

*Unit 7*

**SEGMENTED MEMORY  
AND I/O**

## CONTENTS

Introduction .....	7-3
Unit Objectives .....	7-4
Unit Activity Guide .....	7-5
EXE Programs .....	7-6
Intra And Intersegment Addressing .....	7-23
I/O Addressing .....	7-30
Experiment .....	7-36
Unit 7 Examination .....	7-71
Examination Answers .....	7-73
Self-Review Answers .....	7-75

## INTRODUCTION

Unit 2 introduced you to the concept of COM and EXE program structures. Thereafter, all of the programming examples were limited to COM-type programs. Because COM-type programs are a carry-over from early generation microprocessors like the Intel 8080, they are limited to a 64K area of memory. As a result, they are very easy to structure — all of the segments lie within the same 64K of memory — and they appear to be composed of one unique memory segment. Creating a program composed of truly individual memory segments requires that you use the EXE-type of program structure.

With this type of program, you can address all one megabyte (1MB) of the memory locations accessible through the 8088/8086 MPU. Naturally, the program structure is more complex than that of a COM program. This unit will cover all of the details of writing various types of EXE programs.

While on the subject of addressing all of memory, this unit will also show you how to address the associated Input/Output (I/O) ports. While most I/O addressing is best handled through system interrupts, like interrupt 21H function 9 (write character string), there will be times when you must directly address a specific I/O port. For that reason, we will describe both fixed port and variable port I/O addressing.

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. Specify the align-type, the combine-type, and 'class' of a segment.
2. Use the intersegment (far) addressing instructions JMP, CALL, and RET.
3. State the purpose of the assembler operators ASSUME, LABEL, and "segment override."
4. Define the following terms: segment, segment base address, logical address, physical address, fixed port, and variable port.
5. State the number of 8-bit I/O ports available to the 8088/8086 MPU.
6. Use the fixed and variable port I/O instructions IN and OUT.
7. Use the assembler directives COMMENT, PUBLIC, and EXTRN.

## UNIT ACTIVITY GUIDE

	<b>Completion Time</b>
<input type="checkbox"/> Begin Reading the Section on "EXE Programs."	_____
<input type="checkbox"/> Complete Self-Review Questions 1-17.	_____
<input type="checkbox"/> Continue Reading the Section on "EXE Programs."	_____
<input type="checkbox"/> Complete Self-Review Questions 18-31.	_____
<input type="checkbox"/> Read the Section on "Intra And Intersegment Addressing."	_____
<input type="checkbox"/> Complete Self-Review Questions 32-43.	_____
<input type="checkbox"/> Read the Section on "I/O Addressing."	_____
<input type="checkbox"/> Complete Self-Review Questions 44-51.	_____
<input type="checkbox"/> Perform the Experiment.	_____
<input type="checkbox"/> Complete the Unit 7 Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

## EXE PROGRAMS

Earlier, we said that the segment address locations in a program are determined by the system DOS “program loader.” The loader identifies what area of memory is available for program use, and then loads the program into the bottom, or low address of that memory. Finally, the loader stores the program starting address into the CS and IP registers.

In COM-type programs, the program loader also stores the Code Segment register contents in the DS, SS, and ES registers. That, however, isn’t the case in EXE-type programs. You, as the programmer, must load the segment registers as required by the program. To begin the discussion of EXE programs, let’s review the basic structure of the program.

### Program Structure

Every EXE program must contain at least two segment areas: the **stack segment** and the **code segment**. Once you’ve established the basic segments, you can add as many additional data, code, and stack segments as necessary to run the program. Figure 7-1 is a simple example of an EXE program. In addition to the stack and code segments, we’ve added a data segment.

```

TITLE UNIT 7 -- PROGRAM 1 -- EXE PROGRAM STRUCTURE
;
PROG_STACK SEGMENT STACK
    DW      16 DUP (0FH)      ;Set up stack area
TOP_OF_STACK LABEL WORD      ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_DATA SEGMENT
DATA1 DB      16 DUP (0BH)    ;Set up byte-sized data area
DATA2 DW      8 DUP (0BBH)    ;Set up word-sized data area
PROG_DATA ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,DS:PROG_DATA,SS:PROG_STACK
START: MOV     AX,PROG_STACK    ;Never load a segment register direct
      MOV     SS,AX            ;Use an intermediate register
      MOV     SP,OFFSET TOP_OF_STACK ;Point to the top of stack
                                      ;immediately after loading SS register
      MOV     AX,PROG_DATA      ;Again, indirectly load the
      MOV     DS,AX            ;segment register
      MOV     BL,DATA1          ;Get the first byte of data
      LEA    BX,DATA2          ;Point to the first word of data
      MOV     DX,[BX]          ;Get the first word of data
      INT     3                ;Return to the debugger
;
PROG_CODE ENDS
      END     START

```

**Figure 7-1**  
EXE program structure.

Every segment in an EXE program must be identified by a unique name followed by the assembler directive `SEGMENT`. In the case of our stack segment, the name is `PROG.STACK`. The argument `STACK` following the `SEGMENT` directive identifies this segment as the stack segment in the program. The end of the segment is identified by the segment name followed by the `ENDS` directive.

Memory space is reserved for the stack by an assembler `Define Word` directive. In this example, 16 words are initialized with the value `000FH`. The end of the stack area is identified by the assembler directive `LABEL`. This tells the assembler to associate the current address offset with the name preceding the `LABEL`. The operator `WORD` completes the identification by telling the assembler that the name relates to word-sized values. Remember, the directive `LABEL` does not reserve a memory location, it simply points to it. The address offset associated with the `LABEL` directive will be used to initialize the Stack Pointer register.

The program data segment is similar to the stack segment. Its beginning is identified by a unique name followed by the `SEGMENT` directive, while its end is identified by the same unique name followed by the `ENDS` directive. An EXE program doesn't need a data segment. You can place program data within the code segment, just as in a COM program. Naturally, the combination of code and data must not exceed the 64K size limitation of a program segment.

The last segment in our program is the code segment. As before, its beginning and end are identified by a unique name followed by the assembler directives `SEGMENT` and `ENDS` respectively. Unlike data, all program code must reside within the code segment. In addition, the code segment must contain the assembler directive `ASSUME`.

Recall that the `ASSUME` directive is a promise to the assembler that all code, data, and stack references will be made to the indicated segments. The `ASSUME` directive in the program in Figure 7-1 indicates that all data is in the `PROG.DATA` segment, all code is in the `PROG.CODE` segment, and all stack references must be made to the `PROG.STACK` segment. Had the data resided within the code segment, then all data references would be made to the code segment name.

Assuring the assembler that code and data reside in specific segments doesn't guarantee that the segment registers will be loaded with the correct addresses. This must be handled by the program. The first five instructions in our sample program load the segment registers.

The first instruction loads the base address of the stack segment into the AX register. This is necessary, since there are no 8088/8086 instructions to directly load a segment register; hence, the intermediate step of loading the AX register with the segment address value. The next instruction then loads the Stack Segment register from the AX register.

The third instruction loads the Stack Pointer register with the offset address to the top of the stack. Actually, the offset address points to the first word **beyond** the last stack memory location. This is because the Stack Pointer is always decremented one word before a value is pushed into the stack. Placing the LABEL directive after the stack allocation statement gives us that one-word offset for the Stack Pointer.

The next two instructions follow the same sequence as the first two. The base address of the data segment is loaded into the AX register. Then the AX register contents are moved into the Data Segment register.

The remaining program instructions simply move data from the data segment into the MPU. They have nothing to do with segment register initialization.

The program didn't have to initialize the Code Segment register or the Instruction Pointer. That process is handled by the DOS program loader. For the same reason, the ORG 100H directive statement is not required at the beginning of an EXE-type program.

The Extra Segment register was ignored in this sample EXE program because it served no useful purpose. Generally, it is used to handle unique data groups in large programs. A good example is the "string" instruction where data is moved from a data segment pointed to by the DS register to a second data segment pointed to by the ES register. The Extra Segment register is initialized like the Data Segment register.



## Segment Addressing

The first unit in this course described how memory is addressed. The segment register provides a 16-bit **base** address that points to the beginning of the associated segment. The **logical**, or offset, address is then added to the base to produce the **physical** address of the memory location. In a COM-type program, all of the addresses are calculated from the same base address, because all of the segment registers contain the same address value. In an EXE-type program, each of the segment base addresses can be different. As a result, the offset address to a memory location in a COM-type program may not be the same as the offset address in an EXE-type program. To see how that works, let's compare two similar programs.

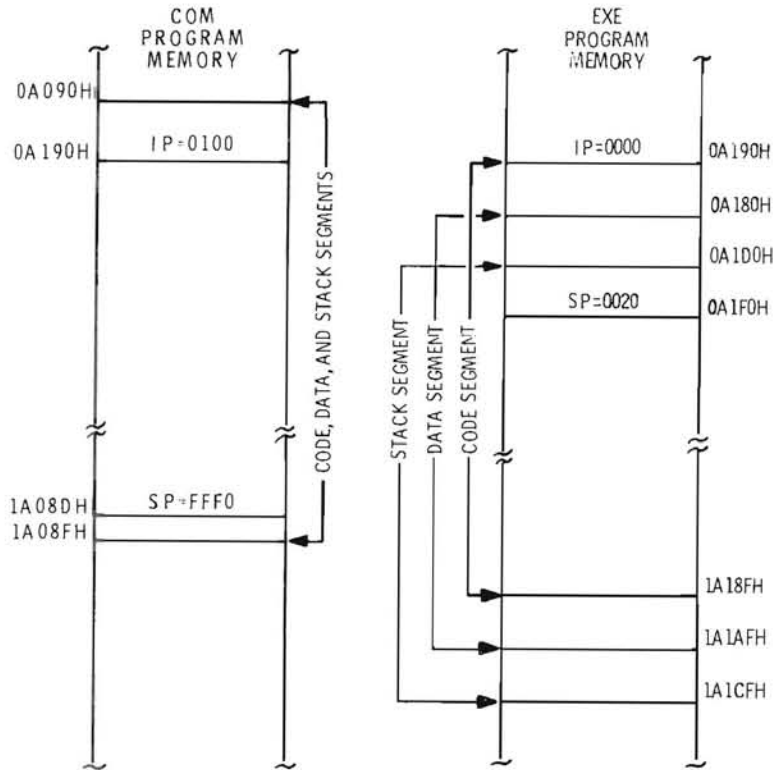
Figure 7-2 shows a COM-type program that contains the same program code, without the segment register initialization steps, and the same data as the EXE-type program in Figure 7-1. Figure 7-3 compares the memory segments for both programs. On the left is the COM program. All of the segments share the same 64K block of memory. Because of this, all offsets into the program are referenced from the same segment base address. The stack area was not defined, therefore it is automatically located at the end of the 64K segment. Actually, the Stack Pointer is shifted 16 bytes into the stack by the DOS program loader.

```
TITLE UNIT 7 -- PROGRAM 2 -- COM PROGRAM STRUCTURE
COM_PROG SEGMENT
    ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
    ORG    100H
START: MOV    BL,DATA1        ;Get the first byte of data
        LEA   BX,DATA2        ;Point to the first word of data
        MOV   DX,[BX]         ;Get the first word of data
        INT   3               ;Return to the debugger
;
DATA1  DB    16 DUP (0BH)     ;Set up byte-sized data area
DATA2  DW    8 DUP (0BBH)     ;Set up word-sized data area
;
COM_PROG ENDS
        END    START
```

**Figure 7-2**  
COM program structure.

The right side of Figure 7-3 shows the memory arrangement for the EXE program. Each program segment is treated as a separate memory block, with a unique base address for each segment. Depending on the size of a segment, it may occupy a separate 64K section of memory, or it may overlap another segment. In this example, each segment requires very little memory; therefore, the segments overlap. The “linker” program determines the size of each segment and arranges the segments so they occupy sequential bytes of memory. Early versions of the linker program (Version 1.10 for instance) place the code segment first, followed by the data segment, and finally the stack segment. Later versions of the linker program (Version 2.00 for instance) place the stack segment first, followed by the data segment, and finally the code segment.

After the program object code is linked, all of the offset address references are fixed relative to the segment base addresses. However, the segment base addresses are not fixed at this time. Each contains a value relative to other segments. The DOS program loader assigns the actual base address for each of the segments when it loads the program into memory.



**Figure 7-3**

Comparison of COM and EXE program segments.

When code or data is accessed in the EXE program, the offset is referenced to the **default** segment register. That is, for code, the offset is added to the contents of the Code Segment register; for data, the offset is added to the Data segment register. The one, immediate, exception to this rule involves indirect register addressing using the **Base Pointer** register. Here, the default segment register is the **Stack Segment** register. Data accessed by the BP register is, by default, in the stack segment. That's no problem in a COM program because all of the program segments reside in the same memory segment area. But in an EXE program, you must make sure the data you are accessing was originally assigned to the stack segment. One way around this potential problem of addressing the wrong segment area is with the segment override assembler operator.

## Overriding The Default Segment

When you begin writing complex programs, there will be times when you need to use all of the base and pointer registers for indirect register addressing into the data segment. Since the BP register defaults to the stack segment, you must be able to tell the MPU that you actually wish to address the data segment. This is handled by a **segment override** assembler operator.

The segment override operator is a two-letter abbreviation of the segment register name followed by a colon. It tells the assembler that a segment other than the default segment is to be used in the instruction. When the assembler sees the operator, it generates a byte of code that identifies the specified segment register. That code is then used to tell the MPU that the accompanying instruction will use the indicated segment register in place of the default segment register. The segment override operator takes the form:

Code Segment = CS:  
Data Segment = DS:  
Stack Segment = SS:  
Extra Segment = ES:

To move a word of data from the data segment into the AX register using the BP register in indirect register addressing, the instruction will take the form:

```
MOV AX,DS:[BP] ;Move using data segment override
```

To reverse the operation, use the instruction:

```
MOV DS:[BP],AX ;Move using data segment override
```

In each case, the segment override operator precedes the instruction operand that contains the offset, or effective address. Remember, any time you address data that resides in a segment that is different than the segment assumed by the assembler, you must use the segment override operator. If you forget, there is no way for the assembler to recognize that an error exists. As a result, it will generate the code that sends the MPU to the wrong memory segment, causing what could be a fatal program error.

If the assembler recognizes that you are attempting to address data in a segment that is not the default segment, it will automatically generate the segment override code. Thus, if it sees the instruction:

```
MOV AX,DATA1 ;Data in CS register
```

and the assembler knows that DATA1 is located in the code segment, it will precede the instruction code with the segment override code for the code segment. When the MPU sees the segment override code, it knows that the address offset to DATA1 is referenced to the Code Segment register rather than the usual Data Segment register.

The segment override assembler operator gives you a way of telling the MPU that data resides in a memory segment other than the normal, or default, segment. Thus, any of the four segment registers can be used to point to data within memory. But what happens when data resides in a memory segment that is not pointed to by a segment register?

## Changing Segments In Mid-Program

When you write a program, you use the ASSUME directive to tell the assembler which segment registers point to specific memory segments. If a segment register is **not** listed in an ASSUME directive statement, the assembler assumes nothing about that segment register. For example, in the following ASSUME directive statement:

```
ASSUME    CS:PROG_CODE,SS:PROG_STACK
```

the assembler assumes that the DS and ES registers will not be used by the program. Any data references will be made to the code segment or the stack segment. Now if the program contained a data segment, separate from the code and stack segments, that referenced:

```
DATA1    DB  16 DUP (0BH)
DATA2    DW   8 DUP (0BBH)
```

the instructions:

```
MOV  BL,DATA1  ;Get first byte of data
LEA  BX,DATA2  ;Get offset of array DATA2
```

will be flagged as errors by the assembler because these data locations can't be reached using the "assumed" segment registers. However, if you rewrite the instructions so that they contain a segment override operator such as:

```
MOV  BL,DS:DATA1  ;Get first byte of data
LEA  BX,DS:DATA2  ;Get offset of array DATA2
```

the assembler will assemble the code properly. The assembler makes the assumption that the DS register will contain the correct segment base address. It's up to you, as the programmer, to make sure the DS register contains the base address of the segment that contains the data being accessed, before the data is accessed.

In this example, we used the DS register as the segment override operator. Because the assembler expected to use the DS register as the base address for the data segment, no segment override code was generated. Had we used any of the other segment registers for the segment override, the assembler would have generated the appropriate segment code byte.

Using a segment override assembler operator only affects the instruction associated with that operator. The operator must be repeated for each instruction that uses a segment register not defined by an ASSUME directive statement.

The segment override assembler operator is convenient for identifying a memory segment area for one or two instructions. But where many instructions are involved, there is always a possibility you will forget to use or use the wrong segment override. To make sure that doesn't happen, it's best to make a blanket change and let the assembler keep track of the segments. This blanket change is made with the ASSUME directive.

You can use the ASSUME directive as often as you wish in a program. Each time you use it, the assembler is given new directions as to what segments are related to the segment registers. For example, the ASSUME directive statement:

```
ASSUME  CS:PROG_CODE,DS:PROG_DATA,SS:PROG_STACK
```

at the beginning of the code segment defines the segment to segment register relationship for the CS, DS, and SS registers. Nothing is assumed for the ES register. Now, when the assembler encounters the ASSUME directive statement:

```
ASSUME  ES:PROG_EXTRA
```

it **begins** associating all data in the segment PROG\_EXTRA with the segment base address in the ES register. However, the previous "assumptions" have not changed. Should the assembler encounter a third ASSUME directive statement:

```
ASSUME  DS:PROG_DATA2
```

the original "assumption" concerning the DS register is changed. Now the DS register is associated with the data in the segment PROG\_DATA2, **and not** the data in the segment PROG\_DATA.

A large amount of information has been presented in this section. Before we describe how the segment attributes in an EXE program are determined, you should review the preceding material. Answer the following questions. Then check your answers at the back of this Unit. If you miss any of the questions, study the related material before you continue.

## Self-Review Questions

1. A \_\_\_\_\_ is a logical unit of memory that is 64K bytes long.
2. Segments may be adjacent to each other but they must not overlap. \_\_\_\_\_  
True/False
3. The assembler "assumes" that instructions and data are randomly located throughout all of the segments of an EXE program.  
\_\_\_\_\_  
True/False
4. Every EXE program must contain at least \_\_\_\_\_ segment areas.
5. Every segment must be identified by a \_\_\_\_\_ name.
6. The argument \_\_\_\_\_ must follow the stack SEGMENT directive so that the assembler knows which segment defines the stack area in memory.
7. The assembler directive LABEL reserves a memory location.  
\_\_\_\_\_  
True/False
8. The beginning of a segment is identified by the \_\_\_\_\_ directive.
9. The end of a segment is identified by the \_\_\_\_\_ directive.
10. The \_\_\_\_\_ segment contains the ASSUME directive.
11. Before any memory operations are performed, the first instructions in an EXE program should be used to: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

12. The \_\_\_\_\_ Segment and \_\_\_\_\_ Pointer registers don't have to be initialized by the program code.
13. A physical address in memory is determined by combining a segment \_\_\_\_\_ address with a \_\_\_\_\_ address.
14. The linker arranges the code, data, and stack segments so that the \_\_\_\_\_ segment is first, the \_\_\_\_\_ segment is next, and the \_\_\_\_\_ segment is last.
15. Data addressed by the \_\_\_\_\_ register is, by default, within the stack segment.
16. You can override the assembler's choice of segment register in a memory operation with the \_\_\_\_\_ assembler operator.
17. You can change the assembler's "assumptions" about a segment with the \_\_\_\_\_ directive.

NOTE: The Self-Review Answers are located at the end of this unit, beginning on Page 7-75.



## Segment Attributes

In Unit 2, you learned that many different attributes can be assigned to a segment. These attributes determine how multiple segments relate to each other when they are linked and loaded into memory. Until now, you haven't specified any segment attributes because the default attributes were acceptable in your COM-type programs. With EXE-type programs, the segment attributes are important.

Segment attributes can take three forms: align-type, combine-type, and 'class-name'. These are shown in the expanded SEGMENT directive statement:

```
<seg-name> SEGMENT <align-type> <combine-type> <'class-name'>
```

You will recall that **seg-name** represents the unique name assigned to each segment in the program. The term **align-type** specifies how the program is to be linked and loaded into memory. There are four align-types:

- PARA — Is the default alignment that stands for paragraph alignment. This specifies that the segment begins on a paragraph boundary — the address is divisible by 16. That is, the last four bits of the 20-bit physical address are zero.
- PAGE — Specifies that the segment begins at any address that is divisible by 256. In this case, the last eight bits of the 20-bit physical address are zero. The PAGE alignment is used to maintain code location compatibility between 8080 assembly language programs and 8088/8086 assembly language programs.
- WORD — Specifies that the segment begins at any even-numbered address boundary. In this case, the last bit in the 20-bit address is zero.
- BYTE — Specifies that the segment begins at any address boundary in memory.

Of the four, “paragraph” alignment is used most often. It insures that every segment in the program begins with an offset to the segment base address of zero. However, it can cause the program to waste from 1 to 15 bytes of memory between segments. This will happen when the preceding segment doesn’t “end” on a paragraph boundary. To maintain alignment, the linker automatically fills-in the unused memory locations with zeros.

The “byte” alignment eliminates any wasted memory by allowing each segment to be loaded “immediately” after the preceding segment. However, a byte alignment should only be used with programs written specifically for the 8088 MPU. While the 8086 MPU can work with byte-aligned code, it is most efficient reading word-aligned code. For that reason, “word” alignment is the best alignment for saving memory between segments when it comes to programs written specifically for the 8086 MPU.

The second segment attribute is **combine-type**. This attribute is used to tell the **linker** how a segment should be combined with other segments of the **same segment name**. Until now, we have limited our discussion of the linker to how it arranges the code, data, and stack segments in a single program. The linker also allows you to “combine” the **object code** of two or more programs to produce a single EXE program. The advantage here is that you can write a number of simple programs to handle specific tasks, and later combine the programs to perform a complex task. We’ll describe the features of the linker in more detail later in the course. For now, let’s continue with segment attributes.

If no combine-type is specified for a segment, it is considered to be non-combinable, or “private.” Thus, it will not be combined with any other segments, even if they have the same segment name, such as PROG\_DATA. Private segments are loaded separately and remain separate. While they may be physically adjacent to other segments within memory, they are not “logically” continuous. Each private segment will have its own segment base address, and all address offsets within the segment will be from that base address. For that reason, you must be very careful how you use private segments in combination with other similar segments. Addressing them can be tricky. You must make sure the program loads the appropriate segment register with the base address value of the private segment before the program attempts to access that segment. Until you become familiar with multiple-segment programming, we suggest you avoid private segments.

Of more interest to you right now are the combine-type options, PUBLIC and STACK. Public segments, with the same segment name, are combined physically and logically. This effectively produces a single segment with a new base address and a length equal to the sum of the lengths of all the combined segments. The address offsets into the combined segments are then recalculated by the linker to accommodate the new effective segment size.

Figure 7-4 shows the effect of combining three public data segments that are part of three separate programs. Each has the common segment name DATA. The left side of the figure shows the three individual program segments prior to running the linker. Each has a unique segment base address, and the data within each segment has a unique address offset from its respective base address. After linking, the program has one DATA segment with a single base address, and each byte of data has a new offset address referenced to that base address. This is shown on the right side of the figure. The linker also makes sure that all program references to the DATA segment use the correct segment base and offset address values. Although we used data to show how segments are combined, the process applies to program code segments that have the same segment name.

BASE	DATA	OFFSET
09D2H	53	0000H
	A1	0001H
	2F	0002H
	CD	0003H
09E4H	6D	0000H
	78	0001H
	AC	0002H
	B3	0003H
09DAH	44	0000H
	9E	0001H
	12	0002H
	38	0003H
	AA	0004H
	BC	0005H
BASE	DATA	OFFSET
0A31H	53	0000H
	A1	0001H
	2F	0002H
	CD	0003H
	6D	0004H
	78	0005H
	AC	0006H
	B3	0007H
	44	0008H
	9E	0009H
	12	000AH
	38	000BH
	AA	000CH
	BC	000DH

**Figure 7-4**  
Combining PUBLIC segments.

Combine-type STACK is similar to PUBLIC. It causes all stack segments with a common name to be combined physically as well as logically. The linker also modifies all of the “end of stack label” references so that they point to the end of the “combined” stack segments.

Although PUBLIC and STACK are the most often used combine-type attributes, there are three other attributes that should be mentioned. First is the combine-type COMMON. It specifies that all segments with this attribute and the same segment name will be linked together. However, the length of the linked segment will equal the length of the longest segment being combined. Thus, if three data segments with lengths of 20H bytes, 10H bytes, and 5H bytes are combined using the COMMON attribute, the resulting segment will only be 20H bytes long. Therefore, you should only use this attribute when you are combining data segments that will only be used as a “scratch pad” for calculations during program execution. Initialized data may be lost when the COMMON segments are combined.

The next combine-type attribute is MEMORY. As defined by Microsoft, this combine-type causes the segment to be located above all other segments being linked together. If several segments have the MEMORY attribute, only the first one encountered by the linker is processed as a MEMORY segment. All others are processed as COMMON segments. In actual fact, all current linkers (including Version 1.10 and 2.00) treat all MEMORY segments as PUBLIC segments.

The last combine-type attribute is AT <expression>. It specifies that this segment is to be located at the **16-bit paragraph number** (segment base address) indicated in the “expression.” For example, if you specify “AT 0H” the segment begins at absolute, or physical, address 00000H. However, the attribute cannot be used to force the loading of data within that segment. Rather it is used to define, or identify, labels or variable names at fixed offsets within fixed areas of memory. When we introduce system interrupts, we will be using this combine type to point to address values within the “interrupt vector table” in low memory.

The last segment attribute is 'class'. This entry in the SEGMENT directive statement is the name (enclosed in single quotes), or classification, used to group segments when they are linked. It serves as a secondary identifier to determine how the segments will be combined. For example, suppose three programs are being linked, and each contains a code SEGMENT directive statement such as:

```
PROG_CODE SEGMENT PARA PUBLIC
PROG_CODE SEGMENT PARA PUBLIC
PROG_CODE SEGMENT PARA PUBLIC
```

the three segments will be combined in the order listed. However, if you add the 'class' attribute to two of the segments, such as:

```
PROG_CODE SEGMENT PARA PUBLIC 'FIRST'
PROG_CODE SEGMENT PARA PUBLIC
PROG_CODE SEGMENT PARA PUBLIC 'FIRST'
```

the three segments will still be combined, but the order will be first segment, third segment, and then second segment.

This completes the presentation on structuring the EXE program. The next section will show you how to structure the EXE program code to make use of multiple program segments. Before you proceed with that section, complete the following Self-Review Questions.

## Self-Review Questions

18. The default “align-type” attribute in a SEGMENT directive statement is \_\_\_\_\_.
19. Align-type \_\_\_\_\_ specifies that the segment can begin at any address that is divisible by 256.
20. Align-type \_\_\_\_\_ specifies that the segment can begin at any address that is divisible by 16.
21. Align-type \_\_\_\_\_ specifies that the segment can begin at any address in memory.
22. Align-type \_\_\_\_\_ specifies that the segment can begin at any even numbered address in memory.
23. The \_\_\_\_\_ program uses the align-type attribute to determine how the object program segments will be arranged in memory.
24. If a segment has no “combine-type” attribute, the segment is considered \_\_\_\_\_.
25. The combine-type attribute determines both the physical as well as the \_\_\_\_\_ addressability of a segment.
26. Combine-type PUBLIC segments will be combined by the linker if their segment \_\_\_\_\_ are identical.
27. Combine-type \_\_\_\_\_ is similar to combine-type PUBLIC.
28. Three segments with combine-type COMMON and lengths of 5, 15, and 10 bytes will produce a segment \_\_\_\_\_ bytes long when combined.
29. Combine-type MEMORY is treated like combine-type \_\_\_\_\_ by the linker.
30. To specify the physical location of a label within memory, you would use combine-type \_\_\_\_\_.
31. A secondary segment identifier is the segment attribute \_\_\_\_\_.

## INTRA AND INTERSEGMENT ADDRESSING

Intrasegment and intersegment addressing refer to all forms of addressing that occur within program code. The instructions involved with this type of addressing include jumps, calls, and return from calls. Specifically, **intrasegment addressing** is restricted to the 64K byte boundary of a single code segment. **Intersegment addressing**, on the other hand, relates to addressing **between** code segments. In the first case, the only address variable is the offset added to the Instruction Pointer to indicate the location of the next instruction. In the second case, both the Instruction Pointer and the Code Segment register are changed to locate the next instruction. How these variables are handled is the subject of this section. We'll begin our discussion with intrasegment addressing.

### Intrasegment Addressing

All of the COM programs you have written thus far have used the intrasegment form of addressing. That is, the jump, call, and return from call instructions have been restricted to a single segment, and all instruction references have been handled through the Instruction Pointer. Most of the EXE programs that you will write will also restrict the program code to one segment. Thus, the techniques you learned with COM programs will also apply to these types of EXE programs.

Linking multiple EXE programs does present a problem. Although the final "linked" program contains only one code segment, it can be composed of two or more code segments. For that reason, both the assembler and the linker must be informed of any references between code segments. This is handled through two assembler directives: PUBLIC and EXTRN.

The assembler directive `PUBLIC` is used to identify those labels or names in a program segment that can be accessed by other program segments. The directive statement takes the form:

**`PUBLIC <symbol>[,...]`**

where **symbol** is the label or name you wish to declare “public.” You can declare as many symbols as you wish in one statement by separating the symbols with a comma. After assembly, information about each symbol is passed on to the linker for final determination of its effective address within the linked program. For example, before the linker can assist a call instruction assembled in one code segment in accessing a subroutine assembled in a second code segment, the name of the subroutine must be declared public:

```
        PUBLIC DELAY_LOOP, COUNT
DELAY_LOOP:
        MOV CX, 0
        LOOP DELAY_LOOP
        RET
COUNT:
        ADD AX, DATA1
        MOV DATA1, AX
        RET
```

In this case, two symbols have been made public. Each points to the beginning of a subroutine.

Making a symbol public tells the linker that another “linked” segment may try to access that symbol. The next step is to identify the instruction that will use the “public” symbol. This is handled by the assembler directive `EXTRN`.

Recall that whenever a symbol is referenced in an instruction, it must be defined elsewhere in the program. If the definition is missing, the assembler generates an error message. The `EXTRN` directive tells the assembler that the indicated symbol is located outside the current program; any reference to that symbol should not be flagged as an error. The directive statement takes the form:

**`EXTRN <symbol:type>[,...]`**



where **symbol** is the label or name you wish to declare as “external” to the current program. **Type** must match the “symbol” that was declared “public” in the external program. It can assume any of the following attributes:

BYTE, WORD, or DOUBLEWORD — Refers to the definition assigned to the “named” data.

NEAR or FAR — Refers to the “linked” location of the label or procedure (to be covered later). If the label or procedure resides in the same segment after linking, the attribute should be NEAR. If, on the other hand, the label or procedure resides in a different segment after linking, the attribute should be FAR.

The assembler uses the symbol and type attributes to determine how much code space should be reserved for the address information. The linker then compares the PUBLIC symbol to the EXTRN symbol to calculate the address values before it loads them into the program code. The following is an example of how an EXTRN directive statement can be used:

```
EXTRN COUNT:NEAR,DATA1:WORD
CALL COUNT
ADD AX,DATA1
```

The subroutine COUNT and the segment containing DATA1 are not part of the current program. The EXTRN directive tells the assembler that the two symbols will be available when the program is linked with another program. The type NEAR indicates that the subroutine COUNT will be in the same code segment as the call instruction, after linking. Had the type been FAR, the subroutine would have been located in a different code segment. The type WORD tells the assembler that DATA1 is composed of one or more word-sized values.

Keeping all the code in one segment makes the addressing process simpler. The three program transfer instructions operate just as they did in your earlier COM programs. The only added wrinkle is the need to identify all references external to the current program, and all public symbols. While all public symbols don't have to be used in a program, all external references must have a matching public symbol, or the program will fail to link properly.

## Intersegment Addressing

Intrasegment addressing refers to the use of program transfer instructions within a code segment. **Intersegment addressing** refers to the use of program transfer instructions **between** code segments. Often, these types of instructions are called **far** instructions because not only do they change the contents of the Instruction Pointer, they also change the contents of the Code Segment register. This last part is necessary, since the program is leaving one code segment and entering another.

Two situations can exist in a “far” jump. First, the jump occurs between two segments within a single program. Second, the jump occurs between segments that exist in two separate programs prior to linking, and remain as separate segments after linking. The second example was mentioned earlier, but not fully described. The assembler directive `EXTRN` is used to identify the symbol and its type (near or far) prior to the jump instruction. Thus, a far jump could be coded:

```
EXTRN SUBF:FAR
JMP SUBF
```

where the jump target is `SUBF` and the type is “far.” From this information, the assembler knows that the jump is a far jump. It automatically reserves space in the program code for the new code segment base address value and the offset address value to the subroutine `SUBF`. When the program is linked, the linker determines the relative address values for the segment and the offset into the segment. Finally, when the program is loaded into memory, the absolute code segment address is determined. Execute the instruction:

```
JMP SUBF
```

and the MPU loads the Instruction Pointer register with the offset address value into the new segment. Then it loads the Code Segment register with the base address of the new code segment. Finally, it executes the instruction at the address pointed to by the CS and IP registers.

A different method must be used when the far jump occurs within a single program. In this case, we use the assembler operator PTR. In addition to specifying operand size, as described in an earlier unit, PTR can be used to specify operand distance. The instruction:

```
JMP FAR PTR SUBF
```

tells the assembler that the jump target is in another code segment. As a result, the assembler will reserve code space for a segment base address and a jump target offset address. As before, the linker will calculate the relative offset and base addresses, and the program loader will determine the absolute segment base address when the program is loaded into memory.

When the far jump instruction is executed, the MPU loads the Instruction Pointer register with the offset address value into the new segment. Then it loads the Code Segment register with the base address of the new code segment. Finally, it executes the instruction at the address pointed to by the CS and IP registers.

Far call instructions are constructed in the same manner as far jump instructions. The EXTRN directive is used for intersegment calls to external programs. The FAR PTR operator is used with intersegment calls within the same program. The difference between the far jump and the far call is that the far jump is final. There is no easy way to return from a jump. Calls, on the other hand, save the return address in the program stack.

Recall that when an intrasegment call is executed, the Instruction Pointer register is updated so that it points to the next instruction in the program. Then the contents of the register are saved in the stack. Popping the address back into the Instruction Pointer with a return instruction sends the MPU back to the instruction following the earlier call instruction.

The same process holds true for intersegment calls. Only now, the contents of the Code Segment register are pushed into the stack along with the updated contents of the Instruction Pointer register. When the return from call instruction is executed at the end of the subroutine, the first word in the stack is popped into the IP register, then the next word in the stack is popped into the CS register. This returns the MPU to the instruction following the earlier far call instruction. The only difference between an intrasegment call and an intersegment (far) call is the number of register address values that are saved in the stack.

The same holds true for the return from call instruction. An intrasegment return pops the first word in the stack into the Instruction Pointer register. An intersegment return from call pops the first word in the stack into the Instruction Pointer register and the next word in the stack into the Code Segment register.

Coding an intrasegment return, you simply use the mnemonic RET. Coding an intersegment return isn't quite that simple. MACRO-86 assumes that the only time you will use a far return instruction is in a "far procedure." You will learn in Unit 9 that procedures are a method of structuring subroutines or other groups of code into convenient, defined modules. A far procedure is accessed through a far call instruction. Thus, a return from a far procedure must be a far return. The instruction mnemonic is still RET, but the assembler "knows" it must generate the code for a far return. If you need to write the code for a far return that isn't in a far procedure, you must "fool" the assembler into thinking the return is in a far procedure by writing a "dummy" far procedure in this manner:

```
FAR_PROC PROC FAR
          RET
FAR_PROC ENDP
```

Like the segment directive, the procedure directive must be identified by a unique name. We have chosen the name FAR\_PROC in this example. The directive PROC identifies the beginning of the procedure. The operator FAR specifies that this is a far procedure. The end of the procedure is identified by the same name as in the beginning, and by the directive ENDP. The instruction RET is the only instruction in the procedure, and it is treated as a far return by the assembler.

## Self-Review Questions

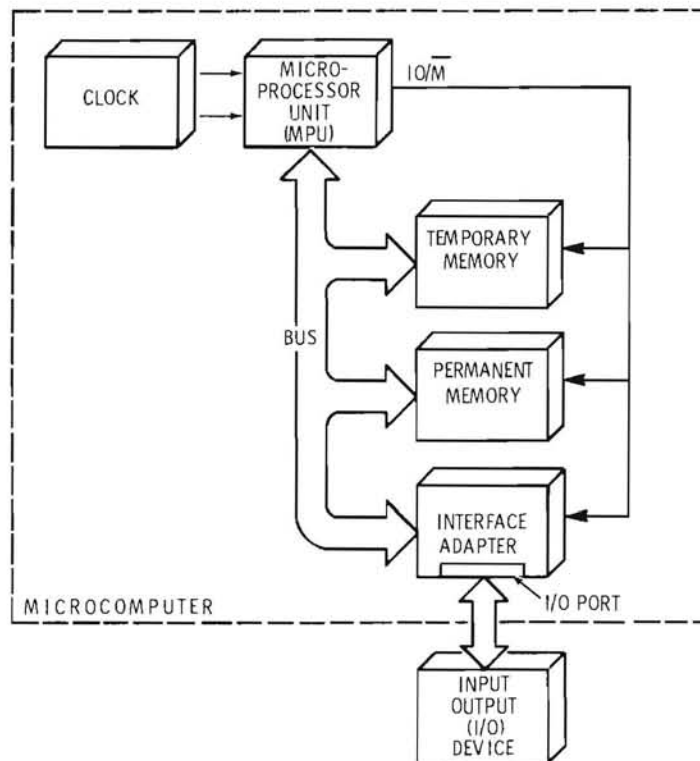
32. \_\_\_\_\_ addressing is restricted to the 64K byte boundary of a single code segment.
33. \_\_\_\_\_ addressing relates to program transfer between code segments.
34. COM-type programs use \_\_\_\_\_ addressing.
35. The assembler directive \_\_\_\_\_ is used to identify those symbols in a program that can be accessed by other programs when they are linked.
36. The \_\_\_\_\_ directive tells the assembler that the indicated symbol is located outside of the current program.
37. The EXTRN directive identifies a symbol by its label or name and by its \_\_\_\_\_.
38. All external references must have a matching public symbol.  
\_\_\_\_\_  
True/False
39. All public symbols must have a matching external reference.  
\_\_\_\_\_  
True/False
40. A far jump between segments in a single program is identified by the assembler operator \_\_\_\_\_.
41. A far call between segments of two programs that have been linked is identified by the external directive type \_\_\_\_\_.
42. When a far call is executed, the contents of the \_\_\_\_\_ register are pushed into the stack; then the contents of the \_\_\_\_\_ register are pushed into the stack.
43. A “dummy” far procedure must be created in order to execute an intrasegment return. \_\_\_\_\_  
True/False

## I/O ADDRESSING

Recall, from Unit 1, that data is moved into or out of the MPU through I/O ports. How those ports are accessed is subject of this section.

### Port Structure

The I/O ports are like memory. In fact, they share the MPU address and data bus with memory, as shown in Figure 7-5. In this example there are two memory sections, or devices, and one Interface Adapter. The Interface Adapter is a form of data translation device that helps the MPU communicate with the outside world. Thus, it is the Interface Adapter that is actually connected to the bus and treated as the I/O port.



**Figure 7-5**  
Memory and I/O control.

From a programming point of view, it's not important how the Interface Adapter is connected to the microcomputer. Rather, the important consideration is how the adapter is configured (using program code), and how the adapter will respond to the MPU and the peripheral device. One of the experiments in this unit will give you a chance to work with an Interface Adapter common to both the Zenith and IBM microcomputer, the 68A45 CRT (display) Controller. Although the CRTC is actually part of the microcomputer, it is treated as an Interface Adapter because it is addressed through an I/O port.

Because the memory and I/O ports share the address and data bus, there must be a way for the MPU to specify the group of devices being accessed. This is handled by the  $IO/\overline{M}$  (I/O-Memory) control line from the MPU. As the figure shows, each device is connected to the control line. When the MPU is addressing memory, the control line is pulled to a logic low level. This enables, or turns-on, the memory devices and disables the I/O devices; data transfer can only occur between the MPU and memory. By the same token, when the MPU is addressing an I/O port with the IN or OUT instruction, the  $IO/\overline{M}$  control line is forced high to disable memory and enable I/O. Now all data transfers are between the MPU and the addressed I/O port.

Like memory, each I/O port can accommodate 8-bit wide data. However, the total number of ports is limited to 64K bytes. This is because all I/O port addressing is handled through a single 16-bit register.

The physical location of every port in the microcomputer is fixed by the manufacturer. Unlike memory, you can't change a port location through a program. For that reason, if you decide to write data directly to a particular port, you must know its exact location, or address.

## Port Addressing

As a general rule, it is not a good idea to directly access an I/O port. The main reason for this rule is that you limit the portability of your program. Remember, each manufacturer assigns a specific port address to every function. The functions as well as the port address will vary from one manufacturer to another. If you access a port directly in your program, then you can't use the program on a different microcomputer. A program written for a Zenith microcomputer won't run on an IBM microcomputer, and vice versa. Therefore, the best way to access an I/O port is through an MS-DOS or BIOS (basic input output system) function like the many interrupt 21H functions we used earlier.

BIOS operation is similar to the MS-DOS functions we have been using except that it is hardware specific. IBM's BIOS is completely different from Zenith's BIOS. A good example is the interrupt 10 function that we used to clear the IBM display.

Naturally, there will be a few rare occasions when a DOS or BIOS function just won't do what you require. That's when you must use direct I/O port addressing. The first step in addressing an I/O port is to determine its location. Then you must determine what data is supplied at the port — the value the port expects you to send, or the value you can expect to read. All of this information can be found in the "Technical Reference Manual" for your microcomputer. The last step is to write the I/O instruction.

The 8088/8086 MPU uses two instructions to access an I/O port: **IN** to input a value, and **OUT** to output a value. With either instruction, the data is processed through the Accumulator register; for 8-bit values use the AL register, and for 16-bit values use the AX register. Finally, each instruction can be implemented in one of two ways. It may be used with either a fixed port address or a variable port address.



## FIXED PORT

With **fixed port** addressing, the instruction is limited to the first 256 I/O port address locations. The term “fixed port” comes from the fact that the port address is identified by an immediate value following the instruction. For instance:

```
IN    AX, 133
IN    AL, 25
OUT   5DH, AX
OUT   0D4H, AL
```

In the first two examples, the source operand is a fixed port address and the destination operand is an Accumulator register. The next two examples position the Accumulator contents in the source operand and the port address in the destination operand.

When you are dealing with port addresses, it's usually a good idea to use symbols to represent the fixed value. This reduces the chance of loading the wrong address in the instruction. It also makes the instruction easier to understand. As an example:

```
TEMP_IN    EQU    3DH
TEMP_OUT    EQU    4DH
INCREASE    EQU    22
DECREASE    EQU    77
LOW_TEMP    EQU    55
START:      IN     AL, TEMP_IN
            CMP    AL, LOW_TEMP
            JA     REDUCE
            MOV   AL, INCREASE
            OUT   TEMP_OUT, AL
            JMP   START
REDUCE:     MOV   AL, DECREASE
            OUT   TEMP_OUT, AL
            JMP   START
```

Here is a simple routine that monitors the temperature signal from a peripheral device. To make the program easy to follow, we equated all constants to meaningful symbols. The temperature is read at port TEMP\_IN and compared to a value. If the temperature is low, the control signal to INCREASE the temperature is loaded into the AL register and then sent to the TEMP\_OUT port. Should the temperature be too high, the control signal to DECREASE the temperature is loaded into the AL register and then sent to the TEMP\_OUT port. After each output, the program loops back to the input instruction.

To access a port higher than 256 (0FFH), use the variable port addressing method.

## VARIABLE PORT

With **variable port** addressing, you are not limited to a small range of fixed port addresses. First, you can address all available port locations from 0000H to 0FFFFH. Second, as the name implies, you can change the port address at any time during program execution. This is all possible because the port address is contained in the DX register. Following are examples of addressing a variable port:

```
IN_PORT    EQU 0EF22H
MOV  DX, IN_PORT
IN   AX, DX
MOV  DX, 12
IN   AL, DX
OUT  DX, AL
MOV  DX, 5555H
OUT  DX, AX
```

In each IN or OUT instruction, the DX register contains the address of the I/O port. The three MOV instructions are included to show port address values being loaded into the DX register.

As you can see, each port addressing type has its trade-offs. Fixed port addressing doesn't tie up the DX register. But then, its addressing range is limited to the first 256 ports, and the address value is fixed at assembly time. Variable port addressing gives you access to all of the ports, and it allows you to change the port address during program run time. However, it does restrict the program in how it uses the DX register.

## Self-Review Questions

44. Memory is disabled and I/O is enabled by the \_\_\_\_\_ and \_\_\_\_\_ instructions.
45. An I/O port can accommodate \_\_\_\_\_ bits of data.
46. The address of a \_\_\_\_\_ port is defined as a constant.
47. The address of a \_\_\_\_\_ port is pointed to by the contents of the DX register.
48. There are \_\_\_\_\_ 8-bit fixed ports available to the 8088/8086 MPU.
49. The AH register can be used as the destination operand for an IN instruction accessing an 8-bit port. \_\_\_\_\_  
True/False
50. The source operand for an IN instruction that uses a fixed port is always a/an \_\_\_\_\_ value.
51. Both the fixed and the variable addressing methods can access the first 256 I/O port locations. \_\_\_\_\_  
True/False

## EXPERIMENT

### EXE Programming and I/O

- OBJECTIVES:*
1. *Demonstrate the differences between EXE and COM programs.*
  2. *Demonstrate the process of linking multiple EXE-type program object files.*
  3. *Demonstrate the various segment attributes.*
  4. *Demonstrate I/O.*

### Introduction

The EXE-type program allows you to escape the 64K byte boundary of the COM-type program. However, you will find that there are trade-offs between the two types of programs. The EXE-type program requires more directives and code to accomplish the same job as a similar COM-type program. On the other hand, the COM-type program is limited in size to 64K of code and data. For a beginning programmer like yourself, you can probably write all of your programs using the COM format. There will come a time, though, when you will need the versatility of an EXE-type program. This experiment will give you experience with the EXE program format.

Once you've gained a foothold in constructing an EXE-type program, we'll show you how the MPU communicates with a peripheral device. Because of the surprising differences between the I/O structure of the IBM and Zenith microcomputers, there is little opportunity to experiment with similar I/O interfaces. This is not to say that they don't use similar interfaces, it's just that they take a unique approach to the way the interfaces are used. The only common device is the CRT Controller mentioned earlier in the unit. Therefore, the I/O portion of this experiment will center on the CRTC. To begin, let's compare a COM-type program with an EXE-type program that performs the same task.

## Procedure

1. Call up the editor and enter the program listed in Figure 7-6. Assemble, link, and convert the program to a COM file. Now delete the program EXE file. It's always a good idea to delete the EXE file that is generated in the process of making a COM file. The file is of no value, and this eliminates the possibility of executing it by mistake, at some later time.

```
TITLE EXPERIMENT 7 -- PROGRAM 1 -- COM PROGRAM STRUCTURE
COM_PROG SEGMENT
    ASSUME CS:COM_PROG,DS:COM_PROG,SS:COM_PROG
;
    ORG    100H
START: MOV    BL,DATA1        ;Get the first byte of data
        LEA   BX,DATA2        ;Point to the first word of data
        MOV   DX,[BX]         ;Get the first word of data
        INT   3               ;Return to the debugger
;
DATA1  DB    16 DUP (0BH)     ;Set up byte-sized data area
DATA2  DW    8 DUP (0BBH)     ;Set up word-sized data area
;
COM_PROG ENDS
        END    START
```

**Figure 7-6**

Typical COM-type program.

- Call up the editor and enter the program listed in Figure 7-7. Be sure to use a different program name. You will be comparing this program with the one you wrote in the previous step. Assemble, link, and try to convert the program to a COM file. Notice that, this time, the linker didn't display the message:

**Warning: No STACK segment**

**There was 1 error detected.**

This is because the program did indeed contain a STACK segment. On the other hand, the EXE2BIN program displayed the message:

**File cannot be converted**

because the program contained too many segments. Examine the disk directory. Notice that the EXE2BIN program didn't even attempt to create a COM file of your program.

```

TITLE EXPERIMENT 7 -- PROGRAM 2 -- EXE PROGRAM STRUCTURE
;
PROG_STACK SEGMENT STACK
    DW      16 DUP (0FH)      ;Set up stack area
TOP_OF_STACK LABEL WORD      ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_DATA SEGMENT
DATA1 DB      8 DUP (0AAH)    ;Set up byte-sized data area
DATA2 DW      8 DUP (0BBBBH)  ;Set up word-sized data area
PROG_DATA ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,DS:PROG_DATA,SS:PROG_STACK
START: MOV     AX,PROG_STACK    ;Never load a segment register direct
      MOV     SS,AX           ;Use an intermediate register
      MOV     SP,OFFSET TOP_OF_STACK ;Point to the top of stack
                                   ;immediately after loading SS register
      MOV     AX,PROG_DATA     ;Again, indirectly load the
      MOV     DS,AX           ;segment register
      MOV     BL,DATA1        ;Get the first byte of data
      LEA    BX,DATA2         ;Point to the first word of data
      MOV     DX,[BX]         ;Get the first word of data
      INT     3               ;Return to the debugger
PROG_CODE ENDS
      END     START

```

**Figure 7-7**

Typical EXE-type program.

3. Again, examine the directory. The COM file of your first program contains \_\_\_\_\_ bytes of code. This value is listed in the third column of the directory display. The EXE file of your second program contains \_\_\_\_\_ bytes of code. The difference in size between the two files is about 600 bytes; and yet, the EXE file contains only 29 more bytes of code and data than the COM file. The difference lies in the size of what we call the “program header.” An EXE file must supply more information about its characteristics to the DOS program loader than a COM file.
4. Call up the debugger and load your EXE file. Now examine the MPU registers by typing “R” and RETURN. The display generated by your debugger will resemble Figure 7-8, although the segment register values may be different. Recall that the contents of the Code Segment register are fixed by the program loader. The other registers must be loaded by the program. The first instruction in your program moves the stack segment base address into the AX register in preparation for moving the value into the Stack Segment register. Notice that the value in the instruction matches the value already in the SS register. That is because the debugger has already loaded the SS register. This is an automatic function of the debugger, since it must know where the stack is located before it can execute any of its commands. For that same reason, it has also determined the top of the stack area and loaded that value into the Stack Pointer register.

```
DEBUG version 1.00
>R
AX=0000 BX=0000 CX=0060 DX=0000 SP=0020 BP=0000 SI=0000 DI=0000
DS=0A09 ES=0A09 SS=0A1D CS=0A19 IP=0000  NV UP  DI PL NZ NA PO NC
0A19:0000 B81D0A      MOV     AX,0A1D
>
```

**Figure 7-8**  
Debugger register display.

5. Examine the program in memory. Use the debugger “D” command followed by the contents of your CS register, a colon, and the offset value 0000. For our program, we used the command “D0A19:0000”. This produced the display shown in Figure 7-9. While your code segment base address was probably different, your display should be similar to the figure. NOTE: If you don’t see a display with the code followed by the data and stack areas, you have a later version linker and debugger.

There are two general versions of linker and debugger. The first, which we’ll call the **early** version, is supplied with system software (DOS) with a version number in the “1” series (1.10, 1.25, etc.). This version produced the display in Figure 7-9. The second version, we’ll call the **late** version. It is supplied with system software in the “2” series (2.02, 2.04, etc.).

```

>R
AX=0000 BX=0000 CX=006D DX=0000 SP=0020 BP=0000 SI=0000 DI=0000
DS=0A09 ES=0A09 SS=0A1D CS=0A19 IP=0000  NV UP DI PL NZ NA PO NC
0A19:0000 B81D0A      MOV     AX,0A1D
>D0A19:0000
0A19:0000 B8 1D 0A 8E D0 BC 20 00-B8 1B 0A 8E D8 8A 1E 00  8...P< .8...X...
0A19:0010 00 8D 1E 08 00 9A 00 00-1F 0A 8B 17 CC 00 00 00  .....L...
0A19:0020 AA AA AA AA AA AA AA AA-BB BB BB BB BB BB BB  *****;
0A19:0030 BB BB BB BB BB BB BB BB-BB 00 00 00 00 00 00  ;
0A19:0040 0F 00 0F 00 0F 00 0F 00-0F 00 0F 00 0F 00 0F 00  .....
0A19:0050 0F 00 0F 00 0F 00 0F 00-0F 00 0F 00 0F 00 0F 00  .....
0A19:0060 8B 0E 08 00 E2 FE CB EA-1A 00 19 0A CC 0F 89 46  ...b^Kj...L..F
0A19:0070 F6 83 F8 FF 75 20 FF 76-0C 9A 0A 00 1D 0D 89 DF  v.x.u .v....._
>

```

**Figure 7-9**

Early version debugger display of registers and memory.



Late version software arranges the code, data, and stack segments in a different order than early version software. This is shown in Figure 7-10. Here the stack area is loaded into memory first, followed by the data, and then the code. To examine your program in memory, use the debugger command "D" followed by the contents of your SS register, a colon, and the offset value 0000. For our program, we used the command "D0D29:0000". While your stack segment base address was probably different, your display should be similar to Figure 7-10.

```

-R
AX=0000 BX=0000 CX=0058 DX=0000 SP=0020 BP=0BCB SI=0B0B DI=0000
DS=0D19 ES=0D19 SS=0D29 CS=0D2D IP=0000 NV UP EI PL NZ NA PO NC
0D2D:0000 B8290D      MOV     AX,0D29
-D0D29:0000
0D29:0000 0F 00 0F 00 0F 00 0F 00-0F 00 0F 00 0F 00 0F 00 .....
0D29:0010 0F 00 0F 00 0F 00 0F 00-0F 00 0F 00 0F 00 0F 00 .....
0D29:0020 AA AA AA AA AA AA AA AA-BB BB BB BB BB BB BB BB *****;
0D29:0030 BB BB BB BB BB BB BB BB-BB-00 00 00 00 00 00 00 00 ;
0D29:0040 B8 29 0D 8E D0 BC 20 00-B8 2B 0D 8E D8 8A 1E 00 8)P< .8+.X..
0D29:0050 00 8D 1E 08 00 8B 17 CC-06 50 B0 FF 50 9A 04 02 .....L.P.P...
0D29:0060 BF 0B A3 BE 13 A1 78 09-8B 1E BE 13 88 47 03 A2 ?.#>.!x...>..G."
0D29:0070 86 0B FF 06 78 09 5D C3-55 8B EC 83 EC 0E C7 06 ....x.JCU.l.l.G.
-

```

**Figure 7-10**

Late version debugger display of registers and memory.

As a general rule, if you have early version software, always use the code segment base address when you wish to view memory. If you have late version software, use the stack segment base address to view memory.

Depending on software version, the program code begins at address offset 0000H or 0040H in the related figure and your display. The last byte is easy to identify because it contains the code CC.

In both cases, the data segment begins at address offset 0020H in the figure and in your display. Since the first byte of data is always located at address offset zero in the data segment, what value would you expect to find stored in the Data Segment register after the program loads that register? \_\_\_H.

Again, depending of software version, the stack segment begins at address offset 0000H or 0040H in the related figure and in your display. We had you load the value 000FH in each word of the stack to make the stack area easier to identify. Normally, you would leave the contents uninitialized. On the other hand, you will find that program debugging is easier if you start off with a known value in each of the bytes. Note that, in the late version software, the last word in the stack is zero, regardless of what value you initialized it to in your program.

Notice that the Stack Pointer register contains the value 0020H. It's pointing at the first byte after the stack. The first time a value is pushed into the stack, it will be decremented by two, and thus point to the first word location in the stack.

The last two lines (beginning at address offset 0060) in your display and in the figure contain random data outside of the program boundary.

6. Execute the first instruction by typing "T" and RETURN. The AX register is loaded with the stack segment base address.
7. Execute the next instruction. The Stack Segment register is loaded from the AX register. Do you notice anything strange in the display? The debugger appears to have jumped right over the instruction to load the Stack Pointer register. Actually, the instruction was executed; the debugger single-step routine just had no control over the execution.

This was caused by a feature of the 8088/8086 MPU. It's a built-in process that guarantees that the instruction immediately following a "move segment register" instruction will be executed no matter what or who is trying to interrupt the operation of the MPU. It makes sure the program will always have a chance to up-date the segment supporting (offset) register before some other operation occurs. In the case of the stack segment, the supporting register is the Stack Pointer register. Imagine what would happen if the program was interrupted before the Stack Pointer register was loaded. At best, the incorrect Stack Pointer contents would point at an unused area of memory; at worst, data or code would be overwritten and the program would fail. Be sure to always load the segment supporting register (if it's needed) immediately after you load a segment register.

8. Execute the next instruction. The AX register is loaded with the base address of the data segment. The BL register contains the value `..H`.

9. Execute the next instruction. The Data Segment register is loaded with the contents of the AX register. The DS register contains the value `_____H`. Does this match the value you recorded in step 5?

Although the data segment is located at offset `0020H` in your display, it is assumed that the segment data is loaded into memory beginning at an offset of zero within the segment. Thus, the value in the DS register is equal to the base address value for the first segment in the program plus the offset in memory to the data area. In Figure 7-9, the data segment base address turns out to be `0A1BH`. Recall from Unit 1 that when you add an offset value to a segment value, you have to shift the segment value four bits to the left. Adding `0A19H` to `0020H`:

$$\begin{array}{rcl} \text{Segment Value} & = & 0A190H \\ \text{Offset Value} & = & 0040H \\ \hline \text{Physical Value} & = & 0A1B0H \end{array}$$

gives the physical address value `0A1B0H`. Since we are assuming an offset of zero, the least significant four bits in the 20-bit value are cut off, leaving the segment base value `0A1BH`, the value loaded into the DS register.

The BL register now contains the value `_____H`. As before, the instruction immediately following a “move segment register” instruction is executed without hesitation on the part of the MPU. The value at `DATA1`, in the data segment, was moved into the BL register.

10. Execute the last two instructions. They simply load the effective address of `DATA2`, in the data segment, into the BX register; and then move the value at that offset address into the DX register using register indirect addressing.

11. Again, display your program in memory. Use the debugger “D” command followed by the contents of your CS register (early version) or SS register (late version), a colon, and four zeros (to represent an offset of zero). Notice that zeros separate the code from data and data from stack (in the early version display), and data from code (in the late version display).

Recall that when the segment “align-type” attribute is not specified, it is by default paragraph alignment. This means that the last four bits of the 20-bit physical address are zero. As a result, each line in your display begins on a paragraph boundary (address offset 0, 10H, 20H, etc.). Any unused memory bytes between the end of one segment and the beginning of another are assumed to be zero, hence the eight bytes of zeros following the data segment. Had the data segment contained only one byte of data, there would be 15 bytes of unused memory between the data segment and the stack (early) or code (late) segment. No zeros are added after the last segment in the program.

## Discussion

We recommend that you always use the default align-type “paragraph.” This way you are sure how the code, data, and stack will be loaded into memory. “Page” alignment gives mixed results. In early version systems, only the code segment will be given page alignment. That is, the data will begin on the next 256-byte boundary. The stack will still start on the next paragraph boundary after the data. On the other hand, late version systems will recognize page boundaries for all segments — the data segment will begin on the next 256-byte boundary after the stack and the code segment will begin on the next 256-byte boundary after the data.

In the case of “word” alignment, the early systems will cause the data segment to begin on the next word (even memory location) boundary after the code. If the code ends at an odd memory location, the linker will add a byte of zero to position the data at a word boundary. The stack will still begin on the next paragraph boundary. Late version systems recognize word boundaries for all segments.

“Byte” alignment follows the general word alignment guidelines. The one difference is that the next segment can begin at an odd memory location as well as an even memory location.

If you wish to observe these different alignment characteristics, modify the segment directives in your EXE program to match the desired alignment. Then assemble and link the program, and then load it into memory with the debugger. Examine the program in memory using the techniques described earlier.

## Procedure Continued

- Exit the debugger and edit the ASM file of your EXE program in the following manner. The ASSUME directive statement currently reads:

```
ASSUME CS:PROG_CODE,DS:PROG_DATA,SS:PROG_STACK
```

Delete the data segment reference so that the directive reads:

```
ASSUME CS:PROG_CODE,SS:PROG_STACK
```

- Assemble the program. Your assembler should display a message similar to the message shown in Figure 7-11. Although your program contains a data segment, the assembler “assumes” that it does not exist. Hence the error generated by the two instructions that address the data segment. Recall that there are two ways to resolve this conflict: You can use a “segment override” prefix code or you can insert a second ASSUME directive statement identifying the data segment.

```
A:MASM P7-5;
The Microsoft MACRO Assembler
Version 1.07, Copyright (C) Microsoft Inc. 1981,82

000D 8A 1E 0000 R           MOV    BL,DATA1       ;Get the first b
yte of data
Error --- 68:Can't reach with segment reg
0011 8D 1E 0008 R           LEA   BX,DATA2       ;Point to the fi
rst word of data
Error --- 68:Can't reach with segment reg

Warning Severe
Errors Errors
0      2

A:
```

**Figure 7-11**

Errors caused by missing data segment assume statement.

14. Refer to Figure 7-12 and re-edit your program as shown. Change the line:

```
MOV BL,DATA1 ;Get the first byte of data
```

to read:

```
MOV BL,DS:DATA1 ;Get the first byte of data
```

following that line, add the line:

```
ASSUME DS:PROG_DATA
```

```
TITLE EXPERIMENT 7 -- PROGRAM 4 -- LOCATING THE DATA SEGMENT
;
PROG_STACK SEGMENT STACK
    DW 16 DUP (0FH) ;Set up stack area
TOP_OF_STACK LABEL WORD ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_DATA SEGMENT
DATA1 DB 8 DUP (0AAH) ;Set up byte-sized data area
DATA2 DW 8 DUP (0BBBBH) ;Set up word-sized data area
PROG_DATA ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,SS:PROG_STACK
START: MOV AX,PROG_STACK ;Never load a segment register direct
      MOV SS,AX ;Use an intermediate register
      MOV SP,OFFSET TOP_OF_STACK ;Point to the top of stack
      ;immediately after loading SS register
      MOV AX,PROG_DATA ;Again, indirectly load the
      MOV DS,AX ;segment register
      → MOV BL,DS:DATA1 ;Get the first byte of data
      → ASSUME DS:PROG_DATA
      LEA BX,DATA2 ;Point to the first word of data
      MOV DX,[BX] ;Get the first word of data
      INT 3 ;Return to the debugger
PROG_CODE ENDS
END START
```

**Figure 7-12**

Locating the data segment.



15. Assemble the program. This time there are no errors. The error is fixed by the segment override prefix code "DS:". It tells the assembler that DATA1 is located in the data segment. Note that the code always precedes the operand symbol, whether it is the source or destination operand. This is a one-time fix. That is, it only affects the instruction where it is located.

The second fix involved a second ASSUME directive. Unlike the segment override, it affects all of the data references that follow the assume. In addition, it will supersede a preceding assumption about the same segment register. For example, suppose the first ASSUME directive read:

```
ASSUME CS:PROG_CODE,DS:PROG_OLD_DATA,SS:PROG_STACK
```

The assembler will make all data references to the segment named PROG\_OLD\_DATA. When the assembler encounters a second ASSUME directive that reads:

```
ASSUME DS:PROG_DATA
```

all future data references will be made to the segment named PROG\_DATA.

If you wish to verify that your program code is properly assembled, link the object file and single-step through the program using the debugger.

16. Edit you EXE program's ASM file one more time. Refer to Figure 7-13. Add the four new lines indicated by arrows numbered "0". Then modify the two lines with arrows numbered "1". Now assemble the program, but DO NOT link it. Trying to link the program will cause the linker to generate an error message stating that it could not resolve an external reference (the call instruction).

```

TITLE EXPERIMENT 7 -- PROGRAM 5 -- GOING PUBLIC
;
PROG_STACK SEGMENT STACK
    DW    16 DUP (0FH)    ;Set up stack area
TOP_OF_STACK LABEL WORD    ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_DATA SEGMENT
0 → PUBLIC DATA2
DATA1 DB    8 DUP (0AAH)    ;Set up byte-sized data area
DATA2 DW    8 DUP (0BBBBH) ;Set up word-sized data area
PROG_DATA ENDS
;
0 → EXTRN  COUNT_DOWN:NEAR
;
1 → PROG_CODE SEGMENT PUBLIC
    ASSUME CS:PROG_CODE,SS:PROG_STACK
START: MOV    AX,PROG_STACK ;Never load a segment register direct
        MOV    SS,AX        ;Use an intermediate register
        MOV    SP,OFFSET TOP_OF_STACK ;Point to the top of stack
                                                ;immediately after loading SS register
        MOV    AX,PROG_DATA ;Again, indirectly load the
        MOV    DS,AX        ;segment register
        MOV    BL,DS:DATA1  ;Get the first byte of data
        ASSUME DS:PROG_DATA
        LEA    BX,DATA2     ;Point to the first word of data
0 → CALL    COUNT_DOWN    ;Run the external subroutine
0 → PUBLIC BEGIN
1 → BEGIN: MOV    DX,[BX]   ;Get the first word of data
        INT    3           ;Return to the debugger
PROG_CODE ENDS
END    START

```

**Figure 7-13**

Modifying the EXE program so that it can be linked with another EXE program.

- Using a new program name, call up the editor and enter the program listed in Figure 7-14. Assemble the program. This will be linked to the previous program.

```
TITLE EXPERIMENT 7 -- PROGRAM 6 -- TIME DELAY SUBROUTINE
;
    EXTRN DATA2:WORD,BEGIN:NEAR
;
PROG_CODE SEGMENT PUBLIC
    ASSUME CS:PROG_CODE
    PUBLIC COUNT_DOWN
COUNT_DOWN:
    MOV     CX,DS:DATA2
AGAIN:    LOOP    AGAIN
        RET
        JMP     BEGIN
        INT     3
PROG_CODE ENDS
END
```

**Figure 7-14**

External program to be linked with original EXE program.

## Discussion

Before you link the two programs, let's examine their structure. The six changes you made in the original EXE program allow it to communicate with the program that will be linked to it. The first change made the contents of DATA2 accessible to any linked program. The second change specified that the label COUNT\_DOWN was located in another program that would be combined with this program. Notice that the EXTRN directive is located outside of any segment area. This is important. If you place it inside a segment, the assembler assumes that the reference is to another program segment that will be combined with its segment. Depending on how the program segments are linked, this could cause an error. To make sure there is no chance of error, the EXTRN directive should always be placed outside of any segment area reference in a program. The third change in this program made the code segment "public," and thus combinable with any other public code segment with the same name. The fourth change added an instruction that would send the MPU off to a subroutine located within a linked program. Since this is a "near" call, the subroutine will have to be linked to the code segment that contains the call. The fifth change made the label BEGIN available to all linked programs. The last change added the label BEGIN to the move instruction. This will be used as a target for a jump from the linked program.

Your last program is a simple time delay that uses the value stored at DATA2 as the count. Two external references are made in this program. Notice that, as before, both references are made outside of the segment area. The first tells the assembler that DATA2 is located in an external program, and that it is a word-sized data reference. The second tells the assembler that the label BEGIN is located in an external program, but it will be a near reference after linking. The code segment name matches the code segment name in the other program, and it has the combine-type PUBLIC. Thus it will share the same segment area with the other program. The ASSUME directive identifies the code segment for the assembler. The PUBLIC directive makes the label COUNT\_DOWN accessible from external programs. The first instruction will move the contents of DATA2 into the CX register. Segment override "DS:" tells the assembler that the data will be addressed through the Data Segment register. The jump instruction serves no useful function other than to illustrate another external operation. We added the instruction:

INT 3

to the program to make it easier to identify the end of the program code when you examined it with the debugger. Recall that the hexadecimal code for that instruction is "CC". Finally, notice that the argument

“START” is missing from the END directive statement. This is important. When you combine programs, only **one** of the programs can contain an END directive argument that identifies the beginning of a “combined” program. Additional arguments will cause a link error.

Figure 7-15 shows the listing of this program. There are a number of areas of interest. First, the two-byte code for the move data instruction is followed by four zeros and the letter “E”. The four zeros reserve space in the program for the offset address of the data within the data segment. The letter “E” tells the linker that it must identify the location of the data at “link time” and fill in the appropriate offset value. This also applies to the jump instruction.

```

The Microsoft MACRO Assembler          05-11-84    PAGE    1-1
EXPERIMENT 7 -- PROGRAM 6 -- TIME DELAY SUBROUTINE

1
2                                TITLE EXPERIMENT 7 -- PROGRAM 6 -- TIME
                                DELAY SUBROUTINE
3                                ;
4                                EXTRN DATA2:WORD,BEGIN:NEAR
5                                ;
6      0000                      PROG_CODE SEGMENT PUBLIC
7                                ASSUME CS:PROG_CODE
8                                PUBLIC COUNT_DOWN
9      0000                      COUNT_DOWN:
10     0000 8B 0E 0000 E          MOV     CX,DS:DATA2
11     0004 E2 FE                AGAIN: LOOP AGAIN
12     0006 C3                   RET
13     0007 E9 0000 E            JMP     BEGIN
14     000A CC                   INT     3
15     000B                      PROG_CODE ENDS
16                                END

The Microsoft MACRO Assembler          05-11-84    PAGE    Symbols
                                -1
EXPERIMENT 7 -- PROGRAM 6 -- TIME DELAY SUBROUTINE

Segments and groups:

      Name                      Size  align  combine class
-----
PROG_CODE. . . . .              000B  PARA   PUBLIC

Symbols:

      Name                      Type   Value  Attr
-----
AGAIN. . . . .                  L NEAR 0004  PROG_CODE
BEGIN. . . . .                  L NEAR 0000  External
COUNT_DOWN. . . . .           L NEAR 0000  PROG_CODE Global
DATA2. . . . .                  V WORD 0000  External

Warning Severe
Errors Errors
0      0

```

**Figure 7-15**

Source listing of the external program.

Down in the “symbols table,” the combine-type for the segment is identified as PUBLIC. Next, the label BEGIN is given a different kind of attribute. Instead of a segment name (for its location), it’s given the attribute “External.” This gives you two characteristics of the label. First, it is not associated with a segment (no segment name). Second, it indicates that the label is not located within this program. The label COUNT\_DOWN is given an additional attribute “Global.” This is another way of saying that the label is “public.” Because of the segment name, you also know that it is located within the segment area called PROG\_CODE. Finally, the name DATA2 is given a type “V WORD.” The “V” indicates that it is an unidentified variable, while “WORD” indicates it is a word-sized variable. The “External” attribute described earlier also applies here.

## Procedure Continued

18. First read this step. Then link the two programs you just assembled. Use the following format:

**LINK <file-name1> + <file-name2>;**

where “file-name1” and “file-name2” represent the names of your two programs **without** the “.OBJ” file extension. The “+” symbol tells the linker to combine the two files named. The semicolon tells the linker to bypass any “link options.” Be sure to place the original EXE program file-name first. This is considered the “primary” program. The other program is considered a “supporting” program. You can link the programs in any order, but the programs are easier to trace through the debugger when you link the primary, or main, program first.

