

Type, LengthOF, SizeOF, Label

The TYPE operator returns the size, in bytes, of a single element of a variable.

For example, the TYPE of a byte is 1, the TYPE of a word is 2, the TYPE of a doubleword is 4, and the TYPE of a quadword is 8. Here are examples of each:

```
.data
    var1 BYTE ?
    var2 WORD ?
    var3 DWORD ?
    var4 QWORD ?
```

;The following table shows the value of each TYPE expression.

Expression	Value
TYPE var1	1
TYPE var2	2
TYPE var3	4
TYPE var4	8

=====

LENGTHOF Operator:

=====

The **LENGTHOF operator** counts the number of elements in an array.

It also takes into account nested DUP operators that multiply the counters.

If you declare an array that spans multiple program lines, LENGTHOF only regards the data from the first line as part of the array.

```
.data
    byte1 BYTE 10,20,30
    array1 WORD 30 DUP(?)
    array2 WORD 5 DUP(3 DUP(?))
    array3 DWORD 1,2,3,4
    digitStr BYTE "12345678",0
```

; The following table lists the values returned by each LENGTHOF expression:

Expression	Value
LENGTHOF byte1	3
LENGTHOF array1	30
LENGTHOF array2	5 * (3 * 3)
LENGTHOF array3	4
LENGTHOF digitStr	9

```
.data
    byte1 BYTE 10,20,30
    array1 WORD 30 DUP(?)
    array2 WORD 5 DUP(3 DUP(?))
```

```
.code
    mov ax, LENGTHOF byte1 ; AX = 3
    mov ax, LENGTHOF array1 ; AX = 30
    mov ax, LENGTHOF array2 ; AX = 15
```

=====

SizeOF Operator:

=====

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE. For example:

```
.data
    intArray WORD 32 DUP(0)

.code
    mov eax, sizeof intArray ; EAX = 32 * 2 = 64
```

`intArray` is indeed an array of 32 words. Each word is 16 bits or 2 bytes in size. Therefore, the correct calculation for `sizeof intArray` should be:

`sizeof intArray = 32 * 2 = 64`

So, the correct value for `EAX` should be 64.

=====

Label Directive:

=====

The `LABEL` directive lets you insert a label and give it a size attribute without allocating any storage. It's commonly used to provide an alternative name and size attribute for the variable declared next in the data segment.

```
.data
    val16 LABEL WORD
    val32 DWORD 12345678h

.code
    mov ax, val16 ; AX = 5678h
    mov dx, [val16+2] ; DX = 1234h
```

In this example, `val16` serves as an alias for the same storage location as `val32`. The `LABEL` directive itself allocates no storage and is used to provide an alternate name and size for the variable.

```

.data
    val16 LABEL WORD
    val32 DWORD 12345678h

    mov ax, val16 ; AX = 5678h
    mov dx, [val16+2] ; DX = 1234h

    LongValue LABEL DWORD
    val1 WORD 5678h
    val2 WORD 1234h

    mov eax, LongValue ; EAX = 12345678h

```

The LABEL directive allows you to create a label and assign it a size attribute without actually allocating any storage. This can be useful for a few different things:

- Creating aliases for variables: You can use a LABEL directive to create a different name for a variable, with a different size attribute. This can be useful for making your code more readable and maintainable. For example, the following code creates a label named val16 that is an alias for the variable val32:

```

.data
    val16 LABEL WORD
    val32 DWORD 12345678h

.code
    mov ax, val16 ; AX = 5678h

```

In this example, the val16 label is used to access the first two bytes of the val32 variable.

- Constructing larger integers from smaller integers: You can also use a LABEL directive to construct a larger integer from two smaller

integers. For example, the following code creates a label named LongValue that represents a 32-bit integer constructed from the two 16-bit variables val1 and val2:

```
.data
    LongValue LABEL DWORD
    val1 WORD 5678h
    val2 WORD 1234h

.code
    mov eax, LongValue ; EAX = 12345678h
```

In this example, the LongValue label is used to access the 32-bit integer that is constructed from the val1 and val2 variables.

The LABEL directive does not actually allocate any storage. It simply creates a label with a specific size attribute. This can be useful for creating aliases for variables, constructing larger integers from smaller integers, and other purposes.

