RAM memory is just below ROM, consisting of four banks of 9 chips each. Each chip represents 64 kilobits, so 8 of these make up 64K bytes of memory. The ninth chip in each bank is for parity checking, a way of catching errors when reading and writing memory. Four banks of 64K memory are equal to 256K of memory, which originally was the maximum memory that could be placed on the standard IBM-PC board. Many computers built more recently use memory chips with a greater capacity, usually 256 kilobits per chip. Most IBM-PCs today have an additional expansion board with another 384K of memory, to fill it up to the maximum 640K. Newer IBM-AT computers allow 1 MB of memory to be socketed on the system board.

THE CPU REGISTERS

Registers are special work areas inside the CPU designed to be accessed at high speed. The registers are 16 bits long, but you have the option of accessing the upper or lower halves of the four data registers:

Data registers	16-bit: AX, BX, CX, DX 8-bit: AH, AL, BH, BL, CH, CL, DH, DL
Segment registers	CS, DS, SS, ES
Index registers	SI, DI, BP
Special registers	IP, SP
Flags registers	Overflow, Direction, Interrupt, Trap, Sign, Zero, Auxiliary Carry, Parity, Carry

Data Registers. Four registers, named *data registers* or *general-purpose registers*, are used for arithmetic and data movement. Each register may be addressed as either a 16-bit or 8-bit value. For example, the AX register is a 16-bit register; its upper 8 bits are called AH, and its lower 8 bits are called AL. Bit positions are always numbered from right to left, starting with 0:

bits: 15 0

16-bit AX register	
AH register	AL register

bits: 7 . . . . . 0 7 0

Instructions may address either 16-bit or 8-bit data registers from the following list:

AX		BX		CX		DX	
AH	AL	BH	BL	CH	CL	DH	DL

When a 16-bit register is modified, so is its corresponding 8-bit half register. Let's say, for example, that the AX register contains 0000h. If we move 126Fh to AX, AL will change to 6Fh.

Each general-purpose register has special attributes:

AX (*accumulator*). AX is called the accumulator register because it is favored by the CPU for arithmetic operations. Other operations are also slightly more efficient when performed using AX.

BX (*base*). Like the other general-purpose registers, the BX register can perform arithmetic and data movement, and it has special addressing abilities. It can hold a memory address that points to another variable. Three other registers with this ability are SI, DI, and BP.

CX (*counter*). The CX register acts as a counter for repeating or looping instructions. These instructions automatically repeat and decrement CX and quit when it equals 0.

DX (*data*). The DX register has a special role in multiply and divide operations. When multiplying, for example, DX holds the high 16 bits of the product.

Segment Registers

The CPU contains four *segment registers,* used as base locations for program instructions, data, and the stack. In fact, all references to memory on the IBM-PC involve a segment register used as a base location. The segment registers are:

CS (*code segment*). The CS register holds the base location of all executable instructions (code) in a program.

DS (*data segment*). The DS register is the default base location for memory variables. The CPU calculates the offsets of variables using the current value of DS.

SS (*stack segment*). The SS register contains the base location for the current program stack.

ES (*extra segment*). The ES register is an additional base location for memory variables.

Index Registers

Index registers contain the offsets of variables. The term *offset* refers to the distance of a variable, label, or instruction from its base segment. Index registers speed up processing of strings, arrays, and other data structures containing multiple elements. The index registers are:

SI (*source index*). This register takes its name from the 8088's string movement instructions, where the source string is pointed to by the SI register. SI usually contains an offset value from the DS register, but it can address any variable.

DI (*destination index*). The DI register acts as the destination for the 8088's string movement instructions. It usually contains an offset from the ES register, but it can address any variable.

BP (*base pointer*). The BP register contains an assumed offset from the SS register, as does the stack pointer. The BP register is often used by a subroutine to locate variables that were passed on the stack by a calling program.

Special Registers

The IP and SP registers are grouped together here, since they do not fit into any of the previous categories:

IP (*instruction pointer*). The IP register always contains the location of the next instruction to be executed. CS and IP registers combine to form the address of the next instruction about to be executed.

SP (*stack pointer*). The SP register contains the *offset*, or distance from the beginning of the stack segment to the top of the stack. The SS and SP registers combine to form the complete top-of-stack address.

Flags Register

The *flags* register is a special 16-bit register with individual bit positions assigned to show the status of the CPU or the results of arithmetic operations. Each relevant bit position is given a name; other positions are undefined:

Bit Position

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	O	D	I	T	S	Z	x	A	x	P	x	C

O = Overflow
D = Direction
I = Interrupt
T = Trap
x = undefined

S = Sign
Z = Zero
A = Auxiliary Carry
P = Parity
C = Carry

Fortunately, we do not have to memorize each flag position. Instead there are special 8088 instructions designed to test and manipulate the flags. A flag or bit is *set* when it equals 1; it is *clear* (or reset) when it equals 0. The CPU sets flags by turning on individual bits in the Flags register. There are two basic types of flags: *control flags* and *status flags*.

Control Flags. Individual bits may be set in the Flags register by the programmer to control the CPU's operation. These are the *Direction*, *Interrupt*, and *Trap* flags. Abbreviations used by DEBUG and CODEVIEW debugger programs are shown in parentheses.

The *Direction* flag controls the assumed direction used by string processing instructions. The flag values are 1 = Up (UP) and 0 = Down (DN). The programmer controls this flag, using the STD and CLD instructions.

The *Interrupt* flag makes it possible for external interrupts to occur. These interrupts are caused by hardware devices such as the keyboard, disk drives, and the system clock timer. The Interrupt flag is cleared by the programmer when an important operation is going on that must not be interrupted. The flag must then be set to allow the system to process interrupts normally again. The flag values are 1 = Enabled (EI) and 0 = Disabled (DI), and are controlled by the CLI and STI instructions.

The *Trap* flag determines whether the CPU should be halted after each instruction. Debugging programs use this flag to allow the user to execute one instruction at a time (called *tracing*). The flag values are 1 = Trap on and 0 = Trap off, and the flag may be set by the INT 3 instruction.

Status Flags. The status flag bits reflect the outcome of arithmetic and logical operations performed by the CPU. These are the *Overflow, Sign, Zero, Auxiliary Carry, Parity, and Carry* flags.

The *Carry* flag is set when the result of an arithmetic operation is too large to fit into the destination. For example, if the values 200 and 56 were added together and placed in an 8-bit destination, the result (256) would be too large and the Carry flag would be set. The flag values are 1 = Carry (CY) and 0 = No carry (NC).

The *Overflow* flag is set when the signed result of an arithmetic operation may be too large to fit into the destination area. The flag values are 1 = overflow (OV) and 0 = no overflow (NV).

The *Sign* flag is set when the result of an arithmetic or logical operation generates a negative result. Since a negative number always has a 1 in the highest bit position, the Sign flag is always a copy of the destination's sign bit. The flag values are Negative (NG) and Positive (PL).

The *Zero* flag is set when the result of an arithmetic or logical operation generates a result of zero. The flag is used primarily by jump and *loop* instructions, in order to allow branching to a new location in a program based on the comparison of two values. The flag values are Zero (ZR) and Not Zero (NZ).

The *Auxiliary Carry* flag is set when an operation causes a carry from bit 3 to bit 4 (or a borrow from bit 4 to bit 3) of an operand. It is rarely used by the programmer. The flag values are Aux Carry (AC) and No Aux Carry (NA).

The *Parity* flag reflects the number of bits in the result of an operation that are set. If there is an even number of bits, the Parity is even (displayed as PE). If there is an odd number of bits, the Parity is odd (displayed as PO). This flag is used by the operating system to verify memory integrity and by communications software to verify correct transmission of data.

The DOS data area (00400h to 005FFh) contains variables used by DOS. For example:

- The keyboard buffer, where all keystrokes are stored until they can be processed.
- The keyboard status flag, showing which keys are currently being pressed.
- The locations of the printer ports.
- The locations of the serial ports.
- A description of the equipment available in the system: the amount of memory, number of disk drives, video monitor type, and so on.

User RAM. The resident portion of DOS is located at address 00600h, and free memory begins immediately above DOS. The size of DOS has increased steadily over the past several years, so its size will vary between 23K and 40K. The total amount of free RAM on the standard IBM-PC and PC/XT can be up to 640K, or address 9FFFFh. The IBM-PC/AT can address up to 1 MB of RAM.