The EXE file generated by the linker will have the name of the first program in the list of programs to be linked. For example, if the first program is called P7-7.OBJ, the linked program will assume the name P7-7.EXE. Now link your two programs.

19. Call up the debugger and load your program into memory. Examine the registers — type “R” and RETURN — to determine the code or stack segment address. Then display the memory area that contains your program. Use the “D” command with the appropriate segment address value. Remember: for an early system, use the code segment base address; for a late system, use the stack segment base address. If you have an early system, your display will be similar to Figure 7-16. A late system will reverse the stack and code values.

Examine your displayed code. The main program begins offset 0000H (early) or 0040H (late). It ends on the next line of displayed code, with the hexadecimal value “CC”. Recall that this is the code for the type 3 interrupt instruction. Because both programs used the default align-type “paragraph,” the code from the second program begins on the next line of displayed code: offset 0020H (early) or 0060H (late). Again, the code ends with the hexadecimal value “CC”.

The five memory locations that separate the two groups of code are filled with zeros to maintain the paragraph alignment. They are insignificant, since they are ignored by the program. However, this does point out that you shouldn’t attempt to write code that “flows” directly from one program to another. Use a jump or call instruction to make the transition.

```
>R
AX=0000 BX=0000 CX=0070 DX=0000 SP=0020 BP=0000 SI=0000 DI=0000
DS=0A09 ES=0A09 SS=0A1E CS=0A19 IP=0000 NV UP DI PL NZ NA PO NC
0A19:0000 B81E0A MOV AX,0A1E
>D0A19:0000
0A19:0000 B8 1E 0A 8E D0 BC 20 00-B8 1C 0A 8E D8 8A 1E 00 B...P< .B...X...
0A19:0010 00 BD 1E 08 00 E8 08 00-8B 17 CC 00 00 00 00 00 .....h....L....
0A19:0020 8B 0E 08 00 E2 FE C3 E9-EE FF CC 00 00 00 00 00 ....b~Cin.L....
0A19:0030 AA AA AA AA AA AA AA AA-AA-BB BB BB BB BB BB BB *****; ; ; ; ; ; ; ;
0A19:0040 BB BB BB BB BB BB BB BB-BB-00 00 00 00 00 00 00 ; ; ; ; ; ; ; ; .....
0A19:0050 0F 00 0F 00 0F 00 0F 00-0F 00 0F 00 0F 00 0F .....
0A19:0060 0F 00 0F 00 0F 00 0F 00-0F 00 0F 00 0F 00 0F .....
0A19:0070 89 46 F6 8B 46 F6 3B 46-FC 77 4C 48 89 46 F4 8B .Fv.Fv;F!wLH.Ft.
>
```

**Figure 7-16**

Debugger display of program from early version system DOS.

20. Single-step through the program up to the call instruction. The Code Segment register contains the value `___H`, the Instruction Pointer register contains the value `___H`, and the Stack Pointer register contains the value `___H`. Execute the call instruction. The Code Segment register contains the value `___H`, the Instruction Pointer register contains the value `___H`, and the Stack Pointer register contains the value `___H`.

The Code Segment register value didn't change, proving that both program code groups reside within the same segment. The Instruction Pointer register changed from 0015H to 0020H, the offset address of the called instruction. Finally, the Stack Pointer register was decremented by two, indicating that the single-word "call return address" was stored in the stack. These three register values show that the MPU executed an **intra-segment** program transfer instruction.

21. Exit the debugger. Using the editor, change the EXTRN directive in your main program from:

```
EXTRN COUNT_DOWN:NEAR
```

to:

```
EXTRN COUNT_DOWN:FAR
```

Assemble the program.

22. Now modify the external program with the editor. Change the EXTRN directive from:

```
EXTRN DATA2:WORD,BEGIN:NEAR
```

to:

```
EXTRN DATA2:WORD,BEGIN:FAR
```

Then delete the PUBLIC combine-type from the SEGMENT directive. It should now read:

```
PROG.CODE SEGMENT
```

This makes the segment private; so that when you link the two programs, you will have two separate code segments. Normally, this isn't a good idea unless your code exceeds the 64K segment boundary. Making a segment private only complicates a program.



Because this code segment is now considered a far segment, you must code the return instruction for a far return from call. Recall that you do this by creating a dummy “far procedure.” Make the following changes to your program. Just before the RET instruction, add the beginning procedure directive:

```
DUMMY PROC FAR
```

Then just after the RET instruction, add the end procedure directive:

```
DUMMY ENDP
```

The return instruction will now assemble as a far return from call.

Since we are dealing with “far” program transfer instructions, there is one more change that should be made. Can you guess what that is? The jump instruction is currently identified as a “near” jump. To make it a “far” jump, you must add an assembler pointer operator. Change the instruction:

```
JMP BEGIN
```

to:

```
JMP FAR PTR BEGIN
```

That completes the modification of your external program. Assemble the program.

## Discussion

Figure 7-17 shows a listing of your external program. The code for two instructions have changed. Originally, the code for the return from call instruction was hexadecimal C3. Now that you have identified it as a far return, its code is hexadecimal CB. The same is true for the jump instruction. As a near jump, its code was hexadecimal E9 0000 E. As a far jump, its code is EA 0000 ---- E. The “EA” is the actual jump instruction; “0000” is a temporary address offset that will be recalculated by the linker; and “----” is the target code segment base address that will be determined by the program loader. As before, the ending “E” identifies this as an external operation that must be supported by the linker and loader. If you examine the main program listing, you will see that the far call instruction is coded in a similar fashion. Far program transfer instructions are the only way the Code Segment register contents can be changed during program execution.

The “Symbols” table in Figure 7-17 shows three changes from before. First, the code segment has no combine-type, making it a private segment. Second, the label BEGIN is now listed as a “far” label. Third, a description of the symbol DUMMY has been added. It has a “type” far procedure; a “value” of 0006 (begins with the sixth byte of code in the program); and its “attributes” are that it is located within segment PROG.CODE, and it is one byte long (Length = 0001).

## Procedure Continued

23. Link the two programs you just assembled. Use the same format you used earlier. Type “LINK <file-name1> + <file-name2>;” and RETURN. Remember, “file-name1” and “file-name2” represent the names of your two programs **without** the “.OBJ” file extension. The “+” symbol tells the linker to combine the two files named. The semicolon tells the linker to bypass any “link options.” Be sure to place the main program file-name first.

As before, the EXE file generated by the linker will have the name of the first program in the list of programs to be linked. For example, if the first program is called P7-9.OBJ, the linked program will assume the name P7-9.EXE.

The Microsoft MACRO Assembler      05-14-84    PAGE    1-1  
 EXPERIMENT 7 -- PROGRAM 8 -- PRIVATE TIME DELAY SUBROUTINE

```

1          TITLE EXPERIMENT 7 -- PROGRAM 8 -- PRIV
           ATE TIME DELAY SUBROUTINE
2          ;
3          EXTRN DATA2:WORD,BEGIN:FAR
4          ;
5          0000      PROG_CODE SEGMENT
6                   ASSUME CS:PROG_CODE
7                   PUBLIC COUNT_DOWN
8          0000      COUNT_DOWN:
9          0000      BB 0E 0000 E      MOV     CX,DS:DATA2
10         0004      E2 FE      AGAIN: LOOP AGAIN
11         0006      DUMMY PROC FAR
12         0006      CB      RET
13         0007      DUMMY ENDP
14         0007      EA 0000 ---- E      JMP     FAR PTR BEGIN
15         000C      CC      INT     3
16         000D      PROG_CODE ENDS
17         END
  
```

The Microsoft MACRO Assembler      05-14-84    PAGE    Symbols  
 -1  
 EXPERIMENT 7 -- PROGRAM 8 -- PRIVATE TIME DELAY SUBROUTINE

Segments and groups:

Name	Size	align	combine	class
PROG_CODE. . . . .	000D	PARA	NONE	

Symbols:

Name	Type	Value	Attr
AGAIN. . . . .	L NEAR	0004	PROG_CODE
BEGIN. . . . .	L FAR	0000	External
COUNT_DOWN . . . . .	L NEAR	0000	PROG_CODE Global
DATA2. . . . .	V WORD	0000	External
DUMMY. . . . .	F PROC	0006	PROG_CODE Length

=0001

Warning Severe  
 Errors Errors  
 0 0

**Figure 7-17**

Source listing of the external program after it is changed to a private combine-type.

24. Call up the debugger and load your program into memory. Examine the registers — type “R” and RETURN — to determine the code or stack segment address. Then display the memory area that contains your program. Use the “D” command with the appropriate segment address value. Remember: for an early system, use the code segment base address; for a late system, use the stack segment base address. If you have an early system, your display will be similar to Figure 7-18. The main code segment is followed by the data, then the stack, and finally the secondary, or private, code last. A late system will arrange the segments so the stack is first, data next, “main” code third, and secondary code last. Although it appears that both code segments share the same segment area in a late system, those of you who have a late system will find that isn’t true. Each code segment has its own segment base address.

```

>R
AX=0000 BX=0000 CX=006D DX=0000 SP=0020 BP=0000 SI=0000 DI=0000
DS=0A09 ES=0A09 SS=0A1D CS=0A19 IP=0000 NV UP DI PL NZ NA PD NC
0A19:0000 B81D0A MOV AX,0A1D
>D0A19:0000
0A19:0000 B8 1D 0A 8E D0 BC 20 00-B8 1B 0A 8E D8 8A 1E 00 8...P< .8...X...
0A19:0010 00 8D 1E 08 00 9A 00 00-1F 0A 8B 17 CC 00 00 00 .....L...
0A19:0020 AA AA AA AA AA AA AA AA-BB BB BB BB BB BB BB *****; ; ; ; ; ; ; ;
0A19:0030 BB BB BB BB BB BB BB BB-00 00 00 00 00 00 00 00 ; ; ; ; ; ; ; ; .....
0A19:0040 0F 00 0F 00 0F 00 0F 00-0F 00 0F 00 0F 00 0F 00 .....
0A19:0050 0F 00 0F 00 0F 00 0F 00-0F 00 0F 00 0F 00 0F 00 .....
0A19:0060 8B 0E 08 00 E2 FE CB EA-1A 00 19 0A CC 0F 89 46 ....b~Kj....L..F
0A19:0070 F6 83 F8 FF 75 20 FF 76-0C 9A 0A 00 1D 0D 89 DF v.x.u .v....._
>

```

**Figure 7-18**

Early system debugger display of combined program with private code segment.

Examine your displayed code. The main program code begins offset 0000H (early) or 0040H (late). In both cases, the secondary program code begins at offset 0060H in the display. However, when you execute the code, you will find that it actually begins at address offset 0000H, with a new segment base address. Both the main and secondary code segments end with the code for a type 3 interrupt, hexadecimal CC. For the secondary code segment, this is at address offset 006CH in your display.

25. Single-step through the program up to the call instruction. The Code Segment register contains the value `---_H`, the Instruction Pointer register contains the value `---_H`, and the Stack Pointer register contains the value `---_H`. Execute the call instruction. The Code Segment register contains the value `---_H`, the Instruction Pointer register contains the value `---_H`, and the Stack Pointer register contains the value `---_H`.

This time, the Code Segment register value changed, proving that the program code groups reside within different segments. The Instruction Pointer register changed from 0015H to 0000H, the offset address of the called instruction within the new code segment. Finally, the Stack Pointer register was decremented by four, indicating that a doubleword “call return address” was stored in the stack — the old CS register value and the old IP register value. The fact that all three register values changed shows that the MPU executed an **intersegment** program transfer instruction.

26. Single-step to the loop instruction. The CX register contains the value `---_H`. This is the value stored at DATA2. Single-step one more time. The CX register contains the value `---_H`.

The CX register holds the loop count. When the loop instruction was executed, the CX register was decremented and the MPU looped back to the loop instruction.

27. We want you to execute the return from call instruction. To save time, change the loop count to 0001H. Type “RCX” and RETURN. The debugger will respond with the specified register name and its current contents. Type the value you want the register to contain “0001” and RETURN.
28. Single-step one more time. The last loop instruction has executed and the debugger is pointing at the return instruction. Depending on the system, the debugger display of the return instruction mnemonic is different. Early systems will display:

```
RET L
```

while late systems will display:

```
RETF
```

In both cases, the debugger is indicating a far return.

29. Examine the memory area that contains your program. Use the “D” command with the appropriate segment address value. Remember: for an early system, use the code segment base address; for a late system, use the stack segment base address. Notice that the stack area contains a number of words of data. Recall that the debugger uses the program stack to temporarily store data. However, it always makes sure that any program data that is pushed into the stack is at the top of the stack. Thus, the Stack Pointer is always pointing at the “program” top of stack and not the top of stack being used by the debugger.

The first word that is stored in the stack is the original Code Segment register contents. The next word in the stack is the original Instruction Pointer contents. Thus, when the return from call instruction is executed, you can expect the CS register to be loaded with the value `___H` and the IP register to be loaded with the value `___H`.

30. Before you execute the return from call instruction, record the following register values:

Code Segment register `___H`  
Instruction Pointer register `___H`  
Stack Pointer register `___H`

Execute the return instruction. Record the following register values:

Code Segment register `___H`  
Instruction Pointer register `___H`  
Stack Pointer register `___H`

The MPU is now pointing at the instruction following the original call instruction in the main program. Do the CS and IP register values match the values you recorded in Step 29? If they don't, you weren't looking at the top of the stack, or you forgot that words are stored in memory low byte first. You can't re-examine the stack. As soon as the MPU popped the data from the stack, the debugger filled-in the empty area with its own data. It knows what part of the stack is available through the Stack Pointer. With the Stack Pointer pointing at offset 0020H, the debugger will use the stack area beginning with offset 0019H. The important point to remember is that the Stack Pointer will always point at the last word your program pushed into the stack.

31. The last instruction we want you to examine is the far jump in the external program. Since we don't know the code segment base address for your program, we're going to let your program load the CS register for you. First, change the contents of your Instruction Pointer to the offset address of the call instruction. Type "RIP" (register Instruction Pointer) and RETURN. Then enter the value "0015" and RETURN. Finally, single-step through the call instruction.

Now that the CS register is set, load the offset address of the jump instruction into the Instruction Pointer. Type "RIP" and RETURN. Then enter the value "0007" and RETURN. Display the instruction by typing "R" and RETURN. The code for the far jump instruction is five bytes long. The first byte is the actual jump code that is decoded by the MPU. The next two bytes contain the offset address to the jump target. The last two bytes contain the segment base address to the "far" code segment. When the jump is executed, the Instruction Pointer and Code Segment registers are loaded with the values contained in the instruction code. These values should be:

Code Segment register \_\_\_H  
Instruction Pointer register \_\_\_H

Execute the instruction. Do the values match?

## Discussion

This portion of the experiment has presented the important areas you should understand in order to write EXE-type programs. We didn't provide examples of every variation described in the text. However, with the information provided in this unit and your system reference manuals, you should be able to handle every programming situation.

It should be apparent by now that there are trade-offs between COM and EXE programs. EXE programs give you more control over the program design, but they also add to the complexity of the program. COM programs are not quite as versatile, but they are more compact and easily written. These trade-offs will, to some extent, dictate which program-type you will use. If you still don't feel confident with the EXE style program, don't worry, we will be using it often throughout the remainder of the course.

The last part of the experiment will introduce you to the concept of direct data I/O. Because of the way every manufacturer treats I/O in his microcomputer, a graphic demonstration is difficult. A majority of the I/O operations are very similar between IBM and ZENITH, but the application is quite different. Rather than fill the experiment with product and model exceptions, we'll limit the example to the CRT Controller interface.

```

TITLE EXPERIMENT 7 -- PROGRAM 9 -- I/O
;
PROG_STACK SEGMENT STACK
    DW    80 DUP (0)          ;Set up stack area, 80 bytes minimum
                                ;to handle system interrupt
TOP_OF_STACK LABEL WORD      ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_CODE SEGMENT PUBLIC
    ASSUME CS:PROG_CODE,SS:PROG_STACK
;
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
;Select the two register equate values that match your system. Right
;now, the equate values for Zenith systems with bit mapped video are
;selected. For systems that use an IBM or IBM compatible video circuit
;board, place a semicolon in front of each of the two Zenith equates,
;then remove the semicolons from in front of the two IBM equates that
;match the board type in your system.
;
INDEX_REG EQU 0DCH           ;Zenith video register select port
DATA_REG  EQU 0DDH           ;Zenith video register R0-R17 port
;
;INDEX_REG EQU 03B4H         ;IBM monochrome adapter circuit board
;DATA_REG  EQU 03B5H         ;IBM monochrome adapter circuit board
;
;INDEX_REG EQU 03D4H         ;IBM color/graphics adapter C.B.
;DATA_REG  EQU 03D5H         ;IBM color/graphics adapter C.B.
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
;
CUR_START EQU 0AH           ;Cursor start register address
CUR_STOP  EQU 0BH           ;Cursor stop register address
;
START: MOV  AX,PROG_STACK    ;Never load a segment register direct
      MOV  SS,AX            ;Use an intermediate register
      MOV  SP,OFFSET TOP_OF_STACK ;Point to the top of stack
                                ;immediately after loading SS register
      PUSH DS                ;Save segment far return value
      SUB  AX,AX            ;Zero AX register
      PUSH AX                ;Save offset of zero for far return
;
      CALL CURSOR_STOP      ;Prepare CRTC for ending cursor line
      MOV  AL,07H          ;Ending cursor scan line
      OUT  DX,AL            ;Write ending line to register
;
      CALL CURSOR_START    ;Prepare CRTC for beginning cursor line
CURSOR_SIZE:
      MOV  CX,07H          ;Cursor size loop count
      MOV  AL,01H          ;Starting cursor scan line value
CHANGE: OUT  DX,AL          ;Write starting line to register
      INC  AL                ;Make cursor one scan line smaller
      MOV  BX,0FFFFH       ;Add some delay to see cursor change
DELAY:  DEC  BX              ;Count down delay
      JNZ  DELAY            ;Is delay done?
      LOOP CHANGE          ;Yes, change cursor size
;
      MOV  AH,0BH          ;Keyboard status interrupt code
      INT  21H             ;Check status -- FF = key pressed
      CMP  AL,0FFH         ;Key pressed?
      JE   STOP            ;AL=FF, end program, otherwise continue
      JMP  CURSOR_SIZE     ;Start all over again

```

**Figure 7-19A**  
I/O program listing.



## Procedure Continued

32. Exit the debugger, call up the editor, and enter the program listed in Figure 7-19. The program is designed to run on any 16-bit variation of Zenith, IBM, or IBM compatible microcomputer. However, you must “configure” two areas of the program to match your system. Right now, the program is configured to run on a Zenith system using a bit mapped video display. Follow the program directions for selecting the two appropriate I/O port equate directives (near the beginning of the program) and the appropriate final cursor code (near the end of the program).

```

;
COMMENT~
Select the final cursor code that matches your system. Zenith bit
mapped video is currently active. To select the cursor code for any
IBM or IBM compatible system, place semicolons in front of the Zenith
lines, then remove the semicolons from in front of the IBM code lines.~
;
;ZENITHZENITHZENITHZENITHZENITHZENITHZENITHZENITHZENITHZENITHZENI
STOP:  MOV    AL,48H           ;Zenith type cursor code to set flash
                                ;rate and beginning cursor line number
        OUT    DX,AL           ;Write code to cursor start register
        CALL   CURSOR_START    ;Prepare the CRTC for cursor stop data
        MOV    AL,08H         ;Ending cursor line number
        OUT    DX,AL           ;Write code to cursor stop register
;ZENITHZENITHZENITHZENITHZENITHZENITHZENITHZENITHZENITHZENITHZENI
;
;IBMIIBMIBMIIBMIBMIIBMIBMIIBMIBMIIBMIBMIIBMIBMIIBMIBMIIBMIBMI
;STOP:  MOV    AL,06H           ;IBM type cursor code
;        OUT    DX,AL           ;Write code to cursor stop register
;IBMIIBMIBMIIBMIBMIIBMIBMIIBMIBMIIBMIBMIIBMIBMIIBMIBMIIBMIBMI
;
EXIT   PROC    FAR             ;Set up far return and
        RET                                ;exit program gracefully
EXIT   ENDP                    ;through built-in system interrupt
;
;CURSOR_START:
        MOV    DX,INDEX_REG    ;Address of register select port
        MOV    AL,CUR_START    ;Address of cursor start register
        OUT    DX,AL           ;Select the cursor start register
        MOV    DX,DATA_REG     ;Address of register R0-R17 port
        RET
;
;CURSOR_STOP:
        MOV    DX,INDEX_REG    ;Address of register select port
        MOV    AL,CUR_STOP     ;Address of cursor stop register
        OUT    DX,AL           ;Select the cursor stop register
        MOV    DX,DATA_REG     ;Address of register R0-R17 port
        RET
;
;
PROG_CODE ENDS
        END    START

```

**Figure 7-19B**

Continuation of the I/O program listing.

33. Assemble and link the program. Execute the program by typing the program name without the file extension EXE.

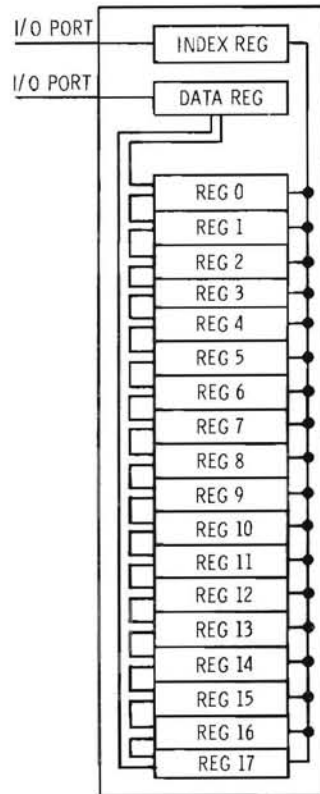
The program will display a new “cursor” symbol composed of seven “scan lines.” After a momentary pause, the top line will disappear. After another pause, the next line will disappear. This will continue until there is only one line left, then the sequence will repeat. To exit the program, press any of the alphanumeric, symbol, or space keys. Control will be returned to the system, the cursor will return to its original configuration, and the key that you pressed will be echoed on the display.

## Discussion

In addition to illustrating I/O addressing, the program uses many of the concepts presented earlier in the course. It also introduces a new variation of interrupt 21H, a new method for exiting an EXE program, and a new assembler directive, the COMMENT.

Because the program you entered is an EXE-type program, it had to establish a program stack segment. Recall that when a program contains a system interrupt, the stack should contain at least 80 memory word locations to support the interrupt subroutines. Thus, the first part of the program established an 80-word stack.

The program code segment began with a number of equate statements. The first three pairs identified the two I/O port addresses for the 68A45 Cathode Ray Tube Controller (CRTC). The three pairs of addresses are necessary because there are three possible system configurations, and each system uses a different port address combination. Figure 7-20 shows why two ports are needed to address the CRTC.



**Figure 7-20**

Simple diagram of the 68A45 CRT Controller programmer accessible control registers.

The CRTC contains 20 programmer accessible registers. Eighteen of these, Register 0 through Register 17, are used to control or provide feedback from the display. In order to access any of these registers, you first must identify the register. You do this by sending the register number to the Index Register, through the I/O port assigned to that register. The Index Register, in turn, enables the specified register for a read or write. Now you can access the specified register through the I/O port assigned to the Data Register.

In your program, the two registers of interest are 10 and 11. These control the characteristics of the display cursor. Register 10 identifies the starting cursor scan line as well as the blink function. Register 11 identifies the ending scan line number. Therefore, if Register 10 contains the value 01H and Register 11 contains the value 07H, the cursor will be seven scan lines tall. If both contain the value 07H, the cursor will contain one scan line. The last two program equate statements identify the two cursor registers.

The first three program instructions are by now quite familiar. They load the Stack Segment register and the Stack Pointer register. The next three instructions, however, are quite new.

When an EXE program is loaded into memory, the program loader places the "system return address" at a location that is identified by the contents of the Data Segment register and offset value zero. This information must be placed in the program stack before any other program instructions are executed. At the end of the program, a far return will be executed, and the information will be loaded into the CS and IP registers. They will then point to the physical address of the monitor subroutine that will return the MPU to the system.

Now that the registers are properly set up, the main program can begin. The first step is to address the cursor stop register. A subroutine call performs the function. The subroutine is located at the end of the program.

The first instruction loads the port address of the CRTC Index Register into the DX register. Recall that variable port addressing, through the DX register, is required for I/O addresses beyond the first 256. The IBM addresses fit into this category. Next, the cursor stop register number is loaded into the AL register. Then the AL register contents are output to the port identified by the DX register. The CRTC Index Register is now pointing to the cursor stop register. Next, the DX register is loaded with the I/O port address of the CRTC Data Register. The CRTC is finally ready to receive the ending scan line number for the cursor. The return instruction sends the MPU back to the instruction following the call.

Here, the scan line number 07H is loaded into the AL register. Then it is output to the I/O port address pointed to by the DX register. The ending scan line number for the cursor is now loaded into the CRTC. The next step is to load the starting scan line number.

As you could see in the display, the starting scan line number, and thus the number of cursor scan lines, was continuously changing. This is a simple process once the cursor start register in the CRTC is selected. Again, the register selection is handled by a subroutine. The CURSOR\_START subroutine is identical to the CURSOR\_STOP subroutine except for the values loaded into the DX and AL registers. When it is complete, the DX register is pointing to the CRTC Data Register and the CRTC Index Register is pointing to the Cursor Start Register. The next 13 steps control the size of the cursor and test for a keyboard entry.

A small loop controls the cursor size. Since there are seven levels of cursor size, the value 07H is loaded into the Count register. Then the cursor starting value is loaded into the AL register and output to the CRTIC. Next, the AL register is incremented to reduce the cursor size by one line. To make it possible to see the cursor change size, a short delay is added to the cursor loop. The BX register is loaded with 0FFFFH, the register is decremented, and the value is tested for zero. If it isn't zero, BX is decremented again. As soon as it reaches zero, the cursor loop continues, and the new cursor start value is output to the CRTIC.

After every cursor loop cycle, the keyboard is tested to see if you pressed a key. This is how you exit the program. Interrupt 21H, function 0BH, tests the status of the keyboard. If a key has been pressed, the interrupt will return the value 0FFH to the AL register. If no key has been pressed, the value 00H will be returned to the register.

The process begins by loading the AH register with the function number 0BH. The interrupt is called. Then the AL register is compared with the value 0FFH to see if a key has been pressed. A match causes the MPU to jump to the subroutine labeled STOP. Otherwise, the MPU jumps back to the beginning of the cursor modification loop.

Two STOP subroutines were supplied with the program to accommodate the two cursor styles. In the case of IBM, the cursor start address is set to line 06B. This produces a two-line cursor. No other changes are necessary.

For Zenith, both the start and stop cursor line values must be changed. First, the value 48H is output to the cursor start register in the CRTIC. This value sets the cursor start line value to line eight. It also enables the cursor blink function. After the cursor start register is loaded, the cursor stop register is addressed through the CURSOR\_STOP subroutine. Then the line value 08H is loaded into the register. This sets the cursor stop line value to line eight. As a result, the Zenith cursor is a single blinking line.

You didn't have to enable the IBM cursor blink function. This is because it is controlled by system software, rather than through the CRTIC.

The last part of the program is a far return. When the instruction is executed, the return address to the system is popped from the stack and loaded into the CS and IP registers. Always use this method to exit an EXE-type program.

One last point before we end this experiment. Two different methods were used to add multiple line comments to the program. The first has been used before. A semicolon is placed before each line to tell the assembler that the line is for comments only. The second method uses the assembler directive **COMMENT** to identify program comments. Notice that we placed the symbol for a “tilde” (horizontal S) immediately after the directive. This identifies the beginning of the comments. Another tilde is placed immediately after the comments to indicate the end. You don’t have to use a tilde, any mark will do, but you must use a symbol that is unique and won’t be found within the comments. The **COMMENT** directive is useful when you are writing a long comment that may need to be edited. You don’t have to worry about placing a semicolon in front of each comment line. Placing borders around a comments section is optional. We used them to make it easier for you to identify unique groups of code and data.

This completes the Experiment for Unit 7. If you anticipate a need for accessing specific I/O ports within your system, now is a good time for you to work a little more with I/O port addressing. Your technical manual lists all of the I/O ports and gives a brief description of how to use them. When you are finished, proceed to the Unit 7 Examination.

## UNIT 7 EXAMINATION

1. An EXE-type program must contain at least two segment areas. They are the \_\_\_\_\_ and \_\_\_\_\_ segments.
2. Before you execute the code in an EXE-type program, you must initialize the \_\_\_\_\_ registers.
3. The default segment register for a data move operation in an EXE-type program is the \_\_\_\_\_ Segment register.
4. You can move the stack segment base address value directly into the SS register. \_\_\_\_\_  
True/False
5. You are allowed a maximum of five ASSUME statements in an EXE-type program. \_\_\_\_\_  
True/False
6. The segment attribute \_\_\_\_\_ specifies how a program is to be linked and loaded into memory.
7. The segment attribute \_\_\_\_\_ specifies how the various segments are grouped within a program.
8. The segment attribute \_\_\_\_\_ specifies how identically named segments are arranged in a program.
9. Program transfer within a segment is called \_\_\_\_\_ addressing.
10. Program transfer from one segment to another is called \_\_\_\_\_ addressing.

11. The assembler directive \_\_\_\_\_ identifies a symbol that is available to an external program.
12. The assembler directive \_\_\_\_\_ identifies a symbol that will be used by an external program.
13. The assembler directive \_\_\_\_\_ provides a convenient means for entering long program descriptions.
14. Near call instructions push \_\_\_\_\_ word(s) into the stack before transferring to the target address.
15. Return instructions that are not part of a procedure are always \_\_\_\_\_ program transfer instructions.
16. Variable I/O port addresses must be stored in the \_\_\_\_\_ register.
17. There are a maximum of \_\_\_\_\_ fixed I/O port locations available to the 8088/8086 MPU.



## EXAMINATION ANSWERS

1. An EXE-type program must contain at least two segment areas. They are the **Code** and **Stack** segments.
2. Before you execute the code in an EXE-type program, you must initialize the **segment** registers.
3. The default segment register for a data move operation in an EXE-type program is the **Data** Segment register.
4. **False.** Before you can move the stack segment base address value into the SS register, you must move the value into an intermediate register such as the AX register.
5. **False.** You are allowed an unlimited number of ASSUME statements in an EXE-type program.
6. The segment attribute **align-type** specifies how a program is to be linked and loaded into memory.
7. The segment attribute **combine-type** specifies how the various segments are grouped within a program.
8. The segment attribute '**class**' specifies how identically named segments are arranged in a program.
9. Program transfer within a segment is called **intra-segment** addressing.
10. Program transfer from one segment to another is called **inter-segment** addressing.

11. The assembler directive **PUBLIC** identifies a symbol that is available to an external program.
12. The assembler directive **EXTRN** identifies a symbol that will be used by an external program.
13. The assembler directive **COMMENT** provides a convenient means for entering long program descriptions.
14. Near call instructions push **one** word into the stack before transferring to the target address.
15. Return instructions that are not part of a procedure are always **near** program transfer instructions.
16. Variable I/O port addresses must be stored in the **DX** register.
17. There are a maximum of **256** fixed I/O port locations available to the 8088/8086 MPU.

## SELF-REVIEW ANSWERS

1. A **segment** is a logical unit of memory that is 64K bytes long.
2. **False.** Segments may be adjacent to each other **and** they can overlap.
3. **False.** The assembler “assumes” that instructions are located in the code segment and data are located primarily in the data segment, although there are some exceptions in the case of data.
4. Every EXE program must contain at least **two** segment areas: code and stack.
5. Every segment must be identified by a **unique** name.
6. The argument **STACK** must follow the stack SEGMENT directive so that the assembler knows which segment defines the stack area in memory.
7. **False.** The assembler directive LABEL only identifies a memory location.
8. The beginning of a segment is identified by the **SEGMENT** directive.
9. The end of a segment is identified by the **ENDS** directive.
10. The **code** segment contains the ASSUME directive.
11. Before any memory operations are performed, the first instructions in an EXE program should be used to initialize the Stack Segment register, the Stack Pointer register, and any data segment register that will be used by the program.
12. The **Code Segment** and **Instruction Pointer** registers don't have to be initialized by the program code.
13. A physical address in memory is determined by combining a segment **base** address with a **logical, or offset**, address.
14. An early version linker arranges the code, data, and stack segments so that the **code** segment is first, the **data** segment is next, and the **stack** segment is last. Late version linkers place the **stack** segment first, **data** segment next, and **code** segment last.
15. Data addressed by the **Base Pointer** register is, by default, within the stack segment.

16. You can override the assembler's choice of segment register in a memory operation with the **segment override** assembler operator.
17. You can change the assembler's "assumptions" about a segment with the **ASSUME** directive.
18. The default "align-type" attribute in a **SEGMENT** directive statement is **PARA**.
19. Align-type **PAGE** specifies that the segment can begin at any address that is divisible by 256.
20. Align-type **PARA** specifies that the segment can begin at any address that is divisible by 16.
21. Align-type **BYTE** specifies that the segment can begin at any address in memory.
22. Align-type **WORD** specifies that the segment can begin at any even numbered address in memory.
23. The **linker** program uses the align-type attribute to determine how the object program segments will be arranged in memory.
24. If a segment has no "combine-type" attribute, the segment is considered **private**.
25. The combine-type attribute determines both the physical as well as the **logical** addressability of a segment.
26. Combine-type **PUBLIC** segments will be combined by the linker if their segment **names** are identical.
27. Combine-type **STACK** is similar to combine-type **PUBLIC**.
28. Three segments with combine-type **COMMON** and lengths of 5, 15, and 10 bytes will produce a segment **15** bytes long when combined.
29. Combine-type **MEMORY** is treated like combine-type **PUBLIC** by the linker.
30. To specify the physical location of a label within memory, you would use combine-type **AT <expression>**.

31. A secondary segment identifier is the segment attribute '**class**'.
32. **Intrasegment** addressing is restricted to the 64K byte boundary of a single code segment.
33. **Intersegment** addressing relates to program transfer between code segments.
34. COM-type programs use **intrasegment** addressing.
35. The assembler directive **PUBLIC** is used to identify those symbols in a program that can be accessed by other programs when they are linked.
36. The **EXTRN** directive tells the assembler that the indicated symbol is located outside of the current program.
37. The **EXTRN** directive identifies a symbol by its label or name and by its **type**.
38. **True**. All external references must have a matching public symbol.
39. **False**. All public symbols do not require a matching external reference.
40. A far jump between segments in a single program is identified by the assembler operator **FAR PTR**.
41. A far call between segments of two programs that have been linked is identified by the external directive type **FAR**.
42. When a far call is executed, the contents of the **Code Segment** register are pushed into the stack; then the contents of the **Instruction Pointer** register are pushed into the stack.

43. **False.** A “dummy” far procedure must be created in order to execute an **intersegment** return.
44. Memory is disabled and I/O is enabled by the **IN** and **OUT** instructions.
45. An I/O port can accommodate **eight** bits of data.
46. The address of a **fixed** port is defined as a constant.
47. The address of a **variable** port is pointed to by the contents of the DX register.
48. There are **256** 8-bit fixed ports available to the 8088/8086 MPU.
49. **False.** The AL register must be used as the destination operand for an IN instruction accessing an 8-bit port.
50. The source operand for an IN instruction that uses a fixed port is always an **immediate** value.
51. **True.** Both the fixed and the variable addressing methods can access the first 256 I/O port locations.

INSERT





*Unit 8*

**INTERRUPTS AND STRINGS**

## CONTENTS

Introduction .....	8-3
Unit Objectives .....	8-4
Unit Activity Guide .....	8-5
Interrupts .....	8-6
String Operations .....	8-25
Experiment .....	8-43
Unit 8 Examination .....	8-73
Examination Answers .....	8-74
Self-Review Answers .....	8-75

## INTRODUCTION

One of the most important functions of the MPU is interrupt handling. Recall that an interrupt is a command to stop whatever is in progress and perform another operation. Through the MPU, internal (program related) as well as external (hardware related) interrupt commands allow complete integration of the microcomputer system. Although we have been using interrupts throughout this course, we haven't fully explained their function or operation. This unit will describe all of the interrupts in detail.

After interrupts, there is only one major area of programming the 8088 MPU that we haven't covered. That area is string handling. In this context, a string is a group of consecutive memory locations that contain some form of data. You will learn how to initialize a string, move a string from one memory location to another, and test a string for a specific value.

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: Interrupt, interrupt request, non-maskable interrupt, interrupt service routine, interrupt vector (pointer) table, divide error, single step, handshaking, reset, DMA, string, string operation, and string element.
2. Describe how the INTR, INTA, HOLD, HLDS, and NMI control lines are used.
3. Use the following instructions in a program: INT, IRET, INTO, CLI, STI, CLD, STD, REP, REPNE, REPZ, REPE, REPZ, MOVS, MOVSB, MOVSW, CMPS, CMPSB, CMPSW, SCAS, SCASB, SCASW, LODS, LODSB, LODSW, STOS, STOSB, and STOSW.
4. Describe what occurs within the MPU during a typical interrupt.
5. State the purpose of the TF (Trap flag), IF (Interrupt flag), and DF (Direction flag).
6. List the priority in which the 8088 MPU services internal and external interrupts.
7. Name two basic types of interrupts.
8. State the contents of an interrupt vector, or pointer.
9. State the use for the single-step operation.
10. Name the two software interrupt instructions available with the 8088 MPU instruction set.
11. List the contents of the segment registers, IP register, Flag register, and queue, after a reset.
12. State the use of the CX register in a string operation.

## UNIT ACTIVITY GUIDE

	<b>Completion Time</b>
<input type="checkbox"/> Read the Section on “Interrupts.”	_____
<input type="checkbox"/> Complete Self-Review Questions 1-18.	_____
<input type="checkbox"/> Continue Reading the Section on “Interrupts.”	_____
<input type="checkbox"/> Complete Self-Review Questions 19-32.	_____
<input type="checkbox"/> Read the Section on “String Operations.”	_____
<input type="checkbox"/> Complete Self-Review Questions 33-47.	_____
<input type="checkbox"/> Continue Reading the Section on “String Operations.”	_____
<input type="checkbox"/> Complete Self-Review Questions 48-59.	_____
<input type="checkbox"/> Perform the Experiment.	_____
<input type="checkbox"/> Complete the Unit 8 Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

## INTERRUPTS

In order to maintain full control of your microcomputer system, you must have a way to halt program execution to service an internal microcomputer function or allow a peripheral access to the MPU. Otherwise, a program, no matter how long, will run until it is finished. As a result, important data from a peripheral may be lost. Program execution can be stopped through the use of an interrupt. Now in order to understand interrupts, you must be familiar with a number of terms. We will introduce these terms first and then continue on with a more in-depth description of interrupts.

Generally speaking, an **interrupt** is a temporary break in the normal execution of a program, after which program execution proceeds at the point of the break. The actions that the MPU takes in response to an interrupt are called the **interrupt service routine** or the **interrupt routine**. Responding to an interrupt is referred to as **servicing** an interrupt. You might think of an interrupt routine as nothing more than a “called” subroutine. When the MPU recognizes the interrupt, it jumps to the area in memory that holds the subroutine to service the interrupt.

There are two basic types of interrupts: the **external interrupt** (generated outside the MPU) and the **internal interrupt** (generated in response to some occurrence within the MPU or program). **Maskable** and **non-maskable interrupts** come under the general heading of external interrupts. The maskable interrupt is one that can be “ignored” by the MPU. Non-maskable interrupts, on the other hand, require an immediate response, usually to some catastrophic system failure. Let’s see how and why interrupts are used.

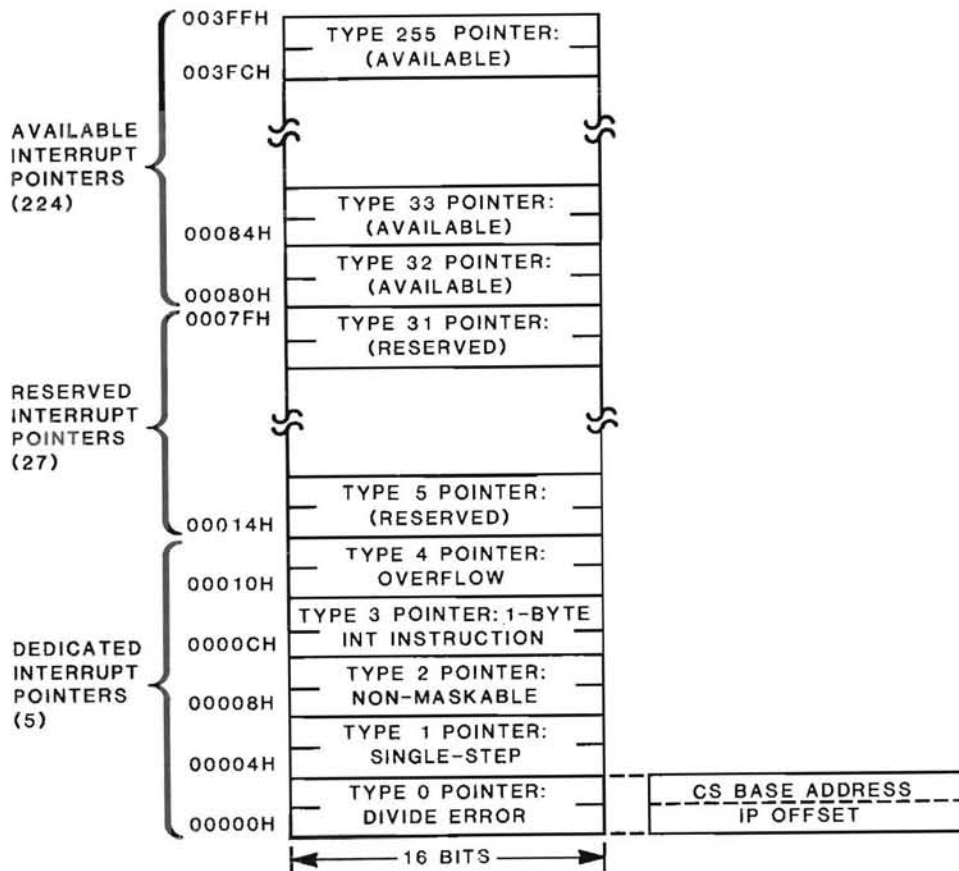
### Internal Interrupts

As mentioned earlier, an internal interrupt is generated in a program or when the MPU determines that some form of “problem” exists that requires immediate service. Because each type of interrupt has its own dedicated service routine, the MPU must know where that routine is located within memory. In the case of the 8088 MPU, the physical starting addresses for all of the interrupt service routines are stored in the first 1K of memory.

### INTERRUPT VECTOR TABLE

The dedicated area of memory that holds the starting address of all of the interrupt service routines is called the **interrupt vector table**, or **interrupt pointer table**. Figure 8-1 shows the table. Each address value is called a **vector**, or **pointer**, and it is assigned a “Type” number. The type numbers begin at physical address 00000H with type 0 and proceed to type 255 at physical address 003FCH.

Each vector, or pointer, is composed of four bytes of data. The first two bytes contain the Instruction Pointer offset address, while the next two bytes contain the Code Segment base address of the interrupt routine. When an interrupt is called, the MPU loads the IP and CS registers with the values found at the specified vector “type” address.



**Figure 8-1**  
Interrupt pointer table.

## INTERRUPT TYPES

As you can see, there are 256 different interrupt vectors, or pointers. The first five are dedicated. That is, these vectors are already used for specific purposes. The next 27 vectors are considered to be reserved, future 8088 MPU support integrated circuits may use any of these vectors. To prevent future system compatibility problems, you should not use these vectors. The last 224 vectors are available to you for program support. Later, you will see how you can use these vector locations with your own interrupt routines.

To get an idea of how the interrupt vector table is used, let's examine a common interrupt that is generated when the MPU senses a problem. The **divide error interrupt** occurs following the execution of a DIV or IDIV instruction if the quotient is larger than the destination register. For example, if you attempt to divide 0FFFFH by 01H, the quotient will be too large for the destination register, AL. Therefore, a divide error interrupt will result.

When this occurs, the MPU automatically generates a **type 0** interrupt. The MPU first completes execution of the current instruction. Then the current contents of the Flag register, the CS register, and the IP register are pushed into the stack. The CS and IP registers are saved just like any subroutine call operation. The Flag register is also saved, because there is no telling how the "flags" will be altered by the interrupt servicing routine.

With the registers saved, the new Code Segment and Instruction Pointer register values are loaded. These values are found at the type 0 location in the vector table. They point to the first instruction of the interrupt routine.

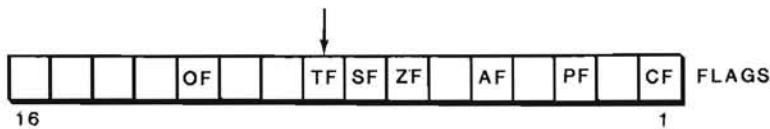
Interrupt servicing now begins. The MPU executes the routine in the same manner as any other subroutine. There is, however, one important difference — the routine is ended with a new instruction. This is the **IRET** (interrupt return) instruction. When the IRET instruction is executed, the IP value, the CS value, and the Flag values are popped from the stack and placed in their appropriate registers. Then program execution continues from the point of the interrupt.

All interrupts follow this same general pattern. Some occurrence, either internal or external, generates an interrupt. The MPU responds by completing execution of the current instruction, executing the interrupt service routine, and finally, continuing program execution.



Here's another example of an internal interrupt. A type 1 interrupt is generated whenever the programmer needs to execute a **single-step procedure**. The single-step procedure is designed to help the programmer correct, or debug, programs. In effect, when single step is implemented, the MPU is directed to a subroutine after execution of each instruction. This subroutine allows the programmer to examine all of the registers within the MPU to determine if the program is functioning as it should. The "Trace" routine in the debugger program operates in this fashion. However, before we can explain how the single-step interrupt works, we must take another look at the Flag register.

Figure 8-2 shows the 8088 Flag register. Note that there is a new flag shown in the eighth bit location of the register. This is the **TF** or **Trap flag**. Any time this flag is set, the MPU automatically generates a type 1 interrupt after the execution of an instruction. Therefore, this flag must be set to use the single-step routine.



**Figure 8-2**  
The trap flag.

Since the Trap flag is not normally set, you must write a routine that sets this flag. Because you cannot directly modify the contents of the Flag register, the routine is more than just a simple single-instruction operation. Our routine, shown below, first pushes the flags into the stack. Then it moves the value of the Stack Pointer register into the Base Pointer register. Remember, the BP register defaults to the stack segment in register indirect addressing. Thus, we can now modify the contents at the top of the stack — the Flag register value. The next instruction ORs the high byte in the stack with 01H to set what will become the TF bit of the Flag register. The last instruction pops the value in the stack back into the Flag register. The Trap flag bit is now set.

```

PUSHF                ;Store the flags in the stack
MOV  BP,SP           ;Get address of flags in stack
OR   BYTE PTR [BP]+1,01H ;Set the Trap flag bit
POPF                ;Retrieve the modified flags
    
```

As soon as the flags are popped back into the Flag register, the Trap flag is ready to trigger an interrupt. The single-step interrupt, however, is not generated until **after** the **next** instruction is executed. After the next instruction is executed, a type 1 interrupt is generated. The Flag, CS, and IP registers are pushed into the stack, and then the Trap flag is cleared. This last step is necessary, or the single-step routine would generate another type 1 interrupt.

The last instruction in the single-step routine is a return from interrupt instruction. When it is executed, the IP, CS, and Flag values are popped back into their respective registers. Popping the flags naturally returns the Trap flag to its **set** condition. If you want to exit the single-step process, the easiest method is to modify the Flag register value stored in the stack while in the single-step routine. Simply AND the high byte of the flag register value with 0FEH. Then when the flags are popped, the Trap flag will be cleared.

In addition to the two interrupts mentioned thus far, there are three other dedicated interrupts. Type 2 is called a **non-maskable interrupt**. This is a hardware-related interrupt that allows a peripheral device to signal the MPU that it needs servicing. We'll describe this interrupt when we cover external interrupts.

The last two interrupts are generated by instructions that you place in the program. The type 3 interrupt is called a **1-byte interrupt instruction**. It's also known as a **breakpoint instruction**. This is the instruction you used with the debugger when you wanted to execute part of a program and then stop. Breakpoints typically are inserted into programs during debugging as a way of displaying registers, memory locations, etc., at critical points in a program.

Although the breakpoint interrupt is an instruction, there is no mnemonic for it. To use the instruction, you must insert the instruction machine code (0CCH) into the assembled machine code for the program. Normally, you would let the debugger perform the operation. If you wish to load the code yourself, use the NOP instruction at every location you want to add a breakpoint. After the program is assembled, use the debugger or a similar program to examine and change the NOP machine code (90H) to the breakpoint code (0CCH).

Finally, the type 4 interrupt is called **interrupt on overflow instruction**. This software interrupt has a mnemonic, **INTO** (interrupt on overflow). This is a conditional interrupt. If the Overflow flag is clear, the instruction is ignored. On the other hand, if the Overflow flag is set, the type 4 interrupt is executed. As with every interrupt, the Flag, CS, and IP register values are pushed into the stack before the CS and IP registers are loaded with the vector address values.

By now, you should be wondering where these interrupt routines are located. Depending on the system, most are located in ROM (read only memory). When the microcomputer is switched on, it performs a number of setup or housekeeping operations. One of these operations involves loading the vector addresses of the various interrupt routines into the interrupt vector table. However, not all of the default interrupts may be supported by any one microcomputer. Check your Technical Manual to determine which are supported. Depending on your needs and your system, you may have to implement your own interrupt supporting routine. Naturally, you will have to load the vector table with the address of your interrupt routine.

This leads us to the last type of software interrupt instruction, the one with which you are most familiar, `INT <type>` (interrupt at vector type number). Recall that there are 224 available (not dedicated) address locations in the interrupt vector table. MS-DOS uses many of these interrupt locations to point to specific system routines. For example, the instruction:

```
INT 33
```

is the same as

```
INT 21H
```

the interrupt you have been using to display data and exit a program, among other things. When the MS-DOS system disk is “booted,” many interrupt routines are loaded into RAM. Their addresses are also loaded into the interrupt vector table. The DOS interrupts and function calls are listed in the Appendix of the system owner’s manual. Each number listed for an interrupt or function call represents an interrupt vector table type number. Thus, you can examine an interrupt routine with the debugger by first locating the vector address in the interrupt table. Keep in mind that the type numbers listed in the DOS owner’s manual are in hexadecimal.

You can also write special interrupt routines and use any of these interrupt vectors as a pointer. This can be very useful when you want to support a peripheral device with a unique interrupt service routine. Because external interrupts also reference vector type numbers, you can use a software interrupt to test the validity of a hardware supporting interrupt. This saves you the trouble of prompting the external device to generate the interrupt. We'll describe external interrupts in the next section.

Writing an interrupt routine is quite easy. You use the same format you use when you write a called subroutine. The only difference is that you end the routine with the IRET instruction instead of the RET instruction. But then, how do you store the address of the routine in the interrupt vector table?

### **SEGMENT ATTRIBUTE AT**

Recall from Unit 7 that the segment combine-type attribute AT <expression> is used to specify the base address of a segment. However, the attribute cannot be used to force the loading of data within that segment. Rather it is used to define, or identify, labels or variable names at fixed offsets within the segment area. You can use the attribute AT to fix the base address of a data segment at address 0000H, and thus place the interrupt vector table at offset zero. Then it's a simple matter to locate any particular offset (vector location) within the table.

Figure 8-3 is an example of how you might initialize a vector address in the interrupt table. The data segment is called VECTOR. Its base address is fixed by the attribute AT 0H in the segment directive. An ORG directive is used to locate the vector "type" address. Recall that each vector occupies four memory locations. Therefore, the first memory location can be identified by multiplying the type number by four. In this example, the vector type number is 96. Identifying the address with the argument  $96*4$  is much more meaningful than the argument 384, or 180H. Once the offset to the vector is established, the LABEL directive is used to assign a name and type. Since word-sized address values will be stored in the vector, the type is WORD. Notice that all we've done is locate and identify the vector. We did not initialize the memory locations. You must use program code to perform that function.

```

TITLE UNIT 8 -- PROGRAM 1 -- INITIALIZING AN INTERRUPT VECTOR ADDRESS
;
PROG_STACK SEGMENT STACK
    DW      80H DUP (?)      ;Set up stack area
TOP_OF_STACK LABEL WORD      ;Identify top of stack for SP register
PROG_STACK ENDS
;
VECTOR SEGMENT AT 0H
    ORG     96*4              ;Point to vector location 96
INT_96 LABEL WORD            ;Identify vector 96
VECTOR ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,DS:VECTOR,SS:PROG_STACK
START: MOV     AX,PROG_STACK    ;Never load a segment register direct
        MOV     SS,AX          ;Use an intermediate register
        MOV     SP,OFFSET TOP_OF_STACK ;Point to the top of stack
                                        ;immediately after loading SS register
        MOV     AX,VECTOR      ;Again, indirectly load the
        MOV     DS,AX          ;segment register
        MOV     INT_96,OFFSET SUB_INT ;Store IP value in vector table
        MOV     INT_96+2,SEG SUB_INT ;Store CS value in vector table
        INT     96              ;Execute the software interrupt
        INT     3              ;Return to the debugger
;
SUB_INT:STC                    ;Set Carry flag
        IRET                   ;Return from interrupt
;
PROG_CODE ENDS
        END     START

```

**Figure 8-3**

Program using the interrupt vector table.

At the end of the code segment, we wrote a two-byte interrupt routine called SUB\_INT (subroutine interrupt). The IP and CS register address values for this routine will be stored in vector 96 of the interrupt vector table by the program. The sixth instruction in the program,

```
MOV INT_96,OFFSET SUB_INT ;Store IP value in vector table
```

uses the operator OFFSET to store the address offset to the interrupt routine in the first word of vector 96. This value will be loaded into the IP register when the interrupt vector is called. The seventh instruction in the program,

```
MOV INT_96,SEG SUB_INT ;Store CS value in vector table
```

uses a new operator **SEG** (segment) to identify the base address of the segment that contains a specified symbol. In this case, the symbol is the subroutine label `SUB_INT`. Since the subroutine is “assumed” to reside in the current code segment, the instruction moves the base address of the code segment into the second word of vector 96. This value will be loaded into the CS register when the interrupt vector is called. The eighth instruction in the program,

```
INT 96          ;Execute the software interrupt
```

calls interrupt vector 96, the vector you initialized with the two previous instructions.

## Self-Review Questions

1. State a general definition of an interrupt. \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_
2. The actions that the MPU takes in response to an interrupt are called the \_\_\_\_\_ routine or \_\_\_\_\_ routine.
3. Responding to an interrupt is referred to as \_\_\_\_\_ the interrupt.
4. The two basic types of interrupts are the \_\_\_\_\_ interrupt and the \_\_\_\_\_ interrupt.
5. Under the general heading of external interrupts, there are the \_\_\_\_\_ interrupt and the \_\_\_\_\_ interrupt.
6. The \_\_\_\_\_ is a list of starting, or physical, addresses for the various interrupt routines.
7. The first two bytes of each interrupt vector, or pointer, contain the \_\_\_\_\_ value, while the next two bytes contain the \_\_\_\_\_ value.
8. Each interrupt routine must end with the \_\_\_\_\_ instruction.
9. A/An \_\_\_\_\_ interrupt will result after the execution of a DIV instruction if the quotient is too large to fit in the destination register.
10. During \_\_\_\_\_ operation, the MPU is directed to a special subroutine after the execution of each instruction.

11. State a use for the single step operation. \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_
12. In order to use the single step operation, you must set the \_\_\_\_\_ flag.
13. The two software interrupt instructions that use an instruction mnemonic are \_\_\_\_\_ and \_\_\_\_\_.
14. Describe what happens when the instruction INT 96 is executed.  
\_\_\_\_\_  
\_\_\_\_\_
15. The interrupt instruction \_\_\_\_\_ will only be executed if an overflow condition exists.
16. Segment attribute \_\_\_\_\_ is used to define the base address of a segment.
17. The assembler operator \_\_\_\_\_ is used to identify the segment base address of a symbol.
18. The segment combine-type attribute AT can be used to force the loading of data within a segment. \_\_\_\_\_  
True/False

NOTE: The Self-Review Answers are located at the end of this unit, beginning on Page 8-75.



## External Interrupts

The 8088 MPU has two external control lines that peripheral (I/O) devices can use to indicate an interrupt. These are the INTR (Interrupt Request) line and the NMI (Non-Maskable Interrupt) line. An input on either of these lines generates its own type of interrupt and has its own specific use. A high on the INTR line generates an interrupt request, while a low-to-high transition on the NMI line generates a non-maskable interrupt. Naturally, there is more to the process than just pulling the lines high. Let's begin with a look at the interrupt request.

### INTERRUPT REQUEST

When the INTR line is high, a peripheral is making an interrupt request. In effect, the **interrupt request** is a request for the MPU to temporarily stop what it is doing and take care of some business for the peripheral.

If an interrupt request is received by the MPU while the MPU is executing a program, the MPU completes the current instruction before it services the interrupt. Now the interrupt request line is not latched inside the MPU. That means the requesting device must hold the line high until the MPU acknowledges the request. If the line goes low before the MPU finishes executing the current instruction, the request will be ignored.

The MPU indicates that it is ready to service the interrupt by pulsing its INTA (Interrupt Acknowledge) line low. This line, from the MPU to the peripheral device, is used for the ready reply from the MPU. The procedure by which the MPU and the peripheral talk back and forth to each other through the use of control lines is called **handshaking**.

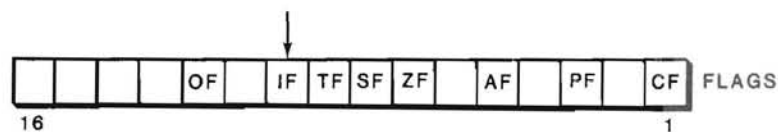
All interrupt requests that arrive at the MPU via the INTR line are vectored interrupts, just like the internal interrupts. Thus, when the MPU responds to an interrupt request, the peripheral must send the MPU the interrupt vector type number. This transfer is handled by the I/O lines that connect the peripheral with the MPU. As you might imagine, this type of interface is complex, and not easily handled by a simple peripheral device. To reduce the complexity, most microcomputers that use the 8088 MPU also use a device called the 8259A Interrupt Controller. This is a programmable device that can communicate with a number of peripheral devices and coordinate their interrupt

requests. It is programmed with the appropriate vector type numbers and handshaking during system "boot."

In the Zenith and IBM systems, all of the peripheral devices that would normally use the INTR line are interfaced to the MPU through the Interrupt Controller. When a particular device such as the disk drive requests service, the Interrupt Controller is toggled by the peripheral INTR line. The controller recognizes the peripheral and sends the appropriate vector number to the MPU. The MPU, in turn, finishes the instruction it is currently executing and then calls the specified interrupt service routine. Once the routine is complete, the MPU returns to what it was doing prior to the interrupt request. We will not go into detail on how the Interrupt Controller operates or is programmed. This is a function that is specific to the design of a microcomputer, not an assembly language programming course.

While you basically have no control over the handling of external interrupts, you can control whether the MPU will recognize an external interrupt request. This is very useful when the MPU is executing a crucial operation or even servicing another more important interrupt. The MPU can ignore an external interrupt request, but to understand how it is done, we must go back to the Flag register again.

Figure 8-4 shows the 8088 MPU Flag register with another new flag. This is the **IF**, or **Interrupt flag**. It is located in bit 9 of the register. If the Interrupt flag is set, the interrupts from the various peripherals are handled as described in the previous paragraphs. However, if the flag is clear, the MPU will ignore all interrupt requests that arrive on the INTR line.



**Figure 8-4**  
The interrupt flag.

Therefore, if you have a routine or program that you do not want interrupted, you merely clear this flag and you disable the incoming interrupt requests. When the MPU is first turned on, this flag is clear. This gives the MPU time to execute its startup routine before it has to process interrupts.

When the MPU services an interrupt request, it clears the Interrupt flag as soon as it is done storing the Flag, CS, and IP registers. This ensures that no other interrupt requests will interfere with the processing of the current request. If there is a need to respond to the other requests, make sure the service routine sets the Interrupt flag.

There are two instructions that let you control the condition of the Interrupt flag. **CLI** (Clear Interrupt flag) is a single-byte instruction that is used to clear the Interrupt flag. To set the Interrupt flag, use the **STI** (Set Interrupt flag) instruction. Both are processor control instructions and do not need any operands.

### NON-MASKABLE INTERRUPT

While you have the capability to enable or mask the INTR line, you have no control over the NMI, or non-maskable interrupt, line. Any low-to-high transition on this line is latched into the MPU. Then, as soon as the MPU completes its current instruction, the interrupt is processed. As we stated earlier, this type of interrupt should only be used in situations where the system is about to fail. For instance, an interrupt from a sensor on the MPU's power supply might indicate that power failure for the system is about to occur. Since this would have grave consequences for the system, the input from this sensor should be connected to the non-maskable interrupt line.

Because the non-maskable interrupt line is dedicated to one function, it is given a dedicated vector type number. The non-maskable vector number is type 2. The address values loaded into this vector are determined by the microcomputer manufacturer.

As you have probably gathered, the MPU is very systematic in its operation. Therefore, you would think that the MPU has some priorities for handling both internal and external interrupts. Well, it does. They are shown in Figure 8-5.

INTERRUPT	PRIORITY
Divide error, INT, INTO NMI INTR Single-step	highest    lowest

**Figure 8-5**  
8088/8086 MPU interrupt priorities.

Notice that the three internal interrupts (divide error, INT, and INTO) have the highest priority. These are followed by the non-maskable interrupt, the interrupt request, and finally, the single-step interrupt. If multiple interrupts arrive at the MPU at the same time, the MPU processes them in this order.

### THE INTERRUPT ROUTINE

As you know, the interrupt routine itself saves the Flag, CS, and IP register values in the stack. But it is up to you to save the information in any register that you may use during the interrupt routine. If you are holding a value in the AX register and then wish to use the AX register in the interrupt routine, be sure to store the current value before you use the register. By the same token, once you have saved a value, be sure to return it to the appropriate register before you terminate the interrupt routine.

In the same vein, many of the system and DOS interrupts and function calls use one or more registers to hold or transfer data. Follow the directions given in the owner's manual for each operation. When in doubt, save all of the important registers before you execute an interrupt instruction.

## Reset

Program execution can be interrupted in another way — RESET. Although not strictly defined as an interrupt, reset does indeed interrupt the operation of the MPU. You use reset any time it is necessary to restart the MPU. This may happen when an error in a program gets you into a loop from which there is no exit, or a power fluctuation can cause the MPU to lose data part way through a program. But no matter what the situation, the reset routine always accomplishes the same thing, it restarts the MPU.

You should, however, keep one point in mind. The reset we are describing is a **hardware** reset, where you actually toggle the RESET control on the MPU. The MPU then performs the reset operation. But while most manufacturers provide a hardware reset from the keyboard, the IBM PC and many of its clones only provide a software reset. This performs essentially the same operation as a hardware reset. The one important exception is when the MPU “hangs-up” and won't respond to any inputs. The only way to regain control in this case is to toggle

the RESET line. With IBM, that means switching the power off and then switching it back on again. Naturally, any data stored in memory is lost when power is lost. Now let's look at the reset process.

When power is first applied to the 8088 MPU, it goes into a reset condition where it performs a routine that initializes the microcomputer system. The initialization process loads the MPU registers with the values it needs to communicate with the rest of the system. Then it calls a subroutine in ROM that programs all of the internal functions necessary for the microcomputer to communicate with the "outside world." We call this a "boot," or "bootstrap," routine. This same initialization process is performed when you generate a reset. While a reset is in progress, the MPU will not recognize an interrupt request or a non-maskable interrupt. The reset routine has priority over all other routines because the MPU cannot function properly until it has been initialized, or in this case, reinitialized.

Upon initiation of a reset, the CS register is set to 0FFFFH and the IP is loaded with 0000H. This points to the boot routine in ROM. The boot routine for both the 8088 and 8086 MPUs is always located at physical address 0FFFF0H. To make sure the first instruction in the boot routine is the first instruction executed by the MPU, the instruction queue is emptied. In addition to setting the CS register and clearing the IP register, the MPU reset function clears the DS, ES, SS, and Flag registers. Figure 8-6 shows the state of all registers within the MPU that are initialized during a reset. All other registers contain random values.

CPU COMPONENT	CONTENT
FLAGS	Clear
Instruction Pointer	0000H
CS Register	FFFFH
DS Register	0000H
SS Register	0000H
ES Register	0000H
Queue	Empty

**Figure 8-6**  
MPU registers following a hardware reset.

## DMA (Direct Memory Access)

In all of the interrupts discussed thus far, the MPU has acted as an intermediary between a peripheral device and memory. The MPU supervises the transfer of data to and from peripherals through the use of the I/O instructions. Data comes from the I/O interface to the MPU and then into memory. Information in memory is transferred first to the MPU and then to the I/O interface.

Some peripheral devices, however, are capable of interfacing directly with memory. These devices can perform their own data transfers without the aid of the MPU. The process by which these devices communicate directly with memory is called **direct memory access** (DMA).

In order for a device to communicate directly with memory, it must have the “permission” of the MPU. To obtain that permission, the peripheral must initiate a **hold request**. This is done by driving the HOLD control line on the MPU high. When the MPU recognizes a hold request, it finishes its current instruction. Then it acknowledges the hold request by outputting a high on the hold acknowledge (HLDA) control line.

Once the MPU acknowledges the hold request, the peripheral gains direct access to the memory through the address and data bus. The MPU releases control of the bus to the peripheral device until the direct memory access is completed. When DMA is finished, the peripheral device pulls the HOLD line low. The MPU responds by pulling the HLDA line low. At this time, the MPU continues program execution at the point of interruption.

You can think of a DMA operation as being similar to an interrupt request. Only in this case, the MPU temporarily suspends operation instead of calling an interrupt routine. The MPU doesn't have to store the flags, IP, or any other register in the stack, because it is not involved in the DMA process. It simply grants memory access to the requesting peripheral.

## Self-Review Questions

19. The two control lines on the 8088 MPU that can be used to indicate an external interrupt are the \_\_\_\_\_ line and the \_\_\_\_\_ line.
20. Describe what occurs when a peripheral device requests an interrupt on the INTR line, assuming the peripheral doesn't require an interrupt controller. \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_
21. The back-and-forth conversation between the MPU and the peripheral over the INTR and INTA lines is called \_\_\_\_\_.
22. Vectored interrupt routines must conclude with the \_\_\_\_\_ instruction.
23. If you want the MPU to "ignore" interrupt requests on the INTR line, then the IF flag must be clear. \_\_\_\_\_  
True/False
24. The \_\_\_\_\_ instruction is used to set the Interrupt flag.
25. The Interrupt flag is cleared by the \_\_\_\_\_ instruction.
26. Even if the Interrupt flag is clear, the \_\_\_\_\_ interrupt will not be disabled.
27. In what order are internal and external interrupts serviced?
  - A. \_\_\_\_\_
  - B. \_\_\_\_\_
  - C. \_\_\_\_\_
  - D. \_\_\_\_\_
28. A \_\_\_\_\_ is used any time it is necessary to restart or initialize the MPU.

29. What are the contents of the following after a reset has occurred?
- A. Flag Register \_\_\_\_\_
  - B. IP Register \_\_\_\_\_
  - C. CS Register \_\_\_\_\_
  - D. DS Register \_\_\_\_\_
  - E. ES Register \_\_\_\_\_
  - F. SS Register \_\_\_\_\_
  - G. Queue \_\_\_\_\_
30. The process by which peripheral devices communicate directly with the microcomputer's memory is called \_\_\_\_\_.
31. In order for a peripheral to obtain "permission" to perform a direct memory access, it must initiate a hold request on the \_\_\_\_\_ control line.
32. The MPU responds to a hold request by outputting a high on the \_\_\_\_\_ control line.



## STRING OPERATIONS

A **string** is a number of bytes or words that reside in sequential memory locations. These bytes or words can represent ASCII characters, numeric values for mathematical calculations, or inputs from peripherals. In fact, they can mean anything that you want them to mean. The important point is that all of the items, or **elements**, in the string are the same size (8- or 16-bit) and that they occupy successive memory locations.

A **string operation** is an operation that is performed on each element of a string. You can use a string operation to successively retrieve each element of a string and compare it to a specific value. By the same token, you can use a string operation to successively move every element, or a number of elements, of a string from one area of memory to another.

Figure 8-7 (Page 8-27) shows a simple program that moves a 100-byte string from the segment `SOURCE_DATA` to the segment `DEST_DATA`. The source string area is pointed to by the DS register, while the destination area for the string is pointed to by the ES register. The first seven instructions set up the MPU segment registers. The next two instructions move the offset address of the source and destination memory locations into the Base and Base Pointer registers respectively. Since the program is using a simple loop to move each element of the string, the loop count is stored in the Count register by the next instruction. That completes register preparation for the string move.

The first instruction in the loop moves the first string element into the AL register using register indirect addressing. The second instruction completes the move by transferring the element to a destination memory location. Two instructions are required for the move because the 8088 MPU cannot perform a direct memory to memory move operation. Notice that we used a segment override prefix in the second move operation. This is necessary because the default segment register for the Base Pointer is the Stack Segment. The next two instructions increment the Base and Base Pointer register contents to point to the next element locations. Finally, the loop instruction decrements and tests the Count register to determine if another loop is required.

At the end of the loop operation, the last instruction sends the MPU back to the debugger. Notice that this instruction is the Break Point interrupt described earlier. The target address stored at location Type 3 in the interrupt vector table is provided by the debugger program. Therefore, in order for the MPU to return to the debugger program, this string move program must be executed from the debugger. If you just load and run the program, vector location Type 3 will contain some random value. Then, when the instruction INT 3 is executed by the program, the microcomputer will “crash.” That is, the MPU will try to execute an instruction that doesn’t exist, with unpredictable results. When we tried this operation on our Zenith microcomputer, the display printed the message “Wild interrupt.” On our IBM PC, the display went blank, the disk drive never switched off, and the keyboard reset function had no effect. We had to switch system power off and then back on to reset and reinitialize the MPU.

You can see from Figure 8-7, that it takes a number of instructions to move a few bytes of data from one memory location to another. The 8088 MPU, however, has a number of instructions that greatly simplify this type of operation. These are the string instructions.

