# x86

bananaapple

# Before we start

Add architecture

- dpkg --add-architecture i386

Update repository

- apt-get update

Install library

- apt-get install ia32-libs
- apt-get install gcc-multilib

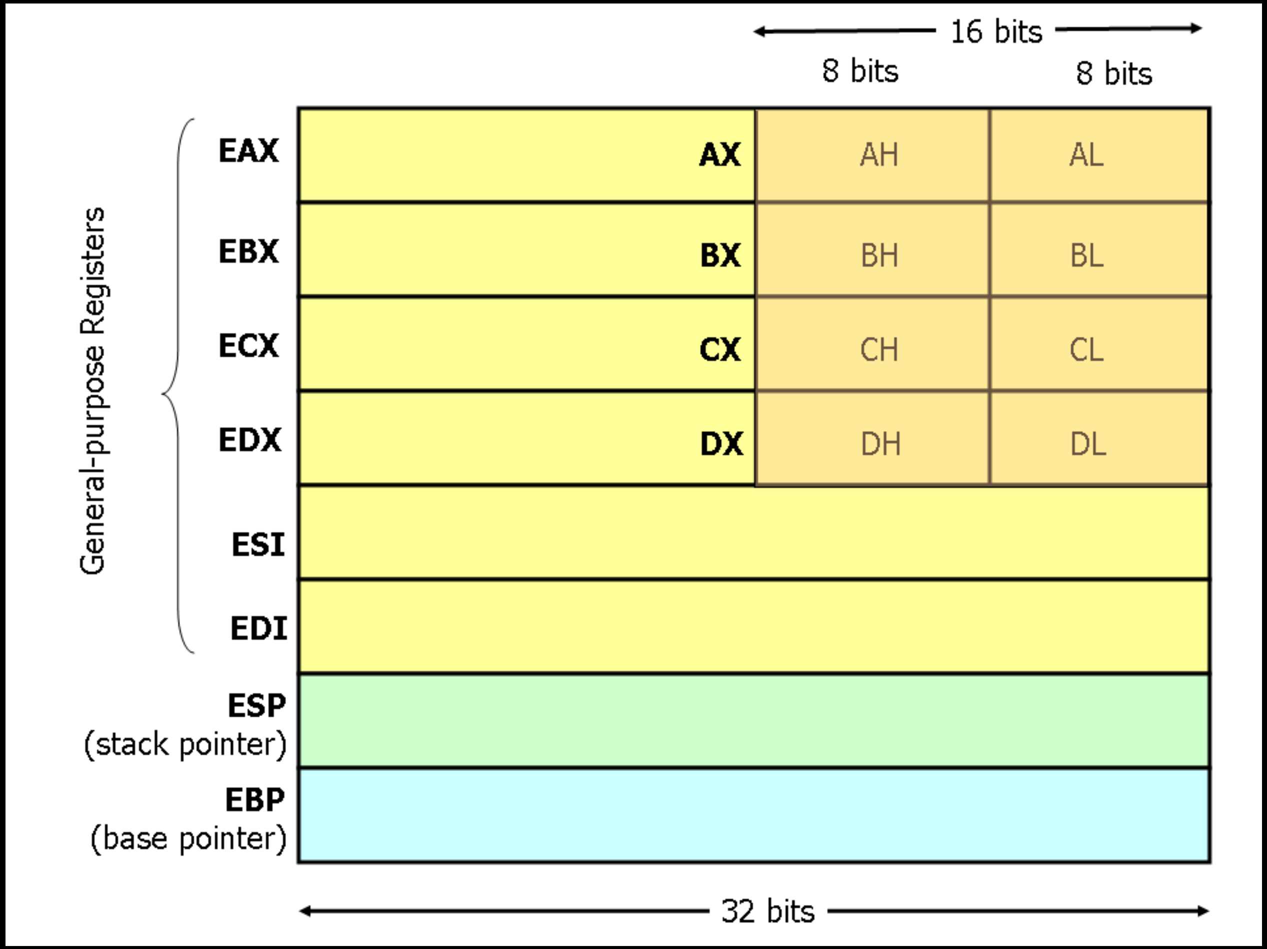
# Who am I?

- ID : bananaapple
- 學校科系 : 交通大學資工系
- 年級 : 大三升大四
- 目前為 Bamboofox 中的一員

# Outline

- Registors
- Flags
- Modes
- Common Instructions
- Intel and AT&T Syntax
- System Call
- Practice
- Example

# Registors



16 bits
8 bits
8 bits
General-purpose Registers
EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
EDI
ESP
(stack pointer)
EBP
(base pointer)
32 bits

# Registors

- eax : accumulator
- ebx : base registor
- ecx : loop counter
- edx : data registor
- esi, edi : index registor
- esp : stack pointer
- ebp : stack base pointer

- eip : instruction pointer

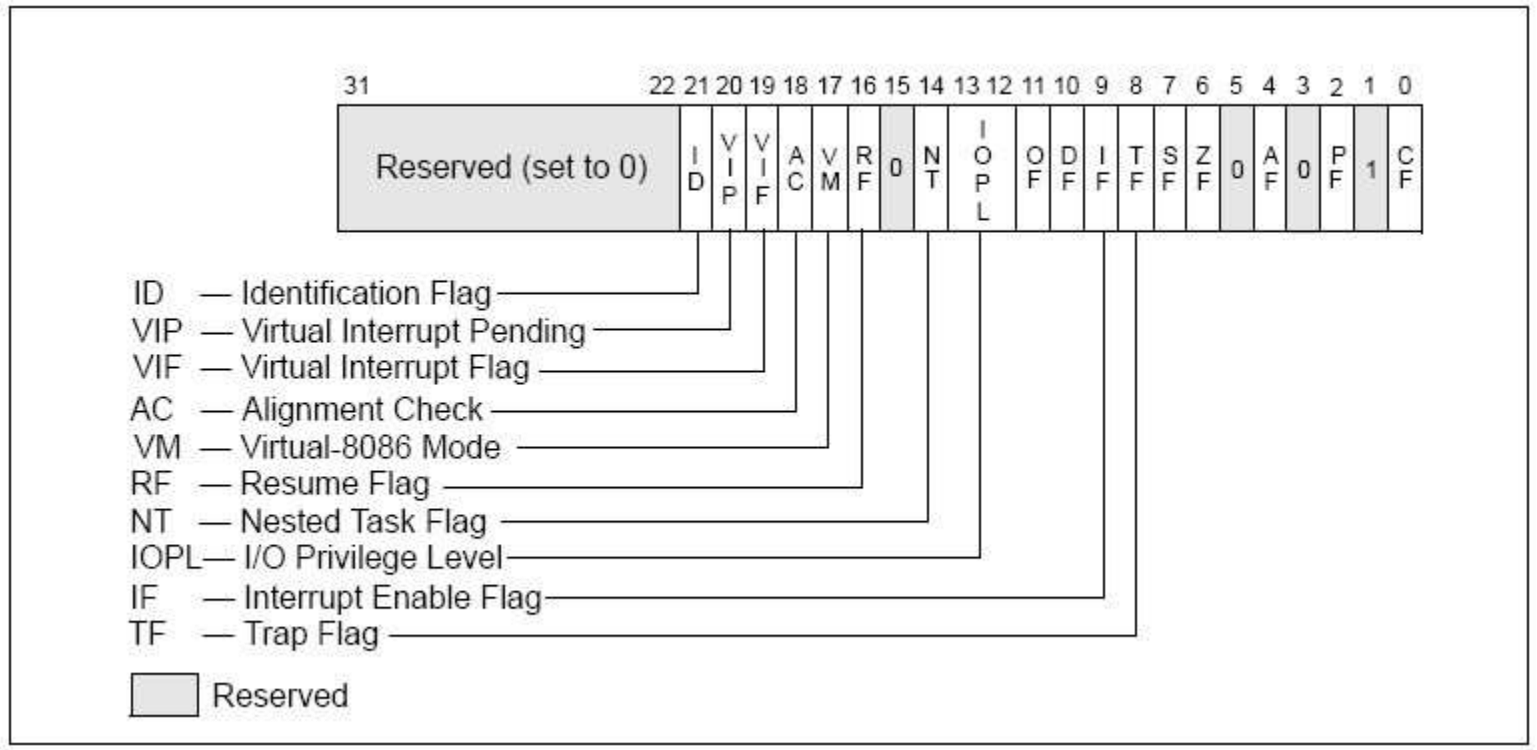
Segment Registers

- cs : code segment
- ds : data segment
- ss : stack segment
- es, fs, gs : additional segment

