4

**Data Transfers, Addressing, and Arithmetic**

## 95

This chapter introduces some essential instructions for transferring data and performing arith- metic. A large part of this chapter is devoted to the basic addressing modes, such as direct, immediate, and indirect, which make it possible to process arrays. Along with that, we show how to create loops, and use some of the basic operators, such as OFFSET, PTR, and LENGTHOF. After reading this chapter, you should have a basic working knowledge of assem- bly language, with the exception of conditional statements.

# Data Transfer Instructions

## Introduction

When programming in languages like Java or C++, it’s easy for beginners to be annoyed when the compilers generate lots of syntax error messages. Compilers perform strict type checking in order to help you avoid possible errors such as mismatching variables and data. Assemblers, on the other hand, let you do just about anything you want, as long as the processor’s instruction set can do what you ask. In other words, assembly language forces you to pay attention to data stor- age and machine-specific details. You must understand the processor’s limitations when you write assembly language code. As it happens, x86 processors have what is commonly known as a *complex instruction set,* so they offer a lot of ways of doing things.

If you take the time to thoroughly learn the material presented in this chapter, the rest of this book will read a lot more smoothly. As the example programs become more complicated, you will rely on mastery of fundamental tools presented in this chapter.

## Operand Types

Chapter 3 introduced x86 instruction formats:

[label:] mnemonic [operands][ ; comment ]

Instructions can have zero, one, two, or three operands. Here, we omit the label and comment fields for clarity:

*mnemonic*

*mnemonic* [*destination*]

*mnemonic* [*destination*],[*source*]

*mnemonic* [*destination*],[*source-1*],[*source-2*]

There are three basic types of operands:

* Immediate—uses a numeric literal expression
* Register—uses a named register in the CPU
* Memory—references a memory location

Table 4-1 describes the standard operand types. It uses a simple notation for operands (in 32-bit mode) freely adapted from the Intel manuals. We will use it from this point on to describe the syntax of individual instructions.

## Direct Memory Operands

Variable names are references to offsets within the data segment. For example, the following declaration for a variable named **var1** says that its size attribute is **byte** and it contains the value 10 hexadecimal:

**Table 4-1** Instruction Operand Notation, 32-Bit Mode.

|  |  |
| --- | --- |
| **Operand** | **Description** |
| *reg8* | 8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL |
| *reg16* | 16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP |
| *reg32* | 32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP |
| *reg* | Any general-purpose register |
| *sreg* | 16-bit segment register: CS, DS, SS, ES, FS, GS |
| *imm* | 8-, 16-, or 32-bit immediate value |
| *imm8* | 8-bit immediate byte value |
| *imm16* | 16-bit immediate word value |
| *imm32* | 32-bit immediate doubleword value |
| *reg/mem8* | 8-bit operand, which can be an 8-bit general register or memory byte |
| *reg/mem16* | 16-bit operand, which can be a 16-bit general register or memory word |
| *reg/mem32* | 32-bit operand, which can be a 32-bit general register or memory doubleword |
| *mem* | An 8-, 16-, or 32-bit memory operand |

.data

var1 BYTE 10h

We can write instructions that dereference (look up) memory operands using their addresses. Suppose **var1** were located at offset 10400h. The following instruction copies its value into the AL register:

mov al var1

It would be assembled into the following machine instruction:

A0 00010400

The first byte in the machine instruction is the operation code (known as the *opcode*). The remaining part is the 32-bit hexadecimal address of **var1**. Although it might be possible to write programs using only numeric addresses, symbolic names such as **var1** make it easier to refer- ence memory.

**Alternative Notation.** Some programmers prefer to use the following notation with direct oper- ands because the brackets imply a dereference operation:

mov al,[var1]

MASM permits this notation, so you can use it in your own programs if you want. Because so many programs (including those from Microsoft) are printed without the brackets, we will only use them in this book when an arithmetic expression is involved:

mov al,[var1 + 5]

(This is called a direct-offset operand, a subject discussed at length in Section 4.1.8.)

## MOV Instruction

The MOV instruction copies data from a source operand to a destination operand. Known as a *data transfer* instruction, it is used in virtually every program. Its basic format shows that the first operand is the destination and the second operand is the source:

MOV *destination,source*

The destination operand’s contents change, but the source operand is unchanged. The right to left movement of data is similar to the assignment statement in C++ or Java:

dest = source;

In nearly all assembly language instructions, the left-hand operand is the destination and the right- hand operand is the source. MOV is very flexible in its use of operands, as long as the following rules are observed:

* + - * Both operands must be the same size.
      * Both operands cannot be memory operands.
      * The instruction pointer register (IP, EIP, or RIP) cannot be a destination operand. Here is a list of the standard MOV instruction formats:

MOV *reg,reg* MOV *mem,reg* MOV *reg,mem* MOV *mem,imm* MOV *reg,imm*

*Memory to Memory* A single MOV instruction cannot be used to move data directly from one memory location to another. Instead, you must move the source operand’s value to a register before assigning its value to a memory operand:

.data

var1 WORD ? var2 WORD ?

.code

mov ax,var1 mov var2,ax

You must consider the minimum number of bytes required by an integer constant when copying it to a variable or register. For unsigned integer constant sizes, refer to Table 1-4 in Chapter 1. For signed integer constants, refer to Table 1-7.

### Overlapping Values

The following code example shows how the same 32-bit register can be modified using differently sized data. When **oneWord** is moved to AX, it overwrites the existing value of AL. When **oneDword** is moved to EAX, it overwrites AX. Finally, when 0 is moved to AX, it overwrites the lower half of EAX.

.data

oneByte BYTE 78h oneWord WORD 1234h

oneDword DWORD 12345678h

.code

mov eax,0 ; EAX = 00000000h mov al,oneByte ; EAX = 00000078h mov ax,oneWord ; EAX = 00001234h mov eax,oneDword ; EAX = 12345678h mov ax,0 ; EAX = 12340000h

## Zero/Sign Extension of Integers

### Copying Smaller Values to Larger Ones

Although MOV cannot directly copy data from a smaller operand to a larger one, programmers can create workarounds. Suppose **count** (unsigned, 16 bits) must be moved to ECX (32 bits). We can set ECX to zero and move **count** to CX:

.data

count WORD 1

.code

mov ecx,0 mov cx,count

What happens if we try the same approach with a signed integer equal to —16?

.data

signedVal SWORD -16 ; FFF0h (-16)

.code

mov ecx,0

mov cx,signedVal ; ECX = 0000FFF0h (+65,520)

The value in ECX (+65,520) is completely different from —16. On the other hand, if we had filled ECX first with FFFFFFFFh and then copied **signedVal** to CX, the final value would have been correct:

mov ecx,0FFFFFFFFh

mov cx,signedVal ; ECX = FFFFFFF0h (-16)

The effective result of this example was to use the highest bit of the source operand (1) to fill the upper 16 bits of the destination operand, ECX. This technique is called *sign extension*. Of course, we cannot always assume that the highest bit of the source is a 1. Fortunately, the engi- neers at Intel anticipated this problem when designing the instruction set and introduced the MOVZX and MOVSX instructions to deal with both unsigned and signed integers.

### MOVZX Instruction

The MOVZX instruction (*move with zero-extend*) copies the contents of a source operand into a destination operand and zero-extends the value to 16 or 32 bits. This instruction is only used with unsigned integers. There are three variants:

MOVZX *reg32,reg/mem8* MOVZX *reg32,reg/mem16* MOVZX *reg16,reg/mem8*

(Operand notation was explained in Table 4-1.) In each of the three variants, the first operand (a register) is the destination and the second is the source. Notice that the source operand cannot be a constant. The following example zero-extends binary 10001111 into AX:

.data

byteVal BYTE 10001111b

.code

movzx ax,byteVal ; AX = 0000000010001111b

Figure 4-1 shows how the source operand is zero-extended into the 16-bit destination.

