OllyDBG's disassembled syntax and c-equivalent

This is probably a pretty simple question as I'm not too used to how the syntax looks for OllyDBG's disassembler.

Does this following assembler statement:

```
MOV EAX, DWORD PTR [ESI + 14]
```

Be roughly translated to this C code:

```
eax = *(esi + 0x14);
```

Have I understood the syntax correctly or am I misunderstanding this?

disassembly assembly x86 ollydbg

edited Jul 9 '13 at 11:33

perror
5,817 4 27 84

asked Jul 8 '13 at 16:15

lfxgroove
189 1 10

1 You're probably looking for `DWORD eax = *(DWORD *)((char *)esi + 0x14)`. (I am not sure, but Olly probably shows offsets in hex, not decimal, by default.) – DCoder Jul 8 '13 at 16:22

So eax will contain the value of what is pointed to by `esi + 0x14`? – lfxgroove Jul 8 '13 at 17:30

Yes, and `DWORD PTR` means it will take four bytes starting with that address. – DCoder Jul 8 '13 at 17:32

1 Ah, thanks! But this later on could be used as a pointer if that's what you'd like? Would you mind turning this into an answer so i have something to accept? :) – lfxgroove Jul 8 '13 at 17:38

In your question, eax = &(esi + 0x14); should be eax = *(esi + 0x14);. It's dereferencing it, not referencing it. – MMavipc Jul 9 '13 at 10:21

2 Answers

The `DWORD PTR [expression]` syntax means "take the value of `expression`, interpret it as an address, and access 4 (size of a `DWORD`) bytes starting with that address". But assembly data types are rather different from those of C, so many C types can be accessed this way.

The instruction you provided is basically equivalent to C code:

```
typedef dword_t ...;

dword_t eax = *(dword_t *)((char *)esi + 0x14);
```

This instruction can be used to access 4 contiguous bytes no matter what the C type of those bytes is - in the line above, you could (on a 32 bit system) define `dword_t` as `int`, `float`, `void *` or another type of the appropriate size, and it would still work the same way, it's just bits and bytes travelling from one place to another. With a reasonably smart compiler, this can even be used to read entire structs or arrays in one step, as long as their length is small enough.

> But this later on could be used as a pointer if that's what you'd like?

As I said, it is not possible to say what the original C type of those bytes is just from this context. You have to look at other places where this value is used and look for indicators of the specific type. If you see it used in `[eax]` or a similar expression - it's probably a pointer. If it's used in a more complex expression, like `[eax + ecx]`, one of the two is a pointer and the other is an array index/byte displacement from that pointer, but there's no telling which is which just from that line, more context is needed.

answered Jul 8 '13 at 19:54

DCoder
1,133 6 14

Thanks for the answer! Could i just ask one small last thing: lets say something like this is apparent: `mov eax, DWORD PTR [ecx + esi*4 + 0x18]` would that indicate that we have an array of structs? Ecx could be the

base adr, esi*4 the size of one element and 0x18 the offset inside the struct to the member we want to access? Not saying that it *has* to be so, but it could? Thanks in advance! – lfxgroove Jul 9 '13 at 11:06

@lfxgroove: you're welcome. Yes, that is the most likely interpretation for such a construct. – DCoder Jul 9 '13 at 11:25

@DCoder has certainly answered this, so here is only some notes, or, at least it started out as a short note, and ended up as a **monster**.

OllyDbg uses `MASM` by default (with some extension). In other words:

```
operation target, source
```

Other syntaxes are available under (depending on version):

- Options->Debugging Options->Disasm
- Options->Code

E.g. `IDEAL`, `HLA` and `AT&T`.

There is also quite a few other options for how the disassembled code looks like. Click around. The changes are instantaneously so easy to find the *right one*.

Numbers are always hex, but without any notation like `0x` or `h` (for compactness sake I guess – and the look is cleaner (IMHO)). Elsewhere, like the instruction details below disassembly, one can see e.g. base 10 numbers – then denoted by a dot at end. E.g. 0x10 (16.)

(And here I stride off …)

When it comes to reading the code

(Talking Intel)

First off tables like the ones at x86asm.net and the Sandpile are definitively valuable assets in the work with assembly code. However one should also have:

- Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture.
- Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z.
- … etc. (There are also some collection volumes.)

From Intel® 64 and IA-32 Architectures Software Developer Manuals.

