Indexed Addressing

==

Indexed Addressing

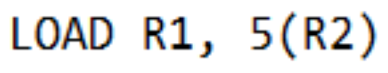
==

Indexed addressing is a memory addressing mode in which the **address of the operand is calculated by adding a constant to the contents of an index register.**

The index register is a special register in the CPU that is used to store a value that is used to offset the address of the operand.

Indexed addressing is useful for accessing elements of an array.

For example, the following assembly language instruction loads the element at index 5 of the array array to the register EAX:



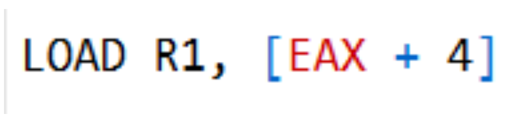
```
LOAD R1, 5(R2)
```

The actual location of an operand which is stored in memory is the **effective address,** which can be calculated using several addressing modes.

This instruction calculates the **effective address** of the operand by adding the constant 5 to the contents of the index register R2. The value in the register R2 is the offset of the array array.

Indexed addressing can also be used to access other data structures, such as linked lists and trees.

For example, the following assembly language instruction loads the next element in a linked list to the register EAX:

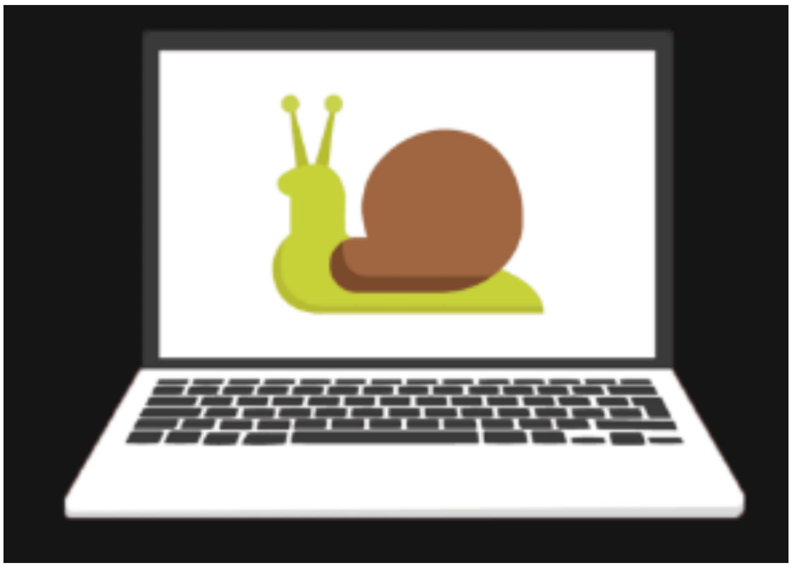


```
LOAD R1, [EAX + 4]
```

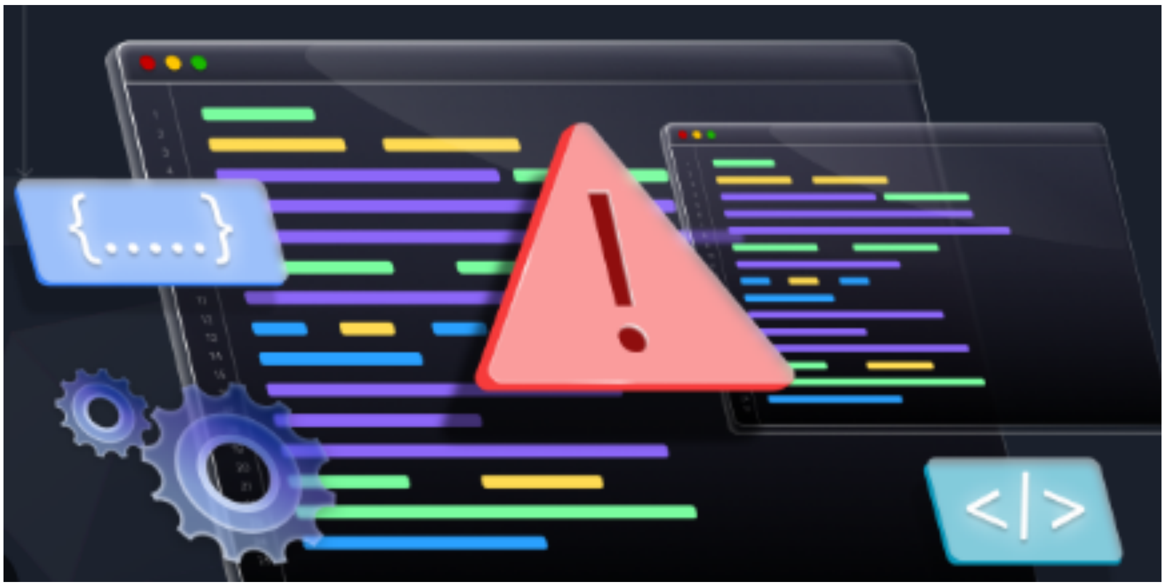
The value in the register EAX is the address of the current element in the linked list. So, this instruction calculates the effective address of the operand by adding the constant 4 to the contents of the register EAX.