Video Display Memory. The video display is *memory-mapped*. Rather than having to send each video character out through a port to the video display, the engineers at IBM decided that it would be more efficient to give each screen position a separate memory address. When DOS writes a character to the display, it calls a subroutine in the ROM BIOS, which in turn writes the character directly to a video memory address.

Video RAM memory (128K) extends from A0000h to BFFFFh, depending on the type of display used. The monochrome display uses only 4K of memory (B0000h to B7FFFh), the color graphics adapter (CGA) uses 16K of memory (B8000h to BBFFFh), and the enhanced graphics adapter (EGA) uses 128K. Many programs write characters directly to the video display buffer. A character written to the monochrome display area will not appear in the color display area because they are at different memory addresses. Programs that write directly to video memory must check the display type first.

ROM Area. Locations C0000h to FFFFFh are reserved by IBM for specialized ROM uses, including the hard disk controller and ROM BASIC. The latter is usually not included on IBM-compatible computers.

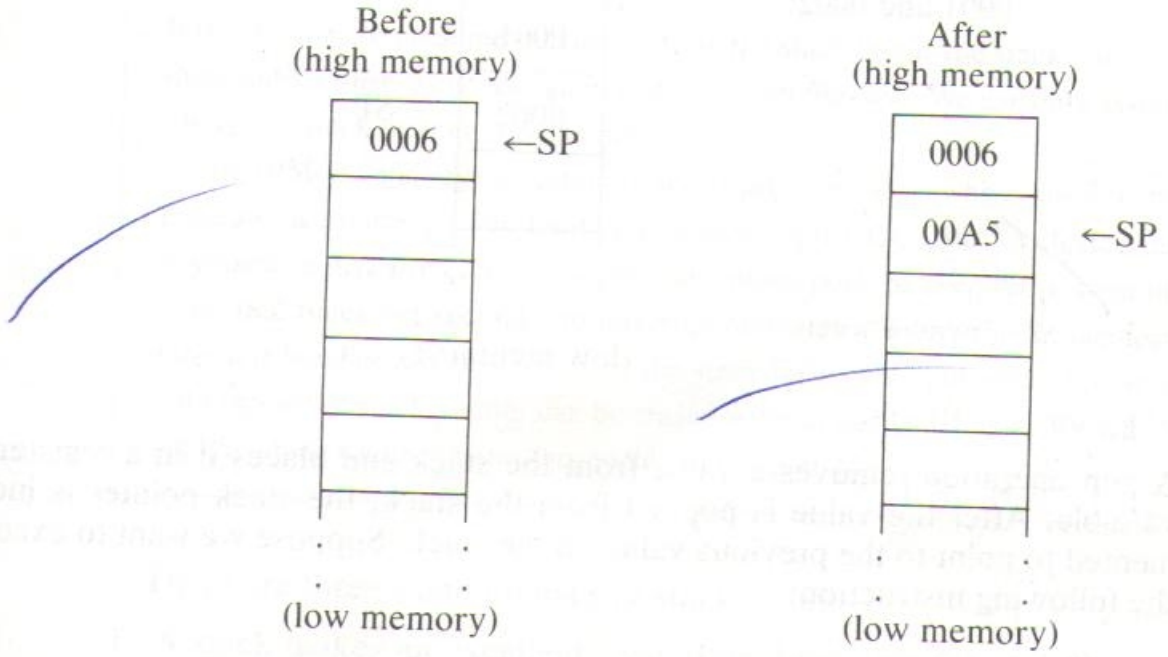
Finally, the ROM BIOS resides in locations F0000h to FFFFFh, the highest area of memory. The BIOS contains low-level subroutines used by DOS for input-output and other basic functions. Programs coded in ROM are often called *firmware,* because they are software stored in a hardware medium.

THE STACK

The *stack* is a special memory buffer used as a holding area for addresses and data. The stack resides in the stack segment. Each 16-bit location on the stack

is pointed to by the SP register, called the *stack pointer*. The stack pointer holds the address of the last data element to be added to, or *pushed* on the stack. The last value added to a stack is also the first one to be removed, or *popped* from the stack, so we call it a *LIFO structure* ("*L*ast *I*n *F*irst *O*ut").

Let's look at a program stack containing one value, 0006, on the left side of the following illustration. The stack pointer (notated by SP) points to the most recently added value:



When we push a new value on the stack, as shown on the right side of this illustration, SP is decremented before the new value is pushed (SP always points to the last value pushed). We use the PUSH instruction to accomplish this, as shown by the following code:

```
mov   ax,00A5  ; move 00A5h to AX
push  ax       ; push AX on the stack
```

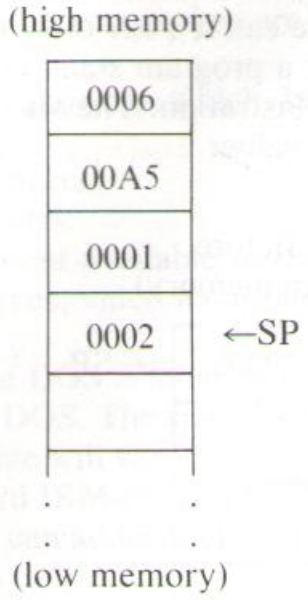
The PUSH instruction does not change the contents of AX; instead, it *copies* the contents of AX onto the stack.

As more values are pushed, the stack grows downward in memory. Let's assume that the BX and CX registers contain the values 0001 and 0002. The following instructions push them on the stack:

```
push  bx   ; push BX on the stack
push  cx   ; push CX on the stack
```

Now that 0001 and 0002 have been pushed on the stack, it appears as follows:

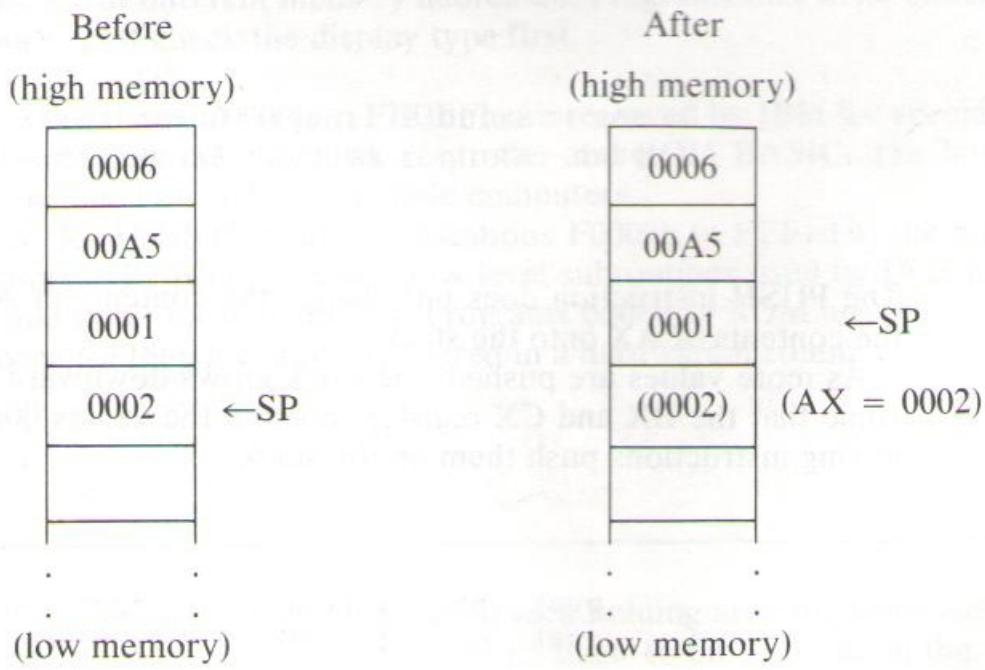
New contents of the stack after pushing 0001 and 0002:



A pop operation removes a value from the stack and places it in a register or variable. After the value is popped from the stack, the stack pointer is incremented to point to the previous value on the stack. Suppose we want to execute the following instruction:

```
pop  ax   ; pop the top of stack into AX
```

We can see the stack before and after the instruction is executed in the following illustration:



follows:

register or
er is incre-
to execute

e following

SP

0002)

After the POP has taken place, AX contains the number that was at the top of the stack (0002). The stack pointer has moved up and is now pointing at 0001. The popped value (0002) is no longer available, and the area of the stack below the stack pointer is available for future use.

