
  FIELD_4. . . . . . . . . . .              0000    0002    0003    0000

Segments and groups:

                    N a m e                 Size    align   combine class

COM_PROG . . . . . . . . . . .              010A    PARA    NONE

Symbols:

                    N a m e                 Type    Value   Attr

DATA_WORD. . . . . . . . . . .              L WORD  0102    COM_PROG
START. . . . . . . . . . . . .              L NEAR  0100    COM_PROG
TEMPLATE1. . . . . . . . . . .              L WORD  0100    COM_PROG
TEMPLATE2. . . . . . . . . . .              L WORD  0104    COM_PROG        Length
                                        =0003

Warning Severe
Errors  Errors
0       0
```

Figure 5-11
Last portion of the assembler source listing of the record program.

The two values following the record template name are the **width**, or number of bits used by the template, and the number of **fields** in the template. After each field name are four columns of values that are used to identify that field. The first is the **shift** value. This is the offset of each field into the record. Notice the FIELD_4 has no shift value — it is the first field in the record. The second column contains the **width** value. This is the number of bits in the field. FIELD_3 contains five data bits. The third column contains the field **mask** value. Recall that the mask is a collection of ones and zeros that can be ANDed with the record to isolate a particular field. The last column contains the **initial** contents of the field. Notice that the value is presented as a bit pattern rather than an absolute value. Recall that field FIELD_3 contains the initial value 10, or 0AH. The bit pattern in the "Initial" column for field FIELD_3 is:

0000000000101000

Separate the pattern into fields:

00000 0000 01010 00

and you can see that the initial value for field FIELD_3 is indeed 10, or 0AH. Because field FIELD_4 contained no initial value in the record template, its "Initial" column value is zero.

The actual program records are identified in the "Symbols" table. Reading from left to right, the record TEMPLATE2 is a word-sized value, it begins at address offset 0104H, it is located in the segment COM_PROG, and it is composed of three records (Length = 0003). The record TEMPLATE1 is also a word-sized value, it begins at address offset 0100H, and it is also located in the segment COM_PROG. Since there is no "Length" specified, you can assume there is a single record associated with the name TEMPLATE1.

This concludes the discussion on records. Remember, the record template is used to establish the basic bit pattern for the records you will use in the program. All of the fields can remain undefined, or they can contain specific data. When you initialize a record — allocate memory space to that record — you can leave the fields unchanged, or you can modify their contents. If you modify a field, make sure the value will fit within the field.

Self-Review Questions

18. A _____ is a collection of data bits that are arranged in byte or word groups.

19. The small groups of data bits in a record are called _____.

20. The assembler directive _____ identifies the template for a program record.

Refer to Figure 5-12 for the following questions.

21. The record template is named _____.

22. Record RECORD1 contains the value _____.

23. Record RECORD3 contains the value _____.

24. The mask for field F4 is the value _____.

25. The contents of the DX register after the SHR instruction is executed is _____.

```
TITLE UNIT 5 -- PROGRAM 3 -- RECORD REVIEW
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
        ORG     100H
START:
        MOV     DX,RECORD2
        AND     DX,MASK F4
        MOV     CL,F4
        SHR     DX,CL
;
TEMPLATE        RECORD  F1:3=7,F2:4,F3:5=25,F4:2
;
RECORD1 TEMPLATE        <>
RECORD2 TEMPLATE        <,0FH,,3>
RECORD3 TEMPLATE        <,,,>
;
COM_PROG  ENDS
        END     START
```

Figure 5-12
Figure for questions 21 through 25.

REGISTER INDIRECT ADDRESSING

Earlier in the course we indicated that the 8088 MPU had three specific addressing modes: immediate addressing, register addressing, and direct memory addressing. With immediate addressing, the constant data is contained within the instruction operand. In register addressing, the source and destination operands are found within the MPU. With direct memory addressing, the effective, or offset, address of the source or destination is calculated from a target label or name in the source code. This section introduces a variation of these addressing schemes called register indirect addressing.

Register indirect addressing is a form of memory addressing that uses an MPU register to hold the address of the target. Because an MPU register holds the offset address value, the program can be more dynamic in the way it accesses memory. To begin this description, let's look at how the instruction is constructed.

Instruction Format

Recall from our discussion of the 8088 MPU machine language format that an instruction used for memory addressing generally contains an **opcode**, an address **mode** code, an **R/M** (register/memory) code, and an 8- or 16-bit **displacement**. This format is shown in Figure 5-14. The displacement serves as the offset or **effective address** (EA) for the memory location in the direct memory addressing mode. This displacement can be either an 8- or 16-bit value. In the indirect register addressing mode, there is no specific displacement to be calculated by the assembler.

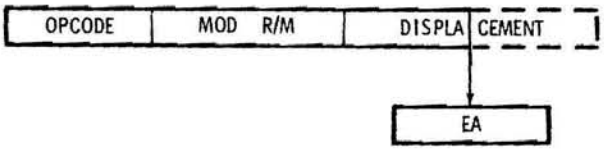


Figure 5-14
Direct memory addressing format.

Figure 5-15 shows the machine language format for the register indirect addressing instruction. The displacement value, or EA, is now located in one of four Base or Index registers within the MPU. These include the Base (BX) register, the Base Pointer (BP) register, the Source Index (SI) register, and the Destination Index (DI) register. Recall that the Base register can normally be used as a 16-bit or two 8-bit registers. However, when it is used to hold an address, it can only be used as a 16-bit register. Thus, the addressing form will always be "BX" and never "BL" or "BH." Naturally, there is no chance of error with the other three registers, since they can never be subdivided into 8-bit register form.

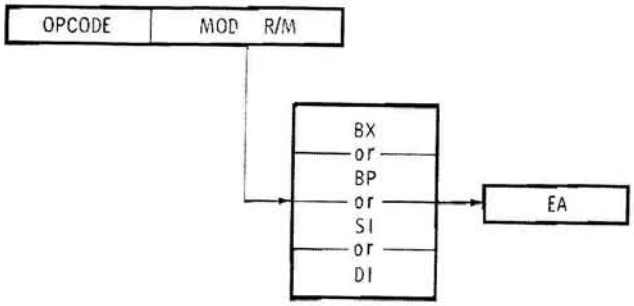


Figure 5-15
Register indirect addressing format.

Break the instruction down into its basic parts as shown in the figure and you find that the OPCODE specifies the operation, the MOD defines the addressing mode, and R/M indicates the register that contains the address displacement or EA. From a programmers point of view, a typical instruction using register indirect addressing would look something like this:

```
MOV    [BX],AX
```

This move instruction specifies that the source of the data is the Accumulator and the effective address of the destination is located in the Base register. The brackets around BX are an **assembler directive**. They indicate that this register will contain an **address displacement** rather than the usual numeric value. When executed, this address displacement will be used as the offset in the calculation of the physical address.

Using The Register Indirect Addressing Instruction

The main advantage of using register indirect addressing is that the address is treated as a **variable**. Earlier, when you used direct memory addressing, you were limited to a **fixed**, or constant, address displacement. Now it's a simple matter to move some value into one of the four Base or Index registers and come up with a new memory address. As an example, let's examine a simple program that uses register indirect addressing to move a string (series of bytes or words) of data from one area of memory to another. The program is listed in Figure 5-16. To keep the program as simple as possible, assume that a block of data is stored in memory at the location identified by the name DATABUFFER.

```
TITLE UNIT 5 -- PROGRAM 4 -- REGISTER INDIRECT ADDRESSING
;
COM_PROG  SEGMENT
        ASSUME  CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  LEA     BX,DATA_BUFFER  ;Get Effective Address of data
        LEA     BP,DATA_STORE   ;Get Effective Address of
                                ;data storage area
        MOV     CX,50           ;Set data size count
NEXT_WORD:
        MOV     AX,[BX]         ;Get word of data
        CMP     AX,0            ;Is data zero?
        JZ      STOP            ;Yes, end program
        MOV     [BP],AX         ;No, save data
        ADD     BX,2            ;Point to next word of data
        ADD     BP,2            ;Point to next storage area
        LOOP    NEXT_WORD       ;Repeat maximum 50 times
STOP:   INT     3               ;Halt the program and
                                ;return to the Debugger
DATA_BUFFER     DW  50 DUP (?)  ;Data source area
DATA_STORE      DW  50 DUP (?)  ;Data storage area
;
COM_PROG ENDS
        END     START
```

Figure 5-16
Program illustrating register indirect addressing.

You will find the term **buffer** often associated with computers. As part of an electrical circuit, a buffer is a form of interface between two circuits or systems. When associated with data handling, a buffer is one or more memory or register locations that are used to temporarily store information. We will always refer to buffer in terms of data storage.

The program begins with a new instruction, **LEA** (Load Effective Address). This instruction loads the effective, or offset, address of the target operand into the specified register. In that capacity, LEA is identical in function to the assembler operator OFFSET. However, as you will find later, the LEA instruction has a few other features that make it a valuable programming tool. Thus, the first instruction moves the effective address of the name DATA_BUFFER into the BX register. The second instruction moves the effective address of the name DATA_STORE into the BP register. These two registers will serve as "pointers" to the source and destination memory buffers. The third instruction establishes a loop count that determines the number of data-words that will be transferred from one buffer to the other. In this case, 50 words will be transferred.

With all the registers prepared, the transfer loop can begin by moving the first word from the source buffer into the AX register. The instruction uses the register indirect addressing mode. Thus, the word of data found at the effective address pointed to by the BX register is moved into AX. The next instruction compares the data to zero. When zero is encountered, the program is supposed to stop; hence the next instruction, jump on zero. As soon as the data being read into the Accumulator becomes zero, the CMP instruction will set the Zero flag and the MPU will jump to the target address identified by the label STOP.

As long as the data is not zero, the jump instruction is ignored and the MPU falls through to the next instruction. This instruction also uses the register indirect addressing mode to move the "data" to its storage area in memory.

Up to this point, the program has retrieved one word of data from the source buffer and saved it in the storage buffer. However, the EA in the Base and Base Pointer registers still address the **first** data word. To access the next word in the source buffer, the Base register is incremented by two. Then, to access the next storage buffer word, the Base Pointer register is incremented by two. Both registers are incremented by two rather than one because the program is handling word values, not byte values. This completes the first data acquisition cycle. The following LOOP instruction sends the program back to the point labeled NEXT_WORD and the cycle repeats.

Recall that the LOOP instruction first decrements the Count register and tests for zero before looping to the target address. Should the Count register reach zero, the program will fall through to the next instruction, INT 3. Thus, there are two possible conditions that can cause the program to stop; either a data word contained the value zero, or the program ran out of storage space in its assigned buffer.

The last part of the program is quite familiar by now. The INT 3 instruction halts the program and returns the MPU to the debugger program. A function 0, interrupt 21H instruction would have worked just as well, sending the MPU back to the disk operating system. The next two assembler directives assign space to the source buffer and the destination buffer.

Now let's look at how the addressing capabilities of register indirect addressing can be expanded.

Base/Index Addressing

You have learned that the Base and Index registers can be used to indirectly address data. The displacement value in the register serves as the offset, or effective, address for a memory read or write. The logical extension to this form of addressing is register indirect addressing with an immediate displacement offset value. For example, the instruction

```
MOV    AX,[BX]+5
```

tells the MPU to add the immediate value five to the displacement in the Base register to determine the effective address of the source operand. Register indirect addressing with an immediate displacement offset value is called **Based addressing, Indexed addressing,** or **Based Index addressing,** depending on the register(s) used to hold the primary displacement value. Because of the unique characteristics of these indirect forms of addressing, they will be treated as unique forms of register indirect addressing.

BASED ADDRESSING

In based addressing, the effective address is determined by adding the displacement value in either the **Base** or **Base Pointer** register to the immediate displacement value in the instruction operand. This is shown in Figure 5-17. Notice that the instruction displacement operand can be either a 1- or 2-byte value. The following are typical examples of the addressing mode:

```
MOV    AX,[BX]+5
SUB    [BP]+NUMBER,DX
ADD    CX,[BP]-0AH
```

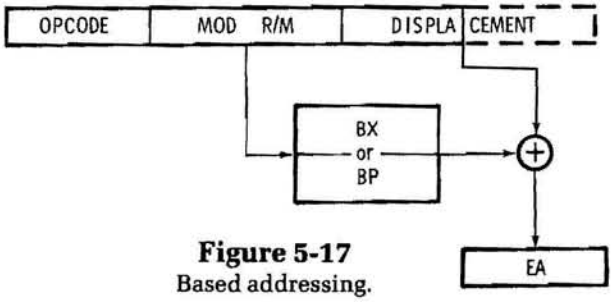


Figure 5-17
Based addressing.

The first instruction determines the EA (effective address) by adding the contents of the Base register to the single-byte displacement five, and then moves the data at that address into the Accumulator. The second instruction can be interpreted in two ways: NUMBER equates to a constant or NUMBER is a variable name. In the first case, the 1- or 2-byte constant is added to the displacement in the Base Pointer register to produce the destination EA. In the second case, the offset address of the variable name is added to the displacement in the Base Pointer register to produce the destination EA. The last instruction gives a new wrinkle to the addressing process. Instead of adding an immediate displacement value to the register displacement, the immediate displacement value is subtracted. Thus, when the EA is determined for this instruction, it will be lower by ten than the original displacement in the Base Pointer register.

Based addressing also provides a straightforward way to address a specific field in an array of data structures. A typical instruction is:

```
MOV    AX,[BX].FIELD_1
```

where the Base register is substituted for the structure name. The period, followed by the field name, tells the assembler that the instruction is addressing a field within a structure, and the offset address of the structure is in the Base register. Taking this idea one step further, it's now a simple matter to update the contents of the Base register to point to any structure during program run-time. For example, suppose you wanted to compare the third field of every structure in a program to a value, and then save the address of every field that contained a different value. Figure 5- 18 is a listing of that program.

```
TITLE UNIT 5 -- PROGRAM 5 -- BASED ADDRESSING
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
COUNT   EQU     100             ;Number of structures
LOOK    EQU     'M'             ;Character being compared
;
        ORG     100H
START:  LEA     BX,PERSON       ;Get address of first structure
        LEA     BP,ODD          ;Get address of storage area
        MOV     CX,COUNT        ;Load counter
        MOV     AL,LOOK         ;Get character to compare
TEST:   MOV     AH,[BX].STATUS  ;Get field character
        CMP     AL,AH           ;Do characters match?
        JZ      OKAY            ;Yes, field okay
        LEA     DX,[BX].STATUS  ;No, get address of bad field
        MOV     [BP],DX         ;Save address
        ADD     BP,2            ;Point to next storage address
OKAY:   ADD     BX,30           ;Point to next structure
        LOOP    TEST            ;Get next character until done
        INT     3               ;Return to debugger command
;
ODD     DW      COUNT DUP (0)   ;Bad character storage area
;
TABLE   STRUC
NAME    DB      'LARSEN LP      ' ;15 character string
DATE    DB      '03-13-84'      ;8 character string
STATUS  DB      'M'             ;Single character string
VAC     DB      15              ;Data byte
RATE    DW      10000           ;Data word
DEPT    DW      638             ;Data word
DIV     DB      18              ;Data byte
TABLE   ENDS
;
PERSON  TABLE   COUNT DUP (<>)  ;Array of structures
;
COM_PROG  ENDS
        END     START
```

Figure 5-18
Program to test the contents of a structure field.

The structure template in the program is called TABLE, and the third field in the template is called STATUS. The statement named PERSON initializes an array of 100 structures. Because the structure "count" is repeated in the program, an equate statement called COUNT is used to establish the value. This way we can always be sure that, if we change the count, every occurrence of that count is changed.

The define word statement named ODD initializes 100 words of memory with the value zero. This area is used to store the EA of every field that doesn't match the character in the equate statement LOOK.

The program begins by setting up four registers to handle the character test operation. First, the effective address of the first structure in the array of structures is loaded into the Base register. Then the Base Pointer register is loaded with the EA of defined word array ODD. Next, the program loop counter is loaded with the structure "count." Finally, the AL register is loaded with the character to be compared with each structure field.

The test begins by moving the contents of the first field into the AH register. Here, the Based addressing mode is used to access the field named STATUS. When the program is assembled, the term ".STATUS" is translated into a constant displacement. (This becomes an immediate value in the instruction.) Then, when the instruction is executed by the MPU, the contents of the Base register are added to that constant to produce the EA of the structure field.

The field value is compared with the ASCII code for the character "M". If there is a match, the Zero flag is set; otherwise, the flag is cleared. The jump if zero instruction then determines the flow of the program. A match causes the MPU to execute the jump and bypass the next three instructions. Let's assume there is no match and the jump is ignored.

The next instruction loads the EA of the field with the odd character into the DX register. This is another form of Based addressing. However, instead of moving the data pointed to by the EA, it's the EA that is moved into the register. Then the EA is stored in memory by the next instruction. Here, the register indirect form of addressing is used to move the contents of the DX register into the memory location pointed to by the contents of the Base Pointer register. Finally, the Base Pointer register is incremented by two to point to the next storage location in memory.

That completes the test loop. All that's left is to point the Base register to the next structure in the array by adding 30 (the number of bytes in the structure) to the register. Now the loop instruction can send the MPU back to the beginning of the loop. To keep everything tidy, an INT 3 instruction is used to end the program after the last loop.

You may have noticed that the program will never store the EA of an odd field character. That is because all of the structures contain the correct character. Normally, a test routine of this nature would be working on an array of structures that contain different data. To keep the program simple, we duplicated just one structure.

INDEXED ADDRESSING

Indexed addressing operates just like Based addressing, except that the register involved in the operation is either the Source Index or the Destination Index register. This is shown in Figure 5-19. Because of the similarities, you can think of Based and Indexed addressing as being the same — indirect register addressing with an immediate displacement value in the instruction. The names are different to distinguish between the registers used in the instruction. This is important when you are dealing with the Based Index form of addressing.

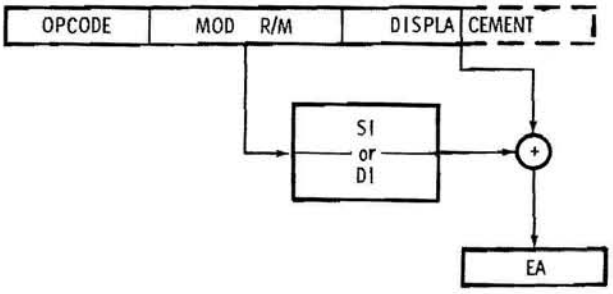


Figure 5-19
Indexed addressing.

BASED INDEX ADDRESSING

Creating different titles for these various forms of indirect register addressing probably seems hardly worth the trouble. However, it does help establish a uniqueness between register functions. This is more apparent when you consider Based Index addressing.

Figure 5-20 shows how the EA is determined from an instruction using Base Indexed addressing. The EA is derived from the sum of both a Base register displacement value and an Index register displacement value, as well as an immediate displacement value. Thus, you have a very flexible mode of addressing, since two of the address components (the registers) can be modified at program run-time. This provides a very convenient way for a program to address an **array** of data within a structure.

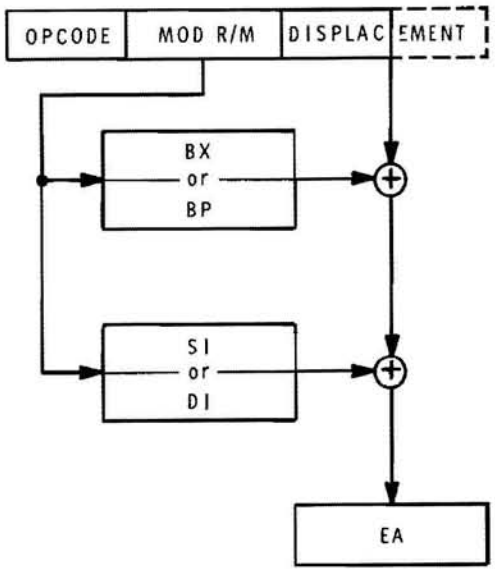


Figure 5-20
Based Index addressing.