flags

- Status flag
- Each flag is one bit

# Flags



| 31 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved (set to 0) | | ID | VIP | VIF | AC | VM | RF | 0 | NT | IOPL | | OF | DF | IF | TF | SF | ZF | 0 | AF | 0 | PF | 1 | CF |

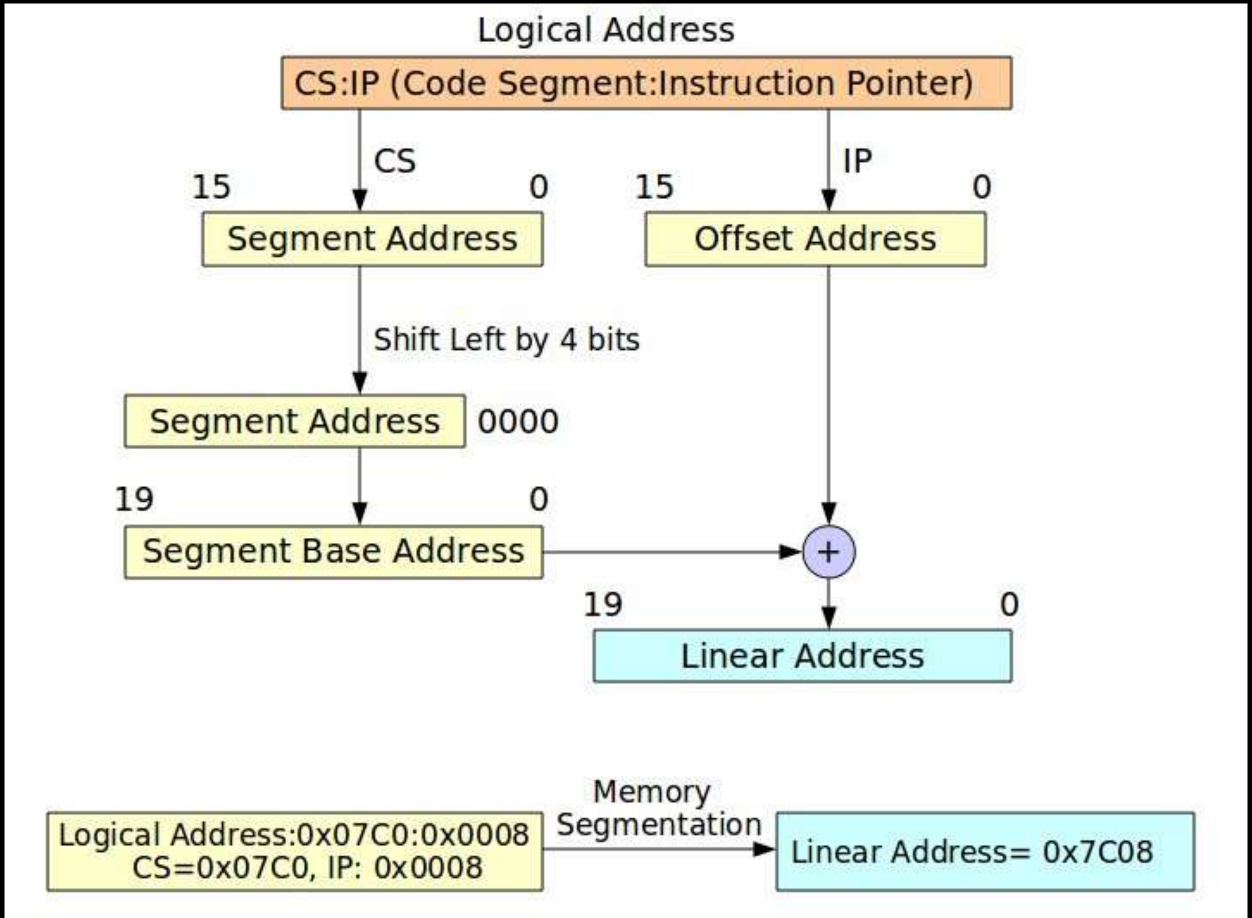
ID — Identification Flag
VIP — Virtual Interrupt Pending
VIF — Virtual Interrupt Flag
AC — Alignment Check
VM — Virtual-8086 Mode
RF — Resume Flag
NT — Nested Task Flag
IOPL— I/O Privilege Level
IF — Interrupt Enable Flag
TF — Trap Flag

