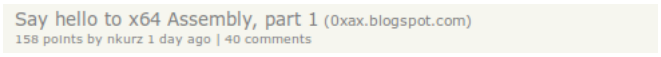
Code as Art

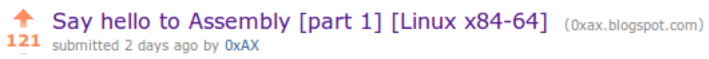
Blog about system programming and not only

Say hello to x64 Assembly [part 2]

Say hello to x64 Assembly [part 2]

Some days ago I wrote the first blog post - introduction to x64 assembly - Say hello to x64 Assembly [part 1] which to my surprise caused great interest:


Say hello to x64 Assembly, part 1 (0xax.blogspot.com)
158 points by nkurz 1 day ago | 40 comments


121 Say hello to Assembly [part 1] [Linux x84-64] (0xax.blogspot.com)
submitted 2 days ago by 0xAX

It motivates me even more to describe my way of learning. During this days I got many feedback from different people. There were many grateful words, but
more important for me, there were many advices and adequate critics. Especially I want to say thank you words for great feedback to:

- Fiennes
- Grienders
- nkurz

And all who took a part in discussion at Reddit and Hacker News. There were many opinions, that first part was a not very clear for absolute beginner, that's
to write more informative posts. So, let's start with second part of *Say hello to x64 assembly.*

Terminology and Concepts

As i wrote above, I got many feedback from different people that some parts of first post are not clear, that's why let's start from description of some termi
we will see in this and next parts.

Register - register is a small amount of storage inside processor. Main point of processor is data processing. Processor can get data from memory, but it is
operation. That's why processor has own internal restricted set of data storage which name is - register.

Little-endian - we can imagine memory as one large array. It contains bytes. Each address stores one element of the memory "array". Each element is one by
example we have 4 bytes: *AA 56 AB FF*. In little-endian the least significant byte has the smallest address:

0 FF
1 AB
2 56
3 AA

where 0,1,2 and 3 are memory addresses.

Big-endian - big-endian stores bytes in opposite order than little-endian. So if we have *AA 56 AB FF* bytes sequence it will be:

0 AA
1 56
2 AB
3 FF

Syscall - is the way a user level program asks the operating system to do something for it. You can find syscall table - here. **Stack** - processor has a very restr
of registers. So stack is a continuous area of memory addressable special registers RSP,SS,RIP and etc... We will take a closer look on stack in next parts.

Section - every assembly program consists from sections. There are following sections:

- data - section is used for declaring initialized data or constants
- bss - section is used for declaring non initialized variables
- text - section is used for code

General-purpose registers - there are 16 general-purpose registers - rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15. Of course, it is not
terms and concepts which related with assembly programming. If we will meet another strange and unfamiliar words in next blog posts, there will be expla
words.

Data Types

The fundamental data types are bytes, words, doublewords, quadwords, and double quadwords. A byte is eight bits, a word is 2 bytes, a doubleword is 4 by
quadword is 8 bytes and a double quadword is 16 bytes (128 bits).

Now we will work only with integer numbers, so let's see to it. There two types of integer: unsigned and signed. Unsigned integers are unsigned binary numb
in a byte, word, doubleword, and quadword. Their values range from 0 to 255 for an unsigned byte integer, from 0 to 65,535 for an unsigned word integer,
2^32 − 1 for an unsigned doubleword integer, and from 0 to 2^64 − 1 for an unsigned quadword integer. Signed integers are signed binary numbers held a
a byte, word and etc... The sign bit is set for negative integers and cleared for positive integers and zero. Integer values range from −128 to +127 for a byte
−32,768 to +32,767 for a word integer,from −2^31 to +2^31 − 1 for a doubleword integer, and from −2^63 to +2^63 − 1 for a quadword integer.

Sections

As i wrote above, every assembly program consists from sections, it can be data section, text section and bss section. Let's look on *data* section.It's main po
declare initialized constants. For example:

```
section .data
    num1:   equ 100
    num2:   equ 50
    msg:    db "Sum is correct", 10
```

gistfile1.asm hosted with ❤ by **GitHub**

Ok, it is almost all clear here. 3 constants with name num1, num2, msg and with values 100, 50 and "Sum is correct", 10. But what is it *db*, *equ*? Actual *NASI*
number of pseudo-instructions:

- DB, DW, DD, DQ, DT, DO, DY and DZ - are used for declaring initialized data. For example:

```
;; Initialize 4 bytes 1h, 2h, 3h, 4h
db 0x01,0x02,0x03,0x04

;; Initialize word to 0x12 0x34
dw    0x1234
```

gistfile1.asm hosted with ❤ by **GitHub** **vi**

- RESB, RESW, RESD, RESQ, REST, RESO, RESY and RESZ - are used for declaring non initialized variables
- INCBIN - includes External Binary Files
- EQU - defines constant. For example:

```
;; now one is 1
one equ 1
```

gistfile1.asm hosted with ❤ by **GitHub** **vi**

- TIMES - Repeating Instructions or Data. (description will be in next posts)

Arithmetic operations

There is short list of arithmetic instructions:

- ADD - integer add
- SUB - substract
- MUL - unsigned multiply
- IMUL - signed multiply
- DIV - unsigned divide
- IDIV - signed divide
- INC - increment
- DEC - decrement
- NEG - negate

Some of it we will see at practice in this post. Other will be covered in next posts.