**Figure 4–1** Using MOVZX to copy a byte into a 16-bit destination.

0 *Source*

1 0 0 0 1 1 1 1

1 0 0 0 1 1 1 1

0 0 0 0 0 0 0 0

*Destination*

The following examples use registers for all operands, showing all the size variations:

mov bx,0A69Bh

movzx eax,bx ; EAX = 0000A69Bh

movzx edx,bl ; EDX = 0000009Bh

movzx cx,bl ; CX = 009Bh

The following examples use memory operands for the source and produce the same results:

.data

byte1 BYTE 9Bh word1 WORD 0A69Bh

.code

movzx eax,word1 ; EAX = 0000A69Bh

movzx edx,byte1 ; EDX = 0000009Bh

movzx cx,byte1 ; CX = 009Bh

### MOVSX Instruction

The MOVSX instruction (move with sign-extend) copies the contents of a source operand into a destination operand and sign-extends the value to 16 or 32 bits. This instruction is only used with signed integers. There are three variants:

MOVSX *reg32,reg/mem8* MOVSX *reg32,reg/mem16* MOVSX *reg16,reg/mem8*

An operand is sign-extended by taking the smaller operand’s highest bit and repeating (repli- cating) the bit throughout the extended bits in the destination operand. The following example sign-extends binary 10001111b into AX:

.data

byteVal BYTE 10001111b

.code

movsx ax,byteVal ; AX = 1111111110001111b

The lowest 8 bits are copied as in Figure 4-2. The highest bit of the source is copied into each of the upper 8 bit positions of the destination.

A hexadecimal constant has its highest bit set if its most significant hexadecimal digit is greater than 7. In the following example, the hexadecimal value moved to BX is A69B, so the leading “A” digit tells us that the highest bit is set. (The leading zero appearing before A69B is just a notational convenience so the assembler does not mistake the constant for the name of an identifier.)

mov bx,0A69Bh

movsx eax,bx ; EAX = FFFFA69Bh

movsx edx,bl ; EDX = FFFFFF9Bh

movsx cx,bl ; CX = FF9Bh

**Figure 4–2** Using MOVSX to copy a byte into a 16-bit destination.

*Source*



(Copy 8 bits)

1 0 0 0 1 1 1 1

*Destination*

1 0 0 0 1 1 1 1

1 1 1 1 1 1 1 1

## LAHF and SAHF Instructions

The LAHF (load status flags into AH) instruction copies the low byte of the EFLAGS register into AH. The following flags are copied: Sign, Zero, Auxiliary Carry, Parity, and Carry. Using this instruction, you can easily save a copy of the flags in a variable for safekeeping:

.data

saveflags BYTE ?

.code

lahf ; load flags into AH

mov saveflags,ah ; save them in a variable

The SAHF (store AH into status flags) instruction copies AH into the low byte of the EFLAGS (or RFLAGS) register. For example, you can retrieve the values of flags saved earlier in a variable:

mov ah,saveflags ; load saved flags into AH

sahf ; copy into Flags register

## XCHG Instruction

The XCHG (exchange data) instruction exchanges the contents of two operands. There are three variants:

XCHG *reg,reg* XCHG *reg,mem* XCHG *mem,reg*

The rules for operands in the XCHG instruction are the same as those for the MOV instruction (Section 4.1.4), except that XCHG does not accept immediate operands. In array sorting applications, XCHG provides a simple way to exchange two array elements. Here are a few examples using XCHG:

xchg ax,bx ; exchange 16-bit regs

xchg ah,al ; exchange 8-bit regs

xchg var1,bx ; exchange 16-bit mem op with BX

xchg eax,ebx ; exchange 32-bit regs

To exchange two memory operands, use a register as a temporary container and combine MOV with XCHG:

mov ax,val1 xchg ax,val2 mov val1,ax

## Direct-Offset Operands

You can add a displacement to the name of a variable, creating a direct-offset operand. This lets you access memory locations that may not have explicit labels. Let’s begin with an array of bytes named **arrayB**:

arrayB BYTE 10h,20h,30h,40h,50h

If we use MOV with **arrayB** as the source operand, we automatically move the first byte in the array:

mov al,arrayB ; AL = 10h

We can access the second byte in the array by adding 1 to the offset of **arrayB**:

mov al,[arrayB+1] ; AL = 20h

The third byte is accessed by adding 2:

mov al,[arrayB+2] ; AL = 30h

An expression such as **arrayB**+**1** produces what is called an *effective address* by adding a constant to the variable’s offset. Surrounding an effective address with brackets makes it clear that the expres- sion is dereferenced to obtain the contents of memory at the address. The assembler does not require you to surround address expressions with brackets, but we highly recommend their use for clarity.

MASM has no built-in range checking for effective addresses. In the following example, assuming **arrayB** holds five bytes, the instruction retrieves a byte of memory outside the array. The result is a sneaky logic bug, so be extra careful when checking array references:

mov al,[arrayB+20] ; AL = ??

*Word and Doubleword Arrays* In an array of 16-bit words, the offset of each array element is 2 bytes beyond the previous one. That is why we add 2 to **ArrayW** in the next example to reach the second element:

.data

arrayW WORD 100h,200h,300h

.code

mov ax,arrayW ; AX = 100h

mov ax,[arrayW+2] ; AX = 200h

Similarly, the second element in a doubleword array is 4 bytes beyond the first one:

.data

arrayD DWORD 10000h,20000h

.code

mov eax,arrayD ; EAX = 10000h

mov eax,[arrayD+4] ; EAX = 20000h

## Example Program (Moves)

Let’s combine all the instructions we’ve covered so far in this chapter, including MOV, XCHG, MOVSX, and MOVDX, to show how bytes, words, and doublewords are affected. We will also include some direct-offset operands.

; Data Transfer Examples (Moves.asm)

.386

.model flat,stdcall

.stack 4096

ExitProcess PROTO,dwExitCode:DWORD

.data

val1 WORD 1000h val2 WORD 2000h

arrayB BYTE 10h,20h,30h,40h,50h arrayW WORD 100h,200h,300h arrayD DWORD 10000h,20000h

.code main PROC

; Demonstrating MOVZX instruction: mov bx,0A69Bh

movzx eax,bx ; EAX = 0000A69Bh

movzx edx,bl ; EDX = 0000009Bh

movzx cx,bl ; CX = 009Bh

; Demonstrating MOVSX instruction: mov bx,0A69Bh

movsx eax,bx ; EAX = FFFFA69Bh

movsx edx,bl ; EDX = FFFFFF9Bh mov bl,7Bh

movsx cx,bl ; CX = 007Bh

; Memory-to-memory exchange:

mov ax,val1 ; AX = 1000h

xchg ax,val2 ; AX=2000h, val2=1000h

mov val1,ax ; val1 = 2000h

; Direct-Offset Addressing (byte array):

mov al,arrayB ; AL = 10h

mov al,[arrayB+1] ; AL = 20h

mov al,[arrayB+2] ; AL = 30h

; Direct-Offset Addressing (word array):

mov ax,arrayW ; AX = 100h

mov ax,[arrayW+2] ; AX = 200h

; Direct-Offset Addressing (doubleword array):

mov eax,arrayD ; EAX = 10000h

mov eax,[arrayD+4] ; EAX = 20000h

mov eax,[arrayD+4] ; EAX = 20000h

INVOKE ExitProcess,0 main ENDP

END main

This program generates no screen output, but you can (and should) run it using a debugger.

### Displaying CPU Flags in the Visual Studio Debugger

To display the CPU status flags during a debugging session, select *Windows* from the *Debug* menu, then select *Registers* from the *Windows* menu. Inside the *Registers* window, right-click and select *Flags* from the dropdown list. You must be currently debugging a program in order to see these menu options. The following table identifies the flag symbols used inside the *Registers* window:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Flag Name** | **Overflow** | **Direction** | **Interrupt** | **Sign** | **Zero** | **Aux Carry** | **Parity** | **Carry** |
| Symbol | OV | UP | EI | PL | ZR | AC | PE | CY |

Each flag is assigned a value of 0 (*clear*) or 1 (*set*). Here’s an example:

OV = 0 UP = 0 EI = 1

PL = 0 ZR = 1 AC = 0

PE = 1 CY = 0

As you step through your code during a debugging session, each flag displays in red when an instruction modifies the flag’s value. You can learn how instructions affect the flags by stepping through instructions and keeping an eye on the changing values of the flags.