▨ Reserved

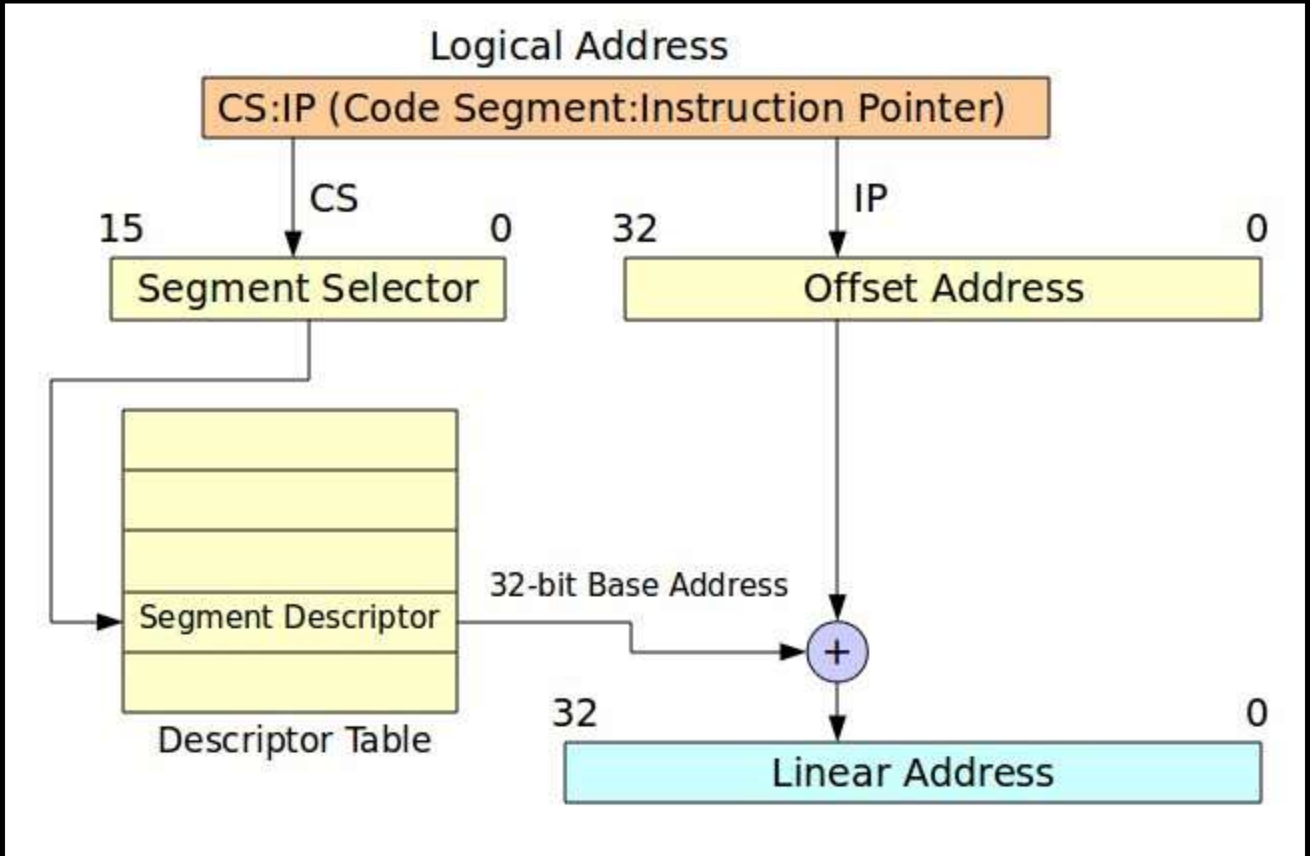
# Modes

- Two Modes, Real Mode and Protect Mode
- Real Mode use two 16 bit registor to represent 20bit address space
- segment:offset => segment << 4 + offset
- Can use up 1MB memory ( 1MB = $2^{20}$ )
- Protect Mode
- segment:offset => Segment Descriptor + offset

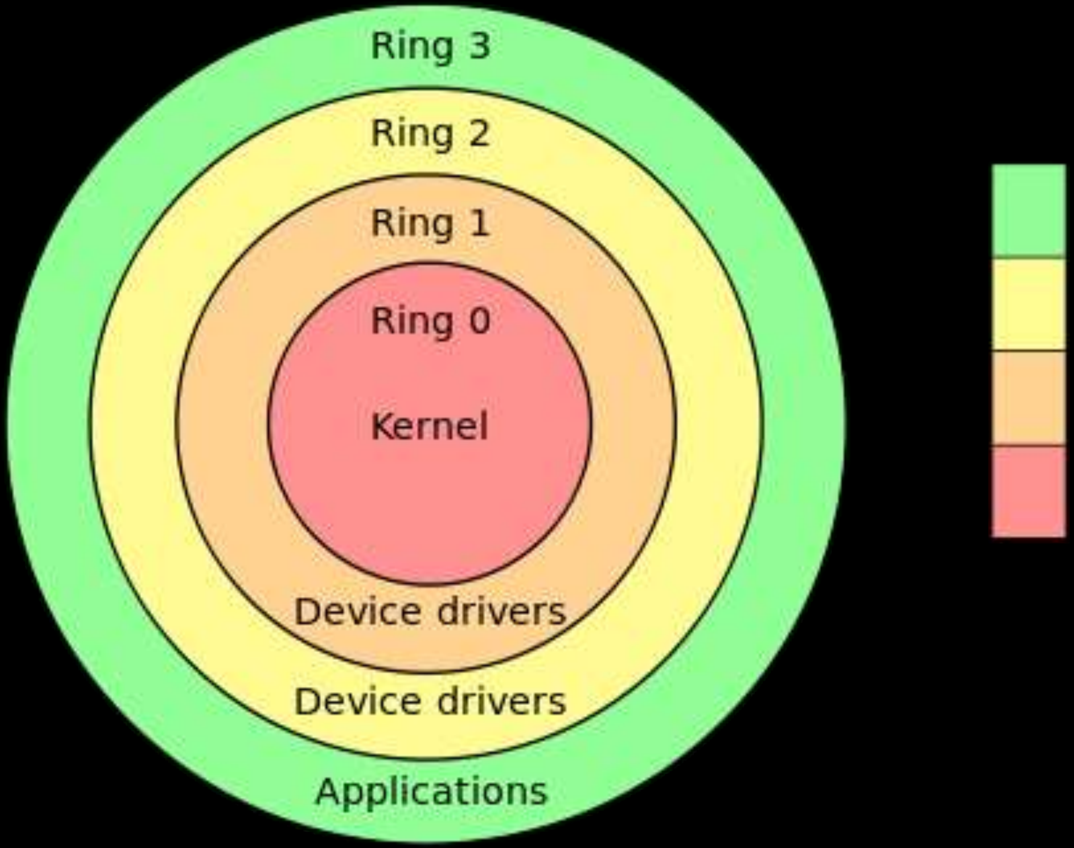
# Real Mode


Logical Address
CS:IP (Code Segment:Instruction Pointer)
CS
IP
15
0
Segment Address
15
0
Offset Address
Shift Left by 4 bits
Segment Address
0000
19
0
Segment Base Address
+
19
0
Linear Address
Logical Address:0x07C0:0x0008
CS=0x07C0, IP: 0x0008
Memory
Segmentation
Linear Address= 0x7C08

# Protect Mode



# Kernel Mode User Mode



# Common Instructions

mov - Move

Syntax

- mov dest, source

Example

- mov eax, [ebx]
- mov eax, [ebp - 4]
- mov [var], ebx

# Common Instructions

push - Push stack

pop - Pop stack

Example

- push eax
- push 0
- pop eax
- pop [ebx]

# Common Instructions

lea - Load effective address

Syntax

- lea <reg32>, <mem>

Example

- lea ebx, [ebx+eax*8]
- lea eax, [ebp-0x44]

# Common Instructions

add, sub, mul, div - Arithmetic

inc ,dec - Increment, Decrement

Syntax

- add dest, source
- inc <reg> or <mem>

Example

- add eax, 10
- inc eax

# Common Instructions

jmp – Jump

- je <label> (jump when equal)
- jne <label> (jump when not equal)
- jz <label> (jump when last result was zero)
- jg <label> (jump when greater than)
- jge <label> (jump when greater than or equal to)
- jl <label> (jump when less than)
- jle <label> (jump when less than or equal to)

# Common Instructions

cmp – Compare

Example

- cmp DWORD PTR [eax], 10
- je loop
- cmp eax, ebx
- jle done
- jmp DWORD PTR [eax]

# Intel and AT&T Syntax

- Prefixes
- Direction of Operands
- Memory Operands
- Suffixes

# Prefixes

## Intex Syntax

- mov eax,1
- mov ebx,0ffh
- int 80h

## AT&T Syntax

- movl $1,%eax
- movl $0xff,%ebx
- int $0x80

# Direction of Operands

## Intex Syntax

- instr dest,source
- mov eax,[ecx]

## AT&T Syntax

- instr source,dest
- movl (%ecx),%eax

# Memory Operands

## Intex Syntax

- mov eax,[ebx]
- mov eax,[ebx+3]

## AT&T Syntax

- movl (%ebx),%eax
- movl 3(%ebx),%eax

# Suffixes

## Intel Syntax

- Instr foo,segreg:[base+index*scale+disp]
- mov eax,[ebx+20h]
- add eax,[ebx+ecx*2h]
- lea eax,[ebx+ecx]
- sub eax,[ebx+ecx*4h-20h]

