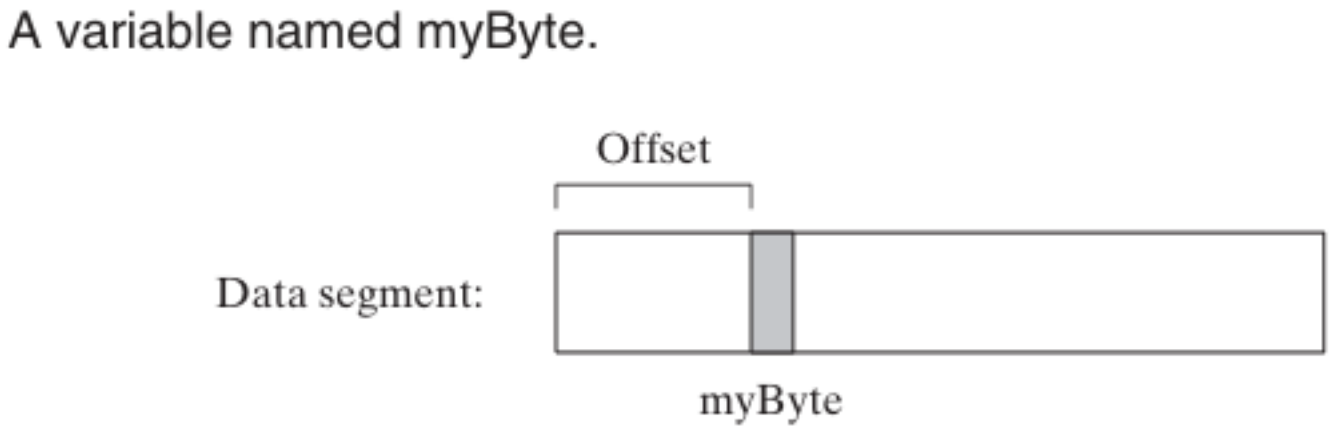
Offset



The image you sent shows a variable called myByte, an **offset,** and the data segment. The data segment is a section of memory that stores program data, such as variables, constants, and strings. The offset is the distance from the beginning of the data segment to the variable myByte.

Bruh, am offset, you add me to the data segment's starting address

For example, if the data segment starts at memory address 1000h and the offset of myByte is 200h, then myByte will be stored at memory address 1200h.

address 1200h.

The offset is used by the processor to access the variable myByte. When the processor needs to access myByte, it adds the offset to the starting address of the data segment to get the memory address of myByte.

The data segment is also used by the operating system to manage program data. The operating system keeps track of the starting address and size of the data segment for each program. This allows the operating system to protect the data of one program from being accessed by another program.

Here is an example of how to access the variable myByte in assembly language:

```
mov ax, data_segment ; Load the starting address of the data segment into the AX register
add ax, offset_myByte ; Add the offset of myByte to the AX register
mov al, [ax] ; Load the value of myByte into the AL register
```

The mov instruction is used to move data from one register to another or between a register and memory. The ax register is a general-purpose register that can be used to store any type of data, including integers, floating-point numbers, and addresses. **The al register is a lower 8-bit register of the ax register.**

The first line of code loads the starting address of the data segment into the ax register. The second line of code adds the offset of myByte to the ax register. This gives us the memory address of myByte. The third line of code loads the value of myByte from memory into the al register.

Is the statement that offset said in the image correct?

Yes, he is correct. The offset is added to the data segment's starting address to get the memory address of a variable in the data segment. This is how the processor addresses variables in the data segment.

For example, if the data segment's starting address is 1000h and the offset of a variable is 200h, then the memory address of the variable will be 1200h.

The **offset is a 16-bit value,** which means that it can **address up to 64KB of memory.** This is the **maximum size of a data segment in the 8086** microprocessor.

The offset is used in many different instructions, such as the **mov instruction** and the **lea instruction.** The **mov instruction** is used to move data from one register to another or between a register and memory. The **lea instruction** is used to load the effective address of an operand into a register.