## Section Review

1. What are the three basic types of operands?
2. *(True/False):* The destination operand of a MOV instruction cannot be a segment register.
3. *(True/False):* In a MOV instruction, the second operand is known as the *destination* operand.
4. *(True/False):* The EIP register cannot be the destination operand of a MOV instruction.
5. In the operand notation used by Intel, what does *reg/mem32* indicate?
6. In the operand notation used by Intel, what does *imm16* indicate?

# Addition and Subtraction

Arithmetic is a surprisingly big topic in assembly language! This chapter will focus on addition and subtraction. Then we will talk about multiplication and division later in Chapter 7. Then we’ll switch over to floating point arithmetic in Chapter 12.

Let’s start with the easiest and most efficient instructions of them all: INC (increment) and DEC (decrement), which add 1 and subtract 1. Then we will move on to the ADD, SUB, and NEG (negate) instructions, which offer more possibilities. Last of all, we will get into a discus- sion about how the CPU status flags (Carry, Sign, Zero, etc.) are affected by arithmetic instruc- tions. Remember, assembly language is all about the details.

## INC and DEC Instructions

The INC (increment) and DEC (decrement) instructions, respectively, add 1 and subtract 1 from a register or memory operand. The syntax is

INC *reg/mem*

DEC *reg/mem*

Following are some examples:

.data

myWord WORD 1000h

.code

inc myWord ; myWord = 1001h mov bx,myWord

dec bx ; BX = 1000h

The Overflow, Sign, Zero, Auxiliary Carry, and Parity flags are changed according to the value of the destination operand. The INC and DEC instructions do not affect the Carry flag (which is something of a surprise).

## ADD Instruction

The ADD instruction adds a source operand to a destination operand of the same size. The syntax is

ADD *dest,source*

*Source* is unchanged by the operation, and the sum is stored in the destination operand. The set of possible operands is the same as for the MOV instruction (Section 4.1.4). Here is a short code example that adds two 32-bit integers:

.data

var1 DWORD 10000h var2 DWORD 20000h

.code

mov eax,var1 ; EAX = 10000h

add eax,var2 ; EAX = 30000h

*Flags* The Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity flags are changed accord- ing to the value that is placed in the destination operand. We will explain how the flags work in Section 4.2.6.

## SUB Instruction

The SUB instruction subtracts a source operand from a destination operand. The set of pos- sible operands is the same as for the ADD and MOV instructions. The syntax is

SUB *dest,source*

Here is a short code example that subtracts two 32-bit integers:

.data

var1 DWORD 30000h var2 DWORD 10000h

.code

mov eax,var1 ; EAX = 30000h

sub eax,var2 ; EAX = 20000h

*Flags* The Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity flags are changed accord- ing to the value that is placed in the destination operand.

## NEG Instruction

The NEG (negate) instruction reverses the sign of a number by converting the number to its two’s complement. The following operands are permitted:

NEG *reg*

NEG *mem*

(Recall that the two’s complement of a number can be found by reversing all the bits in the desti- nation operand and adding 1.)

*Flags* The Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity flags are changed accord- ing to the value that is placed in the destination operand.

## Implementing Arithmetic Expressions

Armed with the ADD, SUB, and NEG instructions, you have the means to implement arithmetic expressions involving addition, subtraction, and negation in assembly language. In other words, you can simulate what a C++ compiler might do when a statement such as this:

Rval = -Xval + (Yval - Zval);

Let’s see how the sample statement would be implemented in assembly language. The following signed 32-bit variables will be used:

|  |  |  |
| --- | --- | --- |
| Rval | SDWORD | ? |
| Xval | SDWORD | 26 |
| Yval | SDWORD | 30 |
| Zval | SDWORD | 40 |

When translating an expression, evaluate each term separately and combine the terms at the end. First, we negate a copy of **Xval** and store it in a register:

; first term: -Xval mov eax,Xval

neg eax ; EAX = -26

Then **Yval** is copied to a register and **Zval** is subtracted:

; second term: (Yval - Zval) mov ebx,Yval

sub ebx,Zval ; EBX = -10

Finally, the two terms (in EAX and EBX) are added:

; add the terms and store:

add eax,ebx

mov Rval,eax ; -36

## Flags Affected by Addition and Subtraction

When executing arithmetic instructions, we often want to know something about the result. Is it neg- ative, positive, or zero? Is it too large or too small to fit into the destination operand? Answers to such questions can help us detect calculation errors that might otherwise cause erratic program behavior. We use the values of CPU status flags to check the outcome of arithmetic operations. We also use status flag values to activate conditional branching instructions, the basic tools of program logic. Here’s a quick overview of the status flags.

* + - * The Carry flag indicates unsigned integer overflow. For example, if an instruction has an 8-bit destination operand but the instruction generates a result larger than 11111111 binary, the Carry flag is set.
      * The Overflow flag indicates signed integer overflow. For example, if an instruction has a 16- bit destination operand but it generates a negative result smaller than —32,768 decimal, the Overflow flag is set.
      * The Zero flag indicates that an operation produced zero. For example, if an operand is sub- tracted from another of equal value, the Zero flag is set.
      * The Sign flag indicates that an operation produced a negative result. If the most significant bit (MSB) of the destination operand is set, the Sign flag is set.
      * The Parity flag indicates whether or not an even number of 1 bits occurs in the least signifi- cant byte of the destination operand, immediately after an arithmetic or boolean instruction has executed.
      * The Auxiliary Carry flag is set when a 1 bit carries out of position 3 in the least significant byte of the destination operand.

To display CPU status flag values when debugging, open the Registers window, right-click in the window, and select *Flags*.

### Unsigned Operations: Zero, Carry, and Auxiliary Carry

The Zero flag is set when the result of an arithmetic operation equals zero. The following exam- ples show the state of the destination register and Zero flag after executing the SUB, INC, and DEC instructions:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| mov | ecx,1 |  | | | | | | |
| sub | ecx,1 | ; | ECX | = | 0, | ZF | = | 1 |
| mov | eax,0FFFFFFFFh |  |  |  |  |  |  |  |
| inc | eax | ; | EAX | = | 0, | ZF | = | 1 |
| inc | eax | ; | EAX | = | 1, | ZF | = | 0 |
| dec | eax | ; | EAX | = | 0, | ZF | = | 1 |

*Addition and the Carry Flag* The Carry flag’s operation is easiest to explain if we consider addition and subtraction separately. When adding two unsigned integers, the Carry flag is a copy of the carry out of the most significant bit of the destination operand. Intuitively, we can say CF = 1 when the sum exceeds the storage size of its destination operand. In the next example, ADD sets the Carry flag because the sum (100h) is too large for AL:

mov al,0FFh

add al,1 ; AL = 00, CF = 1

Figure 4-3 shows what happens at the bit level when 1 is added to 0FFh. The carry out of the highest bit position of AL is copied into the Carry flag.

**Figure 4–3** Adding 1 to 0FFh sets the Carry flag.

1 1 1 1 1 1 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

+

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

CF

1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

On the other hand, if 1 is added to 00FFh in AX, the sum easily fits into 16 bits and the Carry flag is clear:

mov ax,00FFh

add ax,1 ; AX = 0100h, CF = 0

But adding 1 to FFFFh in the AX register generates a Carry out of the high bit position of AX:

mov ax,0FFFFh

add ax,1 ; AX = 0000, CF = 1

*Subtraction and the Carry Flag* A subtract operation sets the Carry flag when a larger unsigned integer is subtracted from a smaller one. Figure 4-4 shows what happens when we sub- tract 2 from 1, using 8-bit operands. Here is the corresponding assembly code:

mov al,1

sub al,2 ; AL = FFh, CF = 1

***Tip:*** The INC and DEC instructions do not affect the Carry flag. Applying the NEG instruction to a nonzero operand always sets the Carry flag.