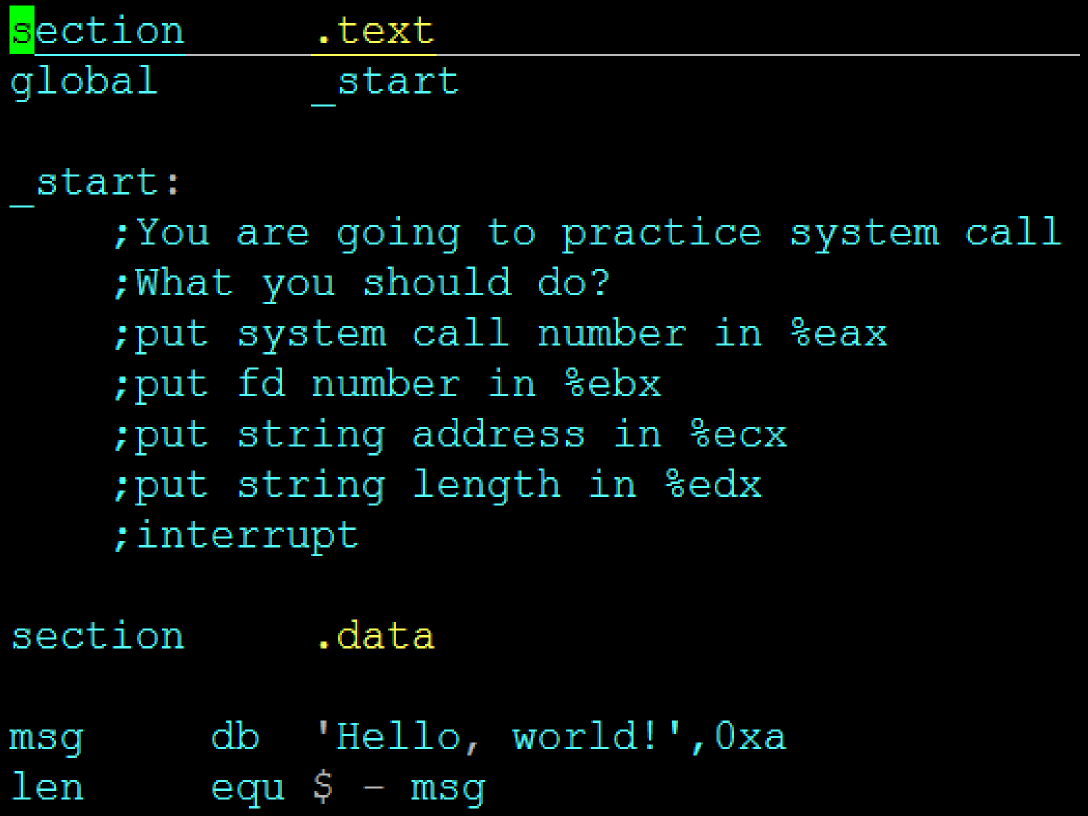
## AT&T Syntax

- Instr %segreg:disp(base,index,scale),foo
- movl 0x20(%ebx),%eax
- addl (%ebx,%ecx,0x2),%eax
- leal (%ebx,%ecx),%eax
- subl -0x20(%ebx,%ecx,0x4),%eax

# System Call

- Syscalls are the interface between user programs and the Linux kernel
- Put value on registers eax, ebx
- eax represent system call number
- ebx, ecx …… represent arguments
- Finally, execute **int 0x80** instruction
- Return value will put on eax register
- If you want to know more about system call, type man 2 system_call (ex:open)
- http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

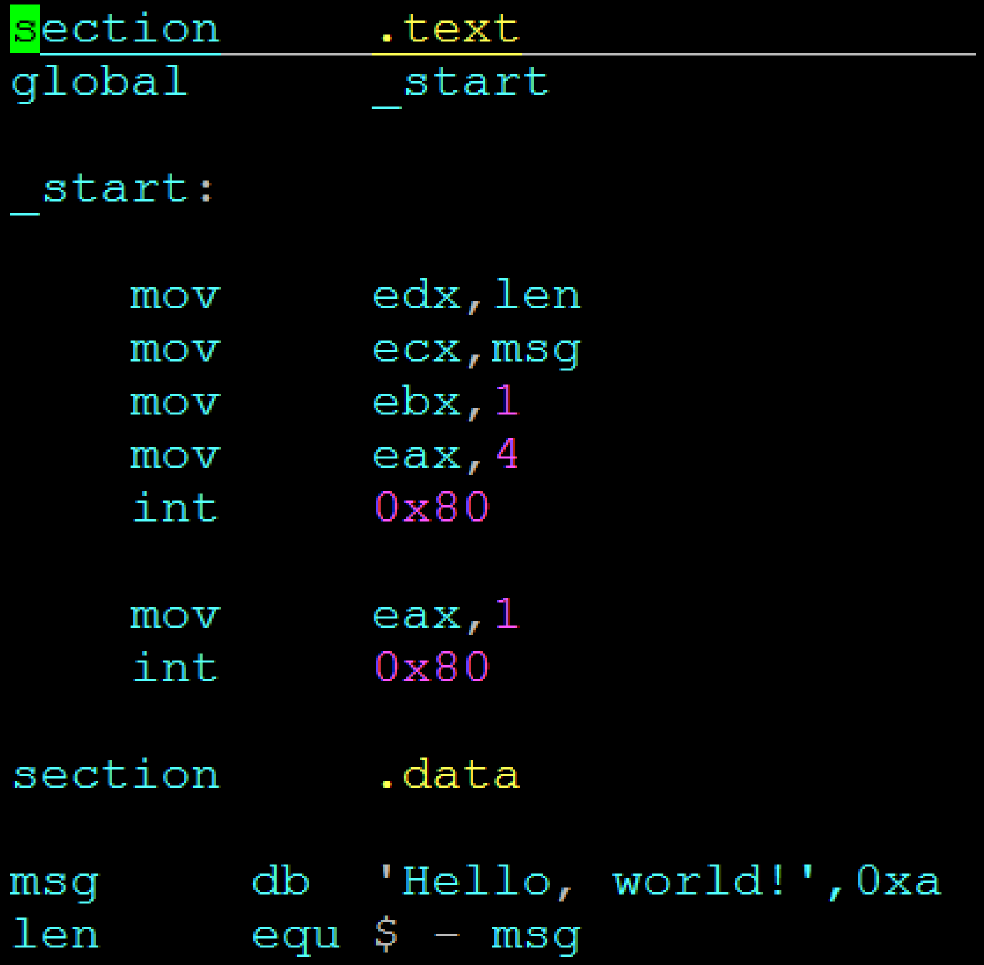
# Practice



```
section     .text
global      _start

_start:
    ;You are going to practice system call
    ;What you should do?
    ;put system call number in %eax
    ;put fd number in %ebx
    ;put string address in %ecx
    ;put string length in %edx
    ;interrupt

section     .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

```
wget http://people.cs.nctu.edu.tw/~wpchen/x86/practice.asm
nasm -f elf practice.asm
ld -m elf_i386 -s -o practice practice.o
./practice
//Hello, world!
```

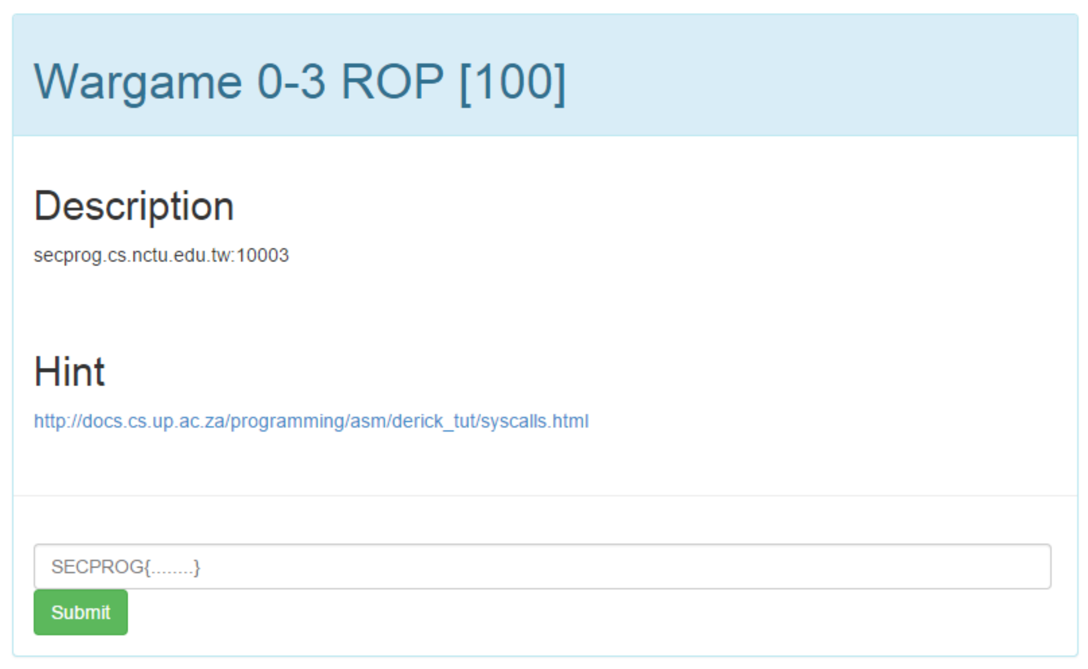
# Answer