Indexed addressing is a powerful and versatile addressing mode, but it is important to be aware of its limitations.

One limitation is that indexed addressing can cause the **code to be slower** than direct addressing, because the CPU has to perform an extra step to calculate the effective address of the operand.



Another limitation of indexed addressing is that it can **lead to errors** if the index register is not initialized properly or if it contains an invalid value



Indexed Operands

Indexed operands are a way to access memory locations using a register and a constant. The register is called the index register, and the constant is called the displacement. The effective address of the operand is calculated by adding the displacement to the value of the index register.

Indexed operands are useful for accessing elements of an array, because they allow you to access any element of the array without having to know the address of the element in advance. For example, the following assembly language instruction loads the element at index 5 of the array array to the register EAX:

```
;Load the element at index 5 of the array `array` to the register `EAX`.
LOAD R1, 5(R2)

;Load the next element in a linked list to the register `EAX`.
LOAD R1, [EAX + 4]

;Store the value of the register `EAX` to the memory location
;at the offset 100 from the base address stored in the register `R4`.
STORE R1, 100(R4)
```

The following C code uses indexed operands to access an element of an array:

```
int main() {
  int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int index = 5;

  ;Load the element at index `index` of the array `array` to the register `EAX`.
  int element = array[index];

  ;Return the value in the register `EAX`.
  return element;
}
```

Indexed Operands with Scale Factors Part 2

Indexed operands in assembly language allow you to add a constant offset to a register to calculate an effective memory address. This addressing mode is particularly useful for working with arrays.

Here, we'll delve into the concept of indexed operands with scale factors, which provides flexibility when accessing array elements of different sizes.

Basic Indexed Operand Forms: Indexed operands can take two notational forms, both utilizing a register and potentially a constant:

```
constant[reg]
[constant + reg]
```

The first form combines a variable name with a register, and the assembler translates the variable's name into a constant representing its offset. Here are examples of both notational forms:

```
.data
    arrayB BYTE 10h, 20h, 30h

.code
    mov esi, 0
    mov al, arrayB[esi]    ; AL = 10h
    mov al, [arrayB + esi]
```

The code you provided is intended to access and retrieve the value of the first element in an array called arrayB, which is defined in the .data section as an array of bytes with the values 10h, 20h, and 30h. Let's break down what each line does:

.data: This section defines the data for the program. Here, you've defined an array named arrayB of type BYTE, containing three values: 10h, 20h, and 30h.

.code: This section contains the executable code for your program.