**Figure 4–4** Subtracting 2 from 1 sets the Carry flag.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

+

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

CF

1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

(1)

(—2)

(FFh)

*Auxiliary Carry* The Auxiliary Carry (AC) flag indicates a carry or borrow out of bit 3 in the destination operand. It is primarily used in binary coded decimal (BCD) arithmetic, but can be used in other contexts. Suppose we add 1 to 0Fh. The sum (10h) contains a 1 in bit position 4 that was carried out of bit position 3:

mov al,0Fh

add al,1 ; AC = 1

Here is the arithmetic:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| + 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

*Parity* The Parity flag (PF) is set when the least significant byte of the destination has an even number of 1 bits. The following ADD and SUB instructions alter the parity of AL:

mov al,10001100b

add al,00000010b ; AL = 10001110, PF = 1

sub al,10000000b ; AL = 00001110, PF = 0

After the ADD instruction executes, AL contains binary 10001110 (four 0 bits and four 1 bits), and PF = 1. After the SUB instruction executes, AL contains an odd number of 1 bits, so the Parity flag equals 0.

### Signed Operations: Sign and Overflow Flags

*Sign Flag* The Sign flag is set when the result of a signed arithmetic operation is negative. The next example subtracts a larger integer (5) from a smaller one (4):

mov eax,4

sub eax,5 ; EAX = -1, SF = 1

From a mechanical point of view, the Sign flag is a copy of the destination operand’s high bit. The next example shows the hexadecimal values of BL when a negative result is generated:

mov bl,1 ; BL = 01h

sub bl,2 ; BL = FFh (-1), SF = 1

*Overflow Flag* The Overflow flag is set when the result of a signed arithmetic operation over- flows or underflows the destination operand. For example, from Chapter 1 we know that the largest possible integer signed byte value is +127; adding 1 to it causes overflow:

mov al,+127

add al,1 ; OF = 1

Similarly, the smallest possible negative integer byte value is —128. Subtracting 1 from it causes underflow. The destination operand value does not hold a valid arithmetic result, and the Over- flow flag is set:

mov al,-128

sub al,1 ; OF = 1

*The Addition Test* There is a very easy way to tell whether signed overflow has occurred when adding two operands. Overflow occurs when:

* Adding two positive operands generates a negative sum
* Adding two negative operands generates a positive sum

Overflow never occurs when the signs of two addition operands are different.

*How the Hardware Detects Overflow* The CPU uses an interesting mechanism to determine the state of the Overflow flag after an addition or subtraction operation. The value that carries out of the highest bit position is exclusive ORed with the carry into the high bit of the result. The resulting value is placed in the Overflow flag. In Figure 4-5, we show that adding the 8-bit binary integers 10000000 and 11111110 produces CF = 1, with carryIn(bit7) = 0. In other words, 1 XOR 0 produces OF = 1.

**Figure 4–5** Demonstration of how the Overflow flag is set.

+

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 0 0 0 0 0 0 0 | | | | | | | |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

CF

1

*NEG Instruction* The NEG instruction produces an invalid result if the destination operand can- not be stored correctly. For example, if we move —128 to AL and try to negate it, the correct value (+128) will not fit into AL. The Overflow flag is set, indicating that AL contains an invalid value:

mov al,-128 ; AL = 10000000b

neg al ; AL = 10000000b, OF = 1

On the other hand, if +127 is negated, the result is valid and the Overflow flag is clear:

mov al,+127 ; AL = 01111111b

neg al ; AL = 10000001b, OF = 0

How does the CPU know whether an arithmetic operation is signed or unsigned? We can only give what seems a dumb answer: It doesn’t! The CPU sets all status flags after an arithmetic operation using a set of boolean rules, regardless of which flags are relevant. You (the programmer) decide which flags to interpret and which to ignore, based on your knowledge of the type of operation performed.

* + 1. **Example Program (*AddSubTest*)**

The *AddSubTest* program shown below implements various arithmetic expressions using the ADD, SUB, INC, DEC, and NEG instructions, and shows how certain status flags are affected:

; Addition and Subtraction (AddSubTest.asm)

.386

.model flat,stdcall

.stack 4096

ExitProcess proto,dwExitCode:dword

.data

|  |  |  |
| --- | --- | --- |
| Rval | SDWORD | ? |
| Xval | SDWORD | 26 |
| Yval | SDWORD | 30 |
| Zval  .code | SDWORD | 40 |

main PROC

; INC and DEC mov ax,1000h

inc ax ; 1001h

dec ax ; 1000h

; Expression: Rval = -Xval + (Yval - Zval) mov eax,Xval

neg eax ; -26

mov ebx,Yval

sub ebx,Zval ; -10 add eax,ebx

mov Rval,eax ; -36

; Zero flag example:

mov cx,1

sub cx,1 ; ZF = 1 mov ax,0FFFFh

inc ax ; ZF = 1

; Sign flag example:

mov cx,0

sub cx,1 ; SF = 1 mov ax,7FFFh

add ax,2 ; SF = 1

; Carry flag example:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| mov | al,0FFh |  | | | | |
| add | al,1 | ; | CF | = | 1, | AL = 00 |

; Overflow flag example: mov al,+127

add al,1 ; OF = 1 mov al,-128

sub al,1 ; OF = 1

INVOKE ExitProcess,0 main ENDP

END main

## Section Review

### Use the following data for Questions 1-5:

.data

val1 BYTE 10h val2 WORD 8000h val3 DWORD 0FFFFh

val4 WORD 7FFFh

1. Write an instruction that increments **val2**.
2. Write an instruction that subtracts **val3** from EAX.
3. Write instructions that subtract **val4** from **val2**.
4. If **val2** is incremented by 1 using the ADD instruction, what will be the values of the Carry and Sign flags?
5. If **val4** is incremented by 1 using the ADD instruction, what will be the values of the Over- flow and Sign flags?
6. Where indicated, write down the values of the Carry, Sign, Zero, and Overflow flags after each instruction has executed:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| mov  add | ax,7FF0h  al,10h | ; | a. CF | = | SF | = | ZF | = | OF | = |
| add | ah,1 | ; | b. CF | = | SF | = | ZF | = | OF | = |
| add | ax,2 | ; | c. CF | = | SF | = | ZF | = | OF | = |

# Data-Related Operators and Directives

Operators and directives are not executable instructions; instead, they are interpreted by the assembler. You can use a number of assembly language directives to get information about the addresses and size characteristics of data:

* The OFFSET operator returns the distance of a variable from the beginning of its enclosing segment.
* The PTR operator lets you override an operand’s default size.
* The TYPE operator returns the size (in bytes) of an operand or of each element in an array.
* The LENGTHOF operator returns the number of elements in an array.
* The SIZEOF operator returns the number of bytes used by an array initializer.

In addition, the LABEL directive provides a way to redefine the same variable with different size attributes. The operators and directives in this chapter represent only a small subset of the operators supported by MASM. You may want to view the complete list in Appendix D.

## OFFSET Operator

The OFFSET operator returns the offset of a data label. The offset represents the distance, in bytes, of the label from the beginning of the data segment. To illustrate, Figure 4-6 shows a vari- able named **myByte** inside the data segment.

**Figure 4–6** A variable named myByte.

Offset

Data segment:

myByte

### OFFSET Examples

In the next example, we declare three different types of variables:

.data

bVal BYTE ? wVal WORD ? dVal DWORD ? dVal2 DWORD ?

If **bVal** were located at offset 00404000 (hexadecimal), the OFFSET operator would return the following values:

mov esi,OFFSET bVal ; ESI = 00404000h

mov esi,OFFSET wVal ; ESI = 00404001h

mov esi,OFFSET dVal ; ESI = 00404003h

mov esi,OFFSET dVal2 ; ESI = 00404007h

OFFSET can also be applied to a direct-offset operand. Suppose **myArray** contains five 16-bit words. The following MOV instruction obtains the offset of **myArray**, adds 4, and moves the resulting address to ESI. We can say that ESI points to the third integer in the array:

.data

myArray WORD 1,2,3,4,5

.code

mov esi,OFFSET myArray + 4

You can initialize a doubleword variable with the offset of another variable, effectively creating a pointer. In the following example, **pArray** points to the beginning of **bigArray**:

.data

bigArray DWORD 500 DUP(?) pArray DWORD bigArray

The following statement loads the pointer’s value into ESI, so the register can point to the begin- ning of the array:

mov esi,pArray

## ALIGN Directive

The ALIGN directive aligns a variable on a byte, word, doubleword, or paragraph boundary. The syntax is

ALIGN *bound*

*Bound* can be 1, 2, 4, 8, or 16. A value of 1 aligns the next variable on a 1-byte boundary (the default). If *bound* is 2, the next variable is aligned on an even-numbered address. If bound is 4, the next address is a multiple of 4. If *bound* is 16, the next address is a multiple of 16, a paragraph boundary. The assembler can insert one or more empty bytes before the variable to fix the alignment. Why bother aligning data? Because the CPU can process data stored at even- numbered addresses more quickly than those at odd-numbered addresses.