The LongValue label is not actually storing the values of val1 and val2. Instead, it's acting as a pointer to the memory location where val1 and val2 are stored. Here's what's happening step by step:

1. In the .data section, you declare LongValue as a LABEL DWORD, which effectively gives a name (LongValue) to a memory location that can hold a doubleword (32 bits or 4 bytes) of data.
2. You declare val1 and val2 as two 16-bit words with specific values (5678h and 1234h, respectively). These values are stored in memory in the order they appear.
3. In the .code section, you use the mov instruction to move the value at the memory location referred to by LongValue into the EAX register. This instruction effectively loads a doubleword (32 bits) from memory into EAX.

So, what's happening is that LongValue is acting as a reference or

memory address, not a variable that stores the values directly. It's common in assembly language to use labels to represent memory addresses, and by using the label as an operand in an instruction, you can access the data stored at that memory location.

In this case, when you move from LongValue to EAX, you are effectively loading the 32 bits of data starting from the memory location referred to by LongValue, which includes the values of val1 and val2 stored consecutively in memory.

The critical concept to understand is memory addressing and how the LongValue label acts as a memory reference. Here's a detailed breakdown:

Memory Allocation: In the .data section, you declare LongValue as a LABEL DWORD. This essentially reserves a 32-bit (4-byte) block of memory without assigning any specific value to it. The memory layout looks like this:

```
Memory:
[ LongValue ] [ val1 ] [ val2 ]
```

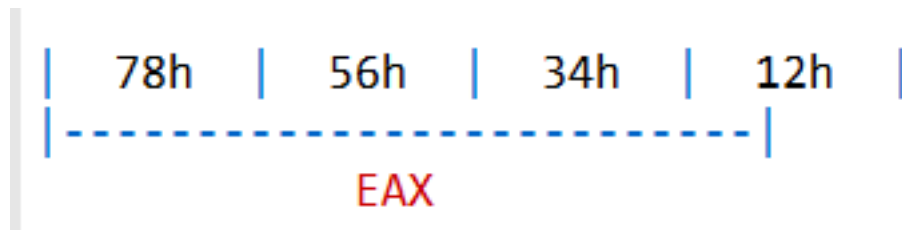
However, at this point, the values of val1 and val2 are stored in their respective memory locations.

Memory Contents: val1 is a 16-bit (2-byte) word with the value 5678h, and val2 is another 16-bit word with the value 1234h. These values are stored in memory in little-endian order, which means the least significant byte comes first:

```
Memory:
[ LongValue ] [ 78h ] [ 56h ] [ 34h ] [ 12h ]
```

Using mov Instruction: In the .code section, you use the mov instruction to move data from memory to a register. When you mov eax,

LongValue, it means you're loading a 32-bit (4-byte) value into the EAX register, starting from the memory location referred to by LongValue. In essence, you're loading the consecutive bytes from that memory location into EAX.



So, EAX gets loaded with 12345678h, which is the concatenation of the values at the memory locations referred to by LongValue, val1, and val2.

So, assembler will check and see longvalue memory locatoin pointer, has nothing and continue to load the val1 and val2 that follow as the next instructions??

Yes, that's correct. In assembly language, the assembler doesn't "know" or store values for labels like LongValue.

Instead, it treats labels as memory addresses or pointers to specific locations in memory. When you use a label in an instruction, the assembler generates machine code that loads or manipulates data from the memory location pointed to by that label.

In the case of `mov eax, LongValue`, the assembler generates machine code that tells the CPU to:

1. Go to the memory location pointed to by the label LongValue.
2. Load the data from that memory location into the EAX register.

Since LongValue doesn't have a specific value assigned to it in the data section, the assembler doesn't "see" any data there. Instead, it continues to load the data from the memory locations that follow in memory, which in this case are val1 and val2.

So, when you `mov eax, LongValue`, you effectively load the data from val1 and val2 into the EAX register because they are the consecutive memory locations following LongValue.

In summary, the LongValue label acts as a reference to a memory location, and when you move it into EAX, you effectively load the 32

bits (4 bytes) of data starting from that memory location, which includes the values of val1 and val2. This demonstrates how labels can be used to access specific memory locations in assembly language programming.

Let's go through the answers to the section review questions:

(True/False): The OFFSET operator always returns a 16-bit value.

False. The OFFSET operator returns the offset or address of a variable or label, which can be either 16-bit or 32-bit, depending on the architecture and context.