There is a lot of good sections and descriptions of how a system is stitched together and how operations affect the overall system as in registers, flags, stacks etc. Read e.g. `6.2 STACKS`, `3.4 BASIC PROGRAM EXECUTION REGISTERS`, `CHAPTER 4 DATA TYPES` from the "*Developers*" volume.

As mentioned x86amd and Sandpile are good resources, but when you wonder about an instruction the manual is a good choice as well; "*Instruction Set Reference A-Z*".

Your whole line is probably something like:

```
00406ED6     8B46 14          MOV EAX,DWORD PTR DS:[ESI+14]
; or
00406ED6     8B46 14          MOV EAX,DWORD PTR [ESI+14]
```

(Depending on *options* and *Always show default segment*.)

In this case we can split the binary as:

```
8B46 14
 | |  |
 | |  +---> Displacement
 | +------> ModR/M
 +--------> Opcode
```

Note that there can be prefixes before opcode and other fields as well. For detail look at manual. E.g. "CHAPTER 2 INSTRUCTION FORMAT" in A-Z manual.

Find the `MOV` operation and you will see:

MOV – move

```
Opcode   Instruction     Op/En   64-bit    Compat   Description
…
8B /r    MOV r32,r/m32   RM      Valid     Valid    Move r/m32 to r32.
                |   |
                |   +---> source
                +-------> destination
…
```

Instruction Operand Encoding

```
Op/En   Operand1         Operand2         Operand3         Operand4
RM      ModRM:reg (w)    ModRM:r/m (r)    NA               NA
```

Read "*3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES*" for details on codes.

In short *MOV – mov* table say:

```
8B   : Opcode.
/r   : ModR/M byte follows opcode that contains register and r/m operand.
r32  : One of the doubleword general-purpose registers.
r/m32: Doubleword general-purpose register or memory operand.
RM   : Code for "Instruction Operand Encoding"-table.
```

The *Instruction Operand Encoding* table say:

```
reg  : Operand 1 is defined by the reg bits in the ModR/M byte.
(w)  : Value is written.
r/m  : Operand 2 is Mod+R/M bits of ModR/M byte.
(r)  : Value is read.
```

The too deep section

OK. Now I'm going to deep here, but can't stop myself. (Often find that knowing the building blocks help understand the process.)

The ModR/M byte is `0x46` which in binary form would be:

```
         7,6   5,4,3   2,1,0  (Bit number)
0x46:    01     000     110
          |      |       |
          |      |       +---> R/M
          |      +-----------> REG/OpExt
          +------------------> Mod
```

1. The value `000` of REG field translates to `EAX`
2. Mod+R/M translates to `ESI+disp8`

(Ref. "*2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes*" table 2-2, in A-Z ref.).

Pt. 2. tells us that a 8-bit value, 8-bit displacement byte, follows the ModR/M byte which should be added to the value of `ESI`. In comparison, if there was a 32-bit displacement or register opcode+ModR/M's would be:

```
32-bit displacement                General-purpose register

 +-----> MOV r32,r/m32              +-----> MOV r32,r/m32

 |                                  |
8Bh 86h                            8Bh C1h
     |         +--> EAX                 |         +--> EAX
     |         |                       |         |
     +---> 10 000 110 b                 +---> 11 000 001 b
           |       |                          |       |
           +---+---+                          +---+---+
               |                                  |
               v                                  v
               ESI + disp32                       ECX
```

As we have a `disp8` the next byte is a 1-byte value that should be added to the value of ESI. In this case `0x14`.

Note that this byte is signed so e.g. `0xfe` would mean `ESI - 0x02`.

Segment to use

ESI is pointer to data in segment pointed to by DS.

A segment selector is comprised of three values:

```
    15 - 3             2                   1 - 0              (Bits)
|-------------|-----------------|--------------------------|
|    Index    | Table Indicator | Requested Privilege Level |
+-------------+-----------------+--------------------------+
```

So say selector = 0x0023 we have:

```
0x23 0000000000100 0 11 b
           |       |  |
           |       |  +----> RPL  : 3   = User land, (0 is kernel)
           |       |  +-------> TI   : 0   = GDT (GDT or LDT)
           |       +-------> TI   : 0   = GDT (GDT or LDT)
           +---------------> Index: 4     Multiplied by 8 and added to TI
```

- GDT = Global Descriptor Table
- LDT = Local Descriptor Table

The segment registers (CS, DS, SS, ES, FS and GS) are designed to hold selectors for code, stack or data. This is to lessen complexity and increase efficiency.