In the following example, **bVal** is arbitrarily located at offset 00404000. Inserting the ALIGN 2 directive before **wVal** causes it to be assigned an even-numbered offset:

bVal BYTE ? ; 00404000h ALIGN 2

wVal WORD ? ; 00404002h

bVal2 BYTE ? ; 00404004h ALIGN 4

dVal DWORD ? ; 00404008h

dVal2 DWORD ? ; 0040400Ch

Note that **dVal** would have been at offset 00404005, but the ALIGN 4 directive bumped it up to offset 00404008.

## PTR Operator

You can use the PTR operator to override the declared size of an operand. This is only necessary when you’re trying to access the operand using a size attribute that is different from the one assumed by the assembler.

Suppose, for example, that you would like to move the lower 16 bits of a doubleword variable named **myDouble** into AX. The assembler will not permit the following move because the oper- and sizes do not match:

.data

myDouble DWORD 12345678h

.code

mov ax,myDouble ; error

But the WORD PTR operator makes it possible to move the low-order word (5678h) to AX:

mov ax,WORD PTR myDouble

Why wasn’t 1234h moved into AX? x86 processors use the *little endian* storage format (Section 3.4.9), in which the low-order byte is stored at the variable’s starting address. In Figure 4-7, the memory layout of **myDouble** is shown three ways: first as a doubleword, then as two words (5678h, 1234h), and finally as four bytes (78h, 56h, 34h, 12h).

We can access memory in any of these three ways, independent of the way a variable was defined. For example, if **myDouble** begins at offset 0000, the 16-bit value stored at that address is 5678h. We could also retrieve 1234h, the word at location **myDouble**+**2**, using the following statement:

mov ax,WORD PTR [myDouble+2] ; 1234h

**Figure 4–7** Memory layout of myDouble.

**Doubleword Word Byte Offset**

|  |  |  |
| --- | --- | --- |
| 12345678 | 5678 | 78 |
|  | | 56 |
|  | 1234 | 34 |
|  | | 12 |

|  |  |
| --- | --- |
| 0000 | myDouble |
| 0001 | myDouble + 1 |
| 0002 | myDouble + 2 |
| 0003 | myDouble + 3 |

Similarly, we could use the BYTE PTR operator to move a single byte from **myDouble** to BL:

mov bl,BYTE PTR myDouble ; 78h

Note that PTR must be used in combination with one of the standard assembler data types, BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, or TBYTE.

*Moving Smaller Values into Larger Destinations* We might want to move two smaller val- ues from memory to a larger destination operand. In the next example, the first word is copied to the lower half of EAX and the second word is copied to the upper half. The DWORD PTR oper- ator makes this possible:

.data

wordList WORD 5678h,1234h

.code

mov eax,DWORD PTR wordList ; EAX = 12345678h

## TYPE Operator

The TYPE operator returns the size, in bytes, of a single element of a variable. For example, the TYPE of a byte equals 1, the TYPE of a word equals 2, the TYPE of a doubleword is 4, and the TYPE of a quadword is 8. Here are examples of each:

.data

var1 BYTE ? var2 WORD ? var3 DWORD ? var4 QWORD ?

The following table shows the value of each TYPE expression.

|  |  |
| --- | --- |
| **Expression** | **Value** |
| TYPE var1 | 1 |
| TYPE var2 | 2 |
| TYPE var3 | 4 |
| TYPE var4 | 8 |

## LENGTHOF Operator

The LENGTHOF operator counts the number of elements in an array, defined by the values appearing on the same line as its label. We will use the following data as an example:

.data

byte1 BYTE 10,20,30 array1 WORD 30 DUP(?),0,0

array2 WORD 5 DUP(3 DUP(?)) array3 DWORD 1,2,3,4

digitStr BYTE "12345678",0

When nested DUP operators are used in an array definition, LENGTHOF returns the prod- uct of the two counters. The following table lists the values returned by each LENGTHOF expression:

|  |  |
| --- | --- |
| **Expression** | **Value** |
| LENGTHOF byte1 | 3 |
| LENGTHOF array1 | 30 + 2 |
| LENGTHOF array2 | 5 \* 3 |
| LENGTHOF array3 | 4 |
| LENGTHOF digitStr | 9 |

If you declare an array that spans multiple program lines, LENGTHOF only regards the data from the first line as part of the array. Given the following data, LENGTHOF myArray would return the value 5:

myArray BYTE 10,20,30,40,50

BYTE 60,70,80,90,100

Alternatively, you can end the first line with a comma and continue the list of initializers onto the next line. Given the following data, LENGTHOF myArray would return the value 10:

myArray BYTE 10,20,30,40,50,

60,70,80,90,100

## SIZEOF Operator

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE. In the following example, **intArray** has TYPE = 2 and LENGTHOF = 32. Therefore, SIZEOF **intArray** equals 64:

.data

intArray WORD 32 DUP(0)

.code

mov eax,SIZEOF intArray ; EAX = 64

## LABEL Directive

The LABEL directive lets you insert a label and give it a size attribute without allocating any storage. All standard size attributes can be used with LABEL, such as BYTE, WORD, DWORD, QWORD or TBYTE. A common use of LABEL is to provide an alternative name and size

attribute for the variable declared next in the data segment. In the following example, we declare a label just before **val32** named **val16** and give it a WORD attribute:

.data

val16 LABEL WORD val32 DWORD 12345678h

.code

mov ax,val16 ; AX = 5678h

mov dx,[val16+2] ; DX = 1234h

**val16** is an alias for the same storage location as **val32**. The LABEL directive itself allocates no storage.

Sometimes we need to construct a larger integer from two smaller integers. In the next example, a 32-bit value is loaded into EAX from two 16-bit variables:

.data

LongValue LABEL DWORD val1 WORD 5678h val2 WORD 1234h

.code

mov eax,LongValue ; EAX = 12345678h

## Section Review

1. *(True/False):* The OFFSET operator always returns a 16-bit value.
2. *(True/False):* The PTR operator returns the 32-bit address of a variable.
3. *(True/False):* The TYPE operator returns a value of 4 for doubleword operands.
4. *(True/False):* The LENGTHOF operator returns the number of bytes in an operand.
5. *(True/False):* The SIZEOF operator returns the number of bytes in an operand.

# Indirect Addressing

Direct addressing is rarely used for array processing because it is impractical to use constant off- sets to address more than a few array elements. Instead, we use a register as a pointer (called *indirect addressing*) and manipulate the register’s value. When an operand uses indirect address- ing, it is called an *indirect operand*.