```
section         .text
global          _start

_start:

    mov         edx,len
    mov         ecx,msg
    mov         ebx,1
    mov         eax,4
    int         0x80

    mov         eax,1
    int         0x80

section         .data

msg     db   'Hello, world!',0xa
len     equ $ - msg
```

```
wget http://people.cs.nctu.edu.tw/~wpchen/x86/hello.asm
nasm -f elf hello.asm
ld -m elf_i386 -s -o hello hello.o
./hello

//Hello, world!
```

# Not enough?



## Wargame 0-3 ROP [100]

### Description

secprog.cs.nctu.edu.tw:10003

### Hint

http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

Try this one:

http://secprog.cs.nctu.edu.tw/problems/3

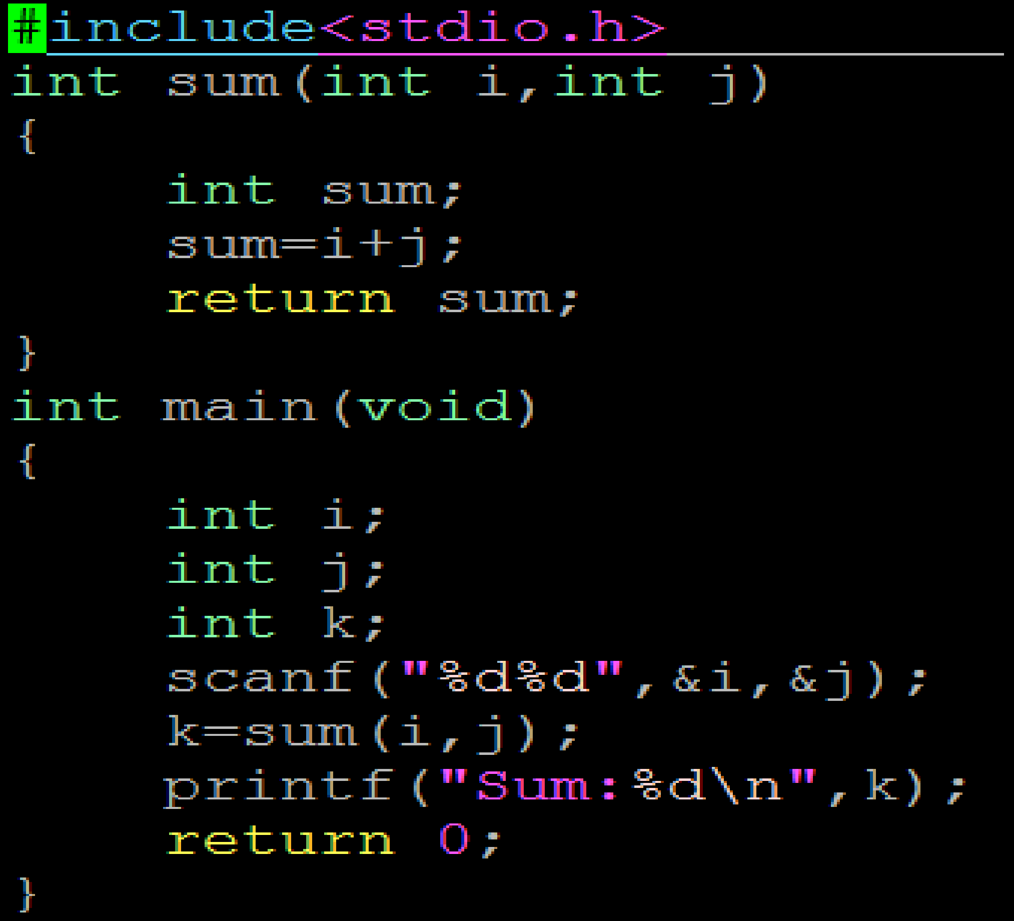
Open your terminal and type:

```
nc secprog.cs.nctu.edu.tw 10003
```

Hint : open /home/rop/flag -> read from fd -> write to stdout

Have fun!!!

# Example



```c
#include<stdio.h>
int sum(int i,int j)
{
    int sum;
    sum=i+j;
    return sum;
}
int main(void)
{
    int i;
    int j;
    int k;
    scanf("%d%d",&i,&j);
    k=sum(i,j);
    printf("Sum:%d\n",k);
    return 0;
}
```

wget http://people.cs.nctu.edu.tw/~wpchen/x86/sum.c

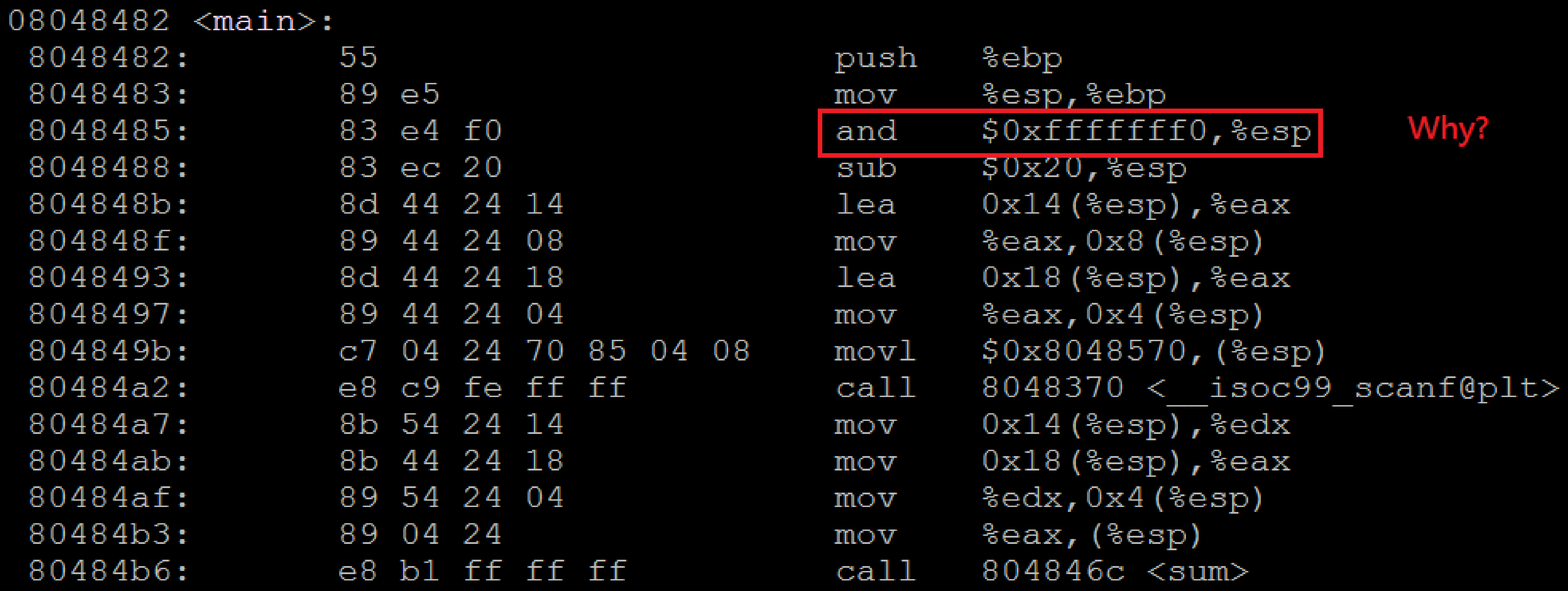
gcc -m32 -o sum sum.c

//or just download it

wget http://people.cs.nctu.edu.tw/~wpchen/sum

objdump -d sum | less

# Example



