Computer Organization & Assembly Language

BS (CS) \_Fall\_2024

**Lab\_5 Manual**



Learning Objectives:

1. Offset
2. Direct and Indirect Addressing
3. Ptr

## Indirect Addressing:

Indirect addressing is used when we need to fetch a value from a memory location. It is used in case of arrays and in case the variables are saved in memory, and we do not know the memory location. So firstly, the memory location is known and is saved in ‘si’ or ‘di’ registers and then value from that memory location is fetched.

*var db 5*

*mov si, offset var*

*mov al, [si]*

*arr\_1 db 10h, 20h, 30h, 40h, 50h*

*mov si, offset arr\_1*

*mov al, [si+2]*

## 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.

OFFSET 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 = 00404000

mov esi,OFFSET wVal ; ESI = 00404001

mov esi,OFFSET dVal ; ESI = 00404003

mov esi,OFFSET dVal2 ; ESI = 00404007

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

## 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 variable using a size attribute that’s different from the one used to declare the variable. 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 operand 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

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

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 values 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 operator makes this possible:

.data

wordList WORD 5678h,1234h

.code

mov eax,DWORD PTR wordList ; EAX = 12345678h

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 beginning of the array:

mov esi, pArray

## ALIGN Directive:

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

ALIGN bound

Bound can be 1, 2, 4, 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, aparagraph 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.

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 ? ; 00404000

ALIGN 2

wVal WORD ? ; 00404002

bVal2 BYTE ? ; 00404004

ALIGN 4

dVal DWORD ? ; 00404008

dVal2 DWORD ? ; 0040400C

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

offset 00404008.

Tasks

1) **Complex Indirect Addressing:**

Define an array complexArray of 3 WORD elements: 0x1234, 0x5678, 0x9ABC.

Define a pointer indexPtr as a DWORD to hold an index value.

Use indexPtr to calculate the address of the second element in complexArray and load this value into AX.

2) **Calculating Array Offsets:**

Define a WORD array data of 6 elements with arbitrary values.

Calculate the address of the 4th element in data and move this address into SI.

3) **Aligning Data with Padding:**

Define a BYTE variable, followed by a DWORD variable, ensuring the DWORD is aligned on a 4-byte boundary.

Insert the necessary alignment directives and determine the actual memory offsets.

4) **Dynamic Pointer Arithmetic:**

Define a DWORD array dynamicArray of 3 elements: 0x10000000, 0x20000000, 0x30000000.

Create a pointer variable dynamicPtr that initially points to the first element.

Move all element’s value into EAX using pointer arithmetic and copy it another array name *copyarray*

5) **Combining Direct and Indirect Addressing:**

Define a BYTE variable baseValue initialized to 0xFF and a WORD variable offsetValue initialized to 0x0020.

Use indirect addressing to compute the final address by adding offsetValue to baseValue's address and move the resultant value into AL.