## Indirect Operands

*Protected Mode* An indirect operand can be any 32-bit general-purpose register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP) surrounded by brackets. The register is assumed to contain the address of some data. In the next example, ESI contains the offset of **byteVal**. The MOV instruction uses the indirect operand as the source, the offset in ESI is dereferenced, and a byte is moved to AL:

.data

byteVal BYTE 10h

.code

mov esi,OFFSET byteVal

mov al,[esi] ; AL = 10h

If the destination operand uses indirect addressing, a new value is placed in memory at the loca- tion pointed to by the register. In the following example, the contents of the BL register are cop- ied to the memory location addressed by ESI.

mov [esi],bl

*Using PTR with Indirect Operands* The size of an operand may not be evident from the context of an instruction. The following instruction causes the assembler to generate an “oper- and must have size” error message:

inc [esi] ; error: operand must have size

The assembler does not know whether ESI points to a byte, word, doubleword, or some other size. The PTR operator confirms the operand size:

inc BYTE PTR [esi]

## Arrays

Indirect operands are ideal tools for stepping through arrays. In the next example, **arrayB** con- tains 3 bytes. As ESI is incremented, it points to each byte, in order:

.data

arrayB BYTE 10h,20h,30h

.code

mov esi,OFFSET arrayB

mov al,[esi] ; AL = 10h inc esi

mov al,[esi] ; AL = 20h inc esi

mov al,[esi] ; AL = 30h

If we use an array of 16-bit integers, we add 2 to ESI to address each subsequent array element:

.data

arrayW WORD 1000h,2000h,3000h

.code

mov esi,OFFSET arrayW

mov ax,[esi] ; AX = 1000h add esi,2

mov ax,[esi] ; AX = 2000h add esi,2

mov ax,[esi] ; AX = 3000h

Suppose **arrayW** is located at offset 10200h. The following illustration shows the initial value of ESI in relation to the array data:

|  |  |  |
| --- | --- | --- |
| Offset | Value |  |
| 10200 | 1000h | [esi] |
| 10202 | 2000h |  |
| 10204 | 3000h |  |

*Example: Adding 32-Bit Integers* The following code example adds three doublewords. A displacement of 4 must be added to ESI as it points to each subsequent array value because doublewords are 4 bytes long:

.data

arrayD DWORD 10000h,20000h,30000h

.code

mov esi,OFFSET arrayD

mov eax,[esi] ; first number add esi,4

add eax,[esi] ; second number add esi,4

add eax,[esi] ; third number

Suppose **arrayD** is located at offset 10200h. Then the following illustration shows the initial value of ESI in relation to the array data:

Offset 10200

10204

10208

Value

 [esi]  [esi] + 4

|  |
| --- |
| 10000h |
| 20000h |
| 30000h |

 [esi] + 8

## Indexed Operands

An *indexed operand* adds a constant to a register to generate an effective address. Any of the 32-bit general-purpose registers may be used as index registers. There are different notational forms permitted by MASM (the brackets are part of the notation):

*constant*[*reg*] [*constant* + *reg*]

The first notational form combines the name of a variable with a register. The variable name is translated by the assembler into a constant that represents the variable’s offset. Here are exam- ples that show both notational forms:

|  |  |
| --- | --- |
| arrayB[esi] | [arrayB + esi] |
| arrayD[ebx] | [arrayD + ebx] |

Indexed operands are ideally suited to array processing. The index register should be initialized to zero before accessing the first array element:

.data

arrayB BYTE 10h,20h,30h

.code

mov esi,0

mov al,arrayB[esi] ; AL = 10h

The last statement adds ESI to the offset of **arrayB**. The address generated by the expression

**[arrayB** + **ESI]** is dereferenced and the byte in memory is copied to AL.

*Adding Displacements* The second type of indexed addressing combines a register with a constant offset. The index register holds the base address of an array or structure, and the con- stant identifies offsets of various array elements. The following example shows how to do this with an array of 16-bit words:

.data

arrayW WORD 1000h,2000h,3000h

.code

mov esi,OFFSET arrayW

mov ax,[esi] ; AX = 1000h

mov ax,[esi+2] ; AX = 2000h

mov ax,[esi+4] ; AX = 3000h

*Using 16-Bit Registers* It is usual to use 16-bit registers as indexed operands in real-address mode. In that case, you are limited to using SI, DI, BX, or BP:

mov al,arrayB[si] mov ax,arrayW[di] mov eax,arrayD[bx]

As is the case with indirect operands, avoid using BP except when addressing data on the stack.

### Scale Factors in Indexed Operands

Indexed operands must take into account the size of each array element when calculating offsets. Using an array of doublewords, as in the following example, we multiply the sub- script (3) by 4 (the size of a doubleword) to generate the offset of the array element contain- ing 400h:

.data

arrayD DWORD 100h, 200h, 300h, 400h

.code

mov esi,3 \* TYPE arrayD ; offset of arrayD[3]

mov eax,arrayD[esi] ; EAX = 400h

Intel designers wanted to make a common operation easier for compiler writers, so they provided a way for offsets to be calculated, using a *scale factor*. The scale factor is the size of the array component (word = 2, doubleword = 4, or quadword = 8). Let’s revise our previous example by setting ESI to the array subscript (3) and multiplying ESI by the scale factor (4) for doublewords:

.data

arrayD DWORD 1,2,3,4

.code

mov esi,3 ; subscript

mov eax,arrayD[esi\*4] ; EAX = 4

The TYPE operator can make the indexing more flexible should arrayD be redefined as another type in the future:

mov esi,3 ; subscript

mov eax,arrayD[esi\*TYPE arrayD] ; EAX = 4

## Pointers

A variable containing the address of another variable is called a *pointer*. Pointers are a great tool for manipulating arrays and data structures because the address they hold can be modified at runtime. You might use a system call to allocate (reserve) a block of memory, for example, and save the address of that block in a variable. A pointer’s size is affected by the processor’s current mode (32-bit or 64-bit). In the following 32-bit code example, **ptrB** contains the offset of arrayB:

.data

arrayB byte 10h,20h,30h,40h ptrB dword arrayB

Optionally, you can declare **ptrB** with the OFFSET operator to make the relationship clearer:

ptrB dword OFFSET arrayB

The 32-bit mode programs in this book use near pointers, so they are stored in doubleword variables. Here are two examples: **ptrB** contains the offset of **arrayB**, and **ptrW** contains the offset of **arrayW**:

|  |  |  |
| --- | --- | --- |
| arrayB | BYTE | 10h,20h,30h,40h |
| arrayW | WORD | 1000h,2000h,3000h |
| ptrB | DWORD | arrayB |
| ptrW | DWORD | arrayW |

Optionally, you can use the OFFSET operator to make the relationship clearer:

ptrB DWORD OFFSET arrayB ptrW DWORD OFFSET arrayW

High-level languages purposely hide physical details about pointers because their implementa- tions vary among different machine architectures. In assembly language, because we deal with a single implementation, we examine and use pointers at the physical level. This approach helps to remove some of the mystery surrounding pointers.

### Using the TYPEDEF Operator

The TYPEDEF operator lets you create a user-defined type that has all the status of a built-in type when defining variables. TYPEDEF is ideal for creating pointer variables. For example, the following declaration creates a new data type PBYTE that is a pointer to bytes:

PBYTE TYPEDEF PTR BYTE

This declaration would usually be placed near the beginning of a program, before the data seg- ment. Then, variables could be defined using PBYTE:

.data

arrayB BYTE 10h,20h,30h,40h

ptr1 PBYTE ? ; uninitialized

ptr2 PBYTE arrayB ; points to an array

*Example Program: Pointers* The following program (*pointers.asm*) uses TYPDEF to create three pointer types (PBYTE, PWORD, PDWORD). It creates several pointers, assigns several array offsets, and dereferences the pointers:

TITLE Pointers (Pointers.asm)

.386

.model flat,stdcall

.stack 4096

ExitProcess proto,dwExitCode:dword

; Create user-defined types.