The offset is an important concept in assembly language programming, and it is essential to understand how it is used to access variables in the data segment.

```
mov ax, data_segment ; Load the starting address of the data segment into the AX register
add ax, offset_myByte ; Add the offset of myByte to the AX register
mov al, [ax] ; Load the value of myByte into the AL register
```

The code above is dereferencing the ax register to get the value of the variable myByte, which is located at the address: **data_segment's address + offset.**

The mov instruction in the code loads the starting address of the data segment into the ax register. The add instruction adds the offset of myByte to the ax register. This gives us the memory address of myByte. The mov al, [ax] instruction loads the value of the variable myByte from memory into the al register.

The [ax] notation is used to dereference the ax register. This means that the processor will load the value from the memory address stored in the ax register.

The following is a step-by-step explanation of how the code works:

1. The mov ax, data_segment instruction loads the starting address of the data segment into the ax register.

2. The add ax, offset_myByte instruction adds the offset of myByte to the ax register. This gives us the memory address of myByte.

the ax register. This gives us the memory address of myByte.

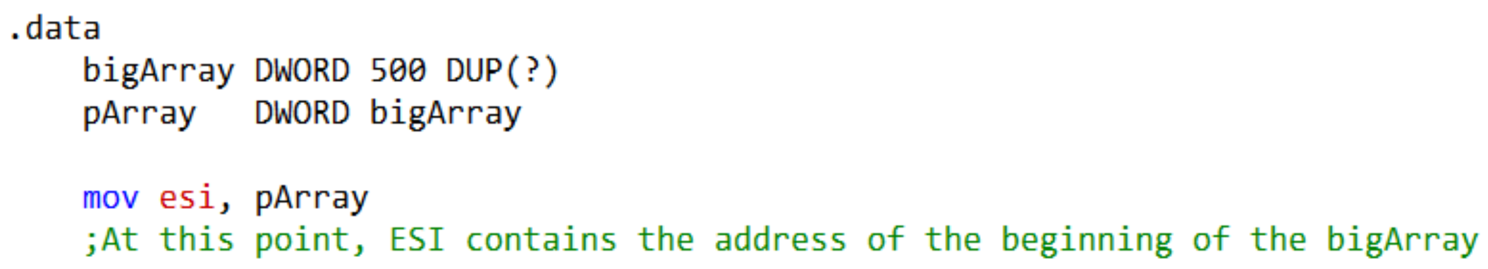
3. The mov al, [ax] instruction loads the value of the variable myByte from memory into the al register.

After the code has executed, the al register will contain the value of the variable myByte.

===

How offset works in ASM:

===



```
.data
    bigArray DWORD 500 DUP(?)
    pArray   DWORD bigArray

    mov esi, pArray
    ;At this point, ESI contains the address of the beginning of the bigArray
```

The line **pArray DWORD bigArray** in assembly language is defining a doubleword (DWORD) variable named pArray and initializing it with the address of the bigArray array. Essentially, it creates a pointer variable (pArray) that points to the beginning of the bigArray array.