## String Instructions

Loosely speaking, the program in Figure 8-7 could be termed a string operation because it is designed to move each element of a string from one location in memory to another location. String instructions, either alone or in combination with other string instructions, provide a much shorter method of programming a string operation.

```

TITLE UNIT 8 -- PROGRAM 2 -- MOVING DATA
;
PROG_STACK SEGMENT STACK
    DW      80H DUP (?)      ;Set up stack area
TOP_OF_STACK LABEL WORD    ;Identify top of stack for SP register
PROG_STACK ENDS
;
SOURCE_DATA SEGMENT
SOURCE_DB   100 DUP (0ABH)
SOURCE_DATA ENDS
;
DEST_DATA SEGMENT
DEST_DB    100 DUP (?)
DEST_DATA ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,DS:SOURCE_DATA,ES:DEST_DATA,SS:PROG_STACK
START: MOV     AX,PROG_STACK  ;Never load a segment register direct
      MOV     SS,AX          ;Use an intermediate register
      MOV     SP,OFFSET TOP_OF_STACK ;Point to the top of stack
      ;immediately after loading SS register
      MOV     AX,SOURCE_DATA ;Next,indirectly load the
      MOV     DS,AX          ;Data Segment register
      MOV     AX,DEST_DATA   ;Finally, indirectly load the
      MOV     ES,AX          ;Extra Segment register
      LEA     BX,SOURCE      ;Get offset to souce data location
      LEA     BP,DEST        ;Get offset to destination location
      MOV     CX,100         ;Set loop count
LOOP_MOV:
      MOV     AL,[BX]        ;Get byte of source data
      MOV     ES:[BP],AL     ;Save byte of data at destination
      INC     BX             ;Point to next byte of source data
      INC     BP             ;Point to next destination location
      LOOP   LOOP_MOV        ;Check count repeat if necessary
      INT    3              ;Return to the debugger
;
PROG_CODE ENDS
      END     START

```

**Figure 8-7**  
String operation.

Since we've given an example involving a string move, the next step is to look at the same program using the string move instruction. Figure 8-8 shows the program. As you can see, there are a number of things that have changed from the original program. First, the SI (Source Index) and DI (Destination Index) registers are used as pointers instead of the BX and BP registers. This is because **all string instructions use the Source Index register to point to the source of the string**. In addition, **all string instructions (that require a destination) use the Destination Index register to point to the string destination**. In some instances, the string source can be located within either the code, data, or stack segments, but the string destination is always located within the extra segment. As a general rule, it's a good idea to always locate the source string within the data segment. This eliminates any possible confusion.

```

TITLE UNIT 8 -- PROGRAM 3 -- STRING MOVE OPERATION
;
PROG_STACK SEGMENT STACK
    DW      80H DUP (?)      ;Set up stack area
TOP_OF_STACK LABEL WORD    ;Identify top of stack for SP register
PROG_STACK ENDS
;
SOURCE_DATA SEGMENT
SOURCE DB      100 DUP (0ABH)
SOURCE_DATA ENDS
;
DEST_DATA SEGMENT
DEST DB      100 DUP (?)
DEST_DATA ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,DS:SOURCE_DATA,ES:DEST_DATA,SS:PROG_STACK
START: MOV     AX,PROG_STACK    ;Never load a segment register direct
      MOV     SS,AX           ;Use an intermediate register
      MOV     SP,OFFSET TOP_OF_STACK ;Point to the top of stack
                                   ;immediately after loading SS register
      MOV     AX,SOURCE_DATA   ;Next,indirectly load the
      MOV     DS,AX           ;Data Segment register
      MOV     AX,DEST_DATA     ;Finally, indirectly load the
      MOV     ES,AX           ;Extra Segment register
      LEA     SI,SOURCE        ;Get offset to source data location
      LEA     DI,DEST          ;Get offset to destination location
      MOV     CX,100           ;Set loop count
      REP MOVS DEST,SOURCE    ;Get byte of source data
                                   ;and save at destination, then
                                   ;auto-increment SI and DI, then
                                   ;decrement count and test for zero,
                                   ;if not zero, repeat string move
      INT     3               ;Return to the debugger
;
PROG_CODE ENDS
      END     START

```

**Figure 8-8**  
Using a string move instruction.

As with the original program, it is still necessary to initialize the pointer and count registers; only now, it's SI, DI, and CX rather than BX, BP, and CX. The big change is in the string move. Rather than a five-instruction loop, the complete operation is handled by a single instruction, **MOVS** (Move String), and the instruction prefix **REP** (Repeat).

The REP string prefix code tells the MPU that the following string instruction should be repeated "count" times. The count is stored in the CX register. After each string operation, the register is decremented and tested for zero. The process will repeat until CX is zero, at which time the loop is broken and the next instruction in the program is executed. You must place a single character-space between the REP prefix and the string instruction mnemonic.

The string instruction MOVS is a single-operation instruction that takes a byte or word of data from the address pointed to by the SI register and moves it to the address pointed to by the DI register. In effect, this is a memory-to-memory move instruction. String instructions are the only instructions in the 8088 MPU instruction set that allow that type of data transfer. After the move is complete, the SI and DI registers are incremented or decremented as determined by a bit in the Flag register. We'll describe that flag bit later in this section.

Data type (byte or word) is determined by the assembler when it examines the source and destination operands of the instruction. A word-to-byte or byte-to-word move is not allowed. The assembler also determines whether or not the source and destination operand symbols can be "reached" from your code segment. These are what you would call "symbolic references." You still must load the source and destination registers with the offset addresses of the respective string locations. The string instruction will not do that for you.

A second variation of the move string instruction uses what we call "anonymous" references, which take the form of a register indirect move instruction. A typical instruction is:

```
REP MOVS ES:BYTE PTR [DI],[SI]
```

where the segment override prefix code tells the MPU that the data pointed to by the DI register is in the extra segment. The assembler operator BYTE PTR tells the assembler that it must generate the appropriate code for a byte move operation. Placing both the DI and SI register symbols within square brackets tells the assembler that both registers

are used for indirect addressing. If the SI register is located in a segment other than the data segment, it also needs a segment override prefix code, such as:

```
REP MOVSB ES:BYTE PTR [DI],SS:[SI]
```

where the source string is located within the stack segment.

Both of these anonymous instruction formats will only work with a string operation. They will not work in a normal move instruction.

The assembler will accept either a symbolically referenced string instruction or an anonymously referenced string instruction. However, the symbolically referenced instruction is the preferable method. This allows the assembler to check the addressability of the operands, and verify the compatibility of the operand types.

As you can see, the anonymous string instruction would be rather complex. The 8088 instruction set has resolved that problem with two other move string instructions. They are **MOVSB** (Move String Byte) and **MOVSW** (Move String Word). The operand type is specified in the mnemonic of the instruction. Thus, the earlier move byte-sized string instruction can be written:

```
REP MOVSB
```

With this instruction, the operands are implied by the mnemonic. However, this implication forces you to place the source string in the data segment and the destination area in the extra segment. Naturally, both string elements should be byte-sized. Since the assembler has no way to verify element sizes, it's up to you.

Earlier we said that after each string move operation the SI and DI registers are automatically incremented or decremented to point to the next string location. Actually, after each byte-sized move, the SI and DI registers are incremented or decremented **once**. After each word-sized move, the SI and DI registers are incremented or decremented **twice**.

Now that you understand the basic concepts of string instructions, and specifically the move string instruction, let's review the other string instructions.

The **CMPS** (Compare String) instruction compares an element from one string with an element from another string. Like the arithmetic compare (**CMP**) instruction, neither string element is physically moved or altered during a compare string operation. Only the six arithmetic flags in the Flag register are affected. These include **OF**, **SF**, **ZF**, **AF**, **PF**, and **CF**. The instruction takes the form:

```
CMPS  SOURCE, DEST
```

where **DEST** and **SOURCE** are symbols for the effective address of the two strings in memory. The effective address for the symbol **SOURCE** is stored in the **SI register**, while the effective address for the symbol **DEST** is stored in the **DI register**. **CAUTION**: The operand arrangement in the **CMPS** instruction is just the opposite from the operand arrangement in the **MOVS** instruction. Don't confuse the structure of the **MOVS** instruction with the structure of the **CMPS** instruction! As before, the operand types and their compatibility are determined by the assembler from the source and destination data definition statements.

Like the move string instruction, the compare string instruction has two "short" anonymous instruction variations. They are **CMPSB** (Compare String Byte) and **CMPSW** (Compare String Word). No operands are used with these instructions.

The **SCAS** (Scan String) instruction is similar to the compare string instruction with one exception. Instead of comparing one string with another, it compares a string with the contents of the accumulator (**AL** register for byte strings, **AX** register for word strings). The instruction causes the six arithmetic flag bits in the Flag register to be updated, but it does not alter the contents of the string or the accumulator. The scan string instruction uses the **DI register** to hold the effective address of the string, not the **SI register** as you might think. Since the **DI register** holds the effective address, the string must (by default) reside in the extra segment. The instruction takes the form:

```
SCAS  DEST
```

where **DEST** is the symbol for the effective address of the string. Notice that the instruction uses a single operand, and that operand identifies the string. The assembler determines the appropriate accumulator register from the string type (byte or word).

As with all string instructions, there are two short anonymous scan string instructions. They are **SCASB** (Scan String Byte) and **SCASW** (Scan String Word). No operands are used with these instructions. NOTE: The symbolically referenced scan string instruction will not assemble properly with the early version of IBM's MACRO-86. If you have that assembler, use the short anonymous version of the scan string instruction.

The next string instruction is **LODS** (Load String). It is used to transfer a string element to either the AL or AX accumulator register, depending on the string element type. Because the instruction is transferring a string element into a register, it is normally not used in conjunction with the REP prefix code. To do so would result in an overwrite of the contents of the register. However, this instruction can be used to reduce the complexity of a large program loop. It simplifies the loop because the string addressing register is automatically incremented or decremented after each execution of the instruction. This eliminates the need for including an increment or decrement instruction in the loop. The load string instruction takes the form:

LODS SOURCE

where SOURCE is the symbol for the effective address of the string element. Again, the string instruction uses a single operand to identify the string. But in this case, the string is a source operand. For that reason, the string element must be addressed through the **SI register**. The assembler identifies the appropriate accumulator register from the string element type.

The load string instruction also comes in two short anonymous instruction forms. They are **LODSB** (Load String Byte) and **LODSW** (Load String Word). No operands are used with these instructions.

The last string instruction, **STOS** (Store String), transfers the contents of either the AL or AX register to the string element identified in the instruction. The operand type is determined from the instruction operand. This instruction, when used in conjunction with the REP



prefix code, provides a unique method for initializing a series of memory locations with a specific value. The instruction takes the form:

```
STOS DEST
```

where DEST is the symbol for the effective address of the string element. Here again the instruction uses a single operand to identify the string. Note that this time the string is a source operand. For that reason, the string element must be addressed through the **DI register**.

The two short anonymous versions of the store string instruction are: **STOSB** (Store String Byte) and **STOSW** (Store String Word). No operands are used with either instruction.

Figure 8-9 summarizes the five string instructions.

MOVS/ MOVSB/MOVS	Copy byte or word source string element to destination string in memory
CMPS/ CMPSB/CMPS	Compare byte or word destination string element to source string element
SCAS/ SCASB/SCAS	Scan (compare) byte or word destination string element with accumulator
LODS/ LODSB/LODS	Load byte or word source string element into accumulator
STOS/ STOSB/STOS	Store accumulator as element in destination string in memory

**Figure 8-9**  
String instructions.

## Self-Review Questions

33. A \_\_\_\_\_ is a number of bytes or words that reside in sequential memory locations.
34. Each item in a string can also be called an \_\_\_\_\_.
35. An operation that is performed on each element of a string is called a \_\_\_\_\_ operation.
36. In a string operation, the number of times the operation is repeated is specified in the \_\_\_\_\_ register.
37. The default segment for the destination string is the \_\_\_\_\_ segment.
38. The default general, or offset address, register that is used to point to the memory location of the destination string is the \_\_\_\_\_ register.
39. The default general, or offset address, register that is used to point to the memory location of the source string is the \_\_\_\_\_ register.
40. A single move instruction can be used to transfer data from one memory location to another. \_\_\_\_\_  
True/False
41. The string prefix code \_\_\_\_\_ causes a string operation to be repeated count times.
42. The instruction  
`MOVS DEST, SOURCE`  
 is said to use a \_\_\_\_\_ reference in determining the source and destination string locations.
43. An anonymously referenced string instruction defaults to the \_\_\_\_\_ segment for determining the location of the source string.

44. The SI and DI registers are incremented or decremented \_\_\_\_\_ after a word-sized string element is moved.  
once/twice

45. If the source string is identified by the symbol SOURCE and the destination string is identified by the symbol DEST, write the symbolically referenced instruction that will:

- A. Move a byte string: \_\_\_\_\_
- B. Compare an element in two word strings: \_\_\_\_\_
- C. Compare a word value to a string element: \_\_\_\_\_
- D. Store a byte string: \_\_\_\_\_
- E. Load a word-sized string element into the accumulator: \_\_\_\_\_

46. Write the short anonymous instruction that will fulfill each of the following conditions:

- A. Move a word string: \_\_\_\_\_
- B. Compare an element in two byte strings: \_\_\_\_\_
- C. Compare a byte value to a string element: \_\_\_\_\_
- D. Store a word string: \_\_\_\_\_
- E. Load a byte-sized string element into the accumulator: \_\_\_\_\_

47. If the destination string element contains the value 42H and the source string element contains the value 88H, indicate the condition of the following arithmetic flags after the instruction CMPSB is executed:

- A. Overflow flag \_\_\_\_\_
- B. Sign flag \_\_\_\_\_
- C. Zero flag \_\_\_\_\_
- D. Auxiliary Carry flag \_\_\_\_\_
- E. Parity flag \_\_\_\_\_
- F. Carry flag \_\_\_\_\_

## String Direction

Earlier, we said that each string instruction increments or decrements the SI and/or DI registers after each operation. This automatic process is performed so that the next element in the string can be identified. Whether the register increments or decrements is determined by a bit in the Flag register called the **Direction Flag**. Figure 8-10 shows the Flag register with the new Direction flag bit identified by the arrow. The flag is located in bit 11 of the Flag register.



**Figure 8-10**  
The complete Flag register.

The Direction flag controls the direction that a string instruction will take when it is implemented. In other words, the Direction flag determines whether the instruction will increment or decrement through a string when it is executed. A zero in the Direction flag causes the string instruction to automatically increment, while a one in the Direction flag causes the string instruction to automatically decrement during each execution.

The status of the Direction flag is controlled by two instructions. To set the flag, use the instruction **STD** (Set Direction). To clear the flag, use the instruction **CLD** (Clear Direction). When the MPU is reset, all of the Flag register bits are cleared. As a result, any string operation performed immediately after a reset will automatically increment.

## Repeat String Variations

All string instructions perform a single move or compare operation. Since most string operations involve more than one string element, the REP string prefix code is used quite often. To make your job of programming a little easier, the 8088 instruction set includes two other forms of the repeat string prefix: **REPZ** (Repeat while Zero) and **REPNZ** (Repeat while Not Zero).

Both prefix codes differ from the REP prefix in that they are **conditional** prefixes. These prefix codes not only decrement and test the count in the CX register, they also test the Zero flag bit in the Flag register. REPZ prefix will cause the string instruction to continue to loop until the contents of the CX register are zero, **or** until the Zero flag is **set**. On the other hand, the REPNZ prefix will cause the string instruction to continue to loop until the contents of the CX register are zero, **or** until the Zero flag is **cleared**. These prefix codes should only be used with the compare string and scan string instructions, since these are the only string instructions that affect the Zero flag.

The REPZ and REPNZ prefix codes have an alternate configuration. They are **REPE** (Repeat while Equal) and **REPNE** (Repeat while Not Equal). Both test the condition of the Zero flag. The REPE prefix operates just like the REPZ prefix, while the REPNE prefix operates just like the REPNZ prefix. The reason for duplicate prefix codes follows the reasoning given earlier in the course for the conditional loop instructions and conditional jump instructions. They offer the programmer a different perspective on the same operation. If you are looking for a zero condition, you would probably use the REPZ prefix code. On the other hand, if you are looking for an equal condition, you would use the REPE prefix code. In both instances, you are checking the condition of the Zero flag.

Figure 8-11 summarizes the three repeat string prefix codes.

REP	Repeat string operation
REPZ/REPE	Repeat string operation while zero/equal
REPNZ/REPNE	Repeat string operation while not zero/not equal

**Figure 8-11**  
Repeat string prefix codes.

## Register Loading

As your programs increase in complexity, you will begin using many different data segments to hold important arrays and strings. Loading new segment and general register values to point to those data segments can use many bytes of program code. To reduce the amount of code needed to load these registers, the 8088 MPU instruction set has two unique instructions: **LDS** (Load Data Segment) and **LES** (Load Extra Segment).

The LDS instruction is similar to the LEA instruction. Only along with loading the address offset of a symbol into a 16-bit general register, it also loads the segment base address of that symbol into the Data segment register. While any 16-bit general register can be used in the operation, you will normally use a base or index register, since you are setting-up the register for an indirect addressing operation. In the case of a string move, you would naturally use the SI register as the general register. The instruction takes the form:

**LDS <general register>,<address location>**

where LDS is the instruction mnemonic; <general register> is the 16-bit general register that will receive the address offset value; and <address location> is the offset address to a four-byte address value that is stored in the current data segment. That four-byte address value is the address of the string or data array that is located in the current data segment, or some other data segment. Thus, before the LDS instruction can be used, the base and offset address it is to load into the general and segment registers must be stored in memory. The best way to handle this memory storage is through the Define Doubleword (DD) assembler directive.

The DD assembler directive, when used in this manner, operates a little differently than described in Unit 5. Rather than initialize or reserve a four-byte memory location, it identifies and stores the base and offset address of a symbol. The LDS instruction then uses the name associated with the DD directive to load the address values. To get a better idea of how the process operates, look at Figure 8-12.

```

TITLE UNIT 8 -- PROGRAM 4 -- USING LOAD SEGMENT REGISTER INSTRUCTIONS
;
PROG_STACK SEGMENT STACK
    DW      80H DUP (?)      ;Set up stack area
TOP_OF_STACK LABEL WORD      ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_DATA SEGMENT
DATA1 DD SOURCE
DATA2 DD DEST
PROG_DATA ENDS
;
SOURCE_DATA SEGMENT
SOURCE DB 100 DUP (0ABH)
SOURCE_DATA ENDS
;
DEST_DATA SEGMENT
DEST DB 100 DUP (?)
DEST_DATA ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,DS:PROG_DATA,SS:PROG_STACK
START: MOV AX,PROG_STACK ;Never load a segment register direct
      MOV SS,AX          ;Use an intermediate register
      MOV SP,OFFSET TOP_OF_STACK ;Point to the top of stack
      ;immediately after loading SS register
      MOV AX,PROG_DATA   ;Next,indirectly load the
      MOV DS,AX          ;Data Segment register
      LES DI,DATA2       ;Get extra segment and offset
      ;address to destination location
      LDS SI,DATA1       ;Get data segment and offset
      ;address to source data location
      ASSUME DS:SOURCE_DATA,ES:DEST_DATA
      MOV CX,100         ;Set loop count
      REP MOVSB DEST,SOURCE ;Get byte of source data
      ;and save at destination, then
      ;auto-increment SI and DI, then
      ;decrement count and test for zero,
      ;if not zero, repeat string move
      INT 3             ;Return to the debugger
;
PROG_CODE ENDS
      END START

```

**Figure 8-12**

String program using new segment loading instructions.

The figure is a repeat of the string program we described earlier, with a few changes. First, a new data segment called `PROG_DATA` contains two DD assembler directives. The first one causes the assembler to determine the offset address and segment base address of the string `SOURCE`. The two address values are then stored at the effective address identified by the name `DATA1`. The offset address and segment base address of the uninitialized string `DEST` is stored at `DATA2`.

The next program change is in the `ASSUME` directive. The “assumed” data segment is now `PROG_DATA`. There is no assumed extra segment. Naturally, the two instructions that load the initial value into the Extra Segment have been deleted. The instructions that load the Data Segment register now reference the `PROG_DATA` segment.

Up to this point, the program has identified the physical address of the two strings in memory and set up the MPU registers. The next two instructions prepare the MPU for the string transfer. The `LES` instruction operates just like the `LDS` instruction, except that it loads the Extra Segment register rather than the Data segment register. Thus, when the instruction

```
LES  DI,DATA2      ;Get extra segment and offset
                   ;address to destination location
```

is executed, the first word stored at `DATA2` is moved into the `DI` register. Then the second word stored at `DATA2` is moved into the `ES` register. When the instruction

```
LDS  SI,DATA1      ;Get data segment and offset
                   ;address to source data location
```

is executed, the first word stored at `DATA1` is moved into the `SI` register. Then the second word stored at `DATA1` is moved into the `DS` register. The MPU registers are now ready for the string transfer.

You may have noticed that the order of loading the `SI` and `DI` registers is reversed from the earlier program. This is a subtle, but very important point. Had we loaded the `SI` register and hence the `DS` register first, the `LES` instruction would have then loaded the **wrong** data into the `DI` and `ES` registers. Remember, `DATA2` is located in what the assembler assumes is the **current** data segment. If you allow an instruction to modify the contents of the `DS` register without informing the assembler, the assembler will use the wrong address values when it assembles the following instruction.



With the instructions to load the MPU registers with the correct string address values arranged in the proper order, there is one more operation to perform. The assembler must be told that the DS and ES registers now point to new memory segments. This is accomplished with a new ASSUME directive. Thus, when the assembler assembles the remaining program code, it will use the correct address values in the move string instruction. Remember, always place an ASSUME directive immediately after an instruction, or group of instructions, that change a segment register value.

Both of the load segment register instructions in the program in Figure 8-12 used a symbolic reference to identify the location of the base and offset address values. You could have also used an anonymous reference such as:

```
LES  DI,[BX]      ;Get extra segment and offset
                  ;address to destination location
```

to identify the location of the base and offset address values stored in memory. Naturally, you first would have to load the address offset, to the memory location that contained those address values, into the BX register.

## Self-Review Questions

48. The contents of the \_\_\_\_\_ determine the direction of a string operation.
49. If the Direction flag is set during a string operation, the SI and/or DI registers will \_\_\_\_\_.  
increment/decrement
50. The \_\_\_\_\_ instruction is used to clear the Direction flag.
51. The \_\_\_\_\_ instruction is used to set the Direction flag.
52. The \_\_\_\_\_ and \_\_\_\_\_ string prefix codes decrement the CX register, test the register for zero, and check to see if the Zero flag is set.
53. The \_\_\_\_\_ and \_\_\_\_\_ string prefix codes decrement the CX register, test the register for zero, and check to see if the Zero flag is clear.
54. The \_\_\_\_\_ instruction is used to load the data segment register and a 16-bit general register.
55. The \_\_\_\_\_ instruction is used to load the extra segment register and a 16-bit general register.
56. When dealing with string operations, the 16-bit general register that is loaded by the LES instruction should be the \_\_\_\_\_ register.
57. The best way to load the base and offset address of a string in memory is with the \_\_\_\_\_ assembler directive.
58. When the LDS and LES instructions are executed one after the other, the \_\_\_\_\_ instruction should be executed first.
59. You can only use symbolic references with the LDS and LES instructions. \_\_\_\_\_  
True/False

## EXPERIMENT

### Software Interrupts and String Operations

- OBJECTIVES:*
1. *Demonstrate the conditional interrupt INTO.*
  2. *Demonstrate a user-defined interrupt.*
  3. *Demonstrate all of the string instructions.*
  4. *Demonstrate the LDS and LES instructions.*

### Introduction

Your introduction to programming in MACRO-86 is nearly complete. This experiment will demonstrate the MACRO-86 instruction set covering the areas of software interrupts and data strings. When you complete this experiment, you will possess the skills needed to write an assembly language program using MACRO-86. The last unit in the course will clean up any loose ends in the areas of program structure, assembly control, conditional assembly, and code macros — unique instructions that you create from the standard MACRO-86 instruction set.

### Procedure

1. Call up the editor and enter the program listed in Figure 8-13, parts A, B, C, D, and E. Part A of the listing contains an optional call instruction. If you have a Zenith microcomputer, load the program as listed. If you have an IBM PC, or one of its clones, place a semicolon in front of the CALL CLEAR\_ZENITH instruction and remove the semicolon from in front of the CALL CLEAR\_IBM instruction. This is identical to the clear screen operation you performed in an earlier experiment. The rest of the program will function as written with either system.

```

TITLE EXPERIMENT 8 -- PROGRAM 1 -- SOFTWARE INTERRUPTS
;
PROG_STACK SEGMENT STACK
    DW    100H DUP (?)    ;Set up stack area
TOP_OF_STACK LABEL WORD    ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_DATA SEGMENT
MULT1_MSG DB 'ENTER 1 TO 4 DIGIT HEX MULTIPLICAND $' ;Data message
MULT1     DB 5          ;Multiplicand storage area size
          DB 6 DUP (0)  ;buffer initialized to zero
MULT2_MSG DB 'ENTER 1 TO 4 DIGIT HEX MULTIPLIER $' ;Data message
MULT2     DB 5          ;Multiplier storage area size
          DB 6 DUP (0)  ;buffer initialized to zero
CLEAR_SCREEN DB 1BH,'E',1BH,'H','$' ;Code to clear screen and home
          ;cursor on Zenith system
PRINT_UNDER DB 'NO OVERFLOW' ;No overflow message
          DB 0DH,0AH ;Carriage return, line feed
          DB '16-BIT PRODUCT = $';Product size message
PRINT_OVER DB 'OVERFLOW' ;Overflow message
          DB 0DH,0AH ;CR, LF
          DB '32-BIT PRODUCT = $';Product size message
DO_AGAIN DB 'MULTIPLY TWO MORE NUMBERS? (Y or N) $' ;Repeat message
PROD_32_BIT DB 4 DUP (0) ;Reserve for high word of product
PROD_16_BIT DB 4 DUP (0) ;Reserve for low word of product
CR_LF_CODE DB 0DH,0AH,'$' ;CR, LF, and end message character
PROG_DATA ENDS
;
VECTOR SEGMENT AT 0H
    ORG    4*4          ;Point to vector location 4
INT_4 LABEL WORD ;Identify vector 4
    ORG    96*4         ;Point to vector location 96
INT_96 LABEL WORD ;Identify vector 96
VECTOR ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,DS:VECTOR,SS:PROG_STACK
START: MOV AX,PROG_STACK ;Never load a segment register direct
      MOV SS,AX ;Use an intermediate register
      MOV SP,OFFSET TOP_OF_STACK ;Point to the top of stack
          ;immediately after loading SS register
      PUSH DS ;Save segment for return value
      SUB AX,AX ;Zero the AX register
      PUSH AX ;Save offset of zero for far return
      MOV AX,VECTOR ;Again, indirectly load the
      MOV DS,AX ;segment register
      MOV INT_4,OFFSET INTO_INT ;Store INTO IP in vector table
      MOV INT_4+2,SEG INTO_INT ;Store INTO CS in vector table
      MOV INT_96,OFFSET SUB_INT ;Store user IP in vector table
      MOV INT_96+2,SEG SUB_INT ;Store user CS in vector table
    ASSUME DS:PROG_DATA
      MOV AX,PROG_DATA ;Indirectly load the
      MOV DS,AX ;new data segment base address
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      CALL CLEAR_ZENITH ;Zenith clear screen routine
;
      CALL CLEAR_IBM ;IBM clear screen routine
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
INPUT_NUMBER:
      CALL GET_NUMBER ;Execute number input routine
      MUL BX ;Multiply the values

```

Figure 8-13A

Program to illustrate software interrupts.

```

        INTO                ;Interrupt if product overflows AX reg.
        JO      OUT_PROD    ;Print the product
        INT      96        ;Software interrupt to display product
OUT_PROD:
        CALL     PRINT_PRODUCT ;Display the product routine
        CALL     QUIT        ;End program response routine
        CMP     AL,'Y'      ;Another number?
        JE      INPUT_NUMBER ;No, get another number; Yes, quit
;
EXIT   PROC   FAR          ;Set up for far return and
      RET    RET          ;exit program gracefully
EXIT   ENDP                ;through built-in system interrupt
;
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
      CALL     DISPLAY      ;Output code to display
      RET
;
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 VIDEO_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
;
GET_NUMBER:                ;Subroutine to input two hexadecimal
                          ;numbers up to four digits long
      CALL     CR_LF      ;Carriage return, line feed routine
      LEA     DX,MULT1_MSG ;Load address of message string
      CALL     DISPLAY    ;Output code to display
      LEA     DX,MULT1    ;Load address of memory buffer area
      MOV     AH,0AH      ;Buffered keyboard input interrupt code
      INT     21H        ;Call the interrupt
      CALL     CR_LF      ;Carriage return, line feed routine
      LEA     DX,MULT2_MSG ;Load address of message string
      CALL     DISPLAY    ;Output code to display
      LEA     DX,MULT2    ;Load address of memory buffer area
      MOV     AH,0AH      ;Buffered keyboard input interrupt code
      INT     21H        ;Call the interrupt
      CALL     CR_LF      ;Carriage return, line feed routine
      LEA     SI,MULT1+2  ;Load address of first (high) byte of
                          ;the muptiplicand
      MOV     DH,MULT1+1  ;Get number byte count from buffer
      SUB     AX,AX       ;Zero the AX register
FIRST:  CALL     ASCII_TO_HEX ;Convert ASCII to hexadecimal value
      ADD     AL,DL       ;Transfer value to AX register
      DEC     DH          ;Maintain count of bytes left in buffer

```

Figure 8-13B

Continuation of the program to illustrate software interrupts.

```

                                CMP     DH,0           ;Last byte?
                                JZ      NEXT          ;Yes, go to NEXT, otherwise continue
                                MOV     CL,4         ;Move shift count into register
                                SHL     AX,CL        ;Shift the AX register left four bits
                                                ;so next hexadecimal value can be added
                                JMP     FIRST        ;Repeat process
NEXT:  LEA     SI,MULT2+2          ;Load address of first (high) byte of
                                ;the multiplier
                                MOV     DH,MULT2+1  ;Get number byte count from buffer
                                SUB     BX,BX        ;Zero the BX register
SECOND: CALL    ASCII_TO_HEX      ;Convert ASCII to hexadecimal value
                                ADD     BL,DL        ;Transfer value to BX register
                                DEC     DH          ;Maintain count of bytes left in buffer
                                CMP     DH,0         ;Last byte?
                                JZ      DONE          ;Yes, go to DONE, otherwise continue
                                MOV     CL,4         ;Move shift count into register
                                SHL     BX,CL        ;Shift the BX register left four bits
                                                ;so next hexadecimal value can be added
                                JMP     SECOND       ;Repeat process
DONE:  RET                               ;Return from subroutine call
;
ASCII_TO_HEX:                    ;Subroutine to convert ASCII code
                                ;to a hexadecimal value
                                MOV     DL,[SI]      ;Get first byte of number
                                INC     SI          ;Point to next byte
                                CMP     DL,40H      ;Check for capital letter
                                JA      BIG         ;If capital letter, jump
                                AND     DL,0FH      ;Otherwise mask high four bits of byte
                                                ;to get absolute hexadecimal value
                                JMP     HEX_NUM      ;Bypass next instruction
BIG:   SUB     DL,37H              ;Change ASCII code for capital letter
                                                ;to absolute hexadecimal value
HEX_NUM:RET                      ;Return from subroutine call
;
INTO_INT:                        ;Subroutine to print the message that
                                ;a multiplication overflow occurred
                                PUSH    AX          ;Save the product low word
                                PUSH    DX          ;Save the product high word
                                LEA     DX,PRINT_OVER ;Address of overflow message
                                CALL    DISPLAY     ;Output code to display
                                POP     DX          ;Retrieve the product high word
                                POP     AX          ;Retrieve the product low word
                                IRET              ;Return from interrupt
;
SUB_INT:                          ;Subroutine to print the message that
                                ;a multiplication overflow didn't occur
                                PUSH    AX          ;Save the product low word
                                LEA     DX,PRINT_UNDER ;Address of no overflow message
                                CALL    DISPLAY     ;Output code to display
                                POP     AX          ;Retrieve the product low word
                                IRET              ;Return from interrupt
;
PRINT_PRODUCT:                    ;Subroutine to convert and display
                                ;the 16- or 32-bit product
                                MOV     BH,0         ;Clear register to use as a flag
                                JNO     SMALL1      ;Go to 16-bit product storage routine
                                LEA     SI,PROD_32_BIT+3;Point to least significant byte in
                                                ;high word of 32-bit storage area
                                MOV     CH,4         ;Number of bytes to be saved
LOOP1: MOV     BL,DL              ;Get first two hexadecimal digits

```

Figure 8-13C

Continuation of the program to illustrate software interrupts.

```

        CALL    HEX_TO_ASCII    ;Convert and save hexadecimal digit
        JZ     DOWN            ;Yes, jump to DOWN, otherwise continue
        MOV    CL,4            ;Load bit shift count in register
        SHR   DX,CL            ;Shift the hexadecimal digits right
        JMP    LOOP1           ;Repeat process
SMALL1: INC    BH              ;Flag to indicate a 16-bit product
DOWN:  MOV    CH,4            ;Number of bytes to be saved
        LEA   SI,PROD_16_BIT+3;Point to least significant byte in
        ;16-bit storage area
LOOP2: MOV    BL,AL           ;Get first two hexadecimal digits
        CALL  HEX_TO_ASCII    ;Convert and save hexadecimal digit
        JZ    PRINT_VALUE     ;Yes, jump to PRINT_VALUE; No, continue
        MOV    CL,4            ;Load bit shift count in register
        SHR   AX,CL            ;Shift the hexadecimal digits right
        JMP    LOOP2           ;Repeat process
PRINT_VALUE: ;Routine to output product
        CMP    BH,1           ;Is it a 16-bit product?
        JE    SMALL2         ;Yes, jump to SMALL2; No, continue
        LEA   DX,PROD_32_BIT  ;Get offset to 32-bit product
        CALL  DISPLAY         ;Output code to display
        JMP    PRN_END        ;Exit routine
SMALL2: LEA   DX,PROD_16_BIT  ;Get offset to 16-bit product
        CALL  DISPLAY         ;Output code to display
PRN_END:RET                    ;Return from subroutine call
;
HEX_TO_ASCII: ;Subroutine to convert a hexadecimal
;digit to ASCII and save it in memory
        AND    BL,0FH         ;Mask the high digit
        CMP    BL,0AH         ;Is hexadecimal digit a letter?
        JB    ASCII_NUM      ;No, jump to ASCII_NUM
        ADD    BL,37H         ;Yes, convert it to ASCII
        JMP    SAVE           ;Bypass next instruction
ASCII_NUM: ;Change hexadecimal number to ASCII
        ADD    BL,30H
SAVE:  MOV    [SI],BL         ;Save the ASCII value in memory
        DEC    SI             ;Point to next (higher) storage area
        DEC    CH             ;Maintain count of bytes to save
        CMP    CH,0           ;Last byte?
        RET                    ;Return from subroutine call
;
QUIT: ;Subroutine to let you exit or continue
;to multiply hexadecimal numbers
        LEA   DX,DO_AGAIN     ;Get offset to QUIT message
        CALL  DISPLAY         ;Output code to display
QUIT1: MOV    AH,1            ;Load examine keyboard interrupt code
        INT    21H           ;Call the interrupt
        CMP    AL,'N'         ;Was capital N pressed?
        JE    QUIT2          ;Yes, exit, No, continue
        CMP    AL,'Y'         ;Was capital Y pressed?
        JE    QUIT2          ;Yes, exit, No, continue
        JMP    QUIT1         ;Repeat process, check keyboard
QUIT2: RET                    ;Return from subroutine call
;
CR_LF: ;Subroutine to output a carriage return
;and line feed to the display
        LEA   DX,CR_LF_CODE   ;Get offset to display control code
        CALL  DISPLAY         ;Output code to display
        RET                    ;Return from subroutine call
;
DISPLAY: ;Subroutine to output code or a

```

Figure 8-13D

Continuation of the program to illustrate software interrupts.

```

;message to the display. Data must
;end with the '$' ASCII value
MOV     AH,9           ;Load output string interrupt code
INT     21H           ;Call the interrupt to output string
RET                                           ;Return from subroutine call
;
PROG_CODE ENDS
END      START

```

**Figure 8-13E**

Last part of the program to illustrate software interrupts.

The program is designed to show you how a user-defined interrupt and the MPU-defined interrupt on overflow software interrupts can be used. Naturally, we could have provided you with the basics for using these interrupts in a few bytes of code. However, we felt that this was a good opportunity to tie what you have learned into a simple, yet comprehensive, program.

The program clears the display and requests an input of one to four hexadecimal digits. After you enter a value and press the RETURN key, the display requests a second hexadecimal value. After you enter the second value and press the RETURN key, the program multiplies the two numbers and determines if the product is a 16- or 32-bit value. It then displays the product size and the value. Finally, it asks whether or not you wish to multiply two more numbers. If you do, type “Y” and the program will start over. If you don’t, type “N” and the program will end. As we said, we tried to keep the program simple. Therefore, you can only multiply hexadecimal numbers. Also, you can only use capital letters in any of your responses.

2. Assemble and link the program. Now execute the program by typing the program name without the file extension. Try multiplying a few numbers. As long as the product is less than 17 bits, the program will display the message “NO OVERFLOW” followed by the message “16-BIT PRODUCT = ” and the value of the product. If the product exceeds 16 bits, the program will display the message “OVERFLOW” followed by the message “32-BIT PRODUCT = ” and the value of the product.



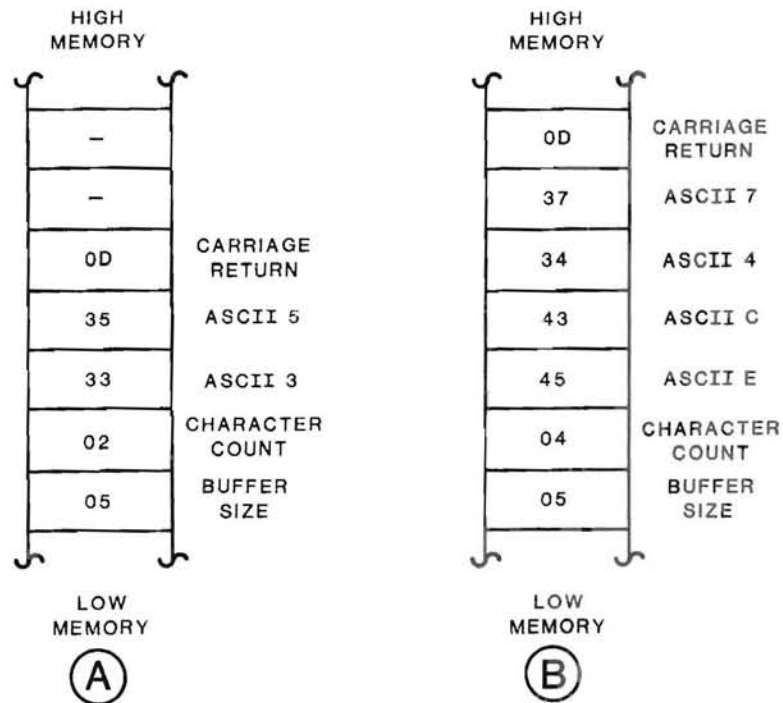
## Discussion

The program is composed of four separate segments. The first is the usual stack segment. We changed the size to 100H words to make sure we had enough storage space for the interrupt routines and the multi-level subroutine calls.

The primary data segment is called `PROG_DATA`. It contains all of the display messages and the data storage areas. While most of it is self-explanatory, there are a few interesting items. First, the program is using a new data input interrupt. Instead of inputting a single character from the keyboard, this interrupt allows you to input any number of characters. The character storage area, or buffer, is composed of three parts. The first part is a byte-sized value that specifies the number of characters that can be stored. You or the program must store a value in this location prior to execution of the interrupt. The second part is a byte-sized location that indicates the number of characters stored after the interrupt has executed. The third part is the actual buffer area that will hold the character string. In addition to the actual storage area, the buffer needs one additional byte to store the carriage return code used to indicate the end of the character string.

In your program, there are two buffers to hold the multiplicand and the multiplier. These are called `MULT1` and `MULT2` respectively. Because each buffer will be storing up to four characters, the first byte in each contains the value five (four characters and the carriage return code). The next buffer define byte statement initializes six bytes of storage space for each buffer. The first byte is used to store the character count, the next four bytes are used to store the characters, and the last byte is used to store the carriage return code.

To see how the buffer works, assume that the interrupt is called and the hexadecimal value 35 is entered through the keyboard. The interrupt is terminated when the RETURN key is pressed. The buffer is shown in Figure 8-14A. If the interrupt is called a second time, and the hexadecimal value EC47 is entered followed by a RETURN, the buffer would contain the data shown in Figure 8-14B.



**Figure 8-14**

Storing character string in a memory buffer.

In Part A, the first value is 05, the buffer size. It must be loaded prior to executing the interrupt. The next value is 02, the number of characters stored in the buffer. The interrupt routine counts the number of characters stored in the buffer and places the value in the second byte location. Notice that the code for a carriage return is not counted as a character. This is followed by the two values stored through the keyboard. They are always stored in their ASCII form. The last value is the ASCII code for a carriage return. The two remaining memory locations are undefined.

In Part B, four values are stored in the buffer; the character count is 04, and the buffer is full. If you try to enter more than the specified number of characters, they will be ignored by the interrupt routine. In this example, the maximum number of characters is four, with the fifth buffer location reserved for the carriage return code.

The product generated by the program is stored at memory location `PROD_32_BIT` or `PROD_16_BIT`. These locations are arranged so that the 32-bit product can share the 16-bit storage area. This saves memory.

Every display line is followed by a carriage return and line feed. The code for this operation is identified by the name `CR_LF_CODE`. It is located immediately after the product storage area. Thus, when the product is displayed, the display cursor is immediately shifted to the left side of the display and down one line. Many of the other display lines also use this code. A subroutine in the program handles the operation. Two messages that don't use this code are `PRINT_UNDER` and `PRINT_OVER`. Since they must display a message on two lines, they have their own carriage return, line feed code.

The next data segment is called `VECTOR`. It is used to identify the interrupt vector locations for the two software interrupts. Because this segment uses the segment combine type `AT`, no data can be initialized within the segment. This is handled by the program code.

Program code begins in its normal manner. The Stack Segment register and Stack Pointer registers are loaded with the appropriate values. The next three instructions then save the current Data Segment register value and an offset value of zero in the stack. This information will be used by the program at termination, as described in the last experiment.

The next two instructions load the base address of the segment `VECTOR` into the DS register. Then the next four instructions load the offset and base address for the two program interrupt subroutines into the appropriate interrupt vector table locations. Here we use a new assembler operator called `SEG` for segment base address. When the assembler encounters the operator `SEG` followed by a label, it determines what segment contains the label and its segment base address. That base address is then moved into the destination operand. In your program, the offset address for the interrupt on overflow supporting subroutine

is moved into interrupt vector location type 4. Then the code segment base address for the subroutine is moved into the second word of the type four vector table. The process is repeated for the supporting subroutine for the user interrupt 96. When you are using a software interrupt in a program, make sure you pick a vector type that is not being used for some other purpose by the system. Refer to the owner's manual or technical reference manual for your system to determine which vector types are free.

After the interrupt vectors are loaded, the program changes data segments. The `ASSUME` statement tells the assembler of the change. However, the program must make the change by loading the new segment base address into the DS register.

You should be familiar with the next two call instructions and their supporting subroutines. They were described in the last experiment. You selected one or the other to clear the display on your microcomputer. The subroutines are located in Figure 8-13B, right after the main program instructions.

The last two instructions in Figure 8-13A retrieve the numbers to be multiplied and multiply the numbers. The subroutine `GET_NUMBER` performs three basic operations: it asks you for the numbers, it stores the numbers, and it translates the numbers so they can be multiplied. The subroutine is found after the `CLEAR_IBM` subroutine in Figure 8-13B.

The first instruction in the subroutine `GET_NUMBER` is a good illustration of subroutine nesting, as described in Unit 4. The instruction is part of the subroutine `GET_NUMBER`. It calls a second subroutine called `CR_LF`. This can be found at the bottom of Figure 8-13D. The subroutine `CR_LF` sends the code for a carriage return and line feed to the display. However, the actual transfer of code is handled by a third subroutine called `DISPLAY`. This subroutine is found immediately after the `CR_LF` subroutine. It loads the AH register with the interrupt command to send a character string to the display. Then it executes the interrupt, another level of subroutine nesting. Thus, you have at least four levels of subroutine nesting to support one instruction, and possibly more through the display interrupt itself.

After the carriage return, line feed instruction, the effective address of the first message to input a number is loaded into the DX register. The message is then displayed through the DISPLAY subroutine. Then the effective address for the multiplicand buffer is loaded into the DX register. This is the buffer we described earlier. With the address of the buffer in the DX register, the AH register is loaded with the interrupt command to input a character string from the keyboard. Finally, the interrupt is called.

The interrupt routine monitors the keyboard. Each time a key is pressed, the key character is echoed to the display, and the ASCII code for the key is stored in the buffer. This will continue until the buffer is full or the RETURN key is pressed, again as described earlier.

At the completion of the interrupt, the next instruction calls the subroutine to generate a carriage return and line feed. The next six instructions repeat the process to retrieve and store the multiplier value. The two values to be multiplied are stored in memory as two sets of ASCII codes. The rest of the instructions in the subroutine GET\_NUMBER convert the ASCII code to hexadecimal characters and store them in the AX and BX registers. The AX register receives the multiplicand and the BX register receives the multiplier.

The first instruction loads the effective address of the first byte of the multiplicand into the SI register. This will serve as a pointer to the ASCII code in the buffer. Recall that the first ASCII byte is located in the third byte of the memory buffer, hence the source operand `MULT1 + 2` in the instruction.

The next instruction loads the effective address of the buffer character count into the DI register. This will serve as a loop count in the conversion and transfer of the multiplicand. Recall that the keyboard input interrupt routine keeps track of the number of characters that are stored in the buffer and stores that count in the second byte of the memory buffer. For that reason, the source operand in this instruction is `MULT1 + 1`.

The last instruction prior to the conversion loop zeros the AX register. This makes sure that you can enter a one, two, or three hex-character value and not worry about the most significant bits in the register affecting the register value.

To begin the conversion, the ASCII\_TO\_HEX subroutine is called. The subroutine is located near the middle of Figure 8-13C. It moves the ASCII code pointed to by the SI register into the DL register. The SI register is then incremented to point to the next ASCII code. The code is compared with 40H to determine if it is a capital letter. If the code is a number, the upper half-byte is masked, leaving a single-digit decimal number. If the code is a capital letter, 37H is subtracted from the code, leaving a single-digit hexadecimal number (letter). That completes the routine, and the MPU returns to the GET\_NUMBER subroutine.

The contents of the DL register are added to the AL register. Then the DH register is decremented to indicate one character converted. Starting at the top of Figure 8-13C, the DH register is compared with zero to see if the last character has been converted. If not, the CL register is loaded with a count of four. That count is used in the next instruction to shift the contents of the AX register left four bits. This makes room in the register for the next character conversion. Finally, the conversion process is repeated.

When the last ASCII code for the multiplicand has been converted, the program jumps down to the instructions that convert the multiplier ASCII code. These begin at the label NEXT. These are identical to the previous conversion instructions, except that they load the converted hexadecimal values into the BX register. When this conversion is complete, the MPU is sent back to the main program; specifically, the last instruction at the bottom of Figure 8-13A. That instruction multiplies the contents of the AX register by the BX register. The product is stored in the AX register or both the AX and DX registers, depending on the size of the product.

The next instruction, at the top of Figure 8-13B, is a software interrupt. If the product was greater than 16 bits, a type 4 interrupt will be executed. The MPU will save the current Flag, CS, and IP register contents. Then it will load the IP and CS register values stored at location type 4 in the interrupt vector. These values will send the MPU to the subroutine labeled INTO\_INT. It can be found near the middle of Figure 8-13C. The routine simply displays the message:

```
OVERFLOW  
32-BIT PRODUCT =
```

Notice that both the AX and DX registers are temporarily saved in the stack. This is because both registers are involved in displaying the message, and the product must be preserved. The return from interrupt instruction restores the IP, CS, and Flag register values originally stored by the interrupt. This returns the MPU to the main program

If the INTO instruction was executed, the following conditional jump instruction will cause the next instruction to be bypassed. If there was no product overflow, both the INTO and conditional jump instructions will be ignored and the INT 96 instruction will be executed.

This is a user-defined interrupt. When executed, it stores the Flag, CS, and IP register contents in the stack. Then it moves the values at location type 96 in the interrupt vector table into the IP and CS registers. These values will send the MPU to the subroutine labeled SUB\_INT, near the bottom of Figure 8-13C. The routine displays the message:

```
NO OVERFLOW
16-BIT PRODUCT =
```

Notice that only the AX register is saved on the stack. That is because the product didn't exceed the capacity of the AX register. Returning from the interrupt, the IP, CS, and Flag registers are restored, and the next instruction in the main program is executed. This calls the subroutine PRINT\_PRODUCT.

This subroutine is located near the bottom of Figure 8-13C. It is used to convert the hexadecimal product into individual ASCII characters and store the characters in memory. The first instruction zeros the BH register. The register will be used as a flag to indicate whether a 16- or 32-bit product was saved. The following conditional jump tests for an overflow. If no overflow occurred, the 32-bit product conversion is ignored. If there was an overflow, the SI register is loaded with the effective address of the 32-bit storage area plus three.

The routine converts each hexadecimal half-byte in the DX register to its ASCII equivalent and stores the code in memory. Because the conversion begins with the least significant half-byte, storage must begin at the fourth byte location in memory.

After the CH register is loaded with the number of bytes to be saved, the conversion process begins with the low byte of the DX register being moved into the BL register. Then, beginning at the top of Figure 8-13D, the HEX\_TO\_ASCII conversion subroutine is called. This can be found near the middle of Figure 8-13D.

The conversion routine is similar to the earlier ASCII-to-hexadecimal conversion routine. First, the upper half of the BL register is masked to isolate the first half-byte. Then the contents are compared to 40H to determine whether the value is a decimal number or a hexadecimal letter. If it is a letter, 37H is added to the register to produce the appropriate ASCII letter code. If it is a number, 30H is added to the register to produce the appropriate ASCII number code. The ASCII code is then saved in memory.

The SI register is incremented to point to the next storage area, and the CH register is decremented to keep track of the number of characters saved. Finally, the CH register is tested to see if the DX register conversion is done. Before any conditional operations are performed, the MPU is returned to the calling routine. If the CH register is zero, the 16-bit conversion is begun; otherwise, the CL register is loaded with the shift count 4. Then the contents of the DX register are shifted four bits to the right so that the next half-byte can be converted, and the process is repeated.

Had the product contained less than 17 bits, the subroutine would have branched to the instruction at label SMALL1. This instruction increments the BH register to indicate a 16-bit product. The following group of instructions convert the contents of the AX register to ASCII characters. If a 32-bit product is being converted, the increment BH instruction will be bypassed.

The 16-bit product conversion begins at label DOWN. It is identical to the 32-bit conversion except that the contents of the AX, rather than the DX, register are converted. When the last half-byte is converted, the subroutine branches to label PRINT\_VALUE.

The BH register is tested for one. If there is a match, the 16-bit product is displayed. If there is no match, the 32-bit product is displayed. Since the 16-bit storage location is immediately after the 32-bit storage



location, both values will be displayed when the 32-bit display instructions are executed. After the product is displayed, the MPU returns to the main program, where the QUIT subroutine is called. This subroutine can be found near the bottom of Figure 8-13D.

First the repeat question is displayed. Then the interrupt code to input a keyboard character is loaded into the AH register. The interrupt is called, and the MPU waits for an input. When a character is received, it is echoed on the display and the value is stored in the AL register. The AL register contents are tested for the ASCII code of the letter "N". If there is a match, the routine is ended. If there is no match, the register is tested for the ASCII code of the letter "Y". Again, the routine is ended if there is a match. If there is no match, the keyboard is again tested. This will continue until there is a match.

When a match occurs, the main program tests the AL register for the ASCII code of the letter "Y". If there is a match, the MPU branches back to label INPUT\_NUMBER and the program repeats. If there is no match, the program is terminated. Remember from the last experiment, that to terminate an EXE program you need a far return. This is handled by the program code:

```
EXIT  PROC FAR
      RET
EXIT  ENDP
```

Before you proceed with the next program, you may wish to try your hand at programming. Try expanding the power of this program. You might give it the capability to use both upper- and lowercase letters. You might change the input from hexadecimal to decimal. Finally, you might include the ability to add, subtract, and divide two numbers. You have the basic program structure to build upon. While some of the suggested changes won't be easy, making them will be a good learning experience.

The rest of the programs in this experiment won't be as complex as the first. Their sole function is to illustrate the operation of the string instructions. The first of the "string" programs uses the move string instruction.

## Procedure Continued

3. Call up your editor and enter the program listed in Figure 8-15.

```

TITLE EXPERIMENT 8 -- PROGRAM 2 -- MOVING DATA
;
PROG_STACK SEGMENT STACK
    DW    80H DUP (?)    ;Set up stack area
TOP_OF_STACK LABEL WORD    ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_DATA SEGMENT
DATA1 DD SOURCE    ;Determine physical address of SOURCE
DATA2 DD DEST    ;Determine physical address of DEST
PROG_DATA ENDS
;
SOURCE_DATA SEGMENT
SOURCE DB 'This is a string that will be transferred from '
        DB 0DH,0AH
        DB 'the source area to the destination area of memory.$'
SOURCE_DATA ENDS
;
DEST_DATA SEGMENT
DEST DB 100 DUP (0)
DEST_DATA ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,DS:PROG_DATA,SS:PROG_STACK
START: MOV AX,PROG_STACK    ;Never load a segment register direct
        MOV SS,AX    ;Use an intermediate register
        NOP    ;No operation to allow observation of
                ;next instruction when single-stepping
        MOV SP,OFFSET TOP_OF_STACK ;Point to the top of stack
                ;immediately after loading SS register
        MOV AX,PROG_DATA    ;Next,indirectly load the
        MOV DS,AX    ;Data Segment register
        NOP    ;Another no operation for observation
        LES DI,DATA2    ;Get extra segment and offset
                ;address to destination location
        LDS SI,DATA1    ;Get data segment and offset
                ;address to source data location
        ASSUME DS:SOURCE_DATA,ES:DEST_DATA
        MOV CX,100    ;Set loop count
        REP MOVS DEST,SOURCE    ;Move the string
        INT 3    ;Return to the debugger
;
PROG_CODE ENDS
        END START

```

**Figure 8-15**

Program using the move string instruction.

This program is a simple illustration of a string move operation. It uses three data segments to hold address information, a source string, and a destination string storage area. We had you add a no operation (do nothing) instruction after each segment register move instruction so you could see the next “useful” instruction while single-stepping through the program.

4. Assemble and link your program.
5. Call up the debugger and load your program. Single-step through the program up to the LES DI,DATA2 instruction. The DS register now contains the base address of the PROG\_DATA segment. Examine that segment — type “D<DS register contents>:0000” and RETURN. The base address for the SOURCE\_DATA segment is \_ \_ \_ \_H. The base address for the DEST\_DATA segment is \_ \_ \_ \_H. Remember, the assembler directive Define Doubleword tells the assembler/linker to identify both the offset and base address of the directive argument.
6. Verify the address values recorded in step 5 by single-stepping through the next two instructions. The DS and ES registers should now contain the values you recorded. Examine the DEST\_DATA segment. It should contain 100 bytes of zeros (actually 112 bytes because of the “paragraph boundary” segment attribute). Examine the SOURCE\_DATA segment. It should contain the character string you entered with the program.
7. Examine the MPU registers — type “R” and RETURN. The SI register contains the value \_ \_ \_ \_H and the DI register contains the value \_ \_ \_ \_H. Single step through the next instruction. The CX register contains the value \_ \_ \_ \_H. Now single step three more times. After each single-step operation, record the information in the following blanks.

Step 1— DF = \_  
CX = \_ \_ \_ \_H.  
SI = \_ \_ \_ \_H.  
DI = \_ \_ \_ \_H.  
IP = \_ \_ \_ \_H.

Step 2— DF = \_\_.  
CX = \_\_\_\_H.  
SI = \_\_\_\_H.  
DI = \_\_\_\_H.  
IP = \_\_\_\_H.

Step 3— DF = \_\_.  
CX = \_\_\_\_H.  
SI = \_\_\_\_H.  
DI = \_\_\_\_H.  
IP = \_\_\_\_H.

Finally, run the remaining portion of the program — type “G” and RETURN. Record the contents of the following registers.

CX = \_\_\_\_H.  
SI = \_\_\_\_H.  
DI = \_\_\_\_H.

8. Examine the DEST\_DATA segment. You should see the message originally stored in the SOURCE\_DATA segment.

## Discussion

Using the load extra segment, load data segment, and move string instructions reduces what could be a complex program into a simple program. The whole operation, including the load CX register instruction, required only five instructions and one ASSUME statement. Probably the only problem you will have when using string instructions is remembering to add an ASSUME statement every time you change the contents of either the Data or Extra Segment register.

When you single-stepped through the move string instruction in step 7, the REP prefix made sure that the move operation was repeated as long as the CX register contained a value other than zero. Each time the instruction was executed, the CX register was decremented by one and the SI and DI registers were incremented by one. The CX register is used by the instruction to keep track of the number of moves. The SI and DI registers were incremented by one because the instruction was moving a byte value rather than a word value. The two registers

were incremented, rather than decremented, because the Direction flag was clear (register display showed direction code UP). Had the Direction flag been set, the registers would have decremented.

After you ran the program to completion, the CX register contained zero, and both the SI and DI registers contained 0064H, the original CX register value. When you examined the DEST\_DATA segment, you found the message originally stored in the SOURCE\_DATA segment.

The program you just executed used “symbolic” references to specify the source and destination operands. The next program will perform the same operation using “anonymous” references.

## Procedure Continued

9. Exit the debugger. Then call up your editor and modify your move string program as follows:

- A. Change the instruction

```
        MOV  CX,100           ;Set loop count  
  
to read
```

```
        MOV  CX,50           ;Set loop count
```

- B. Change the instruction

```
        REP  MOVSB DEST,SOURCE ;Move the string  
  
to read
```

```
        REP  MOVSW           ;Move the string
```

10. Assemble and link the program.
11. Call up the debugger and load your program. Single step up to the move count instruction. You should observe the same results as those obtained in steps 5 and 6 of the previous “Procedure.” Examine the SOURCE\_DATA segment. It should contain the message. Examine the DEST\_DATA segment. It should contain zeros.

12. Single step once. The CX register is loaded with the value 32H. Now single step one more time. The SI and DI registers are incremented twice. Why would that happen?
13. Run the program — type “G” and RETURN. Examine the DEST\_DATA segment. You should again find the message copied to that segment.

## Discussion

With the program modified, it now transfers the message one word at a time. Because words are being transferred, the SI and DI registers are incremented twice for each move operation. You had to halve the value moved into the CX register because the transfer rate was doubled.

Because you used an anonymous string move instruction in the program, you were able to use a “move word” instruction with byte-sized data. This is not a good programming practice. There are too many opportunities for program failure. If you must use an anonymous instruction, make sure the instruction data type matches the data type assigned to the source and/or destination memory location. Now let's look at the compare and scan string instructions.

## Procedure Continued

14. Exit the debugger. Then call up the editor and load your move string program. Now refer to Figure 8-16 and insert the instructions listed between the two rows of Xs into your program at the location shown. You will use these instructions to examine the operation of the compare and scan string instructions.
15. Assemble and link your program.
16. Call up the debugger and load your program. Use the Go command with a break point to run your program up to the first new instruction — type “G001C” and RETURN. Both the SOURCE\_DATA and DEST\_DATA segments now contain the program message.

```

TITLE EXPERIMENT 8 -- PROGRAM 4 -- USING COMPARE AND SCAN STRING
;
PROG_STACK SEGMENT STACK
    DW      80H DUP (?)      ;Set up stack area
TOP_OF_STACK LABEL WORD    ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_DATA SEGMENT
DATA1 DD      SOURCE        ;Determine physical address of SOURCE
DATA2 DD      DEST         ;Determine physical address of DEST
PROG_DATA ENDS
;
SOURCE_DATA SEGMENT
SOURCE DB      'This is a string that will be transferred from '
        DB      0DH,0AH
        DB      'the source area to the destination area of memory.$'
SOURCE_DATA ENDS
;
DEST_DATA SEGMENT
DEST DB      100 DUP (0)
DEST_DATA ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,DS:PROG_DATA,SS:PROG_STACK
START: MOV     AX,PROG_STACK  ;Never load a segment register direct
        MOV     SS,AX        ;Use an intermediate register
        NOP                    ;No operation to allow observation of
                                ;next instruction when single-stepping
        MOV     SP,OFFSET TOP_OF_STACK ;Point to the top of stack
                                ;immediately after loading SS register
        MOV     AX,PROG_DATA  ;Next,indirectly load the
        MOV     DS,AX        ;Data Segment register
        NOP                    ;Another no operation for observation
        LES     DI,DATA2     ;Get extra segment and offset
                                ;address to destination location
        LDS     SI,DATA1     ;Get data segment and offset
                                ;address to source data location
        ASSUME DS:SOURCE_DATA,ES:DEST_DATA
        MOV     CX,50        ;Set loop count
        REP MOVSW            ;Move the string
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        LEA     SI,SOURCE     ;Get source offset address
        LEA     DI,DEST      ;Get destination offset address
        MOV     CX,100       ;Load string length
        MOV     BYTE PTR [SI]+0FH,'X' ;Store odd character in message
        REPE CMPS SOURCE,DEST ;Check string accuracy
        LEA     DI,DEST      ;Get destination offset address
        MOV     CX,100       ;Load string length
        MOV     AL,'g'       ;Load character to be located (scanned)
        REPNE SCASB         ;Look for character 'g'
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        INT     3            ;Return to the debugger
;
PROG_CODE ENDS
        END     START

```

**Figure 8-16**

Program to show the operation of the compare and scan string instructions.

17. Single step through the next three instructions. Both the SI and DI registers are again loaded with the effective address of the message in their respective segments. The CX register is loaded with the message length.
18. Examine offset address 000FH in the SOURCE\_DATA segment (type "D<Data Segment register contents>:0000" and RETURN). It contains the value 67H, which is the ASCII code for the letter "g". Single step through the next instruction. This loads the ASCII code for the letter "X" into the SOURCE\_DATA segment memory location pointed to by the effective address in the SI register plus 0FH. Examine offset address 000FH in the SOURCE\_DATA segment one more time. It now contains 58H, the ASCII code for the letter "X". Record the following register information.

ZF = \_\_.  
CX = \_\_\_\_H.  
SI = \_\_\_\_H.  
DI = \_\_\_\_H.

19. Single step through the compare string instruction one time. Record the following register information.

ZF = \_\_.  
CX = \_\_\_\_H.  
SI = \_\_\_\_H.  
DI = \_\_\_\_H.

20. Continue to single step through the string compare until the string repeat loop ends. Record the following register information.

ZF = \_\_.  
CX = \_\_\_\_H.  
SI = \_\_\_\_H.  
DI = \_\_\_\_H.

21. Again examine the SOURCE\_DATA segment. The character "X" is located in the \_\_\_\_\_ byte position. Does that match the value stored in the SI and DI registers?



- 22. Single step through the next three instructions. Again, the DI register is loaded with the effective address of the message in the DEST\_DATA segment and the CX register is loaded with the message length. Finally, the AL register is loaded with the ASCII code for the character "g". Record the following register information.

ZF = \_\_.  
CX = \_\_\_\_\_H.  
SI = \_\_\_\_\_H.  
DI = \_\_\_\_\_H.

- 23. Single step through the scan string byte instruction one time. Record the following register information.

ZF = \_\_.  
CX = \_\_\_\_\_H.  
SI = \_\_\_\_\_H.  
DI = \_\_\_\_\_H.

- 24. Continue to single step through the string scan until the string repeat loop ends. Record the following information.

ZF = \_\_.  
CX = \_\_\_\_\_H.  
SI = \_\_\_\_\_H.  
DI = \_\_\_\_\_H.

- 25. Again examine the DEST\_DATA segment. The character "g" is located in the \_\_\_\_\_ byte position. Does that match the value stored in the DI register?

## Discussion

The program you just executed showed the operation of the compare string and scan string instructions. The first part of the program simply transferred the message in the SOURCE\_DATA segment into the DEST\_DATA segment so that you could perform the compare and scan string operations. The first three instructions that you single-stepped through returned the CX, SI, and DI registers to their original values in preparation for the next string operation. Finally, to give an odd character for the compare string operation to locate, the character "X" is stored in the sixteenth byte position of the message in the SOURCE\_DATA segment.

Prior to executing the compare string instruction, the Zero flag was clear. After the instruction was executed, the Zero flag was set, the CX register was decremented, and the SI and DI registers were incremented. As long as the Zero flag remained set, the repeat while equal prefix code caused the MPU to loop back to the compare string instruction. The loop ended when the Zero flag was cleared. At that point, the SI and DI registers contained the value 10H, indicating the sixteenth byte position contained the odd character. Keep in mind, however, that if you wish to use the count in the SI or DI register as an offset value to locate the odd character, you must subtract one from the count. Remember, the MPU begins counting from zero. Thus, the sixteenth byte position has an offset of 0FH from the first byte position.

The scan string portion of the program operated in essentially the same manner as the compare string portion. Only in this case, the message in the DEST\_DATA segment was compared to the contents of the AL register. The scan string operation was allowed to repeat as long as the Zero flag remained clear (repeat not equal). The loop ended when the character "g" was identified. Again, the contents of the DI register (10H) indicated that the character was located in the sixteenth byte position. As before, one should be subtracted from that value to give the true character offset of 0FH.

The last program you will write illustrates the load string instruction, the store string instruction, and the two Direction flag control instructions.

## Procedure Continued

26. Exit the debugger. Then call up the editor and load your string program one more time. Delete the

```
DEST DB 100 DUP (0)
```

statement, and all of the instructions after the ASSUME statement:

```
ASSUME DS:SOURCE_DATA,ES:DEST_DATA
```

Refer to Figure 8-17 and add the assembler directive LABEL statement and the EQU statement, inside the two lines of Xs, to the SOURCE\_DATA segment as shown. Then add the Define Byte statement and the LABEL statement, inside the two lines of Xs, to the DEST\_DATA segment as shown. Finally, add the eight instructions, inside the two lines of Xs, to the PROG\_CODE segment as shown.

The statement

```
SOURCE1 LABEL BYTE
```

identifies the next memory location after the message character string. This is used by the EQU statement

```
MESSAGE_SIZE EQU OFFSET SOURCE1 - OFFSET SOURCE
```

to calculate the length of the character string. Subtracting one offset from the other gives you the number of bytes in the string. This can be very useful when you are dealing with a number of character strings and must make sure you have the exact count. To prevent forward reference problems, the equate statement should be positioned after the defined data areas and before you use the “equated” symbol in another statement or instruction. In addition, be sure to position the equate statement inside a program segment area as we did in this program.

```

TITLE EXPERIMENT 8 -- PROGRAM 5 -- LOAD, STORE, AND FLAG STRINGS
;
;
PROG_STACK SEGMENT STACK
    DW      80H DUP (?)      ;Set up stack area
TOP_OF_STACK LABEL WORD    ;Identify top of stack for SP register
PROG_STACK ENDS
;
;
PROG_DATA SEGMENT
DATA1 DD SOURCE            ;Determine physical address of SOURCE
DATA2 DD DEST              ;Determine physical address of DEST
PROG_DATA ENDS
;
;
SOURCE_DATA SEGMENT
SOURCE DB 'This is a string that will be transferred from '
        DB 0DH,0AH
        DB 'the source area to the destination area of memory.$'
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
SOURCE1 LABEL BYTE
MESSAGE_SIZE EQU OFFSET SOURCE1 - OFFSET SOURCE
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
SOURCE_DATA ENDS
;
;
DEST_DATA SEGMENT
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
DEST DB MESSAGE_SIZE DUP (0)
DEST1 LABEL BYTE
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
DEST_DATA ENDS
;
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,DS:PROG_DATA,SS:PROG_STACK
START: MOV AX,PROG_STACK ;Never load a segment register direct
        MOV SS,AX        ;Use an intermediate register
        NOP              ;No operation to allow observation of
                        ;next instruction when single-stepping
        MOV SP,OFFSET TOP_OF_STACK ;Point to the top of stack
                        ;immediately after loading SS register
        MOV AX,PROG_DATA ;Next,indirectly load the
        MOV DS,AX        ;Data Segment register
        NOP              ;Another no operation for observation
        LES DI,DATA2     ;Get extra segment and offset
                        ;address to destination location
        LDS SI,DATA1     ;Get data segment and offset
                        ;address to source data location
        ASSUME DS:SOURCE_DATA,ES:DEST_DATA
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        LEA DI,DEST1-1   ;Offset to last byte of destination
        MOV CX,MESSAGE_SIZE ;Get message size
                        ;to use as a repeat count
REPEAT: CLD              ;Clear DF to increment source register
        LODS SOURCE      ;Move source byte to AL register
        STD              ;Set DF to decrement destination reg.
        STOS DEST        ;Move AL reg. contents to destination
        LOOP REPEAT      ;Repeat string move message size times
        INT 3            ;Return to the debugger
;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
;
PROG_CODE ENDS
        END START

```

Figure 8-17

Program illustrating the remaining string instructions.

The statement

```
DEST      DB  MESSAGE_SIZE DUP (0)
```

uses the equated symbol to define the size of the character string storage area. The statement

```
DEST1     LABEL      BYTE
```

will be used by the program to identify the end of the storage area. We'll describe the new program instructions later.

27. Assemble and link the program.
28. Call up the debugger and load your program. Use the Go command with a break point to run your program up to the first new instruction — type “G0017” and RETURN. Now examine the DEST\_DATA segment to verify that it is filled with zeros. The program will use the load string and store string instructions to move the message stored in the SOURCE\_DATA segment into the DEST\_DATA segment.
29. Single step through the first instruction. The DI register contains the value `_____H`. To keep things interesting, the program will store the message backward in memory. This instruction loads the effective address of the last storage location into the DI register. The assembler directive statement

```
DEST1     LABEL      BYTE
```

identifies the location of the first byte after the message storage area. By taking the effective address of that label and subtracting one, you have the effective address of the last byte in the storage area. Thus, the effective address 63H is stored in the DI register. The effective address for the SOURCE\_DATA segment message was stored earlier in the SI register by the load data segment instruction.

30. Single step through the next instruction. The CX register contains the value `_____H`. Here we have a new method for loading the size of a character string. Instead of counting the characters in the string, we use the equate statement described earlier to load the string size in the Count register. The count, as in the previous programs, is 64H.

31. Single step through the next instruction. The Direction flag is \_\_\_\_\_.
32. Single step through the next instruction. The AL register contains the value \_\_H and the SI register contains the value \_\_\_H.
33. Single step through the next instruction. The Direction flag is \_\_\_\_\_.
34. Single step through the next instruction. The DI register contains the value \_\_\_H.
35. Single step through the next instruction. The CX register contains the value \_\_\_H. Because the Count register is not zero, the MPU has branched back to the clear Direction flag instruction.
36. Single step through the next instruction. The Direction flag is \_\_\_\_\_.
37. Continue to single step through the program loop a couple of times and compare the results with the previous five steps. Then run the program to completion — type “G” and RETURN. The SI register contains the value \_\_\_H and the DI register contains the value \_\_\_H.
38. Examine the DEST\_DATA segment. Notice that the message was transferred front-to-back.