PBYTE TYPEDEF PTR BYTE ; pointer to bytes PWORD TYPEDEF PTR WORD ; pointer to words

PDWORD TYPEDEF PTR DWORD ; pointer to doublewords

.data

arrayB BYTE 10h,20h,30h arrayW WORD 1,2,3

arrayD DWORD 4,5,6

; Create some pointer variables. ptr1 PBYTE arrayB

ptr2 PWORD arrayW ptr3 PDWORD arrayD

.code main PROC

; Use the pointers to access data. mov esi,ptr1

mov al,[esi] ; 10h

mov esi,ptr2

mov ax,[esi] ; 1

mov esi,ptr3

mov eax,[esi] ; 4

invoke ExitProcess,0 main ENDP

END main

## Section Review

1. *(True/False):* Any 32-bit general-purpose register can be used as an indirect operand.
2. *(True/False):* The EBX register is usually reserved for addressing the stack.
3. *(True/False):* The following instruction is invalid: inc [esi]
4. *(True/False):* The following is an indexed operand: array[esi]

### Use the following data definitions for Questions 5 and 6:

myBytes BYTE 10h,20h,30h,40h myWords WORD 8Ah,3Bh,72h,44h,66h myDoubles DWORD 1,2,3,4,5 myPointer DWORD myDoubles

1. Fill in the requested register values on the right side of the following instruction sequence:

mov esi,OFFSET myBytes

mov al,[esi] ; a. AL =

mov al,[esi+3] ; b. AL = mov esi,OFFSET myWords + 2

mov ax,[esi] ; c. AX = mov edi,8

mov edx,[myDoubles + edi] ; d. EDX = mov edx,myDoubles[edi] ; e. EDX = mov ebx,myPointer

mov eax,[ebx+4] ; f. EAX =

1. Fill in the requested register values on the right side of the following instruction sequence:

mov esi,OFFSET myBytes

mov ax,[esi] ; a. AX = mov eax,DWORD PTR myWords ; b. EAX = mov esi,myPointer

mov ax,[esi+2] ; c. AX =

mov ax,[esi+6] ; d. AX =

mov ax,[esi-4] ; e. AX =

# JMP and LOOP Instructions

By default, the CPU loads and executes programs sequentially. But the current instruction might be *conditional*, meaning that it transfers control to a new location in the program based on the values of CPU status flags (Zero, Sign, Carry, etc.). Assembly language programs use condi- tional instructions to implement high-level statements such as IF statements and loops. Each of the conditional statements involves a possible transfer of control (jump) to a different memory address. A *transfer of control*, or *branch*, is a way of altering the order in which statements are executed. There are two basic types of transfers:

* **Unconditional Transfer:** Control is transferred to a new location in all cases; a new address is loaded into the instruction pointer, causing execution to continue at the new address. The JMP instruction does this.
* **Conditional Transfer:** The program branches if a certain condition is true. A wide variety of conditional transfer instructions can be combined to create conditional logic structures. The CPU interprets true/false conditions based on the contents of the ECX and Flags registers.

## JMP Instruction

The JMP instruction causes an unconditional transfer to a destination, identified by a code label that is translated by the assembler into an offset. The syntax is

JMP *destination*

When the CPU executes an unconditional transfer, the offset of *destination* is moved into the instruction pointer, causing execution to continue at the new location.

*Creating a Loop* The JMP instruction provides an easy way to create a loop by jumping to a label at the top of the loop:

top:

.

.

jmp top ; repeat the endless loop

JMP is unconditional, so a loop like this will continue endlessly unless another way is found to exit the loop.

## LOOP Instruction

The LOOP instruction, formally known as *Loop According to ECX Counter*, repeats a block of statements a specific number of times. ECX is automatically used as a counter and is decre- mented each time the loop repeats. Its syntax is

LOOP *destination*

The loop destination must be within —128 to +127 bytes of the current location counter. The execution of the LOOP instruction involves two steps: First, it subtracts 1 from ECX. Next, it compares ECX to zero. If ECX is not equal to zero, a jump is taken to the label identified by *des- tination*. Otherwise, if ECX equals zero, no jump takes place, and control passes to the instruc- tion following the loop.

In real-address mode, CX is the default loop counter for the LOOP instruction. On the other hand, the LOOPD instruction uses ECX as the loop counter, and the LOOPW instruction uses CX as the loop counter.

In the following example, we add 1 to AX each time the loop repeats. When the loop ends, AX = 5 and ECX = 0:

mov ax,0 mov ecx,5

L1:

inc ax loop L1

A common programming error is to inadvertently initialize ECX to zero before beginning a loop. If this happens, the LOOP instruction decrements ECX to FFFFFFFFh, and the loop repeats 4,294,967,296 times! If CX is the loop counter (in real-address mode), it repeats 65,536 times.

Occasionally, you might create a loop that is large enough to exceed the allowed relative jump range of the LOOP instruction. Following is an example of an error message generated by MASM because the target label of a LOOP instruction was too far away:

error A2075: jump destination too far : by 14 byte(s)

Rarely should you explicitly modify ECX inside a loop. If you do, the LOOP instruction may not work as expected. In the following example, ECX is incremented within the loop. It never reaches zero, so the loop never stops:

top:

.

.

inc ecx loop top

If you need to modify ECX inside a loop, you can save it in a variable at the beginning of the loop and restore it just before the LOOP instruction:

.data

count DWORD ?

.code

mov ecx,100 ; set loop count

top:

mov count,ecx ; save the count

.

mov ecx,20 ; modify ECX

.

mov ecx,count ; restore loop count loop top

***Nested Loops*** When creating a loop inside another loop, special consideration must be given to the outer loop counter in ECX. You can save it in a variable:

.data

count DWORD ?

.code

mov ecx,100 ; set outer loop count

L1:

L2:

mov count,ecx ; save outer loop count

mov ecx,20 ; set inner loop count

.

.

loop L2 ; repeat the inner loop

mov ecx,count ; restore outer loop count

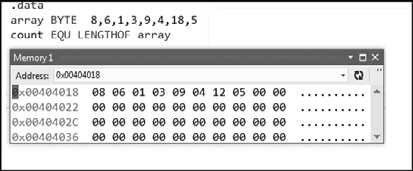
loop L1 ; repeat the outer loop

As a general rule, nested loops more than two levels deep are difficult to write. If the algo- rithm you’re using requires deep loop nesting, move some of the inner loops into subroutines.

## Displaying an Array in the Visual Studio Debugger

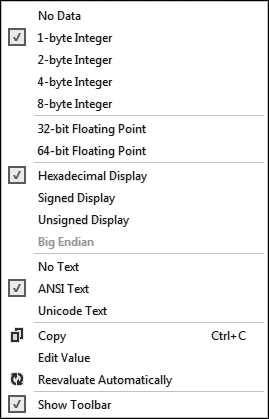
In a debugging session, if you want to display the contents of an array, here’s how to do it: From the *Debug* menu, select *Windows*, select *Memory*, then select *Memory 1*. A memory window will appear, and you can use the mouse to drag and dock it to any side of the Visual Studio work- space. You can also right-click the window’s title bar and indicate that you want the window to float above the editor window. In the *Address* field at the top of the memory window, type the & (ampersand) character, followed by the name of the array, and press *Enter*. For example, **&myArray** would be a valid address expression. The memory window will display a block of memory starting at the array’s address. Figure 4-8 shows an example.

**Figure 4–8** Using the debugger’s memory window to display an array.



If your array values are doublewords, you can right-click inside the memory window and select *4-byte integer* from the popup menu. You can also select from different formats, including *Hexadecimal Display*, signed decimal integer (called *Signed Display*), or unsigned decimal inte- ger (called *Unsigned Display*) formats. The full set of choices is shown in Figure 4-9.

**Figure 4–9** Popup menu for the debugger’s memory window.



## Summing an Integer Array

There’s hardly any task more common in beginning programming than calculating the sum of the elements in an array. In assembly language, you would follow these steps:

1. Assign the array’s address to a register that will serve as an indexed operand.
2. Initialize the loop counter to the length of the array.
3. Assign zero to the register that accumulates the sum.
4. Create a label to mark the beginning of the loop.
5. In the loop body, add a single array element to the sum.
6. Point to the next array element.
7. Use a LOOP instruction to repeat the loop.

Steps 1 through 3 may be performed in any order. Here’s a short program that sums an array of 16-bit integers.