Programming Tip: What Happens to the Stack?

It is very easy to presume that data left on the stack below the stack pointer will remain there until we use the stack again. But that's not the case: We can only assume that data *above* the stack pointer remains intact.

The IBM-PC employs a system of hardware *interrupts*, which literally interrupt any program in progress when hardware devices signal the CPU. A device might be the keyboard, a disk drive, a serial communications port, or even the system timer, which ticks 18.2 times per second. An interrupt activates a subroutine in memory called an *interrupt handler*, which freely uses the stack for its own purposes. So, at any instant, data below the stack pointer can be erased without our knowing it. We will cover interrupt handling routines more thoroughly in Chapter 16.

There are three standard uses of stacks:

1. A stack makes an excellent *temporary save area* for registers if we need to preserve their values. We can then use the registers as a scratch area and restore them when finished.
2. When a subroutine is called, the program saves a *return address* on the stack, the location in the program to which the subroutine is to return.
3. High-level languages create an area on the stack inside subroutines called the *stack frame*. It is in this area that local variables are created while the subroutine is active. They are then discarded when the subroutine returns to the calling program.

ADDRESS CALCULATION

An *address* is a number that refers to an 8-bit memory location. Addresses are numbered consecutively starting at 0, going up to the highest location in memory. Addresses are expressed in one of two hexadecimal formats:

- A 32-bit *segment-offset* address, which combines a base location (segment) with an offset to represent an actual location. An example is 08F1:0100.
- A 20-bit *absolute* address, which refers to an exact memory location. An example is 09010.

3

Assembly Language Fundamentals

Data Definition Directives
- Define Byte (DB)
- Define Word (DW)
- Define Doubleword (DD)
- The LABEL Directive

Program Structure
- The MODEL Directive
- Program Segments
- Procedures
- MASM Versions 1.0 Through 4.0
- COM Format Programs

Data Transfer Instructions
- The MOV Instruction
- Sample Program: Move a List of Numbers
- The XCHG Instruction
- Exchanging Memory Operands

Arithmetic Instructions
- The INC and DEC Instructions
- The ADD Instruction
- The SUB Instruction
- Flags Affected by ADD and SUB

Addressing Modes
- Types of Operands
- Summing a List of Numbers

Points to Remember

Review Questions

Programming Exercises

Answers to Review Questions

DATA DEFINITION DIRECTIVES

In assembly language, we define storage for variables using *data definition* directives. (The term *data declaration* is sometimes used to mean the same thing.) Data definition directives create storage at assembly time and can even initialize a variable to a starting value. The directives are summarized in the following table:

Directive	Description	Number of Bytes	Attribute
DB	Define byte	1	byte
DW	Define word	2	word
DD	Define doubleword	4	doubleword
DQ	Define quadword	8	quadword
DT	Define 10 bytes	10	tenbyte

As we see from this table, the variable being defined is given a particular *attribute*. The attribute refers to the basic unit of storage used when the variable was defined.

These directives also assign names to variables. For example, the following DB directive creates an 8-bit variable called **char**:

```
char  db 'A'
```

The assembler initializes the variable to a starting value, which in this example is 41h, the ASCII code for the letter *A*. The name of the variable is optional, but we normally supply it in order to be able to refer to the variable in an instruction.

Define Byte (DB)

The DB directive creates (allocates) storage for a byte or group of bytes, and optionally assigns starting values. The syntax is:

```
[name] DB initialvalue [,initialvalue] . . .
```

Initialvalue can be one or more 8-bit numeric values, a string constant, a constant expression, or a question mark (?). Commas separate the values when multiple bytes are defined:

```
list  db  10h,20h,41h,2
```

Let us assume that the variable **list** is stored at location 00h, as shown by the following illustration. Each number begins at the offset immediately following the previous one:

Offset:	00	01	02	03
Values:	10	20	41	02

This brings up a very important point: The name **list** identifies the *offset*, or location, of only the first byte in the list of numbers. The other bytes defined by **list** follow at the next few memory locations. Each constant may use a different radix when a list of items is defined:

```
list  db  10d,14h,41h,00000010b
```

Numeric, character, and string constants may be freely mixed:

```
db  10,'A',20h,'ABC'
```

Each storage byte for this data is shown here in hexadecimal:

Values:	0A	41	20	41	42	43

Memory contents may be left undefined by using the question mark (?) operator:

```
count   db   ?
```

A numeric expression can initialize a variable with a value that is calculated at assembly time:

```
count   db  10*20
```

The *DUP* operator repeats a single- or multiple-byte value. For example, 20 bytes containing all binary zeros would be coded as:

```
db   20 dup(0)
```

At the same time, an array containing 20 occurrences of the string 'stack' would be:

```
db   20 dup('stack')
```

This would allocate 100 bytes of storage: 'stack' is 5 bytes long and it occurs 20 times.

When a hexadecimal number begins with a letter (*A–F*), a leading zero is added to prevent the assembler from interpreting it as a label. Examples:

```
db  A6h     ; incorrect
db  0A6h    ; correct
```

Define Word (DW)

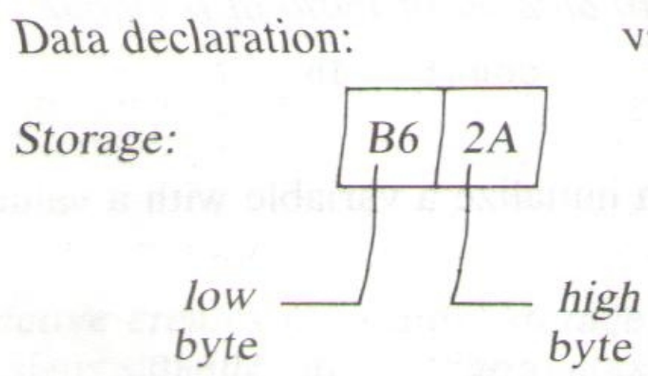
The DW directive creates storage for a word or list of words and optionally gives them starting values. The syntax is:

```
[name] DW initialvalue [,initialvalue] ...
```

Initialvalue can be any 16-bit numeric value up to 0FFFFh (65535), a constant expression, or a question mark (?). If *initialvalue* is signed, the acceptable range is *-32768 to +32767*.

A character constant may be stored in the lower half of a word. The largest string constant that may be stored in a word is 2 characters long, such as: 'ab'.

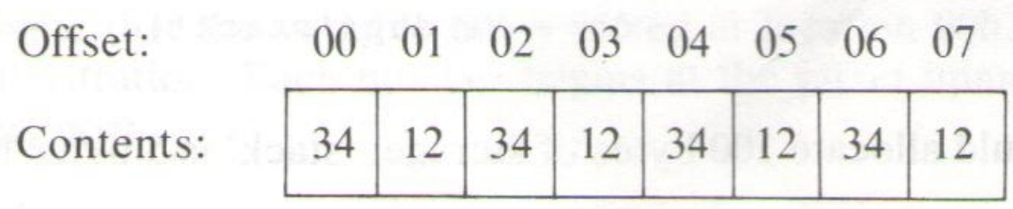
Reversed Storage Format. The assembler reverses the bytes in a word value when storing it in memory; the lowest byte occurs at the lowest address. When the variable is moved to a 16-bit register, the CPU re-reverses the bytes. This is shown in the following illustration, where 2AB6h is stored in memory as B6 2A:



DUP operator. The DUP operator allocates multiple occurrences of a value. For example, the following DW directive creates a list of four 16-bit integers:

```
intarray  dw  4 dup(1234h)
```

As with all 16-bit values, each number is stored in reversed-byte format. Assuming that **intarray** starts at offset 00, we might picture the storage of bytes thus:



Additional examples using the DW directive are as follows:

```
dw   0,0,0                ; define 3 words of storage
dw   0,65535              ; lowest and highest unsigned values
dw   -32768,+32767        ; lowest and highest signed values
dw   256*2                ; constant expression = 512
dw   4000h                ; hexadecimal value
dw   1111000011110000b    ; binary value
dw   1000h,4096,'AB',0    ; mixed data types
dw   ?                    ; single uninitialized word
dw   100 dup(?)           ; 100 uninitialized words
```

In the last two examples, the question mark (?) tells the assembler not to initialize the memory location to any value. Thus, the contents are undetermined at runtime.