```
PAY_FILE        STRUC
EMPLOYEE_ID     DB      0
DEPT            DB      0
VAC_DAYS        DB      0
RATE            DW      0
PAY             DB      12 DUP (0)
PAY_FILE        ENDS
```

Figure 5-21
Structure template containing an array of data.

Figure 5-21 is a structure template containing an array of data. Think of it as a simple company personnel file for handling pay data. Figure 5-22 is a graphic example of a memory structure created from the template in Figure 5-21. It shows two examples of accessing an element in the PAY field using Based Indexed addressing.

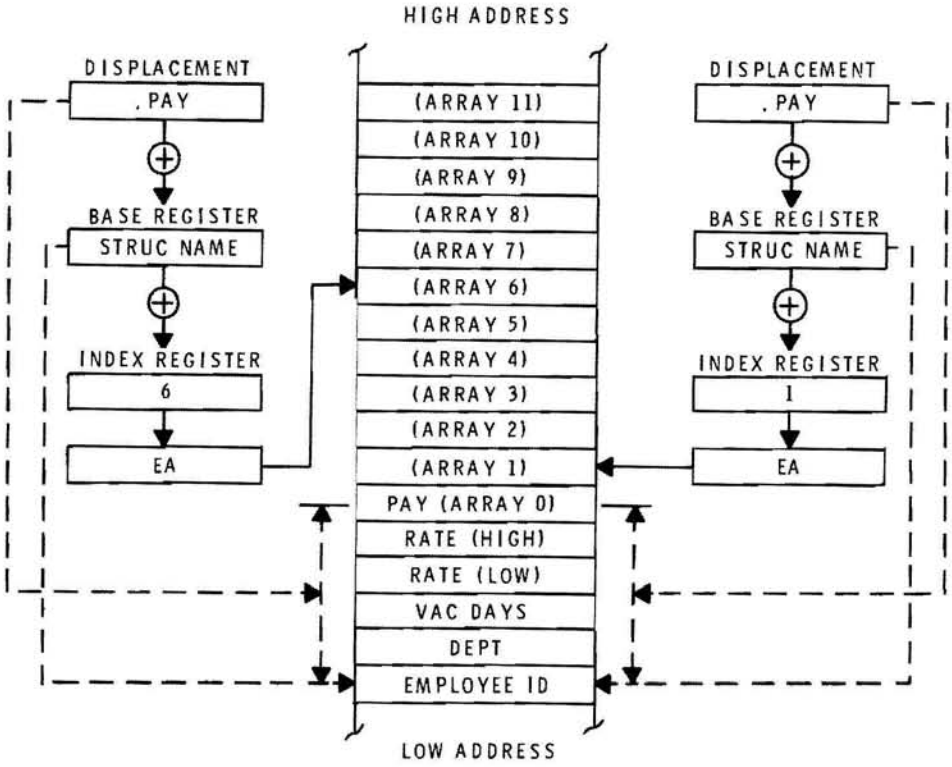


Figure 5-22
Using Based Index addressing to access a byte within an array within a structure.

On the left, the BASE REGISTER contains the offset address of the start or **base** of the structure. The DISPLACEMENT is the offset to the PAY field. The INDEX REGISTER contains the offset to the seventh element in the PAY array. If the data is being moved into the AL register, the instruction can be written:

```
MOV    AL,[BX][SI].PAY
```

where BX is the Base register, SI is the Index register, and ".PAY" is treated as an immediate value in the assembled instruction.

The right side of the figure illustrates the same operation; only here, the second element in the array is being accessed. The instruction is still written:

```
MOV    AL,[BX][SI].PAY
```

because this type of addressing scheme relies on the program to load the Base and Index registers with the appropriate displacement values.

Naturally, we could have substituted the constant 5 for ".PAY" and accomplished the same results. On the other hand, it's always a good idea to have the assembler do as much of the programming as possible, to reduce the chance for error.

Notice that each address displacement value was assigned a specific function in the instruction. While this is not an absolute requirement, it is a good idea from a programming point of view since it adds continuity between instructions using the Based Index addressing mode. It also makes the program easier to follow when you or someone else examines it at a later time. Therefore, we suggest you adopt this format:

1. The starting address of the structure should be in a Base register.

2. The offset to the array field in the structure should be the immediate displacement value of the instruction.

3. The offset to the desired array element should be in an Index register.

Stick to this format and you'll have a dynamic instruction that lets you access any structure in memory, and any array element within that structure.

All of the previous examples of register indirect addressing isolated each displacement value (except the immediate displacement) with square brackets. For example:

```
MOV    AH,[BX]
MOV    AH,[BP]+5
MOV    AH,[DI]-5
MOV    AH,[BX][SI]+5
```

While these are adequate, MACRO-86 does recognize variations of that format. For example:

```
MOV    AH,[BP+5]
MOV    AH,[SI][5]
MOV    AH,[BP+DI]+5
MOV    AH,[BX+SI+5]
MOV    AH,[BP+SI-5]
MOV    AH,[BX+DI][-5]
MOV    AH,[BX+SI].FIELD_1
```

In each case, the assembler recognizes that the register or constant enclosed by square brackets is a displacement in a register indirect addressing instruction. The only item that cannot be enclosed is the field name in a structure. We suggest that you choose one style and stick with it. This way, you'll reduce the chance for error.

While we're on the subject of errors, the instruction:

```
MOV    [BX][DI]+5,2930
```

will cause an assembly error. This is because the assembler has no way to determine the size (byte or word) of the memory location pointed to by the destination operand EA. When size must be determined, use the assembler operator PTR, as in:

```
MOV    WORD PTR [BX][DI]+5,2930
```

This tells the assembler the size of the memory location. You will have no problem with instructions that use a structure field name. The assembler knows the size of every field in a structure.

Self-Review Questions

26. For register indirect addressing, the EA is determined by the displacement value in the instruction operand. _____ True/False

27. Of the four registers used with register indirect addressing, two are base registers, while the other two are _____ registers.

28. The brackets in the instruction:

```
MOV   AX,[DI]
```

indicate that the value being moved into the AX register is in the DI register. _____ True/False

29. The brackets are an assembler directive. _____ True/False

30. The _____ instruction is used to load the effective address of a name into a register.

31. The instruction:

```
MOV   [BP],AL
```

is an example of _____ addressing.

32. The instruction:

```
MOV   [DI],AL
```

is an example of _____ addressing.

33. Based Index addressing is a register _____ form of memory addressing.

34. To maintain instruction continuity in Based Index addressing, you should place the starting, or offset, address of the structure in the _____ register.

35. The instruction:

```
MOV   BYTE PTR [BP+SI+5],AL
```

is the correct way to write a Based Index addressing mode instruction. _____ True/False

EXPERIMENT

Indirect Addressing

OBJECTIVES:

1. *Demonstrate the new assembler directive define doubleword.*
2. *Demonstrate the two new methods for arranging data — Structures and Records.*
3. *Demonstrate the various forms of indirect addressing.*

Introduction

Data manipulation is one of the most important functions of the microcomputer. Storing data within memory and later retrieving that data can be a difficult programming task, as you learned in the earlier experiments. The new methods for arranging and addressing data presented in this unit have made the job a little easier. Now you can access memory through a base or index register rather than rely on a variable name as the sole means for address identification. This experiment will show you how the MPU uses structures, records, and indirect addressing.

The first program you will write simply assigns data to memory. It uses the examples given in the unit to show you how the assembler structures the data without the clutter of instruction code.

Procedure

1. Call up your editor and enter the program listed in Figure 5-23. Assemble, link, and convert the file to a COM file. The program creates three records, four unique structures, and one array of ten structures.

```
TITLE EXPERIMENT 5 -- PROGRAM 1 -- ASSIGNING DATA TO MEMORY
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:
;
CONTROL RECORD  REG:5,STATE:2=3,FIELD:5,LEVEL:4=1
;
FORWARD CONTROL <>
REVERSE CONTROL <25,1,10,7>
STOP    CONTROL <20,,16>
;
TABLE   STRUC
NAME    DB      'LARSEN LP      ' ;15 character string
DATE    DB      '03-13-84'      ;8 character string
STATUS  DB      'M'             ;Single character string
VAC     DB      15              ;Data byte
RATE    DW      10000           ;Data word
DEPT    DW      638             ;Data word
DIV     DB      18              ;Data byte
TABLE   ENDS
;
LARSEN  TABLE   <>
ROBERTS TABLE   <'ROBERTS PA','03-19-84','F',5,5000,638,18>
JOHNSON TABLE   <'JOHNSON RJ','03-19-84','M',10,10000,638,18>
JON     TABLE   <'JON KC','03-19-84','F',10,7500,638,18>
EMPTY   TABLE   10 DUP (<>)     ;Array of duplicated structures
;
COM_PROG  ENDS
        END     START
```

Figure 5-23
Assigning data to memory.

2. Using the "TYPE" command, display the assembler listing. Stop the scrolling when the three records are visible. The first record, FORWARD, duplicates the record template; thus, the REG and FIELD fields are empty, or zero, and the STATE and LEVEL fields contain the predefined data. The second record, REVERSE, assigns new data to each field. The last record, STOP, assigns new data to the REG and FIELD fields, and retains the original data in the STATE and LEVEL fields.

Notice that the record template is assigned no offset address value. This is because the record template is not part of the program. Only the records created from the template are part of the program.

The offset address for the record STOP is ____H. What should be the offset for the structure LARSEN? ____H.

3. Scroll the assembler listing down to the structure LARSEN. The offset address is ____H.

   The structure LARSEN occupies the next consecutive offset address after the record STOP. Like the record template, the structure template is not part of the program.

4. Scroll the assembler listing until you can see the NAME field for both the structure JOHNSON and the structure JON. The NAME field for the structure JOHNSON contains one more character than the structure template. As a result, the original ASCII code is simply overwritten. That isn't the case for the NAME field in the structure JON. Here, the field contains three fewer characters. Rather than leave the extra three original characters unchanged, the assembler automatically converted them to 20H, the ASCII code for the "space" character. Thus, you see only the field characters specified in the structure statement.

5. Scroll the assembler listing until you can see the "Structures and Records" table on the "Symbols" page. Notice that the record template CONTROL contains 16 (10H) field bits (Width) and four fields (# fields). Each field is then listed, with the table showing the offset of each field into the record (Shift), the size of each field (Width), the value needed to isolate a field (Mask), and the initial contents of each field. The structure template TABLE is listed after the record. The template contains 30 (1EH) bytes of data (Width) and seven fields (# fields). The field names are then listed with the offset byte count into the structure (Shift).

6. Call up the debugger and load your program COM file. Then type "D100" and RETURN. Finally type "D180" and RETURN. At least the top half of your display should be similar to the one shown in Figure 5-24. Notice that the first six bytes of data in the figure, and your display, represent the three record values. The next 119 (77H) bytes represent the data in the first four structures. The remaining data represents the first field in the structure template duplicated almost nine times. As you can see, the assembler version (1.07) that was used to assemble this program contains the bug described earlier in the unit. If your display is the same, don't try to duplicate a structure. The assembled data will be useless. Versions 1.10 and later will assemble correctly.

```
DEBUG  version 1.08
>D100
0A09:0100  01 06 A7 CA 01 A7 4C 41-52 53 45 4E 20 4C 50 20   ..'J.'LARSEN LP
0A09:0110  20 20 20 20 20 30 33 2D-31 33 2D 38 34 4D 0F 10        03-13-84M..
0A09:0120  27 7E 02 12 52 4F 42 45-52 54 53 20 50 41 20 20   '~..ROBERTS PA
0A09:0130  20 20 20 30 33 2D 31 39-2D 38 34 46 05 88 13 7E      03-19-84F...~
0A09:0140  02 12 4A 4F 48 4E 53 4F-4E 20 52 4A 20 20 20 20   ..JOHNSON RJ
0A09:0150  20 30 33 2D 31 39 2D 38-34 4D 0A 10 27 7E 02 12    03-19-84M..'~..
0A09:0160  4A 4F 4E 20 4B 43 20 20-20 20 20 20 20 20 20 30   JON KC          0
0A09:0170  33 2D 31 39 2D 38 34 46-0A 4C 1D 7E 02 12 4C 41   3-19-84F.L.~..LA
>D180
0A09:0180  52 53 45 4E 20 4C 50 20-20 20 20 20 20 4C 41 52   RSEN LP      LAR
0A09:0190  53 45 4E 20 4C 50 20 20-20 20 20 20 4C 41 52 53   SEN LP      LARS
0A09:01A0  45 4E 20 4C 50 20 20 20-20 20 20 4C 41 52 53 45   EN LP      LARSE
0A09:01B0  4E 20 4C 50 20 20 20 20-20 20 4C 41 52 53 45 4E   N LP      LARSEN
0A09:01C0  20 4C 50 20 20 20 20 20-20 4C 41 52 53 45 4E 20    LP      LARSEN
0A09:01D0  4C 50 20 20 20 20 20 20-4C 41 52 53 45 4E 20 4C   LP      LARSEN L
0A09:01E0  50 20 20 20 20 20 20 4C-41 52 53 45 4E 20 4C 50   P      LARSEN LP
0A09:01F0  20 20 20 20 20 20 4C 41-52 53 45 4E 20 4C 50 20         LARSEN LP
>
```

Figure 5-24
Debugger display of data assigned to memory.

```
MOV     DX,REVERSE              ;Get record REVERSE
AND     DX,MASK FIELD           ;Mask the record field FIELD
MOV     CL,FIELD                ;Get FIELD shift count
SHR     DX,CL                   ;Shift FIELD to right end
MOV     AL,LARSEN.STATUS        ;Get STATUS field of sturcture
ADD     BYTE PTR JON.VAC,5      ;Add 5 to VAC field of structure
LEA     BX,ROBERTS              ;Get EA of structure ROBERTS
MOV     SI,[BX].RATE            ;Get RATE field of ROBERTS
INT     3                       ;Return to debugger
```

Figure 5-25
Code for the data program.

7. Exit the debugger. Then call up the editor and your program (from Figure 5-23). Add the instructions listed in Figure 5-25 directly after the START label in your program. If you have the IBM version 1.00 assembler, change the third instruction from:

```
MOV    CL,FIELD
```

to:

```
MOV    CL,4
```

to make sure the correct shift count is loaded. (All other assembler versions should load the shift count properly.) Finally, delete the structure table named EMPTY.

8. Assemble, link, and convert the program to a COM file. Examine the assembler listing for your program and record the offset address for the following names:

REVERSE	____H
LARSEN.STATUS	____H
ROBERTS	____H
ROBERTS.RATE	____H
JON.VAC	____H

Then record the "Shift" value ____H to the structure field RATE. Finally, record the "Mask" value ____H and the "Shift" value ____H for the record field FIELD.

9. Call up the debugger and load your program COM file. As you single-step through the program, compare the values you recorded in Step 8 with the values calculated by the debugger. Remember, the debugger never shows an operand name, just the assembled value for that name. The first instruction moves the value found at REVERSE into the DX register. Execute the instruction with the "T" (single step) command. The DX register contains the value ____H.

   Execute the next instruction. The contents of the DX register is ANDed with the "Mask" value for the field FIELD. The DX register should now contain the value for the field FIELD, with all other fields zero.

   Execute the next instruction. The value 04 is moved into the CL (Count) register. This is the shift count calculated by the assembler for field FIELD. If you are using assembler version 1.00, you had to calculate your own shift count. As an option, you could have used the value zero initially; assembled the program; examined the listing to see what shift value the assembler calculated; substituted the correct shift value in the program; and reassembled the program.

   NOTE: If you are using assembler version 1.00 and loaded the CL register with the immediate value 04, the next instruction is the Shift Right instruction. If, on the other hand, you allowed the assembler to calculate and load the CL register with the shift value, the next two instructions are NOP (no operation instructions). These are caused by the assembler and its 2-pass method of assembly. On the first pass, it identifies all the program symbols and guesses how much code will be required to execute each instruction. When it sees a field name, it thinks the name might be a variable, and thus reserves two bytes of code for the offset address. On the second pass, the assembler knows the name is a field name and no offset is necessary. Since the code space is already reserved, the assembler fills out the empty space with two NOP instructions (code 90H). If necessary, execute the two NOP instructions.

   Execute the next instruction. The contents of the DX register are shifted right the bit count in the CL register. The low-order bits of the DX register now contain the contents of the field FIELD (0AH).

   Execute the next instruction. The contents of the structure LARSEN, field STATUS is moved into the AL register. This is the ASCII code for the character "M".

Examine the memory byte at the offset address for structure JON, field VAC — type "D180" and RETURN. (You recorded the address in Step 8.) The value stored at that address is __H.

Execute the next instruction. Five is added to the structure JON, field VAC. Again, examine the memory byte at the offset address for the structure JON, field VAC. The value is __H. Comparing this value to the previous value, you can can see that five was added to the field VAC.

Execute the next instruction. The effective address (EA) for the structure ROBERTS is moved into the Base register.

Execute the next instruction. This instruction used Based addressing to move the contents of the field RATE, in the structure identified by the Base register, into the SI register. When the instruction was assembled, the "Shift" value for the field RATE was used to form the immediate displacement value.

Discussion

By now, you should have a fairly good idea how records and structures are created and their data accessed. One important item was brought out in the experiment. When the assembler encounters a label or name in an instruction, it has no idea how that symbol is going to be used until it finds the source of the application. This type of reference is called a **forward reference**, where the instruction is referencing forward or further on in the program. A forward reference can cause assembly errors, as was the case with assembler version 1.00. Later in the course, you will find where a forward reference can cause other problems.

There is a way to resolve forward reference problems; place the data in front of any reference. This way, the assembler knows the type and source of any reference before it sees the reference. Had we placed the records and structures in front of the code in the last program, the program would have assembled properly regardless of the assembler version.

Figure 5-26 shows how the data could be placed in front of the code in the previous program. Placing the data in this manner resolves the forward reference problem, but it creates another problem. Recall that the MPU assumes that program code begins at offset address 100H. Thus, it will decode and try to execute the first program byte after the ORG 100H assembler directive, not realizing that it's decoding data rather than code. Naturally, the program will not run properly.

To resolve that problem, we place an unconditional jump instruction at the beginning of the program. This forces the MPU to jump around the data to the code portion of the program. The next part of the experiment will give you an opportunity to try the new program arrangement.

```
TITLE EXPERIMENT 5 -- PROGRAM 3 -- ELIMINATING FOWARD REFERENCES
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  JMP     BEGIN
;
CONTROL RECORD  REG:5,STATE:2=3,FIELD:5,LEVEL:4=1
;
FORWARD CONTROL <>
REVERSE CONTROL <25,1,10,7>
STOP    CONTROL <20,,16>
;
TABLE   STRUC
NAME    DB      'LARSEN LP      ' ;15 character string
DATE    DB      '03-13-84'      ;8 character string
STATUS  DB      'M'             ;Single character string
VAC     DB      15              ;Data byte
RATE    DW      10000           ;Data word
DEPT    DW      638             ;Data word
DIV     DB      18              ;Data byte
TABLE   ENDS
;
LARSEN  TABLE   <>
ROBERTS TABLE   <'ROBERTS PA','03-19-84','F',5,5000,638,18>
JOHNSON TABLE   <'JOHNSON RJ','03-19-84','M',10,10000,638,18>
JON     TABLE   <'JON KC','03-19-84','F',10,7500,638,18>
;
BEGIN:  MOV     DX,REVERSE              ;Get record REVERSE
        AND     DX,MASK FIELD           ;Mask the record field FIELD
        MOV     CL,FIELD                ;Get FIELD shift count
        SHR     DX,CL                   ;Shift FIELD to right end
        MOV     AL,LARSEN.STATUS        ;Get STATUS field of sturcture
        ADD     BYTE PTR JON.VAC,5      ;Add 5 to VAC field of structure
        LEA     BX,ROBERTS              ;Get EA of structure ROBERTS
        MOV     SI,[BX].RATE            ;Get RATE field of ROBERTS
        INT     3                       ;Return to debugger
;
COM_PROG  ENDS
        END     START
```

Figure 5-26
Placing data in front of code to eliminate forward reference problems.

Procedure Continued

10. Exit the debugger. Then use your editor to move your program code after the record and structure data, as shown in Figure 5-26. Add the label BEGIN to the first line of code in that group of code. Now add the instruction:

```
JMP  BEGIN
```

after the label START, at the beginning of the program. If you are using assembler version 1.00, change the instruction:

```
MOV  CL,04
```

back to:

```
MOV  CL,FIELD
```

Since the assembler doesn't have to make a forward reference, the instruction will assemble properly. Assemble, link, and convert your program to a COM file.

11. Examine the assembler listing for your program. The first instruction is at offset address 0100H. The second instruction is at offset address ____H.

Notice that the code for the fourth instruction:

```
MOV  CL,FIELD
```

is only two bytes long — there are no 90H (NOP) code bytes. This is because the assembler recognized the name FIELD, and thus knew the exact number of code bytes to reserve for the instruction.

12. Call up the debugger and load your program COM file. Examine the first instruction in the program. Type "R" and RETURN. The first instruction is:

```
JMPS  0181
   or
JMP   0181
```

Your version of DEBUG will determine whether you see the mnemonic JMPS or JMP. The "S" at the end of the mnemonic indicates that the assembler treated the jump instruction as a "short" jump. That is, the target address is within 128 bytes of the instruction. The immediate value following the mnemonic is the offset address of the target. It should match the value you recorded in Step 11. When the jump instruction is executed, the MPU will load that offset value into the Instruction Pointer. Whether your debugger prints JMPS or JMP, you know it is a short jump, rather than a normal jump, because of the hex value in the first byte of the instruction machine code. A short jump has the value EB, while a normal jump has the value E9. A handy reference to the machine code (hex or binary) values for each instruction is provided in Appendix B.

Execute the jump instruction. The IP now contains the value 0181H, and the MPU is ready to execute the first instruction after the block of data. The rest of the program will execute just like the earlier program.

Discussion

Eliminating forward references will reduce the number of possible assembly errors when you begin writing complex programs. For that reason, we suggest that you make it a habit to always place your program data ahead of the code. Just don't forget to use a jump instruction to show the MPU where the code begins. We will structure all future programs with the data preceding the code.

The last part of this experiment deals with accessing arrays of data within structures using Based Index addressing. Let's begin with a program that stores data in four different structures. The program is shown in Figure 5-27. While this program is similar to the last program, we suggest that you enter the program from scratch.