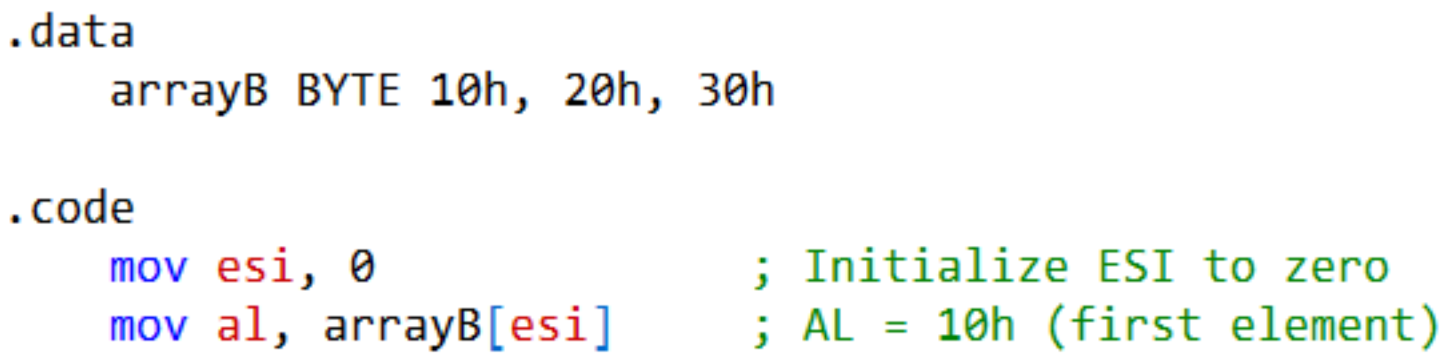
mov esi, 0: This instruction initializes the esi register to zero. The esi register is often used as an index or pointer in assembly language.

mov al, arrayB[esi]: Here, you're using indexed addressing to access the value of the first element of the arrayB. The [esi] part is an index operation that says, **"Access the element in arrayB that's at the memory location specified by the value in esi."** Since you previously set esi to zero, this effectively retrieves the value at the beginning of the arrayB, which is 10h.

mov al, [arrayB + esi]: This line does the same thing as the previous line. It retrieves the value at the memory location specified by arrayB + esi.

So, in summary, these instructions initialize esi to zero and then use it as an index to access and load the value 10h from the first element of the arrayB into the al register. After this code is executed, al will contain the value 10h.

Initializing the Index Register: When working with indexed operands, it's crucial to initialize the index register to zero before accessing the first element in the array. This allows you to start at the beginning of the array.



```
.data
    arrayB BYTE 10h, 20h, 30h

.code
    mov esi, 0              ; Initialize ESI to zero
    mov al, arrayB[esi]     ; AL = 10h (first element)
```

Adding Displacements: The second type of indexed addressing involves combining a register with a constant offset. Here, the index register holds the base address of an array or structure, and the constant specifies offsets to various array elements. Consider this example with a 16-bit word array:

```
.data
    arrayW WORD 1000h, 2000h, 3000h

.code
    mov esi, OFFSET arrayW
    mov ax, [esi]          ; AX = 1000h (first element)
    mov ax, [esi + 2]      ; AX = 2000h (second element)
    mov ax, [esi + 4]      ; AX = 3000h (third element)
```

Scale Factors in Indexed Operands

To work with indexed operands more flexibly, especially when the size of array elements varies, you can use scale factors. A scale factor is the size of each array component (e.g., word = 2, doubleword = 4).

Here's an example with a doubleword array, where we multiply the subscript (3) by the scale factor (4) for doublewords to access the element containing 400h:

```
.data
    arrayD DWORD 100h, 200h, 300h, 400h

.code
    mov esi, 3 * TYPE arrayD ; Calculate the offset for arrayD[3]
    mov eax, [arrayD + esi]  ; EAX = 400h
```

3 * 4bytes = 12

0 + 12(esi) = 12

eax = 12 = 400h's location

----

1. You're correct that 3 multiplied by 4 (bytes) equals 12. This multiplication takes into account the size of each array element.

2. Setting esi to 12 means that you're trying to access the element

in the array that is 12 bytes from the start of the array. However, it's important to note that in assembly language, the array elements are typically indexed starting from zero. So, arrayD[0] is the first element, arrayD[1] is the second element, and so on.

3. When you set esi to 12, you're effectively trying to access arrayD[3], not arrayD[0]. If you want to access the element containing 400h, you should set esi to 0, not 12, because arrayD[0] is the first element in the array.

So, with esi set to 0, you'll access the first element of arrayD, which indeed contains 100h, not 400h. If you want to access arrayD[3] (the element with 400h), you would set esi to 12.

With esi set to 12, you are correctly accessing the fourth element of arrayD, which contains 400h. The code loads the value 400h into the eax register. Thank you for pointing out the correct setting of esi, and I appreciate your patience.