Define Doubleword (DD)

The DD directive creates storage for a 32-bit doubleword variable, with the option of giving it a starting value. The syntax is:

```
[name] DD initialvalue [,initialvalue] . . .
```

Initialvalue can be a binary number up to 0FFFFFFFFh, a segment-offset address, a 4-byte encoded real number, or a decimal real number.

The bytes in a doubleword variable are stored in reverse order, so the least significant digits are stored at the lowest address. For instance, the value 12345678h would be stored in memory as

Offset:	00	01	02	03
	78	56	34	12

You can define a single doubleword location or a list of them. In the first example that follows, **far_pointer1** is uninitialized. In the second example, the assembler automatically initializes **far_pointer2** to the 32-bit segment-offset address of **subroutine1**. In the third example, **list32** marks the starting location of 20 doubleword memory locations:

```
far_pointer1   dd   ?
far_pointer2   dd   subroutine1
list32         dd   20 dup(0)
```

The LABEL Directive

The LABEL directive assigns a specific attribute (e.g., byte, word, doubleword) to a label. If LABEL is used in the code segment, the syntax is:

```
name LABEL distance
```

where *distance* may be either NEAR or FAR. This mainly affects the way a subroutine is called: from within the same segment (NEAR) or from another segment (FAR). We will delay a complete discussion of this until Chapter 7.

This type checking is usually a good idea because it helps us to avoid logic errors. If you need to disable it, you can use the LABEL directive to create a new name *at the same address*. The variable may now be accessed using either name:

```
mov   al,count_low      ; retrieve low byte of count
mov   cx,count          ; retrieve all of count
.
.
count_low label byte    ; byte attribute
count dw  20h           ; word attribute
```

PROGRAM STRUCTURE

In this section, we present an overview of the basic structure of an assembly language program. A program is divided into separate *segments*. Each segment contains instructions or data whose addresses are relative to a segment register (CS, DS, SS, or ES). Beginning with Version 5.0, MASM provides a simplified set of directives for declaring segments, called *simplified segment directives*. (We will deal here only with the simplified directives and will cover complete segment definitions in Chapter 14.)

Figure 3-1 shows the basic program structure to be used in most programs in this book. The optional TITLE directive defines a program title up to 128 characters long, which will be printed on the program listing. The DOSSEG directive

Figure 3-1 Structure of an assembly language program.

```
title  Sample Program Structure

dosseg                  ; set up segment order
.model small            ; small memory model
.stack 100h             ; set stack size

.code
main    proc
        mov  ax,bx      ; code (instructions)
        .
        .
main    endp

.data
count   db  10          ; data (variables)
        .
        .
end     main            ; end of assembly
```

tells MASM to place the program segments in a standard order used by Microsoft high-level languages.

The MODEL Directive

The MODEL directive selects a standard memory model for your program. A *memory model* is a lot like a standard blueprint or configuration. It determines the way segments are linked together, as well as the maximum size of each segment.

The memory models are standard for Microsoft and Borland languages and are common to compilers written by many other companies. The models are defined by the number of bytes that may be used for code (instructions) and data (variables). When we limit code to 64K, for example, we indicate that all instructions must fit within a single 64K memory segment. The following table summarizes the differences between the types of models:

Model	Description
Tiny	Code and data together may not be greater than 64K
Small	Neither code nor data may be greater than 64K
Medium	Only the code may be greater than 64K
Compact	Only the data may be greater than 64K
Large	Both code and data may be greater than 64K
Huge	All available memory may be used for code and data

All of the program models except *tiny* result in the creation of .EXE programs. The *tiny* model creates a .COM program. Unfortunately, simplified segment directives may not be used with this model, and the CODEVIEW debugger does not work well with .COM programs. In this book, we will use the *small* memory model, as it is the smallest one that is fully supported by Microsoft. A complete discussion of .COM programs may be found in Chapter 14.

Program Segments

Three segment directives are normally used: .STACK, .CODE, and .DATA. They may be placed *in any order* within a program, since the DOSSEG directive determines their final order anyway. The .STACK directive sets aside stack space for use by the program. For most small programs, 256 bytes is a more than adequate amount of space. The .CODE directive identifies the start of code (instructions) in a program. Program execution always begins with the first instruction in this section. The .DATA directive identifies the start of the data segment, where variables are declared.

By using the DOSSEG, MODEL, .STACK, .CODE, and .DATA directives, we ensure that a program can be called as a subroutine from a Microsoft high-level language without any conflicts arising from segment definitions.

Procedures

A *procedure* is a group of related program instructions with a common function or purpose. A procedure is given a name that identifies its starting location. We use the PROC directive to identify a procedure. For example, the label **main_routine** defines the starting location of a procedure:

```
main_routine proc
```

Procedures are often called *routines* or *subroutines*. Every program has at least one procedure, and additional procedures may be added as a program expands.

The PROC directive shows where a procedure begins and the ENDP directive shows where it ends. If other procedures were to be added to the program in Figure 3-1, they would fall after the ENDP directive for **main** and before the .DATA directive:

```
.code
main    proc
        .
        .
main    endp

sub1    proc
        .
        .
sub1    endp

sub2    proc
        .
        .
sub2    endp
```

The END Directive. The END directive marks the last line in a source program to be assembled. Any lines of text placed after the END directive are ignored. In the main module of a program, the END directive is accompanied by a label that identifies the *entry point,* or start, of program execution. In Figure 3-1, the program entry point is **main**, the name of the only procedure.

MASM Versions 1.0 Through 4.0

Earlier versions of the Microsoft Assembler required you to declare the code, data, and stack segments yourself. Figure 3-2 shows the basic structure of such a program. First, it does not include the DOSSEG, .MODEL, .CODE, and .STACK directives. Instead, the segments' order is determined by their physical placement in the source program.

grams. Programs assembled using DEBUG, for example, are in COM format. There has been a gradual shift away from using the COM format with the advent of the OS/2 operating system and other multitasking software, because a COM program does not share memory very well with other programs. Also, COM programs cannot use simplified segment directives or be debugged in source mode using Microsoft's CODEVIEW debugger. All these disadvantages aside, COM programs are easy to write and fine for small stand-alone applications.

Figure 3-3 (page 57) shows a COM program shell, which is identical for all versions of MASM. The only segment is called **code**, and the ASSUME directive has been adjusted accordingly. There is no stack segment because a stack area is automatically created at the end of the program segment. Variables may be located anywhere in the program segment, as long as you don't allow the CPU to accidentally process variables as instructions. I usually place variables right after the **main** procedure. A COM program must include the ORG 100h directive, which sets the instruction pointer (IP) to 100h when the program is loaded.

DATA TRANSFER INSTRUCTIONS

The MOV Instruction

The MOV instruction copies data from one operand to another using either 8-bit or 16-bit operands. MOV is called a *data transfer* instruction. The syntax is:

```
MOV  destination,source
```

Data is merely *copied* from the source to the destination, so the source operand is not changed. The *source* operand may be immediate data, a register, or a memory operand. The *destination* operand may be a register or a memory operand. The following types of data transfers are possible:

immediate data	to	register or memory
register	to	register
register	to	memory
memory	to	register

There are a few limitations on the types of operands:

- CS and IP may never be destination operands.
- Immediate data and segment registers may not be moved to segment registers.
- The source and destination operands must be the same size.
- If the source is immediate data, it must not exceed 255 (FFh) for an 8-bit destination or 65,535 (FFFFh) for a 16-bit destination.

MOV does not allow memory-to-memory transfers, so data must be transferred to a register first. For example, to copy **var1** into **var2**, one would have to write

```
mov  ax,var1
mov  var2,ax
```

Register Operands. A move involving only registers is the fastest type, taking only two clock cycles. (A *clock cycle* is the CPU's smallest unit of time measurement; it varies from 50 to 210 nanoseconds, depending on the type of CPU being used.) Registers should be used when an instruction must execute quickly.