```
TITLE EXPERIMENT 5 -- PROGRAM 4 -- WRITING TO STRUCTURES
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
STRUC_SIZE      EQU     54
;
        ORG     100H
START:  JMP     BEGIN
;
TABLE   STRUC
NAME    DB      'LARSEN LP      ';15 character string
DATE    DB      '03-13-84'      ;8 character string
STATUS  DB      'M'             ;Single character string
VAC     DB      15              ;Data byte
RATE    DW      10000           ;Data word
PAY     DW      12 DUP (0)      ;Set up pay table
DEPT    DW      638             ;Data word
DIV     DB      18              ;Data byte
TABLE   ENDS
;
LARSEN  TABLE   <>
ROBERTS TABLE   <'ROBERTS PA','03-19-84','F',5,5000>
JOHNSON TABLE   <'JOHNSON RJ','03-19-84','M',10,10000>
JON     TABLE   <'JON KC','03-19-84','F',10,7500>
;
BEGIN:  LEA     BX,LARSEN               ;Get base address
        SUB     SI,SI                   ;Zero index
        MOV     WORD PTR [BX+SI].PAY,150;Store pay 0
        ADD     SI,2                    ;Increment index
        MOV     WORD PTR [BX+SI].PAY,170;Store pay 1
        ADD     BX,STRUC_SIZE           ;Next base address
        MOV     WORD PTR [BX+SI].PAY,120;Store pay 1
        SUB     SI,2                    ;Decrement index
        MOV     WORD PTR [BX+SI].PAY,100;Store pay 0
        ADD     BX,STRUC_SIZE           ;Next base address
        MOV     WORD PTR [BX+SI].PAY,160;Store pay 0
        ADD     SI,2                    ;Increment index
        MOV     WORD PTR [BX+SI].PAY,180;Store pay 1
        ADD     BX,STRUC_SIZE           ;Next base address
        MOV     WORD PTR [BX+SI].PAY,130;Store pay 1
        SUB     SI,2                    ;Decrement index
        MOV     WORD PTR [BX+SI].PAY,110;Store pay 0
        INT     3                       ;Return to debugger
;
COM_PROG  ENDS
        END     START
```

Figure 5-27
Writing data to structures using Based Index addressing.

Procedure Continued

13. Exit the debugger. Then call up the editor and enter the program in Figure 5-27. The program is designed to create four structures, each with a field called PAY that contains an array of 12 words. The program code loads a value into the first two array locations in each structure field.

14. Assemble, link, and convert your program into a COM file. Then examine the assembler listing and record the offset address for the following structure fields:

    LARSEN.PAY ____H
    ROBERTS.PAY ____H
    JOHNSON.PAY ____H
    JON.PAY ____H

    You will use these addresses to locate data within your program in the next step.

15. Call up the debugger and load your program COM file. Record the value of the word of data stored at the following locations. Use the "D" (Dump) command to examine the locations.

    LARSEN.PAY ____H
    LARSEN.PAY + 2 ____H
    ROBERTS.PAY ____H
    ROBERTS.PAY + 2 ____H
    JOHNSON.PAY ____H
    JOHNSON.PAY + 2 ____H
    JON.PAY ____H
    JON.PAY + 2 ____H

    You should have found the value zero at each of these locations. Any other value would indicate a problem in your program's structure initialization statements.

16. Execute the first instruction of the program — type "T" and RETURN. The MPU is now looking at the first instruction after the program structure data, the load effective address instruction.

    Execute the instruction. This loads the effective, or offset, address of the first structure (LARSEN) in the program into the Base register. The program will use the contents of the Base register to point to the beginning of each structure in the program.

    Execute the next instruction. Here, the Source Index register is zeroed so that it can be used to point to the first word in the structure field array LARSEN.PAY.

    Execute the next instruction. This Based Indexed addressing instruction loads the value 150 (96H) into the first memory word location in the structure field array LARSEN.PAY.

    Execute the next instruction. The Source Index register is incremented by two to point to the next word in the array.

    Execute the next instruction. The value 170 (0AAH) is loaded into the second memory word location in the structure field array LARSEN.PAY.

    Execute the next instruction. The value 54 (36H) is added to the Base register. (This is the number of bytes in the structure template TABLE.) Therefore, the Base register now points to the beginning of the next structure (ROBERTS) in the program.

    The remaining program instructions repeat the data storage process for the other structures. The Source Index register is either incremented or decremented so that the first two memory word locations in each array receive a value. Single step through the remaining instructions, or use the "G" (Go) command to finish the data storage process.

17. Now let's see if the program worked. Type "D100" and RETURN. Then type "D180" and return. Examine the following offset address locations and record the word values at those locations.

LARSEN.PAY	011EH	____H
LARSEN.PAY+2	0120H	____H
ROBERTS.PAY	0154H	____H
ROBERTS.PAY+2	0156H	____H
JOHNSON.PAY	018AH	____H
JOHNSON.PAY+2	018CH	____H
JON.PAY	01C0H	____H
JON.PAY+2	01C2H	____H

These are the hexadecimal values for the data stored by the program.

Discussion

You are probably wondering why we had you write a program that repeated essentially the same operation eight times. First, it gave you an opportunity to see how the Based Index addressing mode of operation worked. Second, it provided a means for loading a collection of unique data into a number of structure field arrays. This data will be used in the next program.

Procedure Continued

18. Here's your chance to show what you have learned. Call up the editor and add the necessary code and defined data area to retrieve and store:

    A. The NAME from each structure.

    B. The **second** word of data from the field PAY associated with each structure name.

    Arrange the stored data so that each structure "name" is followed by its "pay." Do not change the earlier program code.

19. Assemble, link, and convert your program to a COM file. Then call up the debugger and evaluate your program. Remember, just because a program assembles properly does not necessarily mean it will run properly. If you have a problem with execution:

    A. Make sure each instruction performs the operation you expected.

    B. Make sure the address references in an instruction access the data you expected.

    These are the two most likely problems when a program assembles properly but runs improperly.

```
TITLE EXPERIMENT 5 -- PROGRAM 5 -- WRITING/READING STRUCTURES
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
STRUC_SIZE      EQU     54      ;Bytes in structure
STRUC_COUNT     EQU     4       ;Number of structures
NAME_SIZE       EQU     15      ;Bytes in structure NAME
PAY_COUNT       EQU     2       ;Offset to PAY word
;
        ORG     100H
START:  JMP     BEGIN           ;Jump around data area
;
TABLE   STRUC
NAME    DB      'LARSEN LP      ';15 character string
DATE    DB      '03-13-84'      ;8 character string
STATUS  DB      'M'             ;Single character string
VAC     DB      15              ;Data byte
RATE    DW      10000           ;Data word
PAY     DW      12 DUP (0)      ;Set up pay table
DEPT    DW      638             ;Data word
DIV     DB      18              ;Data byte
TABLE   ENDS
;
LARSEN  TABLE   <>
ROBERTS TABLE   <'ROBERTS PA','03-19-84','F',5,5000>
JOHNSON TABLE   <'JOHNSON RJ','03-19-84','M',10,10000>
JON     TABLE   <'JON KC','03-19-84','F',10,7500>
;
PAY_DATA DB     STRUC_COUNT*(NAME_SIZE+2) DUP (0)
;Storage area for structure NAMEs and their pay
;
BEGIN:  LEA     BX,LARSEN               ;Get base address
        SUB     SI,SI                   ;Zero index
        MOV     WORD PTR [BX+SI].PAY,150;Store pay 0
        ADD     SI,2                    ;Increment index
        MOV     WORD PTR [BX+SI].PAY,170;Store pay 1
        ADD     BX,STRUC_SIZE           ;Next base address
        MOV     WORD PTR [BX+SI].PAY,120;Store pay 1
        SUB     SI,2                    ;Decrement index
        MOV     WORD PTR [BX+SI].PAY,100;Store pay 0
        ADD     BX,STRUC_SIZE           ;Next base address
        MOV     WORD PTR [BX+SI].PAY,160;Store pay 0
        ADD     SI,2                    ;Increment index
        MOV     WORD PTR [BX+SI].PAY,180;Store pay 1
        ADD     BX,STRUC_SIZE           ;Next base address
        MOV     WORD PTR [BX+SI].PAY,130;Store pay 1
        SUB     SI,2                    ;Decrement index
        MOV     WORD PTR [BX+SI].PAY,110;Store pay 0
;
        LEA     BX,LARSEN       ;Get structure base address
        MOV     SI,PAY_COUNT    ;Point to pay word in array
        LEA     BP,PAY_DATA     ;Get storage area address
        MOV     DX,STRUC_COUNT  ;Data retrieval loop count
NEXT:   CALL    GET_NAME        ;Name retrieval subroutine
        CALL    PAY_SAVE        ;Pay retrieval subroutine
        DEC     DX              ;Count down loop
        JNZ     NEXT            ;Get more data?
        INT     3               ;No, return to debugger
;
```

Figure 5-28A
Program to write data to and read data from a structure field array.

Discussion

Figure 5-28 Part A and Part B show one method for retrieving and storing the specified data. Again, we must emphasize that there are many ways to accomplish a task in a program. Whether or not your program matches ours is not important; what is important is whether or not your program did the job. Let's look at our example.

To begin, we wanted an easy method for changing the number of structures accessed, the number of NAME field bytes accessed, and the individual array word accessed. Searching the program for each occurrence isn't easy, so we used equate statements for these values.

```
GET_NAME:
        MOV     CX,NAME_SIZE    ;Load name byte count
        SUB     DI,DI           ;Zero the index register
AGAIN:  MOV     AL,[BX+DI]      ;Get character in name
        MOV     [BP],AL         ;Save character
        INC     DI              ;Point to next character
        INC     BP              ;Point to next storage area
        LOOP    AGAIN           ;More characters?
        RET                     ;No, return from call
;
PAY_SAVE:
        MOV     AX,[BX+SI].PAY  ;Get pay word
        MOV     [BP],AL         ;Save low byte first
        INC     BP              ;Point to next storage area
        MOV     [BP],AH         ;Save high byte next
        INC     BP              ;Point to next storage area
        ADD     BX,STRUC_SIZE   ;Point to next structure
        RET                     ;Return from call
;
COM_PROG  ENDS
        END     START
```

Figure 5-28B
Continuation of the program to write data to and read data from a structure field array.

The next order of business was the data table. Since its size could change with a change in the number of structures or the number of bytes assigned to the NAME field, we created a compound expression for the DUP count. STRUC_COUNT defaults to the number of structures in the program. NAME_SIZE defaults to the number of bytes in the NAME field. Two is added to the NAME_SIZE because the PAY field array word will occupy two bytes of storage for each structure.

The code to retrieve and store the required data is located near the bottom of Figure 5-28A. First, the Base register is loaded with the EA for the first structure. Then the Source Index register is loaded with the offset to the field array word to be stored. Finally, the EA for the storage area is loaded into the Base Pointer register. This sets-up three of the four addressing registers used by the program.

Since the program will be repeating two basic move operations a number of times, we decided to use a subroutine for each move. To keep track of the structures that have been accessed, we used the DX register as a counter. For that reason, the next instruction loads the DX register with the number of structures being accessed by the program.

The next two instructions call the subroutine to get the structure name and the subroutine to get the PAY field word. Then the DX register is decremented. If it isn't zero, the two subroutine calls are repeated. When DX finally reaches zero, the jump is ignored and the interrupt is executed.

The last part of the program contains the two subroutines. The first one retrieves and stores the structure name, one byte at a time. To keep track of the bytes, the CX (Count) register is loaded with the number of bytes in the structure NAME field. The Destination Index register is used to point to the character being moved. To begin the routine, it is zeroed to point to the first byte. The next instruction uses Based Index addressing to retrieve the first character. Then the character is stored at the address pointed to by the Base Pointer register. Finally, the Destination Index and the Base Pointer registers are incremented to point to the next "name" character and the next storage location for that character. After the routine loops the NAME_SIZE count (15 in this case), the return from subroutine instruction sends the MPU back to the main program.

When the PAY_SAVE subroutine is called, the second word value from the PAY field array is stored immediately after the structure NAME. Using Based Index addressing, the word is moved into the AX register. Then the low-byte of the word is stored in memory. The Base Pointer register is incremented, and the high-byte of the word is stored. The Base Pointer register is incremented one more time in anticipation of the next storage operation in the GET_NAME subroutine. For that same reason, the value 54 (the size of the structure) is added to the Base register. Now the Base register is pointing to the beginning of the next structure. Finally, the return from subroutine instruction sends the MPU back to the main program.

After the call instructions have executed three more times, the interrupt instruction sends the MPU back to the debugger. The name of each structure and the related "pay" data are stored in the memory area identified by the name PAY_DATA.

This completes the Experiment for Unit 5. Proceed to the Unit 5 Examination.

UNIT 5 EXAMINATION

Refer to Figure 5-29 as you answer the following questions. You can assume that the program will assemble properly.

1. The program contains _____ structures.

2. The program contains _____ records.

3. Structure THREE, field MEDIUM contains the value ____H.

4. Structure TWO, field SMALL contains the value __H.

5. Using record AM for the data, the time is:

   __:__:__

6. The second instruction in the program loads the AL register with the value __H.

7. Structure TWO, field MEDIUM contains the value ____H after the program is executed.

8. The DX register contains the value ____H after the program is executed.

9. The BP register contains the value ____H after the program is executed.

```
TITLE EXAMINATION 5 -- PROGRAM 1 -- RECORDS, STRUCTURES, ETC.
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  JMP     BEGIN           ;Jump around data area
;
TEMP    STRUC
DATE    DB      '03/23/84'
LARGE   DD      765932H
MEDIUM  DW      ?
MANY    DW      65,44,89
SMALL   DB      1
TEMP    ENDS
TIME    RECORD  HOURS:4,MINUTES:6,SECONDS:6
ONE     TEMP    <>
TWO     TEMP    <,99999H,1111H>
THREE   TEMP    <,,5,,255>
AM      TIME    <8>
PM      TIME    <,,30>
;
BEGIN:  MOV     AL,ONE.SMALL
        LEA     BX,TWO
        ADD     [BX].MEDIUM,10
        MOV     DX,PM
        AND     DX,MASK HOURS
        MOV     CL,HOURS
        SHR     DX,CL
        MOV     DI,4
        MOV     BP,[BX+DI].MANY
        INT     3
;
COM_PROG  ENDS
        END     START
```

Figure 5-29
Figure for the Final Examination questions.

EXAMINATION ANSWERS

Refer to Figure 5-30 as you study the following answers.

1. The program contains **three** structures.

2. The program contains **two** records.

3. Structure THREE, field MEDIUM contains the value **0005H**.

4. Structure TWO, field SMALL contains the value **01H**.

5. Using record AM for the data, the time is:

   08:00:00

6. The second instruction in the program loads the AL register with the value **01H**.

7. Structure TWO, field MEDIUM contains the value **111BH** after the program is executed.

8. The DX register contains the value **0000H** after the program is executed.

9. The BP register contains the value **0059H** after the program is executed.

```
TITLE EXAMINATION 5 -- PROGRAM 1 -- RECORDS, STRUCTURES, ETC.
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  JMP     BEGIN           ;Jump around data area
;
TEMP    STRUC
DATE    DB      '03/23/84'
LARGE   DD      765932H
MEDIUM  DW      ?
MANY    DW      65,44,89
SMALL   DB      1
TEMP    ENDS
TIME    RECORD  HOURS:4,MINUTES:6,SECONDS:6
ONE     TEMP    <>
TWO     TEMP    <,99999H,1111H>
THREE   TEMP    <,,5,,255>
AM      TIME    <8>
PM      TIME    <,,30>
;
BEGIN:  MOV     AL,ONE.SMALL
        LEA     BX,TWO
        ADD     [BX].MEDIUM,10
        MOV     DX,PM
        AND     DX,MASK HOURS
        MOV     CL,HOURS
        SHR     DX,CL
        MOV     DI,4
        MOV     BP,[BX+DI].MANY
        INT     3
;
COM_PROG  ENDS
        END     START
```

Figure 5-30
Figure for the Final Examination answers.

SELF-REVIEW ANSWERS

1. The **define doubleword** assembler directive allows you to assign two words of data to memory.

2. An **array** is a group of two or more common data elements.

3. The **define ten-byte** assembler directive creates what could be considered a 10-byte array of data.

4. The **define quadword** assembler directive allows you to assign four words of data to memory.

5. The data width assumed by MACRO-86 for each of the define data assembler directives is as follows:

   DB **byte**
   DW **word**
   DD **word**
   DQ **word**
   DT **word**

6. The instruction that will move the contents of the AL register into the third byte of the array named DATA_BYTE is:

```
MOV    DATA_BYTE+2,AL
```

   Remember, the first byte is byte zero. Therefore, the third byte is in effect, byte two (offset two from the first byte).

7. The instruction that will move the contents of the BX register into the fourth word of the array named DATA_WORD is:

```
MOV    DATA_WORD+6,BX
```

   Again, the first word is found at offset zero. The next word is found at offset two, and so on. Therefore, to address the fourth word in the array, multiply the word location (4) by two, then subtract two from the product. This will give the operator offset value that must be added to the operand in the instruction.

8. Write the instruction that will move the first byte of data in memory, initialized by the assembler directive statement:

```
TEN_BYTE  DT     1234567890H
```

into the AL register.

```
MOV   AX,TEN_BYTE
```

Remember, when you access data initialized by the DT assembler directive, you must do so in word-sized moves. Since the first byte is in the lowest memory address, it will be moved into the low-byte (AL) side of the AX register. Had the question asked for the second byte, the operation would have required two steps. First, the first word of data is moved into general register BX, CX, or DX. Second, the contents of the high-byte of that general register is moved into the AL register.

9. A **structure** is a group of data bytes and/or words that are arranged in a specific manner.

10. The data elements of a structure are called **fields.**

11. The beginning of a structure template is identified by the directive statement **STRUC.**

12. The end of a structure template is identified by the directive statement **ENDS.**

Refer to Figure 5-7 for answers 13 through 17.

```
TEMP    STRUC
F1      DB      '03/05/84'
F2      DD      1290H
F3      DW      ?
F4      DB      24,33,85,12
F5      DW      5 DUP (?)
TEMP    ENDS
```

Figure 5-7
Figure for answers 13 through 17.

13. The structure template contains **five** fields.

14. The fields that can be overridden are:

    F1
    F2
    F3

    The other fields contain multiple elements, and thus cannot be overridden.

15. A structure initialization statement named APPLE that duplicates the template is:

```
APPLE   TEMP    <>
```

16. A structure initialization statement named PEAR that duplicates the template three times and changes the contents of the third field to 0ABCDH is:

```
PEAR    TEMP    3 DUP (<,,0ABCDH>)
```

17. The instructions that will move the ASCII code for the day from the structure PEAR into the AX register, assuming the first field contains the character string for the month, day, and year is:

```
MOV     AH,PEAR.F1+3    ;High byte of day
MOV     AL,PEAR.F1+4    ;Low byte of day
```

18. A **record** is a collection of data bits that are arranged in byte or word groups.

19. The small groups of data bits in a record are called **fields**.

20. The assembler directive **RECORD** identifies the template for a program record.

Refer to Figure 5-13 for answers 21 through 25.

```
TITLE UNIT 5 -- PROGRAM 3 -- RECORD REVIEW
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
        ORG     100H
START:
        MOV     DX,RECORD2
        AND     DX,MASK F4
        MOV     CL,F4
        SHR     DX,CL
;
TEMPLATE        RECORD  F1:3=7,F2:4,F3:5=25,F4:2
;
RECORD1 TEMPLATE        <>
RECORD2 TEMPLATE        <,0FH,,3>
RECORD3 TEMPLATE        <,,,>
;
COM_PROG  ENDS
        END     START
```

Figure 5-13
Figure for answers 21 through 25.

21. The record template is named **TEMPLATE**.

22. Record RECORD1 contains the value **3864H**.

23. Record RECORD3 contains the value **3864H**. The field separator commas, by themselves, have no effect on the contents of the record.

24. The mask for field F4 is the value **0003H**.

25. The contents of the DX register after the SHR instruction is executed is **0003H**.

26. **False**. For register indirect addressing, the EA is determined by the displacement value in the specified register.

27. Of the four registers used with register indirect addressing, two are base registers, while the other two are **index** registers.

28. **False**. The brackets in the instruction:

```
MOV    AX,[DI]
```

indicate that the value being moved into the AX register is found at the effective address pointed to by the DI register.

29. **True**. The brackets are an assembler directive that indicate that the register within the brackets contains the effective address of the source or destination operand.

30. The **LEA**, or **Load Effective Address**, instruction is used to load the effective address of a name into a register.

31. The instruction:

```
MOV    [BP],AL
```

is an example of **Based** addressing.

32. The instruction:

```
MOV    [DI],AL
```

is an example of **Indexed** addressing.

33. Based Index addressing is a register **indirect** form of memory addressing.

34. To maintain instruction continuity in Based Index addressing, you should place the offset address of the structure in the **Base** register.

35. **False**. The instruction:

```
MOV    BYTE PTR [BP+SI+5],AL
```

is not the correct way to write a Based Index addressing mode instruction. The assembler operator PTR should only be used when the assembler cannot determine the size of the memory location.


INSERT

Unit 6

EXPANDING THE INSTRUCTION SET

CONTENTS

INTRODUCTION

The 8088 MPU has about 155 basic instructions. Moreover, when all the different addressing modes and accessible registers are considered, there are nearly 1000 different opcodes to control the MPU.

These instructions can be broken down into several categories. They include data transfer, arithmetic, bit manipulation, strings, program transfer, processor control, and I/O. While we have already discussed many, this section will introduce most of the remaining instructions. The handful of string, processor control, and I/O instructions that are not discussed in this section will be covered in later units.

Appendix D of this course contains a detailed listing of every instruction, along with a few examples of its implementation. After reading this Unit, turn to Appendix D and look over the explanations given there. In the future, when you are in doubt as to exactly what a particular instruction does, look it up in Appendix D.

Use the "Unit Objectives" that follow to evaluate your progress. When you can successfully accomplish all of the objectives, you will have completed this Unit. You can use the "Unit Activity Guide" to keep a record of those sections that you have completed.

UNIT OBJECTIVES

When you complete this Unit, you will be able to:

1. Define the two new general-purpose data transfer instructions: XCHG and XLAT.

2. State the purpose of the two new flag transfer instructions: LAHF and SAHF.

3. Determine the signed binary number, unpacked decimal number, and packed decimal number from an unsigned binary number.

4. State how to use the arithmetic instructions: ADD, ADC, AAA, DAA, SUB, SBB, NEG, AAS, DAS, MUL, IMUL, AAM, DIV, IDIV, AAD, CBW, and CWD.

5. State how to use the bit manipulation instructions: NOT, AND, OR, XOR, TEST, SHL/SAL, SHR, SAR, ROL, RCL, RCR.

UNIT ACTIVITY GUIDE

	Completion Time
☐ Read the Section on "Data Transfer Instructions."	_____
☐ Complete Self-Review Questions 1-4.	_____
☐ Begin Reading the Section on "Arithmetic Instructions."	_____
☐ Complete Self-Review Questions 5-14.	_____
☐ Continue Reading the Section on "Arithmetic Instructions."	_____
☐ Complete Self-Review Questions 15-19.	_____
☐ Continue Reading the Section on "Arithmetic Instructions."	_____
☐ Complete Self-Review Questions 20-25.	_____
☐ Continue Reading the Section on "Arithmetic Instructions."	_____
☐ Complete Self-Review Questions 26-33.	_____

- ☐ Read the Section on "Bit Manipulation Instructions." _____
- ☐ Complete Self-Review Questions 34-41. _____
- ☐ Perform the Experiment. _____
- ☐ Complete the Unit 6 Examination. _____
- ☐ Check the Examination Answers. _____

DATA TRANSFER INSTRUCTIONS

Data transfer instructions fall into three categories: general purpose, address object, and flag transfer. They move single bytes or words between memory and the MPU, and within the MPU. Figure 6-1 lists the instructions. The figure is divided into three sections. The first, Mnemonic, lists the assembler mnemonic for each instruction. Where

MNEMONIC	SYNTAX	RESULT
	GENERAL PURPOSE	
MOV	reg,mem/reg	Move byte or word to register from memory or register
MOV	mem/reg,reg	Move byte or word to memory or register from register
MOV	mem/reg,numb	Move immediate byte or word data to memory or register
PUSH	mem/reg16	Push word into stack
POP	mem/reg16	Pop word off stack
XCHG	reg,mem/reg	Exchange byte or word between register, and memory or register
XCHG	mem/reg,reg	Exchange byte or word between memory or register, and register
XLAT	AL	Translate byte; BX plus AL offset equals AL
	ADDRESS OBJECT	
LEA	reg16,mem	Load effective address of memory location to 16-bit register.
	FLAG TRANSFER	
PUSHF		Push flags into stack
POPF		Pop flags from stack
LAHF		Load AH register from flags
SAHF		Store AH register in flags

Figure 6-1
Data transfer instructions.

the mnemonic is repeated, it serves as an indication that there is more than one way to structure the instruction operands. The next section, Syntax, shows the operand format for each instruction. As an example, the first MOV instruction uses a register for the destination operand and a register or memory location for the source operand. Syntax terminology for this and any similar figures in this unit is as follows:

reg = register — byte or word
reg16 = 16-bit register only
reg8 = 8-bit register only
mem = memory — byte or word
numb = immediate value
label = label — byte or word
lab8 = 8-bit label +127, −128 bytes
AL = contents of AL register
CL = contents of CL register
1 = immediate value one
symbol "," = separates operands
symbol "/" = literal "or"

The "Result" column of the figure gives a short summary of the action performed by the instruction. Now let's look at the data transfer instruction types, beginning with the "General Purpose" category.

General Purpose

Under the General Purpose category, there are five basic instructions: MOV, PUSH, POP, XCHG, and XLAT. You are by now very familiar with the first three. **XCHG** (exchange byte or word) swaps the contents of the source and destination operands. Thus, in the instruction:

```
XCHG AX,DI
```

the contents of the AX register are placed into the DI register, and the contents of the DI register are placed into the AX register. This also works for all of the memory addressing modes. The instruction:

```
XCHG [BX+SI+5],DX
```

places the contents of the memory EA into the DX register, and the contents of the DX register into the memory location specified by the EA.

The instruction **XLAT** (translate-table) replaces a byte in the AL register with a byte from a 256-byte, user-coded, translation table. Register BX is used to hold the offset address of the first byte in the table. The byte in the AL register is used as an index into the table. During execution, the AL register contents are replaced by the byte at the location in the table corresponding to the original contents of the AL register. The first byte in the table has an index of **zero**. As an example, assume that the AL register contains the value 05H, and the **sixth** element in the translation table contains 33H. After the XLAT instruction is executed, the AL register will contain the value 33H.

XLAT is useful for translating characters from one code to another, the classic example being ASCII to EBCDIC or vice versa. The term EBCDIC stands for Expanded Binary Coded Decimal Interchange Code. However, we will stick to ASCII for all of our programming.

Address Object

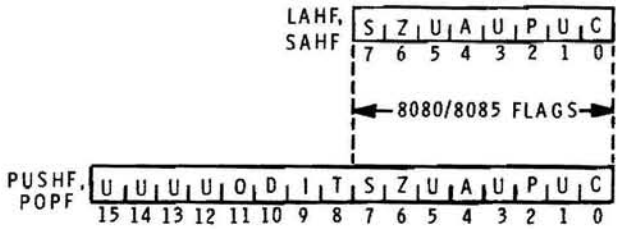
Under Address Object transfers, there are three instructions: LEA, LDS, and LES. The instruction LEA transfers the effective address of the source operand to the destination operand. The source operand must be a variable **name**, while the destination operand must be a 16-bit general register. Normally, you would use a Base or Index register as the destination.

The LDS and LES instructions are used with "string" operations. These will be described in Unit 8, along with the other string instructions.

Flag Transfer

The last data transfer instructions involve Flag Transfer. Recall that PUSHF and POPF are specific stack operation instructions that are used to store and retrieve the complete 16-bit Flag register. Two other Flag Transfer instructions allow you to operate on just the first eight bits (low-order byte) of the Flag register. These instructions are LAHF and SAHF. **LAHF** loads the AH register with the low-order byte of the Flag register. Because Flag register bits 1, 3, and 5 are considered to be undefined, they can assume either a logic 1 or a logic 0 in the AH register. The instruction **SAHF** reverses the operation and stores the contents of the AH register into the low-order byte of the Flag register. Since the contents of bits 1, 3, and 5 are considered undefined, they can be any value and not affect the Flag register.

To help maintain microprocessor family compatibility, Intel made the low-order byte of the 8088/8086 MPU Flag register identical to the Flag register in the less sophisticated 8080/8085 MPU. For that reason, both the LAHF and SAHF instructions are used primarily for converting 8080/8085 assembly language to run on an 8088/8086-based microcomputer. Figure 6-2 shows the Flag register bits affected by the four Flag Transfer instructions.



U= UNDEFINED; VALUE IS INDETERMINATE
O= OVERFLOW FLAG
D= DIRECTION FLAG
I= INTERRUPT ENABLE FLAG
T= TRAP FLAG
S= SIGN FLAG
Z= ZERO FLAG
A= AUXILIARY CARRY FLAG
P= PARITY FLAG
C= CARRY FLAG

Figure 6-2
Flag storage formats.

Self-Review Questions

1. The PUSHF instruction can be used to store a byte of data in the "stack." _____
   True/False

2. The XCHG instruction replaces a byte in the AL register with a byte from a 256-byte code table, whose address is contained in the BX register. _____
   True/False

3. If you wish to update SF, ZF, AF, and CF to known values, then you should use the flag transfer instruction _____.

4. The LEA instruction is used to move the effective address of a variable name into a _____ register.

NOTE: The Self-Review Answers are located at the end of this unit, beginning on Page 6-83.

ARITHMETIC INSTRUCTIONS

The arithmetic instructions cover the four basic math operations: addition, subtraction, multiplication, and division. Figures 6-3A, 6-3B, and 6-3C list the instructions. Each of these let you manipulate four types of numbers: unsigned binary, signed binary (integers), unsigned **packed** decimal, and unsigned **unpacked** decimal. Figure 6-4 gives an example of how these numbers are related, using a byte-sized format. Binary numbers may be 8 or 16 bits long. Decimal numbers are stored in bytes, two digits per byte for packed decimal and one digit per byte for unpacked decimal. The microprocessor always assumes that the operands specified in an arithmetic instruction contain data that represent valid numbers for the type of instruction being performed. Invalid data may produce unpredictable results.

MNEMONIC	SYNTAX	RESULT
ADDITION		
ADD	reg,mem/reg	Add byte or word memory or register to register
ADD	mem/reg,reg	Add byte or word register to memory or register
ADD	mem/reg,numb	Add immediate byte or word data to memory or register
ADC	reg,mem/reg	Add with carry byte or word memory or register to register
ADC	mem/reg,reg	Add with carry byte or word register to memory or register
ADC	mem/reg,numb	Add with carry immediate byte or word data to memory or register
INC	mem/reg	Add one to byte or word memory or register
AAA		ASCII adjust for addition; AL register
DAA		Decimal adjust for addition; AL register

Figure 6-3A
Arithmetic instructions, addition.

MNEMONIC	SYNTAX	RESULT
	SUBTRACTION	
SUB	reg,mem/reg	Subtract byte or word memory or register from register
SUB	mem/reg,reg	Subtract byte or word register from memory or register
SUB	mem/reg,numb	Subtract immediate byte or word data from memory or register
SBB	reg,mem/reg	Subtract with borrow byte or word memory or register from register
SBB	mem/reg,reg	Subtract with borrow byte or word register from memory or register
SBB	mem/reg,numb	Subtract with borrow immediate byte or word data from memory or register
DEC	mem/reg	Subtract one from byte or word memory or register
NEG	mem/reg	Compute 2's complement byte or word memory or register
CMP	reg,mem/reg	Compare byte or word memory or register with register
CMP	mem/reg,reg	Compare byte or word register with memory or register
CMP	mem/reg,numb	Compare immediate byte or word data with memory or register
AAS		ASCII adjust for subtraction; AL register
DAS		Decimal adjust for subtraction; AL register

Figure 6-3B
Arithmetic instructions continued, subtraction.

MNEMONIC	SYNTAX	RESULT
	MULTIPLICATION	
MUL	mem/reg	Multiply unsigned byte (AL) or word (AX) with memory or register; store results in AX (byte) or DX/AX (word)
IMUL	mem/reg	Integer (signed) multiply byte (AL) or word (AX) with memory or register; store results in AX (byte) or DX/AX (word)
AAM		ASCII adjust for multiply; AL register
	DIVISION	
DIV	mem/reg	Divide unsigned byte (AL, extension in AH) or word (AX, extension in DX) with memory or register; store **byte** results AL = quotient, AH = remainder; store **word** results AX = quotient, DX = remainder
IDIV	mem/reg	Integer (signed) divide byte (AL, extension in AH) or word (AX, extension in DX) with memory or register; store **byte** results AL = quotient, AH = remainder; store **word** results AX = quotient, DX = remainder
AAD		ASCII adjust for division, AL register
CBW		Convert byte in AL register to word in AX register using sign extension
CWD		Convert word in AX register to doubleword in DX/AX registers using sign extension

Figure 6-3C
Arithmetic instructions continued, multiplication and division.

HEX	BIT PATTERN	UNSIGNED BINARY	SIGNED BINARY	UNPACKED DECIMAL	PACKED DECIMAL
07	00000111	7	+7	7	7
89	10001001	137	−119	invalid	89
C5	11000101	197	−59	invalid	invalid

Figure 6-4
Arithmetic interpretation of 8-bit numbers.

Unsigned binary numbers use all of the data bits to determine the number's magnitude. The value range for an 8-bit number is 0 through 255, while the range for a 16-bit number is 0 through 65,535. Unsigned binary numbers can be used in addition, subtraction, multiplication, and division operations.

Signed binary numbers (integers) use the high-order (most significant) **bit** to specify the number's sign: 0 = positive and 1 = negative. Negative integers are represented in two's complement notation. Since the high-order bit is used for a sign, the range for an 8-bit integer is −128 through +127; 16-bit integers range from −32,768 through +32,767. The value zero always has a positive sign. Only the multiplication and division operations provide special instructions to handle integers. Addition and subtraction instructions assume that integers do not exist. However, that doesn't mean you can't add and subtract integers. You just treat them as unsigned numbers, and then use the appropriate "flags" to interpret the result. This will be covered in more detail when the arithmetic instructions are described.

Packed decimal numbers are stored as unsigned **byte** quantities. The byte is treated as having one decimal digit in each half-byte (four bits per half-byte); the digit in the high-order half-byte is the most significant digit. The range of a packed decimal number is 0 to 99. Addition and subtraction are performed in two steps. First, an unsigned binary instruction is used to produce an intermediate result in register AL. Then an adjustment operation is performed to change the value in the AL register to a final correct packed decimal result. When we discuss the decimal adjust for addition instruction, we will go into packed decimal operation in more detail. There is no provision for multiplying or dividing packed decimal numbers.

Unpacked decimal numbers are also stored as unsigned **byte** quantities. The magnitude of the number is determined from the low-order half-byte. The high-order half-byte must be zero for multiplication and division; however, it can contain any value for addition and subtraction.

Arithmetic on unpacked decimal numbers is performed in two steps for addition, subtraction, and multiplication. First, an unsigned binary instruction is used to produce an intermediate result in the AL register. Then an adjustment is made on the value in the AL register to produce the correct unpacked decimal number. Division is performed in essentially the same manner, except that after the value in the AL register is adjusted, a second unsigned binary division must be performed to produce the correct result. Since the instruction that is used to adjust the value in the AL register is the ASCII adjust instruction, we will cover unpacked decimal arithmetic in more detail when we describe the ASCII adjust instruction.

Recall from our discussion of jumps that arithmetic instructions post certain characteristics of the result of the operation to six flags. The various instructions affect the flags differently. While they generally follow the rules set down earlier, there are some exceptions. We will cover these exceptions when we describe the individual arithmetic instructions. Note that when we refer to a flag being set, reset, or updated, we will use the following abbreviations:

Auxiliary Carry Flag — AF
Carry Flag — CF
Overflow Flag — OF
Parity Flag — PF
Sign Flag — SF
Zero Flag — ZF

Addition

You've already used the instruction ADD to add two unsigned binary numbers. As Figure 6-3A shows, you can use the ADD instruction to add words to words, bytes to bytes, and immediate values to bytes or words. The sum is always stored in the destination operand. While the ADD instruction updates AF, CF, OF, PF, SF, and ZF; CF and ZF are generally the only flags you are interested in when you are adding unsigned binary numbers.

As we mentioned earlier, there is no special signed number ADD instruction. If you are working with signed numbers, then you must rely on the flags to determine the result. The two primary flags of interest are SF and OF. SF reflects the value of the high-order, or sign, bit (bit 7 for byte add and bit 15 for word add) of the result. Thus, if SF is 0 the number is positive, while a 1 would indicate a negative number. OF indicates whether the result of the add operation produced a value that was out of the range of numbers allowed in the destination: greater than +127 or +32,767, or less than −128 or −32,768, for byte and word values respectively. Naturally, if OF is set, SF is invalid, since the sign bit no longer contains the sign of the number.

Generally, signed arithmetic is used as a test prior to a conditional jump instruction. Therefore, depending on the specific jump, the microprocessor would, after a signed add operation, test: SF, OF, SF XOR OF, or (SF XOR OF) OR ZF. One thing you must keep in mind when you are performing a signed math operation is that the microprocessor will only recognize negative numbers in their 2's complement form. The assembler will automatically make the conversion for you if the value is part of the instruction. On the other hand, if the negative value is introduced while the program is running, then **you** must handle the conversion. Just don't make the mistake of thinking that you can make a positive binary number negative by simply changing the sign bit. While +19H equals 000011001B, −19H does not equal 100011001B. You must take the 2's complement of the positive value. Thus, if +19H equals 00011001B, then −19H must equal 11100111B, the 2's complement.

You can handle the problem of negative numbers in two ways. Either you make the adjustment before introducing the number to the MPU, or you can introduce the positive equivalent and then have the MPU make the conversion. If you choose the latter method, the instruction you should use is **NEG** (negate). The negate instruction essentially subtracts the destination operand from zero and returns the result to the destination operand. What this means is that negating a number will produce its 2's complement, effectively reversing the sign of the number. If the number is zero, it remains zero, but CF is cleared to indicate that the operation occurred. If the number is any value other than zero, CF is set. Attempting to negate a byte containing −128 or a word containing −32,768 again causes no change to the number; but in this case, OF is set to indicate the operation occurred. OF is cleared after negating any other number. Except for these two cases, the condition code flags are updated by the negate operation.

In addition to signed and unsigned binary numbers, you can use the ADD instruction to add **unpacked** decimal numbers, often called Binary Coded Decimal (BCD) numbers. A special instruction, **AAA** (ASCII Adjust for Addition), provides the necessary decoding, or adjustment. The term ASCII adjust may be misleading. Actually, the only ASCII characters that can be used are the numbers 0 through 9, and then only the low-order four bits of the byte are saved. This results in a BCD number, hence the reason for including unpacked BCD number adjustment with the instruction. Because of the way the microprocessor handles the AAA instruction, the data to be adjusted must reside in the AL register. As a result, only byte-wide add operations should precede the adjustment. The AAA instruction is performed in the following manner:

1. If the low-order four bits of the AL register are greater than 9, **or** AF is 1, add 6 to the AL register, add 1 to the AH register, and set AF to 1.

2. Clear the high-order four bits of the AL register.

3. Update CF to the same value in AF.

Notice that the AH register is used to store any overflow from the add and adjust operation. For that reason, you must make sure the AH register has been cleared before you begin any unpacked decimal operation. As an example of BCD addition, consider the following sequence of instructions:

```
SUB   AX,AX       ;Zero the AX register
ADD   AL,3        ;Load the first BCD
ADD   AL,5        ;Add the second BCD
AAA               ;Unpacked decimal adjust
ADD   AL,8        ;Add the third BCD
AAA               ;Unpacked decimal adjust
ADD   AL,7        ;Add the fourth BCD
AAA               ;Unpacked decimal adjust
```

The first instruction clears the AX register.

AH = 0000 0000 AL = 0000 0000

The next instruction adds 3 to the AL register.

AH = 0000 0000 AL = 0000 0011

The third instruction adds 5 to the AL register. Thus, the AL register now contains the number 8. Since there was no carry into the fifth bit of the AL register, AF equals zero. By the same token, there was no carry out of the last bit of the AL register, so CF is zero.

AH = 0000 0000 AL = 0000 1000
AF = 0 CF = 0

The fourth instruction adjusts the contents of the AL register to produce an unpacked decimal. Because the low-order four bits of the register are less than 0AH, AF is zero, CF is zero, and the high-order four bits of the register are zero; there is no immediate indication that the instruction was executed. Nothing has changed.

AH = 0000 0000 AL = 0000 1000
AF = 0 CF = 0

The fifth instruction adds 8 to the AL register. Therefore, the AL register now contains the number 16 (10H). In addition, there was a carry into the fifth bit of the AL register. Thus, AF is set to one. On the other hand, there was no carry out of the last bit of the AL register, so CF is still zero.

```
AH = 0000 0000    AL = 0001 0000
AF = 1            CF = 0
```

The sixth instruction again adjusts the contents of the AL register to produce an unpacked decimal. While the low-order four bits of the register are still less than 0AH, AF is now 1. This indicates that the AL register contents are no longer a valid BCD. Therefore, 6 is added to the AL register to make the low-order four bits a valid BCD. Then the high-order four bits are cleared to make the contents of the AL register a single BCD. Next, one is added to the AH register to represent the 10's digit of the BCD. Finally, both AF and CF are set to one. Although AF was already one in this example, the instruction is designed to accommodate the situation where the low-order four bits of the AL register are greater than 9 and AF is zero. Setting CF to one indicates that the 10's digit (the AH register) was incremented. Decode the contents of the AH and AL registers and you will find that they contain the decimal number 16, the sum of 3 plus 5 plus 8.

```
AH = 0000 0001    AL = 0000 0110
AF = 1            CF = 1
```

The seventh instruction adds 7 to the AL register. Therefore, the AL register now contains the number 13 (0DH). However, there was no carry into the fifth bit or out of the last bit, so both AF and CF are cleared, or zero. The AH register still contains the BCD one for the previous AAA operation.

```
AH = 0000 0001     AL = 0000 1101
AF = 0             CF = 0
```

The last instruction again adjusts the contents of the AL register. Because the low-order four bits of the register are now greater than 9, the instruction adds 6 to the register to make these bits a valid BCD. Then the four high-order bits are cleared. Next, one is added to the AH register to represent the decimal "carry" into the 10's digit. Finally, both AF and CF are set to one.

```
AH = 0000 0010     AL = 0000 0011
AF = 1             CF = 1
```

The four numbers added to the AL register in the preceding eight instructions totaled 23. Decode the BCD contents of the AH and AL registers and you will find the numbers 2 and 3 respectively. They represent the decimal number 23.

Examine the 7-bit ASCII Table in Figure 6-5, on Pages 6-22 and 6-23. (This is a copy of the table shown earlier in Unit 2.) Column 3 lists the decimal numbers.

	COLUMN	0 (3)	1 (3)	2 (3)	3	4	5	6	7 (3)
ROW	BITS 765 / 4321	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
10	1010	LF	SUB	*	:	J	Z	j	z
11	1011	VT	ESC	+	;	K	[	k	{
12	1100	FF	FS	,	<	L	\	l	|
13	1101	CR	GS	-	=	M	]	m	}
14	1110	SO	RS	.	>	N	^ (1)	n	~
15	1111	SI	US	/	?	O	_ (2)	o	DEL

Figure 6-5A
Table of 7-bit American Standard Code for Information Interchange.

NOTES:

(1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.

(2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.

(3) Explanation of special control functions in columns 0, 1, 2, and 7.

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell (audible signal)	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation (punched card skip)	EM	End of Medium
		SUB	Substitute
LF	Line Feed	ESC	Escape
VT	Vertical Tabulation	FS	File Separator
FF	Form Feed	GS	Group Separator
CR	Carriage Return	RS	Record Separator
SO	Shift Out	US	Unit Separator
SI	Shift In	DEL	Delete
SP	Space (blank)		

Figure 6-5B
Continuation of the table of 7-bit American Standard Code for Information Interchange.

Assuming the eighth bit in the ASCII code is a zero, then the 8-bit code for the number 7 is 0011 0111B. Perform an ASCII adjust for addition (AAA) operation on that code and the result is 0000 0111B — the BCD equivalent of the number 7. This would suggest that you can add ASCII numbers together, adjust the result, and come up with the BCD equivalent. And you can. For example, consider the following instructions:

```
SUB  AX,AX      ;Zero the AX register
IN   AL,25      ;Get ASCII number at port 25
MOV  BL,AL      ;Save ASCII number in BL
IN   AL,25      ;Next ASCII number at port 25
ADD  AL,BL      ;Add ASCII numbers
AAA             ;Adjust sum for BCD
```

Here we have a set of instructions that are loading ASCII numbers from a peripheral device, adding the numbers together, and adjusting the sum to produce the BCD equivalent. The first instruction, as before, clears the AX register to make sure we don't add some unknown value to our numbers.

AH = 0000 0000 AL = 0000 0000

The second instruction inputs the first byte of data into the AL register. Assume it's the ASCII code for the number 7. We will be covering the I/O instructions in detail later in the course. So for now, let's just say that the instruction IN inputs a byte of data into the AL register. The immediate value, in this case 25, indicates the I/O port that contains the data.

AH = 0000 0000 AL = 0011 0111

The third instruction saves the first ASCII number in the BL register.

AH = 0000 0000 AL = 0011 0111
BL = 0011 0111

The fourth instruction gets the second ASCII number at I/O port 25. Assume this number is 6.

AH = 0000 0000 AL = 0011 0110
BL = 0011 0111

The fifth instruction adds the number in the BL register to the number in the AL register, and stores the sum in the AL register. Notice that there was no carry into the fifth bit and no carry out of the last bit. Therefore, both AF and CF are zero.