## Discussion

Your program essentially duplicated the operation performed by the move string instruction. However, it gave you the opportunity to observe the operation of the LODS, STOS, CLD, and STD instructions. As each character was moved out of the SOURCE\_DATA segment and into the AL register, the SI register was incremented. Then when the character was moved from the AL register into the DEST\_DATA segment, the DI register was decremented. The state of the Direction flag determined register direction (increment or decrement).

Neither string instruction affected the contents of the CX register. This will only happen when the repeat prefix code is used with a string instruction. In this program the CX register was decremented by the loop instruction.

This completes the Experiment for Unit 8. Proceed to the Unit 8 Examination.





## UNIT 8 EXAMINATION

1. There are two types of interrupts: \_\_\_\_\_  
and \_\_\_\_\_.
2. Describe what happens during a typical interrupt. \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_
3. A form of interrupt that is used to restart or initialize the MPU is called a \_\_\_\_\_.
4. Write the instruction mnemonic and, if necessary, the prefix code that will:
  - A. Compare two byte strings until two like values are found, or the contents of the CX register is zero. \_\_\_\_\_
  - B. Move a string of words into memory. \_\_\_\_\_
  - C. Compare the elements of a string with the contents of the AL register and stop execution when the contents of the CX register are zero, or a value different from the contents of the AL register is found. \_\_\_\_\_
  - D. Load a word of a string into the AX register. \_\_\_\_\_
  - E. Store the contents of the AX register into a memory location. \_\_\_\_\_

## EXAMINATION ANSWERS

1. There are two types of interrupts: **external** and **internal**.
2. During a typical interrupt, the MPU completes the execution of the current instruction, pushes the contents of the CS, IP, and Flag registers into the stack, and retrieves the IP and CS register values from the interrupt vector table. After the interrupt is serviced, the Flag, IP, and CS register values are popped from the stack, and program execution continues from the point of the interrupt.
3. A form of interrupt that is used to restart or initialize the MPU is called a **reset**.
4. 

A.	Compare two byte strings until two like values are found, or the contents of the CX register is zero.	<b>REPNE CMPSB</b>
B.	Move a string of words into memory.	<b>REP MOVSW</b>
C.	Compare the elements of a string with the contents of the AL register and stop execution when the contents of the CX register are zero, or a value different from the contents of the AL register is found.	<b>REPZ SCASB</b>
D.	Load a word of a string into the AX register.	<b>LODSW</b>
E.	Store the contents of the AX register into a memory location.	<b>STOSW</b>

## SELF-REVIEW ANSWERS

1. Generally speaking, an interrupt is a temporary break in the normal execution of a program; after which, program execution resumes at the point of the break.
2. The actions that the MPU takes in response to an interrupt are called the **interrupt service** routine or **interrupt** routine.
3. Responding to an interrupt is referred to as **servicing** the interrupt.
4. The two basic types of interrupts are the **internal** interrupt and the **external** interrupt.
5. Under the general heading of external interrupts, there are the **maskable** interrupt and the **non-maskable** interrupt.
6. The **interrupt vector table**, or **interrupt pointer table**, is a list of starting, or physical, addresses for the various interrupt routines.
7. The first two bytes of each interrupt vector, or pointer, contain the **Instruction Pointer** value, while the next two bytes contain the **Code Segment** value.
8. Each interrupt routine must end with the **IRET (interrupt return)** instruction.
9. A **divide error** interrupt will result after the execution of a DIV instruction if the quotient is too large to fit in the destination register.
10. During **single step** operation, the MPU is directed to a special subroutine after the execution of each instruction.
11. Single step operation can be used to help debug a program.
12. In order to use the single step operation, you must set the **Trap** flag.
13. The two software interrupt instructions that use an instruction mnemonic are **INT (interrupt)** and **INTO (interrupt on overflow)**.

14. When the instruction **INT 96** is executed, a type 96 interrupt is generated. The contents of the Flag, CS, and IP registers are pushed into the stack. Then the physical address of the interrupt routine, found at the address indicated in the type 96 vector, is loaded into the CS and IP registers.
15. The interrupt instruction **INTO** will only be executed if an overflow condition exists.
16. Segment attribute **AT** is used to define the base address of a segment.
17. The assembler operator **SEG (segment)** is used to identify the segment base address of a symbol.
18. **False.** While the segment combine-type attribute **AT** is used to specify the base address of a segment, no data can be initialized within that segment; only symbols may be identified.
19. The two control lines on the 8088 MPU that can be used to indicate an external interrupt are the **INTR** line and the **NMI** line.
20. When a peripheral device requests an interrupt on the **INTR** line, the MPU finishes executing the current instruction, stores the contents of the Flag, CS, and IP registers in the stack, and then generates an interrupt acknowledge. At that time, the peripheral device places the interrupt pointer number on the data bus so that the MPU “knows” which interrupt routine to execute.
21. The back-and-forth conversation between the MPU and the peripheral over the **INTR** and **INTA** lines is called **handshaking**.
22. Vectored interrupt routines must conclude with the **IRET** instruction.
23. **True.** If you want the MPU to “ignore” interrupt requests on the **INTR** line, then the **IF** flag must be clear.
24. The **STI** instruction is used to set the Interrupt flag.
25. The Interrupt flag is cleared by the **CLI** instruction.

26. Even if the Interrupt flag is clear, the **Non-Maskable (NMI)** interrupt will not be disabled.
27. In what order are internal and external interrupts serviced?
  - A. Divide Error, INT, and INTO.
  - B. NMI.
  - C. INTR.
  - D. Single Step.
28. A **reset** is used any time it is necessary to restart or initialize the MPU.
29. What are the contents of the following, after a reset has occurred?

A. Flag Register	0000H
B. IP Register	0000H
C. CS Register	0FFFFH
D. DS Register	0000H
E. ES Register	0000H
F. SS Register	0000H
G. Queue	Empty
30. The process by which peripheral devices communicate directly with the microcomputer's memory is called **Direct Memory Access (DMA)**.
31. In order for a peripheral to obtain "permission" to perform a direct memory access, it must initiate a hold request on the **HOLD** control line.
32. The MPU responds to a hold request by outputting a high on the **HLDA** control line.
33. A **string** is a number of bytes or words that reside in sequential memory locations.
34. Each item in a string can also be called an **element**.
35. An operation that is performed on each element of a string is called a **string** operation.
36. In a string operation, the number of times the operation is repeated is specified in the **CX**, or **Count**, register.

37. The default segment for the destination string is the **extra** segment.
38. The default general, or offset address, register that is used to point to the memory location of the destination string is the **DI**, or **Destination Index**, register.
39. The default general, or offset address, register that is used to point to the memory location of the source string is the **SI**, or **Source Index**, register.
40. **False**. Only a **move string** instruction can be used to transfer data from one memory location to another.
41. The string prefix code **REP**, or **repeat string**, causes a string operation to be repeated count times.
42. The instruction

MOVS DEST, SOURCE

is said to use a **symbolic** reference in determining the source and destination string locations.

43. An anonymously referenced string instruction defaults to the **data** segment for determining the location of the source string.
44. The SI and DI registers are incremented or decremented **twice** after a word-sized string element is moved.
45. If the source string is identified by the symbol SOURCE and the destination string is identified by the symbol DEST, write the symbolically referenced instruction that will:
  - A. Move a byte string: **REP MOVS DEST, SOURCE**
  - B. Compare an element in two word strings: **CMPS SOURCE, DEST**
  - C. Compare a word value to a string element: **SCAS DEST**
  - D. Store a byte string: **REP STOS DEST**
  - E. Load a word-sized string element into the accumulator: **LODS SOURCE**

46. Write the short anonymous instruction that will fulfill each of the following conditions:
- A. Move a word string:                    **REP MOVSW**
  - B. Compare an element in two byte strings:                    **CMPSB**
  - C. Compare a byte value to a string element:                    **SCASB**
  - D. Store a word string:                    **REP STOSW**
  - E. Load a byte-sized string element into the accumulator:   **LODSB**
47. If the destination string element contains the value 42H and the source string element contains the value 88H, the CMPSB instruction will subtract the destination value from the source value, giving the result 46H. This value is then used by the MPU to set or clear the six arithmetic flag bits. The flag conditions are as follows:
- A. **Overflow flag:** Set because there was a borrow from the most significant source bit, but no borrow into the most significant source bit.
  - B. **Sign flag:** Cleared because the most significant result bit was cleared, indicating a positive result.
  - C. **Zero flag:** Cleared because the result was not zero.
  - D. **Auxiliary Carry flag:** Cleared because there was no auxiliary carry during the subtraction.
  - E. **Parity flag:** Cleared because there was an odd number of one-bits in the result.
  - F. **Carry flag:** Cleared because there was no borrow into the most significant source bit.
48. The contents of the **Direction flag** determine the direction of a string operation.
49. If the Direction flag is set during a string operation, the SI and/or DI registers will **decrement**.
50. The **CLD** instruction is used to clear the direction flag.
51. The **STD** instruction is used to set the direction flag.

52. The **REPE** and **REPZ** string prefix codes decrement the CX register, test the register for zero, and check to see if the Zero flag is set.
53. The **REPNE** and **REPNZ** string prefix codes decrement the CX register, test the register for zero, and check to see if the Zero flag is clear.
54. The **LDS** instruction is used to load the data segment register and a 16-bit general register.
55. The **LES** instruction is used to load the extra segment register and a 16-bit general register.
56. When dealing with string operations, the 16-bit general register that is loaded by the LES instruction should be the **DI**, or **Destination Index** register.
57. The best way to load the base and offset address of a string in memory is with the **DD**, or **Define Doubleword** assembler directive.
58. When the LDS and LES instructions are executed one after the other, the **LES** instruction should be executed first, to make sure it retrieves the correct values from the data segment before the data segment reference is possibly changed.
59. **False.** You can use both symbolic and anonymous references with the LDS and LES instructions.



INSERT



*Unit 9*

**CODE MACROS  
AND OTHER INTERESTING  
MACRO-86 FEATURES**

## CONTENTS

Introduction .....	9-3
Unit Objectives .....	9-4
Unit Activity Guide .....	9-5
Procedures .....	9-6
The Group Directive .....	9-15
Conditional Directives .....	9-20
Macro Directives and Operators .....	9-32
Assembler Listing Directives .....	9-54
Experiment .....	9-64
Unit 9 Examination .....	9-83
Examination Answers .....	9-85
Self-Review Answers .....	9-87

## INTRODUCTION

You now have the mental tools to write and assemble a detailed program using MACRO-86. There are, however, certain functions and features of MACRO-86 that can make the job of programming a little easier. While you may never need to use these items, you should be aware of how they work. This can be very important if you are trying to understand a program listing that uses all of the power of MACRO-86.

The first item that we will cover is the procedure. Recall that you used a FAR procedure to exit an EXE-type program. Procedures perform a useful task by providing more structure to a program. You will learn to use both NEAR and FAR procedures.

Next you will study groups. These are special directives that allow you to gather segments under a single symbol for specific coding techniques.

Following groups, you will learn how to use the conditional assembly directives. These give you the power to set up program subsections that will only assemble when certain conditions exist.

Then, you will learn how to create your own unique program instructions. These special instructions are called code macros. While they resemble a program subroutine, they have many additional features.

The last part of the unit will cover the assembler listing directives. While not directly related to the actual assembly of a program, they do control how the program listing is presented both on the display and in a printout.

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 directives: PROC, ENDP, GROUP, IF (and its derivatives), ELSE, ENDIF, MACRO, ENDM, EXITM, LOCAL, PURGE, REPT, IRP, IRPC, PAGE, SUBTTL, %OUT, .LIST, .XLIST, .SFCOND, .LFCOND, .XALL, .LALL, and .SALL.
2. Describe the macro operators: &, <>, ;;, !, and %.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read the Section on "Procedures."	_____
<input type="checkbox"/> Complete Self-Review Questions 1-9.	_____
<input type="checkbox"/> Read the Section on "The Group Directive."	_____
<input type="checkbox"/> Complete Self-Review Questions 10-15.	_____
<input type="checkbox"/> Read the Section on "Conditional Directives."	_____
<input type="checkbox"/> Complete Self-Review Questions 16-31.	_____
<input type="checkbox"/> Begin Reading the Section on "Macro Directives and Operators."	_____
<input type="checkbox"/> Complete Self-Review Questions 32-40.	_____
<input type="checkbox"/> Continue Reading the Section on "Macro Directives and Operators."	_____
<input type="checkbox"/> Complete Self-Review Questions 41-48.	_____
<input type="checkbox"/> Read the Section on "Assembler Listing Directives."	_____
<input type="checkbox"/> Complete Self-Review Questions 49-61.	_____
<input type="checkbox"/> Perform the Experiment.	_____
<input type="checkbox"/> Complete the Unit 9 Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

## PROCEDURES

Procedures are sophisticated forms of subroutines. They add a more identifiable framework to a program. That is, they rigidly define the beginning and end of a subroutine. Because of this rigid structure, they are an aid for constructing “good” program code. And as you learned in Unit 7, they are also necessary if you wish to gracefully exit an EXE-type program. We’ll begin our description of procedures with a look at their structure.

### Structure

The structure of a procedure is similar to the structure of a segment. That is, its beginning and its end must be defined with an assembler directive. The typical structure takes the form

```
<proc-name> PROC <type>
    .
    .
    ;procedure code
    .
    .
    RET
<proc-name> ENDP
```



where **<proc-name>** is a name unique to the program, and **<type>** is the type of procedure: NEAR or FAR. If no type is specified, the default type is **NEAR**. The directive **PROC** identifies the procedure beginning. When a procedure is “called,” the procedure name is used to identify the procedure location. Program execution then begins at the first instruction within the procedure. The directive **ENDP** identifies the end of a procedure for the assembler. It plays no part in program execution.

Notice that we included a RET instruction within the procedure structure. Every procedure should have at least one return instruction; it is, after all, a subroutine. The assembler uses the procedure “type” to determine whether the return instruction is coded for a NEAR return or a FAR return. The assembler also uses the procedure type to determine the coding (NEAR or FAR) for the “calling” instruction.

## Calling A Procedure

When you call a NEAR procedure, the contents of the Instruction Pointer are pushed into the stack, and the effective address of the procedure is loaded into the IP register. This procedure address is also the address of the first instruction in the procedure. The procedure return instruction will pop only the IP register contents because the assembler has given it the same type (NEAR) as the procedure.

```

The Microsoft MACRO Assembler          07-17-84    PAGE    1-1
UNIT 9 -- PROGRAM 1 -- CALLING A NEAR PROCEDURE

1                                     TITLE UNIT 9 -- PROGRAM 1 -- CALLING A
                                     NEAR PROCEDURE
2                                     ;
3      0000                          PROG_STACK SEGMENT STACK
4      0000      50 [                  DW      80 DUP (0)
5                                     0000
6                                     ]
7
8      00A0                          TOP_OF_STACK LABEL WORD
9      00A0                          PROG_STACK ENDS
10                                     ;
11     0000                          PROG_CODE SEGMENT
12                                     ASSUME CS:PROG_CODE,SS:PROG_STA
                                     CK
13     0000 B8 ---- R                  START: MOV    AX,PROG_STACK
14     0003 8E D0                      MOV    SS,AX
15     0005 BC 00A0 R                   MOV    SP,OFFSET TOP_OF_STACK
16     0008 E8 000C R                   CALL   COUNT_DOWN
17     000B CC                          INT    3
18                                     ;
19     000C                          COUNT_DOWN PROC NEAR
20     000C B9 BBBB                      MOV    CX,0BBBBH
21     000F E2 FE                       AGAIN: LOOP AGAIN
22     0011 C3                          RET
23     0012                          COUNT_DOWN ENDP
24                                     ;
25     0012                          PROG_CODE ENDS
26                                     END    START

```

```

The Microsoft MACRO Assembler          07-17-84    PAGE    Symbols
                                     -1
UNIT 9 -- PROGRAM 1 -- CALLING A NEAR PROCEDURE

```

Segments and groups:

Name	Size	align	combine	class
PROG_CODE . . . . .	0012	PARA	NONE	
PROG_STACK . . . . .	00A0	PARA	STACK	

Symbols:

Name	Type	Value	Attr
AGAIN . . . . .	L NEAR	000F	PROG_CODE
COUNT_DOWN . . . . .	N PROC	000C	PROG_CODE
START . . . . .	L NEAR	0000	PROG_CODE
TOP_OF_STACK . . . . .	L WORD	00A0	PROG_STACK

```

Warning Severe
Errors Errors
0      0

```

**Figure 9-1**  
Assembler listing of a program that uses a NEAR procedure.

Figure 9-1 is an assembler listing of a simple program that uses a procedure to identify a subroutine. Comments have been left out of the program to make it easier to read. Program lines 1 through 15 contain the standard segment and register initialization code. The procedure is called by the instruction on line 16.

The procedure `COUNT_DOWN` is listed on lines 19 through 25. It is a simple time delay. The count register is loaded with the value `0BBBBH`; then it is decremented by the loop instruction. The following return from call instruction causes a branch back to the main program. Because the procedure was given a type `NEAR`, the return instruction is also a type `NEAR`. The program ends with a type 3 interrupt.

Following the program listing is the standard symbols listing. Notice that `COUNT_DOWN` has a "Type" `N PROC (NEAR procedure)`, it begins at offset `000CH (Value)`, it is located in the `PROG_CODE` segment (Attribute), and it contains six bytes of code (`Length = 0006H`).

Calling a `FAR` procedure causes the contents of the `CS` and `IP` registers to be pushed into the stack. Again, the procedure address is the same address as the first instruction in the procedure. When the return from call instruction in the procedure is executed, the `IP` and `CS` values are popped from the stack, and the program continues execution from that location.

Figure 9-2 is an assembler listing of a program that performs the same operation as the program in Figure 9-1. Only this time, we are dealing with a FAR procedure call. To accomplish this task, the procedure was pulled from the main code segment and placed in a separate (private) code segment. This segment, called `FAR_CODE`, is listed on lines 11 through 18. In addition to the `SEGMENT` and `ENDS` directives, we have added an `ASSUME` directive to tell the assembler that it is dealing with a different code segment. The only other change is the type `FAR` replacing the previous type `NEAR` in the `PROC` directive. Notice that the “secondary” code segment precedes the “main” code segment. This is necessary to prevent any forward reference problems during assembly. Always use this arrangement when you must use an intersegment FAR procedure.

Compare the two procedure call instructions. In Figure 9-1, the program code contains the instruction opcode for a `NEAR` call and the two-byte address offset. Recall that the letter “R” following the code is a flag for the linker to indicate that it might be necessary to adjust the address offset during linking. In Figure 9-2, the program code contains the instruction opcode for a `FAR` call, the two-byte address offset, and four dashes. The dashes indicate that the “called” segment base address must be calculated by the linker.

The program listed in Figure 9-2 is typical of a program with more than one code segment and no external references. If you wish to link one or more external programs to your main program, use the `EXTRN` and `PUBLIC` directives described in Unit 7. They operate in the same manner whether your programs contain procedures or not.

The two examples of calling a procedure used the procedure name to determine the address of the first instruction in the procedure. You can also call any label within a procedure. However, if the label is within a FAR procedure, you must declare that label a FAR label using the `LABEL` directive. Labels within a `NEAR` procedure are, by default, `NEAR`.

The programs in Figures 9-1 and 9-2 used a type 3 interrupt to end the program. However, as you know, this only works when you are using the debugger or another program that supports a type 3 interrupt. A normal exit for an `EXE`-type program is through a FAR return. Earlier, you used a “dummy” FAR procedure to provide that return. Now that you understand what a procedure is, we’ll show you the proper method for setting up a FAR return.

The Microsoft MACRO Assembler                    07-17-84    PAGE    1-1  
 UNIT 9 -- PROGRAM 2 -- CALLING A FAR PROCEDURE

```

1          TITLE UNIT 9 -- PROGRAM 2 -- CALLING A
2          FAR PROCEDURE
3          ;
4          0000      50 [      PROG_STACK SEGMENT STACK
5          0000      0000      DW      80 DUP (0)
6          ]
7
8          00A0      TOP_OF_STACK LABEL WORD
9          00A0      PROG_STACK ENDS
10         ;
11         0000      FAR_CODE SEGMENT
12         0000      ASSUME CS:FAR_CODE
13         0000      COUNT_DOWN PROC FAR
14         0000 B9 BBBB      MOV      CX,0BBBBH
15         0003 E2 FE      AGAIN: LOOP AGAIN
16         0005 CB      RET
17         0006      COUNT_DOWN ENDP
18         0006      FAR_CODE ENDS
19         ;
20         0000      PROG_CODE SEGMENT
21         0000      ASSUME CS:PROG_CODE,SS:PROG_STA
22         0000 B8 ---- R      CK      START: MOV      AX,PROG_STACK
23         0003 3E D0      MOV      SS,AX
24         0005 BC 00A0 R      MOV      SP,OFFSET TOP_OF_STACK
25         0008 9A 0000 ---- R      CALL     COUNT_DOWN
26         000D CC      INT      3
27         000E      PROG_CODE ENDS
28         000E      END      START

```

The Microsoft MACRO Assembler                    07-17-84    PAGE    Symbols  
 -1  
 UNIT 9 -- PROGRAM 2 -- CALLING A FAR PROCEDURE

Segments and groups:

Name	Size	align	combine	class
FAR_CODE . . . . .	0006	PARA	NONE	
PROG_CODE. . . . .	000E	PARA	NONE	
PROG_STACK . . . . .	00A0	PARA	STACK	

Symbols:

Name	Type	Value	Attr
AGAIN. . . . .	L NEAR	0003	FAR_CODE
COUNT_DOWN . . . . .	F PROC	0000	FAR_CODE      Length
START. . . . .	L NEAR	0000	PROG_CODE
TOP_OF_STACK . . . . .	L WORD	00A0	PROG_STACK

Warning Severe  
 Errors Errors  
 0        0

**Figure 9-2**  
 Assembler listing of a program that uses  
 a FAR procedure.

```

TITLE UNIT 9 -- PROGRAM 3 -- PROPER EXE PROGRAM STRUCTURE
;
PROG_STACK SEGMENT STACK
    DW    80 DUP (0)        ;Set up stack area
TOP_OF_STACK LABEL WORD    ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,SS:PROG_STACK
START PROC FAR
    MOV   AX,PROG_STACK    ;Never load a segment register direct
    MOV   SS,AX            ;Use an intermediate register
    MOV   SP,OFFSET TOP_OF_STACK ;Point to the top of stack
    PUSH  DS              ;Save segment for return value
    SUB   AX,AX            ;Zero the AX register
    PUSH  AX              ;Save offset of zero for far return
    ;
    ;Program code area
    ;
    RET                    ;Exit the program
START ENDP
PROG_CODE ENDS
END    START

```

**Figure 9-3**

EXE program structure for a proper program termination.

Figure 9-3 shows the basic structure for a “proper” EXE-type program. We have left the data segment out of the program to keep it simple. After the Stack Segment and Stack Pointer registers are loaded, the current contents of the Data Segment register and an offset address value of zero are pushed into the stack. Recall that these values are used as the program exit address. Just make sure that when you execute the FAR return at the end of the program, that these values are the next words to be popped from the stack. That is, whenever you push an address or data into the stack, make sure you pop it back out of the stack before you terminate the program.

## Nested and In-Line Procedures

Every procedure is essentially a subroutine. From this generalization, it follows that you can treat each procedure as a subroutine. Therefore, you should assume that every procedure can be accessed through one or more labels, and that every procedure will contain one or more return instructions that have the same type (NEAR or FAR) as the procedure.

Like subroutines, procedures are normally accessed through a call instruction. In addition, other procedures can be called through a procedure. Thus, procedures can be nested. The only limit to the level of nesting is the size of the stack.

In addition to the call instruction, a procedure can be accessed through a jump instruction. Naturally, a jump is a one-way operation. The return address is not saved in the stack. We suggest you not use a jump to access a procedure.

One other method for accessing a procedure is through an “in-line” process. All this means is that the procedure is part of a consecutive string of instructions. Rather than branch to the procedure, the procedure is part of the instruction sequence. Figure 9-4 shows a portion of a program that contains a simple in-line procedure. Rather than serve as a subroutine that is called, this procedure is a process that you might have pulled from a library of often used processes. Of course, this process could also contain a number of conditional instructions that would allow it to be used as a subroutine to support another area of the program. The point is, that you don’t always have to branch to a procedure.

```
      .  
      .  
      .  
      MOV     AX, DATA1  
      MOV     BX, DATA2  
AVERAGE PROC NEAR  
      MOV     DX, AX  
      ADD     DX, BX  
      RCR     DX, 1  
AVERAGE ENDP  
      MOV     DX, DATA3  
      .  
      .  
      .
```

**Figure 9-4**

Example of an in-line procedure.

## Self-Review Questions

1. Procedures have a rigid \_\_\_\_\_.
2. The beginning of a procedure is identified by the directive \_\_\_\_\_.
3. The end of a procedure is identified by the directive \_\_\_\_\_.
4. Every procedure should contain at least one \_\_\_\_\_ instruction.
5. A \_\_\_\_\_ (type) procedure is used to define the main code section of an EXE program.
6. The procedure return instruction type is determined by the procedure \_\_\_\_\_.
7. The EXE-type program termination address is determined by pushing the original contents of the \_\_\_\_\_ segment register and the address offset \_\_\_\_\_ into the stack.
8. The best way to access a procedure is with a \_\_\_\_\_ instruction.
9. Procedures can be nested any number of times, as long as there is room on the stack for all of the return addresses. \_\_\_\_\_

True/False

NOTE: The Self-Review Answers are located at the end of this unit, beginning on Page 9-87.



## THE GROUP DIRECTIVE

The **GROUP** directive allows you to gather any number of program segments into one identifiable collection. This has the advantage that you can then address any location within that group using a single base address. A side benefit is that your program will only need one **ASSUME** statement, since all address references are made from a single base value. There is, however, one restriction. The **group size** cannot exceed 64K bytes of memory.

The assembler does not check the group size to determine if it is too large, that job is handled by the linker. This is only natural since you may be linking one or more external files with the main program, and they may share the same group. If the group size exceeds 64K bytes, the linker will react in one of several ways. Early versions will appear to pause a minute or so during the linking process, while they try to link the files. IBM's version 1.10 will then complete the operation and create an EXE file. However, any group data that exceeds 64K bytes is lost — the program will not run properly. Zenith's version 1.10 will do the same thing, only their linker will print the message:

**Fatal Error:  
Out of space on run file**

This tells you that the EXE file is not valid. With either version, you wind up with a bad EXE file on your disk. For link version 2.00 and later, the linker will not generate an EXE file if the group size exceeds 64K bytes. Zenith's linker will also tell you there was a problem by printing the message:

**Out of space on run file**

## Structure

A GROUP directive statement is structured in the following manner:

```
<group-name> GROUP <[seg-name],[...]>
```

where:

```
<group-name>
```

is a unique name that you have assigned to the group,

```
GROUP
```

is the assembler directive, and

```
<[seg-name],[...]>
```

refers to the names of the segments being combined in the group. The arrangement of the segment names in the directive does not necessarily control the order of the segments when they are linked. This is a function of the linker; and as you learned earlier, there are differences between linkers.

## Program Uses

A GROUP directive is generally used to combine the stack and data segments of a program. However, it can be used to combine all of the segments. If you do this, you in effect create a form of COM-type program. All of the individual segments appear as one (all use the same segment base address), yet the program has the characteristics of an EXE-type program. Naturally, you aren't trying to create a COM-type program with the GROUP directive. You are using the directive to reduce the complexity of addressing many different segments.

Figure 9-5 shows a program where the stack and data segments are combined in a group and the code segment is separate. The group is called DATA\_ GROUP. Notice that the directive is at the top of the program. It can be placed at any point in the program listing as long

```

TITLE UNIT 9 -- PROGRAM 5 -- GROUPING SEGMENTS
;
DATA_GROUP GROUP PROG_STACK,PROG_DATA,SOURCE_DATA,DEST_DATA
;
PROG_STACK SEGMENT STACK
    DW      80 DUP (0)      ;Set up stack area
TOP_OF_STACK LABEL WORD    ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_DATA SEGMENT PUBLIC
DATA DB      16 DUP (0AAH)
PROG_DATA ENDS
;
SOURCE_DATA SEGMENT PUBLIC
SOURCE DB      'THIS IS A SOURCE CHARACTER STRING%'
SOURCE_DATA ENDS
;
DEST_DATA SEGMENT PUBLIC
DEST DB      35 DUP (0)
DEST_DATA ENDS
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE,DS:DATA_GROUP,ES:DATA_GROUP,SS:DATA_GROUP
START PROC FAR
    MOV     AX,DATA_GROUP    ;Use the base address of the GROUP to
    MOV     SS,AX           ;load the Stack Segment register
    MOV     SP,OFFSET DATA_GROUP:TOP_OF_STACK ;Point to stack top
    PUSH    DS              ;Save segment for far return value
    SUB     AX,AX           ;Zero the AX register
    PUSH    AX              ;Save offset of zero for far return
    MOV     AX,DATA_GROUP   ;Again use the GROUP base address
    MOV     DS,AX           ;to load the remaining
    MOV     ES,AX           ;segment base addresses
    ;
    ;Program code area
    ;
    RET                    ;Exit the program
START ENDP
PROG_CODE ENDS
END START

```

**Figure 9-5**  
Program using the GROUP directive.

as it is outside of any SEGMENT/ENDS directive pair. However, the GROUP directive must **precede** any SEGMENT directive that is part of the group or any program reference to the group. Finally, a program can contain more than one group.

Following the GROUP directive are the stack and three data segments that form the group. When a segment is made part of a group, it must be a PUBLIC (a stack is considered PUBLIC) segment. Any attempt to “group” a PRIVATE segment will generate an assembly error.

The ASSUME directive reflects the program segment group. While code is assumed to reside in the PROG\_CODE segment, the stack and program data are assumed to reside in the group DATA\_GROUP. When a group of segments is formed, all address references are made from the base address of the group.

This concept is illustrated in three of the program instructions. The first and seventh instructions move the base address of the group into the AX register. That value is then stored in all of the segment registers except the Code Segment. The third instruction in the program loads the Stack Pointer register with the offset to the end of the stack. A special “group override” operator tells the assembler that the base reference for the offset is the group name. Had you used the common instruction

```
MOV SP, OFFSET TOP_OF_STACK
```

the offset value would have been calculated from the beginning of the stack segment, and that may not be the beginning of the group. Therefore, you **must** use a group override operator when you reference an offset value within a group. That operator is the group name followed by a colon. Here are a few more examples of where the group override operator should be used. The symbol AVERAGE represents an instruction label, the symbol DATA represents a byte of data, and all code and data reside within the group PROG\_GROUP.

```
DW  PROG_GROUP: AVERAGE
DD  PROG_GROUP: AVERAGE
MOV DI, OFFSET PROG_GROUP: DATA
MOV BX, OFFSET PROG_GROUP: AVERAGE
```

When you use a DW directive to load the offset address of a label, the OFFSET operator is implied by the directive. Thus, when you use the group override to obtain the correct offset, you don't have to include the operator OFFSET. When you use the group override with the DD directive, the group base address and the offset from that address is loaded into memory. In either case, if you forget to use the group override, the assembler will calculate the offsets from the segment base address rather than the group base address. The next two instructions determine the offset address value from the group base address. Again, if you forget the group override, the offset will be calculated using the segment base address.

## Self-Review Questions

10. The purpose of the GROUP directive is: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

11. The maximum size of a group is \_\_\_\_\_ bytes.

12. Group size is checked by the \_\_\_\_\_.

13. Your program contains the following segments:

```
PROG_STACK  
PROG_DATA  
PROG_CODE
```

Write a GROUP directive statement that will combine all of the program segments into one group called PGROUP.

\_\_\_\_\_

14. Write the ASSUME directive statement for the program in question 13. The Extra Segment register will not be used by the program.

\_\_\_\_\_

15. Write the instruction that will load the Stack Pointer register for the program in question 13. The end of the stack is identified by the LABEL directive:

```
TOP_OF_STACK LABEL WORD
```

\_\_\_\_\_

## CONDITIONAL DIRECTIVES

Conditional directives are assembler instructions that specify whether a section of a program should be assembled or not. This allows you to design blocks of code for different program situations which are then assembled or not assembled depending on certain conditions. For instance, you could set up a test to determine if the program must run on an IBM or a Zenith microcomputer. If it is to be an IBM, only the code for IBM operation will be assembled; the Zenith code will be ignored. By controlling which code is assembled, you can write program source code to cover every possible situation, yet keep the size of the operating program as small as possible.

There are ten conditional directive types. However, all of them follow the same basic format. Therefore, we will begin their description by looking at the general structure of an IF-ENDIF directive.

### Structure

A conditional directive identifies a block of code or data by enclosing the block within a directive pair much like the PROC-ENDP directive pair. In the case of the conditional directive, its structure takes the form

```
IFxxxx  [argument, expression, or symbol]
.
.
.
ENDIF
```

where **IFxxxx** is one of ten conditional “IF” variations that identify the beginning of the conditional block, and the conditional test. The **[argument, expression, or symbol]** establishes the parameter that is tested. **ENDIF** identifies the end of the conditional block. If the condition is met (tests true), the code or data within the directive pair is assembled. If the condition is not met, the code or data is not assembled.

Any [argument, expression, or symbol] used in a conditional directive must be known on pass one of the assembly operation. Recall that MACRO-86 is a two-pass assembler. During pass one, it builds a symbol table and calculates how much code will be generated. It does not generate object code. On pass two, the assembler uses the values defined in pass one to generate the object code. Definitions of references are checked against the pass one value, which is in the symbol table. Therefore, if the [argument, expression, or symbol] cannot be determined during pass one, because of a forward reference, the assembler will not know if the conditional code or data should be assembled. The assembler will know during pass two, but by that time, it's too late, and an assembly error is generated.

Conditional directives can be nested. However, each IFxxxx must have a matching ENDIF to terminate the condition. Following is a general example of how two conditional directives could be nested within a third.

```
IFxxxx [argument, expression, or symbol]
.
.
    IFxxxx [argument, expression, or symbol]
    .
    .
        IFxxxx [argument, expression, or symbol]
        .
        .
            ENDIF
        .
        .
    ENDIF
.
.
ENDIF
```

Keep in mind that, before a nested conditional directive can be accessed, the higher level condition must test true. In our example, the third conditional directive will only be tested if the first two test true. If the second tests false, its code or data will be ignored, and with it, the third. Conditional directives can be nested up to 255 levels.

## IF Variations

There are ten different forms of the IFxxxx-ENDIF directive. Each will be described in this section.

**IF [expression]** If the expression evaluates to a **nonzero**, the code or data within the conditional block will be assembled. The expression must be an absolute value (constant). Following are acceptable examples of specifying an absolute value:

```
COUNT EQU 1

    IF COUNT
    .
    ;CODE OR DATA
    .
    ENDIF

    IF COUNT + 0FFFFH
    .
    ;CODE OR DATA
    .
    ENDIF

    IF 5
    .
    ;CODE OR DATA
    .
    ENDIF
```

With COUNT equated to one, the first conditional will test true. The second conditional is a little more tricky. It shows that you can arithmetically modify the equated value with a constant. However, the constant cannot exceed hexadecimal FFFF. The result of the arithmetic operation can exceed hexadecimal FFFF, but then only the least significant 16 bits will be evaluated by the conditional directive. In the second conditional, the absolute value is hexadecimal 10000. Since only the 16 least significant bits are evaluated, and they equal zero, the test is false. The block of code or data will not assemble. The last conditional used the constant 5 as the expression. Since it is not zero, the condition will test true.



```
TITLE UNIT 9 -- PROGRAM 6 -- IF-ENDIF
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE
;
IBM    EQU    1        ;1 = True,
ZENITH EQU    0        ;0 = False
;
START  PROC    FAR
;
        IF     IBM
        CALL   DISPLAY1
        ENDIF
;
        IF     ZENITH
        CALL   DISPLAY2
        ENDIF
;
        RET                ;Exit program
;
        IF     IBM
DISPLAY1:
        NOP
        RET
        ENDIF
;
        IF     ZENITH
DISPLAY2:
        NOP
        RET
        ENDIF
;
START  ENDP
PROG_CODE ENDS
        END    START
```

**Figure 9-6**

Program using the condition IF-ENDIF directive pair.

Figure 9-6 shows how the IF conditional could be used in a program. The program doesn't actually do anything. It simply illustrates a process. The symbol IBM is equated to one and the symbol ZENITH is equated to zero. These values will determine which conditional blocks of code will be assembled.

```

1          TITLE UNIT 9 -- PROGRAM 6 -- IF-ENDIF
2          ;
3          0000          PROG_CODE SEGMENT
4                          ASSUME CS:PROG_CODE
5          ;
6          = 0001          IBM      EQU      1          ;1 = True,
7          = 0000          ZENITH   EQU      0          ;0 = False
8          ;
9          0000          START  PROC  FAR
10         ;
11         ;              IF      IBM
12         0000 EB 0004 R    CALL   DISPLAY1
13         ;              ENDIF
14         ;
15         ;              ENDIF
16         ;
17         0003 CB          RET          ;Exit program
18         ;
19         ;              IF      IBM
20         0004          DISPLAY1:
21         0004 90          NOP
22         0005 CB          RET
23         ;              ENDIF
24         ;
25         ;              ENDIF
26         ;
27         0006          START  ENDP
28         0006          PROG_CODE ENDS
29         ;              END      START

```

**Figure 9-7**

Assembler listing of the program in Figure 9-6.

Now look at Figure 9-7. This is the assembler listing of the program in Figure 9-6. The two IF IBM conditional directives tested true, while the two IF ZENITH conditionals tested false. As a result, only the code within the two IF IBM conditional blocks was assembled. The IF and ENDIF directives played no part in the actual program.

Notice that only the ENDIF directives of the unassembled conditional blocks remain in the listing. These are supplied by the assembler to show where the original unassembled blocks were located. Again, they play no part in the assembled program.

By now, it should be apparent that the conditional directive is very useful when you write a program that must be easily modified to run under different conditions. Had we used conditional directives in our earlier display program, it would have been a simple matter of changing two equate statements to accommodate either a Zenith or IBM system. The remaining conditional directives expand the possibilities.

**IFE [expression]** If the expression evaluates to zero, the code or data within the conditional block are assembled. This is just the opposite of the IF [expression] directive.

**IF1** This is called a “pass one conditional.” There is no expression. Rather, this conditional is enabled during pass one of the assembler. Any statements within the conditional block are processed at that time. For instance, you could place an INCLUDE directive statement within the conditional block. Then, during pass one, any included files would be read into the program. However, the included data cannot contain any code that needs to be assembled. The IF1 directive would prevent assembly during pass two of the assembly process.

Using an INCLUDE directive within a pass one conditional can be very handy for transferring a library of record or structure templates into the program. The template is read during pass one. Then the record or structure data is processed by the assembler during pass two. This can speed up the assembly operation because the included file is not opened and reread during pass two. That operation isn’t necessary because there is nothing to assemble.

If you look at the assembler listing of a program with a pass one conditional, you will only find the ENDIF directive. This is because the conditional tested false during pass two of the assembly operation. As before, a false conditional is ignored by the assembler, and only the ENDIF directive remains to show its original location in the program.

**IF2** This is called a “pass two conditional.” It performs the same function as the pass one conditional, except that it is enabled during pass two of the assembler. One handy use for this conditional is that it lets you display a message during pass two of the assembler. When you assemble a long program, it can be reassuring to see a message that tells you the assembler has completed the first pass.

However, to display that message you need a special directive — **%OUT**. This tells the assembler to display the characters following the directive. A typical conditional message could be written:

```
IF2
%OUT PASS 1 OF THE ASSEMBLY OPERATION IS COMPLETE
ENDIF
```

The assembler will display the message

```
PASS 1 OF THE ASSEMBLY OPERATION IS COMPLETE
```

when it begins pass two of the assembly operation. You can enter a message line up to the length of the program line. To enter more than one line, begin each line with the directive **%OUT**. Naturally, you can also use the **%OUT** directive with the pass one conditional directive to display a message at the beginning of the assembly operation. If you use the **%OUT** directive outside of a conditional directive, its message will be displayed twice — once during pass one and once during pass two.

**IFDEF [symbol]** If the symbol **has** been defined in the program, or **has** been declared External, the code or data in the conditional block are assembled.

**IFNDEF [symbol]** If the symbol **has not** been defined in the program, and **has not** been declared External, the code or data in the conditional block are assembled.

Both of these “defined symbol” conditionals are similar to the **IF** and **IFE** conditionals. However, instead of testing an expression for a zero or nonzero condition, they look for the existence or nonexistence of a symbol.

**IFB** [**<argument>**] If the argument is **blank** (none given), or **null** (two angle brackets with nothing in between), the code or data in the conditional block are assembled. Angle brackets must be placed around the argument.

**IFNB** [**<argument>**] If the argument is **not blank**, the code or data in the conditional block are assembled. Again, the angle brackets must be placed around the argument.

Normally, the conditionals IFB and IFNB are used within macro code blocks. The argument following IFB or IFNB is typically a “dummy symbol.” When the macro is called, the dummy is replaced by a “parameter” passed by the macro call. If the macro does not specify a parameter, the argument is null. Dummy symbols, parameters, and macros will be described in the next section.

**IFIDN** [**<argument1>**,**<argument2>**] If the string in argument one **equals** the string in argument two, the code or data in the conditional block are assembled. The arguments are separated by a comma. Angle brackets must be placed around each argument.

**IFDIF** [**<argument1>**,**<argument2>**] If the string in argument one is **different** from the string in argument two, the code or data in the conditional block are assembled. Again, each argument is enclosed by angle brackets and both are separated by a comma.

Like the “if blank or not blank” conditionals, IFIDN and IFDIF are normally used inside a macro code block. In this case, each argument is usually represented by a dummy symbol. When the macro is called, both arguments are replaced by parameters passed by the macro. Depending on the conditional directive and the parameters passed, the conditional block will be assembled or ignored.

## The ELSE Directive

Each of the conditional IF directives has a complement. For instance, the complement to the “if blank” conditional is “if not blank.” Quite often, when you are testing a condition, you want some action taken if the condition is not met. One way to accomplish this is to write the desired conditional block, and then write a complement conditional block. The **ELSE** directive gives you another option. It allows you to combine a block of alternate code or data with the original code or data. Then if the IF condition tests true, the related code or data are assembled. On the other hand, if the condition tests false, the code or data related to the ELSE directive are assembled.

Following is an example of how a conditional with an ELSE directive could be arranged.

```
IFxxxx [argument, expression, or symbol]
.
.
ELSE
.
.
ENDIF
```

During assembly, the assembler evaluates the IFxxxx directive. If it tests true, the code or data between the IFxxxx and ELSE are assembled. If it tests false, the code or data between the ELSE and ENDIF are assembled. Note that only one ELSE directive can be used between every IFxxxx-ENDIF directive pair.

Figure 9-8 shows the program from Figure 9-6 using the ELSE directive to handle alternate mode assembly. There are only two conditional blocks in this program. The first one states that if the expression IBM is a nonzero, assemble the instruction that calls DISPLAY1. If IBM

```
TITLE UNIT 9 -- PROGRAM 7 -- ELSE
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE
;
IBM    EQU    1        ;1 = True,
ZENITH EQU    0        ;0 = False
;
START  PROC    FAR
;
    IF     IBM
CALL    DISPLAY1
    ELSE
CALL    DISPLAY2
    ENDIF
;
    RET                    ;Exit program
;
    IF     ZENITH
DISPLAY2:
    NOP
    RET
    ELSE
DISPLAY1:
    NOP
    RET
    ENDIF
;
START  ENDP
PROG_CODE ENDS
END    START
```

**Figure 9-8**

Program to illustrate the ELSE directive.

equates to zero, assemble the instruction that calls DISPLAY2. The second block states that if the expression ZENITH is a nonzero, assemble the next two instructions. If ZENITH equates to zero, assemble the two instructions following the directive ELSE.

```

1          TITLE UNIT 9 -- PROGRAM 7 -- ELSE
2          ;
3          0000          PROG_CODE SEGMENT
4          ;              ASSUME CS:PROG_CODE
5          ;
6          = 0001          IBM      EQU    1      ;1 = True,
7          = 0000          ZENITH  EQU    0      ;0 = False
8          ;
9          0000          START  PROC  FAR
10         ;
11         ;              IF      IBM
12         0000 E8 0004 R  CALL    DISPLAY1
13         ;              ENDIF
14         ;
15         0003 CB          RET          ;Exit program
16         ;
17         ;              ELSE
18         0004          DISPLAY1:
19         0004 90          NOP
20         0005 CB          RET
21         ;              ENDIF
22         ;
23         0006          START  ENDP
24         0006          PROG_CODE ENDS
25         ;              END      START

```

**Figure 9-9**

Assembler listing of the program in Figure 9-8.

Now look at the assembler listing of the program in Figure 9-9. Because IBM equated to a nonzero, the instruction calling DISPLAY1 was assembled. The “else” portion of the conditional block was not assembled or even listed. In the second conditional block, ZENITH equated to zero. Therefore, the “else” portion was assembled and listed. The IF ZENITH portion was not listed.

## Self-Review Questions

16. Conditional directives are: \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_
17. The beginning of a conditional directive is identified by the \_\_\_\_\_ directive.



18. The end of a conditional directive is identified by the directive \_\_\_\_\_.
19. Conditional directives can be nested. \_\_\_\_\_  
True/False
20. An \_\_\_\_\_ directive will evaluate true if its expression is a nonzero.
21. An \_\_\_\_\_ directive will evaluate true if its expression is a zero.
22. An \_\_\_\_\_ directive is enabled during pass one of the assembler.
23. An \_\_\_\_\_ directive is enabled during pass two of the assembler.
24. An \_\_\_\_\_ directive will evaluate true if its symbol has been defined in the program.
25. An \_\_\_\_\_ directive will evaluate true if its symbol has not been declared External.
26. An \_\_\_\_\_ directive will evaluate true if it has a null argument.
27. An \_\_\_\_\_ directive will evaluate true if it contains an argument.
28. An \_\_\_\_\_ directive will evaluate true if the string in argument one equals the string in argument two.
29. An \_\_\_\_\_ directive will evaluate true if the string in argument one does not equal the string in argument two.
30. The \_\_\_\_\_ directive can be used to display a message during program assembly.
31. The \_\_\_\_\_ directive allows an alternate conditional block to be assembled if the specified condition is not met.

## MACRO DIRECTIVES AND OPERATORS

A macro is a programming tool that allows you to generate or repeat a sequence of code or data anywhere within a program. The basic code or data is arranged in template form, much like a Structure. And like a Structure, the macro template is duplicated and expanded with a set of parameters that are provided by the program “macro call.” Unlike Structures, however, macros can be nested so that one may call another, and that may call another, and so on. Before a macro can be called, or initialized, you must first define the macro and establish what parameters are required.

### Macro Definition

The macro is a form of “blueprint.” It shows the assembler how to arrange the code or data when the macro is called. Following is the general format for a macro template.

```
<name> MACRO <dummy list>
    .
    ;code or data
    .
    ENDM
```

The beginning of the template is identified by a macro directive statement. The <name> is a unique name assigned to the macro. This name will be used to identify the macro when it is called from the program. The MACRO directive tells the assembler that the following code or data is part of a macro template. The <dummy list> is a list of what we call “dummy parameters.” They identify undefined parameters within the macro template. Each dummy parameter within the dummy list is either separated by a comma or enclosed by angle brackets, depending on type of macro. Each dummy is treated as a symbol within the macro template. Therefore you must observe all of the rules for writing a symbol when you specify a dummy.

When the macro is called, the dummy parameters are replaced by real parameters. These can take the form of text, data, program symbols, or register names. In some instances, the dummy parameter may be

replaced by a constant that determines the number of times a macro is duplicated. The number of dummies used in a template is limited only to the length of the directive statement line.

Following the macro directive statement is the body of the template. It contains the code or data that will be assembled into the program when the macro is called. It can also contain special comments, other assembler directives and operators, any of the macro operators, and additional macro directives.

The end of the macro template is identified by the directive **ENDM** (end macro). Every macro directive statement must have a matching ENDM directive. A missing ENDM will generate an assembly error.

Following is an example of how a macro template could be written.

```
STORE    MACRO    XX,YY,ZZ
          MOV     AX,XX
          ADD     AX,YY
          MOV     ZZ,AX
          ENDM
```

The template name is STORE, the macro directive is MACRO, and there are three dummy parameters in the “dummy list.” These match the dummy parameters used in the body of the template. When this template is called, the three dummy parameters will be replaced by defined parameters. The next section will describe the macro call and use this template as an example.

## Calling A Macro

There are two distinct steps in using a macro. In the first step, the macro template is constructed. The second step is the macro call. This occurs when the assembler encounters the macro name in the body of the program. The assembler then substitutes the macro template with its defined parameters in place of the macro call instruction. The assembler performs the macro substitution before it assembles the program code.

A macro call takes the form

**<name>    [<parameter>,...]**

where **<name>** is the name of the macro template, and **[<parameter>,...]** is the list of defined parameters that are passed to the template. Remember, a parameter replaces a dummy parameter on a one-for-one basis. The order of replacement is determined by the dummy list and the parameter list. For example, if the macro definition statement is

```
HOST    MACRO    DUMMY1, DUMMY2, DUMMY3
```

and the macro call is

```
HOST    AX, DATA, 5
```

then register name AX will replace every reference to DUMMY1 in the template body; symbol DATA will replace every reference to DUMMY2 in the template; and constant 5 will replace every reference to DUMMY3 in the template.

The number of parameters is limited only to the length of the macro call line. If you pass more parameters than there are dummy parameters in the template, the extras will be ignored by the assembler. On the other hand, if you pass fewer parameters than there are dummy parameters, the undefined dummy parameters will become nulls (blanks). If you enter more than one parameter, they must be separated by commas, spaces, or tabs (we will use commas in our examples).

In the instance where an individual parameter contains multiple items that are separated by commas, you can identify the parameter by placing angle brackets around the items. For example, the assembler would assume that the macro call

```
TOTAL    3, 5, 9, 4, 6
```

contains five individual parameters, while the macro call

```
TOTAL    <3, 5, 9>, 4, 6
```

contains three parameters. Although the angle brackets identify the beginning and end of a parameter, you still have to separate it from the other parameters with a comma, space, or tab.

Before you can use a macro call, you must define the macro. We'll use the earlier template:

```
STORE    MACRO    XX, YY, ZZ
          MOV     AX, XX
          ADD     AX, YY
          MOV     ZZ, AX
          ENDM
```

If you then enter the macro call statement

```
STORE    TEMP, 32, DATA
```

the assembler will generate the program instructions

```
MOV     AX, TEMP
ADD     AX, 32
MOV     DATA, AX
```

using the macro template and the parameters passed by the macro call.

## Other Macro Directives

There are three other macro directives that can be used to support a MACRO directive statement: REPT, IRP, and IRPC. They are actually different forms of a “repeat macro” directive. In addition to supporting the MACRO directive, these repeat macros can be used as individual macros. Rather than being called from a program, the “stand alone” repeat macro is immediately assembled when it is encountered by the assembler. Let's begin with the **REPT** (repeat) macro.

The REPT (repeat) macro directive is used to repeat its macro template the number of times specified in the macro directive statement. A REPT macro is used to generate a block of data. The macro can be nested within a MACRO, or it can stand alone. As a nested macro, it usually takes the form:

```

<name> MACRO    <dummy list>
.
.
REPT           <expression>
.
.
ENDM           ;For repeat macro
ENDM           ;For main macro

```

Notice that just like the MACRO directive, each REPT macro directive must have a matching ENDM to terminate the macro template. When the REPT macro is not nested, it will take the form:

```

REPT           <expression>
.
.
.
ENDM

```

The REPT macro does not have a name. It doesn't need a name because it is not called. Remember, it is assembled as soon as it is encountered by the assembler. Following is an example of a REPT macro.

```

DATA EQU      0
.
.
REPT          7
DB            DATA
ENDM

```

At some point in the program, prior to the REPT macro, the symbol DATA is equated to zero. The REPT directive is followed by the expression seven. This indicates that the following template will be repeated seven times. When the program is assembled, the repeat macro will generate seven bytes of code, each containing the value equated to DATA, zero.

While this is a convenient way to generate common data bytes, the assembler directive DUP is easier. The real power of the repeat directive is in its ability to modify the data as it is being generated. Here is an example.

```
DATA EQU      0
.
.
REPT      7
DATA = DATA + 1
DB      DATA
ENDM
```

Again DATA is equated to zero and the REPT directive contains the expression seven. Following the directive is a “data statement.” The statement says, in effect, for each use of the constant DATA add one to the previous value. The equals sign has the same meaning as the assembler EQU directive. In fact, you can substitute one for the other in a program. However, we suggest that you use the EQU directive to equate a value in your program, and the equal sign to equate a value in a REPT directive.

```

1          TITLE UNIT 9 -- PROGRAM 8 -- REPEAT
2          .LALL
3          ;
4          0000          PROG_CODE SEGMENT
5                          ASSUME CS:PROG_CODE
6          ;
7          0000          START PROC FAR
8          ;
9          = 0000          DATA EQU 0
10         ;
11         REPT 7
12         DATA = DATA + 1
13         DB DATA
14         ENDM
15         + DATA = DATA + 1
16         0000 01        + DB DATA
17         = 0002        + DATA = DATA + 1
18         0001 02        + DB DATA
19         = 0003        + DATA = DATA + 1
20         0002 03        + DB DATA
21         = 0004        + DATA = DATA + 1
22         0003 04        + DB DATA
23         = 0005        + DATA = DATA + 1
24         0004 05        + DB DATA
25         = 0006        + DATA = DATA + 1
26         0005 06        + DB DATA
27         = 0007        + DATA = DATA + 1
28         0006 07        + DB DATA
29         ;
30         0007 CB        RET
31         ;
32         0008          START ENDP
33         0008          PROG_CODE ENDS
34                                     END START

```

Segments and groups:

Name	Size	align	combine	class
PROG_CODE. . . . .	0008	PARA	NONE	

Symbols:

Name	Type	Value	Attr	Length
DATA . . . . .	Number	0007		
START. . . . .	F PROC	0000	PROG_CODE	

=0008

Warning Severe  
Errors Errors  
0 0

**Figure 9-10**  
Assembler listing of a program showing the REPT macro directive.



Figure 9-10 shows the assembler listing for the preceding REPT directive. Notice that for each operation of the REPT directive, the symbol DATA is redefined according to the data statement following the directive. Thus, for the first repeat, DATA is equal to one; and for the last repeat, DATA is equal to seven. This is supported in the “Symbols” table, at the bottom of the figure, where DATA has a “Type” “Number” and a “Value” of seven. The “Symbols” table always shows the last value assigned to a symbol.

Each listing line that is created by a MACRO directive, or any of the repeat macro directives contains a “plus” symbol. The symbol precedes the text portion of the listing. This is provided to help you locate macros in your program.

At the beginning of the program, we added the assembler directive .LALL. This tells the assembler to list all of the macro data. Without the “list all” directive, the program portion of the assembler listing would have looked like the listing in Figure 9-11. There, only the data that is actually assembled is listed. This default condition leaves out all of the macro information that is not part of the program. If you need to look at all of the macro information, use the .LALL directive. If you don’t need the detail, leave out the directive.

```

The Microsoft MACRO Assembler          07-30-84   PAGE   1-1
UNIT 9 -- PROGRAM 8 -- REPEAT

1                                     TITLE UNIT 9 -- PROGRAM 8 -- REPEAT
2                                     ;
3      0000                            PROG_CODE SEGMENT
4                                     ASSUME CS:PROG_CODE
5                                     ;
6      0000                            START  PROC   FAR
7                                     ;
8      = 0000                          DATA   EQU    0
9                                     ;
10                                     REPT    7
11      DATA = DATA + 1
12                                     DB     DATA
13                                     ENDM
14      0000 01                          +     DB     DATA
15      0001 02                          +     DB     DATA
16      0002 03                          +     DB     DATA
17      0003 04                          +     DB     DATA
18      0004 05                          +     DB     DATA
19      0005 06                          +     DB     DATA
20      0006 07                          +     DB     DATA
21                                     ;
22      0007 CB                          RET
23                                     ;
24      0008                            START  ENDP
25      0008                            PROG_CODE ENDS
26                                     END     START

```

**Figure 9-11**

Assembler listing of the program from Figure 9-10 without the “list all” assembler directive.

The **IRP** (indefinite repeat) directive is a second form of macro repeat directive. Rather than repeat, or augment and repeat a constant like the REPT directive, IRP replaces a “dummy” within its template with one or more parameters. Following is the format for an IRP directive template.

```

IRP  <dummy>,[<parameters>]
      .
      .
      .
ENDM

```

The <**dummy**> is a symbol that identifies every location in the template that will receive a parameter. An IRP directive template can only contain one unique dummy symbol, although that symbol can be repeated as many times as you wish. The [<**parameters**>] are all of the parameters that will be passed to the template dummy. Parameters can be any legal symbol, string, numeric, or character constant. They must be enclosed by a pair of angle brackets. The number of parameters determines the number of times the template is repeated. For example, the template

```

      IRP  DATA,<1,2,3>
      DB  DATA
      DB  0AH
      DB  DATA,5DH,DATA
      ENDM

```

will cause the assembler to generate the equivalent code

```

      DB  1
      DB  0AH
      DB  1,5DH,1
      DB  2
      DB  0AH
      DB  2,5DH,2
      DB  3
      DB  0AH
      DB  3,5DH,3

```

There are three parameters. Therefore, the template is repeated three times. Each time the template is repeated, each template dummy is replaced with the appropriate parameter.

As with the REPT directive, the IRP directive can be nested in a MACRO. Being nested in a MACRO allows the MACRO parameter to be passed onto the IRP parameter. Thus, the previous IRP template could have been written

```
NUMBERS  MACRO    COUNT
          IRP     DATA, <COUNT>
          DB      DATA
          DB      0AH
          DB      DATA, 5DH, DATA
          ENDM
          ENDM
```

When the macro is called with the statement

```
NUMBERS  <1,2,3>
```

the assembler will produce the equivalent code

```
DB  1
DB  0AH
DB  1, 5DH, 1
DB  2
DB  0AH
DB  2, 5DH, 2
DB  3
DB  0AH
DB  3, 5DH, 3
```

This occurred because the parameter <1,2,3> in the macro call was passed to the dummy parameter COUNT in the MACRO template NUMBERS. Since the IRP macro directive statement is considered part of the MACRO template, the parameter <1,2,3> was passed on to the IRP macro template. From there, the parameters were loaded into the dummy locations in the IRP template.

The last repeat macro directive is **IRPC** (indefinite repeat character). It is identical to the IRP macro directive with one exception, the parameters consist of a string of characters that are **not** separated by commas. Angle brackets around the parameter string are optional. Normally the brackets are left out to distinguish this macro directive from the IRP macro directive. The IRPC macro template is repeated for each character in the parameter. The macro template

```
IRPC X,123456
DB X
ENDM
```

will produce the data

```
DB 1
DB 2
DB 3
DB 4
DB 5
DB 6
```

when it is assembled. If a character other than a decimal number is used in the parameter, that character must be defined prior to the macro. For example,

```
E EQU 55
F EQU 6
G EQU 17
IRPC W,EFG
DB W
ENDM
```

will produce the data

```
DB 37H
DB 6
DB 11H
```

when it is assembled. If the three equate statements were missing, the assembler would generate an error statement each time it tried to translate one of the template parameters.

## Self-Review Questions

32. A \_\_\_\_\_ is a programming tool that allows you to generate or repeat a sequence of code or data within a program.
33. The directive \_\_\_\_\_ tells the assembler that the following code or data is part of a macro template.
34. The directive \_\_\_\_\_ identifies the end of a macro template.
35. A macro \_\_\_\_\_ is used by a macro call to identify a macro template.
36. A macro call contains one or more \_\_\_\_\_ that are passed to the called macro template.
37. An individual parameter that contains many items separated by commas is identified by enclosing the items with a pair of \_\_\_\_\_.
38. The \_\_\_\_\_ macro directive is used to repeat its macro template the number of times specified in the macro directive statement.
39. The \_\_\_\_\_ macro directive is used to replace a dummy parameter within its template with one or more parameters. The parameters are separated by commas and enclosed with angle brackets.
40. The \_\_\_\_\_ macro directive is used to replace a dummy parameter within its template with one or more parameters. The parameters consist of a string of characters not separated by commas and not enclosed with angle brackets.

## The Conditional Macro

There will be times when you don't want a macro to be fully assembled. That is, when a specific condition is met, you want macro assembly to stop. This job is handled by the macro directive **EXITM** (exit macro). **EXITM** stops macro expansion from the point where it is encountered. Normally, **EXITM** is used after a conditional (**IFxxxx**) directive, to hide the macro until the right conditions exist. For example, consider the repeat macro template:

```
COUNT EQU 200
X      EQU 0
.
.
REPT   COUNT
X = X + 1
DB     X
ENDM
```

The **REPT** directive will force the assembler to generate a block of 200 defined bytes with ascending values. Now suppose you wanted the data generation to stop after a specific number of bytes had been created. You could add a conditional **ENDM** to the template to test the data, as follows:

```
COUNT EQU 200
X      EQU 0
HALT   EQU 10
.
.
REPT   COUNT
X = X + 1
DB     X
      IFE     X - HALT
      EXITM
      ENDIF
ENDM
```

Again, the repeat count is 200 and the “dummy” X has an initial value of zero. The assembler will expand the macro template, generating ascending byte values until the value of X minus the value of the constant **HALT** equals zero (in this example, when X equals ten). At that time, the conditional directive will test true and “uncover” the **EXITM** directive. When the assembler sees the **EXITM** directive, it will stop template expansion.

If the `EXITM` directive is encountered within a “nested macro,” the assembler will stop expanding that macro and continue with the next outer level macro. That is because the `EXITM` directive only affects the expansion of the template that contains the directive.

You may have noticed in the preceding macro template, that the nested conditional template statements were offset from the macro template statements. This was done to make it easier to identify the nested template. You can do the same thing when you nest macros.

## Macro Support Directives

Two other macro directives support the macro assembler operation. Those directives are `PURGE` and `LOCAL`. The **PURGE** directive, as you might guess, is used to delete a macro template from a program. As you become more experienced in your programming, you will probably begin building a library of “include files” that contain many macros. When you include a file of macros in a program, you may not need all of the macros in that file. Rather than build a new file to accommodate the program, it’s much simpler to use the `PURGE` directive to delete any unnecessary macros. The `PURGE` directive statement takes the form

```
PURGE    [<macro-name>,...]
```

where **PURGE** is the directive and `<macro-name>` is the name of the macro template to be deleted. Multiple macro names are separated by commas.

The **LOCAL** directive is used to create unique names or labels within a macro expansion. To see why that may be necessary, consider the macro template

```
TEST MACRO      COUNT
ONE    DB      5
TWO    DW      COUNT
THREE:
      SUB    AX, AX
      ADD    AX, TWO
      CMP    AX, 20
      JNE    THREE
      ENDM
```

It defines three bytes of data, and then tests part of that data with a simple four-instruction routine. When the macro is called, the dummy COUNT is replaced with a constant. The assembler uses the two names and the label to determine the offset address for the data and the jump target. Now what do you suppose will happen if the macro is called a second time? The assembler will generate an error message for each name or label indicating that it has been redefined. That is, each symbol is pointing at two different offset address locations. The LOCAL directive eliminates the problem of multiple symbol definition within a macro; it forces the assembler to create a unique symbol for each occurrence during assembly. The names and labels created by the LOCAL directive take the form ??nnnn, and range from ??0000 to ??FFFF.

The directive takes the form

**LOCAL**    [<macro-symbol>,...]

where **LOCAL** is the directive, and <macro-symbol> identifies every symbol used within the macro. The symbols are separated by commas. The LOCAL directive statement must be positioned immediately after the macro definition statement. That also means no comments can be placed between the two directive statements.

Figure 9-12 is an assembler listing that shows how the LOCAL directive could be used. The template we just described is called three times. In the first expansion, the symbols range from ??0000 to ??0002; in the second, from ??0003 to ??0005; and in the third, from ??0006 to ??0008. The symbols for each template expansion are unique. If more than one template with symbols is called in a program, the symbols remain unique because the assembler keeps track of what symbols have been created in its symbol table.



The Microsoft MACRO Assembler  
UNIT 9 -- PROGRAM 9 -- LOCAL

08-01-84 PAGE 1-1

```

1          TITLE UNIT 9 -- PROGRAM 9 -- LOCAL
2          .LALL
3          ;
4          0000      PROG_CODE SEGMENT
5                      ASSUME CS:PROG_CODE
6          ;
7          0000      START  PROC  FAR
8          ;
9          TEST     MACRO  COUNT
10                     LOCAL  ONE,TWO,THREE
11                     ONE   DB   5
12                     TWO   DW   COUNT
13                     THREE: SUB  AX,AX
14                             ADD  AX,TWO
15                             CMP  AX,20
16                             JNE  THREE
17                     ENDM
18          ;
19          TEST     5
20          0000 05      + ??0000 DB   5
21          0001 0005      + ??0001 DW   5
22          0003 2B C0      + ??0002: SUB  AX,AX
23          0005 2E: 03 06 0001 R  +      ADD  AX,??0001
24          000A 3D 0014      +      CMP  AX,20
25          000D 75 F4      +      JNE  ??0002
26          TEST     10
27          000F 05      + ??0003 DB   5
28          0010 000A      + ??0004 DW   10
29          0012 2B C0      + ??0005: SUB  AX,AX
30          0014 2E: 03 06 0010 R  +      ADD  AX,??0004
31          0019 3D 0014      +      CMP  AX,20
32          001C 75 F4      +      JNE  ??0005
33          TEST     20
34          001E 05      + ??0006 DB   5
35          001F 0014      + ??0007 DW   20
36          0021 2B C0      + ??0008: SUB  AX,AX
37          0023 2E: 03 06 001F R  +      ADD  AX,??0007
38          0028 3D 0014      +      CMP  AX,20
39          002B 75 F4      +      JNE  ??0008
40          ;
41          002D CB          ;      RET
42          ;
43          002E          START  ENDP
44          002E          PROG_CODE ENDS
45          END          END      START

```

**Figure 9-12**

Assembler listing of a program that uses the directive LOCAL in a macro.

## Special Macro Operators

Five special macro operators give you additional capabilities for defining a macro. The macro operators are:

& <> ;; ! %

& The ampersand is used to concatenate text or symbols. During the course of expanding a macro, you may wish to control the form that a symbol, command, or portion of text takes. Previously, you could only do this by replacing a dummy with a parameter. The dummy, however, had to stand alone; it couldn't be part of "nondummy" text. For example, if you wrote the symbol

```
DATA X
```

and X was the dummy, the assembler would treat the symbol as a symbol, and not as a symbol with a dummy. The ampersand lets you identify the dummy within text. Changing the symbol to

```
DATA&X
```

tells the assembler that the symbol contains a dummy. You must place the ampersand between the text and dummy; or if you are using two dummies, between the dummies. The assembler then determines what is text or dummy during the macro expansion. As an example, consider the following macro template.

```
TEST  MACRO  X, Y, Z
TIME&X      MOV  AX, BX
X&DATA      DB   Y
X&Z         DB   5
          ENDM
```

The macro template TEST contains three dummies. When it is called by the instruction

```
TEST  A, 10, COUNT
```

the assembler expansion of the macro will look like

```
TIMEA      MOV  AX, BX
ADATA      DB   10
ACOUNT     DB   5
```

Here we have modified data and symbols. You can also modify the instruction mnemonics and operands. The following macro template is a good example.

```
TEST  MACRO  X, Y, Z
        MOV  A&X, B&X
        CMP  A&X, Y
        J&Z  STOP
```

Now if the template is expanded through the macro call

```
TEST  H, COUNT, NE
```

the assembler generates the instructions

```
MOV  AH, BH
CMP  AH, COUNT
JNE  STOP
```

In the instance where you have a nested macro that is being passed a parameter from the next outer macro, you must use two ampersands to identify the nested macro dummy. For example, consider the macro template

```
TEST  MACRO  X
        IRP  Z, <1, 2, 3>
        X&&Z  DB  Z
        ENDM
    ENDM
```

The dummy X is being used to pass a parameter to the nested indefinite repeat macro. When the macro call

```
TEST  DATA
```

is assembled, the following action occurs. First the parameter DATA is passed to the nested macro, leaving an intermediate macro

```
        IRP  Z, <1, 2, 3>
DATA&Z  DB  Z
        ENDM
```

Then the intermediate macro is expanded to give

```
DATA1  DB  1
DATA2  DB  2
DATA3  DB  3
```

When you write a nested macro, you must try to visualize how each macro will affect the next nested macro. In our example, the first macro passed the parameter DATA to the nested macro. When the pass was completed, the dummy and its identifying ampersand were replaced with the parameter. Remember, a parameter always replaces a dummy/ampersand pair during macro expansion. The second ampersand remained after the intermediate step to identify the IRP macro dummy. In very complex macros, where nesting is involved, you must supply as many ampersands as there are levels of nesting.

<> You learned earlier that angle brackets must be used with some macros. Angle brackets tell the macro assembler that any text, even if it includes commas, should be treated as a single literal character or term for replacement purposes. For example, the semicolon inside angle brackets <;> becomes a character, not the indicator that a comment follows. Like the ampersand, one set of angle brackets is removed each time the parameter is used in a macro. Therefore, when using nested macros, you need to supply as many sets of angle brackets around parameters as there are levels of nesting.

;; Recall that under the default listing condition, only the code and data actually assembled through a macro call is listed. If the .LALL directive is added to the program, all of the macro expansion data is supplied. The same conditions also apply to any comments within a macro template. Under default listing, the comments are not listed. If the .LALL directive is added to the program, the comments are listed.

There will be times when you wish to list all of the macro expansion data, but not the accompanying comments. To hide comments that would normally be listed after the .LALL directive, precede each comment line with **two** semicolons rather than the normal single semicolon. Keep in mind, however, that this feature will only work with the MACRO directive, or with a “repeat” directive that is nested within a MACRO directive.

! The exclamation point is used in a macro call to tell the assembler that the following character is entered literally. Thus,

```
!;
```

is equivalent to

```
<;>
```

In both instances, the semicolon is treated as a literal character.

% The percent sign is used only in a macro call to convert the expression that follows it (usually a symbol) to a number in the current radix. During macro expansion, the number derived from converting the expression is substituted for the dummy. Using the percent sign allows a macro to be called by a value. (Usually, a macro call is a call by reference to text.) The expression following the percent sign must evaluate to a (non-relocatable) constant. The following macro template and macro call will show you how the percent sign could be used, and a different method for calling a macro.

```
Y      EQU    0
.
.
TEST   MACRO  X
        REPT  X
        Y = Y + 1
        CHECK %Y
        ENDM
        ENDM
.
.
CHECK  MACRO  Z
ERR&Z DB 'ERROR &Z'
        ENDM
```

This template actually consists of two separate macro templates; one is named TEST, and the other is named CHECK. Let's look at the template named CHECK. It uses the MACRO directive, and the dummy parameter is Z. When the macro is called, the name and contents of the define byte directive will be modified through the dummy parameter Z. In both instances, the dummy parameter is identified by the macro operator ampersand.