Any register may be used as a source operand, and any registers except CS and IP may be destination operands. Some examples using register operands are as follows:

```
mov   ax,bx
mov   dl,al
mov   bx,cs
```

Immediate Operands. An immediate value (integer constant) may be moved to any register except a segment register or IP, and may be moved to any memory operand. A common error is to make the immediate value larger than the destination operand. Examples of valid immediate addressing are:

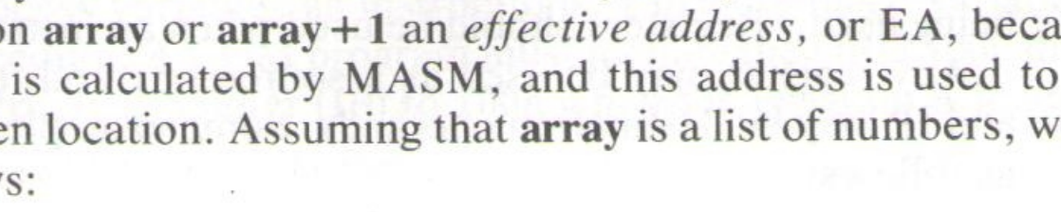
```
mov bl,01          ; move 8-bit value to BL
mov bx,01          ; move 16-bit value to BX
mov bx,1000h       ; move 16-bit value to BX
mov total,1000h    ; move 16-bit value to a variable
```

Direct Addressing. The name of a variable may be coded as one of the operands in a MOV instruction. This causes the contents of memory at the variable's address to be used. Let's say, for example, that an 8-bit variable named **count** contains the number 1. The following MOV statement copies the contents of **count** into AL:

```
mov  al,count
```

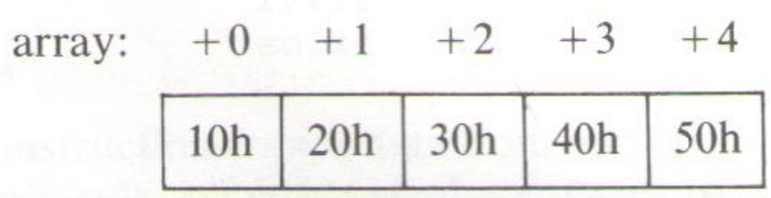
On the other hand, you might want to copy the BL register into the variable **count**:

```
mov  count,bl
```



You can also add an offset to the name of a variable when using direct addressing. The name **array+1** refers to the location 1 byte beyond the offset of **array**. We call the notation **array** or **array+1** an *effective address*, or EA, because the operand's address is calculated by MASM, and this address is used to access memory at the given location. Assuming that **array** is a list of numbers, we might picture it as follows:

```
array  db 10h,20h,30h,40h,50h
```



Therefore, memory location **array+1** contains 20h. The following statements show different ways of accessing the array:

```
mov   al,array      ; contents = 10h
mov   dl,array+1    ; contents = 20h
mov   bh,array+4    ; contents = 50h
mov   array+2,0     ; third element = 0
```

As with register and immediate addressing, the *size attributes* (byte or word) of the two operands must match. For example, a variable named **int_1** created using the DW directive could only be paired with a 16-bit register. The variable **byte_1**, created by using DB, could only be paired with an 8-bit register. Both examples follow:

```
        mov   ax,int_1      ; copy 16 bits
        mov   int_1,si      ; copy 16 bits
        mov   al,byte_1     ; copy 8 bits
        mov   byte_1,dl     ; copy 8 bits
        .
        .
int_1   dw    1000h         ; 16-bit value
byte_1  db    10h           ; 8-bit value
```

PTR Operator. Occasionally you need to clarify an operand's type with the PTR operator. The name PTR incorrectly suggests the use of a pointer, but it really identifies the *attribute* of the memory operand. In the following example, WORD PTR identifies **count** as a word-sized variable, while BYTE PTR identifies **var2** as an 8-bit operand:

```
mov   word ptr count,10
mov   byte ptr var2,5
```

Illegal Moves. Certain combinations of operands are not valid with the MOV instruction:

1. Moves between two memory operands.
2. Moves of segment registers or immediate values to segment registers.
3. Moves to CS and IP.
4. Moves between registers of different sizes.
5. Moves between operands of different sizes unless the PTR operator overrides the default type of a memory operand.
6. Moves to immediate operands (this is probably obvious).

Samples of each illegal move are as follows:

```
mov     word_1,word_2       ; memory to memory
mov     ds,1000h            ; immediate to segment
mov     ip,ax               ; IP destination
mov     al,bx               ; mismatching register types
mov     word_1,al           ; mismatching operand types
mov     1000h,ax            ; immediate destination
.
.
.
word_1      dw    1000h
word_2      dw    0
```

Example. The following program moves immediate values to a register and then transfers values between registers. To get the maximum benefit from the example, assemble and trace it using DEBUG. The line numbers on the left side are not part of the program:

```
                                ;  AX   BX   CX   DX
1:    mov   ax,2B10             ; 2B10 0000 0000 0000
2:    mov   bx,ax               ; 2B10 2B10 0000 0000
3:    mov   dl,ah               ; 2B10 2B10 0000 002B
4:    mov   cl,bl               ; 2B10 2B10 0010 002B
5:    int   20                  ; end program
```

ARITHMETIC INSTRUCTIONS

Hardly any computer program could avoid performing arithmetic. The Intel instruction set has instructions for integer arithmetic, using 8-bit and 16-bit operands. Floating-point operations are handled by the 8087 Math Coprocessor.

In this section, we look at the most basic arithmetic instructions. The INC and DEC instructions add 1 to or subtract 1 from an operand; the ADD and SUB instructions perform 8-bit and 16-bit addition and subtraction.

The INC and DEC Instructions

The INC and DEC instructions add 1 to or subtract 1 from a single operand, respectively. Their syntax is:

```
INC  destination
DEC  destination
```

The destination operand may be an 8- or 16-bit register or memory variable. INC and DEC are faster than the ADD and SUB instructions, so they should be used where practical. All status flags are affected except the Carry flag. Examples are as follows:

```
inc al                  ; increment 8-bit register
dec bx                  ; decrement 16-bit register
inc membyte             ; increment memory operand
dec byte ptr membyte    ; increment 8-bit memory operand
dec memword             ; decrement 16-bit memory operand
inc word ptr memword    ; increment 16-bit memory operand
```

In these examples, the BYTE PTR operator identifies an 8-bit operand and WORD PTR identifies a 16-bit operand.

The ADD Instruction

The ADD instruction adds an 8- or 16-bit source operand to a destination operand of the same size. The syntax is:

```
ADD  destination,source
```

The source operand is unchanged by the operation. The sizes of the operands must match, and only one memory operand may be used. A segment register

may not be the destination. All status flags are affected. Examples are as follows:

```
add al,1          ; add immediate value to 8-bit register
add cl,al         ; add 8-bit register to register
add bx,1000h      ; add immediate value to 16-bit register
add var1,ax       ; add 16-bit register to memory
add dx,var1       ; add 16-bit memory to register
add var1,10       ; add immediate value to memory
```

The SUB Instruction

The SUB instruction subtracts a source operand from a destination operand. The syntax for SUB is:

```
SUB  destination,source
```

The sizes of the two operands must match, and only one may be a memory operand. A segment register may not be the destination operand. All status flags are affected. Examples are as follows:

```
sub al,1          ; subtract immediate value from 8-bit register
sub cl,al         ; subtract 8-bit register from register
sub bx,1000h      ; subtract immediate value from 16-bit register
sub var1,ax       ; subtract 16-bit register from memory
sub dx,var1       ; subtract 16-bit memory from register
sub var1,10       ; subtract immediate value from memory
```

Flags Affected by ADD and SUB

If the result of an addition operation is too large for the destination operand, the Carry flag is set. For example, the following result (300) is too large to fit into AL:

```
mov  al,250
add  al,50    ; Carry flag set
```

If the source is larger than the destination, a subtraction operation requires a borrow. This sets the Carry flag, as in the following example:

```
mov  al,5
sub  al,10    ; Carry flag set
```

01h: Console Input with Echo

DOS function 1 waits for a character to be input from the console and stores the character in AL. CTRL-BREAK is active. The character is *echoed*, meaning that it is redisplayed on the console as soon as it is input. In the following example, a single character is input and placed in a variable named **char**:

```
mov  ah,1      ; console input function
int  21h       ; call DOS, key returned in AL
mov  char,al   ; save the character
```

Any character currently waiting in the keyboard typeahead buffer is automatically returned in AL. Otherwise DOS waits for a character to be input. The *typeahead* buffer is a 15-character circular buffer used by DOS to store keystrokes as they are pressed. This makes it possible for you to type faster than a program is able to act on the input: DOS will "remember" the keystrokes. If too many characters are backed up in the buffer, the computer beeps and extra keystrokes are ignored.

