Code as Art

Blog about system programming and not only

Say hello to x64 Assembly [part 1]

Introduction

There are many developers between us. We write a tons of code every day. Sometime, it is even not a bad code :) Every of us can easily write the simplest code like this:

```c
#include <stdio.h>

int main() {
  int x = 10;
  int y = 100;
  printf("x + y = %d", x + y);
  return 0;
}
```

gistfile1.c hosted with ❤ by GitHub view raw

Every of us can understand what's this C code does. But... How this code works at low level? I think that not all of us can answer on this question, and me too. I thought that i can write code on high level programming languages like Haskell, Erlang, Go and etc..., but i absolutely don't know how it works at low level, after compilation. So I decided to take a few deep steps down, to assembly, and to describe my learning way about this. Hope it will be interesting, not only for me. Something about 5 - 6 years ago I already used assembly for writing simple programs, it was in university and i used Turbo assembly and DOS operating system. Now I use Linux-x86-64 operating system. Yes, must be big difference between Linux 64 bit and DOS 16 bit. So let's start.

Preparation

Before we started, we must to prepare some things like As I wrote about, I use Ubuntu (Ubuntu 14.04.1 LTS 64 bit), thus my posts will be for this operating system and architecture. Different CPU supports different set of instructions. I use *Intel Core i7 870* processor, and all code will be written processor. Also i will use nasm assembly. You can install it with:

> sudo apt-get install nasm

It's version must be 2.0.0 or greater. I use *NASM version 2.10.09 compiled on Dec 29 2013* version. And the last part, you will need in text editor where you will write you assembly code. I use Emacs with *nasm-mode.el* for this. It is not mandatory, of course you can use your favourite text editor. If you use Emacs as me you can download nasm-mode.el and configure your Emacs like this:

```lisp
(load "~/.emacs.d/lisp/nasm.el")
(require 'nasm-mode)
(add-to-list 'auto-mode-alist '("\\.\\(asm\\|s\\)$" . nasm-mode))
```

gistfile1.el hosted with ❤ by GitHub view raw

That's all we need for this moment. Other tools will be describe in next posts.

x64 syntax

Here I will not describe full assembly syntax, we'll mention only those parts of the syntax, which we will use in this post. Usually NASM program divided into sections. In this post we'll meet 2 following sections:

- data section
- text section

The data section is used for declaring constants. This data does not change at runtime. You can declare various math or other constants and etc... The syntax for declaring data section is:

> section .data

The text section is for code. This section must begin with the declaration *global _start*, which tells the kernel where the program execution begins.

> section .text
> global _start
> _start:

Comments starts with *;* symbol. Every NASM source code line contains some combination of the following four fields:

> [label:] instruction [operands] [; comment]

Fields which are in square brackets are optional. A basic NASM instruction consists from two parts. The first one is the name of the instruction which is to be executed, and the second are the operands of this command. For example:

> MOV COUNT, 48 ; Put value 48 in the COUNT variable

Hello world

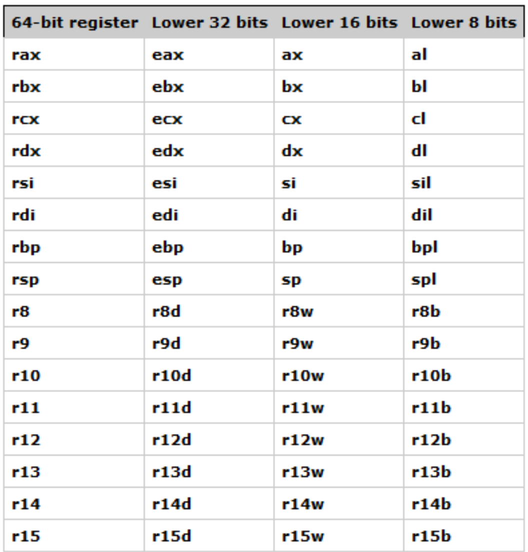
Let's write first program with NASM assembly. And of course it will be traditional Hello world program. Here is the code of it:

```
1   section .data
2       msg db      "hello, world!"
3
4   section .text
5       global _start
6   _start:
7       mov     rax, 1
8       mov     rdi, 1
9       mov     rsi, msg
10      mov     rdx, 13
11      syscall
12      mov    rax, 60
13      mov    rdi, 0
14      syscall
```

Yes, it doesn't look like *printf("Hello world")*. Let's try to understand what is it and how it works. Take a look 1-2 lines. We defined *data* section and put there *msg* constant with *Hello world* value. Now we can use this constant in our code. Next is declaration *text* section and entry point of program. Program will start to execute from 7 line. Now starts the most interesting part. We already know what is it *mov* instruction, it gets 2 operands and put value of second to first. But what is it these *rax*, *rdi* and etc... As we can read at wikipedia:

> A central processing unit (CPU) is the hardware within a computer that carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system.

Ok, CPU performs some operations, arithmetical and etc... But where can it get data for this operations? The first answer in memory. However, reading data from and storing data into memory slows down the processor, as it involves complicated processes of sending the data request across the control bus. Thus CPU has own internal memory storage locations called **registers**:



64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

So when we write *mov rax, 1*, it means to put 1 to the *rax* register. Now we know what is it rax, rdi, rbx and etc... But need to know when to use rax but when rsi and etc...

- rax - temporary register; when we call a syscal, rax must contain syscall number
- rdx - used to pass 3rd argument to functions
- rdi - used to pass 1st argument to functions
- rsi - pointer used to pass 2nd argument to functions

In another words we just make a call of *sys_write* syscall. Take a look on *sys_write*:

```
ssize_t sys_write(unsigned int fd, const char * buf, size_t count)
```

It has 3 arguments:

- fd - file descriptor. Can be 0, 1 and 2 for standard input, standard output and standard error

- buf - points to a character array, which can be used to store content obtained from the file pointed to by fd.
- count - specifies the number of bytes to be written from the file into the character array

So we know that *sys_write* syscall takes three arguments and has number one in syscall table. Let's look again to our hello world implementation. We put 1 to rax register, it means that we will use sys_write system call. In next line we put 1 to rdi register, it will be first argument of sys_write, 1 - standard output. Then we store pointer to *msg* at rsi register, it will be second *buf* argument for sys_write. And then we pass the last (third) parameter (length of string) to rdx, it will be third argument of sys_write. Now we have all arguments of sys_write and we can call it with *syscall* function at 11 line. Ok, we printed "Hello world" string, now need to do correctly exit from program. We pass 60 to rax register, 60 is a number of exit syscall. And pass also 0 to rdi register, it will be error code, so with 0 our program must exit successfully. That's all for "Hello world". Quite simple :) Now let's build our program. For example we have this code in *hello.asm* file. Then we need to execute following commands:

```
nasm -f elf64 -o hello.o hello.asm
ld -o hello hello.o
```

After it we will have executable *hello* file which we can run with ./hello and will see Hello world string in the terminal.

Conclusion

It was a first part with one simple-simple example. In next part we will see some arithmetic. If you will have any questions/suggestions write me a comment.

All source code you can find - here.

183 132

Labels: asm, Linux, x64

25 Comments **0xAX blog** 1 **Login**

Recommend 5 **Share** **Sort by Newest**

Join the discussion…

Hoffstot Lilli • a month ago

There are some efficient guidelines to be followed in the future.

Reply • Share ›

dskecse • a year ago

There were some issues running the program on `Mac OS X Yosemite`. So, in case someone else faces the issues, one could solve them by running the following commands:

```
nasm -f macho64 -o hello.o hello.asm
ld -macosx_version_min 10.10 -e _start -o hello hello.o -lSystem
```

Though I get `34344 bus error ./hello`.

Reply • Share ›

jasonkit → dskecse • a year ago

It is because the system call number for mac is not the same as linux.
sys_write is 4 and sys_exit is 1 in mac (you can find out syscall number in /usr/include/sys/syscall.h)
In addition, we need to offset the syscall number by 0x200000 for using POSIX syscall (refer to http://www.opensource.apple.co...
i.e. the syscall number for sys_write will be 0x2000004 and sys_exit will be 0x2000001

1 • Reply • Share ›

SAAD • a year ago

Firstly thank you for your efforts.
I work on an x86 and when I use "syscall" it gives me an error "impermissible statement" but I find a solution, change this statement to "int 80h"
Is it that I'm right, or is it something else !!
Thank you :)

Reply • Share ›

Pankaj Doharey • a year ago

Thanks for pointing to the nasm mode for emacs.

Reply • Share ›

Marcus • a year ago

I tried the example with the GNU assembler (with command-line switches to enable Intel syntax) and noticed that it assembles

mov rsi, msg

as something like

as something like

mov rsi, [msg] ; i.e. it doesn't load the address, it loads from the address!

If someone else tries the same thing, maybe this will help: I used the lea instruction to load the address into rsi:

lea rsi, msg

but I wonder if there is any other way to do that (and which behavior is more standard or common in assemblers).

By the way, I defined the string constant with .asciz

1 ⌃ | ⌄ • Reply • Share ›

0xAX Mod ➦ Marcus • a year ago

You can find this example with gas and intel syntax here - https://github.com/e12e/asm/bl...

⌃ | ⌄ • Reply • Share ›

Marcus ➦ 0xAX • a year ago

That's interesting, thank you for your reply. I'll experiment with that later.

⌃ | ⌄ • Reply • Share ›

kyokokken • a year ago

Nice big picture view without going for the corner cases. Great article.

4 ⌃ | ⌄ • Reply • Share ›

diogovk • a year ago

Can't wait for the follow up.
This is very interesting to me.

2 ⌃ | ⌄ • Reply • Share ›

Diago • a year ago

You might enjoy this http://www.pentesteracademy.co...

2 ⌃ | ⌄ • Reply • Share ›

Grienders • a year ago

I didn't like the article, most of the things aren't clear:

1) what if we want to call a function with 12 arguments?

2) how do you know that sys_write has number 1 in syscall table?

3) how do you know that sys_write corresponds to printf?

4) Can _start be titled differently?

5) Why do you use "db" for msg and not dw or something else?

9 ⌃ | ⌄ • Reply • Share ›

0xAX Mod ➦ Grienders • a year ago

First of all thank you for so detailed feedback. You and other peoples with the same feedback are right here. Some parts from this post can be unclear. But look, it is just a little introduction with little code,little description of this code and instruction how to run it. I don't how you, but as for me i dont like giant posts. All another parts like registers, data types, memory and etc etc....

8 ⌃ | ⌄ • Reply • Share ›

David Conrad • a year ago

RAX isn't a "temporary register ", it's the accumulator. RSI and RDI are the source and destination index registers. Of course they, like any register, can also be used to pass parameters.

9 ⌃ | ⌄ • Reply • Share ›

David @InfinitelyManic ➦ David Conrad • a year ago

Yep

Glad I live in the present since I heard doing assembly on the 8088 was a far cry from what's available w/ x86-64 or even MIPS... These notes are for everyone.

ax - accumulator for numeric options - still used in x86-64 for multiplication, division, string scans, xlat table translations, printf floating point parameters, etc.
bx - base register (array access)
cx - count register (string opertions) ; x86-64 uses rcx for loop also
dx - data register ; x86-64 rdx is used to hold remainder for div; rdx also holds high order bits for multplication
si - source index; x86-64 rdi used in string instructions
di - destination index; x86-64 same as above +
bp - base pointer; x86-64 -this is a general purpose reg and can be used for stuff other than frames; but ....
sp - stack pointer ;

3 ⌃ | ⌄ • Reply • Share ›

Kenneth • a year ago

Thanks a ton, great article / tutorial. You should continue and go further in depth!

1 ⌃ | ⌄ • Reply • Share ›

0xAX Mod ➦ Kenneth • a year ago

Glad that you liked it. Soon will be more articles.

3 ^ | v • Reply • Share ›

Philipp • a year ago

Thanks to your article wrote my first assembler program! :)

1 ^ | v • Reply • Share ›

0xAX Mod ↪ Philipp • a year ago

Thank you, glad that you liked it

^ | v • Reply • Share ›

Anon • a year ago

Than -> then (sorry, grammar nazi)

Nice article btw.
It would be interesting to have the analysis of the .asm code for the code at the beginning! (x+y)

^ | v • Reply • Share ›

0xAX Mod ↪ Anon • a year ago

Fixed it, thank you for feedback, english is not my first language, so i am appreciate your feedback

1 ^ | v • Reply • Share ›

Robin Glauser • a year ago

Really nice article, I enjoyed reading it.

However one question: Hasn't sys_write the number 4 in the system call table? Or am I looking in the wrong place?
http://docs.cs.up.ac.za/progra...

2 ^ | v • Reply • Share ›

0xAX Mod ↪ Robin Glauser • a year ago

Yes, 4 is for 32 bit systems, you can find syscall table for it here - http://docs.cs.up.ac.za/progra..., and 64 bit if you're interesting - http://blog.rchapman.org/post/...

1 ^ | v • Reply • Share ›

Stefano Borini ↪ Robin Glauser • a year ago

I might be wrong, but it's because he is using the syscall opcode. Those numbers are for INT 0x80 syscall strategy.

^ | v • Reply • Share ›

Robin Glauser ↪ Stefano Borini • a year ago

Oh, that's it. Thank you.
https://filippo.io/linux-sysca...

^ | v • Reply • Share ›

ALSO ON **0XAX BLOG** WHAT'S THIS?

Code as Art: Simple bash script for setting up Golang environment

2 comments • 2 years ago

Avat **0xAX** — О спасибо, не знал

Brain --dump: [Emacs] mu4e + offlineimap multiply accounts

1 comment • a year ago

Avat **Dude** — Seems kinda hackish - could you elaborate why you decided no to use msmtp?

Code as Art: Why i like Golang

4 comments • 2 years ago

Avat **horechek** — О, премного благодарен. Если будет желание статью опубликовать, всегда велкам :)

Code as Art: Say hello to x64 Assembly [part 2]

7 comments • a year ago

Avat **David @InfinitelyManic** — Good stuff! Keep up the good work! Not sure if you already touched on a few items; namely, assembler used in demo (e.g., nasm, yasm), …

✉ Subscribe D Add Disqus to your site Add Disqus Add 🔒 Privacy

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

0xAX. Powered by Blogger.