```
08048482 <main>:
 8048482:       55                      push   %ebp
 8048483:       89 e5                   mov    %esp,%ebp
 8048485:       83 e4 f0                and    $0xfffffff0,%esp
 8048488:       83 ec 20                sub    $0x20,%esp
 804848b:       8d 44 24 14             lea    0x14(%esp),%eax
 804848f:       89 44 24 08             mov    %eax,0x8(%esp)
 8048493:       8d 44 24 18             lea    0x18(%esp),%eax
 8048497:       89 44 24 04             mov    %eax,0x4(%esp)
 804849b:       c7 04 24 70 85 04 08    movl   $0x8048570,(%esp)
 80484a2:       e8 c9 fe ff ff          call   8048370 <__isoc99_scanf@plt>
 80484a7:       8b 54 24 14             mov    0x14(%esp),%edx
 80484ab:       8b 44 24 18             mov    0x18(%esp),%eax
 80484af:       89 54 24 04             mov    %edx,0x4(%esp)
 80484b3:       89 04 24                mov    %eax,(%esp)
 80484b6:       e8 b1 ff ff ff          call   804846c <sum>
```

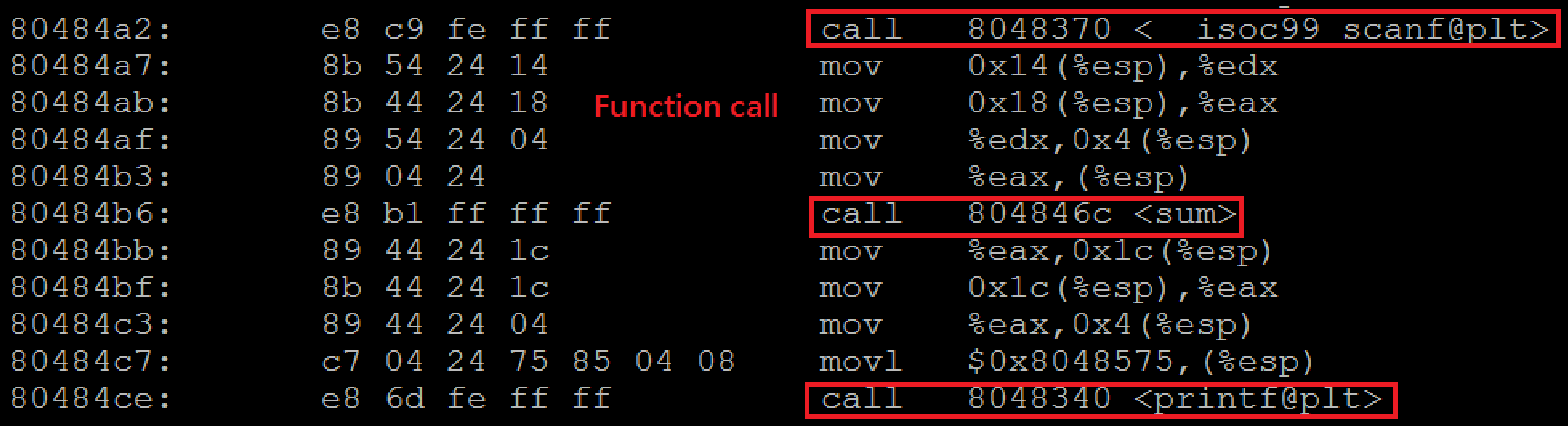
Why?

# Answer

This code makes sure that the stack is aligned to 16 bytes. After this operation esp will be less than or equal to what it was before this operation, so the stack may grow, which protects anything that might already be on the stack. This is sometimes done in main just in case the function is called with an unaligned stack, which can cause things to be really slow (16 byte is a cache line width on x86, I think, though 4 byte alignment is what is really important here). If main has a unaligned stack the rest of the program will too.

http://stackoverflow.com/questions/4228261/understanding-the-purpose-of-some-assembly-statements

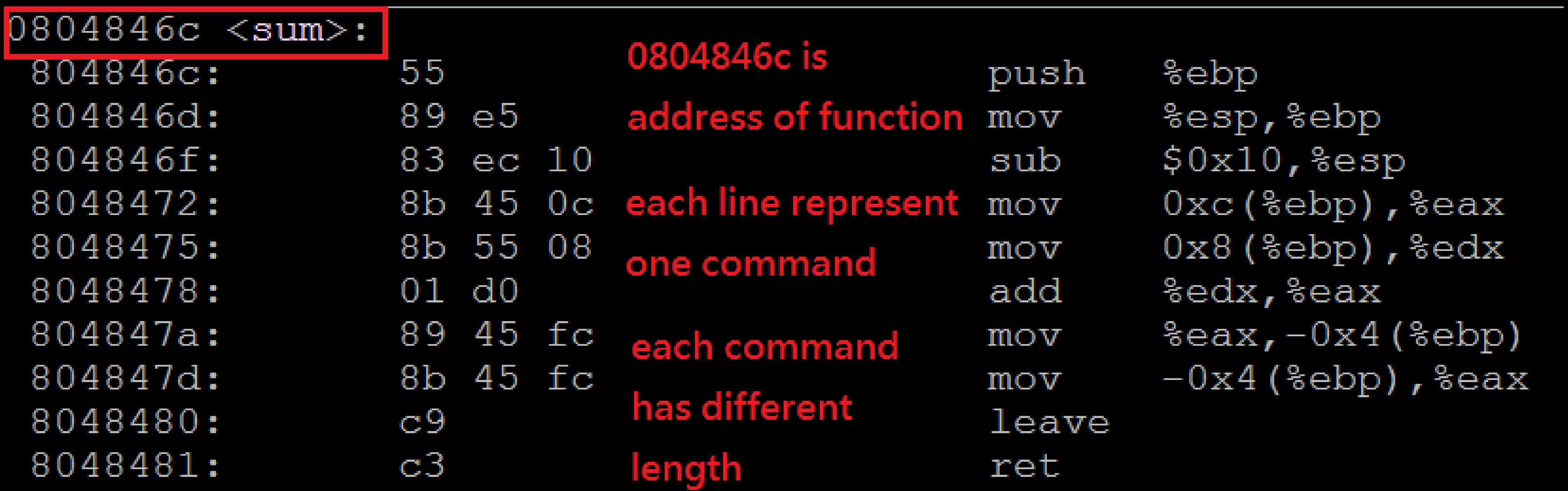
# Example