02h: Character Output

DOS function 2 sends a character to the console. CTRL-BREAK is active. You must place the character to be displayed in DL, as in the following example:

```
mov  ah,2     ; select DOS function 2
mov  dl,'*'   ; character to be displayed
int  21h      ; call DOS to do the job
```

AL may be modified by DOS during the call to INT 21h, so be sure to replace its contents after DOS finishes.

```
mov  byteval,al   ; save contents of AL
mov  ah,2
mov  dl,'*'       ; character to be displayed
int  21h          ; (DOS changes AL)
mov  al,byteval   ; restore original AL
```

05h: Printer Output

To print a character, place it in DL and call function 5. DOS waits until the printer is ready to accept the character. If necessary, you can terminate the wait by pressing CTRL-BREAK. The default output is to printer 1 (device name LPT1). You may also have to send a carriage-return character (0Dh) to force immediate printing. Many printers keep each character in an internal buffer until

either the buffer is full or a carriage return is printed. The following program excerpt prints a dollar-sign ($) character:

```
mov  ah,5     ; select printer output
mov  dl,'$'   ; character to be printed
int  21h      ; call DOS
mov  dl,0Dh   ; print a carriage return
int  21h      ; call DOS
```

It is not necessary to reload AH with 5 before calling INT 21h the second time. This is true for INT 21h functions in general.

06h: Direct Console Input-Output

DOS function 6 either reads from or writes to the console. CTRL-BREAK is not active. In fact, any control characters may be read or written without being acted on, such as the carriage return, tab, and so on. The section entitled "ASCII Control Characters" later in this chapter provides many examples.

To request console input, DL must equal 0FFh. DOS does not wait for a character to be pressed, but will return a character in AL if one is waiting in the keyboard input buffer. If no character is available, the Zero flag is set.

To request console output, DL should equal the character to be displayed (other than 0FFh). Examples of both input and output are as follows:

Character input:

```
mov  ah,6     ; request DOS function 6
mov  dl,0FFh  ; look for input, don't wait
int  21h      ; if key pressed, AL=character,
              ; otherwise the Zero flag is set
```

Character output:

```
mov  ah,6     ; request DOS function 6
mov  dl,'&'   ; character to be output
int  21h      ; call DOS
```

07h: Direct Console Input

Function 7 waits for a character from the console. The character is not echoed, and CTRL-BREAK is inactive. This input function is well suited to special keyboard characters such as function keys and cursor arrows:

```
mov  ah,7     ; console input function
int  21h      ; call DOS
mov  char,al  ; save the character
```

08h: Console Input Without Echo

Function 8 waits for a character from the console without echoing it, and CTRL-BREAK is active. The character is returned in AL. This input function is also

appropriate for special keyboard characters:

```
mov  ah,7      ; console input function
int  21h       ; call DOS
mov  char,al   ; save the character
```

09h: String Output

Function 9 displays a character string on the console. The offset address of the string must be in DX, and the string must be terminated by the dollar-sign ($) character. Control characters such as tabs and carriage returns are recognized by DOS. In the following example, a string is displayed, followed by the carriage return (0Dh) and line feed (0Ah) characters:

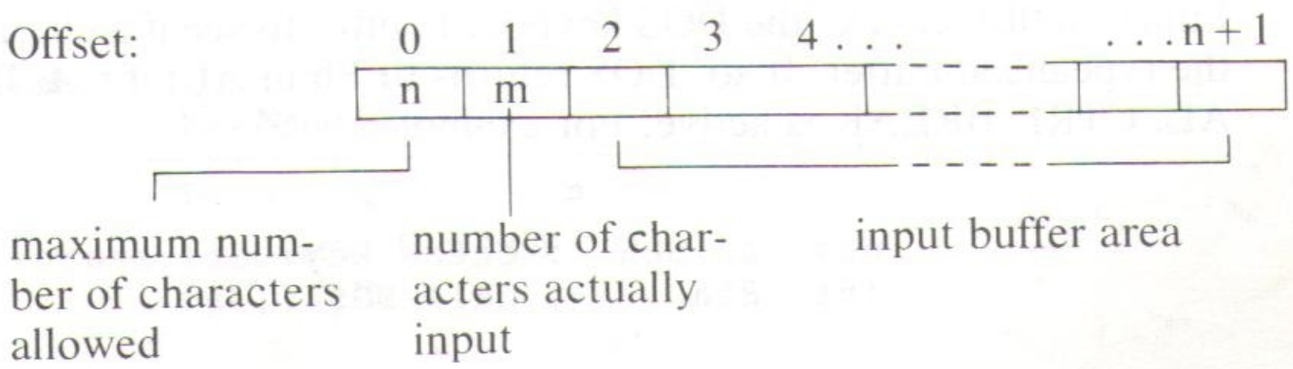
```
        mov  ah,9                ; string output function
        mov  dx,offset string    ; offset address of the string
        int  21h
        .
        .
string  db 'This is a byte string.',0Dh,0Ah,'$'
```

This DOS function has an obvious disadvantage, because a dollar-sign character cannot be displayed as part of a string. A curious thing happens when the dollar sign is omitted; DOS simply displays all subsequent characters in memory until the ASCII value for a dollar sign (24h) is found. Several hundred characters might be displayed before this happens.

0Ah: Buffered Console Input

Function 0Ah reads a character string of up to 255 characters from the console and stores it in a buffer. The backspace key may be used to erase characters and back up the cursor. You terminate the input by pressing ENTER. DOS filters out any extended keys, such as cursor arrows, PgDn, and so on, so they will not be stored in the buffer. CTRL-BREAK is active, and all characters are echoed on the console.

Before the function is called, DX must contain the offset of the keyboard parameter area. The format of this area is:



In the byte at offset 0, place a count of the maximum number of characters you wish to be input. If you choose a count of 5, for example, DOS will permit only 4 characters plus the Enter key to be input. After the interrupt is called, DOS places the number of keys *actually typed* in the byte at offset 1. This count doesn't include the Enter key. The characters themselves are placed by DOS in the buffer, beginning at offset 2.

In the following example, **max_keys** equals 32, **chars_input** is filled in by DOS, and **buffer** holds the input characters:

```
    mov ah,0Ah              ; select console input
    mov dx,offset maxkeys   ; DX points to keyboard parameter area
    int 21h                 ; call DOS
    .
    .
max_keys db 32              ; max characters allowed
chars_input db ?            ; characters actually input
buffer  db 32 dup(0)        ; holds the input
```

Let's assume that we have used these statements in a program, and then we input the following 21 characters from the console:

```
My name is Kip Irvine
```

A dump of the buffer after the characters are input shows that the byte at offset 0001 contains the number of characters actually input (15h), followed by the input characters. (The Enter key appears in the buffer as 0Dh, and the count [15h] does not include the extra key.)

```
max_keys  chars_input
   ↓       ↙
  20 15
  4D 79 20 6E 61 6D 65 20 69 73 20    .My name is     ]   buffer
  4B 69 70 20 49 72 76 69 6E 65 0D    Kip Irvine.     ]
                                  ↑
                              Enter key
```

0Bh: Get Console Input Status

Function 0Bh checks the DOS keyboard buffer to see if a character is waiting in the typeahead buffer. If so, DOS returns 0FFh in AL; if not, DOS returns 00 in AL. CTRL-BREAK is active. For example:

```
    mov  ah,0Bh  ; check keyboard status
    int  21h     ; call DOS
```

6

Loops and Comparisons

More on Addressing Modes
- Direct Addressing Mode
- Indirect Addressing Mode
- The LEA Instruction
- Notes on Using PTR

Transfer-of-Control Instructions
- The JMP Instruction
- The LOOP Instruction

Boolean and Comparison Instructions
- The AND Instruction
- The OR Instruction
- The XOR Instruction
- The NOT Instruction
- The NEG Instruction
- The TEST Instruction
- The CMP Instruction

Conditional Loops
- The LOOPZ (LOOPE) Instruction
- The LOOPNZ (LOOPNE) Instruction

Points to Remember
Review Questions
Programming Exercises
Answers to Review Questions

All the programs we have looked at so far lack one important characteristic: the ability to *loop*, or repeat a block of statements. This chapter introduces two instructions used for looping and transfer of control: JMP (jump) and LOOP. In the second half of the chapter, we will concentrate on boolean and comparison instructions. These give you the ability to manipulate individual bits and to test their values. They are introduced as preparation for the complete coverage of conditional jump instructions in Chapter 7.