```
AH = 0000 0000    AL = 0110 1101
                  BL = 0011 0111
AF = 0            CF = 0
```

The last instruction adjusts the contents of the AL register. Because the low-order four bits of the register are greater than 9, the instruction adds 6 to the register to make these bits a valid BCD. Then the four high-order bits are cleared, since we are now dealing with an unpacked decimal number. Next, one is added to the AH register to represent the decimal "carry" into the 10's digit. Finally, both AF and CF are set to one to indicate that a carry into the AH register did occur. Decode the contents of the AH and AL registers and you will find the numbers 1 and 3. They represent the decimal number 13, which is the sum of the numbers 7 and 6.

```
AH = 0000 0001    AL = 0000 0011
                  BL = 0011 0111
AF = 1            CF = 1
```

The four steps of ASCII Adjust for Addition apply whether you are adding BCD numbers or ASCII coded numbers. After the AAA instruction has been executed, only two flags remain valid. They are AF and CF. The other four condition code flags are considered undefined, and thus cannot be relied upon to regulate a conditional instruction.

The last number type that can be used with the ADD instruction is the packed decimal. The number is similar to an unpacked decimal, only now the four high-order bits contain a second or high-order digit. Of course, both digits are coded in BCD.

Earlier, we needed both the AH and AL registers to represent the number 23 in unpacked decimal.

```
AH = 0000 0010    AL = 0000 0011
```

You only need one 8-bit register to represent a two-digit number, such as 23, in packed decimal.

```
AL = 0010 0011
```

Naturally, adding two packed decimal numbers together could pose a problem. However, a special instruction, **DAA** (Decimal Adjust for Addition), provides the necessary decoding or adjustment. Remember though, DAA is used only with **packed** decimal numbers. If you wish to adjust an **unpacked** decimal number, use the AAA instruction.

Because of the way the microprocessor handles the DAA instruction, the data to be adjusted must reside in the AL register. As a result, only byte-wide add operations should precede the adjustment. The DAA instruction is performed in the following manner:

1. If AF is one, **or** the low-order four bits of the AL register are greater than 9, then add 06H to the AL register and set AF to one.

2. If CF is one, **or** the high-order four bits of the AL register are greater than 9, then add 60H to the AL register and set CF to one.

The operation is similar to the AAA instruction, within the boundaries set by the two previous conditions. As an example, consider the following sequence of instructions:

```
MOV   AL,29H    ;Load the first packed decimal
MOV   BL,49H    ;Load the second packed decimal
ADD   AL,BL     ;Add the numbers
DAA             ;Packed decimal adjust
MOV   BL,34H    ;Load the third packed decimal
ADD   AL,BL     ;Add the numbers
DAA             ;Packed decimal adjust
```

The first instruction loads the first packed decimal into the AL register. The AH register isn't involved in the DAA operation, so you don't have to clear it. You may have noticed that the hexadecimal based number system was used to specify the packed decimal. This is because it graphically shows the two "decimal" numbers being entered, since each hex number occupies four bit locations in a byte. Any other number system would work, but it would be difficult to interpret at a glance. For example, both decimal 41 and binary 0010 1001 are the equivalent of packed decimal 29.

AL = 0010 1001

The second instruction loads packed decimal 49 into the BL register.

```
AL = 0010 1001
BL = 0100 1001
```

The third instruction adds the contents of the BL register to the AL register. Because there was a carry from the fourth bit to the fifth bit, AF is set to one. On the other hand, there was no carry out of the last bit, so CF is cleared to zero.

```
AL = 0111 0010
BL = 0100 1001
AF = 1
CF = 0
```

The fourth instruction adjusts the contents of the AL register to produce two valid packed decimal numbers. First, the validity of the low-order number is checked. Since AF is set, 06H must be added to the register to make the low-order number valid.

```
  0111 0010
+ 0000 0110
-----------
  0111 1000
```

Next, the validity of the high-order number is checked. The high-order four bits are less than 0AH, and CF is zero, so the high-order number must be valid. Therefore, no further action is taken.

```
AL = 0111 1000
BL = 0100 1001
AF = 1
CF = 0
```

The fifth instruction loads packed decimal 34 into the BL register. Since the flags are unaffected by the MOV instruction, AF and CF stay the same.

```
AL = 0111 1000
BL = 0011 0100
AF = 1
CF = 0
```

The sixth instruction adds the contents of the BL register to the AL register. There are no carries, so AF and CF are cleared to zero.

```
AL = 1010 1100
BL = 0011 0100
AF = 0
CF = 0
```

The last instruction, DAA, again adjusts the contents of the AL register to produce two valid packed decimal numbers. First, the validity of the low-order number is checked. This time, AF is zero; however, the four low-order bits are greater than 9. Therefore, 06H is added to the register, to make the low-order number valid.

```
   1010 1100
+  0000 0110
-------------
   1011 0010
```

Notice that the adjustment process produced a carry into the fifth bit. This sets AF to one. It also increases the value of the high-order number by one. The next step in the adjustment process is to check the validity of the high-order number. CF is still zero; however, the four high-order bits are greater than 9. Therefore, 60H is added to the register to make the high-order number valid.

```
          1011 0010
        + 0110 0000
          ---------
CARRY→ 1  0001 0010
```

This last operation produced a carry out of the eighth bit of the register. Thus, CF is set to one.

```
AL = 0001 0010
BL = 0011 0100
AF = 1
CF = 1
```

Decode the AL register and you find it contains the packed decimal 12. Obviously, 29 plus 49 plus 34 does not equal 12. Rather, it equals 112. However, if you consider the CF an indicator of a carry into the 100's digit, then with CF set to one, the sum of the packed decimal numbers does indeed equal 112. Since the only indication of a carry into the 100's digit is the Carry flag, the maximum number that can be represented after a packed decimal adjust is 198 (99 plus 99).

As with the AAA instruction, AF and CF are the principle flags of interest in the DAA instruction. However, the DAA instruction will also update PF, SF, and ZF. OF is the only flag that remains undefined.

The last instruction to be discussed under the Addition is **ADC** (Add with Carry). It operates just like its counterpart ADD instruction, with one exception. After the two numbers are added together, the value of the Carry flag (before this addition) is then added to the sum; hence the term "add with carry." For example, assume the BL register contains the number 25H, the DL register contains the number 50H, and CF is set to one. Execute the instruction:

```
ADC BL,DL     ;Add bytes with carry
```

and the following steps occur:

```
    0010 0101 = BL
+   0101 0000 = DL
-------------------
    0111 0101 = BL
+          1  = CF
-------------------
    0111 0110 = BL
```

The contents of the DL register are added to the contents of the BL register, and the sum is stored in the BL register. Then the value of CF is added to the contents of the BL register and the sum is again stored in the BL register.

The beauty of the add with carry instruction is that you can easily add numbers that are larger than 16 bits in length. For example, suppose you wanted to add 2,097,151 to 1,050,640. Neither of these numbers will fit into a 16-bit register; the highest number the register can hold is 65,535. But you can fit each number into two 16-bit registers. Let's examine the process.

To keep the explanation simple, we'll first convert the numbers into hexadecimal:

2,097,151 = 001F FFFFH
1,050,640 = 0010 0810H

The following program loads the numbers and then performs the add-with-carry operation.

```
MOV   AX,0FFFFH ;Load the low-word of the first number
MOV   BX,001FH  ;Load the high-word of the first number
MOV   CX,0810H  ;Load the low-word of the second number
MOV   DX,0010H  ;Load the high-word of the second number
ADD   AX,CX     ;Add the low-words
ADC   BX,DX     ;Add the high-words and any carry
```

The first four instructions load the numbers into the registers: first number into AX and BX, second number into CX and DX. The fifth instruction then adds the low-word of the second number to the low-word of the first number, and stores the sum in the AX register. Since there was a carry out of the high bit, CF is set.

```
                1111 1111 1111 1111 = AX
              + 0000 1000 0001 0000 = CX
CARRY ──►   1 0000 1000 0000 1111 = AX
```

The ADD instruction was used in this step for two reasons. First, there was no need to accommodate a carry into the number. Second, the status of the Carry flag was unknown. If the ADC instruction had been used, it would have been necessary to also use the CLC instruction to make sure CF was cleared before the ADC instruction was executed.

The last instruction added the high-word of the second number to the high-word of the first number and stored the sum in the BX register. Then it added the contents of CF to the contents of the BX register and again stored the sum in the BX register. Here, ADC was used, since it is important that any carry from the previous addition be added to this number.

```
    0000 0000 0001 1111 = BX
+   0000 0000 0001 0000 = DX
-------------------------------
    0000 0000 0010 1111 = BX
+                    1 = CF
-------------------------------
    0000 0000 0011 0000 = BX
```

Thus, the sum of your two-word add resides in registers AX and BX. Decoded, the decimal number is 3,147,791. While this example used unsigned binary math, it will also work for signed binary, unpacked decimal, and packed decimal.

Self-Review Questions

5. If the decimal equivalent of the unsigned binary number is 25, the signed binary number is _____.

6. If the decimal equivalent of the 8-bit unsigned binary number is 153, the signed binary number is _____.

7. If the decimal equivalent of the unsigned binary number is 8, the unpacked decimal number is _____.

8. If the decimal equivalent of the unsigned binary number is 103, the packed decimal number is _____.

9. The largest negative decimal number that can be represented in eight binary bits is _____.

10. The instruction for adding signed binary words between registers is _____.

11. After you add two unpacked decimal numbers, you must adjust the result with the _____ instruction.

12. For a decimal adjust operation, the operand must reside in the _____ register.

13. In unpacked decimal adjusts, the 10's digit is stored in the AH register; while in packed decimal adjusts, the 100's digit is stored in the _____ _____.

14. The _____ instruction lets you add multiword (or byte) unsigned binary numbers.

Subtraction

You've already used the instruction SUB to subtract one unsigned binary number from another. As previously indicated in Figure 6-3B (Page 6-13), you can use the SUB instruction to subtract words from words, bytes from bytes, and immediate values from words or bytes. The difference is always stored in the destination operand. While the SUB instruction updates AF, CF, OF, PF, SF, and ZF; CF and ZF are generally the only flags you are interested in when subtracting unsigned binary numbers.

As with signed addition, there is no specific signed number subtraction instruction. If you are working with signed numbers, then you must rely on the flags to determine the result. The two primary flags of interest are SF and OF. SF reflects the sign of the high-order bit of the result. OF indicates whether the result of the "add" operation overflowed into the high-order bit. Remember, in a subtraction operation, the "2's complement" of the source is "added" to the destination operand. Thus, if OF is set, SF is invalid, since the sign bit no longer contains the sign of the number.

In addition to signed and unsigned binary numbers, you can use the SUB instruction to subtract packed and unpacked decimal numbers. The **AAS** (ASCII Adjust for Subtraction) instruction is used to adjust the difference after an unpacked decimal subtraction. As with the AAA instruction, the number must reside in the AL register. Therefore, you should only precede this instruction with byte-wide subtraction operations. The AAS instruction is performed in the following manner:

1. If the low-order four bits of the AL register are greater than 9, **or** AF is one, subtract 6 from the AL register, subtract one from the AH register, and set AF to one.

2. Clear the high-order four bits of the AL register.

3. Update CF to the same value in AF.

Notice that the AH register is used to supply any borrow from the subtract and adjust operation. For that reason, you must make sure you know what value is stored in the AH register prior to the adjust operation. Otherwise, you won't have a means of determining the validity of the result.

The three steps of ASCII Adjust for Subtraction apply whether you are subtracting BCD numbers or ASCII encoded numbers. After the AAS instruction has been executed, only two flags remain valid: AF and CF. The other four condition code flags are undefined, and cannot be relied upon to regulate a conditional instruction.

The **DAS** (Decimal Adjust for Subtraction) instruction is used to adjust the difference after a packed decimal subtraction. Since the number must reside in the AL register, you should only precede this instruction with byte-wide subtraction operations. The DAS instruction is performed in the following manner:

1. If AF is one **or** the low-order four bits of the AL register are greater than 9, subtract 06H from the AL register and set AF to one.

2. If CF is one **or** the high-order four bits of the AL register are greater than 9, subtract 60H from the AL register and set CF to one.

The operation is very similar to the DAA instruction. After execution, all of the condition code flags are updated, except for OF. Its status is undefined.

The **SBB** (Subtract with Borrow) instruction carries the simple subtract instruction one step further. After the two numbers are subtracted, the value of CF (in this case it's considered the **borrow** flag) is subtracted from the result. Thus, like its counterpart the ADC instruction, SBB lets you subtract more than one number from another because it incorporates a previous "borrow" into the calculation. For that same reason, you can also use SBB to subtract numbers that are larger than 16 bits in length. This would be similar in operation to the ADC example given earlier. Since the subtract with borrow instruction is an extension of the simple subtract instruction, it too can be used with signed and unsigned binary numbers, and packed and unpacked decimal numbers.

The last three instructions that fit in the category of "subtraction" are compare, decrement, and negate. Recall that **CMP** (Compare) subtracts the source operand from the destination operand, and uses the result to update the condition code flags. However, the operand values are **not** affected by the operation.

The **DEC** (Decrement) instruction subtracts one from the destination operand. All of the condition code flags except CF are updated by this instruction. CF is unaffected by the decrement instruction, and thus retains its previous state.

Last of the subtraction instructions is **NEG** (Negate). It was described earlier in this unit.

Self-Review Questions

15. The instruction for subtracting binary words between registers is _____.

16. The AAS instruction provides the necessary adjustment to allow you to subtract one ASCII number code from another. _____
    True/False

17. The _____instruction lets you subtract more than one multibyte operand from another.

18. The compare instruction subtracts the source operand from the destination operand, and stores the result in the destination operand. _____
    True/False

19. Negating a number produces its 2's complement. _____
    True/False

Multiplication

The multiplication instructions expand your capabilities as a programmer. You no longer have to create complex program loops in order to multiply one number by another; simply load the data and execute the appropriate multiply instruction. The multiplication instructions are listed in Figure 6-3C (Page 6-14).

The **MUL** (Multiply) instruction is used for multiplying unsigned binary values. For byte-sized values, the source operand (the multiplier) can be an 8-bit register or a "defined byte" memory location (variable). The byte value being multiplied (the multiplicand) must, by default, reside in the **AL** register. The product is returned as a 2-byte value, with the **AH** register holding the high byte and the **AL** register holding the low byte.

Word-sized values are multiplied in a similar fashion. The source operand can be a 16-bit general register or a "defined word" memory location. The word value being multiplied must reside in the **AX** register. The product is returned as a 2-word value, with the **DX** register holding the high word and the **AX** register holding the low word. To get an idea of how the MUL instruction can be used to multiply two words, examine the following instructions:

```
MOV  AX,5632H  ;Load the multiplicand
MOV  CX,0124H  ;Load the multiplier
MUL  CX        ;Multiply the registers
```

The first two instructions load the multiplicand and multiplier into the AX and CX registers. The multiplicand must reside in the AX register. The multiplier, on the other hand, can reside almost anywhere. However, if you use the DX register (AH for byte values), the high word (byte) will replace the multiplier value in the register.

```
AX = 0101 0110 0011 0010
CX = 0000 0001 0010 0100
```

The last instruction multiplies the CX register contents times the AX register contents, and stores the low word of the product in the AX register and the high word of the product in the DX register.

```
AX = 0101 0001 0000 1000
CX = 0000 0001 0010 0100
DX = 0000 0000 0110 0010
```

Thus, the result of multiplying 0124H times 5632H equals 00625108H. Decoded into decimal, 292 times 22,066 equals 6,443,272.

If the result of the multiplication operation produces a zero in the upper-half of the product (AH register for byte multiply and DX register for word multiply), CF and OF are cleared; otherwise, they are set. In the previous example, CF and OF would have been set. The other four condition code flags are undefined for an unsigned binary multiply operation.

Unlike the addition and subtraction instruction set, the multiplication instruction set has a specific instruction for **integer** (signed binary) **multiplication** (IMUL). Generally, integer multiplication operates in the same manner as unsigned multiplication. There are three exceptions though. First, for byte operations, both numbers must fall in the range +127 to −128; while for word operations, both numbers must fall in the range +32,767 to −32,768. Second, if the high-order half of the product is zero, the high-order register is automatically sign extended. For example, multiply 3 times 5 and the AL register will contain the value 00001111B. The AH register contains zero, which happens to be the sign of the AL register.

AH = 0000 0000 AL = 0000 1111

On the other hand, if you multiply −3 times 5, the AL register will contain the value 11110001B, the 2's complement of 15. The AH register is again zero, but because of the sign extension rule, its bits are changed from zeros to ones.

AH = 1111 1111 AL = 1111 0001

The sign of the AL register is extended into the AH register whenever the AH register is zero after an integer **byte** multiplication. This is also true for integer **word** multiplication, with the DX register receiving the sign extension.

The third difference between signed and integer multiplication is in the use of the two condition code flags CF and OF. If the high-order half of the result is the sign extension of the low-order half, both CF and OF are cleared. On the other hand, if the high-order half contains a valid number, CF and OF are set. The other four condition code flags are undefined for integer multiplication.

The last instruction in the multiplication instruction set is **AAM** (ASCII Adjust for Multiply). It lets you correct the result of a previous multiplication of two valid unpacked decimal operands. Unlike the AAA and AAS instructions, AAM requires that only valid unpacked BCD numbers be used in the preceding multiply instruction. This means that you **can** multiply 06H times 04H and obtain a valid number after the adjustment, but you **cannot** multiply 36H (ASCII code for decimal 6) times 04H and obtain a valid number after adjustment. If you wish to use ASCII coded numbers, you must first clear, or mask, the four high-order bits of the ASCII code. This is generally done with the logical AND instruction, as was the case when we masked a record field in the last unit. For example, suppose the AL register contains the ASCII code 36H and the BL register contains the value 04H. Before you can multiply and adjust the result, you must use the instruction:

```
AND AL,0FH
```

which leaves the AL register containing 06H. Now the values in AL and BL can be multiplied together, and the result adjusted with the AAM instruction.

The AAM instruction operates as follows:

1. Divide the AL register contents by 0AH. Store the quotient in the AH register. Store the remainder in the AL register.

2. Set the condition code flags in the following manner:

   CF, OF, and AF undefined.
   PF based on the AL register contents.
   SF based on the high-order bit of the AL register.
   ZF based on the AL register contents.

As an example, suppose that the AL register contains 07H and the BL register contains 09H. After the sequence of instructions:

```
MUL  BL    ;AL multiplied by BL, product in AL
AAM        ;ASCII adjust the product in AL
```

the AH register will contain 06H and the AL register will contain 03H. The sequence of events occurred in this manner: First, the MUL instruction produced the results:

AH = 0000 0000 AL = 0011 1111

Then the AAM instruction divided the contents of the AL register by 0AH, stored the quotient in the AH register and the remainder in the AL register, and set the appropriate flags. The flags were set in the following manner:

CF — Undefined
OF — Undefined
AF — Undefined
SF — High-order bit of AL is 0, Sign cleared to 0
ZF — AL register is non-zero, Zero cleared to 0
PF — Two 1-bit in AL, Parity set to 1

Had the result of the multiplication equaled 09H or less, the AH register would contain zero and the AL register would contain the result.

Self-Review Questions

20. The instruction for multiplying two unsigned binary words is _____.

21. The instruction for multiplying two signed binary bytes is _____.

22. If you multiply one word times another, the answer is stored in the _____ and _____ registers.

23. Another term for signed number is _____.

24. If you multiply the byte 2 times the byte −5, the number stored in the AH register is __ H.

25. The AAM instruction lets you adjust the result after multiplying two ASCII encoded numbers together. _____

True/False

Division

Five instructions are provided in the division instruction set, as shown in Figure 6-3C (Page 6-14). Three are directly related to division and adjustment; the other two are sign extension operations to support signed division.

The **DIV** (Divide) instruction is used for dividing unsigned binary values. For byte-sized division, a **word-sized dividend** is placed in the AX register. The **byte-sized divisor** (the source operand) can be located in an 8-bit register or a "defined byte" memory location (variable). When the instruction is executed, the **quotient** is returned to the AL register and the **remainder** is returned to the AH register. If the quotient exceeds the capacity of the AL register, the MPU generates a **type 0 interrupt**; the quotient and remainder are then considered undefined. We'll cover this and other interrupts later in the course. For now, we'll make sure all of our division operations don't exceed the capacity of the "quotient" register.

Word-sized division operates in a similar fashion, only this time the **dividend** is **doubleword-sized**. The **high word** is stored in the DX register and the **low word** is stored in the AX register. The **word-sized divisor** (the source operand) can be located in a 16-bit general register or a "defined word" memory location. When the instruction is executed, the **quotient** is returned to the AX register and the **remainder** is returned to the DX register. As before, if the quotient exceeds the capacity of its register, a type 0 interrupt is generated by the MPU. As an example of word division, consider the following instructions:

```
MOV  DX,068AH  ;Load the dividend high-word
MOV  AX,0F05H  ;Load the dividend low-word
MOV  CX,08E9H  ;Load the divisor
DIV  CX        ;Perform word divide, CX is divisor
```

The first three instructions load the dividend and divisor. By default, the dividend must reside in the DX and AX registers. The divisor can reside almost anywhere. To keep the code simple, we chose the CX register.

