Lecture No.10

Lecture Outlines

4.4 Indirect Addressing

Direct addressing is rarely used for array processing because it is impractical to use constant offsets 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 addressing, it is called an *indirect operand*.

4.4.1 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 location pointed to by the register. In the following example, the contents of the BL register are copied 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 "operand 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]
```

4.4.2 Arrays

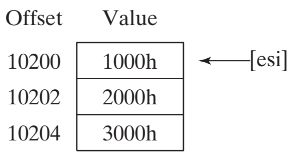
Indirect operands are ideal tools for stepping through arrays. In the next example, **arrayB** contains 3 bytes. As ESI is incremented, it points to each byte, in order:

```
.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
```

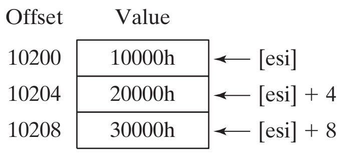
Suppose **arrayW** is located at offset 10200h. The following illustration shows the initial value of ESI in relation to the array data:



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:



4.4.3 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 examples 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 constant 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 subscript (3) by 4 (the size of a doubleword) to generate the offset of the array element containing 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
```

4.4.4 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 implementations 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 segment. 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
```

4.5 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 conditional 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.

4.5.1 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.

4.5.2 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 decremented 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 *destination*. Otherwise, if ECX equals zero, no jump takes place, and control passes to the instruction 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:

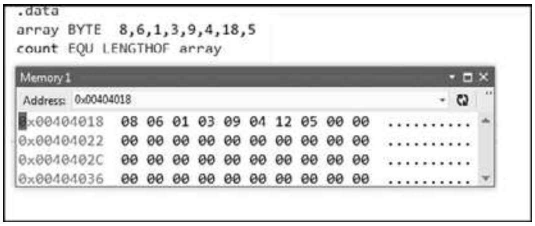
```
.data
count DWORD ?
.code
     mov   ecx,100              ; set outer loop count
L1:
     mov   count,ecx            ; save outer loop count
     mov   ecx,20               ; set inner loop count
L2:
     .
     .
     loop  L2                   ; repeat the inner loop

     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 algorithm you're using requires deep loop nesting, move some of the inner loops into subroutines.

4.5.3 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 workspace. 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.

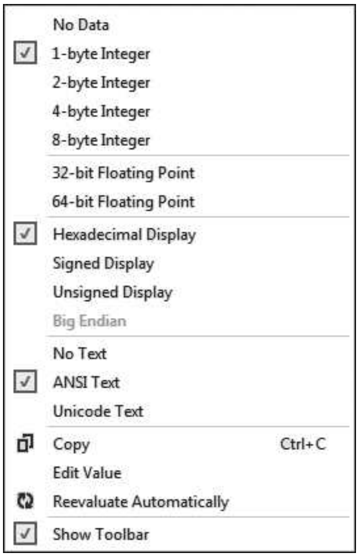


```
.data
array BYTE  8,6,1,3,9,4,18,5
count EQU LENGTHOF array
```

```
0x00404018  08 06 01 03 09 04 12 05 00 00  ..........
0x00404022  00 00 00 00 00 00 00 00 00 00  ..........
0x0040402C  00 00 00 00 00 00 00 00 00 00  ..........
0x00404036  00 00 00 00 00 00 00 00 00 00  ..........
```

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 integer (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.



4.5.4 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
```

4.5.5 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 terminator value. Indexed addressing works well for this type of operation because the same index register 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.

4.6 64-Bit Programming

4.6.1 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 destination 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.

4.6.2 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
```

4.6.3 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 operation 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, subtracting 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** variable 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
```