Understanding subtraction and multiplication in assembly code

Asked 5 years, 3 months ago Active 5 years, 3 months ago Viewed 1k times

0

1

Can someone explain what these steps in the disassembled code would do. I have a general idea but I'm still confused. I know that the first two instructions set up the stack and that eax will be a returned value but that's about it.

What I'm looking for is the purpose of the steps below:

```
push %ebp - base stack frame pointer
mov %esp, %ebp - stack pointer
sub $0x10, %esp - subtracts 16 from ?
mov 0x8(%ebp), %eax - ?
imul 0xc(%ebp), %eax - multiply 12 and ?
mov %eax, -0x4(%ebp) - ?
mov -0x4(%ebp), %eax - puts -0x4(%ebp) not sure what that would be , into eax making
it the return value?
leave
ret
```

c assembly x86 disassembly

Share Improve this question Follow

edited Nov 12 '15 at 20:53
Paul R
194k 32 341 511

asked Nov 12 '15 at 20:50
user5556391
11 3

`gcc -O0` (the default) is full of noisy load/store instructions. Use `gcc -Og -fverbose-asm` (optimize for debugging) for more readable output that tends to keep variables live in registers. – Peter Cordes Nov 13 '15 at 3:38

First try to find a tutorial on assembly language. Learning assembler by guessing is probably not an efficient way. – skyking Nov 13 '15 at 7:31

1 Answer

Active Oldest Votes

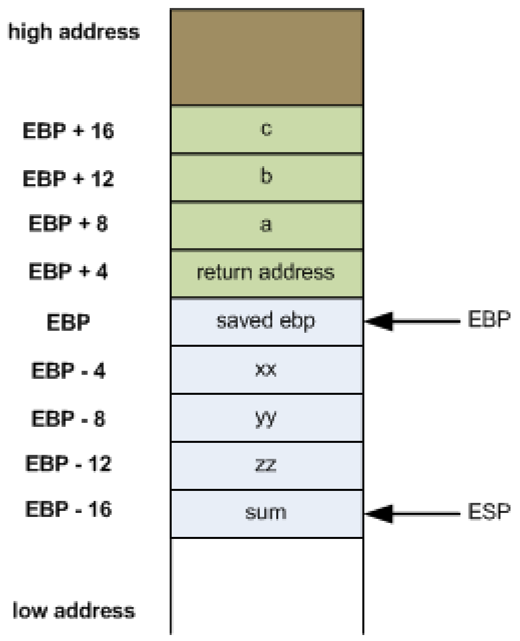
5

```
; Standard prolog: stack frame setup
push ebp               ; save the old frame pointer
mov ebp, esp           ; set the frame pointer to the current top of the stack
sub esp, 0x10          ; make space for 16 bytes of local variables
; Do the stuff
mov eax, [ebp+8]       ; copy the first parameter in eax
```

```
                        ;          pop ebp
ret                     ; return
```

(sorry for changing it to Intel notation, but AT&T syntax looks like an unreadable mess to me, especially the hideous notation for dereferencing and offsets[1])

To understand this keep around this handy diagram of how the stack normally looks like in a *cdecl* function call on x86 just after the function prolog:



and remember that expressions in brackets are pointer dereferencing operations.

Essentially, this is a (quite naive) translation of

```
int multiply(int a, int b) {
    //           \      \ &b == ebp+12
    //            \ &a == ebp+8
    int c = a*b;
    //   \    \ multiplication performed in eax
    //    \ &c == ebp-4
    return c;
    //   \ return value left in eax
}
```

(using the cdecl calling convention, where it's the caller's responsibility to clean up the parameters from the stack)

Probably this was generated by a compiler with optimizations disabled. A more compact version would be:

(since everything can be done in without local variables, there's not even need for setting up a stack frame)

Edit

Just checked, your code matches exactly what gcc produces at `-O0`, while mine is almost identical to what is generated at `-O3`.

Notes

1. For the record: when you see

   ```
   displacement(%register, %offset_register, multiplier)
   ```

   (each component besides `%register` is optional) in AT&T syntax it actually means

   ```
   [register + displacement + offset_register*multiplier]
   ```

   where the brackets mean "take the value stored here".

   Also, almost all parameters are swapped in AT&T syntax (in Intel syntax the destination operand is on the left, i.e. a `mov` reads like an assignment - `mov ebp, esp` => `ebp = esp`).

Share Improve this answer Follow edited Nov 12 '15 at 21:52 answered Nov 12 '15 at 20:58

Matteo Italia
114k 16 177 276

2 AT&T syntax IMHO isn't hideous. I find it quite readable myself. Reading Intel or AT&T syntax takes the same amount of effort for me to read. In your note you are missing the `offset` or displacement in this `[register + offset_register*multiplier]`. Usually to make more sense of that I would say `offset[base_register + index_register*multiplier]`. `offset` could also be described as `displacement` (terms are interchangeable). `multiplier` could also be described as a `scalar` constant. – Michael Petch Nov 12 '15 at 21:49

1 @MichaelPetch: wops, fixed the note. My complaints against AT&T are the usual ones: extreme visual clutter (all those `%` and `$` around where the meaning is already clear), and the offset syntax requires significantly more knowledge and attention to parse correctly (in Intel syntax it reads as a "normal" expression, without having to remember exactly what field means what). But of course as always it's really a matter of habit. – Matteo Italia Nov 12 '15 at 21:56

I agree. The issue of preferred style is a subjective one. I believe the first time I saw AT&T syntax was on the Sun Compiler's back on the Sparc architecture in the 80s. – Michael Petch Nov 12 '15 at 22:00

@MichaelPetch: The big problem with Intel syntax is that some assemblers treat `mov eax, symbol` as a load (same as `mov eax, [symbol]`), while others treat it as a mov-immediate of the address (`mov eax, offset symbol`, with the same result as `lea eax, [symbol]`). Now that I'm more used to Intel syntax, I'm used to the issue, and having to sprinkle `BYTE PTR` or whatever in front of memory operands in some insns. I used to think AT&T syntax was significantly better, but now I prefer Intel. – Peter Cordes between assemblers is what drives me nuts about Intel syntax. – Peter Cordes Nov 13 '15 at 9:09