(True/False): The PTR operator returns the 32-bit address of a variable.

True. The PTR operator is used to specify the size of an operand, and when used in conjunction with a label or variable, it often implies a 32-bit address.

(True/False): The TYPE operator returns a value of 4 for doubleword operands.

True. In many assembly languages, the TYPE operator returns a value that corresponds to the size of a data type in bytes. Doubleword operands are typically 4 bytes in size.

(True/False): The LENGTHOF operator returns the number of bytes in an operand.

False. The LENGTHOF operator returns the number of elements in an array or the length of a data structure in terms of elements, not bytes.

(True/False): The SIZEOF operator returns the number of bytes in an operand.

True. The SIZEOF operator returns the size of an operand in bytes, including any padding or alignment bytes that may be present.

IGNORE THESE QUESTIONS I HAVE EXPLAINED EVERYTHING ABOVE:

How the heck do the val1 and val2 values just fly inside LongValue, without something like a register??

The values of val1 and val2 don't literally fly inside LongValue. The LongValue label is simply an alias/pointer, for the memory location that contains the 32-bit integer that is constructed from the val1 and val2 variables.

When you move the value of LongValue into the eax register, the CPU will read the 32-bit integer from the memory location that LongValue refers to.

In other words, the LongValue label is just a convenient way to refer to the memory location that contains the 32-bit integer. It does not actually store the integer itself.

Here is a diagram that shows how it works:

Memory

0x0000: 5678

0x0002: 1234

LongValue (label)

eax (register)

When you move the value of LongValue into the eax register, the CPU will read the 32-bit integer from the memory location that LongValue refers to. In this case, the CPU will read the value 0x12345678 from the memory locations 0x0000 and 0x0002.

So, why did we just define the label, define val1, define val2, then poof, LongValue contains val1+val2 ?? I don't get what Label does?? Does it concetenate or what?

The LABEL directive does not concatenate anything. It simply creates a label with a specific size attribute.

In the example we were discussing, the LongValue label is used to represent the 32-bit integer that is constructed from the two 16-bit variables val1 and val2.

This is possible because the assembler knows the size of the val1 and val2 variables, and it knows that they are contiguous in memory.

The assembler can therefore use this information to calculate the address of the 32-bit integer that is constructed from the two 16-bit variables.

When you move the value of LongValue into the eax register, the CPU will read the 32-bit integer from the memory location that LongValue refers to. In this case, the CPU will read the value 0x12345678 from the memory locations 0x0000 and 0x0002.

Here is a more detailed explanation of what happens when you assemble and link the code:

1. The assembler creates a symbol table. The symbol table is a list of all the labels and variables in the code, along with their addresses.
2. The assembler generates machine code for the code. The machine code contains instructions for the CPU to load the values of the variables val1 and val2 into registers, and then add the two values together.
3. The linker combines the machine code for the code with the machine code for any libraries that the code depends on.
4. The linker creates an executable file. The executable file contains the machine code for the code, as well as the symbol table.

When you run the executable file, the CPU will load the machine code into memory. The CPU will then use the symbol table to resolve the addresses of all the labels and variables in the code.

Once the CPU has resolved the addresses of all the labels and

variables, it can start executing the machine code. The machine code will load the values of the variables val1 and val2 into registers, and then add the two values together.

The result of the addition will be stored in a register. The CPU will then use the MOV instruction to move the value of the register to the memory location that the LongValue label refers to.

So, to answer your question, the LABEL directive does not concatenate anything. It simply creates a label with a specific size attribute. The assembler then uses this information to calculate the address of the memory location that the label refers to.

Yes, your understanding is correct. The assembler will treat the LongValue label as a memory address, and when you use the mov eax, LongValue instruction, the assembler will generate machine code that tells the CPU to load the data from the memory location pointed to by the LongValue label into the EAX register.

Since the LongValue label does not have a specific value assigned to it in the data section, the assembler will continue to load the data from the memory locations that follow in memory, which in this case are the val1 and val2 variables.

So, when you execute the mov eax, LongValue instruction, the EAX register will contain the value 0x12345678, which is the 32-bit integer that is constructed from the two 16-bit variables val1 and val2.

Labels can be used to access specific memory locations in assembly language programming. This can be useful for a variety of purposes, such as accessing data structures, passing arguments to functions, and returning values from functions.