1) LEA vs MOV

----------

LEA means Load Effective Address

* MOV means Load Value

In short, LEA loads a pointer to the item you're addressing whereas MOV loads the actual value at that address.

The purpose of LEA is to allow one to perform a non-trivial address calculation and store the result [for later usage]

LEA ax, [BP+SI+5] ; Compute address of value

MOV ax, [BP+SI+5] ; Load value at that address

Where there are just constants involved, MOV (through the assembler's constant calculations) can sometimes appear to overlap with the simplest cases of usage of LEA. Its useful if you have a multi-part calculation with multiple base addresses etc.

Given the following code:

L1 db "word", 0

mov al, [L1]

mov eax, L1

What do the brackets ([L1]) represent?

[L1] means the memory contents at address L1. After running mov al, [L1] here, The al register will receive the byte at address L1 (the letter 'w').

Operands of this type, such as [ebp], are called [memory operands](http://www.imada.sdu.dk/Courses/DM18/Litteratur/IntelnATT.htm).

I see that none tells about the caveat in following this as a rigid rule - if brackets, then dereference, except when it's the lea instruction.

lea is an exception to the above rule. Say we've

mov eax, [ebp - 4]

The value of ebp is subtracted by 4 and the brackets indicate that the resulting value is taken as an address and the value residing at that address is stored in eax. However, in lea's case, the brackets wouldn't mean that:

lea eax, [ebp - 4]

The value of ebp is subtracted by 4 and the resulting value is stored in eax. This instruction would just calculate the address and store the calculated value in the destination register. See [this post](https://stackoverflow.com/a/1699778/183120) for further details.

2)

Nasm syntax:

mov eax, var == lea eax, [var] ; i.e. mov r32, imm32

lea eax, [var+16] == mov eax, var+16

lea eax, [eax\*4] == shl eax, 2 ; but without setting flags

3)

-fPIE

-fPIC

|  |  |  |
| --- | --- | --- |
| DB | Define Byte | allocates 1 byte |
| DW | Define Word | allocates 2 bytes |
| DD | Define Doubleword | allocates 4 bytes |
| DQ | Define Quadword | allocates 8 bytes |
| DT | Define Ten Bytes | allocates 10 bytes |

* 1 byte (8 bit): byte, DB, RESB
* 2 bytes (16 bit): word, DW, RESW
* 4 bytes (32 bit): dword, DD, RESD
* 8 bytes (64 bit): qword, DQ, RESQ
* 10 bytes (80 bit): tword, DT, REST
* 16 bytes (128 bit): oword, DO, RESO, DDQ, RESDQ
* 32 bytes (256 bit): yword, DY, RESY
* 64 bytes (512 bit): zword, DZ, RESZ

4) Important notices

* syscall

Instead of pushing a return address onto the kernel stack (like int 0x80 does), syscall:

sets RCX=RIP, R11=RFLAGS (so it's impossible for the kernel to even see the original values of those regs before you executed syscall).

* Using CMP instruction

#### Syntax

CMP destination, source

CMP compares two numeric data fields. The destination operand could be either in register or in memory. The source operand could be a constant (immediate) data, register or memory.

example:

cmp “a”, [rax] ; this in not valid, because the first operand must be a register of memory

cmp [rax], “a” ; OK

* gcc vs ld for linking

gcc using “main” as entry point

ld using “\_start” as entry point

gcc will generate sys\_exit for the assembly program (the binary size will be bigger)

In case of using ld, sys\_exit must be at the end of the program, otherwise segfault will occor. (RIP will use the next memory address and treats as an instruction, but it is usually a memory garbage)

gdb printout example without sys\_exit:

0x0000000000401020 e8 db ff ff ff ? call 0x401000 <function>

0x0000000000401025 b8 00 00 00 00 ? mov eax,0x0

0x000000000040102a 5d ? pop rbp ; last valid instruction

0x000000000040102b 00 00 ? add BYTE PTR [rax],al ; this is memory garbage

0x000000000040102d 00 00 ? add BYTE PTR [rax],al

5) Little Endian (example gdb)

This **only** matters when writing **multi-byte** quantities to memory and reading them differently (e.g., byte per byte)

>>> x/12xg $rsp

0x7fffffffe7d8: 0x0000000000000000 0x0000000000000001

0x7fffffffe7e8: 0x00007fffffffeaa9 0x0000000000000000

0x7fffffffe7f8: 0x00007fffffffeaef 0x00007fffffffeaff

0x7fffffffe808: 0x00007fffffffeb51 0x00007fffffffeb65

0x7fffffffe818: 0x00007fffffffeb7e 0x00007fffffffeb94

0x7fffffffe828: 0x00007fffffffebc1 0x00007fffffffebef

>>> x/32xb $rsp

0x7fffffffe7d8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

0x7fffffffe7e0: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00

0x7fffffffe7e8: 0xa9 0xea 0xff 0xff 0xff 0x7f 0x00 0x00

0x7fffffffe7f0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

>>>

**Example:**

What is the layout and the content of the data memory segment on a Little Endian

machine? Byte per byte, in hex

|  |  |  |  |
| --- | --- | --- | --- |
| section .data: | | | |
| pixels | times 4 | db | 0FDh |
| x | dd | 0001**0111**0011**0110**0001**0101**1101**0011b** |  |
| blurb | db | “ad”, “b”, “h”, 0 |  |
| buffer | times 10 | db | 14o |
| min | dw | -19 |  |

pixels: 0xFD, 0xFD, 0xFD, 0xFD

x: 0001**0111**0011**0110**0001**0101**1101**0011b** = 0x173615D3h (dd→ 4byte)

blurb: “ad”, “b”, “h”, 0 = **0x6164**, 0x62, 0x68, 0x0 (db →1byte, not a multi-byte!!!)

buffer: 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C

min: -19 = 0xFFED (dw → 2byte)

FD,FD,FD,FD,D3,15,36,17,**61**,**64,** 62,68,0,0C, 0C, 0C, 0C, 0C, 0C, 0C, 0C, 0C, 0C**,** **ED, FF**

>>> objdump -s -j .data a.out

a.out: file format elf64-x86-64

Contents of section .data:

402000 fdfdfdfd d3153617 61646268 000c0c0c ......6.adbh....

402010 0c0c0c0c 0c0c0ced ff .........

>>> objdump -s -j .data little.o

little.o: file format elf64-x86-64

Contents of section .data:

0000 fdfdfdfd d3153617 61646268 000c0c0c ......6.adbh....

0010 0c0c0c0c 0c0c0ced ff .........

**Functions**:

; call = push(RIP) + JMP

; ret = pop(RIP) + JMP