```
DX = 0000 0110 1000 1010
AX = 0000 1111 0000 0101
CX = 0000 1000 1110 1001
```

The last instruction divides the contents in DX and AX registers by the contents in CX register. The quotient is stored in AX register and the remainder is stored in DX register.

```
AX = 1011 1011 1110 0001
DX = 0000 0111 0011 1100
CX = 0000 1000 1110 1001
```

Thus, the result of dividing 068A0F05H by 08E9H is a quotient of 0BBE1H and a remainder of 073CH. All of the condition code flags are considered undefined after a word- or byte-sized divide operation.

Like the multiplication instruction set, the division instruction set has a specific instruction for **integer** (signed binary) **division** (IDIV). Generally, integer division operates in the same manner as unsigned division. There are three exceptions though. First, the high bit of the dividend and the divisor are considered the sign bit. Second, the remainder will assume the sign of the dividend. Finally, the quotient and remainder are treated as signed numbers with a maximum allowable range. For byte operations, the quotient and remainder cannot exceed the range +127 to −128. For word operations, the quotient and remainder cannot exceed the range +32,767 to −32,768. Should the quotient exceed either its positive or negative range, a type 0 interrupt will be generated, and both the quotient and remainder will be considered undefined. All condition code flags are considered undefined for an integer division operation.

Two other instructions in the division instruction set are used to support signed division. Essentially, they are used to extend the sign of a number to make the number compatible with the signed division operation. The first, **CBW** (Convert Byte to Word), extends the sign of the sign of the byte in the AL register throughout the AH register. Thus, CBW can be used to produce a double-length dividend from a single byte prior to signed-byte division. The second sign extension instruction is **CWD** (Convert Word to Doubleword). Here, the sign of the word in the AX register is extended throughout the DX register. Thus, CWD can be used to produce a doubleword dividend from a single word prior to signed-word division. Neither CBW or CWD affect the status of the condition code flags.

The last instruction to be covered is **AAD** (ASCII Adjust for Division). This instruction is a little different from the other ASCII adjust instructions; it is executed **prior** to the division operation rather than after. Thus, when you wish to divide two unpacked decimal numbers (BCD not ASCII), you place the most significant number in the AH register and the least significant number in the AL register. The AAD instruction combines the two unpacked decimal numbers and places them in the AL register as an **unsigned binary** value. The AAD instruction operates as follows:

1. Multiply the contents of the AH register by 0AH.
2. Add the contents of the AH register to the AL register.
3. Store 00H in the AH register.
4. Set the condition code flags in the following manner:

   CF, OF, and AF undefined.
   PF based on the AL register contents.
   SF based on the high-order bit of the AL register.
   ZF based on the AL register contents.

As an example, suppose the AX register contains the two unpacked decimal values 0604H. After the instruction AAD is executed, the AX register will contain the unsigned binary value 0040H. The flags are updated in the following manner:

CF — Undefined
OF — Undefined
AF — Undefined
SF — High-order bit of AL is 0, Sign cleared to 0
ZF — AL register is non-zero, Zero cleared to 0
PF — One 1-bit in AL, Parity cleared to 0

The next instruction after AAD should be DIV. This will produce an **unsigned binary quotient and an unsigned binary remainder**, not the unpacked decimal you might expect.

Self-Review Questions

26. If you divide a doubleword value by a word value, the dividend must reside in the _____ and _____ registers.

27. The quotient from a byte divide is stored in the _____ register.

28. If the quotient from a word divide exceeds the capacity of the AX register, the excess is stored in the DX register. _____
True/False

29. The mnemonic for an integer divide word is _____.

30. The maximum negative remainder after an integer divide byte operation is _____.

31. The _____ instruction is used to extend the sign of a byte-sized number before the divide operation.

32. You would use the _____ instruction to convert a word to a doubleword.

33. The AAD instruction is performed prior to the division of two unpacked decimal numbers. _____
True/False

BIT MANIPULATION INSTRUCTIONS

The 8088/8086 MPU provides three groups of instructions for manipulating bits within both bytes and words. These include logical operations, shifts, and rotates, as listed in Figures 6-6A (logicals), 6-6B (shifts), and 6-6C (rotates). These instructions can be performed on both register and memory operands.

Logicals

Logical instructions include the Boolean operators "not," "and," "inclusive or," and "exclusive or," plus a TEST instruction that sets the condition code flags but does not alter either the source or destination operand. Although the logical NOT does not affect the status of the condition code flags, the flags are affected by the logical instructions AND, OR, XOR, and TEST. The flags are updated in the following manner:

OF — Cleared by the operation.
CF — Cleared by the operation.
AF — Undefined after the operation.
PF — Reflects the number of 1-bits in the destination operand (low-order byte for word-sized operands); set to one for an even number, cleared to zero for an odd number.
SF — Reflects the status of the most significant (sign) bit of the destination operand.
ZF — Reflects the numeric value of the destination operand; set to one if value is zero, cleared to zero if value not zero.

NOT inverts the bits to form the 1's complement of the destination operand. This and the other logical, Boolean, operations are fully described and illustrated in Appendix C of this course.

AND performs the logical "and" operation between the source and destination operands, and returns the result to the destination operand. Recall that a bit in the result is set if both corresponding bits of the original operands are set; otherwise, the bit is cleared.

MNEMONIC	SYNTAX	RESULT
	LOGICALS	
NOT	mem/reg	Perform 1's complement of memory or register, byte or word
AND	reg,mem/reg	Perform bitwise logical "and" of register with memory or register, byte or word
AND	mem/reg,reg	Perform bitwise logical "and" of memory or register with register, byte or word
AND	mem/reg,numb	Perform bitwise logical "and" of memory or register with immediate data, byte or word
OR	reg,mem/reg	Perform bitwise logical "inclusive or" of register with memory or register, byte or word
OR	mem/reg,reg	Perform bitwise logical "inclusive or" of memory or register with register, byte or word
OR	mem/reg,numb	Perform bitwise logical "inclusive or" of memory or register with immediate data, byte or word
XOR	reg,mem/reg	Perform bitwise logical "exclusive or" of register with memory or register, byte or word
XOR	mem/reg,reg	Perform bitwise logical "exclusive or" of memory or register with register, byte or word
XOR	mem/reg,numb	Perform bitwise logical "exclusive or" of memory or register with immediate data, byte or word
TEST	reg,mem/reg	Perform bitwise logical "and" of register with memory or register, byte or word; update flags, but not destination
TEST	mem/reg,reg	Perform bitwise logical "and" of memory or register with register, byte or word; update flags, but not destination
TEST	mem/reg,numb	Perform bitwise logical "and" of memory or register with immediate data, byte or word; update flags, but not destination

Figure 6-6A
Bit manipulation instructions, logicals.

MNEMONIC	SYNTAX	RESULT
	SHIFTS	
SHL/SAL	mem/reg,1	Shift logical/arithmetic left, byte or word, memory or register 1 bit; shift in low-order zero bit
SHL/SAL	mem/reg,CL	Shift logical/arithmetic left, byte or word, memory or register number of bits given by CL register; shift in low-order zero bits
SHR	mem/reg,1	Shift logical right, byte or word, memory or register 1 bit; shift in high-order zero bit
SHR	mem/reg,CL	Shift logical right, byte or word, memory or register number of bits given by CL register; shift in high-order zero bits
SAR	mem/reg,1	Shift arithmetic right, byte or word, memory or register 1 bit; shift in high-order bit equal to the original high-order bit
SAR	mem/reg,CL	Shift arithmetic right, byte or word, memory or register number of bits given by CL register; shift in high-order bits equal to the original high-order bit

Figure 6-6B
Bit manipulation instructions continued, shifts.

MNEMONIC	SYNTAX	RESULT
	ROTATES	
ROL	mem/reg,1	Rotate byte or word, memory or register 1 bit left
ROL	mem/reg,CL	Rotate byte or word, memory or register left number of bits given by CL register
ROR	mem/reg,1	Rotate byte or word, memory or register 1 bit right
ROR	mem/reg,CL	Rotate byte or word, memory or register right number of bits given by CL register
RCL	mem/reg,1	Rotate byte or word, memory or register 1 bit left through Carry flag
RCL	mem/reg,CL	Rotate byte or word, memory or register left through Carry flag number of bits given by CL register
RCR	mem/reg,1	Rotate byte or word, memory or register 1 bit right through Carry flag
RCR	mem/reg,CL	Rotate byte or word, memory or register right through Carry flag number of bits given by CL register

Figure 6-6C
Bit manipulation instructions continued, rotates.

OR performs the logical "inclusive or" operation between the source and destination operands, and returns the result to the destination operand. Recall that a bit in the result is set if either or both corresponding bits in the original operands are set; otherwise the bit is cleared.

XOR performs the logical "exclusive or" operation between the source and destination operands, and returns the result to the destination operand. Recall that a bit in the result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is clear); otherwise, the bit is cleared.

TEST performs the logical "and" operation between the source and destination operands, but **does not** return the result to the destination operand. Neither operand is affected by the operation, only the condition code flags are affected.

Shifts

The bits in bytes and words may be shifted arithmetically or logically up to 255 times, although a shift of more than 16 bits is of limited value. The number of shifts is determined by a count specified in the source operand. If the count is 1, that value can be stored as a constant in the source operand. If the count is greater than one, the count must be stored in the CL register and the CL register specified as the source operand. If a count of zero is stored in the CL register, the shift operation will be ignored. Examples of the two methods of specifying the count are:

```
SHR  AL,1       ;Shift logical right one bit
SHR  AL,CL      ;Shift logical right "count" bits
```

Logical shifts can be used to isolate bits on bytes or words (record handling is a good example). Arithmetic shifts can be used to multiply or divide binary numbers by powers of two. Both types of shifts affect the condition code flags in the following manner:

AF — Undefined after the operation.
PF — Reflects the number of 1-bits in the destination operand (low-order byte in word-sized operands); set for an even number of 1-bits, cleared for an odd number.
SF — Reflects the status of the most significant, sign, bit of the destination operand.

ZF — Reflects the numeric value of the destination operand; set if the operand is zero, cleared if the operand contains a value other than zero.

CF — Always contains the value of the **last** bit shifted out of the destination operand.

OF — Always undefined following a multibit shift. In a single-bit shift, set if the value of the sign bit was changed by the operation; cleared if the sign bit retains its original value.

The **SHL** (Shift Logical Left) and **SAL** (Shift Arithmetic Left) instructions operate in the same manner. They shift the destination operand bits to the left the number of times specified by the count in the source operand. As the bits are shifted, the empty bit positions are filled with zeros. To see how this works, suppose the DX register contained the bit pattern 0FFFFH. The following instructions:

```
SHL  DX,1       ;Shift logical left one bit
MOV  CL,0AH     ;Load the shift "count" register
SHL  DX,CL      ;Shift logical left "count" times
```

would produce these results: The first instruction shifts the DX register contents to the left one; the least significant bit is now zero.

```
DX = 1111 1111 1111 1110
```

The next instruction loads the "count" 10 into the CL register.

```
DX = 1111 1111 1111 1110
CL = 0000 1010
```

The last instruction shifts the contents of the DX register to the left the number of bits specified in the CL register. The operation forms a small loop where each time the DX register is shifted left one bit, the CL register is decremented by one. When the count in the CL register reaches zero, the shift operation ends.

```
DX = 1111 1000 0000 0000
CL = 0000 0000
```

As we said earlier, SHL and SAL perform the same shift left operation. It is up to you which instruction mnemonic you use. That isn't the case for the shift right operation. Here, the "logical" and "arithmetic" operations perform different functions. The instruction **SHR**, (Shift Logical Right), shifts the bits in the destination operand to the right the number of bit locations specified in the "count." The empty bit positions are filled with **zeros**. The **SAR** (Shift Arithmetic Right) instruction, on the other hand, doesn't arbitrarily fill the empty bit positions with zeros. Rather, it fills the empty bit positions with bits equal to the original value of the most significant, or sign, bit. Thus, the sign of the original operand is retained regardless of the number of bit shifts.

Now you should be able to see the purpose for the two types of shift operations. Logical shifts are used to isolate a bit pattern. The sign of the original value is unimportant. Arithmetic shifts, on the other hand, need to retain the original sign of the value. Thus in an arithmetic "right" shift, the sign is retained. In an arithmetic "left" shift, this isn't practical; hence the reason for the SHL and SAL instructions operating in the same manner. There is, however, a way to preserve the sign of a byte-sized value in a shift left operation. First, place the value to be shifted into the AL register. Then precede the shift operation with the CBW (Convert Byte to Word) instruction. This will extend the sign of the value in the AL register into the AH register. Finally, instead of the instruction:

```
SAL AL,CL       ;Shift arithmetic left BYTE, "count" times
```

use the instruction:

```
SAL AX,CL       ;Shift arithmetic left WORD, "count" times
```

As long as you don't exceed the capacity of the AH register, less one, the shifted value will retain its original sign.

Rotates

Rotate instructions are similar to the shift instructions, only here, the bits aren't lost as in a shift. Instead, the bits "circle" back into the "other end" of the operand. Again, the number of bits moved is determined by the value in the source operand. This can be the constant one, or some number between 1 and 255 stored in the CL register. A count of zero will cause the rotate to be ignored.

In addition to rotating each bit out one end and into the other end of the operand, the rotate instruction saves the value of the rotated bit in the Carry flag. When more than one bit is rotated around the operand, the Carry flag echoes the value of each bit as it is rotated. The value of the last bit rotated is retained in the Carry flag. This gives you an opportunity to isolate a particular bit in the flag and then test the bit with a conditional jump instruction.

Figure 6-7 illustrates a simple rotate left operation. Prior to execution, the operand contains the value 11110000B and the Carry flag is cleared. After execution, the most significant bit in the operand has rotated around to the least significant bit position and the Carry flag echoes the value of the rotated bit.

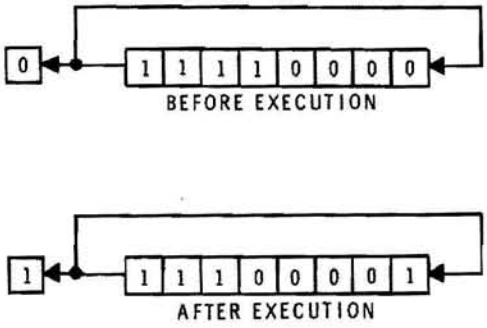


Figure 6-7
Simple rotate left operation.

In addition to the Carry flag, the rotate instruction updates the Overflow flag. In a single-bit rotate, OF is set if the value of the sign bit was changed by the rotate. If there was no change, OF is cleared. In a multi-bit rotate, OF is undefined. The other flags are unaffected by a rotate operation.

There are two basic rotate instructions: **ROL**, (Rotate Left), and **ROR**, (Rotate Right). The rotate left instruction shifts the bits out of the most significant bit (MSB) end of the operand, and immediately shifts them back into the least significant bit (LSB) end of the operand. As an example, suppose the BL register contains 0AFH, the CL register contains 04H, and CF and OF are cleared.

```
BL = 1010 1111
CL = 0000 0100
CF = 0
OF = 0
```

After the instruction:

```
ROL BL,CL      ;Rotate left "count" times
```

is executed, the BL register will contain 0FAH, the CL register will contain zero, the last rotate clears CF, and OF is undefined.

```
BL = 1111 1010
CL = 0000 0000
CF = 0
OF = ?
```

The rotate right instruction operates in the same fashion, except the bits are shifted out of the LSB and into the MSB.

A variation of the simple rotate instruction is the rotate through carry instruction where the Carry flag is part of the operation. Here, the rotated bits **pass through** the Carry flag before returning to the operand. Thus, when a bit is rotated out of the operand, it is shifted into the Carry flag; the empty bit position in the operand is filled with the original contents of the Carry flag. An example of a rotate through carry left operation is shown in Figure 6-8.

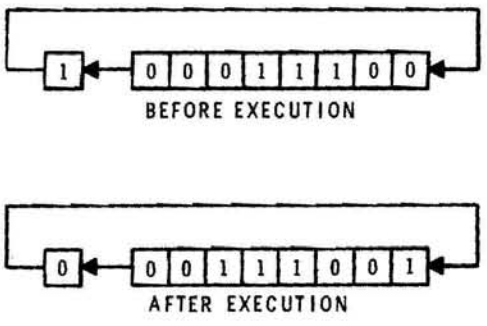


Figure 6-8
Rotate left through carry operation.

There are two rotate through carry instructions: **RCL**, (Rotate Through Carry Left), and **RCR**, (Rotate Through Carry Right). The rotate through carry left instruction shifts a bit out of the MSB of the operand and into the Carry flag. At the same time, the Carry flag bit is shifted into the empty LSB of the operand. The rotate through carry right instruction does just the opposite. Here, the LSB is shifted out of the operand and into the Carry flag. At the same time, the Carry flag bit is shifted into the empty MSB of the operand. As an example, suppose the BL register contains 01FH, and CF and OF are cleared.

BL = 0001 1111
CF = 0
OF = 0

After the instruction:

```
RCR  BL,1
```

is executed, the BL register will contain 0FH and CF is set. Since the sign bit didn't change, OF remains cleared.

BL = 0000 1111
CF = 1
OF = 0

The rotate and rotate through carry instructions are identical except for the way the Carry flag is handled. In the simple rotate operation, the Carry flag echoes the value of the last bit rotated. In the rotate through carry operation, the Carry flag stores the last bit rotated, while the displaced flag bit is shifted into the operand.

Self-Review Questions

34. The TEST instruction uses an arithmetic operation to update the condition code flags. _____
True/False

35. After a "logical" operation, the Auxiliary Carry flag is _____.

36. The NOT operation forms the 2's complement of the destination operand. _____
True/False

37. The shift operation lets you shift up to _____ bits out of the destination operand.

38. The _____ register is used to hold the shift "count."

39. The _____ instruction preserves the original sign of the operand during a shift operation.

40. All of the rotate instructions save the value of the last bit rotated out of the operand in the Carry flag. _____
True/False

41. If the AL register contains 0FH, CF is set, and OF is cleared, then the AL register will contain _____ after the instruction:

```
ROL   AL,1
```

is executed.

EXPERIMENT

Expanding the Instruction Set

OBJECTIVES:

1. *Demonstrate the transfer and translation instructions.*
2. *Demonstrate a few of the arithmetic instructions.*
3. *Demonstrate the bit manipulation instructions.*

Introduction

This experiment will give you an opportunity to use an example of each of the instructions introduced in Unit 6. These include the transfer and translation instructions, a cross section of the arithmetic instructions, and all of the bit manipulation instructions. To add more interest to the programs, some of them will send a message to the display.

The Zenith and IBM microcomputer systems use different methods for accessing their display. To accommodate these differences, the programs that use the display will contain a separate display control routine for each system. We'll examine these different routines in the first program in the experiment.

Procedure

1. Figures 6-9A and 6-9B contain a program that illustrates the transfer and translation instructions. It also sends a decoded message to the display. Two different interrupt service routines are used to control the display. One is called CLEAR_ZENITH, the other is called CLEAR_IBM. Each is accessed through a CALL instruction. You will find these instructions between two rows of Xs in Figure 6-9A. As the program is listed, the:

```
CALL CLEAR_IBM
```

   instruction is disabled, because it is preceded by a semicolon. Thus, the program will only work on a Zenith computer using Z-DOS as a "system program." To use the program on an IBM or IBM compatible microcomputer, remove the semicolon from in front of the instruction:

```
CALL CLEAR_IBM
```

   and place a semicolon in front of the instruction:

```
CALL CLEAR_ZENITH
```

   Call up the editor and enter the correct program for your system. Then assemble, link, and convert your program into a COM file. (Be sure to load the complete program, not just the code and data for your system. Later, you will be asked to examine specific memory locations. If you leave out part of the program, those memory locations will be incorrect.)

2. The program is essentially divided into three sections. The first illustrates the four transfer instructions. The second uses the translation instruction to decode a message. The last is a simple routine to display the decoded message. It really is simple; it just appears complex because we have to accommodate two different display systems.

   Call up the debugger and load your program COM file. In the following steps you will single-step through the first two sections of the program. Then you will let the debugger run the last section of the program.

3. First, type "R" and RETURN to display the registers and the first instruction in the program. Single-step through the instruction. The AL register bits are loaded with ones.

```
TITLE EXPERIMENT 6 -- PROGRAM 1 -- DATA TRANSFER AND TRANSLATION
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  MOV     AL,0FFH         ;Fill register with ones
        LAHF                    ;Get low byte of flag register
        XCHG    AL,AH           ;Swap data between registers
        SAHF                    ;Store ones to low byte flag reg.
        LEA     BX,TABLE        ;Get address of data table
        LEA     BP,CODE         ;Get address of data code
        LEA     DI,DECODE       ;Get address of decoded code storage
REPEAT: MOV     AL,[BP]         ;Get byte of code
        INC     BP              ;Point to next code byte
        XLAT                    ;Decode the code
        MOV     [DI],AL         ;Store decoded code
        INC     DI              ;Point to next storage area
        CMP     AL,'$'          ;Was last decoded byte a '$'?
        JNE     REPEAT          ;No, decode next code byte
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        CALL    CLEAR_ZENITH    ;Zenith clear screen routine
;       CALL    CLEAR_IBM       ;IBM clear screen routine
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        MOV     AH,2            ;Load interrupt "display
                                ;character" command
        LEA     DI,DECODE       ;Get address of decoded code
OUTPUT: MOV     DL,[DI]         ;Load first byte to be "displayed"
        CMP     DL,'$'          ;Last byte of decoded code?
        JE      STOP            ;Yes, go to end of program
        INC     DI              ;No, point to next decoded byte
        INT     21H             ;Call the interrupt routine
        JMP     OUTPUT          ;Display next character
;
CLEAR_ZENITH:                   ;Subroutine to clear the screen and
                                ;home the cursor on a Zenith system
        LEA     DX,CLEAR_SCREEN ;Get the address of the code to
                                ;clear screen and home cursor
        MOV     AH,9            ;Load interrupt "send character
                                ;string" command
        INT     21H             ;Call the interrupt routine
        RET                     ;Return from subroutine call
;
CLEAR_IBM:                      ;Subroutine to clear the screen and
                                ;home the cursor on an IBM PC
        MOV     AH,6            ;Load interrupt VIDEO_IO "scroll
                                ;active page up" command
        MOV     AL,0            ;Number of lines blanked at bottom
                                ;of display window
        MOV     CX,0            ;Address of first display byte
        MOV     DX,1950H        ;Address of last display byte
        MOV     BH,7            ;Normal display attributes
        INT     10H             ;Call video interrupt routine
        MOV     AH,2            ;Load interrupt VIEDO_IO "cursor
                                ;position" command
        MOV     DX,0            ;Cursor address location, make zero
                                ;to home the cursor
        MOV     BH,0            ;Make display page number zero
        INT     10H             ;Call video interrupt routine
        RET                     ;Return from subroutine call
;
```

Figure 6-9A
Program to illustrate data transfer and translation.