In assembler, a *jump* is the same thing as a GO TO statement in a high-level language. A new address is moved to the IP (instruction pointer) register, which causes the CPU to begin executing instructions at the new location. JMP is called an *unconditional* transfer instruction because no test or condition is necessary; the jump always takes place.

The LOOP instruction is also a type of jump instruction that decrements and tests a counter. Therefore, LOOP is a *conditional transfer* instruction.

MORE ON ADDRESSING MODES

In Chapter 3 we looked briefly at all of the addressing modes, but at that time we had seen very few programs. Now it's time to take a closer look at the mem-

ory addressing modes. The MASM *Programmer's Guide* contains an excellent description of a direct memory operand:

> A direct memory operand is a symbol that represents the address (segment and offset) of an instruction or data. The offset address represented by a direct memory operand is calculated at assembly time. The address of each operand relative to the start of the program is calculated at link time. The actual (or effective) address is calculated at load time.*

This statement also tells us something about the assembly, link, and execution steps. The offset address of an operand is calculated during the assembly step and displayed in the listing file. The following instruction, for example, moves **count** to DL:

```
8A 16 0005 R    mov  dl,count
```

The R shown in the listing file identifies this as a *relocatable* operand, and 0005 is the operand's offset from the beginning of the data segment.

During the link step, segments are grouped together and placed in their correct order. The offsets of variables are recalculated as offsets from the start of a group of segments.

When a program is loaded into memory, DOS determines the absolute addresses of variables. The program's load address is always the lowest memory location that is currently not in use by another program.

A memory operand may be the name of a procedure, such as **main**, it may be the offset of a variable, as in

```
mov   bx,offset count
```

or it may refer to the contents of a variable:

```
mov   dl,count
```

There are several basic elements that can be combined to create a memory operand:

Base register	BX, BP
Index register	SI, DI
Displacement	The name of a label or variable, which represents an offset from the beginning of a segment
Direct operand	The contents of memory at a location identified by the operand's name
Indirect operand	An address stored in a register or variable that is used to locate another variable

*From the MASM 5.0 *Programmer's Guide*. Used by permission of Microsoft Corporation.

Much of the power of IBM-PC assembly language lies in its flexibility when addressing data. The Intel instruction set offers a wide variety of addressing modes that help to make programming easier. Chapter 3 introduced operands constructed from registers, immediate data, and memory variables. Let us add to what we know about direct and indirect addressing.

Direct Addressing Mode

The MASM documentation distinguishes between two types of operands used in direct addressing: direct-memory operands and relocatable operands. A *direct-memory* operand combines a segment value with an offset that represents an absolute memory address at runtime. A *relocatable* operand is any label or symbol identifying a 16-bit displacement from a segment register. The syntax for creating a direct-memory operand is:

segment:offset

Segment refers to either a segment register or a segment name. Its value is unknown at assembly time, because it depends on where DOS will eventually load the program. *Offset* may be an integer, symbol, label, or variable. Examples are:

```
mov  ax,ds:5        ; segment register and offset
mov  bx,cseg:2Ch    ; segment name and offset
mov  ax,es:count    ; segment register and variable
```

Relocatable operands are more common. Their location depends on the offset of a label from the beginning of a segment. Depending on which segment the label is located in, the following segment registers are used by default:

Type of Label	Default Segment Register
Program code (instructions)	CS
Variables (data)	DS

The ES register usually addresses variables, but it can contain the base location of any segment.

Indirect Addressing Mode

Indirect operands use registers to point to locations in memory. If a register is used in this way, we can change its value and access different memory locations at runtime. Two types of registers are used: *base* registers (BX, BP) and *index*

registers (SI, DI). BP is assumed to contain an offset from the stack segment. SI, DI, and BX contain offsets from DS, the data segment register.

There are five indirect addressing modes, identified by the types of operands used:

Addressing Mode	Example
Register indirect	[bx]
Based	table[bx]
Indexed	table[si]
Based indexed	[bx + si]
Based indexed with displacement	[bx + si + 2]

Register Indirect. Indirect operands are particularly powerful when processing lists or arrays, because a base or index register may be modified at runtime. In the following example, BX points to two different array elements:

```
        mov  bx,offset array  ; point to start of array
        mov  al,[bx]          ; get first element
        inc  bx               ; point to next
        mov  dl,[bx]          ; get second element
        .
        .
array   db   10h,20h,30h
```

The brackets around BX signify that we are referring to the *contents* of memory, using the address stored in BX.

In the following example, the three bytes in **array** are added together:

```
        mov  si,offset array  ; address of first byte
        mov  al,[si]          ; move the first byte to AL
        inc  si               ; point to next byte
        add  al,[si]          ; add second byte
        inc  si
        add  al,[si]          ; add third byte
        inc  si
        .
        .
array   db   10h,20h,30h
```

The LEA Instruction

The LEA (load effective address) instruction loads the offset of a variable into a register, as does the MOV . . . OFFSET instruction. The following instructions both place the same value in BX:

```
mov  bx,offset bytelist
lea  bx,bytelist
```

be used. A *transfer of control,* or *branch,* is a way of altering the order in which statements are executed. All programming languages contain statements to do this. We divide such statements into two categories:

Unconditional Transfer. The program branches to a new location in all cases; a new value is loaded into the instruction pointer, causing execution to continue at the new address. The JMP instruction is a good example.

Conditional Transfer. The program branches if a certain condition is true. Intel provides a wide range of conditional transfer instructions that may be combined to make up conditional logic structures. The CPU interprets true/false conditions based on the contents of the CX and Flags registers.

The JMP Instruction

The JMP instruction tells the CPU to begin execution at another location. The location must be identified by a label, which is translated by MASM into an address. If the jump is to a label in the current segment, the label's offset is loaded into the IP register. If the label is in another segment, the segment's address is also loaded into CS. The syntax is

```
JMP [{ SHORT    }]  destination
    [{ NEAR PTR }]
    [{ FAR PTR  }]
```

where *destination* is a label or 32-bit segment-offset address.

The JMP instruction is amazingly flexible. It can jump to a label in the current procedure, from one procedure to another, from one segment to another, completely out of the current program, or to any place in RAM or ROM.* Structured programming discourages such jumps, but they are occasionally necessary in systems programming applications. Examples of various jumps are shown here:

```
jmp L1                  ; NEAR:  dest. in current segment
jmp near ptr L1         ; NEAR:  dest. in current segment
jmp short nextval       ; SHORT: dest. within -128 to +127
                                 bytes
jmp far ptr error_rtn ; FAR:    dest. in another segment
```

*The Microsoft/IBM OS/2 operating system restricts a program's ability to jump outside the current memory partition. Also, the Intel 80386 processor offers hardware memory protection, further restricting such jumps.

```
Address  Object Code      Source Code
0100     B4 02       start: mov ah,2      ; start of loop
0102     B2 41              mov dl,'A'    ; display "A"
0104     CD 21              int 21h       ; call DOS
0106     EB F8              jmp start     ; jump back to start
0108     ...(etc.)
```

Before assembling an instruction, MASM increments its own *location counter,* which tells it the offset of the next instruction. When assembling the instruction at 0106h, for example, MASM has already set the location counter to 0108h.

The jump at 0106h is automatically assembled as a short jump, because the distance from the location counter to the label **start** is less than 128 bytes. Two object code bytes are generated for the JMP at location 0106h: EBh and F8h. The first byte (EBh) is the op code for a short jump instruction. The second byte (F8h, or −8) is a *displacement* that tells the CPU how far to jump. MASM calculates this by subtracting the location counter (0108h) from the offset of the destination (0100h). The resulting displacement (F8h) is assembled as part of the instruction:

```
op code → EB F8 ← displacement
```

The LOOP Instruction

The LOOP instruction is the easiest way to repeat a block of statements a specific number of times. CX is automatically used as a counter and is decremented each time the loop repeats. Its syntax is:

```
LOOP destination
```

First, the LOOP instruction subtracts 1 from CX. Then, if CX is greater than zero, control transfers to *destination*. The destination operand must be a short label, in the range −128 to +127 bytes from the current location.

In the following example, the loop repeats five times:

```
        mov   cx,5    ; CX is the loop counter
start:
        .
        .
        loop  start   ; jump to START
```

If CX is equal to zero after having been decremented, no jump takes place and control passes to the following LOOP instruction. The flags are not affected when CX is decremented, even if it equals zero.