; Summing an Array (SumArray.asm)

.386

.model flat,stdcall

.stack 4096

ExitProcess proto,dwExitCode:dword

.data

intarray DWORD 10000h,20000h,30000h,40000h

.code

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| main | PROC |  | | | |
|  | mov | edi,OFFSET intarray | ; | 1: | EDI = address of intarray |
|  | mov | ecx,LENGTHOF intarray | ; | 2: | initialize loop counter |
|  | mov | eax,0 | ; | 3: | sum = 0 |
| L1: |  |  | ; | 4: | mark beginning of loop |
|  | add | eax,[edi] | ; | 5: | add an integer |
|  | add | edi,TYPE intarray | ; | 6: | point to next element |
|  | loop | L1 | ; | 7: | repeat until ECX = 0 |

invoke ExitProcess,0 main ENDP

END main

## Copying a String

Programs often copy large blocks of data from one location to another. The data may be arrays or strings, but they can contain any type of objects. Let’s see how this can be done in assembly language, using a loop that copies a string, represented as an array of bytes with a null termina- tor value. Indexed addressing works well for this type of operation because the same index regis- ter references both strings. The target string must have enough available space to receive the copied characters, including the null byte at the end:

; Copying a String (CopyStr.asm)

.386

.model flat,stdcall

.stack 4096

ExitProcess proto,dwExitCode:dword

.data

source BYTE "This is the source string",0 target BYTE SIZEOF source DUP(0)

.code main PROC

mov esi,0 ; index register mov ecx,SIZEOF source ; loop counter

L1:

mov al,source[esi] ; get a character from source mov target[esi],al ; store it in the target

inc esi ; move to next character

loop L1 ; repeat for entire string invoke ExitProcess,0

main ENDP END main

The MOV instruction cannot have two memory operands, so each character is moved from the source string to AL, then from AL to the target string.

## Section Review

1. *(True/False):* A JMP instruction can only jump to a label inside the current procedure.
2. *(True/False):* JMP is a conditional transfer instruction.
3. If ECX is initialized to zero before beginning a loop, how many times will the LOOP instruction repeat? (Assume ECX is not modified by any other instructions inside the loop.)
4. *(True/False):* The LOOP instruction first checks to see whether ECX is not equal to zero; then LOOP decrements ECX and jumps to the destination label.
5. *(True/False):* The LOOP instruction does the following: It decrements ECX; then, if ECX is not equal to zero, LOOP jumps to the destination label.
6. In real-address mode, which register is used as the counter by the LOOP instruction?
7. In real-address mode, which register is used as the counter by the LOOPD instruction?
8. *(True/False):* The target of a LOOP instruction must be within 256 bytes of the current location.
9. *(Challenge):* What will be the final value of EAX in this example?

mov eax,0

mov ecx,10 ; outer loop counter

L1:

L2:

mov eax,3

mov ecx,5 ; inner loop counter

add eax,5

loop L2 ; repeat inner loop

loop L1 ; repeat outer loop

1. Revise the code from the preceding question so the outer loop counter is not erased when the inner loop starts.

# 64-Bit Programming

## MOV Instruction

The MOV instruction in 64-bit mode has a great deal in common with 32-bit mode. There are just a few differences, which we will discuss here. Immediate operands (constants) may be 8, 16, 32, or 64 bits. Here’s a 64-bit example:

mov rax,0ABCDEFGAFFFFFFFFh ; 64-bit immediate operand

When you move a 32-bit constant to a 64-bit register, the upper 32 bits (bits 32–63) of the desti- nation are cleared (equal to zero):

mov rax,0FFFFFFFFh ; rax = 00000000FFFFFFFF

When you move a 16-bit constant or an 8-bit constant into a 64-bit register, the upper bits are also cleared:

|  |  |  |
| --- | --- | --- |
| mov | rax,06666h | ; clears bits 16-63 |
| mov | rax,055h | ; clears bits 8-63 |

When you move memory operands into 64-bit registers, however, the results are mixed. For example, moving a 32-bit memory operand into EAX (the lower half of RAX) causes the upper 32 bits in RAX to be cleared:

.data

myDword DWORD 80000000h

.code

mov rax,0FFFFFFFFFFFFFFFFh

mov eax,myDword ; RAX = 0000000080000000

But when you move an 8-bit or a 16-bit memory operand into the lower bits of RAX, the highest bits in the destination register are not affected:

.data

myByte BYTE 55h myWord WORD 6666h

.code

mov ax,myWord ; bits 16-63 are not affected

mov al,myByte ; bits 8-63 are not affected

The MOVSXD instruction (move with sign-extension) permits the source operand to be a 32-bit register or memory operand. The following instructions cause RAX to equal FFFFFFFFFFFFFFFFh:

mov ebx,0FFFFFFFFh movsxd rax,ebx

The OFFSET operator generates a 64-bit address, which must be held by a 64-bit register or variable. In the following example, we use the RSI register:

.data

myArray WORD 10,20,30,40

.code

mov rsi,OFFSET myArray

The LOOP instruction in 64-bit mode uses the RCX register as the loop counter.

With these basic concepts, you can write quite a few programs in 64-bit mode. Most of the time, programming is easier if you consistently use 64-bit integer variables and 64-bit registers. ASCII strings are a special case because they always contain bytes. Usually, you use indirect or indexed addressing when processing them.

## 64-Bit Version of SumArray

Let’s recreate the **SumArray** program in 64-bit mode. It calculates the sum of an array of 64-bit integers. First, we use the QWORD directive to create an array of quadwords. Then, we change all 32-bit register names to 64-bit names. This is the complete program listing:

; Summing an Array (SumArray\_64.asm) ExitProcess PROTO

.data

intarray QWORD 1000000000000h,2000000000000h QWORD 3000000000000h,4000000000000h

.code main PROC

mov rdi,OFFSET intarray ; RDI = address of intarray mov rcx,LENGTHOF intarray ; initialize loop counter mov rax,0 ; sum = 0

L1: ; mark beginning of loop

add rax,[rdi] ; add an integer

add rdi,TYPE intarray ; point to next element loop L1 ; repeat until RCX = 0

mov ecx,0 ; ExitProcess return value call ExitProcess

main ENDP END

## Addition and Subtraction

The ADD, SUB, INC, and DEC instructions affect the CPU status flags in the same way in 64-bit mode as in 32-bit mode. In the following example, we add 1 to a 32-bit number in RAX. Each bit carries to the left, causing a 1 to be inserted in bit 32:

mov rax,0FFFFFFFFh ; fill the lower 32 bits

add rax,1 ; RAX = 100000000h

It always pays to know the sizes of your operands. When you use a partial register operand, be aware that the remainder of the register is not modified. In the next example, the 16-bit sum in AX rolls over to zero without affecting the upper bits in RAX. This happens because the opera- tion uses 16-bit registers (AX and BX):

|  |  |  |
| --- | --- | --- |
| mov | rax,0FFFFh | ; RAX = 000000000000FFFF |
| mov | bx,1 |  |
| add | ax,bx | ; RAX = 0000000000000000 |

Similarly, in the following example, the sum in AL does not carry into any other bits within RAX. After the ADD, RAX equals zero:

|  |  |  |
| --- | --- | --- |
| mov | rax,0FFh | ; RAX = 00000000000000FF |
| mov | bl,1 |  |
| add | al,bl | ; RAX = 0000000000000000 |

The same principle applies to subtraction. In the following code excerpt, subtracting 1 from zero in EAX causes the lower 32 bits of RAX to become equal to –1 (FFFFFFFFh). Similarly, sub- tracting 1 from zero in AX causes the lower 16 bits of RAX to become equal to –1 (FFFFh).

mov rax,0 ; RAX = 0000000000000000

mov ebx,1

sub eax,ebx ; RAX = 00000000FFFFFFFF

mov rax,0 ; RAX = 0000000000000000

mov bx,1

sub ax,bx ; RAX = 000000000000FFFF

A 64-bit general-purpose register must be used when an instruction contains an indirect operand. Remember that you must use the PTR operator to clarify the target operand’s size. Here are examples, including one with a 64-bit target:

dec BYTE PTR [rdi] ; 8-bit target

inc WORD PTR [rbx] ; 16-bit target

inc QWORD PTR [rsi] ; 64-bit target

In 64-bit mode, you can use scale factors in indexed operands, just as you do in 32-bit mode. If you’re working with an array of 64-bit integers, use a scale factor of 8. Here’s an example

.data

array QWORD 1,2,3,4

.code

mov esi,3 ; subscript

mov eax,array[rsi\*8] ; EAX = 4

In 64-bit mode, a pointer variable holds a 64-bit offset. In the following example, the **ptrB** vari- able holds the offset of arrayB:

.data

arrayB BYTE 10h,20h,30h,40h ptrB QWORD arrayB

Optionally, you can declare ptrB with the OFFSET operator to make the relationship clearer:

ptrB QWORD OFFSET arrayB