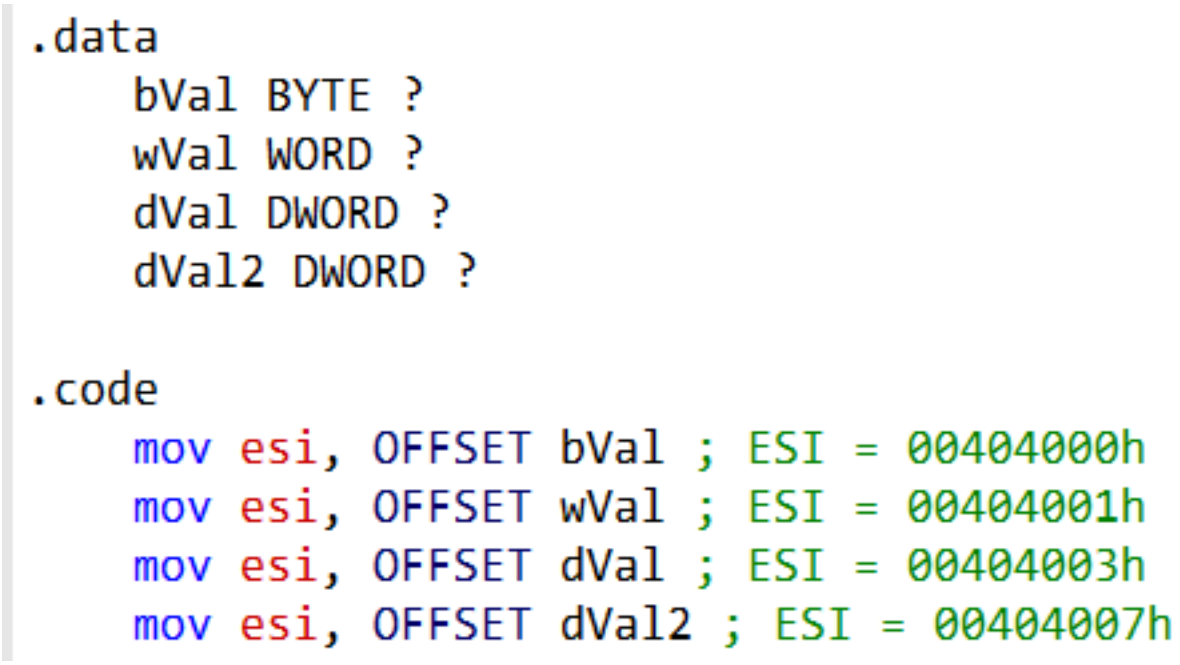
bigArray contains 500 doubleword(DWORD) elements.

The OFFSET operator can also be used to access variables in memory. To access a variable in memory using the OFFSET operator, you simply add the offset of the variable to the starting address of the data segment.

For example, the following code accesses the variable bVal in memory:

```
.data
    bVal BYTE ?
    mov esi, OFFSET bVal
    ; At this point, ESI contains the address of the variable bVal.
    mov al, [esi]
    ; At this point, AL contains the value of the variable bVal.
```

Dereferencing is done ie. accessing data from that address stored in esi(If you don't get this, you should go through the C programming code from my github).



```
.data
    bVal BYTE ?
    wVal WORD ?
    dVal DWORD ?
    dVal2 DWORD ?

.code
    mov esi, OFFSET bVal ; ESI = 00404000h
    mov esi, OFFSET wVal ; ESI = 00404001h
    mov esi, OFFSET dVal ; ESI = 00404003h
    mov esi, OFFSET dVal2 ; ESI = 00404007h
```

After the code has executed, the ESI register will contain the offset of the variable that was specified in the MOV instruction.

For example, after the first MOV instruction has executed, the ESI register will contain the offset of the variable bVal. This offset is 00404000h, because that is where the variable bVal is located in memory.

After the second MOV instruction has executed, the ESI register will contain the offset of the variable wVal. This offset is 00404001h, because that is where the variable wVal is located in memory.

And so on.

The OFFSET operator is a useful tool for accessing variables in memory. It is also useful for initializing pointers.

```
.data
    myArray WORD 1,2,3,4,5

.code
    mov esi, OFFSET myArray + 4
```

The first line of code in the data segment declares an array called myArray that contains five 16-bit words. The second line of code in the code segment moves the offset of myArray plus 4 to the ESI register.

The OFFSET operator returns the offset of a variable or label. The offset is the distance from the beginning of the data segment to the variable or label. In this case, the offset of myArray is 0000h, because myArray is the first variable in the data segment.