```
STOP:   MOV     AH,0            ;Load interrupt "program terminate,
                                ;return to monitor" command
        INT     21H             ;Call the interrupt routine
;
CLEAR_SCREEN    DB      1BH,'E',1BH,'H','$' ;Code to clear screen
                                ;and home cursor on Zenith system
TABLE   DB      'ADEFHILNORST $';Code translation table
CODE    DB      0BH,4,5,0AH,0CH,5,0AH,0CH,0,0CH,0BH,2,0AH,0BH,0CH
        DB      0BH,9,0,7,0AH,6,0,0BH,5,8,7,0DH ;Coded message
DECODE  DB      50 DUP (' ')    ;Storage for the decoded message
;
COM_PROG  ENDS
        END     START
```

Figure 6-9B
Continuation of the program to illustrate data transfer and translation.

4. Single-step through the LAHF instruction. The AH register is loaded with the contents of the low-order byte of the Flag register. Your register contains the value _ _ H. In binary, this is _ _ _ _ _ _ _ _ B. Since the debugger starts a program with the Flag register bits cleared to zero, your AH register should contain binary the value:

   00X0X0X0

   where the Xs represent undefined bits. Thus, your register can contain any value from 0 to 2AH, depending on the state of those undefined bits.

5. The AX register contains the value _ _ _ _ H. Single-step through the next instruction. The AX register now contains the value _ _ _ _ H. The AH and AL register contents have been exchanged.

6. The Flag register bit values are _ _ _ _ _ _ _ _ _. Single-step through the next instruction. The Flag register bit values are now _ _ _ _ _ _ _ _ _. The SAHF instruction stored the contents of the AH register (0FFH) into the low-order byte of the Flag register. This set the Sign, Zero, Auxiliary Carry, Parity, and Carry flags to one. The other three flags in the high-order byte remain clear.

7. Single-step through the next three instructions. They load the two base registers and an index register in preparation for the code translation loop.

8. The loop begins by loading the first byte in the array CODE into the AL register. Single-step through the instruction. The AL register contains the value _ _ H.

9. Single-step through the next instruction. The BP register is incremented to point to the next byte in the array CODE.

10. Single-step through the next instruction. The AL register contains the value __ H. The XLAT instruction has used the EA formed by the contents of the BX and AL registers to retrieve a byte of data in memory. Recall that BX contains the base address of the array TABLE. The AL register contents (0BH) serve as an index into the array. Thus, the twelfth byte (the first byte is zero, and 0BH equals decimal 11) is moved into the AL register. That byte is the ASCII code (54H) for the letter "T".

11. Examine offset address 0186H, type "D180" and RETURN. It is the first byte in the array DECODE, and it contains the value __ H. Single-step through the next instruction. Again examine offset address 0186H. The memory location contains the value __ H. This should match the value that was stored in the AL register.

    The program has just stored the first character that will be displayed. Notice that the character was stored in the form of ASCII code. This type of coding is required by the display routine in this program.

12. Single-step through the next instruction. The DI register is incremented to point to the next storage location.

13. The status of the Zero flag is _____. Single-step through the next instruction. It compared the contents of the AL register with the ASCII code for the character "$". The status of the Zero flag is _____. The two ASCII codes do not match, so the flag is clear (NZ). The character "$" is used to indicate the end of the message to be displayed.

14. Single-step through the next instruction. The IP register contains the value ____ H. This is the address of the first instruction in the REPEAT loop. Had the previous compare instruction set the Zero flag, this conditional jump instruction would have been ignored. Since the flag was clear, the jump was taken, causing the loop to repeat. If you wish to observe the operation of the loop a few more times, continue single-stepping. When you've had enough, proceed to Step 15.

15. The rest of the program clears the display, "homes" the cursor to the top left-hand corner of the display, and prints the decoded message on the display. Run the remainder of the program by typing "G" and RETURN. The display will clear and the message "THIS IS A TEST TRANSLATION" will appear at the top of the display. The next line in the display contains the debugger message "Program terminated normally" to tell you the program ran okay, and that you are back under the control of the debugger. The third line in the display contains the debugger "prompt" and cursor.

16. While you are still in the debugger, examine the array DECODE one more time. Type "D180" and RETURN. The array begins at offset address 0186H. The ASCII characters in the message are shown on right side of the display. As you can see, the message uses the first 27 locations in the array, leaving 23 locations empty. Thus, if you wish, you can rewrite your program to print a 50-character message. Just make sure the end of the message is identified with the "$" character.

17. Exit the debugger. Run your program again; type the file name without the extension ".COM" and RETURN. The display will clear and the message "THIS IS A TEST TRANSLATION" will appear on the first line. The next line contains the system monitor prompt and cursor.

Discussion

After the program finished the REPEAT loop, the next instruction called either the CLEAR_ZENITH subroutine or the CLEAR_IBM subroutine. We'll describe the CLEAR_ZENITH subroutine first. Then we'll describe the remainder of the program. Finally, we'll describe the CLEAR_IBM subroutine. Whether you have an IBM or a Zenith system, read the whole "Discussion." The material presented is important for your complete understanding of assembly language.

CLEAR_ZENITH sends two "escape" codes to the system to clear the display and send the cursor to its "home" location. These escape codes are part of a series of codes that can be used to control many functions of the system, including the cursor characteristics, the video display, the programmable function keys, and the characteristics of the displayed characters. All of the escape codes are listed in the Appendix of the manual that came with your Z-DOS system software.

The escape code for "clear display" is **ESC E**, while the code for "home cursor" is **ESC H**. Before these escape codes can be transmitted to the system, they must be converted to their ASCII code values. Normally, you enclose the character, or characters, to be converted within single or double quotation marks and let the assembler do the conversion. However, that won't work with **ESC**. The assembler will think it's converting three individual characters. For that reason, **you** must determine the correct ASCII code and load it into the program as a constant. Figure 6-5A, on Page 6-22, is a listing of the ASCII codes. Notice that columns 1 and 2 of the figure contain special control function codes. (These are explained in Figure 6-5B.) Escape (ESC) is listed in column 2. The ASCII code for ESC is 1BH. (As before, you can assume that the eighth bit in an ASCII code byte is zero.) Thus, the assembler directive statement to clear the display and home the cursor can be written:

```
DB    1BH,'E',1BH,'H'
```

While you must "tell" the assembler the ASCII code for special control functions, like escape, it's always a good idea to let the assembler perform the conversion for normal alphanumeric characters and symbols. That cuts down the chances for error in your program.

The escape codes are sent to the system using another variation of the interrupt 21H command. Recall that interrupt 21H, **function 0** returns the microprocessor to the system program. Subroutine CLEAR_ZENITH uses interrupt 21H, **function 9** to send a "character string" containing the escape codes to the display. A character string transmitted by function 9 can contain printable or nonprintable ASCII characters. Your decoded message contained printable characters. Special control functions and their supporting character codes are nonprintable characters. Thus, when the "clear display" and "home cursor" commands are transmitted, they will not be displayed.

To execute the interrupt 21H, function 9 command, the subroutine CLEAR_ZENITH first loads the effective address of the escape codes stored in memory into the DX register. (You must load the address of the character string into the **DX register** for the interrupt to function properly.) The address is identified by the name CLEAR_SCREEN. Notice that, along with the escape codes, the CLEAR_SCREEN define byte statement also contains the character "$". This is used by the interrupt routine to identify the **end** of the character string. The next instruction loads the AH register with the function number for the "send character string" interrupt. The third instruction executes the interrupt, sending the escape codes to the display. The system interprets the escape codes, clears the display, and homes the cursor. Finally, the return from interrupt instruction loads the Instruction Pointer register with the offset address of the instruction following the subroutine call.

With the screen cleared, the program is ready to send the decoded message. We could have used the "send character string" interrupt a second time, but we wanted you to see how the "display character" interrupt is used. Interrupt 21H, **function 2**, sends a single ASCII character to the display each time the interrupt is executed. The character must be in the DL register.

The two instructions immediately after the subroutine call prepare the MPU for the character OUTPUT loop. The first instruction loads the interrupt function number (2) into the AH register. Then the starting address of the decoded message is loaded into the DI register.

Now the loop begins. The first instruction loads the first byte of the message into the DL register. The byte is then tested to see if it is the "$" character. Since it isn't, the conditional jump is ignored, and the DI register is incremented to point to the next message byte. The next instruction sends the character in the DL register to the display. Finally, the unconditional jump forces the loop to repeat. When the last character in the message, the "$" character, is loaded into the DL register, the conditional jump if equal instruction sends the MPU to the end of the program. This location is identified by the label STOP. Interrupt function zero is loaded into the AH register, and the interrupt 21H command is executed. This sends the MPU back to the system monitor. The three interrupt 21H commands used in this program, plus all of the other interrupt 21H commands, are described in the Appendix of both the IBM and Zenith system DOS manuals.

Now that you are familiar with the program, let's look at the subroutine for IBM and IBM compatible systems. First, we didn't use a different subroutine because IBM doesn't support interrupt 21H, function 9 — it does. What IBM doesn't support are the Zenith escape codes. This is virtually true on all microcomputer systems. Everybody has their own way of doing things.

IBM controls its system characteristics through a number of BIOS routines in its Read Only Memory (ROM) monitor. A listing of the BIOS ROM code is provided in the appendix of the IBM Technical Reference manual. The procedure used to clear the screen and home the cursor is listed under the category INTerrupt 10 VIDEO_IO.

IBM clears its display by scrolling blank lines onto the screen using an interrupt 10, function 6 command. Therefore, the first instruction in the subroutine CLEAR_IBM loads the AH register with the function number 6. The next code loaded into the AL register identifies the number of blank lines at the bottom of the "display window." Since the whole display will be blank, this value can be zero. Next, the CX register is loaded with the row (CH) and column (CL) address of the first byte of the display area being scrolled. Since that is the first byte in the display, the CX register is loaded with zero. Then the DX register is loaded with the row (DH) and column (DL) of the last byte in display being scrolled. Assuming a 25-row by 80-column display, the value 1950H is loaded into the DX register. Finally, the BH register is loaded with the display attributes. These determine the foreground and background color in the display and whether the cursor is blinking or non-blinking. The value 7 produces a normal display. Again, the Technical Reference manual describes all of the attribute variations for an IBM color or monochrome display. After all of the control codes have been loaded into the appropriate registers, the interrupt 10 command is executed and the display is blanked.

After the display is blanked, the subroutine moves the cursor to the top left-hand corner of the display. First, the DX register is loaded with the desired cursor location. "Home" is at row (DH) zero and column (DL) zero. The next instruction identifies the display "page" number. Depending on the display resolution, video memory can store up to eight displays worth (pages) of data. Identifying the page number tells the system what byte the cursor is pointing to in video memory. For this program, we are pointing at the first byte in the first page of video memory; hence the value zero loaded into the BH register. Executing the interrupt 10 instruction moves the cursor to its "home" location. The last instruction in the subroutine returns the MPU to the main program.

Now that you've had a chance to exercise the transfer and translation instructions, let's review the bit manipulation instructions. As you discovered earlier, many of the conditional instructions rely on various logical operations to determine the outcome of the response. Since logical operations are an integral part of programming, all of the basic types are included in the 8088/8086 MPU instruction set. The next program will show you how all of the logical instructions operate. In addition, the program will present a cross section of the shift and rotate instructions.

Procedure Continued

18. Call up the editor and enter the program listed in Figure 6-10. Assemble, link, and convert the program into a COM file.

19. Call up the debugger and load your program COM file. Single-step through the program and answer the following questions. Use the Instruction Pointer register value as a guide to the questions. Note that some of the program instructions simply prepare the MPU for a particular operation. These have been ignored in the questions.

IP = 010AH

AX = ____H.
BX = ____H.
Flags = _ _ _ _ _ _ _ _.
OD I S Z A P C
Address 0141H = __H, low byte of AX.
Address 0142H = __H, high byte of AX.

```
TITLE EXPERIMENT 6 -- PROGRAM 2 -- LOGICALS, SHIFTS, AND ROTATES
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
;
;LOGICAL INSTRUCTIONS
;
START:  MOV     AX,0FF00H       ;Fill register half ones, half zeros
        LEA     BX,SAVE         ;Get the effective address
        MOV     SAVE,AX         ;Store the register for later use
        AND     AX,0FF0H        ;Mask the first and last four bits
        OR      WORD PTR [BX],0FF0H ;Add 0F0H to low byte at SAVE
        XOR     AX,0FF0H        ;Zero high byte, add 0F0H to low byte
        NOT     WORD PTR [BX]   ;Invert the bits at SAVE
        MOV     CX,0FFFFH       ;Fill register with ones
        PUSH    CX              ;Save register in stack
        POPF                    ;Load ones in Flag register bits
        TEST    [BX],AX         ;Update the flags
;
;SHIFT INSTRUCTIONS
;
        MOV     CL,7            ;Load a shift count
        SHL     AX,1            ;Shift logic left one bit
        SAL     AX,CL           ;Shift arithmetic left count bits
        SHR     AH,CL           ;Shift logic right count
        SAR     AH,1            ;Shift arithmetic right one bit
        MOV     AX,8080H        ;Set high bit of high and low bytes
        SAR     AX,CL           ;Shift arithmetic right count bits
;
;ROTATE INSTRUCTIONS
;
        SUB     CX,CX           ;Zero the register
        PUSH    CX              ;Save register in stack
        POPF                    ;Load zeros in Flag register bits
        MOV     CL,8            ;Load rotate count
        MOV     DX,[BX]         ;Get the value at SAVE
        ROL     DX,CL           ;Rotate left count
        ROR     DX,CL           ;Rotate right count
        RCR     DX,CL           ;Rotate through carry right count
        RCL     DX,1            ;Rotate through carry left one bit
        RCR     AX,1            ;Rotate through carry right one bit
        CLC                     ;Clear Carry flag
        RCL     AX,1            ;Rotate through carry left one bit
;
SAVE    DW      0               ;Initialize one word to zero
;
COM_PROG  ENDS
        END     START
```

Figure 6-10
Program to illustrate the operation of the bit manipulation instructions.

IP = 010DH AX = ____H.
Flags = _ _ _ _ _ _ _ _ _.
OD I S Z AP C

IP = 0111H Address 0141H = __H.
Address 0142H = __H.
Flags = _ _ _ _ _ _ _ _ _.
OD I S Z AP C

IP = 0114H AX = ____H.
Flags = _ _ _ _ _ _ _ _ _.
OD I S Z AP C

IP = 0116H Address 0141H = __H.
Address 0142H = __H.
Flags = _ _ _ _ _ _ _ _ _.
OD I S Z AP C

IP = 011BH CX = ____H.
Flags = _ _ _ _ _ _ _ _ _.
OD I S Z AP C

IP = 011DH AX = ____H.
Flags = _ _ _ _ _ _ _ _ _.
OD I S Z AP C

IP = 0121H AX = ____H.
Flags = _ _ _ _ _ _ _ _ _.
OD I S Z AP C

IP = 0123H CL = __H.
AX = ____H.
Flags = _ _ _ _ _ _ _ _ _.
OD I S Z AP C

IP = 0125H AX = ____H.
Flags = _ _ _ _ _ _ _ _ _H.
OD I S Z AP C

IP = 0127H AX = ____H.
Flags = _ _ _ _ _ _ _ _ _.
OD I S Z AP C

IP = 012AH AX = _ _ _ _H.

IP = 012CH AX = _ _ _ _H.
Flags = _ _ _ _ _ _ _ _ _.
OD I S Z AP C

IP = 0130H CX = _ _ _ _H.
Flags = _ _ _ _ _ _ _ _ _.
OD I S Z AP C

IP = 0134H CL = _ _H.
DX = _ _ _ _H.

IP = 0136H CL = _ _H.
DX = _ _ _ _H.
OF = _.
CF = _.

IP = 0138H DX = _ _ _ _H.
OF = _.
CF = _.

IP = 013AH DX = _ _ _ _H.
OF = _.
CF = _.

IP = 013CH DX = _ _ _ _H.
OF = _.
CF = _.
AX = _ _ _ _H.

IP = 013EH AX = _ _ _ _H.
OF = _.
CF = _.

IP = 013FH OF = _.
CF = _.

IP = 0141H AX = _ _ _ _.
OF = _.
CF = _.

Discussion

With a few exceptions, all of the steps in the program are self-explanatory. To get a good cross-section of instruction types, the program manipulates data in both the MPU registers and in memory. If you have any problems understanding a particular bit pattern after an operation, try converting the hex value to binary.

The OR and NOT instructions required the assembler operator WORD PTR to help the assembler determine the data size.

To illustrate a point, we had to set all of the Flag register bits, and later clear all of the Flag register bits. To do this, we stored the necessary data in the CX register, pushed the register contents into the stack, and immediately popped the data into the Flag register. You saw the first occurrence when the IP was at 011BH.

Did you remember that, when you execute a logic or shift instruction, some of the Flag bits will either be set, cleared, or undefined? Also, the only flags affected by a rotate instruction are CF and OF; the other flags are undefined. In addition, OF is undefined after a multiple bit rotate.

The remaining programs in this experiment will show how the various arithmetic instructions can be used.

Procedure Continued

20. Call up the editor and enter the program in Figure 6-11. Assemble, link, and convert the program to a COM file.

```
TITLE EXPERIMENT 6 -- PROGRAM 3 -- ADDING WITH CARRY, BCD NUMBERS
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  MOV     CX,6            ;Set the loop count
        CLC                     ;Clear flag for first add with carry
        LEA     BX,NUMBER1      ;Get address of first BCD number
        LEA     BP,NUMBER2      ;Get address of second BCD number
        LEA     SI,SUM          ;Get address of storage for BCD sum
COUNT:  MOV     AH,0            ;Prepare register to hold overflow
        MOV     AL,[BX]         ;Get one byte of first number
        ADC     AL,[BP]         ;Add one byte of second number
                                ;to first number with carry
        AAA                     ;Adjust BCD sum in AL register
        MOV     [SI],AL         ;Save the adjusted BCD
        INC     BX              ;Point to next first BCD number
        INC     BP              ;Point to next second BCD number
        INC     SI              ;Point to next storage for BCD sum
        LOOP    COUNT           ;Any more BCD numbers?
        ADD     [SI],AH         ;Save overflow as most sig. BCD number
        INT     3               ;Return to debugger
;
NUMBER1 DB      6,6,6,6,6,6     ;Least significant number first
NUMBER2 DB      8,8,8,8,8,8     ;Least significant number first
SUM     DB      7 DUP (0)       ;Reserve seven locations for sum,
                                ;store least significant number first
;
COM_PROG  ENDS
        END     START
```

Figure 6-11
Adding two multibyte BCD numbers together.

21. Load your debugger and program COM file. Single-step through the program until the program loops back to COUNT (offset address 0110H). Answer the following questions.

IP = 0114H AL = __H.

IP = 0117H AL = __H.
CF = _.

IP = 0118H AL = __H.
AH = __H.
CF = _.
Address 012EH = __H.

IP = 011AH Address 012EH = __H.

IP = 0110H AL = __H.
AH = __H.
CF = _.

22. Let the debugger run the remainder of the program — type "G" and RETURN. Answer the following questions.

    Address 012EH = __H, the least significant BCD of the sum.
    Address 012FH = __H.
    Address 0130H = __H.
    Address 0131H = __H.
    Address 0132H = __H.
    Address 0133H = __H.
    Address 0134H = __H, the most significant BCD of the sum.

Discussion

Your program operates in a fashion similar to a six-digit calculator. Memory arrays NUMBER1 and NUMBER2 are the six-digit "input registers." To keep the program simple, we had you load them with BCD values prior to running the program. The program then adds the two six-digit numbers together, one byte at a time, and stores the result in a third memory array called SUM. An additional byte was reserved in SUM to store any overflow from the add-with-carry operation.

To accomplish its task, the program uses the loop instruction to add the six BCD together. Had there been less than six digits, the remaining memory locations would have been loaded with zeros. Once the loop count was set, the Carry flag was cleared to make sure a carry isn't added to the first number pair. Then the offset address to each of the memory arrays was loaded into the MPU. Finally, the loop was executed.

Procedure Continued

23. Exit the debugger and call up the editor. Change the value in NUMBER1 to the number string '666666' and the value in NUMBER2 to the number string '888888'. This will cause the **ASCII code** for each number to be loaded into the memory arrays, instead of the BCD value stored earlier.

24. Assemble, link, and convert the program to a COM file. Load the debugger and your COM file, and execute the program by typing "G" and RETURN. Record the values found at the following addresses.

    Address 012EH = __H.
    Address 012FH = __H.
    Address 0130H = __H.
    Address 0131H = __H.
    Address 0132H = __H.
    Address 0133H = __H.
    Address 0134H = __H.

    Compare these values to the values recorded in Step 22. Do they match? Unless you used the wrong values in your program, they should match. This verifies that the ASCII adjust for add can accommodate both BCD and ASCII coded number values.

25. Now that you have an idea of how to add multibyte numbers, write a program similar to the "addition" program that will subtract one multibyte ASCII number from another. Assume that NUMBER1 will always be greater than NUMBER2. Use the value '666888' for NUMBER1 and the value '888666' for NUMBER2 (remember, the **least significant byte** is the first byte in each of these values). Try not to look at Figure 6-12 until you have your program working.

Discussion

Figure 6-12 shows one way you could have written your subtraction program. Notice that it is quite similar to the preceding addition program. In fact, in addition to the data, only three instructions were changed. Add with carry is now subtract with borrow; ASCII adjust for add is now ASCII adjust for subtract; and finally, the instruction that added the carry in the AH register is now a simple move operation. The last change insures that, if by mistake a larger number is subtracted from a smaller number, an indication of a "borrow" is saved as the most significant byte of the difference.

Recall that when AL contains a value greater than 9, **or** the Auxiliary Carry flag is set, a "borrow" has occurred. This causes the ASCII adjust for subtraction instruction to, among other things, subtract 1 from the AH register. Since the AH register is zero prior to executing AAS in this program, subtracting 1 leaves the difference 0FFH. Thus, 0FFH is stored as the most significant byte of the difference to indicate a subtraction error. Naturally, if no borrow occurred, the AH register remains zero and zero is stored to indicate no subtraction error.

The next program you will examine multiplies one multibyte number by another. Because of the complexity of the operation, it will provide good examples of both the multiply and divide instructions. To keep the program relatively simple, we limited the size of the numbers that can be multiplied to three digits, and the product to six. The first half of the program converts and adds the ASCII number sets to produce two 16-bit numbers that can be easily multiplied. After the numbers are multiplied, the last half of the program translates the 32-bit product into BCD digits that represent the value of the product.