In the following example, the loop repeats five times, adding 1 to AX each time. When the loop ends, AX = 5 and CX = 0.

```
        mov   ax,0    ; set AX to 0
        mov   cx,5    ; loop count

top:    inc   ax      ; add 1 to AX
        loop  top     ; repeat until CX = 0
```

Caution: If CX = 0 before the LOOP instruction is reached for the first time, CX becomes equal to FFFFh. The loop then repeats another 65,535 times!

Example: Repeat the Letter A. The following program excerpt prints the letter *A* on the screen 960 times, using the LOOP instruction. The count placed in CX represents 12 rows of the screen multiplied by 80 characters per row:

```
        mov   cx,12*80  ; set count to 960

next:
        mov   ah,2      ; function: display character
        mov   dl,'A'    ; display the letter A
        int   21h       ; call DOS
        loop  next      ; decrement CX and repeat
```

Indirect Addressing. The LOOP instruction is particularly powerful when used with an indirect operand. The following program excerpt copies 6 numbers from one array to another, using indirect addressing:

```
 1:          mov   si,offset array1  ; initialize SI and DI
 2:          mov   di,offset array2
 3:          mov   cx,6              ; initialize loop counter
 4:
 5:  move_byte:
 6:          mov   al,[si]           ; get a byte from array1
 7:          mov   [di],al           ; store it in array2
 8:          inc   si
 9:          inc   di
10:          loop  move_byte
11:          .
12:          .
13:  array1  db    10h,20h,30h,40h,50h,60h
14:  array2  db    6 dup(?)
```

On lines 1 and 2, the MOV instructions set SI and DI to the starting offsets of each array. Lines 6 and 7 copy a character from **array1** to **array2**. Lines 8 and 9

be computed correctly. This appears to be a contradiction, but it is not. Suppose, for example, that we add FFFFh to 3000h. The result is 2FFFh:

```
        mov   cx,3000h        ; starting value of CX
        add   cx,0FFFFh       ; add -1

result:     CX = 2FFFh    Flags: NV PL NZ CY
```

The *unsigned* sum of 3000h and 0FFFFh is 12FFFh, which overflows CX. The highest digit is truncated, the Carry flag is set, and the result is incorrect. If the operands are signed, however, the statement is really adding -1 to 3000h. The result, 2FFFh, is correct. The Carry flag is irrelevant.

Signed Values Out of Range. The Overflow flag (notated OF) tells us something about the result of a signed addition or subtraction operation. If OF = 1, the signed result may be too large to fit into the destination operand or the result's sign bit may have changed. The CPU examines the sign bit of the result operand before and after the operation. If the bit has changed value, it sets the Overflow flag (OF = 1).

Let us compare two examples in which 1 is added to +127. In the first example, we store the result in AX, so OF = 0; in the second example, the result changes the sign bit of AL, so OF = 1:

Instruction	Before	After	Flags
1. add AX,1	AX = 007F (+127)	AX = 0080 (+128)	NV PL NC
2. add AL,1	AL = 7F (+127)	AL = 80 (−128)	OV NG NC

Figure 8-3 presents a more detailed description of these operations. In Example 1 the result in AX is correct (+128), and neither the Overflow flag nor the Sign flag is set. In Example 2 the result in AL is −128, although we added +1 to +127. The Overflow flag is set, showing that the sign bit was changed, and the result is invalid. The Sign flag is also set because the number in AL is negative.

MULTIPLICATION AND DIVISION

There are instructions to perform integer multiplication and division on 8-bit and 16-bit numbers. All operands are assumed to be binary, so if decimal or binary-coded decimal operands are involved, you must make all adjustments. Floating-point operations are handled either by a separate Intel math coprocessor or by

Example 1

```
ADD AX, 1

        Sign bit (before)
             |
          00000000  01111111          AX = 007Fh    (+127)
      +                    1
      ------------------------
Result: 00000000  10000000          AX = 0080h    (+128)
             |
        Sign bit (after)
```

Example 2

```
ADD AL, 1

        Sign bit (before)
             |
          01111111          AL = 7Fh    (+127)
      +          1
      ---------------
Result: 10000000          AL = 80h    (-128)
             |
        Sign bit (after)

        (Overflow and Sign flags set)
```

Figure 8-3 *Examples of signed addition.*

software emulation supplied in a language library. (See Chapter 15 for a discussion of the Intel 8087 math coprocessor.)

The MUL (multiply) and DIV (divide) instructions are for unsigned binary numbers. The IMUL (integer multiply) and IDIV (integer divide) instructions are for signed binary numbers.

MUL and IMUL Instructions

The MUL and IMUL instructions multiply an 8-bit or 16-bit operand by AL or AX. If an 8-bit source operand is supplied, it is automatically multiplied by AL, and the result is stored in AX. If a 16-bit source operand is supplied, it is multiplied by AX, and the result is stored in DX and AX (the high 16 bits are in DX). The syntax formats are:

```
MUL   source
IMUL  source
```

ing example, the result of the 8-bit IMUL operation is −1, so AX equals 1111111111111111b. The Carry and Overflow flags are clear:

```
mov   al,1
mov   bl,-1
imul  bl          ; AX = 1111111111111111b
                  ; CF = 0, OF = 0
```

16-Bit Multiplication Example. The following instructions multiply 10 by −48 and store the product in a variable named **result**. The 32-bit result is FFFFFE20h:

```
          mov   ax,10d
          mov   cx,-48d
          imul  cx              ; DX:AX = FFFF:FE20h (-480d)
          mov   result,ax
          mov   result+2,dx
          .
          .
result    dw    0,0
```

DIV and IDIV Instructions

The DIV and IDIV instructions perform both 8- and 16-bit division, signed and unsigned. A single operand is supplied (register or memory operand), which is assumed to be the divisor. The syntax formats for DIV and IDIV are:

```
DIV   source
IDIV  source
```

If the divisor is 8 bits long, AX is the dividend, AL the quotient, and AH the remainder. If the divisor is 16 bits, DX:AX is the dividend, AX the quotient, and DX the remainder:

Dividend	/	Divisor	=	Quotient	Remainder
AX		Operand		AL	AH
DX:AX		Operand		AX	DX

Example 1. 8-bit division (83h / 2 = 40h, remainder 3):

```
mov   ax,0083h  ; dividend
mov   bl,2      ; divisor
div   bl        ; AL = 40h, AH = 03h
```

For 32-bit dividends, the CWD (convert word to doubleword) instruction sign extends AX into DX:

```
mov   ax,-5000    ; AX = EC78h
cwd               ; DX:AX = FFFFEC78h
mov   bx,256
idiv  bx          ; AX = FFEDh, (-19) quotient
                  ; DX = FF78h, (-136) remainder
```

Divide Overflow

When a division operation generates too large a result, a *divide overflow* condition results, which calls system interrupt 0. On many machines this hangs the computer system, requiring a cold start. An attempt to divide by 0 will also produce a divide overflow. High-level languages have built-in protection against this, of course, but the CPU itself does not perform any error checking before dividing.

One solution is to always break up a 32-bit dividend into two separate 16-bit operands. For example, if we divide 08010020h by 10h, we will generate a divide overflow, because the quotient (801002h) does not fit into AX. The division may be performed in two steps, as shown here:

```
1:          mov  ax,dividend+2    ; dividend, high
2:          cwd                   ; sign extend
3:          mov  cx,divisor
4:          div  cx               ; AX = quotient (high)
5:                                ; DX = remainder (high)
6:          mov  bx,ax            ; save quotient, high
7:          mov  ax,dividend      ; dividend, low
8:          div  cx               ; AX = quotient (low)
9:          mov  remainder,dx     ; save remainder (low)
10:         .                     ; BX:AX = 0080:1002h
11:         .
12:   dividend   label word
13:              dd   08010020h
14:   divisor    dw   10h
15:   remainder  dw   ?            ; result = 0000h
```

First, we divide the most significant word of the dividend. Lines 1 and 2 load the dividend into AX and sign-extend it into DX. Line 4 divides the high part of the dividend (0801h) by 10h. The quotient is 0080h and the remainder is 0001h:

```
0000:0801h  /  10h  =  0080h,   remainder 1
 (DX:AX)      (CX)     (AX)        (DX)
```

Line 6 saves the high quotient in BX. The remainder (DX) will become the most significant part of the new dividend. When line 7 loads AX with the low half of