```
80484a2:       e8 c9 fe ff ff          call   8048370 <__isoc99_scanf@plt>
80484a7:       8b 54 24 14             mov    0x14(%esp),%edx
80484ab:       8b 44 24 18             mov    0x18(%esp),%eax
80484af:       89 54 24 04             mov    %edx,0x4(%esp)
80484b3:       89 04 24                mov    %eax,(%esp)
80484b6:       e8 b1 ff ff ff          call   804846c <sum>
80484bb:       89 44 24 1c             mov    %eax,0x1c(%esp)
80484bf:       8b 44 24 1c             mov    0x1c(%esp),%eax
80484c3:       89 44 24 04             mov    %eax,0x4(%esp)
80484c7:       c7 04 24 75 85 04 08    movl   $0x8048575,(%esp)
80484ce:       e8 6d fe ff ff          call   8048340 <printf@plt>
```

Function call

# Example



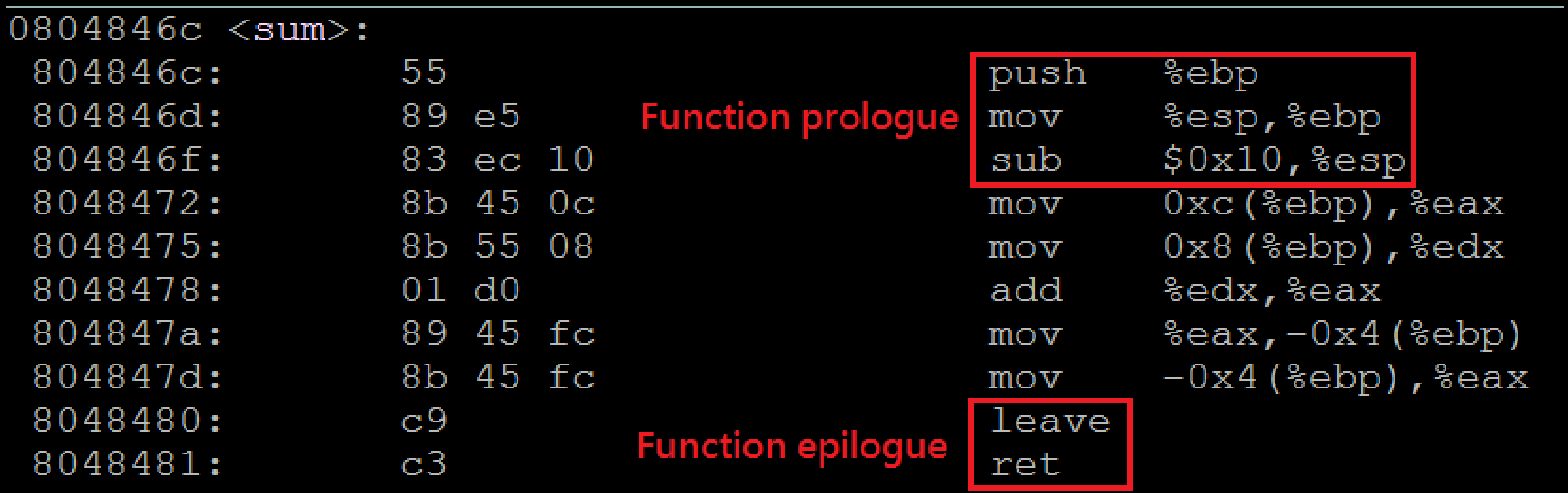
```
08048846c <sum>:
 804846c:       55                      push   %ebp
 804846d:       89 e5                   mov    %esp,%ebp
 804846f:       83 ec 10                sub    $0x10,%esp
 8048472:       8b 45 0c                mov    0xc(%ebp),%eax
 8048475:       8b 55 08                mov    0x8(%ebp),%edx
 8048478:       01 d0                   add    %edx,%eax
 804847a:       89 45 fc                mov    %eax,-0x4(%ebp)
 804847d:       8b 45 fc                mov    -0x4(%ebp),%eax
 8048480:       c9                      leave
 8048481:       c3                      ret
```

0804846c is address of function

each line represent one command

each command has different length

# Example

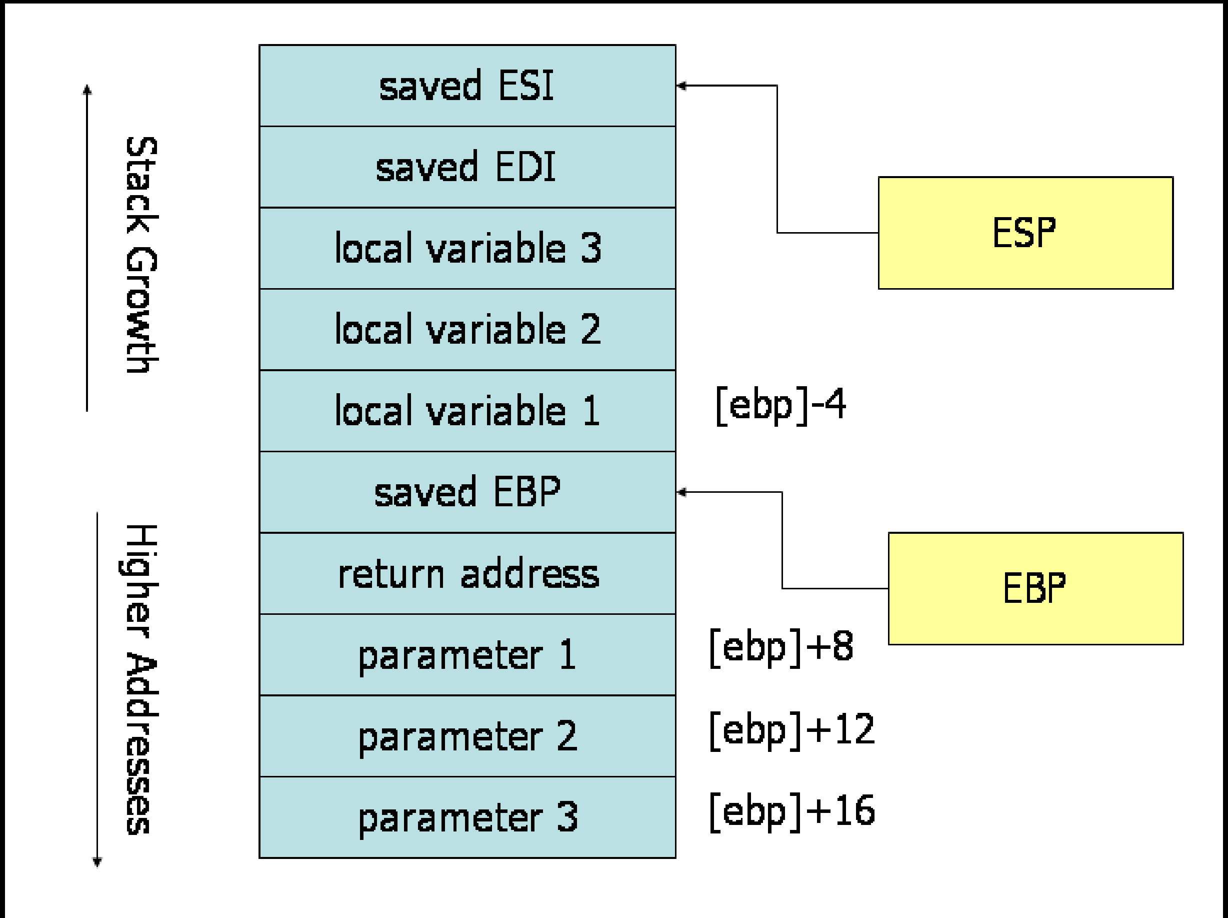