```
TITLE EXPERIMENT 6 -- PROGRAM 4 -- SUBTRACTING ASCII NUMBERS
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  MOV     CX,6            ;Set the loop count
        CLC                     ;Clear Carry (borrow) flag for
                                ;first subtract with borrow
        LEA     BX,NUMBER1      ;Get address of first (larger) number
        LEA     BP,NUMBER2      ;Get address of second (smaller) number
        LEA     SI,DIFFERENCE   ;Get address of difference storage
COUNT:  MOV     AH,0            ;Prepare register to hold borrow
        MOV     AL,[BX]         ;Get one byte of first number
        SBB     AL,[BP]         ;Subtract one byte of second number
                                ;then subtract one if CF set
        AAS                     ;Adjust difference in AL register
        MOV     [SI],AL         ;Save the adjusted byte
        INC     BX              ;Point to next first number byte
        INC     BP              ;Point to next second number byte
        INC     SI              ;Point to next storage for difference
        LOOP    COUNT           ;Any more bytes?
        MOV     [SI],AH         ;Flag any extra borrow in the most
                                ;significant byte of the difference
        INT     3               ;Return to debugger
;
NUMBER1 DB      '666888'        ;Least significant byte first
NUMBER2 DB      '888666'        ;Least significant byte first
DIFFERENCE  DB  7 DUP (0)       ;Reserve six locations for difference
                                ;store least significant byte first
                                ;reserve most significant byte
                                ;for extra borrow indicator
;
COM_PROG  ENDS
        END     START
```

Figure 6-12
Program to subtract one multibyte ASCII number from another.

Procedure Continued

26. Call up the editor and enter the program in Figure 6-13. Assemble, link, and convert the program to a COM file.

27. Load the debugger and your program COM file. Single-step through the program to address 0121H. As you execute each instruction, observe the results in AX, BX, CX, BP, and DI registers. The AX register contains the value ____H, while the SI register contains the value ____H.

28. Now let the debugger run the second conversion loop. Type "G11E" and RETURN. The AX register contains the value ____H, while the SI register contains the value ____H.

    The AX register doesn't appear to have changed, but that's only because both NUMBER1 and NUMBER2 are identical. The SI register contains the value that was left in the AX register after the first conversion loop.

29. Single-step to address 0134H. The DX register contains the value ____H, while the AX register contains the value ____H. These are the high and low words of the doubleword product.

30. That completes the multiplication portion of the program. The rest of the program converts the doubleword product to BCD. Run the program to completion — type "G" and RETURN. Record the contents of the following addresses.

    0161H = __H, least significant byte of six-digit product.
    0162H = __H.
    0163H = __H.
    0164H = __H.
    0165H = __H.
    0166H = __H, most significant byte of six-digit product.

```
TITLE EXPERIMENT 6 -- PROGRAM 5 -- MULTIBYTE MULTIPLICATION
COM_PROG  SEGMENT
        ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
        ORG     100H
START:  SUB     BH,BH           ;Clear register for later compare
        MOV     DI,10           ;Load conversion constant
        MOV     CX,3            ;Set conversion count
        LEA     BP,NUMBER1+2    ;Get address of most significant byte
        SUB     AX,AX           ;Get ready for conversion
CONVERT:MOV     BL,[BP]         ;Get byte
        AND     BL,0FH          ;Translate ASCII to BCD
        ADD     AL,BL           ;Add BCD to accumulator
        DEC     CX              ;Count down
        JZ      SECOND          ;Done, start next conversion
        MUL     DI              ;Adjust BCD value, multiply AX times DI
        DEC     BP              ;Point to next byte
        JMP     CONVERT         ;Do it again
SECOND: CMP     BH,0            ;Is second conversion done?
        JNZ     HERE            ;Yes, go multiply numbers
        MOV     SI,AX           ;Store first conversion number
        SUB     AX,AX           ;Get ready for next conversion
        INC     BH              ;Set flag for second conversion
        MOV     CX,3            ;Set count for conversion
        LEA     BP,NUMBER2+2    ;Get address of most significant byte
        JMP     CONVERT         ;Start second conversion
HERE:   MUL     SI              ;Multiply AX and SI registers together
        LEA     BP,PRODUCT      ;Get address for storage of product
        MOV     BX,1000         ;Set number partition constant
        DIV     BX              ;Partition double-word value
        MOV     BX,100          ;Set correction constant
        PUSH    AX              ;Save high word for later
        XCHG    AX,DX           ;Get low word for conversion
DECODE: SUB     DX,DX           ;Zero high word for double-word divide
        DIV     DI              ;Make BCD conversion
        MOV     [BP],DL         ;Store BCD product
        INC     BP              ;Point to next storage location
        POP     DX              ;Get high word saved earlier
        PUSH    AX              ;Save low word for later
        XCHG    AX,DX           ;Get high word for translation
        SUB     DX,DX           ;Zero high word for double-word divide
        DIV     DI              ;Translate high word
        XCHG    AX,DX           ;Get remainder
        POP     CX              ;Get low word and set aside
        PUSH    DX              ;Save what's left of high word
        MUL     BX              ;Shift remainder value
        ADD     AX,CX           ;Restore low word validity
        JNZ     DECODE          ;Conversion to BCD done?
        INT     3               ;Yes, return to debugger
;
NUMBER1 DB      '999'           ;Least significant byte first
NUMBER2 DB      '999'           ;Least significant byte first
PRODUCT DB      6 DUP (0)       ;Set aside 6 bytes for the product
                                ;Least significant byte first
;
COM_PROG  ENDS
        END     START
```

Figure 6-13
Program to multiply one multibyte number by another.

Discussion

The first pass through the conversion routine takes the ASCII values stored at NUMBER1 and compresses them into a 16-bit value. The process involves converting the ASCII values to BCD, and then shifting and adding the digits. Since we are shifting decimal numbers, the shift instruction won't work. Therefore, we first take the most significant digit and multiply it by ten. Then we add the next digit to the product. Again we multiply that value by ten. Finally, we add the last digit to the product. Had there been more digits, we would simply continue the process of shifting and multiplying. The result of the operation converts the first digit, 9, to 900; the second digit, 9, to 90; and the last digit, 9, remains 9. The total, 999, is in the AX register at the end of the process. Naturally, the value you recorded was 03E7H, the hexadecimal equivalent of decimal 999.

When you ran the program to address 011EH, the first product was stored in the SI register, and the second number, NUMBER2, was converted and stored in the AX register. As you single-stepped to address 0134H, the two numbers were multiplied together and the product was stored in the DX (high word) and AX (low word) registers, 000FH and 3A71H respectively. A word multiply will always produce a doubleword result, even if you multiply one times one. By the same token, a byte multiply produces a word result, with register AH the high byte and register AL the low byte.

The last half of the program converted the doubleword product to BCD digits. Unfortunately, we couldn't simply reverse the process we used earlier to compact the digits. Recall that when you divide a 32-bit number by a 16-bit number, the quotient is stored in the AX register and the remainder is stored in the DX register. If the quotient is larger than 16 bits, the MPU generates a type 0 interrupt. Therefore, we couldn't divide the product, generated earlier, by ten and store the remainder as a BCD digit. The quotient is too large. Instead, we must first reduce the product to a size that, when divided by ten, will produce a 16-bit or smaller value. Figure 6-14 shows how this operation is performed. To make this process easier to follow, we'll use the product calculated by the program.

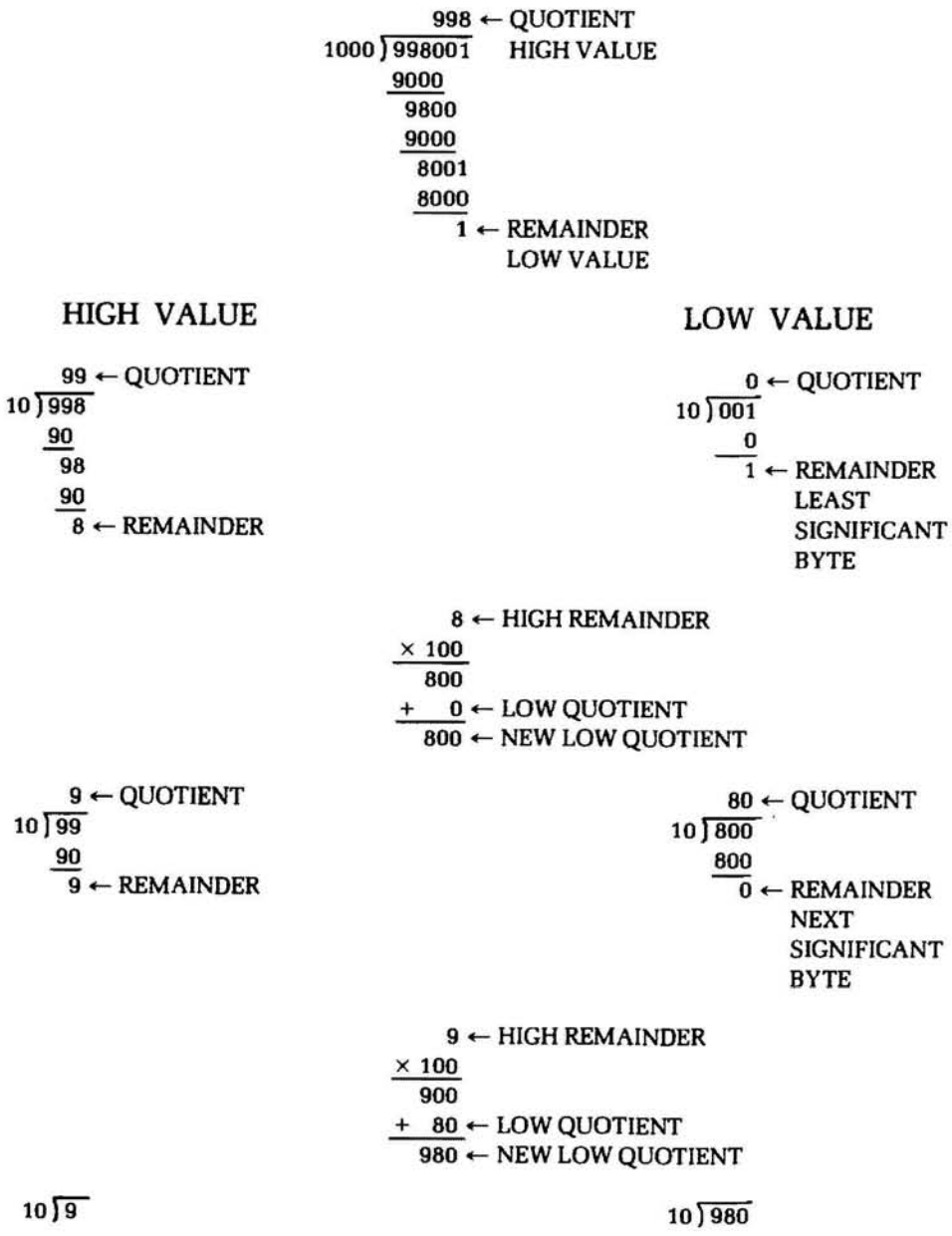


Figure 6-14
Determining the BCD digits
for a large number.

First, the product is divided by 1000. The quotient from that division is considered the "high value," while the remainder is considered the "low value." Now, to determine the first, or least significant byte, the low value is divided by 10 and the remainder saved as the first BCD. Next, we divide the high value by 10. The remainder is 8. However, it has a true value of 800 because of the earlier splitting of the original value. Thus, we add 800 to the low quotient, 0, to produce a new low quotient.

Now we repeat the process. Dividing 800 by 10 gives a remainder of 0, the next significant byte stored, and a quotient of 80. Likewise, dividing 99 by 10 gives a remainder of 9 with a quotient of 9. Since the remainder 9 again has a true value of 900, 900 is added to the low quotient of 80 to produce the new low quotient 980. The process continues until both the high value and low value are reduced to zero. All of the remainders that were stored from the low value division operation represent the decimal value of the original product.

That is how the program converts the product to BCD form. Because the AX and DX registers must be used for each division operation, the stack is used a number of times to save the various high and low value quotients. After the initial dividing of the product, the BX register holds the constant 100 that is used to convert the high value remainder to its true value. The DI register holds the constant 10 that is used to generate the high and low values remainders. Note that the least significant byte of the product is stored at address 0161H, and the high byte is stored at address 0166H.

This completes the Experiment for Unit 6. Proceed to the Unit 6 Examination.

UNIT 6 EXAMINATION

1. Which of the following instructions is used to move the contents of the AX register to the BX register, and at the same time move the contents of the BX register to the AX register?

   A. MOV.
   B. XCHG.
   C. SAHF.
   D. XLAT

2. Which of the following instructions is used to load flags into the AH register?

   A. PUSHF.
   B. POPF.
   C. SAHF.
   D. LAHF.

3. The DAA instruction is used to:

   A. Pack unpacked decimal numbers.
   B. Prepare the AX register for a packed decimal add.
   C. Compress the contents of the AX register into the AL register.
   D. Adjust the result of a packed decimal add.

4. Which of the following instructions lets you subtract only the contents of the AL register from the contents of the BL register?

   A. AAS.
   B. SUB.
   C. SBB.
   D. DAS.

5. The DAS instruction is used to:

   A. Adjust the result of a packed decimal subtraction.
   B. Pack unpacked decimal numbers.
   C. Compress the contents of the AX register into the AL register.
   D. Prepare the AX register for a packed decimal subtraction.

6. Which of the following instructions automatically extends the sign of the result into the high-order register if the result only occupies the low-order register?

   A. MUL.
   B. IMUL.
   C. CBW.
   D. AAM.

7. Which of the following instructions is used for an integer divide instruction?

   A. CBW.
   B. DIV.
   C. IDIV.
   D. AAD.

8. All of the following flags are updated by a logical operation, except:

   A. OF.
   B. CF.
   C. SF.
   D. AF.

9. The "exclusive OR" operation returns a one to the destination operand bit if:

   A. Either or both corresponding bits of the original operands are set.
   B. The corresponding bits of the original operands contain opposite values.
   C. The corresponding bits of the original operands are set.
   D. The corresponding bits of the original operands are identical.

10. Which of the following instructions perform the same operation as the instruction SHL?

    A. SHR.
    B. SAR.
    C. SAL.
    D. SBB.

11. Which of the following flags is undefined after a single-bit shift operation?

    A. AF.
    B. PF.
    C. CF.
    D. OF.

12. In a rotate operation, the CL register can be used to hold the count. What is the maximum count value that can be specified?

    A. 256.
    B. +127.
    C. 255.
    D. −128.

13. In a single-bit rotate, which instruction stores the value of CF in the least significant bit of the destination operand?

    A. ROR.
    B. ROL.
    C. RCR.
    D. RCL.

EXAMINATION ANSWERS

1. B — The XCHG instruction is used to swap the contents of two registers.

2. D — The instruction LAHF is used to load the lower eight bits of the Flag register into the AH register.

3. D — The DAA instruction is used to adjust the result of a packed decimal add.

4. B — The instruction SUB lets you subtract the contents of the AL register from the contents of the BL register.

5. A — The DAS instruction is used to adjust the result of a packed decimal subtraction.

6. B — The IMUL instruction automatically extends the sign of the result into the high-order register if the result only occupies the low- order register.

7. C — The instruction IDIV is used for an integer divide operation.

8. D — All of the following flags are updated by a logical operation except AF.

9. B — The "exclusive OR" operation returns a one to the destination operand if the corresponding bits of the original operands contain opposite values.

10. C — The SAL instruction performs the same operation as the SHL instruction.

11. A — AF is undefined after a single-bit shift operation. OF is only undefined after a multibit shift operation.

12. C — The maximum count that can be stored in the CL register is 255.

13. D — In a single-bit rotate, the instruction RCL will store the value of CF in the least significant bit of the destination operand.

SELF-REVIEW ANSWERS

1. **False.** The PUSHF instruction is used to store a **word** of data in the "stack."

2. **False.** The XCHG instruction "swaps" the contents of the source and destination operands. The XLAT instruction replaces a byte in the AL register with a byte from a 256-byte code table whose address is contained in the BX register.

3. If you wish to update SF, ZF, AF, and CF to known values, you should use the flag transfer instruction **SAHF**.

4. The LEA instruction is used to move the effective address of a variable name into a **16-bit general** register.

5. If the decimal equivalent of the unsigned binary number is 25, the signed binary number is **+25**. Since the sign of the number is positive, the number is not complemented; hence the signed and unsigned values are the same.

6. If the decimal equivalent of the 8-bit unsigned binary number is 153, the signed binary number is **−103**. Because the unsigned binary number has a one in its high-order bit, you must take its 2's complement to determine the signed value.

7. If the unsigned binary number is 8, the unpacked decimal number is **8**. Because the unsigned binary number is less than 0AH, it has the same value as the unpacked decimal.

8. If the unsigned binary number is 103, the packed decimal number is **67**. Here, you must break the unsigned binary number into two groups of four bits:

   103 = 0110 0111

   Since each group has a value less than 0AH, you can convert each one to its BCD equivalent. Had the lower four bits been greater than 9, you would have to add 6 to that value and then add one to the upper 4-bit value. Carrying the process one step further, if the upper 4-bit value is now greater than 9, then you would have to add 6 to these four bits and then set the Carry flag. Setting CF would indicate you have the value one in the 100's digit.

9. The largest negative decimal number that can be represented in eight binary bits is **−128**.

10. The instruction for adding signed binary words between registers is **ADD**.

11. After you add two unpacked decimal numbers, you must adjust the result with the **AAA** instruction.

12. For a decimal adjust operation, the operand must reside in the **AL** register.

13. In unpacked decimal adjusts, the 10's digit is stored in the AH register; while in packed decimal adjusts, the 100's digit is stored in the **Carry flag**.

14. The **ADC** instruction lets you add multiword (or byte) unsigned binary numbers.

15. The instruction for subtracting binary words between registers is **SUB**.

16. **True.** The AAS instruction provides the necessary adjustment to allow you to subtract one ASCII number code from another.

17. The **SBB** instruction lets you subtract more than one multibyte operand from another.

18. **False.** While the compare instruction does subtract the source operand from the destination operand, it does not store the result in the destination operand. Rather, it is used to update the condition code flags without changing the destination operand.

19. **True.** Negating a number produces its 2's complement.

20. The instruction for multiplying two unsigned binary words is **MUL**.

21. The instruction for multiplying two signed binary bytes is **IMUL**.

22. If you multiply one word times another, the answer is stored in the **DX** and **AX** registers.

23. Another term for signed number is **integer**.

24. If you multiply the byte 2 times the byte −5, the number stored in the AH register is **FFH.**

25. **False**. The AAM instruction will only adjust the product of two valid BCD numbers.

26. If you divide a doubleword value by a word value, the dividend must reside in the **DX** and **AX** registers.

27. The quotient from a byte divide is stored in the **AL** register.

28. **False.** If the quotient from a word divide exceeds the capacity of the AX register, a type 0 interrupt will occur and the contents of the AX and DX registers will be undefined.

29. The mnemonic for an integer divide word is **IDIV**.

30. The maximum negative remainder after an integer divide byte operation is **−128.**

31. The **CBW** instruction is used to extend the sign of a byte-sized number before the divide operation.

32. You would use the **CWD** instruction to convert a word to a doubleword.

33. **True.** The AAD instruction is performed prior to the division of two unpacked decimal numbers.

34. **False.** The TEST instruction uses a logical AND process to update the condition code flags.

35. After a "logical" operation, the Auxiliary Carry flag is **undefined**.

36. **False.** The NOT operation forms the **1's** complement of the destination operand.

37. The shift operation lets you shift up to **255** bits out of the destination operand.

38. The **CL** register is used to hold the shift "count."

39. The **SAR** instruction preserves the original sign of the operand during a shift operation.

40. **True.** All of the rotate instructions save the value of the last bit rotated out of the operand in the Carry flag.

41. If the AL register contains 0FH, CF is set, and OF is cleared. Then the AL register will contain **1EH** after the instruction

ROL AL,1

is executed. The Carry flag value is not rotated into the destination operand in a ROL instruction. This will only happen in a RCL instruction.

INSERT