The other template, TEST, also uses the MACRO directive. Its single dummy parameter is X. The parameter that is passed to this macro will also be passed to the dummy expression in the repeat macro nested in the template. Recall that the dummy expression following a REPT macro directive is used to specify the number of times the macro is repeated. The only instruction in the repeat macro is a macro call. It is used to call the macro template CHECK. The parameter that is passed to the template by this call is %Y. Placing the percent sign in front of the Y tells the assembler that it should use the **current value** of Y, rather than the literal character, to expand the macro CHECK. The constant Y is defined as zero. The repeat macro is set up so that each expansion of the macro will add one to the original value of Y. When the macro TEST is called by the macro call

```
TEST 3
```

the following steps occur. First, the parameter three is passed through the macro dummy parameter X to the nested repeat macro. This determines the number of times the macro is repeated. Next, one is added to the current value of Y ( $0 + 1 = 1$ ). Then, the macro CHECK is called. The parameter one is used to expand that macro. This produces the assembler data statement

```
ERR1 DB 'ERROR 1'
```

The repeat macro operation is repeated two more times, producing the assembler data statements

```
ERR2 DB 'ERROR 2'  
ERR3 DB 'ERROR 3'
```

Keep in mind that these percent sign features apply only when they are used with macro calls. Outside of a macro call, the percent sign is treated as a simple character or as part of an assembler directive, as in the case of the directive %OUT, described earlier.

## Self-Review Questions

41. The \_\_\_\_\_ macro directive is used to end a conditional macro operation.
42. The \_\_\_\_\_ macro directive is used to delete unwanted macro templates from a program.
43. The \_\_\_\_\_ macro directive is used to create unique names or labels within a macro expansion.
44. The macro operator \_\_\_\_\_ is used to concatenate text or symbols during macro expansion.
45. The macro operator angle brackets can be used to create a \_\_\_\_\_ character.
46. Two \_\_\_\_\_ are used to hide a macro comment in an assembler listing.
47. The macro operator \_\_\_\_\_ is used to identify a literal character.
48. The macro operator \_\_\_\_\_ is used in a macro call to convert the expression that follows it into a number.

## ASSEMBLER LISTING DIRECTIVES

The assembler listing directives perform two functions: program format control and program listing control. The format directives allow you to control the page layout. This includes page headings, page breaks, and the length and width of the page. The listing directives “turn on” or “turn off” the listing of all or part of the assembled file.

### Format Directives

There are three format directives: **PAGE**, **TITLE**, and **SUBTTL**. The **PAGE** directive is used to control the length and width of the assembler listing page, the way the pages are numbered, and any page breaks. The **TITLE** directive is used to specify the title of a program and the major heading of each listing page. The **SUBTTL** directive is used to specify program subsection titles and the subheading of each listing page.

#### **PAGE**

The **PAGE** (page format) directive determines the structure of a listing page. Generally it is used to set up a listing to accommodate the characteristics of a printer. It also allows you to control when the listing for one page ends and another begins. This can be very useful for structuring the program listing for easier reading. Finally, the **PAGE** directive can be used to control how the pages are numbered.

Recall from earlier program listings that every page begins with the assembler name, the program assembly date, and a page count. The page count is composed of two numbers separated by a hyphen. The number on the left is the major page number; it has always been one. The number on the right is the minor page number; it increments for every new page in the listing.



When the assembler encounters the PAGE directive with no argument, it will stop the current listing at that point and begin a new page. This is called a **page break**. The minor page number will be incremented by one, while the major page number will not change. Figure 9-13 shows how the PAGE directive can affect the listing. Notice that the first ten lines of the listing are on page 1-1, and the remaining nine lines are on page 1-2. The PAGE directive is on the first listing line of the new page.

```

The Microsoft MACRO Assembler          08-06-84   PAGE   1-1
UNIT 9 -- PROGRAM 10 -- PAGE

1          TITLE UNIT 9 -- PROGRAM 10 -- PAGE
2          ;
3          0000      PROG_CODE SEGMENT
4                      ASSUME CS:PROG_CODE
5          ;
6          0000      START  PROC  FAR
7          ;
8          ;
9          ;
10         ;

The Microsoft MACRO Assembler          08-06-84   PAGE   1-2
UNIT 9 -- PROGRAM 10 -- PAGE

11         PAGE
12         ;
13         ;
14         ;
15         0000  CB          RET
16         ;
17         0001      START  ENDP
18         0001      PROG_CODE ENDS
19         END          START

```

**Figure 9-13**  
Assembler listing using the PAGE directive.

The Microsoft MACRO Assembler  
UNIT 9 -- PROGRAM 11 -- PAGE+

08-06-84 PAGE 1-1

```

1          TITLE UNIT 9 -- PROGRAM 11 -- PAGE+
2          ;
3          0000      PROG_CODE SEGMENT
4                      ASSUME CS:PROG_CODE
5          ;
6          0000      START  PROC  FAR
7          ;
8          ;
9          ;
10         ;

```

The Microsoft MACRO Assembler  
UNIT 9 -- PROGRAM 11 -- PAGE+

08-06-84 PAGE 2-1

```

11         PAGE+
12         ;
13         ;
14         ;
15         0000 CB          RET
16         ;
17         0001      START  ENDP
18         0001      PROG_CODE ENDS
19         ;                END      START

```

**Figure 9-14**

Assembler listing using the PAGE+ directive.

There will be times when you want to start a new listing page, and at the same time, increment the major page number. For this, you use the PAGE directive followed by a plus sign (PAGE+). Figure 9-14 shows how the listing is affected by the directive. Each time the assembler encounters the directive PAGE+: it begins a new listing page, the major page number is incremented, and the minor page number is reset to one. The minor page number will continue to increment for each new listing page, but the major page number will stay at the value established by the last PAGE+ directive. The only way to increment the major page number is with another PAGE+ directive.

The last area controlled by the PAGE directive is the length and width of a page listing. The directive takes the form

**PAGE <length>,<width>**

In this arrangement, the `PAGE` directive does not cause a page break. The argument `<length>` specifies the number of lines that will be printed on a page. More precisely, it determines the number of lines the microcomputer will send before it sends a **form feed** character. A form feed character is an ASCII code (0CH) that tells the printer to begin a new page. Naturally, the printer must be set up to accommodate a page listing. If the listing length is 40, and the printer is set up for 88 lines to a page, each listing will only fill about half a page. You can use any length value within the range of 10 to 255 lines. The default length is 58 lines per page — an ideal length for a printer set up for a 66-line page.

The argument `<width>` specifies the number of characters that will be printed on a listing line. This is the maximum line width. If a line exceeds the specified width, the assembler will automatically continue the line on the next line in the listing. We call this operation **word wrap**. Keep in mind, however, that a listing line is actually composed of two parts: the assembler generated code and the source statement. Because there are two parts to a listing line, word wrap can occur in either part. Figure 9-15 is an example.

```

The Microsoft MACRO Assembler      08-07-84   PAGE   1-1
UNIT 9 -- PROGRAM 12 -- PAGE [LENGTH],[WIDTH] DISPLAY SIZE

1          PAGE 58,80
2          TITLE UNIT 9 -- PROGRAM 12 -- PAGE [LEN
3          GTH],[WIDTH] DISPLAY SIZE
4          ;
5          0000          PROG_CODE SEGMENT
6                   ASSUME CS:PROG_CODE
7          ;
8          0000          START  PROC  FAR
9          0000  41 42 43 44 45 46  ;
                   DATA  DB      'ABCDEFHIJKLMNQRSTU
                   WXYZ'
10         47 48 49 4A 4B 4C
11         4D 4E 4F 50 51 52
12         53 54 55 56 57 58
13         59 5A
14         ;
15         001A  CB          ;          RET
16         ;
17         001B          START  ENDP
18         001B          PROG_CODE ENDS
19         END          END      START

```

**Figure 9-15**

Assembler listing to show how line width is controlled.

The first line of the program contains the PAGE directive. The first number in the directive statement is the listing page length; the second number is the listing line width. To keep the figure simple, we used the normal default values for length and width.

Line nine of the listing contains a define byte statement. Because the line is longer than 80 characters, it wraps around to the next line. However, the extra characters are not continued at the left end of the next line as you might expect, they are offset 33 spaces. This is because the first 32 character locations of each listing line are reserved for the program line numbers, code, and data. To make it easier for you to determine when a source statement is wrapped around to the next line, the extra line is not given a line number in the listing.

The code and data portion of the listing also wraps around to the next line. In this case, the wrap occurs at character location 32. The next line then begins at character location 10. This offset is used to keep the code and data out of the line number and memory address columns.

You can use any width value within the range of 60 to 132 characters to determine the width of a listing line. The default width is 80 characters. This was chosen to match the number of characters that could be displayed by the microcomputer. You should use a width that matches the width of your printer. Remember, if the width exceeds 80 characters, you won't be able to see the complete listing on your computer display.

Early versions of the assembler (1.00, 1.07, etc.) do not handle the assembler listing properly if you specify a page size that exceeds the default values. Shorter lengths and widths will list properly. Later versions of the assembler properly handle any page size, up to the maximum length and width.

## **TITLE**

The directive **TITLE** (program title) was described earlier in the course. Since you have been using it in most of your programs, you should understand how it works. Therefore, we won't repeat the description.

## **SUBTTL**

The directive **SUBTTL** (program subtitle) allows you to identify sections of a program with a title related to that section. After a subtitle is identified by the assembler, it is displayed on the line following the program title, at the top of each succeeding page of the program listing. The directive takes the form

```
SUBTTL  <text>
```

where **SUBTTL** is the directive and <text> is the subtitle that will be listed at the top of the page. The text can contain up to 60 characters.

The Microsoft MACRO Assembler  
UNIT 9 -- PROGRAM 13 -- SUBTITLES

08-07-84 PAGE 1-1

```

1          TITLE UNIT 9 -- PROGRAM 13 -- SUBTITLES
2          ;
3          0000      PROG_CODE SEGMENT
4                      ASSUME CS:PROG_CODE
5          ;
6          0000      START  PROC  FAR
7          ;
8          ;
9          ;
10         SUBTTL  ***SUBTITLE #1***

```

The Microsoft MACRO Assembler  
UNIT 9 -- PROGRAM 13 -- SUBTITLES

08-07-84 PAGE 1-2

```

***SUBTITLE #1***
11         PAGE
12         ;
13         ;
14         ;
15         SUBTTL  ***SUBTITLE #2***

```

The Microsoft MACRO Assembler  
UNIT 9 -- PROGRAM 13 -- SUBTITLES

08-07-84 PAGE 2-1

```

***SUBTITLE #2***
16         PAGE+
17         ;
18         0000  CB          RET
19         ;
20         0001      START  ENDP
21         0001      PROG_CODE ENDS
22         END          START

```

**Figure 9-16**

Assembler listing to show how subtitles are presented.

Any number of SUBTTL directives can be used in a program. Each time the assembler encounters the SUBTTL directive, it replaces the text from the previous subtitle with the text from the latest subtitle. Keep in mind that the text will be displayed on the **next** page of the program listing. Therefore, if you want to identify a program section with a subtitle, it's a good idea to place the directive prior to a listing "page break." Then arrange the listing so that the identified program section begins on the next page. Depending on the complexity of your program, you could use the basic PAGE directive to force a page break, or you could use the PAGE+ directive to force the page break and increment the major page number. Figure 9-16 shows an example of each.

Once a subtitle is "turned on" it can only be "turned off" by entering the SUBTTL directive without any text. Naturally, a new subtitle will replace the previous subtitle in a listing.

## Listing Control Directives

The listing control directives let you determine what part of the assembly listing will be saved in the listing file. They can be used to hide parts of a program listing to save disk space, or they can be used to reveal areas of a program that are normally hidden.

### %OUT

The %OUT (display message) directive was described earlier. It is used to display one or more messages during the assembly process. By embedding a message with the %OUT directive in your program, you can follow the assembly process for very long programs. The directive is formatted

```
%OUT <text>
```

Where <text> represents the message to be displayed. The message will be displayed on both passes of the assembler. If you wish to display the message only once, use one of the conditional directives described earlier.

### **.LIST/.XLIST**

The **.LIST** and **.XLIST** directives are used to enable or suppress the listing of all source statements, code, and data. The default condition is **.LIST**. That is, all program source statements, code, and data are saved in the assembler source listing. When the assembler encounters the **.XLIST** directive, it will stop generating a listing. The **.LIST** directive is used after the **.XLIST** directive to turn the assembler listing back on. These directives are handy if you wish to list a portion of a program and ignore the rest.

### **.XALL/.LALL/.SALL**

The **.XALL**, **.LALL**, and **.SALL** directives are used to control the listing of macros. The default directive is **.XALL**. It causes the source statements, code, and data to be listed, but suppresses the listing of any source lines that do not generate code or data.

The **.LALL** directive causes all macro source statements, code, and data to be listed. This was described earlier under macros. To return to the default condition after using the **.LALL** directive, use the **.XALL** directive. The **.SALL** directive is used to suppress all listing of macro code, data, or source statements. This is similar to the **.XLIST** directive except that it only affects macros.

### **.LFCOND/.SFCOND**

The **.LFCOND** and **.SFCOND** directives are used to control the listing of false conditional expressions. The default directive is **.SFCOND**. It causes all conditional expressions that evaluate as false to be suppressed by the assembler. To turn on the listing of false conditional expressions, use the **.LFCOND** directive. To return to the default condition, use the **.SFCOND** directive.



## Self-Review Questions

49. Write the directive that will cause a page break and increment the major page number. \_\_\_\_\_
50. Write the directive that will cause a page break and increment the minor page number. \_\_\_\_\_
51. Write the directive that will set the page length to 80 and the page width to 132. \_\_\_\_\_
52. The characters following the directive \_\_\_\_\_ are used by the assembler to determine every page heading in the assembler listing.
53. The characters following the directive \_\_\_\_\_ are used by the assembler to determine the secondary page heading in the assembler listing.
54. To display a message during an assembly operation, precede the message with the directive \_\_\_\_\_ in the program source listing.
55. To suppress an assembler listing, use the \_\_\_\_\_ directive.
56. To “turn on” an assembler listing, use the \_\_\_\_\_ directive.
57. To suppress the listing of any source lines in a macro that do not generate code or data, use the \_\_\_\_\_ directive.
58. To “turn on” the complete listing in a macro, use the \_\_\_\_\_ directive.
59. To suppress the complete listing in a macro, use the \_\_\_\_\_ directive.
60. To suppress the listing of any false conditional expressions, use the \_\_\_\_\_ directive.
61. To “turn on” the listing of any false conditional expressions, use the \_\_\_\_\_ directive.

## EXPERIMENT

### Loose Ends

*OBJECTIVES:*

1. *Demonstrate the GROUP and PROCEDURE directives.*
2. *Demonstrate how the conditional directives can be used in a program.*
3. *Demonstrate the features of code macros.*
4. *Demonstrate the assembler listing directives.*

### Introduction

This experiment will complete your introduction to programming in MACRO-86 assembly language. We call this experiment “Loose Ends” because it presents many features of MACRO-86 that most programmers will never use. However, these features are significant, and a course in MACRO-86 programming would not be complete without illustrating their operation.

### Procedure

1. Call up the editor and enter the program listed in Figure 9-17. Then assemble and link the program.

This program shows you how the GROUP and PROCEDURE directives can be used. The stack, data, and code segments for the main part of the program are clustered under A\_GROUP. A second code segment is in B\_GROUP. The main program code is part of far procedure START. This allows you to exit the program with a FAR return. Two call instructions access a near procedure (ADDUP) in the code segment of A\_GROUP, and a far procedure (COUNT\_DOWN) in the code segment of B\_GROUP.

```

TITLE EXPERIMENT 9 -- PROGRAM 1 -- PROCEDURES AND GROUPS
;
A_GROUP GROUP PROG_STACK,PROG_DATA,PROG_CODE
;
PROG_STACK SEGMENT STACK
    DW      80 DUP (0)      ;Set up stack area
TOP_OF_STACK LABEL WORD    ;Identify top of stack for SP register
PROG_STACK ENDS
;
PROG_DATA SEGMENT PUBLIC
DATA DB 0
PROG_DATA ENDS
;
PROG_CODE SEGMENT PUBLIC
    ASSUME CS:A_GROUP,DS:A_GROUP,SS:A_GROUP
COUNT EQU 50
START PROC FAR
    MOV     AX,A_GROUP      ;Use the GROUP base address to
    MOV     SS,AX          ;load the Stack Segment register
    MOV     SP,OFFSET A_GROUP:TOP_OF_STACK ;Point to top of stack
    PUSH   DS              ;Save segment for FAR return value
    SUB    AX,AX           ;Zero the AX register
    PUSH   AX              ;Save offset of zero for far return
    MOV    AX,A_GROUP      ;Use the GROUP base address to
    MOV    DS,AX           ;load the Data Segment register
    NOP                    ;Allow next instruction to be visible
                        ;when single-stepping in debugger
DO_AGAIN:
    CALL   FAR PTR COUNT_DOWN;Create a short time delay
    CALL   ADDUP            ;Count the number of time delays
    CMP    A_GROUP:DATA,COUNT;Is the count finished?
    JNE    DO_AGAIN        ;NO, repeat - YES, quit
    RET                    ;Exit the program
START ENDP
;
ADDUP PROC NEAR           ;Begin a near procedure
    INC    A_GROUP:DATA    ;Add one to memory location DATA
    RET                    ;Return from a near procedure
ADDUP ENDP
PROG_CODE ENDS
;
B_GROUP GROUP FAR_CODE
;
FAR_CODE SEGMENT PUBLIC
    ASSUME CS:B_GROUP
COUNT_DOWN PROC FAR     ;Begin a far procedure
    MOV    CX,0BBBBH      ;Set up a delay loop count
AGAIN: LOOP AGAIN        ;Loop until count is zero
    RET                    ;Return from a far procedure
COUNT_DOWN ENDP
FAR_CODE ENDS
;
    END    START

```

**Figure 9-17**

Program showing how groups and procedures can be used.

2. Load the EXE file for your program into the debugger. Examine the MPU registers. Record the following register values.

Code Segment	----_H.
Data Segment	----_H.
Stack Segment	----_H.
Stack Pointer	----_H.

Right now, the only register that contains the base address for A\_GROUP is CS. The Stack Pointer register points to what is currently the top of the stack. This will change when you begin executing program code.

3. Single-step to the first call instruction. Record the following register values.

Code Segment	----_H.
Data Segment	----_H.
Stack Segment	----_H.
Stack Pointer	----_H.

Now the registers have the correct program values. All three segment registers contain the base address for A\_GROUP, and the Stack Pointer register points to the current top of the program stack. The stack also contains the FAR return address for program termination.

4. Single-step through the call instruction. The CS register now contains the value \_\_\_ \_H. This is the base address for the code segment in B\_GROUP.

When you entered the program listing in step one, you may have wondered about the assembler operator FAR PTR in this call instruction. The operator was needed to tell the assembler how many bytes to reserve for the target address of the call. We didn't use the operator in our earlier example of a far procedure because the target label preceded the call instruction — there was no forward reference. The operator was needed in this program because of the forward reference to the target label.

5. The next three instructions form a short time delay loop. You can single-step through the loop 0BBBBH times, or you can let the debugger do it for you. When you are ready to complete the loop, type "G0005" and RETURN. This tells the debugger to execute the program instructions down to the instruction at offset address 0005H. After a slight pause, the debugger will display the FAR return instruction.
6. Single-step through the return instruction. The CS register now contains the value `___H`. This is the base address for the main program code segment. The IP register contains the value `____H`.
7. Single-step through the call instruction. This instruction calls the near procedure ADDUP. The IP register contains the value `___H`. Calls to near procedures update the IP register, but not the CS register. Calls to far procedures update both.
8. Before you single-step through the increment instruction, examine the memory location pointed to by the instruction. Type "D" (and the four numbers located in the source operand of the instruction) and RETURN. The first byte in the memory display contains the value `__H`.
9. Single-step through the increment instruction. It adds one to the memory location DATA. Previously, all of your programs incremented a register. We incremented a memory location just to show you that it will work. Notice that the program used the group override operator to identify the base address of the data segment. The compare instruction also uses that operator.
10. Examine the memory location that you just incremented. Does it contain the value one?
11. Single-step through the next three instructions. The return from call instruction is executed, the data in memory location DATA is compared to the value 50 (COUNT equated to 50), and the jump if not equal instruction is taken (DATA does not equal 50). This puts the MPU back at the FAR call instruction. The sequence will repeat until DATA equals 50.

12. Now run the program to completion — type “G” and return. The program will take a few seconds to execute. When it is finished, the debugger will print the message:

**Program terminated normally**

13. Examine the DATA memory location one more time. It should contain the value 32H (decimal 50).
14. Exit the debugger. Then call up your editor and modify your program as follows:
  - A. Just before the A\_GROUP group statement, add the listing directive .XLIST.
  - B. Just before the B\_GROUP group statement, add the listing directive .LIST.
15. Assemble the program. Then examine the assembler listing. You should see the program title, the .LIST directive, and the rest of the assembler listing of the program, beginning with the B\_GROUP group statement. Remember, the default listing directive is .LIST. This is “turned off” by the .XLIST directive. To turn the listing back on, use the .LIST directive.
16. Call up your editor and enter the program listed in Figure 9-18. This program will be used to show you how the conditional directives affect the assembly of a program.
17. Assemble the program. Did the assembler display four messages?

The first message in your program should have been displayed twice, once for each pass of the assembler. The next two messages should have been displayed once. Their display is controlled by the conditionals IF1 and IF2.

18. Examine the program listing. The only define byte statement to be assembled is DATA2. Do you understand why?

```
TITLE EXPERIMENT 9 -- PROGRAM 2 -- CONDITIONALS
;
.LFCOND
;
%OUT THIS MESSAGE WILL BE DISPLAYED TWICE
;
    IF1
%OUT THIS MESSAGE WILL BE DISPLAYED ON THE FIRST ASSEMBLER PASS
    ENDIF
;
    IF2
%OUT THIS MESSAGE WILL BE DISPLAYED ON THE SECOND ASSEMBLER PASS
    ENDIF
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE
;
COUNT EQU 0
TEST EQU 0
;
START PROC FAR
;
    IFDEF TEST
        IF COUNT
            DATA1 DB 'FIRST'
        ENDIF
;
        IF COUNT + 5
            DATA2 DB 'SECOND'
        ENDIF
    ENDIF
;
    IFNDEF TEST
        IFE COUNT + 5
            DATA3 DB 'THIRD'
        ELSE
            DATA4 DB 'FOURTH'
        ENDIF
    ENDIF
;
    RET
START ENDP
PROG_CODE ENDS
    END START
```

**Figure 9-18**

Program showing how conditional directives can be used.

## Discussion

The main part of the program contains two sets of nested conditional directives. Depending on the status of the symbol TEST, either the first or second set of conditionals are assembled. Since TEST has been defined, the conditional IFDEF TEST tests true. This allows the next two conditionals to be tested. The first conditional, IF COUNT, tests false because COUNT is zero — DATA1 is not assembled. The second conditional, IFE COUNT, tests true. Therefore, DATA2 is assembled. Because TEST has been defined, the conditional IFNDEF TEST tests false. As a result, the nested conditionals following IFNDEF are not tested or assembled.

## Procedure Continued

19. Call up your editor and delete the equate statement

```
TEST EQU 0
```

from your program.

20. Assemble the program. Then examine the assembler listing. The only define byte directive to be assembled is DATA4.

## Discussion

With the symbol TEST missing from the program, the IFDEF TEST conditional tests false, and its nested conditionals are not tested or assembled. On the other hand, IFNDEF TEST tests true, allowing its nested conditionals to be tested. Conditional IFE COUNT + 5 tests



false because the value of COUNT plus five does not equal zero — define byte DATA3 is not assembled. This causes the ELSE conditional to test true — define byte DATA4 is assembled.

## Procedure Continued

21. Call up your editor and change the .LFCOND directive to .SFCOND.
22. Assemble your program. Notice that all four assembler messages are displayed. Now examine the assembler listing. Is the display what you expected to see?

## Discussion

The .SFCOND directive suppressed the listing of all false conditionals. The first of these was the IF1 conditional. It tested true on the first pass of the assembler and its message was printed. On the second pass, it tested false. Since the listing is generated from the second pass, it was not displayed. Only the ENDIF directive remained to show its original location.

The second conditional to be suppressed was IFDEF TEST. Its ENDIF directive remains to show its original location. Because IFDEF TEST tested false, its nested conditionals were never tested by the assembler. As a result, they are completely suppressed by the .SFCOND directive.

The last conditional and its related data statement to be suppressed is IFE COUNT + 5.

## Procedure Continued

23. Call up your editor and enter the program listed in Figure 9-19. This is the first of the programs that will be used to show you how the macro directives and their operators affect the assembly of a program.
24. Assemble the program. Now examine the assembler listing.

```

TITLE EXPERIMENT 9 -- PROGRAM 3 -- MACROS 1
;
.LALL
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE
;
START  PROC  FAR
;
TEST   MACRO  X,Y,Z
ONE    DW     X
TWO:   SUB    AX,AX
        ADD   AX,ONE
        CMP   AX,Y
        J&Z  TWO
        ENDM
;
TEST   5,10,NE
;
        RET
START  ENDP
PROG_CODE ENDS
        END    START

```

**Figure 9-19**

The first macro program.

## Discussion

The macro template contains three dummy parameters. These are replaced by parameters supplied by the macro call

```
TEST 5,10,NE
```

The assembler expanded the macro template, and assembled the code and data. This expansion is identified by plus signs in the listing. Notice that the macro operator “&” caused the assembler to replace the dummy “Z” with text characters “NE” to produce the instruction mnemonic JNE.

## Procedure Continued

25. Call up your editor and add the macro directive statement

```
TEST 5,5,E
```

pointed to by the arrow in Figure 9-20. Do not add the macro directive statement

```
LOCAL ONE,TWO
```

at this time.

26. Assemble the program. One simple macro call generated ten assembly errors. Why do you think that happened?

```
TITLE EXPERIMENT 9 -- PROGRAM 4 -- MACROS 2
;
.LALL
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE
;
START PROC FAR
;
TEST  MACRO X,Y,Z
      LOCAL ONE,TWO
ONE    DW X
TWO:   SUB AX,AX
      ADD AX,ONE
      CMP AX,Y
      J&Z TWO
      ENDM
;
TEST 5,10,NE
      TEST 5,5,E
;
      RET
START ENDP
PROG_CODE ENDS
      END START
```

**Figure 9-20**

Macro program expanded with the LOCAL directive.

## Discussion

The program now contains two macro calls that expand the same macro template. This produces two sets of code and data with the same symbols. The assembly errors were generated because the symbols defined more than one memory location.

## Procedure Continued

27. Call up your editor and add the macro directive statement

```
LOCAL ONE, TWO
```

pointed to by the arrow in Figure 9-20.

28. Assemble the program. Examine the assembler listing. All of the assembly errors have been corrected by the LOCAL directive. It has also created a unique symbol for each symbol in each macro expansion.
29. Call up your editor and delete the LOCAL directive statement and the macro call, pointed to by arrows in Figure 9-20, from your program. Refer to Figure 9-21 and add the define byte directive pointed to by an arrow. Then modify the macro statement and macro call, pointed to by arrows, as shown.
30. Assemble the program. Examine the assembler listing.

```

TITLE EXPERIMENT 9 -- PROGRAM 5 -- MACROS 3
;
.LALL
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE
;
START PROC FAR
;
→ TEST MACRO X, Y, Z, U, V
→ ZERO DB '&U', '&V'
ONE DW X
TWO: SUB AX, AX
ADD AX, ONE
CMP AX, Y
J&Z TWO
ENDM
;
→ TEST 5, 10, NE, !;, <COUNT>
;
RET
START ENDP
PROG_CODE ENDS
END START

```

**Figure 9-**  
Macro program with literal macro operators added.

## Discussion

Both the semicolon and the word COUNT were treated as literal characters in the macro expansion. The dummy parameters "U" and "V" were identified with an ampersand in the define byte statement. Without the ampersand, the two parameters would not have been used.

## Procedure Continued

31. Call up your editor with your program. Delete the macro template and the macro call. Then add the equate statement, macro template, and macro call in Figure 9-22.
32. Assemble the program. Examine the assembler listing.

```
TITLE EXPERIMENT 9 -- PROGRAM 6 -- MACROS 4
;
.LALL
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE
;
START PROC FAR
;
X EQU 0
TEST MACRO Z
    REPT Z
    X = X + 1
    DB X
    ENDM
    ENDM
;
TEST 3
;
    RET
START ENDP
PROG_CODE ENDS
END START
```

**Figure 9-22**

Macro program using the repeat directive.

## Discussion

The macro call TEST passes the parameter “3” to the repeat macro nested in the macro template TEST. The expression “X” is originally equated to the value zero. Each time the define byte statement in the macro is repeated, the value of “X” is incremented. That incremented value becomes part of the program machine code. The incrementing process is shown in the listing because of the listing directive .LALL (list all). Now lets see how the listing is changed when .LALL is removed from the program.

## Procedure Continued

33. Call up your editor and delete the directive .LALL.
34. Assemble the program. Examine the assembler listing. Now the only part of the macro listing that is displayed are the three define byte statements that generate machine code. This is equivalent to using the .XALL directive.
35. Call up your editor and add the .SALL directive where the .LALL directive was originally located in your program.
36. Assemble the program. Examine the assembler listing. The macro template and macro call are listed, but all of the macro expansion has been suppressed.
37. Call up your editor and modify your program as shown in Figure 9-23. First replace the .SALL directive with the .LALL directive. Then add the conditional directive statement and the ENDIF directive. These changes are pointed to by arrows. You might want to offset the repeat macro template one tab to make the nesting levels clear, as shown in the figure.
38. Assemble the program. Examine the assembler listing.

```

TITLE EXPERIMENT 9 -- PROGRAM 7 -- MACROS 5
;
->.LALL
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE
;
START PROC FAR
;
X EQU 0
TEST MACRO Z
->IFNB <Z>
    REPT Z
    X = X + 1
    DB X
    ENDM
->ENDIF
    ENDM
;
TEST 3
;
    RET
START ENDP
PROG_CODE ENDS
    END START

```

**Figure 9-23**

Macro program using conditional expansion.

## Discussion

The conditional directive tests the parameter passed by the macro call. As long as the parameter is not blank, the conditional will test true, and the nested macro will be expanded. In this example, the test was true, and the repeat macro was allowed to create three defined bytes using the passed parameter. If the parameter is blank, the macro will not be expanded. Let's see what will happen if the parameter is blank.

## Procedure Continued

39. Call up your editor and change the macro call statement in your program to read:

```
TEST <>
```

40. Assemble the program. Examine the assembler listing.

## Discussion

The parameter passed by the macro call was blank (null). Because of that, the conditional tested false, and the macro repeat was not assembled. The ENDIF directive was listed because that was all that was left in the macro template after the conditional tested false.

You may have noticed earlier that when the repeat macro was expanded, no symbols were assigned to the define byte statements. That's because you can't concatenate a value directly to text with the ampersand operator. If you want to assign a symbol to data in a macro repeat, you must use the repeat macro call method described in the section "Other Macro Directives." Let's try an example.

## Procedure Continued

41. Call up your editor and modify your program as shown in Figure 9-24. First, delete the conditional directive statement and the ENDIF directive. Then, change the macro call statement back to:

```
TEST 3
```

Next, replace the define byte statement, in the repeat macro, to the macro call statement:

```
CHANGE %X
```

Finally, add the macro template CHANGE after the macro template TEST. These modifications are pointed to by arrows in the figure.

42. Assemble the program. Examine the assembler listing.



```

TITLE EXPERIMENT 9 -- PROGRAM 8 -- MACROS 6
;
.LALL
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE
;
START PROC FAR
;
X EQU 0
TEST MACRO Z
    REPT Z
    X = X + 1
    CHANGE %X
    ENDM
    ENDM
;
CHANGE MACRO Y
DATA&Y DB Y
ENDM
;
TEST 3
;
    RET
START ENDP
PROG_CODE ENDS
END START

```

**Figure 9-24**

Macro program using a nested repeat macro call.

## Discussion

Macro call TEST passes the parameter three to macro template TEST. This value is used to determine the number of times the repeat macro is repeated. Each time it is repeated, the macro call CHANGE is executed. The percent sign in front of the macro call parameter “Y” tells the assembler that it will be passing a variable represented by “Y” instead of text to the called macro template CHANGE. That variable then replaces each dummy parameter “Y” in the macro template. The result of this macro expansion is the creation of three define byte statements with a unique symbol for each.

Because the IRP and IRPC macro directives are similar in function, we’ll only give you an example of one, the IRP macro directive.

## Procedure Continued

43. Call up your editor and delete the equate statement, the two macro templates, and the macro call in your program. Then add the macro template and macro call shown in Figure 9-25.

```

TITLE EXPERIMENT 9 -- PROGRAM 9 -- MACROS 7
;
.LALL
;
PROG_CODE SEGMENT
    ASSUME CS:PROG_CODE
;
START PROC FAR
;
LAST MACRO X
    IRP Z,<1,2,3>
        X&&Z DB Z
    ENDM
;
LAST DATA
;
    RET
START ENDP
PROG_CODE ENDS
    END START

```

**Figure 9-25**

Macro program using an indefinite repeat macro.

44. Assemble the program. Examine the assembler listing.

## Description

Notice that two ampersands are used in the define byte symbol of the macro template. When the macro is called, the parameter DATA is substituted for the dummy parameter "X" and the first ampersand is "consumed." This is shown in the intermediate step of the macro expansion. The second ampersand is consumed when a repeat directive parameter replaces the repeat dummy parameter in the define byte symbol.

The last part of this experiment will cover the PAGE and SUBTTL directives.

## Procedure Continued

45. Call up your editor, with your last program, and enter the directive PAGE before the program line

```
START PROC FAR
```

before the program line

```
LAST DATA
```

and before the program line

```
START ENDP
```

46. Assemble the program. Examine the assembler listing. Notice that a page break occurs before each PAGE directive, the major page number doesn't change, and the minor page number increments for each new page.
47. Call up your editor and add a plus sign after the second PAGE directive in your program. Then add the directive statement

```
SUBTTL ***MACRO EXPANSION***
```

before the PAGE + directive.

48. Assemble the program. Examine the assembler listing. This time the major page number on the third page has incremented to two and the minor page number has reset to one. In addition, the third page has the subtitle

```
***MACRO EXPANSION***
```

centered under the program title line. The fourth page's major page number remains at two, while the minor page number has incremented to two. The fourth page also has the subtitle

```
***MACRO EXPANSION***
```

49. Call up your editor and add the directive SUBTTL before the last PAGE directive in your program.

50. Assemble the program. Examine the assembler listing. This time the subtitle is not listed on the fourth page. It was “turned off” by the SUBTTL directive with no text.
  
51. Call up your editor and delete the three page directives and the two subtitle directives. Then enter the page directive

PAGE 10,60

before the TITLE directive statement.

52. Assemble the program. Examine the assembler listing. The source listing portion of the display is 20 characters shorter than before. As a result, many of the lines wrap around. The page length is nine lines. You specified ten lines; why do you think there are only nine lines displayed? The tenth line holds the “form feed” character. Since this is not a printable character, you can only see nine lines. For that same reason, there are only 59 characters on a line. The sixtieth character is the nonprintable carriage return character.

If you have an early version assembler (1.00, 1.07, etc.) you can't specify a listing size that exceeds the default values. However, if you have a later version, you might try different listing values to see how the display is handled. Add a long comment line or extend the title to 60 characters, to see the effect.

If you have a printer, try printing the listing to see what size best fits your needs. Naturally, the early version assemblers will limit what you can do to your page format.

This completes the Experiment for Unit 9. Proceed to the Unit 9 Examination.

## UNIT 9 EXAMINATION

1. A procedure defines the beginning and end of a \_\_\_\_\_.
2. A procedure can have a "type" \_\_\_\_\_ or \_\_\_\_\_.
3. A type \_\_\_\_\_ procedure is used to set up the exit to the system (return) from an EXE program.
4. The \_\_\_\_\_ directive allows you to gather the segments of a program under one segment base address value.
5. The code or data following a conditional directive will be assembled if the condition being tested tests \_\_\_\_\_.
6. Every conditional operation must be ended with the \_\_\_\_\_ directive.
7. The IF directive will test true if the expression is \_\_\_\_\_.
8. The IFE directive will test true if the expression is \_\_\_\_\_.
9. The IF1 directive will test true on the \_\_\_\_\_ pass of the assembler.
10. The IF2 directive will test true on the \_\_\_\_\_ pass of the assembler.
11. The \_\_\_\_\_ directive will test true if its symbol has been defined.
12. The \_\_\_\_\_ directive will test true if its argument is not blank.
13. The \_\_\_\_\_ directive will test true if string argument one is different from string argument two.
14. The \_\_\_\_\_ directive is used as an alternative at assembly time to a false conditional.
15. The \_\_\_\_\_ directive is a programming tool that allows you to generate or repeat a sequence of code or data within a program.

16. A macro call is used to pass one or more \_\_\_\_\_ to a macro template.
17. A repeat macro directive can take one of three forms: \_\_\_\_\_, \_\_\_\_\_, or \_\_\_\_\_.
18. The indefinite repeat macro directive that requires angle brackets around its parameters is \_\_\_\_\_.
19. The exit directive for a conditional is \_\_\_\_\_.
20. The \_\_\_\_\_ directive eliminates the problem of multiple symbol definition when a macro template is expanded more than once.
21. The macro operator \_\_\_\_\_ is used to concatenate text or symbols within a macro expansion.
22. The minor page number is incremented after a page break generated by the \_\_\_\_\_ directive.
23. The minor page number is reset to one after a page break generated by the \_\_\_\_\_ directive.
24. The length and width of an assembler listing page listing is determined by the \_\_\_\_\_ directive.
25. If a subtitle is specified on the third line of a program, the first time it will be used is on the \_\_\_\_\_ page of the program listing.
26. To “turn off” the listing of a program, use the \_\_\_\_\_ directive.
27. To “turn off” the listing of a false conditional, use the \_\_\_\_\_ directive.
28. To “turn off” the listing of a macro expansion, use the \_\_\_\_\_ directive.

## EXAMINATION ANSWERS

1. A procedure defines the beginning and end of a **subroutine or program segment**.
2. A procedure can have a “type” **NEAR** or **FAR**.
3. A type **FAR** procedure is used to set up the exit to the system (return) from an EXE program.
4. The **GROUP** directive allows you to gather the segments of a program under one segment base address value.
5. The code or data following a conditional directive will be assembled if the condition being tested tests **true**.
6. Every conditional operation must be ended with the **ENDIF** directive.
7. The **IF** directive will test true if the expression is **nonzero**.
8. The **IFE** directive will test true if the expression is **zero**.
9. The **IF1** directive will test true on the **first** pass of the assembler.
10. The **IF2** directive will test true on the **second** pass of the assembler.
11. The **IFDEF** directive will test true if its symbol has been defined.
12. The **IFNB** directive will test true if its argument is not blank.
13. The **IFDIF** directive will test true if string argument one is different from string argument two.
14. The **ELSE** directive is used as an alternative at assembly time to a false conditional.
15. The **MACRO** directive is a programming tool that allows you to generate or repeat a sequence of code or data within a program.

16. A macro call is used to pass one or more **parameters** to a macro template.
17. A repeat macro directive can take one of three forms: **REPT**, **IRP**, or **IRPC**.
18. The indefinite repeat macro directive that requires angle brackets around its parameters is **IRP**.
19. The exit directive for a conditional is **EXITM**.
20. The **LOCAL** directive eliminates the problem of multiple symbol definition when a macro template is expanded more than once.
21. The macro operator **ampersand** is used to concatenate text or symbols within a macro expansion.
22. The minor page number is incremented after a page break generated by the **PAGE** directive.
23. The minor page number is reset to one after a page break generated by the **PAGE+** directive.
24. The length and width of an assembler listing page listing is determined by the **PAGE** directive.
25. If a subtitle is specified on the third line of a program, the first time it will be used is on the **second** page of the program listing.
26. To “turn off” the listing of a program, use the **.XLIST** directive.
27. To “turn off” the listing of a false conditional, use the **.SFCOND** directive.
28. To “turn off” the listing of a macro expansion, use the **.SALL** directive.



## SELF-REVIEW ANSWERS

1. Procedures have a rigid **structure**.
2. The beginning of a procedure is identified by the directive **PROC**.
3. The end of a procedure is identified by the directive **ENDP**.
4. Every procedure should contain at least one **RET (return)** instruction.
5. A **FAR** procedure is used to define the main code section of an EXE program.
6. The procedure return instruction type is determined by the procedure **type**.
7. The EXE-type program termination address is determined by pushing the original contents of the **Data** segment register and the address offset **zero** into the stack.
8. The best way to access a procedure is with a **call** instruction.
9. **True**. Procedures can be nested any number of times, as long as there is room on the stack for all of the return addresses.
10. The **GROUP** directive allows you to gather any number of program segments into one identifiable collection and reference the data or code in the segments to a single base address value.
11. The maximum size of a group is **64K** bytes.
12. Group size is checked by the **linker**.
13. Your program contains the following segments:

```
PROG_STACK  
PROG_DATA  
PROG_CODE
```

Write a **GROUP** directive statement that will combine all of the program segments into one group called **PGROUP**.

```
PGROUP GROUP PROG_STACK,PROG_DATA,PROG_CODE
```

14. Write the ASSUME directive statement for the program in question 13. The Extra Segment register will not be used by the program.

**ASSUME CS:PGROUP,DS:PGROUP,SS:PGROUP**

15. Write the instruction that will load the Stack Pointer register for the program in question 13. The end of the stack is identified by the LABEL directive:

TOP\_OF\_STACK LABEL WORD

**MOV SP,OFFSET PGROUP:TOP\_OF\_STACK**

16. Conditional directives are assembler instructions that specify whether a section of a program should be assembled or not.
17. The beginning of a conditional directive is identified by the **IFxxxx** directive.
18. The end of a conditional directive is identified by the directive **ENDIF**.
19. **True.** Conditional directives can be nested up to 255 levels.
20. An **IF** directive will evaluate true if its expression is a nonzero.
21. An **IFE** directive will evaluate true if its expression is a zero.
22. An **IF1** directive is enabled during pass one of the assembler.
23. An **IF2** directive is enabled during pass two of the assembler.
24. An **IFDEF** directive will evaluate true if its symbol has been defined in the program.
25. An **IFNDEF** directive will evaluate true if its symbol has not been declared External.
26. An **IFB** directive will evaluate true if it has a null argument.
27. An **IFNB** directive will evaluate true if it contains an argument.

28. An **IFIDN** directive will evaluate true if the string in argument one equals the string in argument two.
29. An **IFDIF** directive will evaluate true if the string in argument one does not equal the string in argument two.
30. The **%OUT** directive can be used to display a message during program assembly.
31. The **ELSE** directive allows an alternate conditional block to be assembled if the specified condition is not met.
32. A **macro** is a programming tool that allows you to generate or repeat a sequence of code or data within a program.
33. The directive **MACRO** tells the assembler that the following code or data is part of a macro template.
34. The directive **ENDM** identifies the end of a macro template.
35. A macro **name** is used by a macro call to identify a macro template.
36. A macro call contains one or more **parameters** that are passed to the called macro template.
37. An individual parameter that contains many items separated by commas is identified by enclosing the items with a pair of **angle brackets**.
38. The **REPT (repeat)** macro directive is used to repeat its macro template the number of times specified in the macro directive statement.
39. The **IRP (indefinite repeat)** macro directive is used to replace a dummy parameter within its template with one or more parameters. The parameters are separated by commas and enclosed with angle brackets.
40. The **IRPC (indefinite repeat character)** macro directive is used to replace a dummy parameter within its template with one or more parameters. The parameters consist of a string of characters not separated by commas and not enclosed with angle brackets.

41. The **EXITM (exit macro)** macro directive is used to end a conditional macro operation.
42. The **PURGE** macro directive is used to delete unwanted macro templates from a program.
43. The **LOCAL** macro directive is used to create unique names or labels within a macro expansion.
44. The macro operator **& (ampersand)** is used to concatenate text or symbols during macro expansion.
45. The macro operator angle brackets can be used to create a **literal** character.
46. Two **semicolons** are used to hide a macro comment in an assembler listing.
47. The macro operator **! (exclamation point)** is used to identify a literal character.
48. The macro operator **% (percent sign)** is used in a macro call to convert the expression that follows it into a number.
49. The directive that will cause a page break and increment the major page number is

**PAGE+**

50. The directive that will cause a page break and increment the minor page number is

**PAGE**

51. The directive that will set the page length to 80 and the page width to 132 is

**PAGE 80,132**

52. The characters following the directive **TITLE** are used by the assembler to determine every page heading in the assembler listing.
53. The characters following the directive **SUBTTL** are used by the assembler to determine the secondary page heading in the assembler listing.
54. To display a message during an assembly operation, precede the message with the directive **%OUT** in the program source listing.
55. To suppress an assembler listing, use the **.XLIST** directive.
56. To “turn on” an assembler listing, use the **.LIST** directive.
57. To suppress the listing of any source lines in a macro that do not generate code or data, use the **.XALL** directive.
58. To “turn on” the complete listing in a macro, use the **.LALL** directive.
59. To suppress the complete listing in a macro, use the **.SALL** directive.
60. To suppress the listing of any false conditional expressions, use the **.SFCOND** directive.
61. To “turn on” the listing of any false conditional expressions, use the **.LFCOND** directive.



INSERT





*Appendix A*

**NUMBER SYSTEMS DATA**

## DECIMAL NUMBER SYSTEM

A basic distinguishing feature of a number system is its **base** or **radix**. The base indicates the number of characters or digits used to represent quantities in that number system. The decimal number system has a base or radix of 10 because we use the ten digits 0 through 9 to represent quantities. When a number system is used where the base is not known, a subscript is used to show the base. For example, the number  $4603_{10}$  is derived from a number system with a base of 10.

**Positional Notation** The decimal number system is positional or weighted. This means each digit position in a number carries a particular weight which determines the magnitude of that number. Each position has a weight determined by some power of the number system base, in this case 10. The positional weights are  $10^0$  (units)\*,  $10^1$  (tens),  $10^2$  (hundreds), etc. Refer to Figure 1 for a condensed listing of powers of 10.

$10^0 = 1$
$10^1 = 10$
$10^2 = 100$
$10^3 = 1,000$
$10^4 = 10,000$
$10^5 = 100,000$
$10^6 = 1,000,000$
$10^7 = 10,000,000$
$10^8 = 100,000,000$
$10^9 = 1,000,000,000$

Figure 1  
Condensed listing of powers of 10.

\*Any number with an exponent of zero is equal to one.

We evaluate the total quantity of a number by considering the specific digits and the weights of their positions. For example, the decimal number 4603 is written in the shorthand notation with which we are all familiar. This number can also be expressed with positional notation.

$$\begin{aligned}(4 \times 10^3) + (6 \times 10^2) + (0 \times 10^1) + (3 \times 10^0) &= \\(4 \times 1000) + (6 \times 100) + (0 \times 10) + (3 \times 1) &= \\4000 + 600 + 0 + 3 &= 4603_{10}\end{aligned}$$

To determine the value of a number, multiply each digit by the weight of its position and add the results.

**Fractional Numbers** So far, only **integer** or whole numbers have been discussed. An integer is any of the natural numbers, the negatives of these numbers, or zero (that is, 0, 1, 4, 7, etc.). Thus, an integer represents a whole or complete number. But, it is often necessary to express quantities in terms of fractional parts of a whole number.

Decimal fractions are numbers whose positions have weights that are **negative powers of ten** such as  $10^{-1} = \frac{1}{10} = 0.1$ ,  $10^{-2} = \frac{1}{100} = 0.01$ , etc.

Figure 2 provides a condensed listing of negative powers of 10 (decimal fractions).

$$\begin{aligned}10^{-1} &= \frac{1}{10} = 0.1 \\10^{-2} &= \frac{1}{100} = 0.01 \\10^{-3} &= \frac{1}{1000} = 0.001 \\10^{-4} &= \frac{1}{10,000} = 0.0001 \\10^{-5} &= \frac{1}{100,000} = 0.00001 \\10^{-6} &= \frac{1}{1,000,000} = 0.000001\end{aligned}$$

Figure 2  
Condensed listing of negative  
powers of 10.

A radix point (decimal point for base 10 numbers) **separates** the **integer** and **fractional** parts of a number. The integer or whole portion is to the left of the decimal point and has positional weights of units, tens, hundreds, etc. The fractional part of the number is to the right of the decimal point and has positional weights of tenths, hundredths, thousandths, etc. To illustrate this, the decimal number 278.94 can be written with positional notation as shown below.

$$\begin{aligned}(2 \times 10^2) + (7 \times 10^1) + (8 \times 10^0) + (9 \times 10^{-1}) + (4 \times 10^{-2}) &= \\(2 \times 100) + (7 \times 10) + (8 \times 1) + (9 \times 1/10) + (4 \times 1/100) &= \\200 + 70 + 8 + 0.9 + 0.04 &= 278.94_{10}\end{aligned}$$

In this example, the left-most digit ( $2 \times 10^2$ ) is the **most significant digit** or MSD because it carries the greatest weight in determining the value of the number. The right-most digit, called the **least significant digit** or LSD, has the lowest weight in determining the value of the number.

## BINARY NUMBER SYSTEM

The simplest number system that uses positional notation is the binary number system. As the name implies, a **binary** system contains only two elements or states. In a number system this is expressed as a base of 2, using the digits 0 and 1. These two digits have the same basic value as 0 and 1 in the decimal number system.

### Positional Notation

As with the decimal number system, each bit (digit) position of a binary number carries a particular weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example 2). To evaluate the total quantity of a number, consider the specific bits and the weights of their positions. (Refer to Figure 3 for a condensed listing of powers of 2.) For example, the binary number 110101 can be written with positional notation as follows:

$$(1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

To determine the decimal value of the binary number 110101, multiply each bit by its positional weight and add the results.

$$(1 \times 32) + (1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 32 + 16 + 0 + 4 + 0 + 1 = 53_{10}$$

$2^0 = 1_{10}$	$2^6 = 64_{10}$
$2^1 = 2_{10}$	$2^7 = 128_{10}$
$2^2 = 4_{10}$	$2^8 = 256_{10}$
$2^3 = 8_{10}$	$2^9 = 512_{10}$
$2^4 = 16_{10}$	$2^{10} = 1024_{10}$
$2^5 = 32_{10}$	$2^{11} = 2048_{10}$

Figure 3  
Condensed listing of powers of 2.

Fractional binary numbers are expressed as negative powers of 2. Figure 4 provides a condensed listing of negative powers of 2. In positional notation, the binary number 0.1101 can be expressed as follows:

$$(1 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$$

To determine the decimal value of the binary number 0.1101, multiply each bit by its positional weight and add the results.

$$(1 \times 1/2) + (1 \times 1/4) + (0 \times 1/8) + (1 \times 1/16) = \\ 0.5 + 0.25 + 0 + 0.0625 = 0.8125_{10}$$

In the binary number system, the radix point is called the binary point.

$$2^{-1} = \frac{1}{2} = 0.5_{10}$$

$$2^{-2} = \frac{1}{4} = 0.25_{10}$$

$$2^{-3} = \frac{1}{8} = 0.125_{10}$$

$$2^{-4} = \frac{1}{16} = 0.0625_{10}$$

$$2^{-5} = \frac{1}{32} = 0.03125_{10}$$

$$2^{-6} = \frac{1}{64} = 0.015625_{10}$$

$$2^{-7} = \frac{1}{128} = 0.0078125_{10}$$


$$2^{-8} = \frac{1}{256} = 0.00390625_{10}$$

Figure 4  
Condensed listing of negative  
powers of 2.

## Converting Between the Binary and Decimal Number Systems

**Binary to Decimal** To convert a binary number into its decimal equivalent, add together the weights of the positions in the number where binary 1's occur. The weights of the integer and fractional positions are indicated below.

INTEGER								FRACTIONAL		
$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$
128	64	32	16	8	4	2	1	.5	.25	.125

Binary Point 

As an example, convert the binary number 1010 into its decimal equivalent. The right-most bit, called the **least significant bit** or LSB, has the lowest integer weight of  $2^0 = 1$ . The left-most bit is the **most significant bit** (MSB) because it carries the greatest weight in determining the value of the number. In this example, it has a weight of  $2^3 = 8$ . To evaluate the number, add together the weights of the positions where binary 1's appear. In this example, 1's occur in the  $2^3$  and  $2^1$  positions. The decimal equivalent is ten.

Binary Number	1	0	1	0	
Position Weights	$2^3$	$2^2$	$2^1$	$2^0$	
Decimal Equivalent	8	+ 0	+ 2	+ 0	= $10_{10}$

**Decimal to Binary** A decimal integer number can be converted to a different base or radix through successive divisions by the desired base. To convert a decimal integer number to its binary equivalent, successively divide the number by 2 and note the remainders. When you divide by 2, the remainder will always be 1 or 0.

The remainders form the equivalent binary number.

As an example, the decimal number 25 is converted into its binary equivalent.

$$\begin{array}{rll}
 25 \div 2 = 12 & \text{with remainder } 1 & \leftarrow \text{LSB} \\
 12 \div 2 = 6 & & 0 \\
 6 \div 2 = 3 & & 0 \\
 3 \div 2 = 1 & & 1 \\
 1 \div 2 = 0 & & 1 \leftarrow \text{MSB}
 \end{array}$$

Divide the decimal number by 2 and note the remainder. Then divide the quotient by 2 and again note the remainder. Then divide the quotient by 2 and again note the remainder. Continue this division process until 0 results. Then collect remainders beginning with the last or most significant bit (MSB) and proceed to the first or least significant bit (LSB). The number  $11001_2 = 25_{10}$ . Notice that the remainders are collected in the reverse order. That is, the first remainder becomes the least significant bit, while the last remainder becomes the most significant bit.

**NOTE:** Do not attempt to use a calculator to perform this conversion. It would only supply you with confusing results.

To convert a decimal fraction to a different base or radix, multiply the fraction successively by the desired base and record any integers produced by the multiplication as an overflow. For example, to convert the decimal fraction 0.3125 into its binary equivalent, multiply repeatedly by 2.

$$\begin{array}{rll}
 0.3125 \times 2 = 0.625 = 0.625 & \text{with overflow } 0 & \leftarrow \text{MSB} \\
 0.6250 \times 2 = 1.250 = 0.250 & & 1 \\
 0.2500 \times 2 = 0.500 = 0.500 & & 0 \\
 0.5000 \times 2 = 1.000 = 0 & & 1 \leftarrow \text{LSB}
 \end{array}$$

These multiplications will result in numbers with a 1 or 0 in the units position (the position to the left of the decimal point). By recording the value of the units position, you can construct the equivalent binary fraction. This units position value is called the "overflow." Therefore, when 0.3125 is multiplied by 2, the overflow is 0. This becomes the most significant bit (MSB) of the binary equivalent fraction. Then 0.625 is multiplied by 2. Since the product is 1.25, the overflow is 1. When there is an overflow of 1, it is effectively subtracted from the product when the value is recorded. Therefore, only 0.25 is multiplied by 2 in the next multiplication process. This method continues until an overflow with no fraction results. It is important to note that you can not always obtain 0 when you multiply by 2. Therefore, you should only continue the conver-



sion process to the precision you desire. Collect the conversion overflows beginning at the radix (binary) point with the MSB and proceed to the LSB. This is the same order in which the overflows were produced. The number  $0.0101_2 = 0.3125_{10}$ .

If the decimal number contains both an integer and fraction, you must separate the integer and fraction using the decimal point as the break point. Then perform the appropriate conversion process on each number portion. After you convert the binary integer and binary fraction, recombine them. For example, the decimal number 14.375 is converted into its binary equivalent.

$$14.375_{10} = 14_{10} + 0.375_{10}$$

$14 \div 2 = 7$	with remainder	0	←	LSB
$7 \div 2 = 3$		1		
$3 \div 2 = 1$		1		
$1 \div 2 = 0$		1	←	MSB

$$\boxed{14_{10} = 1110_2}$$

$0.375 \times 2 = 0.75 = 0.75$	with overflow	0	←	MSB
$0.750 \times 2 = 1.50 = 0.50$		1		
$0.500 \times 2 = 1.00 = 0$		1	←	LSB

$$\boxed{0.375_{10} = 0.011_2}$$

$$14.375_{10} = 14_{10} + 0.375_{10} = 1110_2 + 0.011_2 = 1110.011_2.$$

## HEXADECIMAL NUMBER SYSTEM

Hexadecimal is another number system that is often used with microprocessors. As the name implies, hexadecimal has a base (radix) of  $16_{10}$ . It uses the digits 0 through 9 and the letters A through F.

The letters are used because it is necessary to represent  $16_{10}$  different values with a single digit for each value. Therefore, the letters A through F are used to represent the number values  $10_{10}$  through  $15_{10}$ . The following discussion will compare the decimal number system with the hexadecimal number system.

All of the numbers are of equal value between systems ( $0_{10} = 0_{16}$ ,  $3_{10} = 3_{16}$ ,  $9_{10} = 9_{16}$ , etc.). For numbers greater than 9, this relationship exists:  $10_{10} = A_{16}$ ,  $11_{10} = B_{16}$ ,  $12_{10} = C_{16}$ ,  $13_{10} = D_{16}$ ,  $14_{10} = E_{16}$ , and  $15_{10} = F_{16}$ . Using letters in counting may appear awkward until you become familiar with the system. Figure 8 illustrates the relationship between decimal, hexadecimal, and binary integers, while Figure 9 illustrates the relationship between decimal, hexadecimal, and binary fractions.

DECIMAL	HEXADECIMAL	BINARY
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000

Figure 8  
Sample comparison of decimal,  
hexadecimal, and binary integers.

DECIMAL	HEXADECIMAL	BINARY
0.00390625	0.01	0.00000001
0.0078125	0.02	0.0000001
0.01171875	0.03	0.00000011
0.015625	0.04	0.000001
0.01953125	0.05	0.00000101
0.0234375	0.06	0.0000011
0.02734375	0.07	0.00000111
0.03125	0.08	0.00001
0.03515625	0.09	0.00001001
0.0390625	0.0A	0.0000101
0.04296875	0.0B	0.00001011
0.046875	0.0C	0.00011
0.05078125	0.0D	0.00001101
0.0546875	0.0E	0.0000111
0.05859375	0.0F	0.00001111

Figure 9  
Sample comparison of decimal,  
hexadecimal, and binary fractions.

As with the previous number systems, each digit position of a hexadecimal number carries a positional weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example,  $16_{10}$ ). The total quantity of a number can be evaluated by considering the specific digits and the weights of their positions. (Refer to Figure 10 for a condensed listing of powers of  $16_{10}$ .) For example, the hexadecimal number E5D7.A3 can be written with positional notation as follows:

$$(E \times 16^3) + (5 \times 16^2) + (D \times 16^1) + (7 \times 16^0) + (A \times 16^{-1}) + (3 \times 16^{-2})$$

The decimal value of the hexadecimal number E5D7.A3 is determined by multiplying each digit by its positional weight and adding the results. As with the previous number systems, the radix (hexadecimal) point separates the integer from the fractional part of the number.

$$\begin{aligned} (14 \times 4096) + (5 \times 256) + (13 \times 16) + (7 \times 1) + (10 \times 1/16) + (3 \times 1/256) = \\ 57344 + 1280 + 208 + 7 + 0.625 + 0.01171875 = \\ 58839.63671875_{10} \end{aligned}$$

$$\begin{aligned} 16^{-4} &= \frac{1}{65536} = 0.0000152587890625_{10} \\ 16^{-3} &= \frac{1}{4096} = 0.000244140625_{10} \\ 16^{-2} &= \frac{1}{256} = 0.00390625_{10} \\ 16^{-1} &= \frac{1}{16} = 0.0625_{10} \end{aligned}$$

$$\begin{aligned} 1_{10} &= 16^0 \\ 16_{10} &= 16^1 \\ 256_{10} &= 16^2 \\ 4096_{10} &= 16^3 \\ 65536_{10} &= 16^4 \\ 1048576_{10} &= 16^5 \\ 16777216_{10} &= 16^6 \end{aligned}$$

Figure 10  
Condensed listing of powers of 16.

## Conversion From Decimal to Hexadecimal

Decimal to hexadecimal conversion is accomplished in the same manner as decimal to binary, but with a base number of  $16_{10}$ . As an example, the decimal number 156 is converted into its hexadecimal equivalent.

$$\begin{array}{rcl} 156 \div 16 = 9 & \text{with remainder } 12 = C & \leftarrow \text{LSD} \\ 9 \div 16 = 0 & & 9 = 9 \leftarrow \text{MSD} \end{array}$$

Divide the decimal number by  $16_{10}$  and note the remainder. If the remainder exceeds 9, convert the 2-digit number to its hexadecimal equivalent ( $12_{10} = C$  in this example). Then divide the quotient by 16 and again note the remainder. Continue dividing until a quotient of 0 results. Then collect the remainders beginning with the last or most significant digit (MSD) and proceed to the first or least significant digit (LSD). The number  $9C_{16} = 156_{10}$ . NOTE: The letter H after a number is sometimes used to indicate hexadecimal.

To convert a decimal fraction to a hexadecimal fraction, multiply the fraction successively by  $16_{10}$  (hexadecimal base). As an example the decimal fraction 0.78125 is converted into its hexadecimal equivalent.

$$\begin{array}{rcl} 0.78125 \times 16 = 12.5 = 0.5 & \text{with overflow } 12 = C & \leftarrow \text{MSD} \\ 0.50000 \times 16 = 8.0 = 0 & & 8 = 8 \leftarrow \text{LSD} \end{array}$$

Multiply the decimal by  $16_{10}$ . If the product exceeds one, subtract the integer (overflow) from the product. If the "overflow" exceeds 9, convert the 2-digit number to its hexadecimal equivalent. Then multiply the product fraction by  $16_{10}$  and again note any overflow. Continue multiplying until an overflow, with 0 for a fraction, results. Remember, you can not always obtain 0 when you multiply by 16. Therefore, you should only continue the conversion to the precision you desire. Collect the conversion overflows beginning at the radix point with the MSD and proceed to the LSD. The number  $0.C8_{16} = 0.78125_{10}$ .

As shown in this section, conversion of an integer from decimal to hexadecimal requires a different technique than for conversion of a fraction. Therefore, when you convert a hexadecimal number composed of an integer and a fraction, you must separate the integer and fraction, then perform the appropriate operation on each. After you convert them, you must recombine the integer and fraction. For example, the decimal number 124.78125 is converted into its hexadecimal equivalent.

$$\begin{aligned}
 124.78125_{10} &= 124_{10} + 0.78125_{10} \\
 124 \div 16 &= 7 \quad \text{with remainder } 12 = \text{C} \quad \leftarrow \text{LSD} \\
 7 \div 16 &= 0 \quad \quad \quad \quad \quad \quad \quad 7 = 7 \quad \leftarrow \text{MSD}
 \end{aligned}$$

$$\boxed{124_{10} = 7\text{C}_{16}}$$

$$\begin{aligned}
 0.78125 \times 16 &= 12.5 = 0.5 \quad \text{overflow} \quad 12 = \text{C} \quad \leftarrow \text{MSD} \\
 0.50000 \times 16 &= 8.0 = 0 \quad \quad \quad \quad \quad \quad \quad 8 = 8 \quad \leftarrow \text{LSD}
 \end{aligned}$$

$$\boxed{0.78125_{10} = 0.\text{C8}_{16}}$$

$$124.78125_{10} = 124_{10} + 0.78125_{10} = 7\text{C}_{16} + 0.\text{C8}_{16} = 7\text{C}.\text{C8}_{16}$$

First separate the decimal integer and fraction. Then convert the integer and fraction to hexadecimal.

Finally, recombine the integer and fraction.

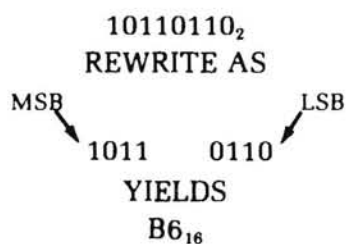
## Converting Between the Hexadecimal and Binary Number Systems

The hexadecimal number system is an excellent shorthand form to express large binary quantities. Figures 8 and 9 illustrate the relationship between hexadecimal and binary integers and fractions.

As you know, four bits of a binary number exactly equal  $16_{10}$  value combinations. Therefore, you can represent a 4-bit binary number with a 1-digit hexadecimal number:

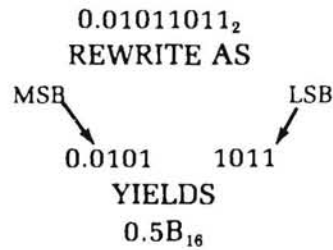
$$1101_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13_{10} = D_{16}$$

Because of this relationship, converting binary to hexadecimal is simple and straightforward. For example, binary number 10110110 is converted into its hexadecimal equivalent.



To convert a binary number to hexadecimal, first separate the number into groups containing four bits, beginning with the least significant bit. Then convert each 4-bit group into its hexadecimal equivalent. Don't forget to use letter digits as required. This gives you a hexadecimal number equal in value to the binary number.

Binary fractions can also be converted to their hexadecimal equivalents using the same process, with one exception; the binary bits are separated into groups of four, beginning with the most significant bit (at the radix point). For example, the binary fraction  $0.01011011_2$  is converted into its hexadecimal equivalent.



Again, you must separate the binary number into groups of four, beginning with the radix point. Then convert each 4-bit group into its hexadecimal equivalent. This gives you a hexadecimal number equal in value to the binary number.

As with octal to binary conversions, you may add zeros to fill out the binary number for calculation. However, when you add zeros to a binary integer, place them to the left of the MSB. In a binary fraction, zeros are placed to the right of the LSB. Never, under any circumstances, move the radix to perform conversions.



Now, a binary number containing both an integer and a fraction ( $110110101.01110111_2$ ) will be converted into its hexadecimal equivalent.

$$\begin{array}{r}
 110110101.01110111_2 \\
 \text{REWRITE AS} \\
 \begin{array}{cccc}
 \text{MSB} \swarrow & & & \nwarrow \text{LSB} \\
 \underline{0001} & 1011 & 0101.0111 & 0111
 \end{array} \\
 \text{YIELDS} \\
 1B5.77_{16}
 \end{array}$$

The integer part of the number is separated into groups of four, **beginning** at the radix point. Note that three zeros were added to the third group to complete the group. The fractional part of the number is separated into groups of four, **beginning** at the radix point. (No zeros were needed to complete the fractional groups.) The integer and fractional 4-bit groups are then converted to hexadecimal. The number  $110110101.01110111_2 = 1B5.77_{16}$ . **Never** shift the radix point in order to form 4-bit groups.

Converting hexadecimal to binary is just the opposite of the previous process; simply convert each hexadecimal number into its 4-bit binary equivalent. For example, convert the hexadecimal number  $8F.41_{16}$  into its binary equivalent.

$$\begin{array}{r}
 8F.41_{16} \\
 \text{YIELDS} \\
 \begin{array}{ccc}
 \text{MSB} \swarrow & & \nwarrow \text{LSB} \\
 1000 & 1111.0100 & 0001
 \end{array} \\
 \text{REWRITE AS} \\
 10001111.01000001_2
 \end{array}$$

Convert each hexadecimal digit into a 4-bit binary number. Then condense the 4-bit groups to form the binary value equal to the hexadecimal value. The number  $8F.41_{16} = 10001111.01000001_2$ .

## BINARY CODES

Converting a decimal number into its binary equivalent is called “coding.” A decimal number is expressed as a binary code or binary number. The **binary number system**, as discussed, is known as the pure binary code. This name distinguishes it from other types of binary codes. This section will discuss some of the other types of binary codes used in computers.

### Binary Coded Decimal

It is difficult to quickly glance at a binary number and recognize its decimal equivalent. For example, the binary number 1010011 represents the decimal number 83. However, within a few minutes, using the procedures described earlier, you could readily calculate its decimal value. The amount of time it takes to convert or recognize a binary number quantity is a distinct disadvantage in working with this code despite the numerous hardware advantages. Engineers recognized this problem early and developed a special form of binary code that was more compatible with the decimal system. This special compromise code is known as binary coded decimal (BCD). The BCD code combines some of the characteristics of both the binary and decimal number systems.

**8421 BCD Code** The BCD code is a system of representing the decimal digits 0 through 9 with a 4-bit binary code. This BCD code uses the standard 8421 position **weighting system** of the pure binary code. The standard 8421 BCD code and the decimal equivalents and binary are shown in Figure 11. As with the pure binary code, you can convert the BCD numbers into their decimal equivalents by simply adding together the weights of the bit positions whereby the binary 1's occur. Note, however, that there are only ten possible valid 4-bit code arrangements. The 4-bit binary numbers representing the decimal numbers 10 through 15 are invalid in the BCD system.

DECIMAL	8421 BCD	BINARY
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001
10	0001 0000	1010
11	0001 0001	1011
12	0001 0010	1100
13	0001 0011	1101
14	0001 0100	1110
15	0001 0101	1111

Figure 11  
Codes.

To represent a decimal number in BCD notation, substitute the appropriate 4-bit code for each decimal digit. For example, the decimal integer 834 in BCD would be 1000 0011 0100. Each decimal digit is represented by its equivalent 8421 4-bit code. A space is left between each 4-bit group to avoid confusing the BCD format with the pure binary code. This method of representation also applies to decimal fractions. For example, the decimal fraction 0.764 would be 0.0111 0110 0100 in BCD. Again, each decimal digit is represented by its equivalent 8421 4-bit code, with a space between each group.

The BCD code simplifies the man-machine interface but it is less efficient than the pure binary code. It takes more bits to represent a given decimal number in BCD than it does with pure binary notation. For example, the decimal number 83 in pure binary form is 1010011. In BCD code the decimal number 83 is written as 1000 0011.

Decimal-to-BCD conversion is simple and straightforward. However, binary-to-BCD conversion is not direct. An intermediate conversion to decimal must be performed first. For example, the binary number 1011.01 is converted into its BCD equivalent.

First the binary number is converted to decimal.

$$\begin{aligned} 1011.01_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) \\ &= 8 + 0 + 2 + 1 + 0 + 0.25 \\ &= 11.25_{10} \end{aligned}$$

Then the decimal result is converted to BCD.

$$11.25_{10} = 0001\ 0001.0010\ 0101$$

To convert from BCD to binary, the previous operation is reversed. For example, the BCD number 1001 0110.0110 0010 0101 is converted into its binary equivalent.

First, the BCD number is converted to decimal.

$$1001\ 0110.0110\ 0010\ 0101 = 96.625_{10}$$

Then the decimal result is converted to binary.

$$96.625_{10} = 96_{10} + 0.625_{10}$$

$96 \div 2 = 48$	with remainder	0	←	LSB
$48 \div 2 = 24$		0		
$24 \div 2 = 12$		0		
$12 \div 2 = 6$		0		
$6 \div 2 = 3$		0		
$3 \div 2 = 1$		1		
$1 \div 2 = 0$		1	←	MSB

$$96_{10} = 1100000_2$$

$0.625 \times 2 = 1.25$	$= 0.25$	with overflow	1	←	MSB
$0.250 \times 2 = 0.50$	$= 0.50$		0		
$0.500 \times 2 = 1.00$	$= 0$		1	←	LSB

$$0.625_{10} = 0.101_2$$

$$96.625_{10} = 96_{10} + 0.625_{10} = 1100000_2 + 0.101_2 = 1100000.101_2$$

Therefore:

$$1001\ 0110.0110\ 0010\ 0101 = 96.625_{10} = 1100000.101_2$$

Because the intermediate decimal number contains both an integer and fraction, each number portion is converted as described under "Binary Number System." The binary sum (integer plus fraction) 1100000.101 is equivalent to the BCD number 1001 0110.0110 0010 0101.