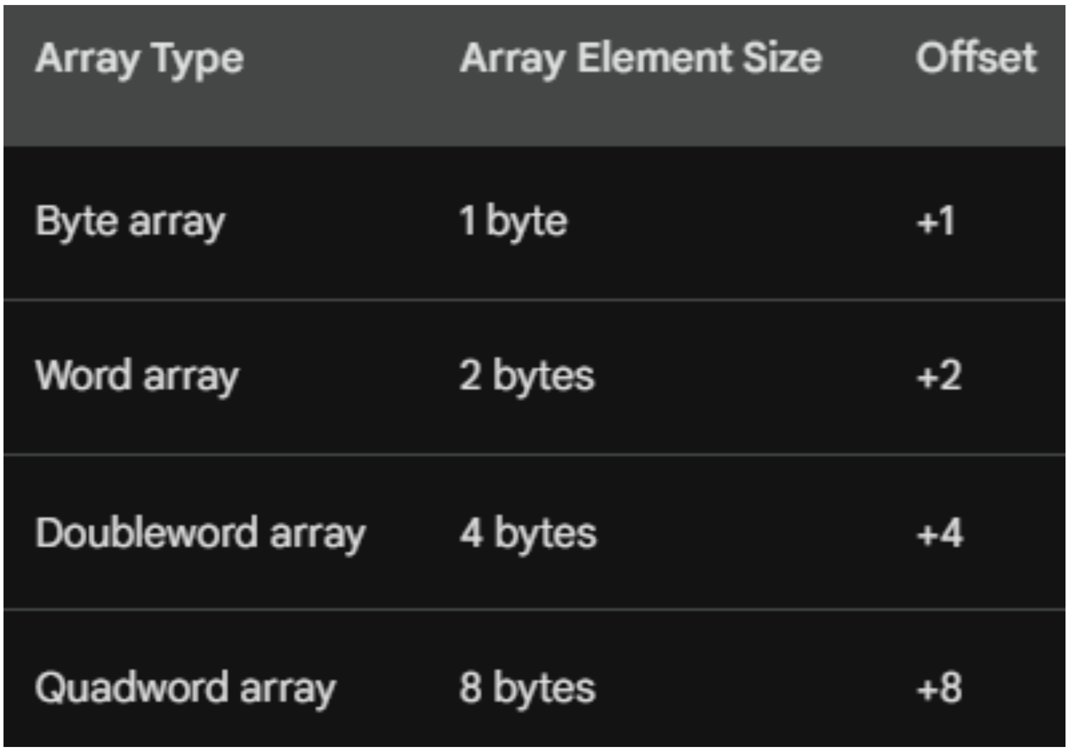
Adding 4 to the offset of myArray gives us the offset of the third integer in the array. This is because the first integer in the array is at offset 0000h, the second integer is at offset 0002h, and the third integer is at offset 0004h.

The ESI register is a general-purpose register that can be used to store any type of data, including addresses. By moving the offset of the third integer in myArray to the ESI register, we are essentially creating a pointer to the third integer in the array.

The ESI register can now be used to access the third integer in the array. For example, the following code would load the value of the third integer in myArray into the AL register:

```
mov al, [esi]
```

***Calculating offsets* for arrays**



Array Type	Array Element Size	Offset
Byte array	1 byte	+1
Word array	2 bytes	+2
Doubleword array	4 bytes	+4
Quadword array	8 bytes	+8

BYTE Arrays:
If you are working with a BYTE array, where each element is 1 byte in size, you can access the next element by adding +1 to the current element's address.

```
; Assuming byteArray starts at address 00400000h
mov esi, OFFSET byteArray ; Load the starting address of byteArray
add esi, 1 ; Move to the second element (1 byte offset)
mov al, [esi] ; Dereference/Load the value of the second element into AL
```

WORD Arrays:
For WORD arrays, where each element is 2 bytes (16 bits) in size, you should add +2 to move to the next element's address.

```
; Assuming wordArray starts at address 00400000h
mov esi, OFFSET wordArray ; Load the starting address of wordArray
add esi, 2 ; Move to the third element (2 bytes offset)
mov ax, [esi] ; Load the value of the third element into AX
```

DWORD Arrays:
DWORD arrays have elements that are 4 bytes (32 bits) in size. To access the next element, add +4 to the current element's address.

```
; Assuming dwordArray starts at address 00400000h
mov esi, OFFSET dwordArray ; Load the starting address of dwordArray
add esi, 4 ; Move to the fourth element (4 bytes offset)
mov eax, [esi] ; Load the value of the fourth element into EAX
```

QWORD Arrays:
If you are dealing with QWORD arrays, where each element is 8 bytes (64 bits) in size, you should add +8 to move to the address of the next element.

```
; Assuming qwordArray starts at address 00400000h
mov esi, OFFSET qwordArray ; Load the starting address of qwordArray
add esi, 8 ; Move to the fifth element (8 bytes offset)
mov rax, [esi] ; Load the value of the fifth element into RAX
```