What do the instructions mov %edi and mov %rsi do?

Asked 4 years, 6 months ago Active 4 years, 6 months ago Viewed 4k times

8

1

I've written a basic C program that defines an integer variable x, sets it to zero and returns the value of that variable:

```
#include <stdio.h>

int main(int argc, char **argv) {
    int x;
    x = 0;
    return x;
}
```

When I dump the object code using objdump (compiled on Linux X86-64 with gcc):

```
0x0000000000400474 <main+0>:    push   %rbp
0x0000000000400475 <main+1>:    mov    %rsp,%rbp
0x0000000000400478 <main+4>:    mov    %edi,-0x14(%rbp)
0x000000000040047b <main+7>:    mov    %rsi,-0x20(%rbp)
0x000000000040047f <main+11>:   movl   $0x0,-0x4(%rbp)
0x0000000000400486 <main+18>:   mov    -0x4(%rbp),%eax
0x0000000000400489 <main+21>:   leaveq
0x000000000040048a <main+22>:   retq
```

I can see the function prologue, but before we set x to 0 at address `0x000000000040047f` there are two instructions that move %edi and %rsi onto the stack. What are these for?

In addition, unlike where we set x to 0, the mov instruction as shown in GAS syntax does not have a suffix.

> If the suffix is not specified, and there are no memory operands for the instruction, GAS infers the operand size from the size of the destination register operand.

In this case, are `-0x14(%rsbp)` and `-0x20(%rbp)` both memory operands and what are their sizes? Since %edi is a 32 bit register, are 32 bits moved to `-0x14(%rsbp)` whereas since %rsi is a 64 bit register, 64 bits are moved to `%rsi,-0x20(%rbp)` ?

c gas disassembly

asked Jul 14 '15 at 11:22

user4099632

2 After those two movs `%rsi` is `argv` and `%edi` is `argc` . – Jabberwocky Jul 14 '15 at 11:26

Interesting! It's been my understanding that the arguments (i.e. argc and argv) are pushed onto the stack before the stack frame (i.e. push %rbp). Would that mean that argc and argv are within the stack frame for main? – user4099632 Jul 14 '15 at 11:29

Actually I may be wrong. Write another small function that just calls your `main` function and objdump the file again, and you will see. – Jabberwocky Jul 14 '15 at 11:34

1 @GeorgeRobinson In x86_64 there's a very commonly used calling convention where the first 6 arguments are put into registers. en.wikipedia.org/wiki/… – SeriousBusiness Jul 14 '15 at 16:48

2 Answers

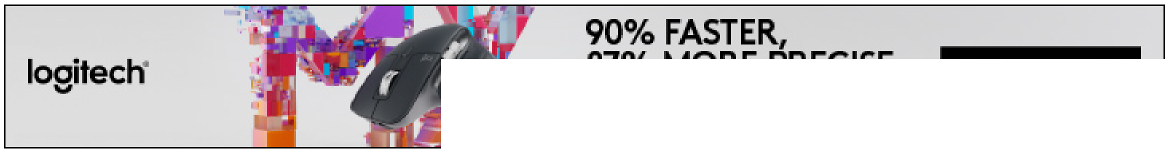
9

```
main:
    pushq   %rbp    #
    movq    %rsp, %rbp  #,
    movl    %edi, -20(%rbp) # argc, argc
    movq    %rsi, -32(%rbp) # argv, argv
    movl    $0, -4(%rbp)    #, x
    movl    -4(%rbp), %eax  # x, D.2607
    popq    %rbp    #
    ret
```

So, yes, they save `argv` and `argv` onto the stack by using the "old" frame pointer method since new architectures allow subtracting/adding from/to the stack pointer directly, thus omitting the frame pointer (`-fomit-frame-pointer`).

answered Jul 14 '15 at 11:40
edmz
7,199 2 18 42

This un-optimized code did *not* use `-fomit-frame-pointer` . It's using the red-zone below RSP (a software convention that's part of the x86-64 calling convention), but it's doing it with RBP as a frame pointer. Ever since 386 it's been possible to reference memory relative to the stack pointer; that's not new, but like I said not what's happening here. – Peter Cordes Jul 15 '18 at 11:55



[Purpose of ESI & EDI registers?](#)

0

Based on this and the context, I'm not an expert, but my guess is these are capturing the `main()` input parameters. EDI takes a standard width, which would match the `int argc` , whereas RSI takes a long, which would match the `char **argv` pointer.

edited May 23 '17 at 10:30
Community ♦
1 1

answered Jul 14 '15 at 11:25
underscore_d
4,107 3 24 52