## Alphanumeric Codes

Several binary codes are called alphanumeric codes because they are used to represent characters as well as numbers. The most common of these codes, ASCII, will be discussed here.

**ASCII Code** The American Standard Code for Information Interchange commonly referred to as ASCII, is a special form of binary code that is widely used in microprocessors and data communications equipment. ASCII is binary code that is used in transferring data between microprocessors and their peripheral devices, and in communicating data by radio and telephone. A 7-bit code called full ASCII can be represented by  $2^7 = 128$  different characters. In addition to the characters and numbers generated by 6-bit ASCII, 7-bit ASCII contains lower-case letters of the alphabet, and additional characters for punctuation and control. The 7-bit ASCII code is shown in Figure 12.

COLUMN	0 <sup>(3)</sup>	1 <sup>(3)</sup>	2 <sup>(3)</sup>	3	4	5	6	7 <sup>(3)</sup>	
ROW	BITS 4321 765	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	^ <sup>(1)</sup>	n	~
15	1111	SI	US	/	?	O	_ <sup>(2)</sup>	o	DEL

Figure 12

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
LF	Line Feed	SUB	Substitute
VT	Vertical Tabulation	ESC	Escape
FF	Form Feed	FS	File Separator
CR	Carriage Return	GS	Group Separator
SO	Shift Out	RS	Record Separator
SI	Shift In	US	Unit Separator
SP	Space (blank)	DEL	Delete

Figure 12  
(Continued.)

The 7-bit ASCII code for each number, letter or control function is made up of a 4-bit group and a 3-bit group. Figure 13 shows the arrangement of these two groups and the numbering sequence. The 4-bit group is on the right and bit 1 is the LSB. Note how these groups are arranged in rows and columns in Figure 12.

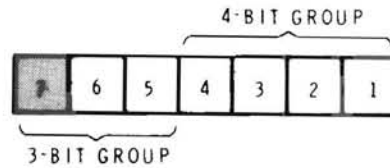


Figure 13

ASCII code word format.

To determine the ASCII code for a given number letter or control operation, locate that item in the table. Then use the 3- and 4-bit codes associated with the row and column in which the item is located. For example, the ASCII code for the letter L is 1001100. It is located in column 4, row 12. The most significant 3-bit group is 100, while the least significant 4-bit group is 1100. When 6-bit ASCII is used, the 3-bit group is reduced to a 2-bit group as shown in Figure 14.

In 7-bit ASCII code, an eighth bit is often used as a **parity** or check bit to determine if the data (character) has been transmitted correctly. The value of this bit is determined by the type of parity desired. **Even parity** means the sum of all the 1 bits, including the parity bit, is an even number. For example, if G is the character transmitted, the ASCII code is 1000111. Since four 1's are in the code, the parity bit is 0. The 8-bit code would be written 01000111.

**Odd Parity** means the sum of all the 1 bits, including the parity bit, is an odd number. If the ASCII code for G was transmitted with odd parity, the binary representation would be 11000111.



		COLUMN	0	1	2	3
ROW	BITS 4321	65	10	11	00	01
0	0000		SP <sup>(3)</sup>	0	@	P
1	0001		!	1	A	Q
2	0010		"	2	B	R
3	0011		#	3	C	S
4	0100		\$	4	D	T
5	0101		%	5	E	U
6	0110		&	6	F	V
7	0111		'	7	G	W
8	1000		(	8	H	X
9	1001		)	9	I	Y
10	1010		*	:	J	Z
11	1011		+	;	K	
12	1100		,	<	L	\
13	1101		-	=	M	
14	1110		.	>	N	⌒ <sup>(1)</sup>
15	1111		/	?	O	— <sup>(2)</sup>

Figure 14  
Table of 6-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) SP — Space (blank) for machine control.



*Appendix B*

**MACHINE CODING INSTRUCTIONS**

Data supplied courtesy of  
Intel Corporation.

---

## HARDWARE REFERENCE INFORMATION

---

### Machine Instruction Encoding and Decoding

Writing a MOV instruction in ASM-86 in the form:

MOV destination,source

will cause the assembler to generate 1 of 28 possible forms of the MOV machine instruction. A programmer rarely needs to know the details of machine instruction formats or encoding. An exception may occur during debugging when it may be necessary to monitor instructions fetched on the bus, read unformatted memory dumps, etc. This section provides the information necessary to translate or decode an 8086 or 8088 machine instruction.

To pack instructions into memory as densely as possible, the 8086 and 8088 CPUs utilize an efficient coding technique. Machine instructions vary from one to six bytes in length. One-byte instructions, which generally operate on single registers or flags, are simple to identify. The keys to decoding longer instructions are in the first two bytes. The format of these bytes can vary, but most instructions follow the format shown in figure 4-20.

The first six bits of a multibyte instruction generally contain an opcode that identifies the basic instruction type: ADD, XOR, etc. The following bit, called the D field, generally specifies the "direction" of the operation: 1 = the REG field in the second byte identifies the destination operand, 0 = the REG field identifies the source operand. The W field distinguishes between byte and word operations: 0 = byte, 1 = word.

One of three additional single-bit fields, S, V or Z, appears in some instruction formats. S is used in conjunction with W to indicate sign extension

**HARDWARE REFERENCE INFORMATION**

of immediate fields in arithmetic instructions. V distinguishes between single- and variable-bit shifts and rotates. Z is used as a compare bit with

the zero flag in conditional repeat and loop instructions. All single-bit field settings are summarized in table 4-7.

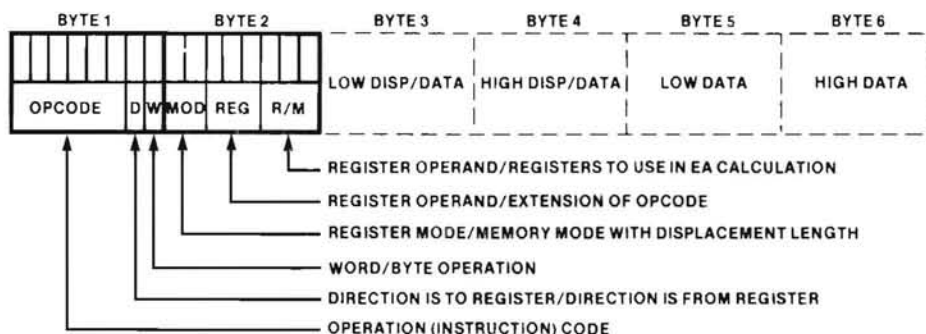


Figure 4-20. Typical 8086/8088 Machine Instruction Format

Table 4-7. Single-Bit Field Encoding

Field	Value	Function
S	0	No sign extension
	1	Sign extend 8-bit immediate data to 16 bits if W=1
W	0	Instruction operates on byte data
	1	Instruction operates on word data
D	0	Instruction source is specified in REG field
	1	Instruction destination is specified in REG field
V	0	Shift/rotate count is one
	1	Shift/rotate count is specified in CL register
Z	0	Repeat/loop while zero flag is clear
	1	Repeat/loop while zero flag is set

HARDWARE REFERENCE INFORMATION

The second byte of the instruction usually identifies the instruction's operands. The MOD (mode) field indicates whether one of the operands is in memory or whether both operands are registers (see table 4-8). The REG (register) field identifies a register that is one of the instruction operands (see table 4-9). In a number of instructions, chiefly the immediate-to-memory variety, REG is used as an extension of the opcode to identify the type of operation. The encoding of the R/M (register/memory) field (see table 4-10) depends on how the mode field is set. If MOD = 11 (register-to-register mode), then R/M identifies the second register operand. If MOD selects memory mode, then R/M indicates how the effective address of the memory operand is to be calculated. Effective address calculation is covered in detail in section 2.8.

Bytes 3 through 6 of an instruction are optional fields that usually contain the displacement value of a memory operand and/or the actual value of an immediate constant operand.

Table 4-8. MOD (Mode) Field Encoding

CODE	EXPLANATION
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

\*Except when R/M = 110, then 16-bit displacement follows

Table 4-9. REG (Register) Field Encoding

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

There may be one or two displacement bytes; the language translators generate one byte whenever possible. The MOD field indicates how many displacement bytes are present. Following Intel convention, if the displacement is two bytes, the most-significant byte is stored second in the instruction. If the displacement is only a single byte, the 8086 or 8088 automatically sign-extends this quantity to 16-bits before using the information in further address calculations. Immediate values always follow any displacement values that may be present. The second byte of a two-byte immediate value is the most significant.

Table 4-12 lists the instruction encodings for all 8086/8088 instructions. This table can be used to predict the machine encoding of any ASM-86 instruction. Table 4-13 lists the 8086/8088 machine instructions in order by the binary value of their first byte. This table can be used to decode any machine instruction from its binary representation. Table 4-11 is a key to the abbreviations used in tables 4-12 and 4-13. Table 4-14 is a more compact instruction decoding guide.

Table 4-10. R/M (Register/Memory) Field Encoding

MOD = 11			EFFECTIVE ADDRESS CALCULATION			
R/M	W = 0	W = 1	R/M	MOD = 00	MOD = 01	MOD = 10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX)	(BX) + D8	(BX) + D16

## HARDWARE REFERENCE INFORMATION

Table 4-11. Key to Machine Instruction Encoding and Decoding

IDENTIFIER	EXPLANATION
MOD	Mode field; described in this chapter.
REG	Register field; described in this chapter.
R/M	Register/Memory field; described in this chapter.
SR	Segment register code: 00=ES, 01=CS, 10=SS, 11=DS.
W, S, D, V, Z	Single-bit instruction fields; described in this chapter.
DATA-8	8-bit immediate constant.
DATA-SX	8-bit immediate value that is automatically sign-extended to 16-bits before use.
DATA-LO	Low-order byte of 16-bit immediate constant.
DATA-HI	High-order byte of 16-bit immediate constant.
(DISP-LO)	Low-order byte of optional 8- or 16-bit unsigned displacement; MOD indicates if present.
(DISP-HI)	High-order byte of optional 16-bit unsigned displacement; MOD indicates if present.
IP-LO	Low-order byte of new IP value.
IP-HI	High-order byte of new IP value
CS-LO	Low-order byte of new CS value.
CS-HI	High-order byte of new CS value.
IP-INC8	8-bit signed increment to instruction pointer.
IP-INC-LO	Low-order byte of signed 16-bit instruction pointer increment.
IP-INC-HI	High-order byte of signed 16-bit instruction pointer increment.
ADDR-LO	Low-order byte of direct address (offset) of memory operand; EA not calculated.
ADDR-HI	High-order byte of direct address (offset) of memory operand; EA not calculated.
--	Bits may contain any value.
XXX	First 3 bits of ESC opcode.
YYY	Second 3 bits of ESC opcode.
REG8	8-bit general register operand.
REG16	16-bit general register operand.
MEM8	8-bit memory operand (any addressing mode).
MEM16	16-bit memory operand (any addressing mode).
IMMED8	8-bit immediate operand.
IMMED16	16-bit immediate operand.
SEGREG	Segment register operand.
DEST-STR8	Byte string addressed by DI.

**HARDWARE REFERENCE INFORMATION**

**Table 4-11. Key to Machine Instruction Encoding and Decoding (Cont'd.)**

IDENTIFIER	EXPLANATION
SRC-STR8	Byte string addressed by SI.
DEST-STR16	Word string addressed by DI.
SRC-STR16	Word string addressed by SI.
SHORT-LABEL	Label within ±127 bytes of instruction.
NEAR-PROC	Procedure in current code segment.
FAR-PROC	Procedure in another code segment.
NEAR-LABEL	Label in current code segment but farther than -128 to +127 bytes from instruction.
FAR-LABEL	Label in another code segment.
SOURCE-TABLE	XLAT translation table addressed by BX.
OPCODE	ESC opcode operand.
SOURCE	ESC register or memory operand.

**Table 4-12. 8086 Instruction Encoding**

**DATA TRANSFER**

**MOV = Move:**

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory to/from register	1 0 0 0 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w = 1
Immediate to register	1 0 1 1 w reg	data	data if w = 1			
Memory to accumulator	1 0 1 0 0 0 w	addr-lo	addr-hi			
Accumulator to memory	1 0 1 0 0 0 1 w	addr-lo	addr-hi			
Register/memory to segment register	1 0 0 0 1 1 1 0	mod 0 SR r/m	(DISP-LO)	(DISP-HI)		
Segment register to register/memory	1 0 0 0 1 1 0 0	mod 0 SR r/m	(DISP-LO)	(DISP-HI)		

**PUSH = Push:**

Register/memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m	(DISP-LO)	(DISP-HI)
Register	0 1 0 1 0 reg			
Segment register	0 0 0 reg 1 1 0			

**POP = Pop:**

Register/memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)
Register	0 1 0 1 1 reg			
Segment register	0 0 0 reg 1 1 1			



**HARDWARE REFERENCE INFORMATION**

**Table 4-12. 8086 Instruction Encoding (Cont'd.)**

**DATA TRANSFER (Cont'd.)**

**XCHG = Exchange:**

Register/memory with register

Register with accumulator

7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
1 0 0 0 0 1 1 w	mod reg r/m	(DISP-LO)	(DISP-HI)		
1 0 0 1 0 reg					

**IN = Input from:**

Fixed port

Variable port

1 1 1 0 0 1 0 w	DATA-8
1 1 1 0 1 1 0 w	

**OUT = Output to:**

Fixed port

Variable port

**XLAT = Translate byte to AL**

**LEA = Load EA to register**

**LDS = Load pointer to DS**

**LES = Load pointer to ES**

**LAHF = Load AH with flags**

**SAHF = Store AH into flags**

**PUSHF = Push flags**

**POPF = Pop flags**

1 1 1 0 0 1 1 w	DATA-8		
1 1 1 0 1 1 1 w			
1 1 0 1 0 1 1 1			
1 0 0 0 1 1 0 1	mod reg r/m	(DISP-LO)	(DISP-HI)
1 1 0 0 0 1 0 1	mod reg r/m	(DISP-LO)	(DISP-HI)
1 1 0 0 0 1 0 0	mod reg r/m	(DISP-LO)	(DISP-HI)
1 0 0 1 1 1 1 1			
1 0 0 1 1 1 1 0			
1 0 0 1 1 1 0 0			
1 0 0 1 1 1 0 1			

**ARITHMETIC**

**ADD = Add:**

Reg/memory with register to either

Immediate to register/memory

Immediate to accumulator

0 0 0 0 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
1 0 0 0 0 0 s w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if s = 01
0 0 0 0 0 1 0 w	data	data if w = 1			

**ADC = Add with carry:**

Reg/memory with register to either

Immediate to register/memory

Immediate to accumulator

0 0 0 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
1 0 0 0 0 0 s w	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)	data	data if s = 01
0 0 0 1 0 1 0 w	data	data if w = 1			

**INC = Increment:**

Register/memory

Register

**AAA = ASCII adjust for add**

**DAA = Decimal adjust for add**

1 1 1 1 1 1 1 w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)
0 1 0 0 0 reg			
0 0 1 1 0 1 1 1			
0 0 1 0 0 1 1 1			

HARDWARE REFERENCE INFORMATION

Table 4-12. 8086 Instruction Encoding (Cont'd.)

ARITHMETIC (Cont'd.)

**SUB** = Subtract:

Reg/memory and register to either  
 Immediate from register/memory  
 Immediate from accumulator

7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
0 0 1 0 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
1 0 0 0 0 0 s w	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=01
0 0 1 0 1 1 0 w	data	data if w=1			

**SBB** = Subtract with borrow:

Reg/memory and register to either  
 Immediate from register/memory  
 Immediate from accumulator

0 0 0 1 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
1 0 0 0 0 0 s w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=01
0 0 0 1 1 1 0 w	data	data if w=1			

**DEC** Decrement:

Register/memory

Register

**NEG** Change sign

1 1 1 1 1 1 1 w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)	
0 1 0 0 1 reg				
1 1 1 1 0 1 1 w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)	

**CMP** = Compare:

Register/memory and register  
 Immediate with register/memory  
 Immediate with accumulator

**AAS** ASCII adjust for subtract

**DAS** Decimal adjust for subtract

**MUL** Multiply (unsigned)

**IMUL** Integer multiply (signed)

**AAM** ASCII adjust for multiply

**DIV** Divide (unsigned)

**IDIV** Integer divide (signed)

**AAD** ASCII adjust for divide

**CBW** Convert byte to word

**CWD** Convert word to double word

0 0 1 1 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
1 0 0 0 0 0 s w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=1
0 0 1 1 1 1 0 w	data				
0 0 1 1 1 1 1					
0 0 1 0 1 1 1					
1 1 1 1 0 1 1 w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)		
1 1 1 1 0 1 1 w	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)		
1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0	(DISP-LO)	(DISP-HI)		
1 1 1 1 0 1 1 w	mod 1 1 0 r/m	(DISP-LO)	(DISP-HI)		
1 1 1 1 0 1 1 w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)		
1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0	(DISP-LO)	(DISP-HI)		
1 0 0 1 1 0 0 0					
1 0 0 1 1 0 0 1					

LOGIC

**NOT** Invert

**SHL/SAL** Shift logical/arithmetic left

**SHR** Shift logical right

**SAR** Shift arithmetic right

**ROL** Rotate left

1 1 1 1 0 1 1 w	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)	
1 1 0 1 0 0 v w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)	
1 1 0 1 0 0 v w	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)	
1 1 0 1 0 0 v w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)	
1 1 0 1 0 0 v w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	

## HARDWARE REFERENCE INFORMATION

Table 4-12. 8086 Instruction Encoding (Cont'd.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
LOGIC (Cont'd.)						
ROR Rotate right	1 1 0 1 0 0 v w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)		
RCL Rotate through carry flag left	1 1 0 1 0 0 v w	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)		
RCR Rotate through carry right	1 1 0 1 0 0 v w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)		
<b>AND = And:</b>						
Reg/memory with register to either	0 0 1 0 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate to accumulator	0 0 1 0 0 1 0 w	data	data if w=1			
<b>TEST = And function to flags no result:</b>						
Register/memory and register	0 0 0 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate data and register/memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate data and accumulator	1 0 1 0 1 0 0 w	data				
<b>OR = Or:</b>						
Reg/memory and register to either	0 0 0 0 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate to accumulator	0 0 0 0 1 1 0 w	data	data if w=1			
<b>XOR = Exclusive or:</b>						
Reg/memory and register to either	0 0 1 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	0 0 1 1 0 1 0 w	data	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate to accumulator	0 0 1 1 0 1 0 w	data	data if w=1			
<b>STRING MANIPULATION</b>						
REP = Repeat	1 1 1 1 0 0 1 z					
MOVS = Move byte/word	1 0 1 0 0 1 0 w					
CMPS = Compare byte/word	1 0 1 0 0 1 1 w					
SCAS = Scan byte/word	1 0 1 0 1 1 1 w					
LODS = Load byte/wd to AL/AX	1 0 1 0 1 1 0 w					
STOS = Stor byte/wd from AL/A	1 0 1 0 1 0 1 w					

HARDWARE REFERENCE INFORMATION

Table 4-12. 8086 Instruction Encoding (Cont'd.)

CONTROL TRANSFER

CALL = Call: 7 8 5 4 3 2 1 0 7 8 5 4 3 2 1 0 7 8 5 4 3 2 1 0 7 8 5 4 3 2 1 0 7 8 5 4 3 2 1 0

Direct within segment	1 1 1 0 1 0 0 0	IP-INC-LO	IP-INC-HI	
Indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)
Direct intersegment	1 0 0 1 1 0 1 0	IP-lo	IP-hi	
		CS-lo	CS-hi	
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)

JMP = Unconditional Jump:

Direct within segment	1 1 1 0 1 0 0 1	IP-INC-LO	IP-INC-HI	
Direct within segment-short	1 1 1 0 1 0 1 1	IP-INC8		
Indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)
Direct intersegment	1 1 1 0 1 0 1 0	IP-lo	IP-hi	
		CS-lo	CS-hi	
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)

RET = Return from CALL:

Within segment	1 1 0 0 0 0 1 1		
Within seg adding immediate to SP	1 1 0 0 0 0 1 0	data-lo	data-hi
Intersegment	1 1 0 0 1 0 1 1		
Intersegment adding immediate to SP	1 1 0 0 1 0 1 0	data-lo	data-hi
JE/JZ = Jump on equal/zero	0 1 1 1 0 1 0 0	IP-INC8	
JL/JNGE = Jump on less/not greater or equal	0 1 1 1 1 1 0 0	IP-INC8	
JLE/JNG = Jump on less or equal/not greater	0 1 1 1 1 1 1 0	IP-INC8	
JB/JNAE = Jump on below/not above or equal	0 1 1 1 0 0 1 0	IP-INC8	
JBE/JNA = Jump on below or equal/not above	0 1 1 1 0 1 1 0	IP-INC8	
JP/JPE = Jump on parity/parity even	0 1 1 1 1 0 1 0	IP-INC8	
JO = Jump on overflow	0 1 1 1 0 0 0 0	IP-INC8	
JS = Jump on sign	0 1 1 1 1 0 0 0	IP-INC8	
JNE/JNZ = Jump on not equal/not zero	0 1 1 1 0 1 0 1	IP-INC8	
JNL/JGE = Jump on not less/greater or equal	0 1 1 1 1 1 0 1	IP-INC8	
JNLE/JG = Jump on not less or equal/greater	0 1 1 1 1 1 1 1	IP-INC8	
JNB/JAE = Jump on not below/above or equal	0 1 1 1 0 0 1 1	IP-INC8	
JNBE/JA = Jump on not below or equal/above	0 1 1 1 0 1 1 1	IP-INC8	
JNP/JPO = Jump on not par/par odd	0 1 1 1 1 0 1 1	IP-INC8	
JNO = Jump on not overflow	0 1 1 1 0 0 0 1	IP-INC8	

**HARDWARE REFERENCE INFORMATION**

**Table 4-12. 8086 Instruction Encoding (Cont'd.)**

**CONTROL TRANSFER (Cont'd.)**

<b>RET = Return from CALL:</b>	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
<b>JNS = Jump on not sign</b>	0 1 1 1 1 0 0 1					IP-INC8
<b>LOOP = Loop CX times</b>	1 1 1 0 0 0 1 0					IP-INC8
<b>LOOPZ/LOOPE = Loop while zero/equal</b>	1 1 1 0 0 0 0 1					IP-INC8
<b>LOOPNZ/LOOPNE = Loop while not zero/equal</b>	1 1 1 0 0 0 0 0					IP-INC8
<b>JCXZ = Jump on CX zero</b>	1 1 1 0 0 0 1 1					IP-INC8

**INT = Interrupt:**

Type specified	1 1 0 0 1 1 0 1	DATA-8
Type 3	1 1 0 0 1 1 0 0	
<b>INTO = Interrupt on overflow</b>	1 1 0 0 1 1 1 0	
<b>IRET = Interrupt return</b>	1 1 0 0 1 1 1 1	

**PROCESSOR CONTROL**

<b>CLC = Clear carry</b>	1 1 1 1 1 0 0 0			
<b>CMC = Complement carry</b>	1 1 1 1 0 1 0 1			
<b>STC = Set carry</b>	1 1 1 1 1 0 0 1			
<b>CLD = Clear direction</b>	1 1 1 1 1 1 0 0			
<b>STD = Set direction</b>	1 1 1 1 1 1 0 1			
<b>CLI = Clear interrupt</b>	1 1 1 1 1 0 1 0			
<b>STI = Set interrupt</b>	1 1 1 1 1 0 1 1			
<b>HLT = Halt</b>	1 1 1 1 0 1 0 0			
<b>WAIT = Wait</b>	1 0 0 1 1 0 1 1			
<b>ESC = Escape (to external device)</b>	1 1 0 1 1 x x x	mod y y y r/m	(DISP-LO)	(DISP-HI)
<b>LOCK = Bus lock prefix</b>	1 1 1 1 0 0 0 0			
<b>SEGMENT = Override prefix</b>	0 0 1 reg 1 1 0			

**Table 4-13. Machine Instruction Decoding Guide**

1ST BYTE		2ND BYTE	BYTES 3, 4, 5, 6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
00	0000 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG8/MEM8,REG8
01	0000 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG16/MEM16,REG16
02	0000 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG8,REG8/MEM8
03	0000 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG16,REG16/MEM16
04	0000 0100	DATA-8		ADD	AL,IMMED8
05	0000 0101	DATA-LO	DATA-HI	ADD	AX,IMMED16
06	0000 0110			PUSH	ES
07	0000 0111			POP	ES

HARDWARE REFERENCE INFORMATION

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
08	0000 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG8/MEM8,REG8
09	0000 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG16/MEM16,REG16
0A	0000 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG8,REG8/MEM8
0B	0000 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG16,REG16/MEM16
0C	0000 1100	DATA-8		OR	AL,IMMED8
0D	0000 1101	DATA-LO	DATA-HI	OR	AX,IMMED16
0E	0000 1110			PUSH	CS
0F	0000 1111				(not used)
10	0001 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG8/MEM8,REG8
11	0001 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG16/MEM16,REG16
12	0001 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG8,REG8/MEM8
13	0001 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG16,REG16/MEM16
14	0001 0100	DATA-8		ADC	AL,IMMED8
15	0001 0101	DATA-LO	DATA-HI	ADC	AX,IMMED16
16	0001 0110			PUSH	SS
17	0001 0111			POP	SS
18	0001 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG8/MEM8,REG8
19	0001 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG16/MEM16,REG16
1A	0001 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG8,REG8/MEM8
1B	0001 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG16,REG16/MEM16
1C	0001 1100	DATA-8		SBB	AL,IMMED8
1D	0001 1101	DATA-LO	DATA-HI	SBB	AX,IMMED16
1E	0001 1110			PUSH	DS
1F	0001 1111			POP	DS
20	0010 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG8/MEM8,REG8
21	0010 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG16/MEM16,REG16
22	0010 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG8,REG8/MEM8
23	0010 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG16,REG16/MEM16
24	0010 0100	DATA-8		AND	AL,IMMED8
25	0010 0101	DATA-LO	DATA-HI	AND	AX,IMMED16
26	0010 0110			ES:	(segment override prefix)
27	0010 0111			DAA	
28	0010 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG8/MEM8,REG8
29	0010 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG16/MEM16,REG16
2A	0010 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG8,REG8/MEM8
2B	0010 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG16,REG16/MEM16
2C	0010 1100	DATA-8		SUB	AL,IMMED8
2D	0010 1101	DATA-LO	DATA-HI	SUB	AX,IMMED16
2E	0010 1110			CS:	(segment override prefix)
2F	0010 1111			DAS	
30	0011 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG8/MEM8,REG8
31	0011 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG16/MEM16,REG16
32	0011 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG8,REG8/MEM8
33	0011 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG16,REG16/MEM16
34	0011 0100	DATA-8		XOR	AL,IMMED8
35	0011 0101	DATA-LO	DATA-HI	XOR	AX,IMMED16
36	0011 0110			SS:	(segment override prefix)

## HARDWARE REFERENCE INFORMATION

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
37	0011 0110			AAA	
38	0011 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP	REG8/MEM8,REG8
39	0011 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP	REG16/MEM16,REG16
3A	0011 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP	REG8,REG8/MEM8
3B	0011 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP	REG16,REG16/MEM16
3C	0011 1100	DATA-8		CMP	AL,IMMED8
3D	0011 1101	DATA-LO	DATA-HI	CMP	AX,IMMED16
3E	0011 1110			DS:	(segment override prefix)
3F	0011 1111			AAS	
40	0100 0000			INC	AX
41	0100 0001			INC	CX
42	0100 0010			INC	DX
43	0100 0011			INC	BX
44	0100 0100			INC	SP
45	0100 0101			INC	BP
46	0100 0110			INC	SI
47	0100 0111			INC	DI
48	0100 1000			DEC	AX
49	0100 1001			DEC	CX
4A	0100 1010			DEC	DX
4B	0100 1011			DEC	BX
4C	0100 1100			DEC	SP
4D	0100 1101			DEC	BP
4E	0100 1110			DEC	SI
4F	0100 1111			DEC	DI
50	0101 0000			PUSH	AX
51	0101 0001			PUSH	CX
52	0101 0010			PUSH	DX
53	0101 0011			PUSH	BX
54	0101 0100			PUSH	SP
55	0101 0101			PUSH	BP
56	0101 0110			PUSH	SI
57	0101 0111			PUSH	DI
58	0101 1000			POP	AX
59	0101 1001			POP	CX
5A	0101 1010			POP	DX
5B	0101 1011			POP	BX
5C	0101 1100			POP	SP
5D	0101 1101			POP	BP
5E	0101 1110			POP	SI
5F	0101 1111			POP	DI
60	0110 0000			(not used)	
61	0110 0001			(not used)	
62	0110 0010			(not used)	
63	0110 0011			(not used)	
64	0110 0100			(not used)	
65	0110 0101			(not used)	
66	0110 0110			(not used)	
67	0110 0111			(not used)	

HARDWARE REFERENCE INFORMATION

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT
HEX	BINARY			
68	0110 1000			(not used)
69	0110 1001			(not used)
6A	0110 1010			(not used)
6B	0110 1011			(not used)
6C	0110 1100			(not used)
6D	0110 1101			(not used)
6E	0110 1110			(not used)
6F	0110 1111			(not used)
70	0111 0000	IP-INC8		JO SHORT-LABEL
71	0111 0001	IP-INC8		JNO SHORT-LABEL
72	0111 0010	IP-INC8		JB/JNAE/ SHORT-LABEL
				JC
73	0111 0011	IP-INC8		JNB/JAE/ SHORT-LABEL
				JNC
74	0111 0100	IP-INC8		JE/JZ SHORT-LABEL
75	0111 0101	IP-INC8		JNE/JNZ SHORT-LABEL
76	0111 0110	IP-INC8		JBE/JNA SHORT-LABEL
77	0111 0111	IP-INC8		JNBE/JA SHORT-LABEL
78	0111 1000	IP-INC8		JS SHORT-LABEL
79	0111 1001	IP-INC8		JNS SHORT-LABEL
7A	0111 1010	IP-INC8		JP/JPE SHORT-LABEL
7B	0111 1011	IP-INC8		JNP/JPO SHORT-LABEL
7C	0111 1100	IP-INC8		JL/JNGE SHORT-LABEL
7D	0111 1101	IP-INC8		JNL/JGE SHORT-LABEL
7E	0111 1110	IP-INC8		JLE/JNG SHORT-LABEL
7F	0111 1111	IP-INC8		JNLE/JG SHORT-LABEL
80	1000 0000	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	ADD REG8/MEM8,IMMED8
80	1000 0000	MOD 001 R/M	(DISP-LO),(DISP-HI), DATA-8	OR REG8/MEM8,IMMED8
80	1000 0000	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-8	ADC REG8/MEM8,IMMED8
80	1000 0000	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-8	SBB REG8/MEM8,IMMED8
80	1000 0000	MOD 100 R/M	(DISP-LO),(DISP-HI), DATA-8	AND REG8/MEM8,IMMED8
80	1000 0000	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-8	SUB REG8/MEM8,IMMED8
80	1000 0000	MOD 110 R/M	(DISP-LO),(DISP-HI), DATA-8	XOR REG8/MEM8,IMMED8
80	1000 0000	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-8	CMP REG8/MEM8,IMMED8
81	1000 0001	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	ADD REG16/MEM16,IMMED16
81	1000 0001	MOD 001 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	OR REG16/MEM16,IMMED16
81	1000 0001	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	ADC REG16/MEM16,IMMED16
81	1000 0001	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	SBB REG16/MEM16,IMMED16



## HARDWARE REFERENCE INFORMATION

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
81	1000 0001	MOD 100 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	AND	REG16/MEM16,IMMED16
81	1000 0001	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	SUB	REG16/MEM16,IMMED16
81	1000 0001	MOD 110 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	XOR	REG16/MEM16,IMMED16
81	1000 0001	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	CMP	REG16/MEM16,IMMED16
82	1000 0010	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	ADD	REG8/MEM8,IMMED8
82	1000 0010	MOD 001 R/M		(not used)	
82	1000 0010	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-8	ADC	REG8/MEM8,IMMED8
82	1000 0010	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-8	SBB	REG8/MEM8,IMMED8
82	1000 0010	MOD 100 R/M		(not used)	
82	1000 0010	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-8	SUB	REG8/MEM8,IMMED8
82	1000 0010	MOD 110 R/M		(not used)	
82	1000 0010	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-8	CMP	REG8/MEM8,IMMED8
83	1000 0011	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-SX	ADD	REG16/MEM16,IMMED8
83	1000 0011	MOD 001 R/M		(not used)	
83	1000 0011	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-SX	ADC	REG16/MEM16,IMMED8
83	1000 0011	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-SX	SBB	REG16/MEM16,IMMED8
83	1000 0011	MOD 100 R/M		(not used)	
83	1000 0011	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-SX	SUB	REG16/MEM16,IMMED8
83	1000 0011	MOD 110 R/M		(not used)	
83	1000 0011	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-SX	CMP	REG16/MEM16,IMMED8
84	1000 0100	MOD REG R/M	(DISP-LO),(DISP-HI)	TEST	REG8/MEM8,REG8
85	1000 0101	MOD REG R/M	(DISP-LO),(DISP-HI)	TEST	REG16/MEM16,REG16
86	1000 0110	MOD REG R/M	(DISP-LO),(DISP-HI)	XCHG	REG8,REG8/MEM8
87	1000 0111	MOD REG R/M	(DISP-LO),(DISP-HI)	XCHG	REG16,REG16/MEM16
88	1000 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG8/MEM8,REG8
89	1000 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG16/MEM16/REG16
8A	1000 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG8,REG8/MEM8
8B	1000 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG16,REG16/MEM16
8C	1000 1100	MOD 0SR R/M	(DISP-LO),(DISP-HI)	MOV	REG16/MEM16,SEGREG
8C	1000 1100	MOD 1- R/M		(not used)	
8D	1000 1101	MOD REG R/M	(DISP-LO),(DISP-HI)	LEA	REG16,MEM16
8E	1000 1110	MOD 0SR R/M	(DISP-LO),(DISP-HI)	MOV	SEGREG,REG16/MEM16
8E	1000 1110	MOD 1- R/M		(not used)	
8F	1000 1111	MOD 000 R/M	(DISP-LO),(DISP-HI)	POP	REG16/MEM16
8F	1000 1111	MOD 001 R/M		(not used)	
8F	1000 1111	MOD 010 R/M		(not used)	

HARDWARE REFERENCE INFORMATION

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT
HEX	BINARY			
8F	1000 1111	MOD 011 R/M		(not used)
8F	1000 1111	MOD 100 R/M		(not used)
8F	1000 1111	MOD 101 R/M		(not used)
8F	1000 1111	MOD 110 R/M		(not used)
8F	1000 1111	MOD 111 R/M		(not used)
90	1001 0000			NOP (exchange AX,AX)
91	1001 0001			XCHG AX,CX
92	1001 0010			XCHG AX,DX
93	1001 0011			XCHG AX,BX
94	1001 0100			XCHG AX,SP
95	1001 0101			XCHG AX,BP
96	1001 0110			XCHG AX,SI
97	1001 0111			XCHG AX,DI
98	1001 1000			CBW
99	1001 1001			CWD
9A	1001 1010	DISP-LO	DISP-HI,SEG-LO, SEG-HI	CALL FAR...PROC
9B	1001 1011			WAIT
9C	1001 1100			PUSHF
9D	1001 1101			POPF
9E	1001 1110			SAHF
9F	1001 1111			LAHF
A0	1010 0000	ADDR-LO	ADDR-HI	MOV AL,MEM8
A1	1010 0001	ADDR-LO	ADDR-HI	MOV AX,MEM16
A2	1010 0010	ADDR-LO	ADDR-HI	MOV MEM8,AL
A3	1010 0011	ADDR-LO	ADDR-HI	MOV MEM16,AL
A4	1010 0100			MOVS DEST-STR8, SRC-STR8
A5	1010 0101			MOVS DEST-STR16, SRC-STR16
A6	1010 0110			CMPS DEST-STR8, SRC-STR8
A7	1010 0111			CMPS DEST-STR16, SRC-STR16
A8	1010 1000	DATA-8		TEST AL,IMMED8
A9	1010 1001	DATA-LO	DATA-HI	TEST AX,IMMED16
AA	1010 1010			STOS DEST-STR8
AB	1010 1011			STOS DEST-STR16
AC	1010 1100			LODS SRC-STR8
AD	1010 1101			LODS SRC-STR16
AE	1010 1110			SCAS DEST-STR8
AF	1010 1111			SCAS DEST-STR16
B0	1011 0000	DATA-8		MOV AL,IMMED8
B1	1011 0001	DATA-8		MOV CL,IMMED8
B2	1011 0010	DATA-8		MOV DL,IMMED8
B3	1011 0011	DATA-8		MOV BL,IMMED8
B4	1011 0100	DATA-8		MOV AH,IMMED8
B5	1011 0101	DATA-8		MOV CH,IMMED8
B6	1011 0110	DATA-8		MOV DH,IMMED8
B7	1011 0111	DATA-8		MOV BH,IMMED8
B8	1011 1000	DATA-LO	DATA-HI	MOV AX,IMMED16
B9	1011 1001	DATA-LO	DATA-HI	MOV CX,IMMED16
BA	1011 1010	DATA-LO	DATA-HI	MOV DX,IMMED16
BB	1011 1011	DATA-LO	DATA-HI	MOV BX,IMMED16

## HARDWARE REFERENCE INFORMATION

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
BC	1011 1100	DATA-LO	DATA-HI	MOV	SP,IMMED16
BD	1011 1101	DATA-LO	DATA-HI	MOV	BP,IMMED16
BE	1011 1110	DATA-LO	DATA-HI	MOV	SI,IMMED16
BF	1011 1111	DATA-LO	DATA-HI	MOV	DI,IMMED16
C0	1100 0000			(not used)	
C1	1100 0001			(not used)	
C2	1100 0010	DATA-LO	DATA-HI	RET	IMMED16 (intraseg)
C3	1100 0011			RET	(intrasegment)
C4	1100 0100	MOD REG R/M	(DISP-LO),(DISP-HI)	LES	REG16,MEM16
C5	1100 0101	MOD REG R/M	(DISP-LO),(DISP-HI)	LDS	REG16,MEM16
C6	1100 0110	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	MOV	MEM8,IMMED8
C6	1100 0110	MOD 001 R/M		(not used)	
C6	1100 0110	MOD 010 R/M		(not used)	
C6	1100 0110	MOD 011 R/M		(not used)	
C6	1100 0110	MOD 100 R/M		(not used)	
C6	1100 0110	MOD 101 R/M		(not used)	
C6	1100 0110	MOD 110 R/M		(not used)	
C6	1100 0110	MOD 111 R/M		(not used)	
C7	1100 0111	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	MOV	MEM16,IMMED16
C7	1100 0111	MOD 001 R/M		(not used)	
C7	1100 0111	MOD 010 R/M		(not used)	
C7	1100 0111	MOD 011 R/M		(not used)	
C7	1100 0111	MOD 100 R/M		(not used)	
C7	1100 0111	MOD 101 R/M		(not used)	
C7	1100 0111	MOD 110 R/M		(not used)	
C7	1100 0111	MOD 111 R/M		(not used)	
C8	1100 1000			(not used)	
C9	1100 1001			(not used)	
CA	1100 1010	DATA-LO	DATA-HI	RET	IMMED16 (intersegment)
CB	1100 1011			RET	(intersegment)
CC	1100 1100			INT	3
CD	1100 1101	DATA-8		INT	IMMED8
CE	1100 1110			INTO	
CF	1100 1111			IRET	
D0	1101 0000	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROL	REG8/MEM8,1
D0	1101 0000	MOD 001 R/M	(DISP-LO),(DISP-HI)	ROR	REG8/MEM8,1
D0	1101 0000	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL	REG8/MEM8,1
D0	1101 0000	MOD 011 R/M	(DISP-LO),(DISP-HI)	RCR	REG8/MEM8,1
D0	1101 0000	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL	REG8/MEM8,1
D0	1101 0000	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR	REG8/MEM8,1
D0	1101 0000	MOD 110 R/M		(not used)	
D0	1101 0000	MOD 111 R/M	(DISP-LO),(DISP-HI)	SAR	REG8/MEM8,1
D1	1101 0001	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROL	REG16/MEM16,1
D1	1101 0001	MOD 001 R/M	(DISP-LO),(DISP-HI)	ROR	REG16/MEM16,1
D1	1101 0001	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL	REG16/MEM16,1
D1	1101 0001	MOD 011 R/M	(DISP-LO),(DISP-HI)	RCR	REG16/MEM16,1
D1	1101 0001	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL	REG16/MEM16,1

HARDWARE REFERENCE INFORMATION

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT
HEX	BINARY			
D1	1101 0001	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR REG16/MEM16,1 (not used)
D1	1101 0001	MOD 110 R/M	(DISP-LO),(DISP-HI)	SAR REG16/MEM16,1
D1	1101 0001	MOD 111 R/M	(DISP-LO),(DISP-HI)	ROL REG8/MEM8,CL
D2	1101 0010	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROR REG8/MEM8,CL
D2	1101 0010	MOD 001 R/M	(DISP-LO),(DISP-HI)	RCL REG8/MEM8,CL
D2	1101 0010	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCR REG8/MEM8,CL
D2	1101 0010	MOD 011 R/M	(DISP-LO),(DISP-HI)	SAL/SHL REG8/MEM8,CL
D2	1101 0010	MOD 100 R/M	(DISP-LO),(DISP-HI)	SHR REG8/MEM8,CL
D2	1101 0010	MOD 101 R/M	(DISP-LO),(DISP-HI)	(not used)
D2	1101 0010	MOD 110 R/M	(DISP-LO),(DISP-HI)	SAR REG8/MEM8,CL
D2	1101 0010	MOD 111 R/M	(DISP-LO),(DISP-HI)	ROL REG16/MEM16,CL
D3	1101 0011	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROR REG16/MEM16,CL
D3	1101 0011	MOD 001 R/M	(DISP-LO),(DISP-HI)	RCL REG16/MEM16,CL
D3	1101 0011	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCR REG16/MEM16,CL
D3	1101 0011	MOD 011 R/M	(DISP-LO),(DISP-HI)	SAL/SHL REG16/MEM16,CL
D3	1101 0011	MOD 100 R/M	(DISP-LO),(DISP-HI)	SHR REG16/MEM16,CL
D3	1101 0011	MOD 101 R/M	(DISP-LO),(DISP-HI)	(not used)
D3	1101 0011	MOD 110 R/M	(DISP-LO),(DISP-HI)	SAR REG16/MEM16,CL
D3	1101 0011	MOD 111 R/M	(DISP-LO),(DISP-HI)	AAM
D4	1101 0100	00001010		AAD
D5	1101 0101	00001010		(not used)
D6	1101 0110			XLAT SOURCE-TABLE
D7	1101 0111			ESC CPCODE,SOURCE
D8	1101 1000	MOD 000 R/M	(DISP-LO), (DISP-HI)	LOOPNE/ SHORT-LABEL
		1XXX MOD YYY R/M		LOOPNZ
DF	1101 1111	MOD 111 R/M		LOOPE/ SHORT-LABEL
E0	1110 0000	IP-INC-8		LOOPZ
E1	1110 0001	IP-INC-8		LOOP SHORT-LABEL
E2	1110 0010	IP-INC-8		JCXZ SHORT-LABEL
E3	1110 0011	IP-INC-8		IN AL,IMMED8
E4	1110 0100	DATA-8		IN AX,IMMED8
E5	1110 0101	DATA-8		OUT AL,IMMED8
E6	1110 0110	DATA-8		OUT AX,IMMED8
E7	1110 0111	DATA-8		CALL NEAR-PROC
E8	1110 1000	IP-INC-LO	IP-INC-HI	JMP NEAR-LABEL
E9	1110 1001	IP-INC-LO	IP-INC-HI	JMP FAR-LABEL
EA	1110 1010	IP-LO	IP-HI,CS-LO,CS-HI	JMP SHORT-LABEL
EB	1110 1011	IP-INC8		IN AL,DX
EC	1110 1100			IN AX,DX
ED	1110 1101			OUT AL,DX
EE	1110 1110			OUT AX,DX
EF	1110 1111			LOCK (prefix)
F0	1111 0000			(not used)
F1	1111 0001			REPNE/REPNZ
F2	1111 0010			REP/REPE/REPZ
F3	1111 0011			HLT
F4	1111 0100			CMC
F5	1111 0101			

## HARDWARE REFERENCE INFORMATION

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
F6	1111 0110	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	TEST	REG8/MEM8,IMMED8
F6	1111 0110	MOD 001 R/M		(not used)	
F6	1111 0110	MOD 010 R/M	(DISP-LO),(DISP-HI)	NOT	REG8/MEM8
F6	1111 0110	MOD 011 R/M	(DISP-LO),(DISP-HI)	NEG	REG8/MEM8
F6	1111 0110	MOD 100 R/M	(DISP-LO),(DISP-HI)	MUL	REG8/MEM8
F6	1111 0110	MOD 101 R/M	(DISP-LO),(DISP-HI)	IMUL	REG8/MEM8
F6	1111 0110	MOD 110 R/M	(DISP-LO),(DISP-HI)	DIV	REG8/MEM8
F6	1111 0110	MOD 111 R/M	(DISP-LO),(DISP-HI)	IDIV	REG8/MEM8
F7	1111 0111	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	TEST	REG16/MEM16,IMMED16
F7	1111 0111	MOD 001 R/M		(not used)	
F7	1111 0111	MOD 010 R/M	(DISP-LO),(DISP-HI)	NOT	REG16/MEM16
F7	1111 0111	MOD 011 R/M	(DISP-LO),(DISP-HI)	NEG	REG16/MEM16
F7	1111 0111	MOD 100 R/M	(DISP-LO),(DISP-HI)	MUL	REG16/MEM16
F7	1111 0111	MOD 101 R/M	(DISP-LO),(DISP-HI)	IMUL	REG16/MEM16
F7	1111 0111	MOD 110 R/M	(DISP-LO),(DISP-HI)	DIV	REG16/MEM16
F7	1111 0111	MOD 111 R/M	(DISP-LO),(DISP-HI)	IDIV	REG16/MEM16
F8	1111 1000			CLC	
F9	1111 1001			STC	
FA	1111 1010			CLI	
FB	1111 1011			STI	
FC	1111 1100			CLD	
FD	1111 1101			STD	
FE	1111 1110	MOD 000 R/M	(DISP-LO),(DISP-HI)	INC	REG8/MEM8
FE	1111 1110	MOD 001 R/M	(DISP-LO),(DISP-HI)	DEC	REG8/MEM8
FE	1111 1110	MOD 010 R/M		(not used)	
FE	1111 1110	MOD 011 R/M		(not used)	
FE	1111 1110	MOD 100 R/M		(not used)	
FE	1111 1110	MOD 101 R/M		(not used)	
FE	1111 1110	MOD 110 R/M		(not used)	
FE	1111 1110	MOD 111 R/M		(not used)	
FF	1111 1111	MOD 000 R/M	(DISP-LO),(DISP-HI)	INC	MEM16
FF	1111 1111	MOD 001 R/M	(DISP-LO),(DISP-HI)	DEC	MEM16
FF	1111 1111	MOD 010 R/M	(DISP-LO),(DISP-HI)	CALL	REG16/MEM16 (intra)
FF	1111 1111	MOD 011 R/M	(DISP-LO),(DISP-HI)	CALL	MEM16 (intersegment)
FF	1111 1111	MOD 100 R/M	(DISP-LO),(DISP-HI)	JMP	REG16/MEM16 (intra)
FF	1111 1111	MOD 101 R/M	(DISP-LO),(DISP-HI)	JMP	MEM16 (intersegment)
FF	1111 1111	MOD 110 R/M	(DISP-LO),(DISP-HI)	PUSH	MEM16
FF	1111 1111	MOD 111 R/M		(not used)	

HARDWARE REFERENCE INFORMATION

Table 4-14. Machine Instruction Encoding Matrix

Hi	Lo															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD b,r/m	ADD w,r/m	ADD b,r/m	ADD w,r/m	ADD b,ia	ADD w,ia	PUSH ES	POP ES	OR b,r/m	OR w,r/m	OR b,r/m	OR w,r/m	OR b,i	OR w,i	PUSH CS	
1	ADC b,r/m	ADC w,r/m	ADC b,r/m	ADC w,r/m	ADC b,i	ADC w,i	PUSH SS	POP SS	SBB b,r/m	SBB w,r/m	SBB b,r/m	SBB w,r/m	SBB b,i	SBB w,i	PUSH DS	POP DS
2	AND b,r/m	AND w,r/m	AND b,r/m	AND w,r/m	AND b,i	AND w,i	SEG ES	DAA	SUB b,r/m	SUB w,r/m	SUB b,r/m	SUB w,r/m	SUB b,i	SUB w,i	SEG CS	DAS
3	XOR b,r/m	XOR w,r/m	XOR b,r/m	XOR w,r/m	XOR b,i	XOR w,i	SEG SS	AAA	CMP b,r/m	CMP w,r/m	CMP b,r/m	CMP w,r/m	CMP b,i	CMP w,i	SEG DS	AAS
4	INC AX	INC CX	INC DX	INC BX	INC SP	INC BP	INC SI	INC DI	DEC AX	DEC CX	DEC DX	DEC BX	DEC SP	DEC BP	DEC SI	DEC DI
5	PUSH AX	PUSH CX	PUSH DX	PUSH BX	PUSH SP	PUSH BP	PUSH SI	PUSH DI	POP AX	POP CX	POP DX	POP BX	POP SP	POP BP	POP SI	POP DI
6																
7	JO	JNO	JB/ JNAE	JNB/ JAE	JE/ JZ	JNE/ JNZ	JBE/ JNA	JNBE/ JJA	JS	JNS	JP/ JPE	JNP/ JPO	JL/ JNGE	JNL/ JGE	JLE/ JNG	JNLE/ JG
8	Immed b,r/m	Immed w,r/m	Immed b,r/m	Immed w,r/m	TEST b,r/m	TEST w,r/m	XCHG b,r/m	XCHG w,r/m	MOV b,r/m	MOV w,r/m	MOV b,r/m	MOV w,r/m	MOV b,r/m	MOV w,r/m	LEA b,r/m	POP r/m
9	XCHG AX	XCHG CX	XCHG DX	XCHG BX	XCHG SP	XCHG BP	XCHG SI	XCHG DI	CBW	CWD	CALL d	WAIT	PUSHF	POPF	SAHF	LAHF
A	MOV m-AL	MOV m-AX	MOV AL-m	MOV AX-m	MOVS	MOVS	CMPS	CMPS	TEST b,ib	TEST w,ib	STOS	STOS	LODS	LODS	SCAS	SCAS
B	MOV r-AL	MOV r-CL	MOV r-DL	MOV r-BL	MOV r-AH	MOV r-CH	MOV r-DH	MOV r-BH	MOV r-AX	MOV r-CX	MOV r-DX	MOV r-BX	MOV r-SP	MOV r-BP	MOV r-SI	MOV r-DI
C			RET (i,SP)	RET	LES	LDS	MOV b,r/m	MOV w,r/m			RET l,(i,SP)	RET i	INT Type 3	INT (Any)	INTO	IRET
D	Shift b	Shift w	Shift b,v	Shift w,v	AAM	AAD		XLAT	ESC 0	ESC 1	ESC 2	ESC 3	ESC 4	ESC 5	ESC 6	ESC 7
E	LOOPNZ/ LOOPNE	LOOPZ/ LOOPE	LOOP	JCJZ	IN b	IN w	OUT b	OUT w	CALL d	JMP d	JMP s,d	JMP s,d	IN v,b	IN v,w	OUT v,b	OUT v,w
F	LOCK		REP	REP z	HLT	CMC	Grp 1 b,r/m	Grp 1 w,r/m	CLC	STC	CLI	STI	CLD	STD	Grp 2 b,r/m	Grp 2 w,r/m

where

mod	r/m	000	001	010	011	100	101	110	111
Immed		ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
Shift		ROL	ROR	RCL	RCR	SHL/SAL	SHR	—	SAR
Grp 1		TEST	—	NOT	NEG	MUL	IMUL	DIV	IDIV
Grp 2		INC	DEC	CALL id	CALL l,d	JMP id	JMP l,d	PUSH	—

- b - byte operation
- d - direct
- f - from CPU reg
- i - immediate
- ia - immed. to accum
- id - indirect
- is - immed. byte. sign ext
- l - long ie intersegment
- m - memory
- r/m - EA is second byte
- si - short intrasegment
- sr - segment register
- t - to CPU reg
- v - variable
- w - word operation
- z - zero

*Appendix C*

# **COMPUTER ARITHMETIC**

## BINARY ARITHMETIC

A number system can be used to perform two basic operations: addition and subtraction. But by using addition and subtraction, you can then perform multiplication, division, and any other numerical operation. For simplicity, we will use decimal arithmetic as a guide.

### Binary Addition

Binary addition is performed somewhat like decimal addition. If two decimal numbers,  $56719_{10}$  and  $31863_{10}$  for example, are added together, the sum  $88582_{10}$  is obtained. You can analyze the details of this operation in the following manner.

NOTE: In the following explanations, the term “first column” refers to the first column of figures you work with in the problem — the column on the right (9, 3, and 2 in the following example). The term “second column” refers to the second column you work with, etc.

Carry:	00101
Addend:	56719
Augend:	+ 31863
Sum:	<u>88582</u>

Adding the first column, decimal numbers 9 and 3, gives the sum of 12. This is expressed in the sum as the digit 2 with a carry of 1. The carry is then added to the next column. Adding the second column decimal numbers 1 and 6, and the carry from the first column, 1, gives the sum of 8, with no carry. This process continues until all of the columns (including carries) have been added. The sum represents the numeric value of the addend and the augend. (The addend is the number to be added to another number, while the augend is the number to which the addend is added.)



When you add two binary numbers, you perform the same operation. The example below summarizes the four rules of addition with binary numbers.

1.  $0 + 0 = 0$
2.  $0 + 1 = 1$
3.  $1 + 1 = 0$  with a carry of 1
4.  $1 + 1 + 1 = 1$  with a carry of 1

To illustrate the process of binary addition, let's add  $1101_2$  to  $1101_2$ .

$$\begin{array}{r}
 \text{Carry:} \quad 1101 \\
 \text{Addend:} \quad 1101_2 \\
 \text{Augend:} \quad + 1101_2 \\
 \hline
 \text{Sum:} \quad 11010_2
 \end{array}$$

In the first column, 1 plus 1 equals 0 with a carry of 1 to the second column. This agrees with rule 3.

In the second column, 0 plus 0 equals 0 with no carry. The carry from the first column is added to this. Thus, 0 plus 1 equals 1 with no carry. These two additions in the second column give a total sum of 1 with a carry of 0. Rules 1 and 2 were used to obtain the sum.

In column three, 1 plus 1 equals 0 with a carry of 1. To this sum, the second column is added. This yields a third column sum of 0 with a carry of 1 to column four. Rules 3 and 1 were used to obtain the sum.

In column four, 1 plus 1 equals 0 with a carry of 1. To this sum, the third column carry is added. This yields a fourth column sum of 1 with a carry to the fifth column. Rule 4 allows you to add three binary 1's and obtain 1 with a carry of 1.

In column five, there is no addend or augend. Therefore, you can assume rule 2 and add the carry to obtain the sum of 1. Thus, the sum of  $1101_2$  plus  $1101_2$  equals  $11010_2$ . You can verify this by converting the binary numbers to decimal numbers.

Now study the following two examples of binary addition, where  $10001111_2$  is added to  $10110101_2$  and  $111011_2$  is added to  $11001100_2$ .

$$\begin{array}{r}
 \text{Carry:} \quad 10111111 \\
 \text{Addend:} \quad 10110101_2 \\
 \text{Augend:} \quad + 10001111_2 \\
 \hline
 \text{Sum:} \quad 101000100_2
 \end{array}$$

$$\begin{array}{r}
 \text{Carry:} \quad 11111000 \\
 \text{Addend:} \quad 11001100_2 \\
 \text{Augend:} \quad + 00111011_2 \\
 \hline
 \text{Sum:} \quad 100000111_2
 \end{array}$$

When a microprocessor adds binary numbers, 8-bit numbers are generally used. As shown in the last example, two zeros were added after the MSB of the augend to produce an 8-bit number. After addition, a 1 in the ninth bit is represented as the “carry” bit by the microprocessor.

## Binary Subtraction

Binary subtraction is performed exactly like decimal subtraction. Therefore, before you attempt binary subtraction, you should reexamine decimal subtraction. You know that in decimal arithmetic, if 5486 is subtracted from 8303, the difference, 2817 is obtained.

$$\begin{array}{r}
 \text{Minuend after borrow} \quad 7 \quad 12 \quad 9 \quad 13 \\
 \text{Minuend:} \quad 8 \quad 3 \quad 0 \quad 3 \\
 \text{Subtrahend:} \quad - \quad 5 \quad 4 \quad 8 \quad 6 \\
 \hline
 \text{Difference:} \quad 2 \quad 8 \quad 1 \quad 7
 \end{array}$$

Because the digit 6 in the subtrahend is larger than the digit 3 in the minuend, a 1 is borrowed from the next high-order digit in the minuend. If that digit is a 0, as in this example, 1 is borrowed from the next high-order digit that contains a number other than 0. That digit is reduced by 1 (from 3 to 2 in our example) and the digits skipped in the minuend are given the value 9. This is equivalent to removing 1 from 30 with the result of 29, as in our example. In the decimal system, the digit borrowed has the value of 10. Therefore, the minuend digit now has the value 13, and 6 from 13 equals 7.

In the second column, 8 from 9 equals 1. Since the subtrahend is larger than the minuend in the third column, 1 is borrowed from the next higher-order digit. This raises the minuend value from 2 to 12, and 4 from 12 equals 8. In the fourth column, the minuend was reduced from 8 to 7 because of the previous borrow, and 5 from 7 equals 2.

Whenever 1 is borrowed from a higher-order digit, the borrow is equal in value to the radix or base of the number system. As you know, the radix or base of the decimal number system is 10, and the radix or base in the binary system is 2. Therefore, a borrow in the decimal number system equals 10, while a borrow in the binary number system equals 2.

When you subtract one binary number from another, you use the same method described for decimal subtraction. This is summarized by the following for binary subtraction.

1.  $0 - 0 = 0$
2.  $1 - 1 = 0$
3.  $1 - 0 = 1$
4.  $0 - 1 = 1$  with a borrow of 1.

To illustrate the process of binary subtraction, let's subtract  $1101_2$  from  $11011_2$ .

Minuend after borrow:	0 10 10 1 1
Minuend:	1 1 0 1 1
Subtrahend:	- 1 1 0 1
Difference:	1 1 1 0

The “minuend after borrow” now shows the value of each minuend digit after a borrow occurs. **Remember that binary 10 equals decimal 2.**

In the first column, 1 from 1 equals 0 (rule 2). Then, 0 from 1 in the second column equals 1 (rule 3). In the third column, 1 from 0 requires a borrow from the fourth column. Thus, 1 from 10 equals 1 (rule 4). The minuend in the fourth column is now 0, from the previous borrow. Therefore, a borrow is required from the fifth column, so that 1 from 10 in the fourth column equals 1 (rule 4). Because of the previous borrow, the minuend in the fifth column is now 0 and the subtrahend is 0 (nonexistent), so that 0 from 0 equals 0 (rule 1). The 0 in the fifth column is not shown in the difference because it is not a significant bit. Thus, the difference between  $11011_2$  and  $1101_2$  is  $1110_2$ . You can verify this by converting the binary number to a decimal number and subtracting.

As a further example of binary subtraction, subtract  $00100101_2$  from  $11000100_2$ , as shown below. Then proceed to the next example and subtract  $10111010_2$  from  $11101110_2$ .

$$\begin{array}{r}
 \text{Minuend after borrow:} \quad 1 \ 0 \ 1 \ 1 \ 1 \ 10 \ 1 \ 10 \\
 \text{Minuend:} \quad 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \\
 \text{Subtrahend:} \quad - \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \\
 \hline
 \text{Difference:} \quad 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1
 \end{array}$$

$$\begin{array}{r}
 \text{Minuend after borrow:} \quad 0 \ 0 \ 10 \ 10 \ 1 \ 1 \ 1 \ 0 \\
 \text{Minuend:} \quad 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \\
 \text{Subtrahend:} \quad - \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \\
 \hline
 \text{Difference:} \quad 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0
 \end{array}$$

When a borrow is required in the minuend, 1 is obtained from the next high-order bit that contains a 1. That bit then becomes 0, and all bits skipped (0 value bits) are given the value of 1. This is equivalent to removing 1 from  $1000_2$  with the result of  $0111_2$ .

As with binary addition, microprocessors generally subtract with 8-bit number groups. In the previous example, the answer contained only six significant bits, but two 0 bits were added to maintain the 8-bit grouping. This would also hold true for the minuend and subtrahend.

## Binary Multiplication

Multiplication is a short method of adding a number to itself as many times as it is specified by the multiplier. However, if you were to multiply  $324_{10}$  by  $233_{10}$  you would probably use the following method.

$$\begin{array}{r}
 \text{Multiplicand:} \quad 324 \\
 \text{Multiplier:} \quad \times 223 \\
 \hline
 \text{First partial product:} \quad 972 \\
 \text{Second partial product:} \quad 648 \\
 \text{Third partial product:} \quad 648 \\
 \text{Carry:} \quad 0121 \\
 \hline
 \text{Final product:} \quad 72252
 \end{array}$$

Using the short form of multiplication, you multiply the multiplicand by each digit of the multiplier and then sum the partial products to obtain the final product. Note that, for convenience, the additive carries are set down under the partial products rather than over them as in normal addition.

Binary multiplication follows the same general principles as decimal multiplication. However, with only two possible multiplier bits (1 or 0), binary multiplication is a much simpler process. The example below lists the rules of binary multiplication. These rules will be used to multiply  $1111_2$  by  $1101_2$ .

1.  $0 \times 0 = 0$
2.  $0 \times 1 = 0$
3.  $1 \times 0 = 0$
4.  $1 \times 1 = 1$

Multiplicand:	1111
Multiplier:	× 1101
First partial product:	<u>1111</u>
Second partial product:	<u>0000</u>
Carry:	<u>0000</u>
Sum of partial products:	1111
Third partial product:	<u>1111</u>
Carry:	<u>111100</u>
Sum of partial products:	1001011
Fourth partial product:	<u>1111</u>
Carry:	<u>1111000</u>
Final product:	11000011

As with decimal multiplication, you multiply the multiplicand by each bit in the multiplier and add the partial sums. First you multiply  $1111_2$  by the least significant multiplier bit (1) and set down the partial product so the least significant bit (LSB) is under the multiplier bit. Then you multiply the multiplicand by the next multiplier bit (0) and set down the partial product so the LSB is under the multiplier bit. Now that there are two partial products, they should be added. Although it is possible to add more than two binary numbers, keeping track of multiple carries may become confusing. Therefore, for these examples, add only two partial products at a time.

Notice that the first partial product is identical to the multiplicand. The second partial product is all zeros. Since the binary number system contains only ones and zeros, the partial product will always equal either the multiplicand or zero. Because of this, you can obtain the third partial product by copying the multiplicand. Begin with the LSB under the third multiplier bit. Add this value to the previous partial sum. Now obtain the fourth partial product by copying the multiplicand. Begin with the LSB under the fourth multiplier bit. Add this value to the previous partial sum. This is the final product. Again, you can verify the result by converting the binary numbers to decimal.

Reexamine the illustration for the previous multiplication example and you will notice that binary multiplication is a process of shift and add. For each 1 bit in the multiplier you copy down the multiplicand, beginning with the LSB under the bit. You can ignore any zeros in the multiplier. But do not make the mistake of setting down the multiplicand under the 0 bit.

To make sure you fully understand binary multiplication, multiply  $1001_2$  by  $1100_2$  and then multiply  $1101_2$  by  $1111_2$ .

Multiplicand:	1001
Multiplier:	× 1100
First partial product:	<u>0000</u>
Second partial product:	<u>0000</u>
Carry:	<u>0000</u>
Sum of partial products:	00000
Third partial product:	<u>1001</u>
Carry:	<u>00000</u>
Sum of partial products:	100100
Fourth partial product:	<u>1001</u>
Carry:	<u>000000</u>
Final product:	1101100

Multiplicand:	1101
Multiplier:	× 1111
First partial product:	<u>1101</u>
Second partial product:	<u>1101</u>
Carry:	<u>11000</u>
Sum of partial products:	100111
Third partial product:	<u>1101</u>
Carry:	<u>100100</u>
Sum of partial products:	1011011
Fourth partial product:	<u>1101</u>
Carry:	<u>1111000</u>
Final product:	11000011

In the first of these examples, the two zeros in the multiplier were included in the multiplication process. This was to insure that the multiplicand was copied down under the proper multiplier bits. The multiplication process could have been represented in this manner:

Multiplicand:	1001
Multiplier:	× 1100
Third partial product:	<u>100100</u>
Fourth partial product:	<u>1001</u>
Carry:	<u>000000</u>
Final product:	<u>1101100</u>

Remember, just as in decimal multiplication, you must keep track of any zeros by setting a zero in the product under the 0 bit in the multiplier. This is very important when the zero occupies the LSB.

## Binary Division

Division is the reverse of multiplication. Therefore, it is a procedure for determining how many times one number can be subtracted from another. The process you are probably familiar with is called “long” division. If you were to divide decimal 181 by 45, you would obtain the quotient,  $4\frac{1}{45}$ , as follows:

$$\begin{array}{r}
 \text{Divisor } 45 \quad \begin{array}{r} 004 \\ \overline{)181} \\ 180 \\ \hline 1 \end{array} \\
 \text{Quotient} \\
 \text{Dividend} \\
 \text{Remainder}
 \end{array}$$

Using long division, you would examine the most significant digit in the dividend and determine if the divisor was smaller in value. In this example, the divisor is larger, so the quotient is zero. Next, you examine the two most significant digits, and here again, the divisor is larger, so the quotient is again zero. Finally, you examine the whole dividend and discover it is approximately four times the divisor value. Therefore, you give the quotient a value of 4. Next, you subtract the product of 45 and 4 (180) from the dividend. The difference of 1 represents a fraction of the divisor. This fraction is added to the quotient to produce the correct answer of  $4\frac{1}{45}$ .

Binary division is performed in a similar manner. However, binary division is a simpler process since the number base is two rather than ten. First, let's divide  $100011_2$  by  $101_2$ .

$$\begin{array}{r}
 \text{Divisor: } 101 \quad \begin{array}{r} 000111 \\ \overline{)100011} \\ 101 \phantom{000} \\ \hline 111 \phantom{00} \\ 101 \phantom{0} \\ \hline 101 \\ 101 \\ \hline 0 \end{array} \\
 \text{Quotient} \\
 \text{Dividend} \\
 \text{Remainder} \\
 \text{Remainder} \\
 \text{Remainder}
 \end{array}$$

Using long division, you examine the dividend beginning with the MSB and determine the number of bits required to exceed the value of the divisor. When you find this value, place a 1 in the quotient and subtract the divisor from the selected dividend value. Then carry the next least significant bit in the dividend down to the remainder. If you can subtract



the divisor from the new remainder, place a 1 in the quotient. Then subtract the divisor from the remainder and carry the next least significant bit in the dividend (LSB in this example) down to the remainder. If the divisor can be subtracted from the new remainder, place a 1 in the quotient and subtract the divisor from the remainder. Continue the process until all of the dividend bits have been carried down. Then express any remainder as a fraction of the divisor in the quotient. Thus,  $100011_2$  divided by  $101_2$  equals  $111_2$ . You can verify the answer by converting the binary numbers to decimal.

To make sure you fully understand binary division, work out the following examples of long division. Divide  $101000_2$  by  $1000_2$  and then divide  $100111_2$  by  $110_2$ .

Divisor 1000	$\begin{array}{r} 000101 \\ \overline{)101000} \\ 1000 \phantom{00} \\ \underline{\phantom{1000}000} \\ 1000 \\ \underline{\phantom{1000}000} \\ 0 \end{array}$	Quotient
	$\begin{array}{r} 1000 \\ \phantom{1000}00 \\ \phantom{1000}00 \\ \phantom{1000}00 \end{array}$	Dividend
	$\begin{array}{r} 1000 \\ \phantom{1000}00 \\ \phantom{1000}00 \\ \phantom{1000}00 \end{array}$	Remainder
	$\begin{array}{r} 1000 \\ \phantom{1000}00 \\ \phantom{1000}00 \\ \phantom{1000}00 \end{array}$	Remainder
	$\begin{array}{r} 0 \\ \phantom{0}0 \\ \phantom{0}0 \\ \phantom{0}0 \end{array}$	Remainder

Divisor 110	$\begin{array}{r} 000110.1 \\ \overline{)100111.0} \\ 110 \phantom{00} \\ \underline{\phantom{110}001} \\ 110 \phantom{00} \\ \underline{\phantom{110}001} \\ 110 \phantom{00} \\ \underline{\phantom{110}001} \\ 110 \\ \underline{\phantom{110}001} \\ 0 \end{array}$	Quotient
	$\begin{array}{r} 110 \\ \phantom{110}00 \\ \phantom{110}00 \\ \phantom{110}00 \end{array}$	Dividend
	$\begin{array}{r} 110 \\ \phantom{110}00 \\ \phantom{110}00 \\ \phantom{110}00 \end{array}$	Remainder
	$\begin{array}{r} 110 \\ \phantom{110}00 \\ \phantom{110}00 \\ \phantom{110}00 \end{array}$	Remainder
	$\begin{array}{r} 110 \\ \phantom{110}00 \\ \phantom{110}00 \\ \phantom{110}00 \end{array}$	Remainder
	$\begin{array}{r} 0 \\ \phantom{0}0 \\ \phantom{0}0 \\ \phantom{0}0 \end{array}$	Remainder

In the second example, the quotient was not a whole number, but rather a whole number plus a fraction (remainder divided by the divisor). The answer  $110-11/110$  is correct. You could have left the answer in this form or, as in the example, continue the division process until the remainder was zero. This is made possible by adding a sufficient number of zeros after the binary point to permit division by the divisor. In the previous example, only one zero was added after the binary point. As in the decimal number system, adding zeros after the binary point, in the binary number system, will not affect the value of the number. Note that some numbers cannot be solved in this manner (e.g., decimal  $1/3$ ).

## Representing Negative Numbers

Until now, we have been examining binary arithmetic using unsigned numbers. However, when you perform some arithmetic operations with a microprocessor, you must be able to express both positive and negative (signed) numbers. Over the years, three methods have been developed for representing signed numbers. Of these, only one method has survived. The two older methods will be briefly examined first, followed by the system that is used today.