```
0804846c <sum>:
 804846c:       55                      push   %ebp
 804846d:       89 e5                   mov    %esp,%ebp
 804846f:       83 ec 10                sub    $0x10,%esp
 8048472:       8b 45 0c                mov    0xc(%ebp),%eax
 8048475:       8b 55 08                mov    0x8(%ebp),%edx
 8048478:       01 d0                   add    %edx,%eax
 804847a:       89 45 fc                mov    %eax,-0x4(%ebp)
 804847d:       8b 45 fc                mov    -0x4(%ebp),%eax
 8048480:       c9                      leave
 8048481:       c3                      ret
```

Function prologue

Function epilogue

# Example



Stack Growth

Higher Addresses

saved ESI

saved EDI

local variable 3

local variable 2

local variable 1 — [ebp]-4

saved EBP

return address

parameter 1 — [ebp]+8

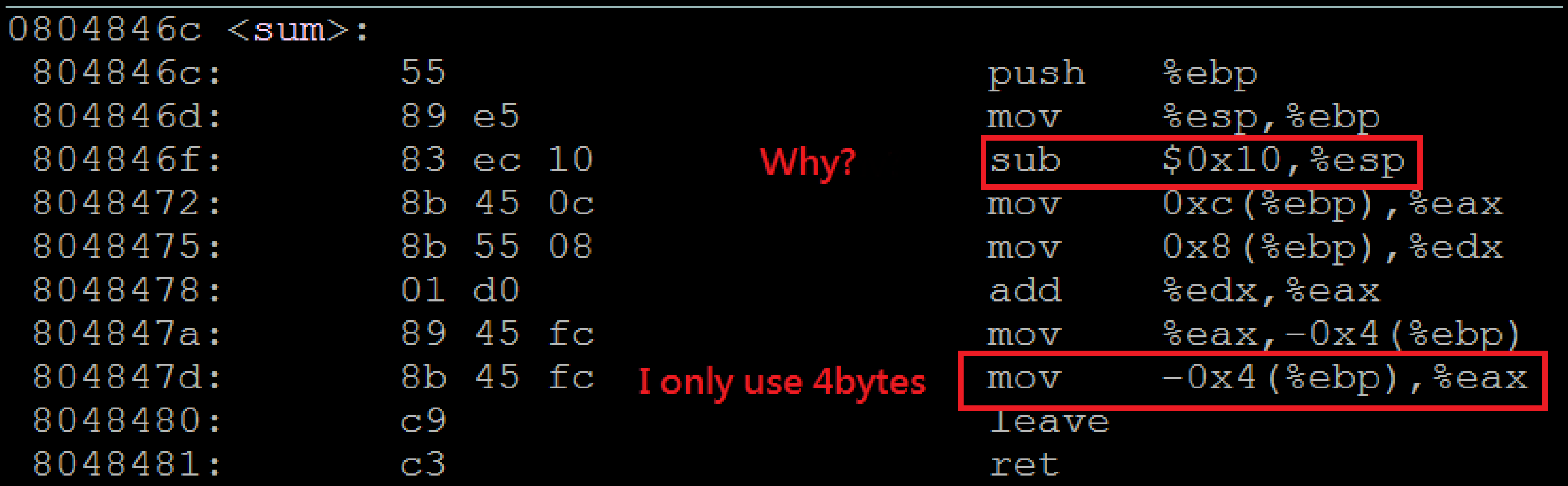
parameter 2 — [ebp]+12

parameter 3 — [ebp]+16

ESP

EBP

# Example



```
0804846c <sum>:
 804846c:       55                      push   %ebp
 804846d:       89 e5                   mov    %esp,%ebp
 804846f:       83 ec 10                sub    $0x10,%esp
 8048472:       8b 45 0c                mov    0xc(%ebp),%eax
 8048475:       8b 55 08                mov    0x8(%ebp),%edx
 8048478:       01 d0                   add    %edx,%eax
 804847a:       89 45 fc                mov    %eax,-0x4(%ebp)
 804847d:       8b 45 fc                mov    -0x4(%ebp),%eax
 8048480:       c9                      leave
 8048481:       c3                      ret
```

Why?

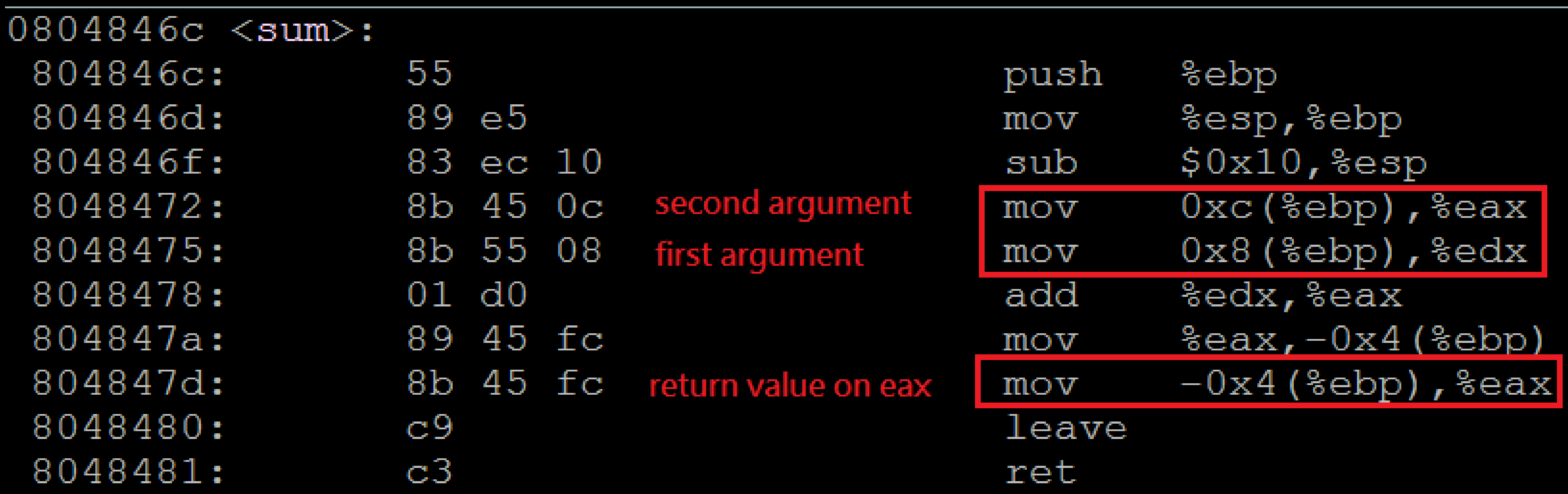
I only use 4bytes

# Answer

Sometimes , compiler will optimize the code by adding some padding to make it align to word boundary

You have to inspect the assembly code to know the exactly stack position

# Example



```
0804846c <sum>:
 804846c:      55                                        push   %ebp
 804846d:      89 e5                                     mov    %esp,%ebp
 804846f:      83 ec 10                                  sub    $0x10,%esp
 8048472:      8b 45 0c     second argument              mov    0xc(%ebp),%eax
 8048475:      8b 55 08     first argument               mov    0x8(%ebp),%edx
 8048478:      01 d0                                     add    %edx,%eax
 804847a:      89 45 fc                                  mov    %eax,-0x4(%ebp)
 804847d:      8b 45 fc     return value on eax          mov    -0x4(%ebp),%eax
 8048480:      c9                                        leave
 8048481:      c3                                        ret
```

# Example

- Intel and AT&T Syntax

http://asm.sourceforge.net/articles/linasm.html

- hello.asm

http://asm.sourceforge.net/intro/hello.html

- Stack overflow

http://stackoverflow.com/questions/4228261/understanding-the-purpose-of-some-assembly-statements

# Reference

- x86 Assembly Guide ( recommended )

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

- Linux System Call Table

http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

- Wiki

https://en.wikipedia.org/wiki/X86_assembly_language

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux