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
```

```
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:

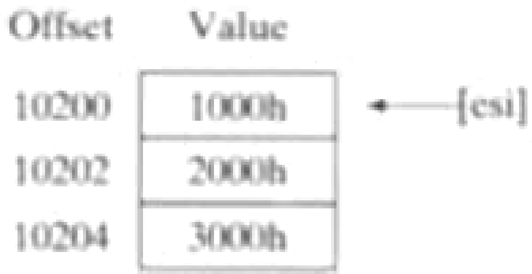
```
.data
```

```
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:

Offset	Value	
10200	10000h	← [esi]
10204	20000h	← [esi] + 4
10208	30000h	← [esi] + 8

4.4.3 Indexed Operands

An *indexed operand* adds a constant to a register to generate an effective address. Any of the

Offset	Value	
10200	10000h	← [esi]
10204	20000h	← [esi] + 4
10208	30000h	← [esi] + 8

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:

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]
```

```
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 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:

```
include irvine32.inc
.data

array dword 10h,20h,30h,40h

.code
 main proc

        mov esi,offset array
        mov eax,[esi]
        call writehex
        mov al," "
        call writechar
        add esi,4
        mov eax,[esi]
        call writehex
        mov al," "
        call writechar
        add esi,4
        mov eax,[esi]
        call writehex
```

TextPad - [Document1 *]

```
        mov eax,[esi]
        call writehex
        mov al," "
        call writechar
        add esi,4
        mov eax,[esi]
        call writehex
        mov al," "
        call writechar
        add esi,4
        mov eax,[esi]
        call writehex
        mov al," "
        call writechar
        add esi,4
        mov eax,[esi]
        call writehex
        call crlf
exit
main endp
end main
```

C:\WINDOWS\system32\cmd.exe

```
00000010 00000020 00000030 00000040
Press any key to continue . . .
```