### SIGN AND MAGNITUDE

Using this system, a binary number contained both the sign (+ or -) and the value of the number. Therefore, positive and negative values were expressed as follows:

$$\begin{array}{rcc} +45_{10} & = & 0 \ 0101101 \\ & \swarrow & \searrow \\ & \text{Sign} & \text{Magnitude} \\ & \swarrow & \searrow \\ -45_{10} & = & 1 \ 0101101 \end{array}$$

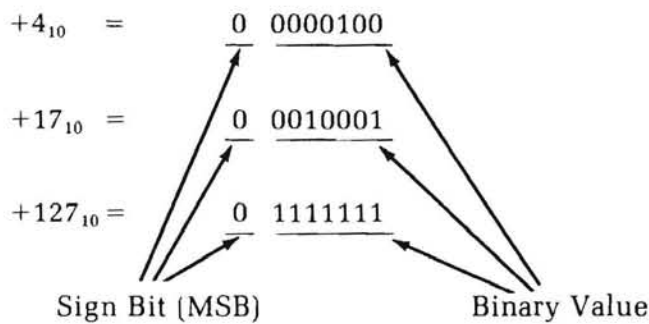
(MSB)

The MSB of the binary number indicated the sign, while the remaining bits contained the value of the number. As you can see, a zero sign bit indicated a positive value, while a one sign bit indicated a negative value.

While this method of representing negative numbers may seem logical, its popularity was short lived. Because it required complex and slow arithmetic circuitry, it was abandoned long before microprocessors were invented.

### ONE'S COMPLEMENT

Another method of representing negative numbers became popular in the early days of computers. It was called the one's complement method. Using this system, positive numbers were represented in the same way as in the sign-magnitude system. That is, the MSB in any number was considered to be a sign bit. A sign bit of 0 represented positive. Using 8-bit numbers, positive values were represented like this:



Negative numbers were represented by the one's complement of the positive value. The one's complement of a number is formed by changing all the 0's to 1's and all the 1's to 0's. As shown above,  $+4_{10}$  is represented as  $0\ 0000100$ . By changing all 0's to 1's and all 1's to 0's, the representation for  $-4_{10}$  was formed. In this case:

$$-4_{10} = \underline{1} \quad \underline{1111011}_2$$

Notice that all the bits, including the sign bit, were inverted. In the same way:

$$-17_{10} = \underline{1} \quad \underline{1101110}_2$$

$$-127_{10} = \underline{1} \quad \underline{0000000}_2$$

The one's complement method is not used for representing signed numbers in microprocessors, but if you need to find the one's complement of a number, simply change all the 0's to 1's and all the 1's to 0's.

Figure 1 shows an interesting relationship. In the first column, 8-bit patterns of 0's and 1's are shown. The second column shows the decimal number that each pattern represents if you consider the pattern to be an unsigned binary number. Notice that an 8-bit pattern can represent unsigned numbers between 0 and  $255_{10}$ .

BIT PATTERN	UNSIGNED BINARY	1's COMPLEMENT
00000000	0	+0
00000001	1	+1
00000010	2	+2
00000011	3	+3
.	.	.
.	.	.
.	.	.
.	.	.
01111100	124	+124
01111101	125	+125
01111110	126	+126
01111111	127	+127
10000000	128	-127
10000001	129	-126
10000010	130	-125
10000011	131	-124
.	.	.
.	.	.
.	.	.
.	.	.
11111100	252	-3
11111101	253	-2
11111110	254	-1
11111111	255	-0

Figure 1

Table of bit pattern values for unsigned binary numbers and 1's complement numbers.

The third column shows the decimal number that each pattern represents if you consider the pattern to be a one's complement binary number. Notice that the range of numbers is from  $-127_{10}$  to  $+127_{10}$ . Notice also that there are two representations of zero. The pattern  $0000\ 0000_2$  represents  $+0$  while its one's complement ( $1111\ 1111_2$ ) represents  $-0$ .

## TWO'S COMPLEMENT

The method used to represent signed numbers in microprocessors is called two's complement. In this system, positive numbers are represented just as they were with the sign-and-magnitude method and the one's complement method. That is, it uses the same bit pattern for all positive values up to  $+127_{10}$ . However, negative numbers are represented as the two's complement of positive numbers.

The two's complement of a number is formed by taking the one's complement and then adding 1. For example, if you work with 8-bit numbers and use the two's complement system,  $+4_{10}$  is represented by  $0000100_2$ . To find  $-4_{10}$ , you must take the two's complement of this number. You do this by first taking the one's complement, which is  $1111011_2$ . Next, add 1 to form the two's complement:

$$\begin{array}{r} 1111011_2 \\ + \quad 1 \\ \hline 1111100_2 \end{array}$$

Thus, the two's complement representation of  $-4_{10}$  is  $1111100_2$ .

To be sure you have the idea, look at a second example. How do you express  $-17_{10}$  as an 8-bit two's complement number? Start with binary representation of  $+17_{10}$ , which is  $00010001_2$ . Take the one's complement by changing all the 0's to 1's and 1's to 0's. Thus, the one's complement of  $+17_{10}$  is  $11101110_2$ . Next find the two's complement by adding 1:

$$\begin{array}{r} 11101110_2 \\ + \quad 1 \\ \hline 11101111_2 \end{array}$$

Figure 2 compares unsigned two's complement and one's complement numbers. Several 8-bit patterns are shown in the left column, while the other three columns show the decimal number represented by these patterns.

BIT PATTERN	UNSIGNED BINARY	2's COMPLEMENT	1's COMPLEMENT
00000000	0	0	+0
00000001	1	+1	+1
00000010	2	+2	+2
00000011	3	+3	+3
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
01111100	124	+124	+124
01111101	125	+125	+125
01111110	126	+126	+126
01111111	127	+127	+127
10000000	128	-128	-127
10000001	129	-127	-126
10000010	130	-126	-125
10000011	131	-125	-124
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
11111100	252	-4	-3
11111101	253	-3	-2
11111110	254	-2	-1
11111111	255	-1	-0

Figure 2

Table of bit pattern values for unsigned binary, 2's complement and 1's complement numbers.

Notice that the range of 8-bit two's complement numbers is from  $-128_{10}$  to  $+127_{10}$ . Notice also that there is only one representation for 0.

If this table included all  $256_{10}$  possible 8-bit patterns, you could look up any pattern to see what number it represents. The patterns that have 0 as their MSB are easy to determine without a table (the pattern represents the binary number directly). But what decimal number is represented by the two's complement number  $11110011_2$ ? You should know that this represents some negative number because the MSB is a 1.

Actually, you can determine the value very easily by taking the two's complement to find the equivalent positive number. Remember, you find the two's complement by taking the one's complement and adding 1. The one's complement is  $00001100_2$ . Thus, the two's complement is:

$$\begin{array}{r} 00001100_2 \\ + \quad \quad 1 \\ \hline 00001101_2 \quad \text{or } +13_{10} \end{array}$$

Since the two's complement of  $11110011_2$  represents  $+13_{10}$ , then  $11110011_2$  must equal  $-13_{10}$ .

## TWO'S COMPLEMENT ARITHMETIC

In the previous discussion you saw that signed numbers are represented in microprocessors in two's complement form. Now you will see why.

In digital electronic devices, such as computers, simple circuits cost less and operate faster than more complex ones. Two's complement numbers are used with arithmetic because they allow the simplest, cheapest, and fastest circuits.

A characteristic of the two's complement system is that both signed and unsigned numbers can be added by the same circuit. For example, suppose you wish to add the unsigned numbers  $132_{10}$  and  $14_{10}$ . The addition would look like this:

$$\begin{array}{r}
 \text{Addend:} \quad 10000100 \quad 132_{10} \\
 \text{Augend:} \quad \underline{00001110} \quad + 14_{10} \\
 \text{Sum:} \quad \quad 10010010 \quad 146_{10}
 \end{array}$$

As you know, the microprocessor has an ALU circuit that can add unsigned binary numbers in this manner. The adder of the ALU is designed so that when the bit pattern  $10000100_2$  appears at one input and  $00001110_2$  appears at the other, the bit pattern  $10010010_2$  appears at the output.

The question arises, "How does the ALU know that the bit patterns at the inputs represent unsigned numbers and not two's complement numbers?" The answer is simple, "it doesn't!"

The ALU always adds as if the inputs were unsigned binary numbers. Nevertheless, it still produces the correct sum even if the inputs are signed two's complement numbers.

Look at the example given above. If you assume that the inputs are two's complement signed numbers, then the addend, augend, and sum are:

$$\begin{array}{r}
 \text{Addend:} \quad 10000100_2 \quad -124_{10} \\
 \text{Augend:} \quad \underline{00001110_2} \quad + 14_{10} \\
 \text{Sum:} \quad \quad 10010010_2 \quad -110_{10}
 \end{array}$$



Notice that the bit patterns are the same. Only the meaning of the bit patterns have changed. In the first example, we assumed that the bit patterns represented unsigned numbers and the adder produced the proper unsigned result. In the second example, we assumed that the bit patterns represented signed numbers. Again, the adder produced the proper signed result.

This proves a very important point. The adder in the ALU always adds bit patterns as if they are unsigned numbers. It is our interpretation of these bit patterns that decides if unsigned or signed numbers are indicated. The advantage of two's complement is that the bit patterns can be interpreted either way. This allows us to work with either signed or unsigned numbers without requiring different circuits for each.

Two's complement arithmetic also simplifies the arithmetic logic unit in another way. All microprocessors have a subtract instruction. Thus, the ALU must be able to subtract one number from another. However, if this required a separate subtraction circuit, the ALU would be more complex and costly. Fortunately, two's complement arithmetic allows the ALU to subtract using an adder circuit. That is, the MPU uses the same circuit to add and subtract.

The MPU subtracts by a binary addition process. To see how this works, it may be helpful to look at a similar process with the decimal number system. The decimal equivalent of two's complement is called ten's complement. Since you are more familiar with the decimal number system, let's briefly examine ten's complement arithmetic.

## Ten's Complement Arithmetic

An easy way to illustrate ten's complement is to consider an analogy. Visualize an automobile odometer or mileage indicator. Generally, this is a 6-digit device that indicates mileage between 00,000.0 and 99,999.9 miles. Let's ignore the tenths digit and concentrate on the other five.

In an automobile, the odometer generally operates in only one direction, forward. However, consider what happens if it is turned backwards instead. Starting at +3 miles, the count would proceed backwards as follows:

00,003  
00,002  
00,001  
00,000  
99,999  
99,998  
99,997  
etc.

It is easy to visualize that 99,999 represents  $-1$  mile. Also 99,998 represents  $-2$  miles; 99,997 represents  $-3$  miles; etc. This is how signed numbers are represented in ten's complement form.

Once you accept this system for representing positive and negative numbers, you can perform arithmetic with these signed numbers. For example, if you add  $+3$  and  $-2$ , the result is  $+1$ . Using the system developed above,  $+3$  is represented by 00003 while  $-2$  is represented by 99,998. Thus, the addition looks like this:

$$\begin{array}{r}
 00003 \quad +3 \\
 + 99998 \quad -2 \\
 \hline
 100001 \quad +1
 \end{array}$$

Discard final carry  $\xrightarrow{\hspace{1.5cm}}$

If you now discard the final carry on the left side of the sum, the answer is 00001, the representation of  $+1$ . You can also find the ten's complement of a digit by subtracting the digit from ten. For example, the ten's complement of 6 is 4 since  $10 - 6 = 4$ . To complement a number containing more than one digit, raise ten to a power equal to the total number of digits, then subtract the number from it. As an example, to obtain the ten's

complement of  $654_{10}$  first raise ten to the third power since there are three digits in the number. Then subtract 654 from the result.

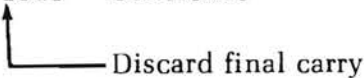
$$\begin{array}{r}
 10^3 = 1000 \\
 -654 \\
 \hline
 346
 \end{array}$$

Thus, the ten's complement of  $654_{10}$  is  $346_{10}$ .

Once you find the ten's complement, you can subtract one number from another by an indirect method using only addition. Most of us have learned to subtract like this:

$$\begin{array}{r}
 \text{Minuend:} \quad 973 \\
 \text{Subtrahend:} \quad -654 \\
 \hline
 \text{Difference:} \quad 319
 \end{array}$$

However, you can arrive at the same answer by using ten's complement of the subtrahend and adding. Recall that the ten's complement of  $654_{10}$  is  $346_{10}$ . Let's compare these two methods of subtraction:

<u>STANDARD METHOD</u>		<u>TEN'S COMPLEMENT METHOD</u>	
Minuend	973	973	Minuend
Subtrahend	-654	+346	Ten's complement of Subtrahend
Difference	319	1319	Difference
			

Notice that when you use the ten's complement method, the answer is too large, by  $1000_{10}$ . However, you can still arrive at the correct answer by simply discarding the final carry.

While the ten's complement method of subtraction works, it is not readily used because it is more complex than the standard method. In fact, it does not eliminate subtraction entirely since the ten's complement itself is found by subtraction.

The binary equivalent of ten's complement is two's complement. It overcomes the disadvantage of ten's complement in that the two's complement can be formed without any subtraction at all. Recall that you can form the two's complement of a binary number by changing all the 0's to 1's and all the 1's to 0's and then adding. Let's examine two's complement arithmetic in more detail.

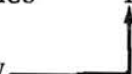
## Two's Complement Subtraction

As in ten's complement arithmetic, you can form the two's complement by subtracting from a power of the base or radix (two). However, because the MPU cannot subtract directly, it uses the method given earlier for finding the two's complement. Once the two's complement is formed, the MPU can subtract indirectly by adding the two's complement of the subtrahend to the minuend.

To illustrate this point, observe the following two ways of subtracting  $26_{10}$  from  $69_{10}$ . The two numbers are expressed as they would appear to an 8-bit microprocessor. The standard method of subtraction looks like this:

Minuend	$01000101_2$	$69$
Subtrahend	$-00011010_2$	$-26$
Difference	$00101011_2$	$43$

While this method works fine on paper, it's of little use to the microprocessor since the MPU has no subtraction circuitry. However, the MPU can still perform subtraction by the indirect method of adding the two's complement of the subtrahend to the minuend:

	Minuend	$01000101$
Two's complement of	Subtrahend	$+11100110$
	Difference	$100101011$
	Discard final carry	

This illustrates a major reason for using the two's complement system to represent signed numbers. It allows the MPU to subtract and add with the same circuit.

How microprocessors subtract is of little importance to the people who use them. Most microprocessors have a subtract instruction. This instruction is used like any other, without regard for how the operation is implemented internally. When the subtract instruction is implemented, the MPU automatically takes care of operations like complementing the subtrahend, adding, and discarding the carry. The procedure has been explained here so you can appreciate the importance of two's complement arithmetic.

## Arithmetic With Signed Numbers

There are many applications in which the microprocessor must work with signed numbers. In these cases, signed numbers are represented in two's complement form. While this greatly simplifies the circuitry of the MPU, it places an extra burden on the user. The programmer must ensure that all signed numbers are entered into the microprocessor in two's complement form. Also, the resulting data produced by the MPU may be in two's complement form. Here's how an 8-bit MPU handles signed numbers.

### ADDING POSITIVE NUMBERS

Assume that the MPU is to add the two positive numbers +7 and +3. Since an 8-bit MPU is assumed, the arithmetic operation looks like this:

$$\begin{array}{r}
 \underline{0000}111 \quad +7 \\
 + \underline{0000}011 \quad +3 \\
 \hline
 00001010 \quad +10
 \end{array}$$

The sign bits are underlined. Remember, with signed numbers, the MSB is the sign bit. A 0 represents "+" and a 1 represents "-." In this example, you added +7 and +3 to form a sum of +10<sub>10</sub>. You know that all three numbers are positive since the MSB's are all 0's.

While the operation seems straightforward enough, it is easy to make an error when adding positive numbers. Remember, the highest 8-bit positive number you can represent in two's complement form is +127<sub>10</sub>. If the sum exceeds this value, an error occurs. For example, suppose you attempt to add +65<sub>10</sub> to +67<sub>10</sub>. The MPU adds the numbers as if they were unsigned binary:

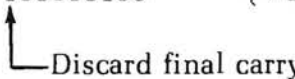
$$\begin{array}{r}
 \underline{0}1000001 \\
 + \underline{0}1000011 \\
 \hline
 \underline{1}0000100
 \end{array}$$

If the answer is interpreted as a two's complement number, an error has occurred. You have added two positive numbers and yet the answer appears to be negative since the MSB of the sum is 1. This is called two's complement overflow. It occurs when the sum exceeds +127<sub>10</sub>. Many microprocessors have a way of detecting this condition, which we will discuss later.

### ADDING POSITIVE AND NEGATIVE NUMBERS

The real advantage of the two's complement system is best illustrated when you add numbers with unlike signs. For example, assume that an 8-bit microprocessor is to add +7 and -3. Remember, since these are signed numbers, they must be represented in two's complement form. That is, +7 is represented as  $00000111_2$  while -3 is represented as  $11111101_2$ . If these two numbers are added, the sum will be:

$$\begin{array}{r}
 \text{Addend} \quad 00000111 \quad (+7) \\
 \text{Augend} \quad + 11111101 \quad + (-3) \\
 \hline
 \text{Sum} \quad 100000100 \quad (+4)
 \end{array}$$



Notice that the sum is correct if you ignore the final carry bit. Keep in mind that the MPU adds the two numbers as if they were unsigned binary numbers. It is merely our interpretation of the answer that makes the system work for signed numbers.

The system also works when the negative number is larger. For example, when -9 is added to +8 the result should be -1. Remember, the signed numbers must be represented in two's complement form:

$$\begin{array}{r}
 \text{Addend} \quad 11110111 \quad (-9) \\
 \text{Augend} \quad + 00001000 \quad + (+8) \\
 \hline
 \text{Sum} \quad 11111111 \quad (-1)
 \end{array}$$

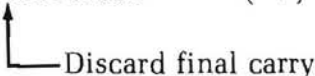
Notice that the sum is the two's complement representation for -1.

### ADDING NEGATIVE NUMBERS

The final case involves two negative numbers. If both numbers are negative, then the sum should also be negative.

For example, suppose the MPU is to add  $-3$  to  $-4$ . Obviously, the result should be  $-7$ . The two signed numbers must be represented in two's complement form. That is,  $-3$  must be represented as  $11111101_2$  while  $-4$  must be represented as  $11111100_2$ . The MPU adds these two bit patterns as if they were unsigned binary numbers. Thus the result is:

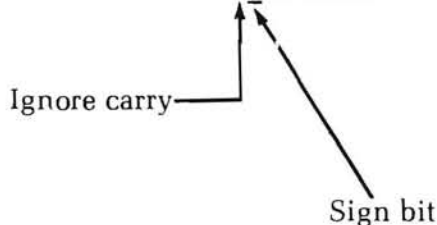
Addend	11111101	(-3)
Augend	+ 11111100	+(-4)
Sum	111111001	(-7)



Once again, the answer is correct if you ignore the final carry bit.

When you add two negative numbers, you must remember the capacity of the MPU. The largest negative number that can be represented by 8 bits is  $-128_{10}$ . If the sum exceeds this value, the sum will appear to be in error. For example, suppose you add  $-120_{10}$  to  $-18_{10}$ .

	10001000	(-120)
	+ 11101110	+(-18)
	101110110	(-138)



Notice that the sign bit in the sum is 0, representing a positive number. Thus, the MPU has added two negative numbers and has produced a positive result. This apparent error is caused by exceeding the 8-bit capacity of the microprocessor. This is another example of two's complement overflow.

## BOOLEAN OPERATIONS

Along with the basic mathematical processes examined earlier, the microprocessor can manipulate binary numbers logically. This system was conceived using the theorems developed by mathematician George Boole. As a result, this branch of binary mathematics is given the name Boolean Algebra. In this section, the Boolean operations performed by the microprocessor will be examined. A more detailed description of Boolean Algebra is provided in the Heathkit/Zenith Education Systems course titled "Digital Techniques."

### AND Operation

The AND function produces the logical product of two or more logic variables. That is, the logic product of an AND operation is logic 1 if all of the variable inputs are logic 1. If any of the variable inputs are logic 0, the logical product is 0. This process can be represented by the formula  $(A \cdot B = C)$  where A and B represent input variables (logic 1 or 0) and C represents the output or logical product of the AND operation. The AND function is designated by a dot between variables. Do not confuse it with the mathematical multiplication sign.

Figure 3 is a "truth table" for a 2-variable AND function. The 1's and 0's represent all of the possible logic combinations. Thus, you can see that the AND function is a sort of "all-or-nothing" operation. Unless all the input variables are logic 1, the output logic cannot be logic 1.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3

Truth table for a two-variable AND function.



When the microprocessor implements the logic AND operation, one 8-bit binary number is ANDed with a second 8-bit binary number. Refer to the example below for an illustration of this process.

	<u>8-BIT</u>		<u>8-BIT</u>		<u>RESULTS OF</u>	
	<u>NUMBER</u>		<u>NUMBER</u>		<u>AND OPERATION</u>	
MSB	1	•	1	=	1	MSB
	0	•	0	=	0	
	0	•	1	=	0	
	1	•	0	=	0	
	1	•	1	=	1	
	0	•	1	=	0	
	1	•	0	=	0	
LSB	0	•	0	=	0	LSB

Although more than two logic variables can be ANDed together, the microprocessor operates on only two variables at a time. Now try one more example of the AND operation. AND 10011101 with 11000110.

1	•	1	=	1	MSB
0	•	1	=	0	
0	•	0	=	0	
1	•	0	=	0	
1	•	0	=	0	
1	•	1	=	1	
0	•	1	=	0	
1	•	0	=	0	LSB

## OR Operation

The OR function (more precisely, inclusive OR) produces the logical sum of two or more logic variables. That is, the logical sum of an OR operation is logic 1 if either input is logic 1. The logic sum is 0 if ALL of the input variables are logic 0. This process can be represented by the formula ( $A + B = C$ ) where A and B represent input variables and C represents the output or logical sum of the OR operation. The OR function is designated by a plus sign, or in some cases, a circle dot  $\odot$ , between the variables. Do not confuse the plus sign with the mathematical add sign.

Figure 4 is a “truth table” for a 2-variable OR function. The 1’s and 0’s represent all of the possible logic combinations. Thus, you can see that the OR function is a sort of “either or both” operation. If either or both input variables are logic 1, the output must be logic 1.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4

Truth table for a two-variable OR function.

When the microprocessor implements the logic OR operation, one 8-bit binary number is ORed with a second 8-bit binary number. This process can be seen in the example given below.

	<u>8-BIT</u>		<u>8-BIT</u>		<u>RESULTS OF</u>	
	<u>NUMBER</u>		<u>NUMBER</u>		<u>OR OPERATION</u>	
MSB	1	+	1	=	1	MSB
	0	+	0	=	0	
	0	+	1	=	1	
	1	+	0	=	1	
	1	+	1	=	1	
	0	+	1	=	1	
	1	+	0	=	1	
LSB	0	+	0	=	0	

As with the AND function, two or more logic variables can be ORed together. However, the microprocessor operates only on two variables at a time. Now try one more example of the OR operation. OR  $10011101_2$  with  $11000101_2$ .

$$\begin{array}{l}
 1 + 1 = 1 \text{ MSB} \\
 0 + 1 = 1 \\
 0 + 0 = 0 \\
 1 + 0 = 1 \\
 1 + 0 = 1 \\
 1 + 1 = 1 \\
 0 + 0 = 0 \\
 1 + 1 = 1 \text{ LSB}
 \end{array}$$

## Exclusive OR Operation

The Exclusive OR (EOR or XOR) function performs a logical test for “equality” between two logic variables. That is, if two variable inputs are equal (both logic 1 or 0), the output or result of the EOR operation is logic 0. If the inputs are not equal (one is logic 1, the other logic 0), the output is logic 1. This can be represented by the formula  $(A \oplus B = C)$  where A and B represent input variables and C represents the output or result. The EOR function is designated by a circled plus sign between the variables.

Figure 5 is a “truth table” for the EOR function. The 1’s and 0’s represent all of the possible logic combinations. You can see that the EOR function is a sort of “either but not both” operation. Hence, either input can be logic 1 or logic 0, but not both, for a logic 1 output.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5

Truth table for a two-variable EOR function.

When the microprocessor implements the logic EOR operation, one 8-bit binary number is exclusively ORed with a second 8-bit binary number. This process is shown in the following example.

	8-BIT NUMBER		8-BIT NUMBER		RESULTS OF EOR OPERATION	
MSB	1	$\oplus$	1	=	0	MSB
	0	$\oplus$	0	=	0	
	0	$\oplus$	1	=	1	
	1	$\oplus$	0	=	1	
	1	$\oplus$	1	=	0	
	0	$\oplus$	1	=	1	
	1	$\oplus$	0	=	1	
LSB	0	$\oplus$	0	=	0	LSB

Now try one more example of the EOR operation. EOR  $10011101_2$  with  $11000101_2$ .

$$\begin{aligned}
 1 \oplus 1 &= 0 \text{ MSB} \\
 0 \oplus 1 &= 1 \\
 0 \oplus 0 &= 0 \\
 1 \oplus 0 &= 1 \\
 1 \oplus 0 &= 1 \\
 1 \oplus 1 &= 0 \\
 0 \oplus 0 &= 0 \\
 1 \oplus 1 &= 0 \text{ LSB}
 \end{aligned}$$

## Invert Operation

The invert operation performs a direct complement of a single input variable. That is, a logic 1 input will produce a logic 0 output and a logic 0 input will produce a logic 1 output. This process is represented by the "truth table" shown in Figure 6.

INPUT	OUTPUT
A	$\bar{A}$
1	0
0	1

Figure 6

Truth table for an INVERT function.

Note that the complement of A is  $\bar{A}$ . The bar above the A indicates that A has been inverted, and is read "not A." Conversely, the complement of  $\bar{A}$  is A.

When the microprocessor implements the logic invert operation, the 8-bit binary number is complemented. This operation is also known as 1's complement. Thus, the complement of  $11010110_2$  is  $00101001_2$ . As with the previous logic operations, the invert function operates on each individual bit of the 8-bit number.

## *Appendix D*

# **INSTRUCTION SET**

Instruction set data supplied  
courtesy of Intel Corporation.  
Modifications made to reflect  
instruction coding unique to  
MACRO-86.

# REF                      REFERENCES                      REF

## FOR INSTRUCTION SET

### Key to following Instruction Set Reference Pages

IDENTIFIER	USED IN	EXPLANATION
destination	data transfer, bit manipulation	A register or memory location that may contain data operated on by the instruction, and which receives (is replaced by) the result of the operation.
source	data transfer, arithmetic, bit manipulation	A register, memory location or immediate value that is used in the operation, but is not altered by the instruction.
source-table	XLAT	Name of memory translation table addressed by register BX.
target	JMP, CALL	A label to which control is to be transferred directly, or a register or memory location whose <i>content</i> is the address of the location to which control is to be transferred indirectly.
short-label	cond. transfer, iteration control	A label to which control is to be conditionally transferred; must lie within -128 to +127 bytes of the first byte of the next instruction.
accumulator	IN, OUT	Register AX for word transfers, AL for bytes.
port	IN, OUT	An I/O port number; specified as an immediate value of 0-255, or register DX (which contains port number in range 0-64k).
source-string	string ops.	Name of a string in memory that is addressed by register SI; used only to identify string as byte or word and specify segment override, if any. This string is used in the operation, but is not altered.
dest-string	string ops.	Name of string in memory that is addressed by register DI; used only to identify string as byte or word. This string receives (is replaced by) the result of the operation.
count	shifts, rotates	Specifies number of bits to shift or rotate; written as immediate value 1 or register CL (which contains the count in the range 0-255).
interrupt-type	INT	Immediate value of 0-255 identifying interrupt pointer number
optional-pop-value	RET	Number of bytes (0-64k, ordinarily an even number) to discard from stack.
external-opcode	ESC	Immediate value (0-63) that is encoded in the instruction for use by an external processor

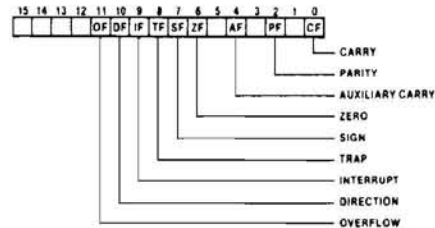
# REF                      REFERENCES                      REF

## FOR INSTRUCTION SET

### Key to Operand Types

IDENTIFIER	EXPLANATION
(no operands)	No operands are written
register	An 8- or 16-bit general register
reg 16	An 16-bit general register
seg-reg	A segment register
accumulator	Register AX or AL
immediate	A constant in the range 0-FFFFH
immed8	A constant in the range 0-FFH
memory	An 8- or 16-bit memory location <sup>(1)</sup>
mem8	An 8-bit memory location <sup>(1)</sup>
mem16	A 16-bit memory location <sup>(1)</sup>
source-table	Name of 256-byte translate table
source-string	Name of string addressed by register SI
dest-string	Name of string, addressed by register DI
DX	Register DX
short-label	A label within -128 to +127 bytes of the end of the instruction
near-label	A label in current code segment
far-label	A label in another code segment
near-proc	A procedure in current code segment
far-proc	A procedure in another code segment
memptr16	A word containing the offset of the location in the current code segment to which control is to be transferred <sup>(1)</sup>
memptr32	A doubleword containing the offset and the segment base address of the location in another code segment to which control is to be transferred <sup>(1)</sup>
regptr16	A 16-bit general register containing the offset of the location in the current code segment to which control is to be transferred
repeat	A string instruction repeat prefix

<sup>(1)</sup> Any addressing mode—direct, register indirect, based, indexed, or based indexed—may be used (see section 2.8).



### Effective Address Calculation Time

EA COMPONENTS	CLOCKS*
Displacement Only	6
Base or Index Only (BX, BP, SI, DI)	5
Displacement + Base or Index (BX, BP, SI, DI)	9
Base + Index BP + DI, BX + SI BP + SI, BX + DI	7 8
Displacement + Base + Index BP + DI + DISP BX + SI + DISP BP + SI + DISP BX + DI + DISP	11 12

\*Add 2 clocks for segment override

# REF                      REFERENCES                      REF

## FOR INSTRUCTION SET

### “reg” Field Bit Assignments:

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

### “mod” Field Bit Assignments:

mod xxx r/m

mod	Displacement
00	DISP = 0*, disp-low and disp-high are absent
01	DISP = disp-low sign-extended to 16-bits, disp-high is absent
10	DISP = disp-high:disp-low
11	r/m is treated as a “reg” field

### “r/m” Field Bit Assignments:

r/m	Operand Address
000	(BX) + (SI) + DISP
001	(BX) + (DI) + DISP
010	(BP) + (SI) + DISP
011	(BP) + (DI) + DISP
100	(SI) + DISP
101	(DI) + DISP
110	(BP) + DISP
111	(BX) + DISP

DISP follows 2nd byte of instruction (before data if required).

\*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low.



**AAA****ASCII ADJUST  
FOR ADDITION****AAA****Operation:**

if  $((AL) \& OFH) > 9$  or  $(AF) = 1$  then  
 $(AL) \leftarrow (AL) + 6$   
 $(AH) \leftarrow (AH) + 1$   
 $(AF) \leftarrow 1$   
 $(CF) \leftarrow (AF)$   
 $(AL) \leftarrow (AL) \& OFH$

**Flags Affected:**

AF, CF.  
 OF, PF, XF, ZF undefined

**Description:**

AAA (ASCII Adjust for Addition) changes the contents of register AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAA updates AF and CF; the content of OF, PF, SF and ZF is undefined following execution of AAA.

**Encoding:**

00110111
----------

AAA Operands	Clocks	Transfers	Bytes	AAA Coding Example
(no operands)	4	—	1	AAA

**AAD****ASCII ADJUST  
FOR DIVISION****AAD****Operation:**

$$(AL) \leftarrow (AH) * 0AH + (AL)$$

$$(AH) \leftarrow 0$$
**Flags Affected:**

PF, SF, ZF.  
AF, CF, OF undefined

**Description:**

AAD (ASCII Adjust for Division) modifies the numerator in AL *before* dividing two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH must be zero

for the subsequent DIV to produce the correct result. The quotient is returned in AL, and the remainder is returned in AH; both high-order half-bytes are zeroed. AAD updates PF, SF and ZF; the content of AF, CF and OF is undefined following execution of AAD.

**Encoding:**

11010101	00001010
----------	----------

AAD Operands	Clocks	Transfers	Bytes	AAD Coding Example
(no operands)	60	—	2	AAD

**AAM****ASCII ADJUST  
FOR MULTIPLY****AAM****Operation:**

(AH) ← (AL) / OAH  
 (AL) ← (AL) % OAH

**Flags Affected:**

PF, SF, ZF.  
 AF, CF, OF undefined

**Description:**

AAM (ASCII Adjust for Multiply) corrects the result of a previous multiplication of two valid unpacked decimal operands. A valid 2-digit unpacked decimal number is derived from the content of AH and AL and is

returned to AH and AL. The high-order half-bytes of the multiplied operands must have been 0H for AAM to produce a correct result. AAM updates PF, SF and ZF; the content of AF, CF and OF is undefined following execution of AAM.

**Encoding:**

11010100	00001010
----------	----------

AAM Operands	Clocks	Transfers	Bytes	AAM Coding Example
(no operands)	83	—	1	AAM

**AAS****ASCII ADJUST  
FOR SUBTRACTION****AAS****Operation:**

if  $((AL) \& OFH) > 9$  or  $(AF) = 1$  then  
 $(AL) \leftarrow (AL) - 6$   
 $(AH) \leftarrow (AH) - 1$   
 $(AF) \leftarrow 1$   
 $(CF) \leftarrow (AF)$   
 $(AL) \leftarrow (AL) \& OFH$

**Flags Affected:**

AF, CF.  
 OF, PF, SF, ZF undefined

**Description:**

AAS (ASCII Adjust for Subtraction) corrects the result of a previous subtraction of two valid unpacked decimal operands (the destination operand must have been specified as

register AL). AAS changes the content of AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAS updates AF and CF; the content of OF, PF, SF and ZF is undefined following execution of AAS.

**Encoding:**

00111111
----------

AAS Operands	Clocks	Transfers	Bytes	AAS Coding Example
(no operands)	4	—	1	AAS

# ADC

# ADD WITH CARRY

# ADC

## Operation:

if (CF) = 1 then (DEST)  $\leftarrow$  (LSRC)  
+ (RSRC) + 1  
else (DEST)  $\leftarrow$  (LSRC) + (RSRC)

## Flags Affected:

AF, CF, OF, PF, SF, ZF

## Description:

### *ADC destination, source*

ADC (Add with Carry) sums the operands, which may be bytes or words, adds one if CF is set and replaces the destination operand with the result. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADC updates AF, CF, OF, PF, SF and ZF. Since ADC incorporates a carry from a previous operation, it can be used to write routines to add numbers longer than 16 bits.

**ADC****ADD WITH CARRY****ADC****Encoding:****Memory or Register Operand with Register Operand:**

000100d w	mod reg r/m
-----------	-------------

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG  
 else LSRC = EA, RSRC = REG, DEST = EA

**Immediate Operand to Memory or Register Operand:**

100000s w	mod 010 r/m	data	data if s:w=01
-----------	-------------	------	----------------

LSRC = EA, RSRC = data, DEST = EA

**Immediate Operand to Accumulator:**

0001010 w	data	data if w=1
-----------	------	-------------

if w = 0 then LSRC = AL, RSRC = data, DEST = AL  
 else LSRC = AX, RSRC = data, DEST = AX

ADC Operands	Clocks*	Transfers	Bytes	ADC Coding Examples
register, register	3	—	2	ADC AX, SI
register, memory	9(13) + EA	1	2-4	ADC DX, BETA [SI]
memory, register	16(24) + EA	2	2-4	ADC ALPHA [BX + SI], DI
register, immediate	4	—	3-4	ADC BX, 256
memory, immediate	17(25) + EA	2	3-6	ADC GAMMA, 30H
accumulator, immediate	4	—	2-3	ADC AL, 5

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# ADD

# ADDITION

# ADD

**Operation:** $(DEST) \leftarrow (LSRC) + (RSRC)$ **Flags Affected:**

AF, CF, OF, PF, SF, ZF

**Description:****ADD** *destination, source*

The sum of the two operands, which may be bytes or words, replaces the destination operand. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADD updates AF, CF, OF, PF, SF and ZF.

# ADD

# ADDITION

# ADD

## Encoding:

### Memory or Register Operand with Register Operand:

000000 d w	mod reg r/m
------------	-------------

if  $d = 1$  then LSRC = REG, RSRC = EA, DEST = REG  
 else LSRC = EA, RSRC = REG, DEST = EA

### Immediate Operand to Memory or Register Operand:

100000 s w	mod 000 r/m	data	data if s:w=01
------------	-------------	------	----------------

LSRC = EA, RSRC = data, DEST = EA

### Immediate Operand to Accumulator:

0000010 w	data	data if w=1
-----------	------	-------------

if  $w = 0$  then LSRC = AL, RSRC = data, DEST = AL  
 else LSRC = AX, RSRC = data, DEST = AX

ADD Operands	Clocks*	Transfers	Bytes	ADD Coding Examples
register, register	3	—	2	ADD CX, DX
register, memory	9(13) + EA	1	2-4	ADD DI, [BX].ALPHA
memory, register	16(24) + EA	2	2-4	ADD TEMP, CL
register, immediate	4	—	3-4	ADD CL, 2
memory; immediate	17(25) + EA	2	3-6	ADD ALPHA, 2
accumulator, immediate	4	—	2-3	ADD AX, 200

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.



# AND

## AND LOGICAL

# AND

### Operation:

(DEST) ← (LSRC) & (RSRC)  
(CF) ← 0  
(OF) ← 0

### Flags Affected:

CF, OF, PF, SF, ZF.  
AF undefined

### Description:

**AND** *destination, source*

AND performs the logical “and” of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both corresponding bits of the original operands are set; otherwise the bit is cleared.

# AND                      AND LOGICAL                      AND

## Encoding:

### Memory or Register Operand with Register Operand:

001000d	w	mod reg r/m
---------	---	-------------

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG  
 else LSRC = EA, RSRC = REG, DEST = EA

### Immediate Operand to Memory or Register Operand:

1000000w	mod 100 r/m	data	data if w=1
----------	-------------	------	-------------

LSRC = EA, RSRC = data, DEST = EA

### Immediate Operand to Accumulator:

0010010w	data	data if w=1
----------	------	-------------

if w = 0 then LSRC = AL, RSRC = data, DEST = AL  
 else LSRC = AX, RSRC = data, DEST = AX

AND Operands	Clocks*	Transfers	Bytes	AND Coding Examples
register, register	3	—	2	AND AL, BL
register, memory	9(13) + EA	1	2-4	AND CX, FLAG_WORD
memory, register	16(24) + EA	2	2-4	AND ASCII [DI], AL
register, immediate	4	—	3-4	AND CX, 0F0H
memory, immediate	17(25) + EA	2	3-6	AND BETA, 01H
accumulator, immediate	4	—	2-3	AND AX, 01010000B

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# CALL CALL PROCEDURE CALL

## Operation:

```

if Inter-Segment then
  (SP) ← (SP) - 2
  ((SP) + 1:(SP)) ← (CS)
  (CS) ← SEG
(SP) ← (SP) - 2
((SP) + 1:(SP)) ← (IP)
(IP) ← DEST

```

## Flags Affected:

None

## Description:

### CALL *procedure-name*

CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. The assembler generates a different type of CALL instruction depending on whether the programmer has defined the procedure name as NEAR or FAR. For control to return properly, the type of CALL instruction must match the type of RET instruction that exits from the procedure. (The potential for a mismatch exists if the procedure and the CALL are contained in separately assembled programs.) Different forms of the CALL instruction allow the address of the target procedure to be obtained from the instruction itself (direct CALL) or from a memory location or register referenced by the instruction (indirect CALL). In the following descriptions, bear in mind that the processor automatically adjusts IP to point to the next instruction to be *executed* before saving it on the stack.

For an intrasegment direct CALL, SP (the stack pointer) is decremented by two and IP is pushed onto the stack. The target procedure's relative displacement (up to  $\pm 32k$ ) from the CALL instruction is then added to the instruction pointer. This CALL instruction

form is "self-relative" and appropriate for position-independent (dynamically relocatable) routines in which the CALL and its target are moved together in the same segment.

An intrasegment indirect CALL may be made through memory or a register. SP is decremented by two; IP is pushed onto the stack. The target procedure offset is obtained from the memory word or 16-bit general register referenced in the instruction and replaces IP.

For an intersegment direct CALL, SP is decremented by two, and CS is pushed onto the stack. CS is replaced by the segment word contained in the instruction. SP again is decremented by two. IP is pushed onto the stack and replaced by the offset word in the instruction.

For an intersegment indirect CALL (which only may be made through memory), SP is decremented by two, and CS is pushed onto the stack. CS is then replaced by the content of the second word of the doubleword memory pointer referenced by the instruction. SP again is decremented by two, and IP is pushed onto the stack and replaced by the content of the first word of the doubleword pointer referenced by the instruction.

# CALL CALL PROCEDURE CALL

## Encoding:

### Intra-segment direct:

11101000	disp-low	disp-high
----------	----------	-----------

DEST = (EA)

### Intra-Segment Indirect:

11111111	mod 010 r/m
----------	-------------

DEST = (IP) + disp

### Inter-Segment Direct:

10011010	offset-low	offset-high
	seg-low	seg-high

DEST = offset, SEG = seg

### Inter-Segment Indirect:

11111111	mod 011 r/m
----------	-------------

DEST = (EA), SEG = (EA + 2)

CALL Operands	Clocks*	Tranfers	Bytes	CALL Coding Examples
near-proc	19(23)	1	3	CALL NEAR__PROC
far-proc	28(36)	2	5	CALL FAR__PROC
memptr 16	21(29) + EA	2	2-4	CALL PROC__TABLE [SI]
regptr 16	16(24)	1	2	CALL AX
memptr 32	37(57) + EA	4	2-4	CALL [BX].TASK [SI]

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

**CBW****CONVERT BYTE  
TO WORD****CBW****Operation:**

if (AL) &lt; 80H then (AH) ← 0 else (AH) &lt;&lt; FFH

**Flags Affected:**

None

**Description:**

CBW (Convert Byte to Word) extends the sign of the byte in register AL throughout register AH. CBW does not affect any flags. CBW can be used to produce a double-length (word) dividend from a byte prior to performing byte division.

**Encoding:**

10011000

CBW Operands	Clocks	Transfers	Bytes	CBW Coding Example
(no operands)	2	—	1	CBW

**CLC****CLEAR CARRY****CLC****Operation:**

(CF) ← 0

**Flags Affected:**

CF

**Description:**

CLC (Clear Carry flag) zeroes the carry flag (CF) and affects no other flags. It (and CMC and STC) is useful in conjunction with the RCL and RCR instructions.

**Encoding:**

11111000
----------

CLC Operands	Clocks	Transfers	Bytes	CLC Coding Example
(no operands)	2	—	1	CLC

**CLD****CLEAR DIRECTION  
FLAG****CLD****Operation:**

(DF) ← 0

**Flags Affected:**

DF

**Description:**

CLD (Clear Direction flag) zeroes DF causing the string instructions to auto-increment the SI and/or DI index registers. CLD does not affect any other flags.

**Encoding:**

11111100
----------

CLD Operands	Clocks	Transfers	Bytes	CLD Coding Example
(no operands)	2	—	1	CLD

**CLI****CLEAR INTERRUPT-  
ENABLE FLAG****CLI****Operation:**

(IF) ← 0

**Flags Affected:**

IF

**Description:**

CLI (Clear Interrupt-enable flag) zeroes IF. When the interrupt-enable flag is cleared, the 8086 and 8088 do not recognize an external interrupt request that appears on the INTR line; in other words maskable interrupts are disabled. A non-maskable interrupt appearing on the NMI line, however, is honored, as is a software interrupt. CLI does not affect any other flags.

**Encoding:**

11111010
----------

CLI Operands	Clocks	Transfers	Bytes	CLI Coding Example
(no operands)	2	—	1	CLI



# CMC

# COMPLEMENT CARRY FLAG

# CMC

**Operation:**

if (CF) = 0 then (CF)  $\leftarrow$  1 else (CF)  $\leftarrow$  0

**Flags Affected:**

CF

**Description:**

CMC (Complement Carry flag) “toggles” CF to its opposite state and affects no other flags.

**Encoding:**

11110101
----------

CMC Operands	Clocks	Transfers	Bytes	CMC Coding Example
(no operands)	2	—	1	CMC

# CMP

# COMPARE

# CMP

**Operation:**

(LSRC) - (RSRC)

**Flags Affected:**

AF, CF, OF, PF, SF, ZF

**Description:****CMP** *destination, source*

CMP (Compare) subtracts the source from the destination, which may be bytes or words, but does not return the result. The operands are unchanged, but the flags are updated and can be tested by a subsequent conditional jump instruction. CMP updates AF, CF, OF, PF,

SF and ZF. The comparison reflected in the flags is that of the destination to the source. If a CMP instruction is followed by a JG (jump if greater) instruction, for example, the jump is taken if the destination operand is greater than the source operand.

# CMP

# COMPARE

# CMP

## Encoding:

### Memory or Register Operand with Register Operand:

001110d w	mod reg r/m
-----------	-------------

if d = 1 then LSRC = REG, RSRC = EA  
 else LSRC = EA, RSRC = REG

### Immediate Operand with Memory or Register Operand:

100000s w	mod 111 r/m	data	data if s:w=01
-----------	-------------	------	----------------

LSRC = EA, RSRC = data

### Immediate Operand with Accumulator:

0011110 w	data	data if w=1
-----------	------	-------------

if w = 0 then LSRC = AL, RSRC = data  
 else LSRC = AX, RSRC = data

CMP Operands	Clocks*	Transfers	Bytes	CMP Coding Examples
register, register	3	—	2	CMP BX, CX
register, memory	9(13) + EA	—	2-4	CMP DH, ALPHA
memory, register	9(13) + EA	—	2-4	CMP [BP] + 2, SI
register, immediate	4	—	3-4	CMP BL, 02H
memory, immediate	10(14) + EA	—	3-6	CMP [BX].RADAR [DI], 3420H
accumulator, immediate	4	—	2-3	CMP AL, 00010000B

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# CMPS COMPARE STRING (BYTE OR WORD) CMPS

## Operation:

```
(LSRC) ← (RSRC)
if (DF) = 0 then
  (SI) ← (SI) + DELTA
  (DI) ← (DI) + DELTA
else
  (SI) ← (SI) - DELTA
  (DI) ← (DI) - DELTA
```

## Flags Affected:

AF, CF, OF, PF, SF, ZF

## Description:

### CMPS *destination-string,source-string*

CMPS (Compare String) subtracts the destination byte or word (addressed by DI) from the source byte or word (addressed by SI). CMPS affects the flags but does not alter either operand, updates SI and DI to point to the next string element and updates, AF, CF, OF, PF, SF and ZF to reflect the relationship of the destination element to the source element. For example, if a JG (Jump if Greater) instruction follows CMPS, the jump is taken if the des-

tinuation element is greater than the source element. If CMPS is prefixed with REPE or REPZ, the operation is interrupted as "compare while not end-of-string (CX not zero) and strings are equal (ZF = 1)." If CMPS is preceded by REPNE or REPNZ, the operation is interrupted as "compare while not end-of-string (CX not zero) and strings are not equal (ZF = 0)." Thus, CMPS can be used to find matching or differing string elements.

## Encoding:

1010011w

if w = 0 then LSRC = (SI), RSRC = (DI), DELTA = 1  
 else LSRC = (SI) + 1:(SI), RSRC = (DI) + 1:(DI), DELTA = 2

CMPS Operands	Clocks*	Transfers	Bytes	CMPS CodingExamples
dest-string, source-string	22(30)	2	1	CMPS BUFF1, BUFF2
(repeat) dest-string, source-string	9 + 22(30)/rep	2/rep	1	REP COMPS ID, KEY

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# CMPSB      COMPARE      CMPSB

## STRING BYTE

### Operation:

```
(LSRC) - (RSRC)
if (DF) = 0 then
  (SI) ← (SI) + DELTA
  (DI) ← (DI) + DELTA
else
  (SI) ← (SI) - DELTA
  (DI) ← (DI) - DELTA
```

### Flags Affected:

AF, CF, OF, PF, SF, ZF

### Description:

CMPSB (Compare String Byte) subtracts the destination byte (addressed by DI) from the source byte (addressed by SI). CMPSB affects the flags but does not alter either operand, updates SI and DI to point to the next string element and updates, AF, CF, OF, PF, SF and ZF to reflect the relationship of the destination element to the source element. For example, if a JG (Jump if Greater) instruction follows CMPSB, the jump is taken if the

destination element is greater than the source element. If CMPSB is prefixed with REPE or REPZ, the operation is interrupted as "compare while not end-of-string (CX not zero) and strings are equal (ZF = 1)." If CMPSB is preceded by REPNE or REPNZ, the operation is interrupted as "compare while not end-of-string (CX not zero) and strings are not equal (ZF = 0)." Thus, CMPSB can be used to find matching or differing string elements.

### Encoding:

```
10100110
```

LSRC = (SI), RSRC = (DI), DELTA = 1

CMPSB Operands	Clocks	Transfers	Bytes	CMPSB Coding Examples
(NO OPERANDS)	22	2	1	CMPSB
(NO OPERANDS)	9 + 22/rep	2/rep	1	REP CMPSB

# CMPSW COMPARE STRING WORD CMPSW

## Operation:

```
(LSRC) ← (RSRC)
if (DF) = 0 then
  (SI) ← (SI) + DELTA
  (DI) ← (DI) + DELTA
else
  (SI) ← (SI) - DELTA
  (DI) ← (DI) - DELTA
```

## Flags Affected:

AF, CF, OF, PF, SF, ZF

## Description:

CMPSW (Compare String Word) subtracts the destination word (addressed by DI) from the source word (addressed by SI). CMPSW affects the flags but does not alter either operand, updates SI and DI to point to the next string element and updates AF, CF, OF, PF, SF and ZF to reflect the relationship of the destination element to the source element. For example, if a JG (Jump if Greater) instruction follows CMPSW, the jump is taken if the destination element is greater

than the source element. If CMPSW is prefixed with REPE or REPZ, the operation is interrupted as “compare while not end-of-string (CX not zero) and strings are equal (ZF = 1).” If CMPSW is preceded by REPNE or REPNZ, the operation is interrupted as “compare while not end-of-string (CX not zero) and strings are not equal (ZF = 0).” Thus, CMPSW can be used to find matching or differing string elements.

## Encoding:

```
10100111
```

LSRC = (SI) + 1:(SI), RSRC = (DI) + 1:(DI), DELTA = 2

CMPSW Operands	Clocks	Transfers	Bytes	CMPSW Coding Examples
(NO OPERANDS)	30	2	1	CMPSW
(NO OPERANDS)	9 + 30/rep	2/rep	1	REP CMPSW

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

**CWD****CONVERT WORD  
TO DOUBLEWORD****CWD****Operation:**

if (AX) < 8000H then (DX) ← 0  
 else (DX) ← FFFFH

**Flags Affected:**

None

**Description:**

CWD (Convert Word to Doubleword) extends the sign of the word in register AX throughout register DX. CWD does not affect any flags. CWD can be used to produce a double-length (doubleword) dividend from a word prior to performing word division.

**Encoding:**

10011001
----------

CWD Operands	Clocks	Transfers	Bytes	CWD Coding Example
(no operands)	5	—	1	CWD

**DAA****DECIMAL ADJUST  
FOR ADDITION****DAA****Operation:**

if  $((AL) \& OFH) > 9$  or  $(AF) = 1$  then  
 $(AL) \leftarrow (AL) + 6$   
 $(AF) \leftarrow 1$   
 if  $(AL) > 9FH$  or  $(CF) = 1$  then  
 $(AL) \leftarrow (AL) + 60H$   
 $(CF) \leftarrow 1$

**Flags Affected:**

AF, CF, PF, SF, ZF  
 OF undefined

**Description:**

DAA (Decimal Adjust for Addition) corrects the result of previously adding two valid packed decimal operands (the destination operand must have been register AL). DAA changes the content of AL to a pair of valid packed decimal digits. It updates AF, CF, PF, SF and ZF; the content of OF is undefined following execution of DAA.

**Encoding:**

00100111
----------

DAA Operands	Clocks	Transfers	Bytes	DAA Coding Example
(no operands)	4	—	1	DAA



**DAS****DECIMAL ADJUST  
FOR SUBTRACTION****DAS****Operation:**

if  $((AL) \& OFH) > 9$  or  $(AF) = 1$  then  
 $(AL) \leftarrow (AL) - 6$   
 $(AF) \leftarrow 1$   
 if  $(AL) > 9FH$  or  $(CF) = 1$  then  
 $(AL) \leftarrow (AL) - 60H$   
 $(CF) \leftarrow 1$

**Flags Affected:**

AF, CF, PF, SF, ZF.  
 OF undefined

**Description:**

DAS (Decimal Adjust for Subtraction) corrects the result of a previous subtraction of two valid packed decimal operands (the destination operand must have been specified as register AL). DAS changes the content of AL to a pair of valid packed decimal digits. DAS updates AF, CF, PF, SF and ZF; the content of OF is undefined following execution of DAS.

**Encoding:**

00101111
----------

DAS Operands	Clocks	Transfers	Bytes	DAS Coding Example
(no operands)	4	—	1	DAS

# DEC

# DECREMENT

# DEC

**Operation:** $(DEST) \leftarrow (DEST) - 1$ **Flags Affected:**

AF, OF, PF, SF, ZF

**Description:**

DEC (Decrement) subtracts one from the destination operand. The operand may be a byte or a word and is treated as an unsigned binary number (see AAA and DAA). DEC updates AF, OF, PF, SF and ZF; it does not affect CF.

**Encoding:****Memory or Register Operand:**

1111111w	mod 001r/m
----------	------------

DEST = EA

**Register Operand:**

01001reg
----------

DEST = REG

DEC Operands	Clocks*	Transfers	Bytes	DEC Coding Example
reg16	2	—	1	DEC AX
reg8	3	—	2	DEC AL
memory	15(23) + EA	2	2-4	DEC ARRAY [SI]

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# DIV

# DIVIDE

# DIV

## Operation:

```

(temp) ← (NUMR)
if (temp) / (DIVR) > MAX then the
  following, in sequence
  (QUO), (REM) undefined
  (SP) ← (SP) - 2
  ((SP) + 1:(SP)) ← FLAGS
  (IF) ← 0
  (TF) ← 0
  (SP) ← (SP) - 2
  ((SP) + 1:(SP)) ← (CS)
  (CS) ← (2) i.e., the contents of
    memory locations 2 and 3
  (SP) ← (SP) - 2
  ((SP) + 1:(SP)) ← (IP)
  (IP) ← (0) i.e., the contents of
    locations 0 and 1
else
  (QUO) ← (temp) / (DIVR), where
    / is unsigned division
  (REM) ← (temp) % (DIVR) where
    % is unsigned modulo

```

## Flags Affected:

AF, CF, OF, PF, SF, ZF undefined

## Description:

### *DIV source*

DIV (divide) performs an unsigned division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH. The single-length quotient is returned in AL, and the single-length remainder is returned in AH. If the source operand is a word, it is divided into the double-length dividend in registers AX and DX. The single-length quo-

tient is returned in AX, and the single-length remainder is returned in DX. If the quotient exceeds the capacity of its destination register (FFH for byte source, FFFFFFFH for word source), as when division by zero is attempted, a type 0 interrupt is generated, and the quotient and remainder are undefined. Non-integral quotients are truncated to integers. The content of AF, CF, OF, PF, SF and ZF is undefined following execution of DIV.

**DIV****DIVIDE****DIV****Encoding:**

1111011w	mod110r/m
----------	-----------

if  $w = 0$  then NUMR = AX, DIVR = EA, QUO = AL, REM = AH, MAX = FFH  
 else NUMR = DX:AX, DIVR = EA, QUO = AX, REM = DX, MAX = FFFFH

DIV Operands	Clocks*	Transfers	Bytes	DIV Coding Example
reg8	80-90	—	2	DIV CL
reg16	144-162	—	2	DIV BX
mem8	(86-96) + EA	1	2-4	DIV ALPHA
mem16	(154-172) + EA	1	2-4	DIV TABLE [SI]

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# ESC

# ESCAPE

# ESC

**Operation:**if mod  $\neq$  11 then data bus  $\leftarrow$  (EA)**Flags Affected:**

None

**Description:**

The ESC (Escape) instruction provides a mechanism by which other processors (coprocessors) may receive their instructions from the 8086 or 8088 instruction stream and make use of the 8086 or 8088 addressing modes. The CPU (8086 or 8088) does a no operation (NOP) for the ESC instruction other than to access a memory operand and place it on the bus.

**Encoding:**

11011x	mod x r/m
--------	-----------

ESC Operands	Clocks*	Transfers	Bytes	ESC Coding Example
immediate, memory immediate, register	8(12) + EA 2	1 —	2-4 2	ESC 6,ARRAY [SI] ESC 20,AL

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# HLT

# HALT

# HLT

**Operation:**

None

**Flags Affected:**

None

**Description:**

HLT (Halt) causes the 8086, 8088 to enter the halt state. The processor leaves the halt state upon activation of the RESET line, upon receipt of a non-maskable interrupt request on NMI, or, if interrupts are enabled, upon

receipt of a maskable interrupt request on INTR. HLT does not affect any flags. It may be used as an alternative to an endless software loop in situations where a program must wait for an interrupt.

**Encoding:**

11110100
----------

HLT Operands	Clocks	Transfers	Bytes	HLT Coding Example
(no operands)	2	—	1	HLT

# IDIV                      INTEGER DIVIDE                      IDIV

## Operation:

```

(temp) ← (NUMR)
if (temp) / (DIVR) > 0 and (temp)
 / (DIVR) > MAX
or (temp) / (DIVR) < 0 and (temp)
 / (DIVR) < 0 - MAX - 1 then
(QUO), (REM) undefined
(SP) ← (SP) - 2
((SP) + 1:(SP)) ← FLAGS
(IF) ← 0
(TF) ← 0
(SP) ← (SP) - 2
((SP) + 1:(SP)) ← (CS)
(CS) ← (2)
(SP) ← (SP) - 2
((SP) + 1:(SP)) ← (IP)
(IP) ← (0)
else
(QUO) ← (temp) / (DIVR), where
 / is signed division
(REM) ← (temp) % (DIVR) where
 % is signed modulo

```

## Flags Affected:

AF, CF, OF, PF, SF, ZF undefined

## Description:

### IDIV source

IDIV (Integer Divide) performs a signed division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH; the single-length quotient is returned in AL, and the single-length remainder is returned in AH. For byte integer division, the maximum positive quotient is +127 (7FH) and the minimum negative quotient is -127 (81H). If the source operand is a word, it is divided into the double-length dividend in registers AX and DX; the single-length quotient is returned in

AX, and the single-length remainder is returned in DX. For word integer division, the maximum positive quotient is +32,767 (7FFFH) and the minimum negative quotient is -32,767 (8001H). If the quotient is positive and exceeds the maximum, or is negative and is less than the minimum, the quotient and remainder are undefined, and a type 0 interrupt is generated. In particular, this occurs if division by 0 is attempted. Nonintegral quotients are truncated (toward 0) to integers, and the remainder has the same sign as the dividend. The content of AF, CF, OF, PF, SF and ZF is undefined following IDIV.

**IDIV****INTEGER DIVIDE****IDIV****Encoding:**

1111011w	mod111r/m
----------	-----------

if w = 0 then NUMR = AX, DIVR = EA, QUO = AL, REM = AH, MAX = 7FH  
 else NUMR = DX:AX, DIVR = EA, QUO = AX, REM = DX, MAX = 7FFFH

IDIV Operands	Clocks*	Transfers	Bytes	IDIV Coding Example
reg8	101-112	—	2	IDIV BL
reg16	165-184	—	2	IDIV CX
mem8	(107-118) + EA	1	2-4	IDIV DIVISOR_BYTE [SI]
mem16	(175-194) + EA	1	2-4	IDIV [BX].DIVISOR_WORD

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.



# IMUL INTEGER MULTIPLY IMUL

## Operation:

(DEST)  $\leftarrow$  (LSRC) \* (RSRC) where  
 \* is signed multiply  
 if (ext) = sign-extension of (LOW)  
 then (CF)  $\leftarrow$  0  
 else (CF)  $\leftarrow$  1;  
 (OF)  $\leftarrow$  (CF)

## Flags Affected:

CF, OF  
 AF, PF, SF, ZF undefined

## Description:

### IMUL source

IMUL (Integer Multiply) performs a signed multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. If the upper

half of the result (AH for byte source, DX for word source) is not the sign extension of the lower half of the result, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF and ZF is undefined following execution of IMUL.

## Encoding:

1111011w	mod101r/m
----------	-----------

if w = 0 then LSRC = AL, RSRC = EA, DEST = AH, EXT = AH, LOW = AL  
 else LSRC = AX, RSRC = EA, DEST = DX:AX, EXT = DX, LOW = AX

IMUL Operands	Clocks*	Transfers	Bytes	IMUL Coding Example
reg8	80-98	—	2	IMUL CL
reg16	128-154	—	2	IMUL BX
mem8	(86-104) + EA	1	2-4	IMUL RATE_BYTE
mem16	(138-164) + EA	1	2-4	IMUL RATE_WORD [BP + DI]

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# IN INPUT BYTE OR WORD IN

## Operation:

(DEST) ← (SRC)

## Flags Affected:

None

## Description:

**IN** *accumulator, port*

IN transfers a byte or a word from an input port to the AL register or the AX register, respectively. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through

255, or with a number previously placed in the DX register, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.

## Encoding:

### Fixed Port:

1110010w	port
----------	------

if  $w = 0$  then SRC = port, DEST = AL  
 else SRC = port + 1:port, DEST = AX

### Variable Port:

1110110w
----------

if  $w = 0$  then SRC = (DX), DEST = AL  
 else SRC = (DX) + 1:(DX), DEST = AX

IN Operands	Clocks*	Transfers	Bytes	IN Coding Example
accumulator, immed8	10(14)	1	2	IN AL,0FFEAH
accumulator, DX	8(12)	1	1	IN AX, DX

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# INC

# INCREMENT

# INC

**Operation:** $(\text{DEST}) \leftarrow (\text{DEST}) + 1$ **Flags Affected:**

AF, OF, PF, SF, ZF

**Description:***INC destination*

INC (Increment) adds one to the destination operand. The operand may be a byte or a word and is treated as an unsigned binary number (see AAA and DAA). INC updates AF, OF, PF, SF and ZF; it does not affect CF.

**Encoding:****Memory or Register Operand:**

1111111w	mod000r/m
----------	-----------

DEST = EA

**Register Operand:**

01000reg
----------

DEST = REG

INC Operands	Clocks*	Transfers	Bytes	INC Coding Example
reg16	2	—	1	INC CX
reg8	3	—	2	INC BL
memory	15(23) + EA	2	2-4	INC ALPHA [DI + BX]

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

**INT****INTERRUPT****INT****Operation:**

```

(SP) ← (SP) - 2
((SP) + 1:(SP)) ← FLAGS
(IF) ← 0
(TF) ← 0
(SP) ← (SP) - 2
((SP) + 1:(SP)) ← (CS)
(CS) ← (TYPE * 4 + 2)
(SP) ← (SP) - 2
((SP) + 1:(SP)) ← (IP)
(IP) ← (TYPE * 4)

```

**Flags Affected:**

IF, TF

**Description:****INT** *interrupt-type*

INT (Interrupt) activates the interrupt procedure specified by the interrupt-type operand. INT decrements the stack pointer by two, pushes the flags onto the stack, and clears the trap (TF) and interrupt-enable (IF) flags to disable single-step and maskable interrupts. The flags are stored in the format used by the PUSHF instruction. SP is decremented again by two, and the CS register is pushed onto the stack. The address of the interrupt pointer is calculated by multiplying interrupt-type by four; the second word of the interrupt pointer replaces CS. SP again is decremented by two, and IP is pushed onto the stack and is replaced

by the first word of the interrupt pointer. If interrupt-type = 3, the assembler generates a short (1 byte) form of the instruction, known as the breakpoint interrupt.

Software interrupts can be used as “supervisor calls,” i.e., requests for service from an operating system. A different interrupt-type can be used for each type of service that the operating system could supply for an application program. Software interrupts also may be used to check out interrupt service procedures written for hardware-initiated interrupts.

# INT

# INTERRUPT

# INT

### Encoding:

1100110v	type if v=1
----------	-------------

if v = 0 then TYPE = 3  
 else TYPE = type

INT Operands	Clocks*	Transfers	Bytes	INT Coding Example
immed8 (type = 3)	52(72)	5	1	INT 3
immed8 (type ≠ 3)	51(71)	5	2	INT 67

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# INTO                      INTERRUPT ON OVERFLOW                      INTO

## Operation:

```

if (OF) = 1 then
  (SP) ← (SP) - 2
  ((SP) + 1:(SP)) ← FLAGS
  (IF) ← 0
  (TF) ← 0
  (SP) ← (SP) - 2
  ((SP) + 1:(SP)) ← (CS)
  (CS) ← (12H)
  (SP) ← (SP) - 2
  ((SP) + 1:(SP)) ← (IP)
  (IP) ← (10H)

```

## Flags Affected:

None

## Description:

INTO (Interrupt on Overflow) generates a software interrupt if the overflow flag (OF) is set; otherwise control proceeds to the following instruction without activating an interrupt procedure. INTO addresses the target interrupt procedure (its type is 4) through the inter-

rupt pointer at location 10H; it clears the TF and IF flags and otherwise operates like INT. INTO may be written following an arithmetic or logical operation to activate an interrupt procedure if overflow occurs.

## Encoding:

11001110
----------

INTO Operands	Clocks*	Transfers	Bytes	INTO Coding Example
(no operands)	53(73) or 4	5	1	INTO

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# IRET      INTERRUPT RETURN      IRET

## Operation:

```
(IP) ← ((SP) + 1:(SP))
(SP) ← (SP) + 2
(CS) ← ((SP) + 1:(SP))
(SP) ← (SP) + 2
FLAGS ← ((SP) + 1:(SP))
(SP) ← (SP) + 2
```

## Flags Affected:

All

## Description:

IRET (Interrupt Return) transfers control back to the point of interruption by popping IP, CS and the flags from the stack. IRET thus affects all flags by restoring them to previously saved values. IRET is used to exit any interrupt procedure, whether activated by hardware or software.

## Encoding:

11001111
----------

IRET Operands	Clocks*	Transfers	Bytes	IRET Coding Example
(no operands)	32(44)	3	1	IRET

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# JA                      JUMP ON ABOVE                      JA

## JNBE    JUMP ON NOT BELOW    JNBE

### OR EQUAL

**Operation:**

if (CF) & (ZF) = 0 then  
 (IP) ← (IP) + disp (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

Jump on Above (JA)/Jump on Not Below or Equal (JNBE) transfers control to the target operand (IP + displacement). If the conditions (CF and ZF = 0) are above/not below or equal to the tested value.

**Encoding:**

01110111	disp
----------	------

JA/JNBE Operands	Clocks	Transfers	Bytes	JA Coding Example
short-label	16 or 4	—	2	JA ABOVE
				JNBE Coding Example
				JNBE ABOVE



<b>JAE</b>	<b>JUMP ON ABOVE OR EQUAL</b>	<b>JAE</b>
<b>JNB</b>	<b>JUMP ON NOT BELOW</b>	<b>JNB</b>

**Operation:**

if (CF) = 0 then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended  
to 16-bits)

**Flags Affected:**

None

**Description:**

JAE (Jump on Above or Equal)/JNB (Jump on Not Below) transfers control to the target operand (IP + displacement) if the condition (CF = 0) is above or equal/not below the tested value.

**Encoding:**

01110011	disp
----------	------

JAE/JNB Operands	Clocks	Transfers	Bytes	JAE Coding Example
short-label	16 or 4	—	2	JAE ABOVE_EQUAL

# JB                      JUMP ON BELOW                      JB

## JNAE                      JUMP ON NOT ABOVE OR EQUAL                      JNAE

**Operation:**

if (CF) = 1 then  
 (IP) ← (IP) + disp (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

JB (Jump on Below)/JNAE (Jump on Not Above or Equal) transfers control to the target operand (IP + displacement) if the condition (CF = 1) is below/not above or equal to the tested value.

**Encoding:**

01110010	disp
----------	------

JB/JNAE Operands	Clocks	Transfers	Bytes	JB Coding Example
short-label	16 or 4	—	2	JB BELOW

**JBE****JUMP ON BELOW  
OR EQUAL****JBE****JNA****JUMP ON  
NOT ABOVE****JNA****Operation:**

IF (CF) or (ZF) = 1 then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

JBE (Jump on Below or Equal)/JNA (Jump on Not Above) transfers control to the target operand (IP + displacement) if the conditions (CF or ZF = 1) are below or equal/or not above the tested conditions.

**Encoding:**

01110110	disp
----------	------

JBE/JNA Operands	Clocks	Transfers	Bytes	JNA Coding Example
short-label	16 or 4	—	2	JNA NOT_ABOVE

**JC****JUMP ON CARRY****JC****Operation:**

if (CF) = 1 THEN  
 (IP) ← (IP) + disp (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

JC (Jump on Carry) transfers control to the target operand (IP + displacement) on the condition CF = 1.

**Encoding:**

01110010	disp
----------	------

JC Operands	Clocks	Transfers	Bytes	JC Coding Example
short-label	16 or 4	—	2	JC CARRY_SET

# JCXZ

# JUMP IF CX REGISTER ZERO

# JCXZ

**Operation:**

if (CX) = 0 then  
 (IP) ← (IP) + disp (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:****JCXZ short-label**

JCXZ (Jump if CX Zero) transfers control to the target operand if CX is 0. This instruction is useful at the beginning of a loop to bypass the loop if CX has a zero value, i.e., to execute the loop zero times.

**Encoding:**

11100011	disp
----------	------

JCXZ Operands	Clocks	Transfers	Bytes	JCXZ Coding Example
short-label	18 or 6	—	2	JCXZ COUNT_DONE

<b>JE</b>	<b>JUMP ON EQUAL</b>	<b>JE</b>
<b>JZ</b>	<b>JUMP ON ZERO</b>	<b>JZ</b>

**Operation:**

if (ZF) = 1 then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended  
to 16-bits)

**Flags Affected:**

None

**Description:**

JE (Jump on Equal)/JZ (Jump on Zero) transfers control to the target operand (IP + displacement) if the condition (ZF = 1) is equal/zero on the tested value.

**Encoding:**

01110100	disp
----------	------

JE/JZ Operands	Clocks	Transfers	Bytes	JZ Coding Example
short-label	16 or 4	—	2	JZ ZERO

# JG                      JUMP ON GREATER                      JG

## JNLE                      JUMP ON NOT LESS OR EQUAL                      JNLE

**Operation:**

if  $((SF) = (OF)) \ \& \ ((ZF) = 0)$  then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended  
to 16-bits)

**Flags Affected:**

None

**Description:**

JG (Jump on Greater Than)/JNLE (Jump on Not Less Than or Equal) transfers control to the target operand (IP + displacement) if the conditions  $((SF \text{ XOR } OF) \text{ or } ZF = 0)$  are greater than/not less than or equal to the tested value.

**Encoding:**

01111111	disp
----------	------

JG/JNLE Operands	Clocks	Transfers	Bytes	JG Coding Example
short-label	16 or 4	—	2	JG GREATER

<b>JGE</b>	<b>JUMP ON GREATER OR EQUAL</b>	<b>JGE</b>
<b>JNL</b>	<b>JUMP ON NOT LESS</b>	<b>JNL</b>

**Operation:**

if (SF) = (OF) 0 then  
 (IP) ← (IP) + disp (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

JGE (Jump on Greater Than or Equal)/JNL (Jump on Not Less Than) transfers control to the target operand (IP + displacement) if the condition (SF XOR OF = 0) is greater than or equal/not less than the tested value.

**Encoding:**

01111101	disp
----------	------

JGE/JNL Operands	Clocks	Transfers	Bytes	JGE Coding Example
short-label	16 or 4	—	2	JGE GREATER_EQUAL



# JL                      JUMP ON LESS                      JL

# JNGE                      JUMP ON NOT                      JNGE

## GREATER OR EQUAL

**Operation:**

if (SF) ≠ (OF) then  
 (IP) ← (IP) + disp (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

JL (Jump on Less Than)/JNGE (Jump on Not Greater Than or Equal), transfers control to the target operand if the condition (SF XOR OF = 1) is less than/not greater than or equal to the tested value.

**Encoding:**

01111100	disp
----------	------

JL/JNGE Operands	Clocks	Transfers	Bytes	JL Coding Example
short-label	16 or 4	—	2	JL LESS

# JLE                      JUMP ON LESS OR EQUAL                      JLE

# JNG                      JUMP ON NOT GREATER                      JNG

**Operation:**

if  $((SF) \neq (OF))$  or  $((ZF) = 1)$  then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended  
to 16-bits)

**Flags Affected:**

None

**Description:**

JLE (Jump on Less Than or Equal to)/JNG (Jump on Not Greater Than) transfers control to the target operand (IP + displacement) if the conditions tested  $((SF \text{ XOR } OF)$  or  $ZF = 1$ ) are less than or equal to/not greater than the tested value.

**Encoding:**

01111110	disp
----------	------

JLE/JNG Operands	Clocks	Transfers	Bytes	JNG Coding Example
short-label	16 or 4	—	2	JNG NOT_GREATER

# JMP JUMP UNCONDITIONALLY JMP

## Operation:

if Inter-Segment then (CS) ← SEG  
(IP) ← DEST

## Flags Affected:

None

## Description:

### *JMP target*

JMP unconditionally transfers control to the target location. Unlike a CALL instruction, JMP does not save any information on the stack; no return to the instruction following the JMP is expected. Like CALL, the address of the target operand may be obtained from the instruction itself (direct JMP), or from memory or a register referenced by the instruction (indirect JMP).

An intrasegment direct JMP changes the instruction pointer by adding the relative displacement of the target from the JMP instruction. If the assembler can determine that the target is within 127 bytes of the JMP, it automatically generates a two-byte instruction form called a SHORT JMP; otherwise, it generates a NEAR JMP that can address a target within  $\pm 32k$ . Intrasegment direct JMPS are self-relative and appropriate in position-

independent (dynamically relocatable) routines in which the JMP and its target are moved together in the same segment.

An intrasegment indirect JMP may be made either through memory or a 16-bit general register. In the first case, the word content referenced by the instruction replaces the instruction pointer. In the second case, the new IP value is taken from the register named in the instruction.

An intersegment direct JMP replaces IP and CS with values contained in the instruction.

An intersegment indirect JMP may be made only through memory. The first word of the doubleword pointer referenced by the instruction replaces IP and the second word replaces CS.

# JMP JUMP UNCONDITIONALLY JMP

## Encoding:

### Intra-Segment Direct:

11101001	disp-low	disp-high
----------	----------	-----------

DEST = (IP) + disp

### Intra-Segment Direct Short:

11101011	disp
----------	------

DEST = (IP) + disp sign extended to 16-bits

### Intra-Segment Indirect:

11111111	mod100 r/m
----------	------------

DEST = (EA)

### Inter-Segment Direct:

11101010	offset-low	offset-high
----------	------------	-------------

seg-low	seg-high
---------	----------

DEST = offset, SEG = seg

### Inter-Segment Indirect:

11111111	mod101 r/m
----------	------------

DEST = (EA), SEG = (EA + 2)

JMP Operands	Clocks	Transfers	Bytes	JMP Coding Example
short-label	15	—	2	JMP SHORT
near-label	15	—	3	JMP WITHIN_SEGMENT
far-label	15	—	5	JMP FAR_LABEL
memptr16	18 + EA	—	2-4	JMP [BX].TARGET
regptr16	11	—	2	JMP CX
memptr32	24 + EA	—	2-4	JMP OTHER.SEG [SI]

# JNC JUMP ON NOT CARRY JNC

## Operation:

if (CF) = 0 THEN  
 (IP) ← (IP) + disp (sign-extended  
 to 16-bits)

## Flags Affected:

None

## Description:

JNC (Jump on Not Carry) transfers control to the target operand (IP + displacement) on the condition CF = 0.

## Encoding:

01110011	disp
----------	------

JNC Operands	Clocks	Transfers	Bytes	JNC Coding Example
short-label	16 or 4	—	2	JNC NO_CARRY

# JNE      JUMP ON NOT EQUAL      JNE

## JNZ      JUMP ON NOT ZERO      JNZ

**Operation:**

if (ZF) = 0 then  
 (IP) ← (IP) + disp (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

JNE (Jump on Not Equal to)/ JNZ (Jump on Not Zero) transfers control to the target operand (IP + displacement) if the condition tested (ZF = 0) is true.

**Encoding:**

01110101	disp
----------	------

JNE/JNZ Operands	Clocks	Transfers	Bytes	JNE Coding Example
short-label	16 or 4	—	2	JNE NOT_EQUAL

# JNO

# JUMP ON NOT OVERFLOW

# JNO

**Operation:**

if (OF) = 0 then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

JNO (Jump on Not Overflow) transfers control to the target operand (IP + displacement) if the condition tested (OF = 0) is true.

**Encoding:**

01110001	disp
----------	------

JNO Operands	Clocks	Transfers	Bytes	JNO Coding Example
short-label	16 or 4	—	2	JNO NO__OVERFLOW

# JNS      JUMP ON NOT SIGN      JNS

## Operation:

if (SF) = 0 then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended  
 to 16-bits)

## Flags Affected:

None

## Description:

JNS (Jump on Not Sign) transfers control to the target operand (IP + displacement) when the tested condition (SF = 0) is true.

## Encoding:

01111001	disp
----------	------

JNS Operands	Clocks	Transfers	Bytes	JNS Coding Example
short-label	16 or 4	—	2	JNS POSITIVE



# JNP      JUMP ON NOT PARITY      JNP

## JPO      JUMP ON PARITY ODD      JPO

**Operation:**

if (PF) = 0 then  
 (IP) ← (IP) + disp (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

JNP (Jump on Not Parity)/JPO (Jump on Parity Odd) transfers control to the target operand if the condition tested (PF = 0) is true.

**Encoding:**

01111011	disp
----------	------

JNP/JPO Operands	Clocks	Transfers	Bytes	JPO Coding Example
short-label	16 or 4	—	2	JPO ODD__PARITY

# JO                      JUMP ON OVERFLOW                      JO

## Operation:

if (OF) = 1 then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended  
 to 16-bits)

## Flags Affected:

None

## Description:

JO (Jump on Overflow) transfers control to the target operand (IP + displacement) if the tested condition (OF = 1) is true.

## Encoding:

01110000	disp
----------	------

JO Operands	Clocks	Transfers	Bytes	JO Coding Example
short-label	16 or 4	—	2	JO SIGNED_OVERFLOW

# JP                    JUMP ON PARITY                    JP

## JPE                  JUMP ON PARITY EQUAL                  JPE

**Operation:**

if (PF) = 1 then  
 (IP) ← (IP) + disp (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

JP (Jump on Parity)/JPE (Jump on Parity Equal) transfers control to the target operand (IP + displacement) if the condition tested (PF = 1) is true.

**Encoding:**

01111010	disp
----------	------

JP/JPE Operands	Clocks	Transfers	Bytes	JPE Coding Example
short-label	16 or 4	—	2	JPE EVEN__PARITY

**JS****JUMP ON SIGN****JS****Operation:**

if (SF) = 1 then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

JS (Jump on Sign) transfers control to the target operand (IP + displacement) if the tested condition (SF = 1) is true.

**Encoding:**

01111000	disp
----------	------

JS Operands	Clocks	Transfers	Bytes	JS Coding Example
short-label	16 or 4	—	2	JS NEGATIVE

# LAHF    LOAD REGISTER AH    LAHF

## FROM FLAGS

### Operation:

(AH) ← (SF):(ZF):X:(AF):X:(PF):X:(CF)

### Flags Affected:

None

### Description:

LAHF (load register AH from flags) copies SF, ZF, AF, PF and CF (the 8080/8085 flags) into bits 7, 6, 4, 2 and 0, respectively, of register AH. The content of bits 5, 3 and 1 is undefined; the flags themselves are not affected. LAHF is provided primarily for converting 8080/8085 assembly language programs to run on an 8086 or 8088.

### Encoding:

10011111
----------

LAHF Operands	Clocks	Transfers	Bytes	LAHF Coding Example
(no operands)	4	—	1	LAHF

# LDS LOAD POINTER USING DS LDS

**Operation:**

(REG) ← (EA)  
 (DS) ← (EA + 2)

**Flags Affected:**

None

**Description:**

*LDS destination, source*

LDS (load pointer using DS) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register DS. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the

pointer is transferred to register DS. Specifying SI as the destination operand is a convenient way to prepare to process a source string that is not in the current data segment (string instructions assume that the source string is located in the current data segment and that SI contains the offset of the string).

**Encoding:**

11000101	mod reg r/m
----------	-------------

if mod = 11 then undefined operation

LDS Operands	Clocks	Transfers	Bytes	LDS Coding Example
reg16, mem32	24 + EA	2	2-4	LDS SI, DATA.SEG [DI]

**LEA****LOAD EFFECTIVE  
ADDRESS****LEA****Operation:**

(REG) ← EA

**Flags Affected:**

None

**Description:***LEA destination, source*

LEA (load effective address) transfers the offset of the source operand (rather than its value) to the destination operand. The source operand must be a memory operand, and the destination operand must be a 16-bit general

register. LEA does not affect any flags. The XLAT and string instructions assume that certain registers point to operands; LEA can be used to load these registers (e.g., loading BX with the address of the translate table used by the XLAT instruction).

**Encoding:**

1 0 0 0 1 1 0 1	mod reg r/m
-----------------	-------------

if mod = 11 then undefined operation

LEA Operands	Clocks	Transfers	Bytes	LEA Coding Example
reg16, mem16	2 + EA	—	2-4	LEA BX,[BP + DI]

# LES LOAD POINTER USING ES LES

## Operation:

(REG) ← (EA)  
(ES) ← (EA + 2)

## Flags Affected:

None

## Description:

### *LES destination,source*

LES (load pointer using ES) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register ES. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the

pointer is transferred to register ES. Specifying DI as the destination operand is a convenient way to prepare to process a destination string that is not in the current extra segment. (The destination string must be located in the extra segment, and DI must contain the offset of the string.)

## Encoding:

11000100	mod reg r/m
----------	-------------

if mod = 11 then undefined operation

LES Operands	Clocks	Transfers	Bytes	LES Coding Example
reg16, mem32	24 + EA	2	2-4	LES DI,[BX].TEXT_BUFF



# LOCK LOCK THE BUS LOCK

## Operation:

None

## Flags Affected:

None

## Description:

LOCK is a one-byte prefix that causes the 8088 (configured in maximum mode) to assert its bus LOCK signal while the following instruction executes. LOCK does not affect any flags.

The instruction most useful in this context is an exchange register with memory. A simple software lock may be implemented with the following code sequence:

```

Check:  MOV     AL,1           ;set AL to 1 (implies locked)
LOCK:   XCHG  Sema,AL        ;test and set lock
        TEST  AL,AL         ;set flags based on AL
        JNZ  Check          ;retry if lock already set

        MOV  Sema,0         ;clear the lock when done
  
```

The LOCK prefix may be combined with the segment override and/or REP prefixes.

## Encoding:

11110000
----------

LOCK Operands	Clocks	Transfers	Bytes	LOCK Coding Example
(no operands)	2	—	1	LOCK XCHG FLAG,AL

**LODS****LOAD STRING  
(BYTE OR WORD)****LODS****Operation:**

$(DEST) \leftarrow (SRC)$   
 if  $(DF) = 0$  then  $(SI) \leftarrow (SI) + DELTA$   
 else  $(SI) \leftarrow (SI) - DELTA$

**Flags Affected:**

None

**Description:****LODS** *source-string*

LODS (Load String) transfers the byte or word string element addressed by SI to register AL or AX, and updates SI to point to the next element in the string. This instruction is not ordinarily repeated since the accumulator would be

overwritten by each repetition, and only the last element would be retained. However, LODS is very useful in software loops as part of a more complex string function built up from string primitives and other instructions.

**Encoding:**

1 0 1 0 1 1 0 w

if  $w = 0$  then  $SRC = (SI)$ ,  $DEST = AL$ ,  $DELTA = 1$   
 else  $SRC = (SI) + 1$ ;  $(SI)$ ,  $DEST = AX$ ,  $DELTA = 2$

LODS Operands	Clocks*	Transfers	Bytes	LODS Coding Example
source-string	12(16)	1	1	LODS CUSTOMER_NAME
(repeat) source-string	9 + 13(17)/rep	1/rep	1	REP LODS NAME

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.



# LODSW LOAD STRING WORD LODSW

**Operation:**

(DEST) ← (SRC)  
 if (DF) = 0 then (SI) ← (SI) + DELTA  
 else (SI) ← (SI) - DELTA

**Flags Affected:**

None

**Description:**

LODSW (Load String Word) transfers the word string element addressed by SI to register AX, and updates SI to point to the next element in the string. This instruction is not ordinarily repeated since the accumulator would be overwritten by each repetition, and

only the last element would be retained. However, LODSW is very useful in software loops as part of a more complex string function built up from string primitives and other instructions.

**Encoding:**

1010110w

SRC = (SI) + 1:(SI), DEST = AX, DELTA = 2

LODSW Operands	Clocks	Transfers	Bytes	LODSW Coding Examples
(NO OPERANDS)	16	1	1	LODSW
(NO OPERANDS)	9 + 17/rep	1/rep	1	REP LODSW

# LOOP

# LOOP

# LOOP

**Operation:**

$(CX) \leftarrow (CX) - 1$   
 if  $(CX) \neq 0$  then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:**

**LOOP** *short-label*

LOOP decrements CX by 1 and transfers control to the target operand if CX is not 0; otherwise the instruction following LOOP is executed.

**Encoding:**

11100010	disp
----------	------

LOOP Operands	Clocks	Transfers	Bytes	LOOP Coding Example
short-label	17/5	—	2	LOOP AGAIN

<b>LOOPE</b>	<b>LOOP WHILE EQUAL</b>	<b>LOOPE</b>
<b>LOOPZ</b>	<b>LOOP WHILE ZERO</b>	<b>LOOPZ</b>

**Operation:**

$(CX) \leftarrow (CX) - 1$   
 if  $(ZF) = 1$  and  $(CX) \neq 0$  then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended  
 to 16-bits)

**Flags Affected:**

None

**Description:***LOOPE/LOOPZ short-label*

LOOPE and LOOPZ (Loop While Equal and Loop While Zero) are different mnemonics for the same instruction (similar to the REPE and REPZ repeat prefixes). CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is set; otherwise the instruction following LOOPE/LOOPZ is executed.

**Encoding:**

11100001	disp
----------	------

LOOPE/LOOPZ Operands	Clocks	Transfers	Bytes	LOOPE Coding Example
short-label	18 or 6	—	2	LOOPE AGAIN

# LOOPNZ    LOOP WHILE NOT ZERO    LOOPNZ

# LOOPNE    LOOP WHILE NOT EQUAL    LOOPNE

## Operation:

$(CX) \leftarrow (CX) - 1$   
 if  $(ZF) = 0$  and  $(CX) \neq 0$  then  
 $(IP) \leftarrow (IP) + \text{disp}$  (sign-extended to 16-bits)

## Flags Affected:

None

## Description:

**LOOPNE/LOOPNZ** *short-label*

LOOPNE and LOOPNZ (Loop While Not Equal and Loop While Not Zero) are also synonyms for the same instruction. CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is clear; otherwise the next sequential instruction is executed.

## Encoding:

11100000	disp
----------	------

LOOPNE/LOOPNZ Operands	Clocks	Transfers	Bytes	LOOPNE Coding Example
short-label	19 or 5	—	2	LOOPNE AGAIN

# MOV MOVE (BYTE OR WORD) MOV

## Operation:

(DEST) ← (SRC)

## Flags Affected:

None

## Description:

*MOV destination, source*

MOVE transfers a byte or a word from the source operand to the destination operand.

## Encoding:

### Memory or Register Operand to/from Register Operand:

100010dw	mod reg r/m
----------	-------------

if d = 1 then SRC = EA, DEST = REG  
 else SRC = REG, DEST = EA

### Immediate Operand to Memory or Register Operand:

1100011w	mod 000 r/m	data	data if w=1
----------	-------------	------	-------------

SRC = data, DEST = EA

### Immediate Operand to Register:

1011w reg	data	data if w=1
-----------	------	-------------

SRC = data, DEST = REG



# MOV MOVE (BYTE OR WORD) MOV

## Encoding:

### Memory Operand to Accumulator:

1010000w	addr-low	addr-high
----------	----------	-----------

if w = 0 then SRC = addr, DEST = AL  
 else SRC = addr + 1:addr, DEST = AX

### Accumulator to Memory Operand:

1010001w	addr-low	addr-high
----------	----------	-----------

if w = 0 then SRC = AL, DEST = addr  
 else SRC = AX, DEST = addr + 1:addr

### Memory or Register Operand to Segment Register:

10001110	mod 0 reg r/m
----------	---------------

if reg ≠ 01 then SRC = EA, DEST = REG  
 else undefined operation

### Segment Register to Memory or Register Operand:

10001100	mod 0 reg r/m
----------	---------------

SRC = REG, DEST = EA

MOV Operands	Clocks*	Transfers	Bytes	MOV Coding Example
memory, accumulator	10(14)	1	3	MOV ARRAY [SI], AL
accumulator, memory	10(14)	1	3	MOV AX, TEMP_RESULT
register, register	2	—	2	MOV AX, CX
register, memory	8(12) + EA	1	2-4	MOV BP, STACK_TOP
memory, register	9(13) + EA	1	2-4	MOV COUNT [DI], CX
register, immediate	4	—	2-3	MOV CL, 2
memory, immediate	10(14) + EA	1	3-6	MOV MASK [BX + SI], 2CH
seg-reg, reg16	2	—	2	MOV ES, CX
seg-reg, mem16	8(12) + EA	1	2-4	MOV DS, SEGMENT_BASE
reg16, seg-reg	2	—	2	MOV BP, SS
memory, seg-reg	9(13) + EA	1	2-4	MOV [BX], SEG_SAVE, CS

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# MOVS      MOVE STRING      MOVS

## Operation:

```
(DEST) ← (SRC)
if (DF) = 0 then
  (SI) ← (SI) + DELTA
  (DI) ← (DI) + DELTA
else
  (SI) ← (SI) - DELTA
  (DI) ← (DI) - DELTA
```

## Flags Affected:

None

## Description:

**MOVS** *destination-string, source-string*

MOVS (Move String) transfers a byte or a word from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element. When used in conjunction with REP, MOVS performs a memory-to-memory block transfer.

## Encoding:

```
1 0 1 0 0 1 0 w
```

if  $w = 0$  then SRC = (SI), DEST = AL, DELTA = 1  
 else SRC = (SI) + 1:(SI), DEST = AX, DELTA = 2

MOVS Operands	Clocks*	Transfers	Bytes	MOVS Coding Example
dest-string, source-string	18(26)	2	1	MOVS LINE_ _EDIT_ _DATA
(repeat) dest-string, source-string	9 + 17(25) / rep	2 / rep	1	REP MOVS SCREEN, BUFFER

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

**MOVSB****MOVE STRING  
BYTE****MOVSB****Operation:**

```

(DEST) ← (SRC)
if (DF) = 0 then
  (SI) ← (SI) + DELTA
  (DI) ← (DI) + DELTA
else
  (SI) ← (SI) - DELTA
  (DI) ← (DI) - DELTA

```

**Flags Affected:**

None

**Description:**

MOVSB (Move String Byte) transfers a byte from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element. When used in conjunction with REP, MOVSB performs a memory-to-memory block transfer.

**Encoding:**

10100100
----------

SRC = (SI), DEST = AL, DELTA = 1

MOVSB Operands	Clocks	Transfers	Bytes	MOVSB Coding Examples
(NO OPERANDS)	18	2	1	MOVSB
(NO OPERANDS)	9 + 17/rep	2/rep	1	REP MOVSB

# MOVSW MOVE STRING WORD MOVSW

## Operation:

```
(DEST) ← (SRC)
if (DF) = 0 then
  (SI) ← (SI) + DELTA
  (DI) ← (DI) + DELTA
else
  (SI) ← (SI) - DELTA
  (DI) ← (DI) - DELTA
```

## Flags Affected:

None

## Description:

MOVSW (Move String Word) transfers a word from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element. When used in conjunction with REP, MOVSW performs a memory-to-memory block transfer.

## Encoding:

10100101

SRC = (SI) + 1:(SI), DEST = AX, DELTA = 2

MOVSW Operands	Clocks	Transfers	Bytes	MOVSW Coding Examples
(NO OPERANDS)	26	2	1	MOVSW
(NO OPERANDS)	9 + 25/rep	2/rep	1	REP MOVSW

# MUL

# MULTIPLY

# MUL

## Operation:

$(DES) \leftarrow (LSRC) * (RSRC)$ , where \*  
 is unsigned multiply  
 if  $(EXT) = 0$  then  $(CF) \leftarrow 0$   
 else  $(CF) \leftarrow 1$ ;  
 $(OF) \leftarrow (CF)$

## Flags Affected:

CF, OF.  
 AF, PF, SF, ZF undefined

## Description:

### MUL source

MUL (Multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source operand is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. The oper-

ands are treated as unsigned binary numbers (see AAM). If the upper half of the result (AH for byte source, DX for word source) is non-zero, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF and ZF is undefined following execution of MUL.

## Encoding:

1111011w	mod 100 r/m
----------	-------------

if  $w = 0$  then  $LSRC = AL$ ,  $RSRC = EA$ ,  $DEST = AX$ ,  $EXT = AH$   
 else  $LSRC = AX$ ,  $RSRC = EA$ ,  $DEST = DX:AX$ ,  $EXT = DX$

MUL Operands	Clocks*	Transfers	Bytes	MUL Coding Example
reg8	70-77	—	2	MUL BL
reg16	118-113	—	2	MUL CX
mem8	(76-83) + EA	1	2-4	MUL MONTH [SI]
mem16	(128-143) + EA	1	2-4	MUL BAUD_RATE

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# NEG

# NEGATE

# NEG

## Operation:

(EA) ← SRC - (EA)  
 (EA) ← (EA) + 1 (affecting flags)

## Flags Affected:

AF, CF, OF, PF, SF, ZF

## Description:

### NEG destination

NEG (Negate) subtracts the destination operand, which may be a byte or a word, from 0 and returns the result to the destination. This forms the two's complement of the number, effectively reversing the sign of an integer. If the operand is zero, its sign is not changed.

Attempting to negate a byte containing -128 or a word containing -32,768 causes no change to the operand and sets OF. NEG updates AF, CF, OF, PF, SF and ZF. CF is always set except when the operand is zero, in which case it is cleared.

## Encoding:

1111011w	mod011r/m
----------	-----------

if w = 0 then SRC = FFH  
 else SRC = FFFFH

NEG Operands	Clocks*	Transfers	Bytes	NEG Coding Example
register memory	3 16(24) + EA	— 2	2 2-4	NEG AL NEG MULTIPLIER

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# NOP

# NO OPERATION

# NOP

**Operation:**

None

**Flags Affected:**

None

**Description:****NOP**

NOP (No Operation) causes the CPU to do nothing. NOP does not affect any flags.

**Encoding:**

10010000
----------

NOP Operands	Clocks	Transfers	Bytes	NOP Coding Example
(no operands)	3	—	1	NOP

# NOT LOGICAL NOT NOT

## Operation:

$(EA) \leftarrow SRC - (EA)$

## Flags Affected:

None

## Description:

**NOT** *destination*

NOT inverts the bits (forms the one's complement) of the byte or word operand.

## Encoding:

1111011w	mod010r/m
----------	-----------

if  $w = 0$  then SRC = FFH  
 else SRC = FFFFH

NOT Operands	Clocks*	Transfers	Bytes	NOT Coding Example
register memory	<sup>3</sup> 16(24) + EA	— 2	— —	NOT AX NOT CHARACTER

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.



# OR LOGICAL OR OR

**Operation:**

(DEST) ← (LSRC) OR (RSRC)  
(CF) ← 0  
(OF) ← 0

**Flags Affected:**

CF, OF, PF, SF, ZF.  
AF undefined

**Description:**

**OR** *destination, source*

OR performs the logical “inclusive or” of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if either or both corresponding bits in the original operands are set; otherwise the result bit is cleared.

# OR LOGICAL OR OR

## Encoding:

### Memory or Register Operand with Register Operand:

000010d w	mod reg r/m
-----------	-------------

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG  
 else LSRC = EA, RSRC = REG, DEST = EA

### Immediate Operand to Memory or Register Operand:

1000000 w	mod 001 r/m	data	data if w=1
-----------	-------------	------	-------------

LSRC = EA, RSRC = data, DEST = EA

### Immediate Operand to Accumulator:

0000110 w	data	data if w=1
-----------	------	-------------

if w = 0 then LSRC = AL, RSRC = data, DEST = AL  
 else LSRC = AX, RSRC = data, DEST = AX

OR Operands	Clocks*	Transfers	Bytes	OR Coding Example
register, register	3	—	2	OR AL, BL
register, memory	9(13) + EA	1	2-4	OR DX, PORT_ID [DI]
memory, register	16(24) + EA	2	2-4	OR FLAG_BYTE, CL
accumulator, immediate	4	—	2-3	OR AL, 01101100B
register, immediate	4	—	3-4	OR CX, 01H
memory, immediate	17(25) + EA	2	3-6	OR [BX].CMD_WORD, 0CFH

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# OUT                      OUTPUT                      OUT

## Operation:

(DEST) ← (SRC)

## Flags Affected:

None

## Description:

*OUT port, accumulator*

OUT transfers a byte or a word from the AL register or the AX register, respectively, to an output port. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through

255, or with a number previously placed in register DX, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.

## Encoding:

### Fixed Port:

1110011w	port
----------	------

if w = 0 then SRC = AL, DEST = port  
 else SRC = AX, DEST = port + 1:port

### Variable Port:

1110111w
----------

if w = 0 then SRC = AL, DEST = (DX)  
 else SRC = AX, DEST = (DX) + 1:(DX)

OUT Operands	Clocks*	Transfers	Bytes	OUT Coding Example
immed8, accumulator	10(14)	1	2	OUT 44, AX
DX, accumulator	8(12)	1	1	OUT DX, AL

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# POP

# POP

# POP

**Operation:**
$$\begin{aligned}(\text{DEST}) &\leftarrow ((\text{SP}) + 1 : (\text{SP})) \\ (\text{SP}) &\leftarrow (\text{SP}) + 2\end{aligned}$$
**Flags Affected:**

None

**Description:***POP destination*

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand, and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

# POP

# POP

# POP

## Encoding:

### Memory or Register Operand:

10001111	mod000r/m
----------	-----------

DEST = EA

### Register Operand:

01011 reg
-----------

DEST = REG

### Segment Register:

000 reg111
------------

if reg ≠ 01 then DEST = REG  
 else undefined operation

POP Operands	Clocks*	Transfers	Bytes	POP Coding Example
register	12	1	1	POP DX
seg-reg (CS illegal)	12	1	1	POP DS
memory	25 + EA	2	2-4	POP PARAMETER

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# POPF

# POP FLAGS

# POPF

**Operation:**

Flags  $\leftarrow ((SP) + 1):(SP)$   
 (SP)  $\leftarrow (SP) + 2$

**Flags Affected:**

All

**Description:****POPF**

POPF transfers specific bits from the word at the current top of stack (pointed to by register SP) into the 8086/8088 flags, replacing whatever values the flags previously contained (see figure 2-32). SP is then incremented by two to point to the new top of stack. PUSHF

and POPF allow a procedure to save and restore a calling program's flags. They also allow a program to change the setting of TF (there is no instruction for updating this flag directly). The change is accomplished by pushing the flags, altering bit 8 of the memory-image and then popping the flags.

**Encoding:**

10011100
----------

POPF Operands	Clocks	Transfers	Bytes	POPF Coding Example
(no operands)	12	1	1	POPF

# PUSH

# PUSH

# PUSH

**Operation:**
$$\begin{aligned} (SP) &\leftarrow (SP) - 2 \\ ((SP) + 1 : (SP)) &\leftarrow (SRC) \end{aligned}$$
**Flags Affected:**

None

**Description:****PUSH** *source*

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

# PUSH

# PUSH

# PUSH

## Encoding:

### Memory or Register Operand:

11111111	mod110 r/m
----------	------------

SRC = EA

### Register Operand:

01010	reg
-------	-----

SRC = REG

### Segment Register:

000	reg	110
-----	-----	-----

SRC = REG

PUSH Operands	Clocks	Transfers	Bytes	PUSH Coding Example
register	15	1	1	PUSH SI
seg-reg (CS legal)	14	1	1	PUSH ES
memory	24 + EA	2	2-4	PUSH RETURN_CODE [SI]



# PUSHF      PUSH FLAGS      PUSHF

## Operation:

$(SP) \leftarrow (SP) - 2$   
 $((SP) + 1:(SP)) \leftarrow \text{Flags}$

## Flags Affected:

None

## Description:

### PUSHF

PUSHF decrements SP (the stack pointer) by two and then transfers all flags to the word at the top of stack pointed to by SP. The flags themselves are not affected.

## Encoding:

10011101
----------

PUSHF Operands	Clocks	Transfers	Bytes	PUSHF Coding Example
(no operands)	14	1	1	PUSHF

**RCL****ROTATE THROUGH  
CARRY LEFT****RCL****Operation:**

```

(temp) ← COUNT
do while (temp) ≠ 0
  (tmpcf) ← (CF)
  (CF) ← high-order bit of (EA)
  (EA) ← (EA) * 2 + (tmpcf)
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined

```

**Flags Affected:**

CF, OF

**Description:**

**RCL** *destination, count*

RCL (Rotate through Carry Left) rotates the bits in the byte or word destination operand to the left by the number of bits specified in the count operand. The carry flag (CF) is treated as "part of" the destination operand; that is, its value is rotated into the low-order bit of the destination, and itself is replaced by the high-order bit of the destination.

# RCL      ROTATE THROUGH CARRY LEFT      RCL

## Encoding:

110100vw	mod010r/m
----------	-----------

if  $v = 0$  then COUNT = 1  
else COUNT = (CL)

RCL Operands	Clocks*	Transfers	Bytes	RCL Coding Example
register 1,	2	—	2	RCL CX, 1
register, CL	8 + 4/bit	—	2	RCL AL, CL
memory, 1	15(23) + EA	2	2-4	RCL ALPHA, 1
memory, CL	20(28) + EA + 4/bit	2	2-4	RCL [BP].PARAM, CL

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# RCR ROTATE THROUGH CARRY RIGHT RCR

## Operation:

```
(temp) ← COUNT
do while (temp) ≠ 0
  (tmpcf) ← (CF)
  (CF) ← low-order bit of (EA)
  (EA) ← (EA) / 2
  high-order bit of (EA) ← (tmpcf)
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ next-
    to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

## Flags Affected:

CF, OF

## Description:

*RCR destination, count*

RCR (Rotate through Carry Right) operates exactly like RCL except that the bits are rotated right instead of left.

## Encoding:

110100vw	mod011r/m
----------	-----------

if  $v = 0$  then  $COUNT = 1$   
 else  $COUNT = (CL)$

RCR Operands	Clocks	Transfers	Bytes	RCR Coding Example
register, 1	2	—	2	RCR BX, 1
register, CL	$8 + 4/\text{bit}$	—	2	RCR BL, CL
memory, 1	$15(23) + EA$	2	2-4	RCR [BX].STATUS, 1
memory, CL	$20(28) + EA + 4/\text{bit}$	2	2-4	RCR ARRAY [DI], CL

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

**REP**                      **REPEAT**                      **REP**

**REPE/REPZ**              **REPE/REPZ**  
REPEAT WHILE EQUAL/  
REPEAT WHILE ZERO

**REPNE/REPZ** **REPNE/REPZ**  
REPEAT WHILE NOT EQUAL/  
REPEAT WHILE NOT ZERO

**Operation:**

do while (CX) ≠ 0  
  service pending interrupt (if  
  any) execute primitive string  
  operation in succeeding byte  
  (CX) ← (CX) - 1  
  if primitive operation is CMPB,  
  CMPW, SCAB, or SCAW and  
  (ZF) ≠ z then exit from  
  while loop

**Flags Affected:**

None

**REP****REPEAT****REP****REPE/REPZ****REPE/REPZ****REPEAT WHILE EQUAL/  
REPEAT WHILE ZERO****REPNE/REPNZ REPNE/REPNZ****REPEAT WHILE NOT EQUAL/  
REPEAT WHILE NOT ZERO****Description:****REP/REPE/REPZ/REPNE/REPNZ**

Repeat, Repeat While Equal, Repeat While Zero, Repeat While Not Equal and Repeat While Not Zero are mnemonics for two forms of the prefix byte that controls subsequent string instruction repetition. The different mnemonics are provided to improve program clarity. The repeat prefixes do not affect the flags.

REP is used in conjunction with the MOVS (Move String) and STOS (Store String) instructions and is interpreted as “repeat while not end-of-string” (CX not 0). REPE and REPZ operate identically and are physically the same prefix byte as REP. These instructions are used with the CMPS (Compare String) and SCAS (Scan String) instructions and require ZF (posted by these instructions) to be set before initiating the next repetition. REPNE and REPNZ are mnemonics for the same prefix byte. These instructions function the same as REPE and REPZ except that the zero flag must be cleared or the repetition is terminated. ZF does not need to be initialized before executing the repeated string instruction.

Repeated string sequences are interruptable; the processor will recognize the interrupt before processing the next string element. System interrupt processing is not affected in any way. Upon return from the interrupt, the repeated operation is resumed from the point of interruption. However, execution does *not* resume properly if a second or third prefix (i.e., segment override or LOCK) has been specified in addition to any of the repeat prefixes. At interrupt time, the processor “remembers” only the prefix that immediately precedes the string instruction. After returning from the interrupt, processing resumes, but any additional prefixes specified are not in effect. If more than one prefix must be used with a string instruction, interrupts may be disabled for the duration of the repeated execution. However, this will not prevent a non-maskable interrupt from being recognized. Also, the time that the system is unable to respond to interrupts may be unacceptable if long strings are being processed.

**REP**                      **REPEAT**                      **REP**

**REPE/REPZ**              **REPE/REPZ**  
 REPEAT WHILE EQUAL/  
 REPEAT WHILE ZERO

**REPNE/REPZ** **REPNE/REPZ**  
 REPEAT WHILE NOT EQUAL/  
 REPEAT WHILE NOT ZERO

### Encoding:

1111001z

REP Operands	Clocks	Transfers	Bytes	REP Coding Example
(no operands)	2	—	1	REP MOVS DEST, SRCE
REPE/REPZ Operands	Clocks	Transfers	Bytes	REPE Coding Example
(no operands)	2	—	1	REPE CMPS DATA, KEY
REPNE/REPZ Operands	Clocks	Transfers	Bytes	REPNE Coding Example
(no operands)	2	—	1	REPNE SCAS INPUT_LINE

# RET

# RETURN

# RET

**Operation:**

$(IP) \leftarrow ((SP)=1:(SP))$   
 $(SP) \leftarrow (SP) + 2$   
if Inter-Segment then  
     $(CS) \leftarrow ((SP) + 1:(SP))$   
     $(SP) \leftarrow (SP) + 2$   
if Add Immediate to Stack Pointer  
    then  $(SP) \leftarrow (SP) + \text{data}$

**Flags Affected:**

None

**Description:****RET** *optional-pop-value*

RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure. The assembler generates an intrasegment RET if the programmer has defined the procedure NEAR, or an intersegment RET if the procedure has been defined as FAR. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and

increments SP by two. If RET is intersegment, the word at the new top of stack is popped into the CS register, and SP is again incremented by two. If an optional pop value has been specified, RET adds that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the CALL instruction.



**RET****RETURN****RET****Encoding:****Intra-Segment:**

11000011

**Intra-Segment and Add Immediate to Stack Pointer:**

11000010 | data-low | data-high

**Inter-Segment:**

11001011

**Inter-Segment and Add Immediate to Stack Pointer:**

11001010 | data-low | data-high

RET Operands	Clocks	Transfers	Bytes	RET Coding Example
(intra-segment, no pop)	20	1	1	RET
(intra-segment, pop)	24	1	3	RET 4
(inter-segment, no pop)	32	2	1	RET
(inter-segment, pop)	31	2	3	RET 2

# ROL

# ROTATE LEFT

# ROL

## Operation:

```

(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← high-order bit of (EA)
  (EA) ← (EA) * 2 + (CF)
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined

```

## Flags Affected:

CF, OF

## Description:

*ROL destination, count*

ROL (Rotate Left) rotates the destination byte or word left by the number of bits specified in the count operand.

## Encoding:

110100vw	mod 000r/m
----------	------------

if v = 0 then COUNT = 1  
else COUNT = (CL)

ROL Operands	Clocks*	Transfers	Bytes	ROL Coding Example
register, 1	2	—	2	ROL BX, 1
register, CL	8 + 4/bit	—	2	ROL DI, CL
memory, 1	15(23) + EA	2	2-4	ROL FLAG_BYTE [DI], 1
memory, CL	20(28) + EA + 4/bit	2	2-4	ROL ALPHA, CL

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# ROR ROTATE RIGHT ROR

## Operation:

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← low-order bit of (EA)
  (EA) ← (EA) / 2
  high-order bit of (EA) ← (CF)
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ next-
  to-high-order bit of (EA)
  then (OF) ← 1
  else (OF) ← 0
  else (OF) undefined
```

## Flags Affected:

CF, OF

## Description:

ROR *destination, count*

ROR (Rotate Right) operates similar to ROL except that the bits in the destination byte or word are rotated right instead of left.

## Encoding:

110100vw	mod 001r/m
----------	------------

if  $v = 0$  then  $COUNT = 1$   
 else  $COUNT = (CL)$

ROR Operand	Clocks*	Transfers	Bytes	ROR Coding Example
register, 1	2	—	2	ROR AL, 1
register, CL	8 + 4/bit	—	2	ROR BX, CL
memory, 1	15(23) + EA	2	2-4	ROR PORT_STATUS, 1
memory, CL	20(28) + EA + 4/bit	2	2-4	ROR CMD_WORD, CL

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# SAHF STORE REGISTER AH INTO FLAGS SAHF

## Operation:

(SF):(ZF):X:(AF):X:(PF):X:(CF) ← (AH)

## Flags Affected:

AF, CF, PF, SF, ZF

## Description:

### SAHF

SAHF (store register AH into flags) transfers bits 7, 6, 4, 2 and 0 from register AH into SF, ZF, AF, PF and CF, respectively, replacing whatever values these flags previously had. OF, DF, IF and TF are not affected. This instruction is provided for 8080/8085 compatibility.

## Encoding:

10011110
----------

SAHF Operands	Clocks	Transfers	Bytes	SAHF Coding Example
(no operands)	4	—	1	SAHF

# SAL SHIFT ARITHMETIC LEFT SAL

# SHL SHIFT LOGICAL LEFT SHL

## Operation:

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← high-order bit of (EA)
  (EA) ← (EA) * 2
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CE)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

## Flags Affected:

CF, OF, PF, SF, ZF.  
AF undefined

## Description:

**SHL/SAL** *destination, count*

SHL and SAL (Shift Logical Left and Shift Arithmetic Left) perform the same operation and are physically the same instruction. The destination byte or word is shifted left by the number of bits specified in the count operand. Zeros are shifted in on the right. If the sign bit retains its original value, then OF is cleared.

**SAL**    **SHIFT ARITHMETIC LEFT**    **SAL**  
**SHL**    **SHIFT LOGICAL LEFT**    **SHL**

### Encoding:

110100vw	mod 100r/m
----------	------------

if  $v = 0$  then  $COUNT = 1$   
else  $COUNT = (CL)$

SAL/SHL Operands	Clocks*	Transfers	Bytes	SAL/SHL Coding Example
register, 1	2	—	2	SAL AH, 1
register, CL	8 + 4/bit	—	2	SHL DI, CL
memory, 1	15(23) + EA	2	2-4	SHL [BX].OVERDRAW, 1
memory, CL	20(28) + EA + 4/bit	2	2-4	SAL STORE_COUNT, CL

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# SAR SHIFT ARITHMETIC RIGHT SAR

## Operation:

```

(temp) ← COUNT
do while (temp) ≠ 0
    (CF) ← low-order bit of (EA)
    (EA) ← (EA) / 2, where / is
        equivalent to signed division,
        rounding down
    (temp) ← (temp) - 1
if COUNT = 1 then
    if high-order bit of (EA) ≠ next-
        to-high-order bit of (EA)
        then (OF) ← 1
    else (OF) ← 0
else (OF) ← 0

```

## Flags Affected:

```

CF, OF, PF, SF, ZF.
AF undefined

```

## Description:

**SAR** *destination, count*

SAR (Shift Arithmetic Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Bits equal to the original high-order (sign) bit are shifted in on the left, preserving the sign of the original value. Note that SAR does not produce the same result as the dividend of an "equivalent" IDIV instruc-

tion if the destination operand is negative and 1-bits are shifted out. For example, shifting -5 right by one bit yields -3, while integer division -5 by 2 yields -2. The difference in the instructions is that IDIV truncates all numbers toward zero, while SAR truncates positive numbers toward zero and negative numbers toward negative infinity.

**SAR****SHIFT ARITHMETIC  
RIGHT****SAR****Encoding:**

110100vw	mod 111r/m
----------	------------

if  $v = 0$  then COUNT = 1  
 else COUNT = (CL)

SAR Operands	Clocks*	Transfers	Bytes	SAR Coding Example
register, 1	2	—	2	SAR DX, 1
register, CL	8 + 4/bit	—	2	SAR DI, CL
memory, 1	15(23) + EA	2	2-4	SAR N_BLOCKS, 1
memory, CL	20(28) + EA + 4/bit	2	2-4	SAR N_BLOCKS, CL

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.



**SBB****SUBTRACT WITH  
BORROW****SBB****Operation:**

if (CF) = 1 then (DEST) = (LSRC) -  
(RSRC) - 1  
else (DEST) ← (LSRC) - (RSRC)

**Flags Affected:**

AF, CF, OF, PF, SF, ZF

**Description:**

**SBB** *destination, source*

SBB (Subtract with Borrow) subtracts the source from the destination, subtracts one if CF is set, and returns the result to the destination operand. Both operands may be bytes or words. Both operands may be signed or

unsigned binary numbers (see AAS and DAS). SBB updates AF, CF, OF, PF, SF, and ZF. Since it incorporates a borrow from a previous operation, SBB may be used to write routines that subtract numbers longer than 16 bits.

# SBB                      SUBTRACT WITH BORROW                      SBB

## Encoding:

### Memory or Register Operand and Register Operand:

000110dw	mod reg r/m
----------	-------------

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG  
 else LSRC = EA, RSRC = REG, DEST = EA

### Immediate Operand from Memory or Register Operand:

100000sw	mod 011 r/m	data	data if s:w=01
----------	-------------	------	----------------

LSRC = EA, RSRC = data, DEST = EA

### Immediate Operand from Accumulator:

0001110w	data	data if w=1
----------	------	-------------

if w = 0 then LSRC = AL, RSRC = data, DEST = AL  
 else LSRC = AX, RSRC = data, DEST = AX

SBB Operands	Clocks*	Transfers	Bytes	SBB Coding Example
register, register	3	—	2	SBB BX, CX
register, memory	9(13) + EA	1	2-4	SBB DI, [BX].PAYMENT
memory, register	16(24) + EA	2	2-4	SBB BALANCE, AX
accumulator, immediate	4	—	2-3	SBB AX, 2
register, immediate	4	—	3-4	SBB CL, 1
memory, immediate	17(25) + EA	2	3-6	SBB COUNT [SI], 10

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# SCAS

# SCAN (BYTE OR WORD) STRING

# SCAS

## Operation:

(LSRC) - RSRC  
 if (DF) = 0 then (DI) ← (DI) + DELTA  
 else (DI) ← (DI) - DELTA

## Flags Affected:

AF, CF, OF, PF, SF, ZF

## Description:

### SCAS *destination-string*

SCAS (Scan String) subtracts the destination string element (byte or word) addressed by DI from the content of AL (byte string) or AX (word string) and updates the flags, but does not alter the destination string or the accumulator. SCAS also updates DI to point to the next string element and AF, CF, OF, PF, SF and ZF to reflect the relationship of the scan value in AL/AX to the string element. If

SCAS is prefixed with REPE or REPZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element = scan-value (ZF = 1)." This form may be used to scan for departure from a given value. If SCAS is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element is not equal to scan-value (ZF = 0)." This form may be used to locate a value in a string.

## Encoding:

1010111w

if w = 0 then LSRC = AL, RSRC = (DI), DELTA = 1  
 else LSRC = AX, RSRC = (DI) + 1:(DI), DELTA = 2

SCAS Operands	Clocks*	Transfers	Bytes	SCAS Coding Example
dest-string	15(19)	1	1	SCAS INPUT_LINE
(repeat) dest-string	9 + 15(19)/rep	1/rep	1	REPNE SCAS BUFFER

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# SCASB

# SCAN BYTE STRING

# SCASB

## Operation:

(LSRC) - RSRC)  
 if (DF) = 0 then (DI) ← (DI) + DELTA  
 else (DI) ← (DI) - DELTA

## Flags Affected:

AF, CF, OF, PF, SF, ZF

## Description:

SCASB (Scan Byte String) subtracts the destination string element addressed by DI from the content of AL and updates the flags, but does not alter the destination string or the accumulator. SCASB also updates DI to point to the next string element and AF, CF, OF, PF, SF and ZF to reflect the relationship of the scan value in AL to the string element. If SCASB is prefixed with REPE or REPZ, the operation is interpreted as "scan while not

end-of-string (CX not 0) and string-element = scan-value (ZF = 1)." This form may be used to scan for departure from a given value. If SCASB is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element is not equal to scan-value (ZF = 0)." This form may be used to locate a value in a string.

## Encoding:

10101110

LSRC = AL, RSRC = (DI), DELTA = 1

SCASB Operands	Clocks	Transfers	Bytes	SCASB Coding Examples
(NO OPERANDS)	15	1	1	SCASB
(NO OPERANDS)	9 + 15/rep	1/rep	1	REPNE SCASB

# SCASW      SCAN WORD STRING      SCASW

## Operation:

(LSRC) - RSRC)  
 if (DF) = 0 then (DI) ← (DI) + DELTA  
 else (DI) ← (DI) - DELTA

## Flags Affected:

AF, CF, OF, PF, SF, ZF

## Description:

SCASW (Scan Word String) subtracts the destination string element addressed by DI from the content of AX and updates the flags, but does not alter the destination string or the accumulator. SCASW also updates DI to point to the next string element and AF, CF, OF, PF, SF and ZF to reflect the relationship of the scan value in AX to the string element. If SCASW is prefixed with REPE or REPZ, the operation is interpreted as “scan while not

end-of-string (CX not 0) and string-element = scan-value (ZF = 1).” This form may be used to scan for departure from a given value. If SCASW is prefixed with REPNE or REPNZ, the operation is interpreted as “scan while not end-of-string (CX not 0) and string-element is not equal to scan-value (ZF = 0).” This form may be used to locate a value in a string.

## Encoding:

10101111
----------

LSRC = AX, RSRC = (DI) + 1:(DI), DELTA = 2

SCASW Operands	Clocks	Transfers	Bytes	SCASW Coding Examples
(NO OPERANDS)	19	1	1	SCASW
(NO OPERANDS)	9 + 19/rep	1/rep	1	REPNE SCASW

# SHR    SHIFT LOGICAL RIGHT    SHR

## Operation:

```
(temp) ← COUNT
do while (temp) ≠ 0
  CF ← low-order bit of (EA)
  (EA) ← (EA) / 2, where / is
    equivalent to unsigned
    division
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ next-
    to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

## Flags Affected:

CF, OF, PF, SF, ZF.  
AF undefined

## Description:

*SHR destination, source*

SHR (Shift Logical Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Zeros are shifted in on the left. If the sign bit retains its original value, then OF is cleared.

# SHR SHIFT LOGICAL RIGHT SHR

## Encoding:

110100vw	mod 101r/m
----------	------------

if  $v = 0$  then COUNT = 1  
 else COUNT = (CL)

SHR Operands	Clocks*	Transfers	Bytes	SHR Coding Example
register, 1	2	—	2	SHR SI, 1
register, CL	$8 + 4/\text{bit}$	—	2	SHR SI, CL
memory, 1	$15(23) + \text{EA}$	2	2-4	SHR ID_BYTE [SI + BX], 1
memory, CL	$20(28) + \text{EA} + 4/\text{bit}$	2	2-4	SHR INPUT_WORD, CL

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# STC

# SET CARRY

# STC

**Operation:**

(CF) ← 1

**Flags Affected:**

CF

**Description:**

STC

STC (Set Carry flag) sets CF to 1 and affects no other flags.

**Encoding:**

11111001
----------

STC Operands	Clocks	Transfers	Bytes	STC Coding Example
(no operands)	2	—	1	STC



# STD SET DIRECTION FLAG STD

**Operation:**

(DF) ← 1

**Flags Affected:**

DF

**Description:****STD**

STD (Set Direction flag) sets DF to 1 causing the string instructions to auto-decrement the SI and/or DI index registers. STD does not affect any other flags.

**Encoding:**

11111101
----------

Timing: 2 clocks

STD Operands	Clocks	Transfers	Bytes	STD Coding Example
(no operands)	2	—	1	STD

**STI****SET INTERRUPT-  
ENABLE FLAG****STI****Operation:**

(IF) ← 1

**Flags Affected:**

IF

**Description:**

STI (Set Interrupt-enable flag) sets IF to 1, enabling processor recognition of maskable interrupt requests appearing on the INTR line. Note however, that a pending interrupt will not actually be recognized until the instruction following STI has executed. STI does not affect any other flags.

**Encoding:**

11111011
----------

STI Operands	Clocks	Transfers	Bytes	STI Coding Example
(no operands)	2	—	1	STI

# STOS STORE (BYTE OR WORD) STRING STOS

## Operation:

$(DEST) \leftarrow (SRC)$   
 if  $(DF) = 0$  then  $(DI) \leftarrow (DI) + DELTA$   
 else  $(DI) \leftarrow (DI) - DELTA$

## Flags Affected:

None

## Description:

### STOS *destination-string*

STOS (Store String) transfers a byte or word from register AL or AX to the string element addressed by DI and updates DI to point to the next location in the string. As a repeated operation, STOS provides a convenient way to initialize a string to a constant value (e.g., to blank out a print line).

## Encoding:

1 0 1 0 1 0 1 w

if  $w = 0$  then  $SRC = AL$ ,  $DEST = (DI)$ ,  $DELTA = 1$   
 else  $SRC = AX$ ,  $DEST = (DI) + 1:(DI)$ ,  $DELTA = 2$

STOS Operands	Clocks*	Transfers	Bytes	STOS Coding Example
dest-string	11(15)	1	1	STOS PRINT_LINE
(repeat) dest-string	9 + 10(14)/rep	1/rep	1	REP STOS DISPLAY

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# STOSB      STORE BYTE STRING      STOSB

## Operation:

$(DEST) \leftarrow (SRC)$   
 if  $(DF) = 0$  then  $(DI) \leftarrow (DI) + DELTA$   
 else  $(DI) \leftarrow (DI) - DELTA$

## Flags Affected:

None

## Description:

STOSB (Store Byte String) transfers a byte from register AL to the string element addressed by DI and updates DI to point to the next location in the string. As a repeated operation, STOSB provides a convenient way to initialize a string to a constant value (e.g., to blank out a print line).

## Encoding:

10101010

SRC = AL, DEST = (DI), DELTA = 1

STOSB Operands	Clocks	Transfers	Bytes	STOSB Coding Examples
(NO OPERANDS)	11	1	1	STOSB
(NO OPERANDS)	9 + 10/rep	1/rep	1	REP STOSB

# STOSW STORE WORD STRING STOSW

## Operation:

$(DEST) \leftarrow (SRC)$   
 if  $(DF) = 0$  then  $(DI) \leftarrow (DI) + DELTA$   
 else  $(DI) \leftarrow (DI) - DELTA$

## Flags Affected:

None

## Description:

STOSW (Store Word String) transfers a word from register AX to the string element addressed by DI and updates DI to point to the next location in the string. As a repeated operation, STOSW provides a convenient way to initialize a string to a constant value (e.g., to blank out a print line).

## Encoding:

10101011

SRC = AX, DEST = (DI) + 1:(DI), DELTA = 2

STOSW Operands	Clocks	Transfers	Bytes	STOSW Coding Examples
(NO OPERANDS)	15	1	1	STOSW
(NO OPERANDS)	9 + 14/rep	1/rep	1	REP STOSW

**SUB****SUBTRACT****SUB****Operation:** $(DEST) \leftarrow (LSRC) - (RSRC)$ **Flags Affected:**

AF, CF, OF, PF, SF, ZF

**Description:****SUB** *destination, source*

The source operand is subtracted from the destination operand, and the result replaces the destination operand. The operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SUB updates AF, CF, OF, PF, SF and ZF.

# SUB

# SUBTRACT

# SUB

## Encoding:

### Memory or Register Operand and Register Operand:

001010 d w	mod reg r/m
------------	-------------

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG  
 else LSRC = EA, RSRC = REG, DEST = EA

### Immediate Operand from Memory or Register Operand:

100000 s w	mod 101 r/m	data	data if s:w=01
------------	-------------	------	----------------

LSRC = EA, RSRC = data, DEST = EA

### Immediate Operand from Accumulator:

0010110 w	data	data if w=1
-----------	------	-------------

if w = 0 then LSRC = AL, RSRC = data, DEST = AL  
 else LSRC = AX, RSRC = data, DEST = AX

SUB Operands	Clocks*	Transfers	Bytes	SUB Coding Example
register, register	3	—	2	SUB CX, BX
register, memory	9(13) + EA	1	2-4	SUB DX, MATH_TOTAL [SI]
memory, register	16(24) + EA	2	2-4	SUB [BP + 2], CL
accumulator, immediate	4	—	2-3	SUB AL, 10
register, immediate	4	—	3-4	SUB SI, 5280
memory, immediate	17(25) + EA	2	3-6	SUB [BP].BALANCE, 1000

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# TEST

# TEST

# TEST

**Operation:**

(LSRC) & (RSRC)  
(CF) ← 0  
(OF) ← 0

**Flags Affected:**

CF, OF, PF, SF, ZF.  
AF undefined

**Description:**

**TEST** *destination, source*

TEST performs the logical “and” of the two operands (byte or word), updates the flags, but does not return the result, i.e., neither operand is changed. If a TEST instruction is followed by a JNZ (jump if not zero) instruction, the jump will be taken if there are any corresponding 1-bits in both operands.



**TEST****TEST****TEST****Encoding:****Memory or Register Operand with Register Operand:**

1000010w	mod reg r/m
----------	-------------

LSRC = REG, RSRC = EA

**Immediate Operand with Memory or Register Operand:**

1111011w	mod 000 r/m	data	data if w=1
----------	-------------	------	-------------

LSRC = EA, RSRC = data

**Immediate Operand with Accumulator:**

1010100w	data	data if w=1
----------	------	-------------

if w = 0 then LSRC = AL, RSRC = data  
 else LSRC = AX, RSRC = data

TEST Operands	Clocks	Transfers	Bytes	TEST Coding Example
register, register	3	—	2	TEST SI, DI
register, memory	9(13) + EA	1	2-4	TEST SI, END_COUNT
accumulator, immediate	4	—	2-3	TEST AL, 00100000B
register, immediate	5	—	3-4	TEST BX, 0CC4H
memory, immediate	11 + EA	—	3-6	TEST RETURN_CODE, 01H

# WAIT

# WAIT

# WAIT

**Operation:**

None

**Flags Affected:**

None

**Description:**

WAIT causes the CPU to enter the wait state while its TEST line is not active. WAIT does not affect any flags.

**Encoding:**

10011011
----------

WAIT Operands	Clocks	Transfers	Bytes	WAIT Coding Example
(no operands)	$3 + 5n$	—	1	WAIT

# XCHG

# EXCHANGE

# XCHG

**Operation:**

(temp) ← (DEST)  
(DEST) ← (SRC)  
(SRC) ← (temp)

**Flags Affected:**

None

**Description:**

**XCHG** *destination, source*

XCHG (exchange) switches the contents of the source and destination (byte or word) operands. When used in conjunction with the LOCK prefix, XCHG can test and set a semaphore that controls access to a resource shared by multiple processors (see section 2.5).

# XCHG          EXCHANGE          XCHG

## Encoding:

### Memory or Register Operand with Register Operand:

1000011w | mod reg r/m

SRC = EA, DEST = REG

### Register Operand with Accumulator:

10010 reg

SRC = REG, DEST = AX

XCHG Operands	Clocks*	Transfers	Bytes	XCHG Coding Example
accumulator, reg16	3	—	1	XCHG AX, BX
memory, register	17(25) + EA	2	2-4	XCHG SEMAPHORE, AX
register, register	4	—	2	XCHG AL, BL

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.

# XLAT                      TRANSLATE                      XLAT

## Operation:

$AL \leftarrow ((BX) + (AL))$

## Flags Affected:

None

## Description:

### *XLAT translate-table*

XLAT (translate) replaces a byte in the AL register with a byte from a 256-byte, user-coded translation table. Register BX is assumed to point to the beginning of the table. The byte in AL is used as an index into the table and is replaced by the byte at the offset in the table corresponding to AL's binary value.

The first byte in the table has an offset of 0. For example, if AL contains 5H, and the sixth element of the translation table contains 33H, then AL will contain 33H following the instruction. XLAT is useful for translating characters from one code to another, the classic example being ASCII to EBCDIC or the reverse.

## Encoding:

11010111
----------

XLAT Operands	Clocks	Transfers	Bytes	XLAT Coding Example
source-table	11	1	1	XLAT ASCII_TAB

# XOR

# EXCLUSIVE OR

# XOR

**Operation:**

(DEST) ← (LSRC) XOR (RSRC)  
(CF) ← 0  
(OF) ← 0

**Flags Affected:**

CF, OF, PF, SF, ZF.  
AF undefined

**Description:**

*XOR destination, source*

XOR (Exclusive Or) performs the logical “exclusive or” of the two operands and returns the result to the destination operand. A bit in the result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is cleared); otherwise the result bit is cleared.

# XOR EXCLUSIVE OR XOR

## Encoding:

### Memory or Register Operand with Register Operand:

001100d w	mod reg r/m
-----------	-------------

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG  
 else LSRC = EA, RSRC = REG, DEST = EA

### Immediate Operand to Memory or Register Operand:

1000000 w	mod 110 r/m	data	data if w=1
-----------	-------------	------	-------------

LSRC = EA, RSRC = data, DEST = EA

### Immediate Operand to Accumulator:

0011010 w	data	data if w=1
-----------	------	-------------

if w = 0 then LSRC = AL, RSRC = data, DEST = AL  
 else LSRC = AX, RSRC = data, DEST = AX

XOR Operands	Clocks*	Transfers	Bytes	XOR Coding Example
register, register	3	—	2	XOR CX, BX
register, memory	9(13) + EA	1	2-4	XOR CL, MASK_BYTE
memory, register	16(24) + EA	2	2-4	XOR ALPHA [SI], DX
accumulator, immediate	4	—	2-3	XOR AL, 01000010B
register, immediate	4	—	3-4	XOR SI, 00C2H
memory, immediate	17(25) + EA	2	3-6	XOR RETURN_CODE, 0D2H

\*b(w): where b denotes the number of clock cycles for byte operands and w denotes the number of clock cycles for word operands.





# INDEX

## A

- Add ( + ) Operator, 2-33
- Addition Instructions, 6-17
- ALU, 1-18
- Argument, 2-15
- Arithmetic Instructions, 6-12
- Arithmetic Logic Unit, 1-18
- Arithmetic Operators, 2-33
- Array, 5-6
- ASCII Table, 2-19, 6-22
- Assembler Directive, 2-6, 14
- Assembly Language, 2-11
- Assembly Language Listing, 2-7
- ASSUME Directive, 2-39, 7-13
- Auxiliary Carry Flag, 3-22

## B

- Base Address, 7-9
- Base Pointer Register, 1-30, 7-11
- Based Addressing, 5-40
- Based Index Addressing, 5-45
- Binary Code, 2-11
- Bit, 1-10
- Bit Manipulation Instructions, 6-44
- Breakpoint Interrupt Instruction, 8-10
- Buffer, 5-38
- Bus, 1-9
- Bus Cycle, 1-33
- Bus Interface Unit, 1-18
- Byte, 1-10

## C

- CALL, 4-9, 24
- Call Instruction, 4-6, 23
- Carry Flag, 3-21
- Central Processing Unit, 1-7

- Character String, 2-21
- Clock, 1-9
- Clock Cycle, 1-33
- Code Segment, 1-29
- Command File, 2-8
- Comments, 2-12, 15
- Compare Instruction, 3-28
- Conditional Directive, 9-20
- Conditional Jump Table, 3-24
- Conditional Jump, 3-20, 25
- Conditional Loop, 3-36
- Conditional Macro, 9-44
- Count register, 3-33
- CPU, 1-7
- CX Register, 3-33

## D

- Data Register, 1-17
- Data Segment, 1-30
- Data Transfer Instructions, 6-7
- DB Directive, 2-17, 5-8, 8-39
- Default Register Table, 1-21
- Delimiter, 2-12
- Destination Index, 1-30
- Destination Operand, 1-36, 2-27
- Direct Jump, 3-17
- Direct Memory Access, 8-22
- Direct Memory Addressing, 2-29
- Direction Flag, 8-36
- Directive Statement, 2-14
- Directive, 2-6
- Displacement, 3-19
- Divide (/) Operator, 2-34
- Divide Error Interrupt, 8-8
- Division Instructions, 6-40
- Doubleword, 1-11
- DQ Directive, 5-8
- DT Directive, 5-8
- DW Directive, 2-22, 5-8

- E**
- Editor, 2-7
  - Effective Address, 1-30
  - Element, 5-12
  - ELSE Directive, 9-28
  - END directive, 2-23
  - ENDIF Directive, 9-20
  - ENDM Directive, 9-32
  - ENDP Directive, 7-28
  - ENDS Directive, 2-39
  - EQU Directive, 2-16
  - EXE Program Structure, 7-6
  - EXE2BIN, 2-8
  - Executable File, 2-8
  - Execution Unit, 1-17
  - External Interrupts, 8-17
  - Extra Segment, 1-30
  - EXTRN Directive, 7-24
- F**
- Far Call, 7-27
  - Far Jump, 7-26
  - Field, 5-12, 31, 33
  - File Extension, 2-7
  - Fixed Port Addressing, 7-33
  - Flag Register, 1-21, 3-20
  - Flowchart, 3-6
  - Flowchart Symbols, 3-7
- G**
- General Register, 1-17, 20
  - GROUP Directive, 9-15
- H**
- Handshaking, 8-17
  - Hardware Reset, 8-20
  - HLT Instruction, 2-26
  - Hold Request, 8-22
- I**
- I/O Addressing, 7-30
  - I/O Devices, 1-9
  - I/O Port, 1-9
  - I/O Port Addressing, 7-32
  - I/O Port Structure, 7-30
  - IF Directive, 9-22
  - IF1 Directive, 9-25
  - IF2 Directive, 9-25
  - IFB Directive, 9-27
  - IFDEF Directive, 9-26
  - IFDIF Directive, 9-27
  - IFE Directive, 9-25
  - IFIDN Directive, 9-27
  - IFNB Directive, 9-27
  - IFNDEF Directive, 9-26
  - Immediate Operand, 1-36, 2-26
  - INCLUDE Directive, 4-36
  - Include File Format, 4-37
  - Indexed Addressing, 5-44
  - Indirect Jump, 3-19
  - Inherent Control, 2-26
  - Input, 1-9, 7-30
  - Instruction Coding, 1-34
  - Instruction Pointer, 1-19
  - Instruction Queue, 1-19
  - Instruction Set, 1-34, 2-12, D-1
  - Instruction Statement, 2-11
  - Integer Numbers, 6-37, 41
  - Interface Adapter, 1-9
  - Internal Interrupts, 8-6
  - Interrupt Flag, 8-18
  - Interrupt On Overflow Instruction, 8-11
  - Interrupt Priority Table, 8-19
  - Interrupt Request, 8-17
  - Interrupt Type 0, 6-40
  - Interrupt Type 1, 8-9
  - Interrupt Type 2, 8-10
  - Interrupt Type 3, 8-10
  - Interrupt Type 4, 8-11
  - Interrupt Types, 8-8
  - Interrupt Vector Table, 8-7
  - Interrupts, 8-6
  - Intersegment, 5-3

Intrasegment Addressing, 7-23, 26  
Intrasegment, 3-17, 5-3  
IRP Macro Directive, 9-40  
IRPC Macro Directive, 9-42

## J

JMP Instruction, 3-18  
Jump Conditional, 3-20, 25  
Jump Direct, 3-17  
Jump Indirect, 3-19  
Jump Instruction Table, 3-24  
Jump Instructions, 3-16  
Jump Unconditional, 3-17

## L

Label, 2-11  
LABEL Directive, 4-14  
LALL Directive, 9-39, 62  
LFCOND Directive, 9-62  
Linker, 2-8  
LIST Directive, 9-62  
LOCAL Directive, 9-45  
Logical Address, 1-26, 30, 7-9  
Logical Instructions, 6-44  
Loop Conditional, 3-36  
Loop Unconditional, 3-33  
Loops, 3-32

## M

Machine Code, 1-10, 34, 2-11  
Machine Language, 2-11  
Macro Call, 9-33  
Macro Definition, 9-32  
MACRO Directive, 9-32  
Macro Operator !, 9-51  
Macro Operator %, 9-51  
Macro Operator &, 9-48  
Macro Operator ;;, 9-50

Macro Operator <>, 9-50  
MASK Operator, 5-31, 33  
Memory, 1-9  
Memory Length, 1-24  
Memory Operand, 2-29  
Memory Segmentation, 1-25  
Memory Width, 1-23  
Microcomputer Language, 1-10  
Microcomputer, 1-7  
Microprocessing Unit, 1-7  
Mnemonic, 2-11  
MOD Operator, 2-35  
MPU, 1-7  
Multiplication Instructions, 6-36  
Multiply (\*) Operator, 2-34

## N

Name, 2-14  
Non-Maskable Interrupt, 8-10, 19  
NOP Instruction, 2-26

## O

Object File, 2-7  
Offset Address, 2-15  
OFFSET Operator, 4-14  
Opcode, 1-35, 2-12  
Operands, 2-12  
Operation Code, 1-35  
ORG Directive, 2-15  
Output, 1-9, 7-30  
Overflow Flag, 3-23  
%OUT Directive, 9-26, 61

## P

Packed Decimal, 6-12, 26, 34  
PAGE Directive, 9-54  
Parity Flag, 3-21  
Physical Address, 1-26, 7-9

Pointer Table, 8-7  
Pop Instructions, 4-19  
Port Addressing, 7-32  
PROC Directive, 7-28, 9-6  
Procedure Call, 9-10  
Procedure Nesting, 9-12  
Procedure, 7-28, 9-6  
Processor Control, 2-26  
Program Organization, 3-6  
Program Segmentation, 2-38  
Pseudo-Opcode, 2-15  
PTR Operator, 2-30  
PURGE Directive, 9-45  
PUBLIC Directive, 7-24  
Push Instructions, 4-15

**R**

RADIX Directive, 2-26  
RAM, 1-24  
Record Access, 5-29  
RECORD Directive, 5-23  
Record Initialization, 5-26  
Record Template, 5-23  
Record Width, 5-33  
Register Indirect Addressing, 5-35  
Register Loading, 8-38  
Register Operand, 2-28  
Relocatable Code, 2-8, 38  
Remainder (MOD) Operator, 2-35  
Repeat String Instruction Table, 8-37  
Repeat String Prefix, 8-29, 37  
REPT Macro Directive, 9-35  
Reserved Word, 2-13, 39  
Reset, 8-20  
RET Instruction, 4-9, 25  
Return From Interrupt Instruction, 8-8  
Return Instruction, 4-9, 26  
ROM, 1-24  
Rotate Instructions, 6-50  
Run File, 2-8

**S**

SALL Directive, 9-62  
SEG Operator, 8-13  
Segment, 1-25  
Segment Addressing, 7-9  
Segment Align-Type, 7-17  
Segment Attribute AT, 8-12  
Segment Attributes, 7-17  
Segment Combine-Type, 7-18  
SEGMENT Directive, 2-38  
Segment Override Prefix, 2-40, 7-11  
SFCOND Directive, 9-62  
Shift Instructions, 6-48  
Shift Operator, 2-35  
Sign Flag, 3-22  
Single-Step Interrupt, 8-9  
Source Code, 2-7  
Source Operand, 1-36, 2-27  
Source Program, 2-7  
Stack Pointer Register, 4-16  
Stack Segment, 1-30, 2-8  
Stack, 4-11  
Stored Program Concept, 1-10  
String Direction, 8-36  
String Instruction Table, 8-33  
String Instructions, 8-26  
String Operations, 8-25  
String, 8-25  
STRUC Directive, 5-12  
Structure Access, 5-19  
Structure Initialization, 5-13  
Structure Template, 5-12  
Subroutine Call, 4-6  
Subroutine Nesting, 4-32  
Subtract ( - ) Operator, 2-34  
Subtraction Instructions, 6-33  
SUBTTL Directive, 9-59  
Symbol, 2-15  
Symbolic Link, 2-40  
Symbolic Notation, 2-6

**T**

Target, 3-17  
TITLE Directive, 9-59  
Trap Flag, 8-9

**U**

Unconditional Jump, 3-17  
Unconditional Loop, 3-33  
Unpacked Decimal, 6-12, 18, 33, 42

**V**

Variable Port Addressing, 7-34  
Variable, 2-15  
Vector Table, 8-7

**W**

Word, 1-10

**X**

XALL Directive, 9-62  
XLIST Directive, 9-62

**Z**

Zero Flag, 3-22