Each of these registers also have a *hidden part* aka "*shadow register*" or "*descriptor cache*" which holds *base address*, *segment limit* and *access control information*. These values are automatically loaded by the processor when a segment selector is loaded into the visible part of the segment registers.

```
   | Segment Selector |      Shadow Register     |
   +------------------+--------------------------+
   |  Idx  | TI | RPL | BASE  | Seg Lim | Access | CS, SS, DS, ES, FS, GS
   +-------+----+-----+-------+---------+--------+
```

The BASE address is a linear address. ES, DS and SS are not used in 64-bit mode.

Result

Read a 32-bit value from segment address ESI+disp8. Example:

```
ESI = 0x005056A0

Dump of DS segment:
           0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
005056A0  00 00 00 00 9C 8F 41 7E 4C 1F 42 00 C0 1E 42 00  ....œA~LB.ÀB.
005056B0  E0 1F 42 00 70 20 42 00 48 21 42 00 4A A8 42 7E  àB.p B.H!B.J¨B~

ESI + 0x14 = 0x005056B4 => 70 20 42 00 …

EAX = (DWORD)70 20 42 00 = 00 42 20 70 (4333680.)
```

Simulate in C

One problem with your example is that `esi` is an integer (strictly speaking). The value, however, can be that one of a segment address. Then you have to take into consideration that each segment has a *base address*, (offset), – as in:

```
seg = malloc(4096);
seg[0]
    |
    +---> at base address, e.g. 0x505000

       +----------------+
       |                |
       |                |
        …
505000 |                | seg[00 - 0f]
505010 |                | seg[10 - 1f]
505020 |                | seg[20 - 2f]
        …
```

In this case, as it is ESI, that segment would be the one pointed to by DS.

To simulate this in C you would need variables for the general-purpose registers, but you would also need to create segments (from where to read/write data.) Roughly such a code **could** be something

need to create segments (from where to read/write data.) Roughly such a code **could** be something like:

```
void dword_m2r(uint32_t *x, struct segment *seg, uint32_t offset)
{
    *x = *((uint32_t*)(seg->data + (offset - seg->base)));
}

dword_m2r(&eax, &ds, esi + 0x14);
```

Where `struct segment` and `ds` are:

```
struct segment {
    u8 *data;
    u32 base;
    u32 size;
    u32 eip;
};

struct segment ds;
ds.base = 0x00505000;
ds.size = 0x3000;
ds.data = malloc(ds.size);
ds.eip  = 0x00;
```

To further develop on this concept you could create another `struct` with registers, use defines or variables for registers, add default segments etc.

For Intel-based architecture that could be something in the direction of this (as a not to nice beginning):

```
#include <stdint.h>

#define u64  uint64_t
#define u32  uint32_t
#define u16  uint16_t
#define u8   uint8_t

union gen_reg {
    u64 r64;
    u32 r32;
    u16 r16;
    u8   l8;
};

struct CPU {
    union gen_reg accumulator;
    u8 *ah;
    union gen_reg counter;
    u8 *ch;
    …
    struct segment s_stack;
    struct segment s_code;
    struct segment s_data;
    …

    u32 eflags;
    u32 eip;
    …
};


#define RAX   CPU.accumulator.rax

#define EAX   CPU.accumulator.eax
#define AX    CPU.accumulator.ax
#define AH    *((u8*)&AX + 1)
#define AL    CPU.accumulator.al
…


/* and then some variant of */
ESI = 0x00505123;
dword_m2r(&EAX, &DS, ESI + 0x14);
```

For a more compact way, ditching ptr to `H` register etc. have a look at e.g. the code base of

virtualbox. ***Note:*** require some form of pack directive for most compilers to prevent filling of bits in structs – so that e.g. AH and AL really align up with correct bytes of AX.

edited Jul 9 '13 at 12:40

answered Jul 9 '13 at 11:22

Sukminder

456 ● 1 ● 5

I think i've got something to read up on right now ;) Thanks a lot for the information! – lfxgroove Jul 9 '13 at 11:30

@lfxgroove: Your welcome :) It is rather educational to emulate assembly in C, (if are used to C). – Sukminder Jul 9 '13 at 11:45

I can imagine, haven't ever really thought of doing that in that way :) – lfxgroove Jul 9 '13 at 11:46

Harf. I also see I have messed up the #defines in last code block. Should obviously be `CPU.accumulator.r64` etc. – Sukminder Jul 9 '13 at 15:44