Control flow

Usually programming languages have ability to change order of evaluation (with if statement, case statement, goto and etc...) and assembly has it too. Here some of it. There is *cmp* instruction for performing comparison between two values. It is used along with the conditional jump instruction for decision makin example:

```
;; compare rax with 50
cmp rax, 50
```

gistfile1.asm hosted with ❤ by **GitHub**

cmp instruction just compares 2 values, but doesn't affect them and doesn't execute anything depend on result of comparison. For performing any actions af comparison there is *conditional jump* instructions. It can be one of it:

- JE - if equal
- JZ - if zero
- JNE - if not equal
- JNZ - if not zero
- JG - if first operand is greater than second
- JGE - if first operand is greater or equal to second
- JA - the same that JG, but performs unsigned comparison
- JAE - the same that JGE, but performs unsigned comparison

For example if we want to make something like if/else statement in C:

```
if (rax != 50) {
    exit();
} else {
    right();
}
```

gistfile1.c hosted with ❤ by **GitHub**

It will be in assembly:

```
;; compare rax with 50
cmp rax, 50
;; perform .exit if rax is not equal 50
jne .exit
jmp .right
```

gistfile1.asm hosted with ❤ by **GitHub**

There is also unconditional jump with syntax:

> JMP label

For example:

```
_start:
    ;; ....
    ;; do something and jump to .exit label
    ;; ....
    jmp .exit

.exit:
    mov    rax, 60
    mov    rdi, 0
    syscall
```

gistfile1.asm hosted with ❤ by **GitHub**

Here we have can have some code which will be after *_start* label, and all of this code will be executed, assembly transfer control to *.exit* label, and code af start to execute.

Often unconditional jump uses in loops. For example we have *label* and some code after it. This code executes anything, than we have condition and jump t this code if condition is not successfully. Loops will be covered in next parts.

Example

Let's see simple example. It will take two integer numbers, get sum of these numbers and compare it with predefined number. If predefined number is equa
will print something on the screen, if not - just exit. Here is the source code of our example:

```
 ;initialised data section
section .data
    ; Define constants
    num1:   equ 100
    num2:   equ 50
    ; initialize message
    msg:    db "Sum is correct\n"

section .text

    global _start

;; entry point
_start:
    ; set num1's value to rax
    mov rax, num1
    ; set num2's value to rbx
    mov rbx, num2
    ; get sum of rax and rbx, and store it's value in rax
    add rax, rbx
    ; compare rax and 150
    cmp rax, 150
    ; go to .exit label if rax and 150 are not equal
    jne .exit
    ; go to .rightSum label if rax and 150 are equal
    jmp .rightSum

; Print message that sum is correct
.rightSum:
    ;; write syscall
    mov     rax, 1
    ;; file descritor, standard output
    mov     rdi, 1
    ;; message address
    mov     rsi, msg
    ;; length of message
    mov     rdx, 15
    ;; call write syscall
    syscall
    ; exit from program
    jmp .exit

; exit procedure
.exit:
    ; exit syscall
    mov    rax, 60
    ; exit code
    mov    rdi, 0
    ; call exit syscall
    syscall
```

gistfile1.asm hosted with ❤ by **GitHub**

Let's go through the source code. First of all there is *data* section with two constants *num1*, *num2* and variable *msg* with "Sum is correct\n" value. Now look
There is begin of program's entry point. We transfer *num1* and *num2* values to general purpose registers *rax* and *rbx*. Sum it with *add* instruction. After execu
instruction, it calculates sum of values from *rax* and *rbx* and store it's value to *rax*. Now we have sum of *num1* and *num2* in the *rax* register.

Ok we have num1 which is 100 and num2 which is 50. Our sum must be 150. Let's check it with *cmp* instruction. After comparison *rax* and 150 we check resu
comparison, if *rax* and 150 are not equal (checking it with jne) we go to *.exit* label, if they are equal we go to *.rightSum* label.

Now we have two labels: *.exit* and *.rightSum*. First is just sets 60 to *rax*, it is exit system call number, and 0 to *rdi*, it is a exit code. Second is *.rightSum* is pre
just prints *Sum is correct\n*. If you don't understand how it works, see first post.

Conclusion

It was a second part of series 'say hello to x64 assembly', if you will have a questions/suggestions write me a comment.

All source code you can find as everytime - here.