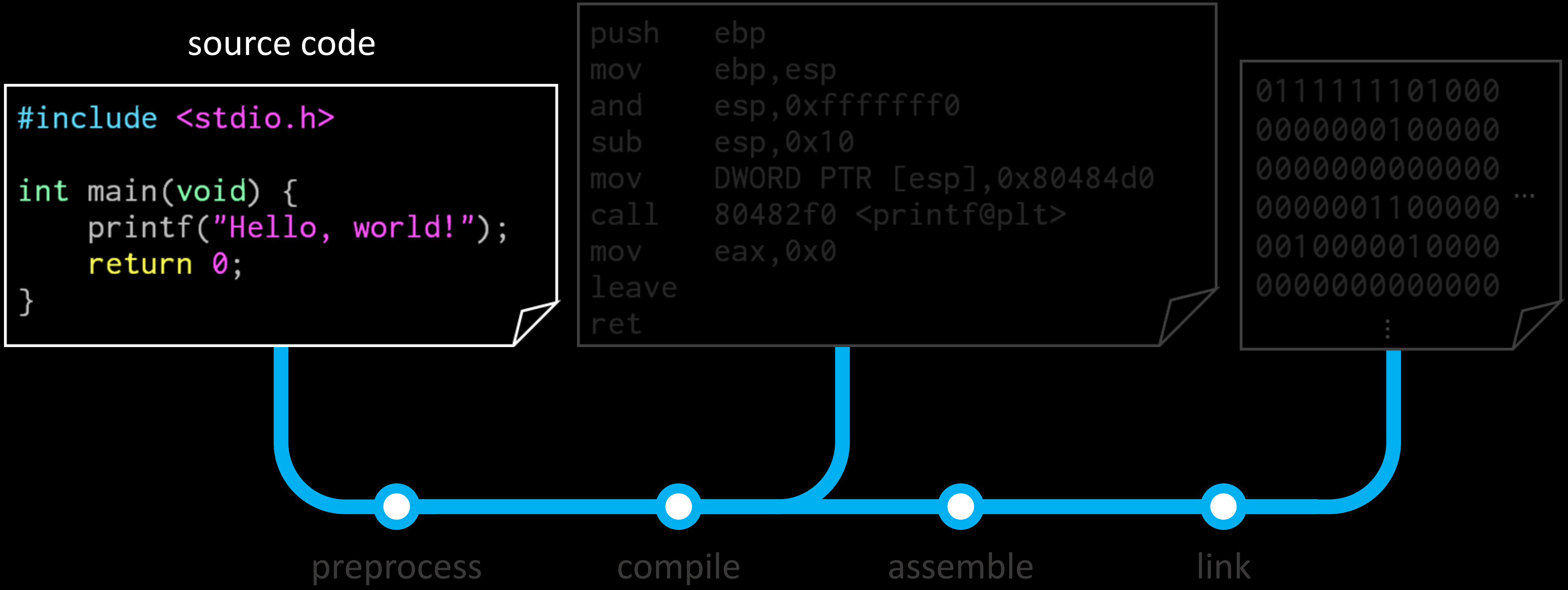
x86 assembly & GDB

briansp8210
briantaipei8210@gmail.com

How to create an executable file



source code

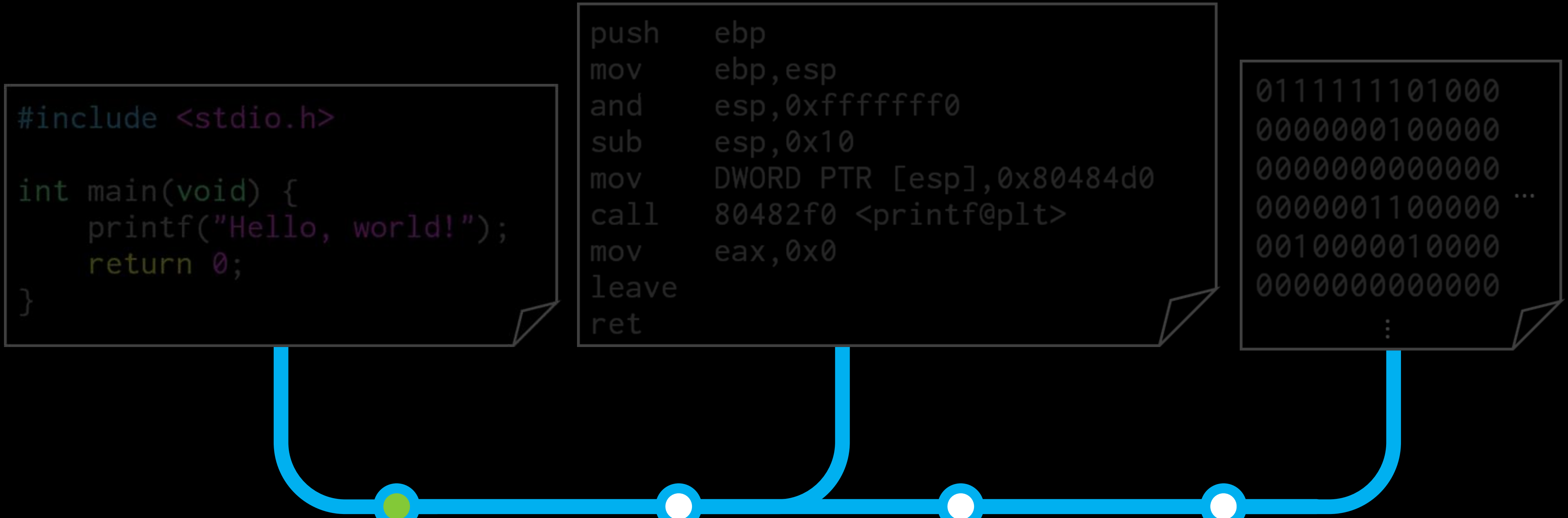
```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!");
    return 0;
}
```

```
push    ebp
mov     ebp,esp
and     esp,0xfffffff0
sub     esp,0x10
mov     DWORD PTR [esp],0x80484d0
call    80482f0 <printf@plt>
mov     eax,0x0
leave
ret
```

```
0111111101000
0000000100000
0000000000000
0000001100000 ...
0010000010000
0000000000000
      ⋮
```

How to create an executable file



```
#include <stdio.h>

int main(void) {
    printf("Hello, world!");
    return 0;
}
```

```
push    ebp
mov     ebp,esp
and     esp,0xfffffff0
sub     esp,0x10
mov     DWORD PTR [esp],0x80484d0
call    80482f0 <printf@plt>
mov     eax,0x0
leave
ret
```

```
0111111101000
0000000100000
0000000000000
0000001100000 …
0010000010000
0000000000000
      ⋮
```

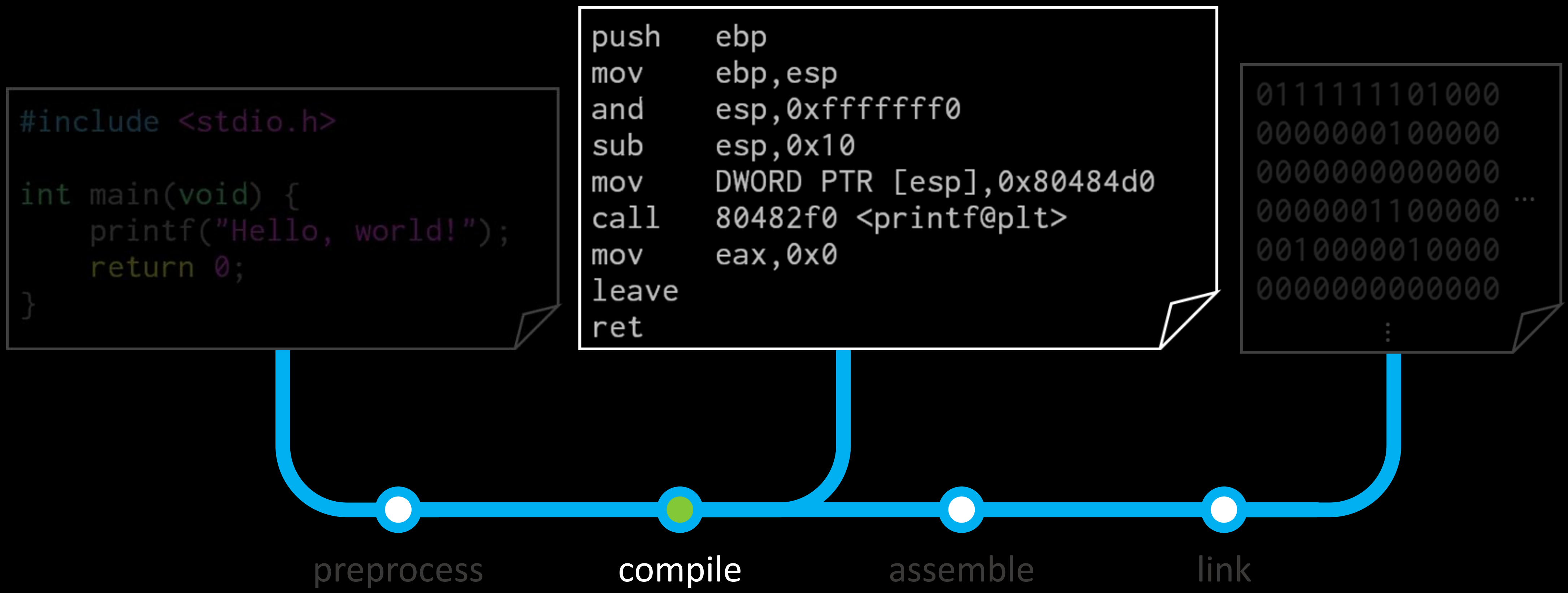
preprocess

compile

assemble

link

How to create an executable file



```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!");
    return 0;
}
```

assembly language

```
push    ebp
mov     ebp,esp
and     esp,0xfffffff0
sub     esp,0x10
mov     DWORD PTR [esp],0x80484d0
call    80482f0 <printf@plt>
mov     eax,0x0
leave
ret
```

```
0111111101000
0000000100000
0000000000000
0000001100000  …
0010000010000
0000000000000
      ⋮
```

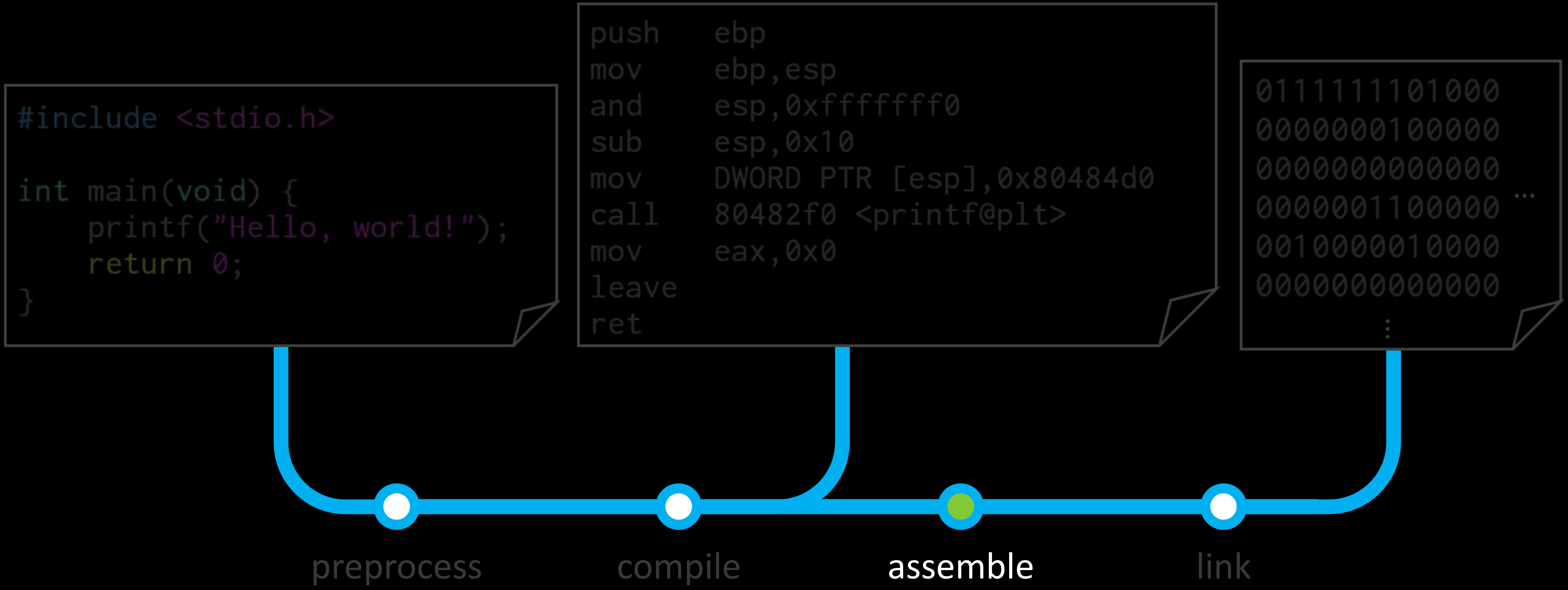
preprocess

compile

assemble

link

How to create an executable file



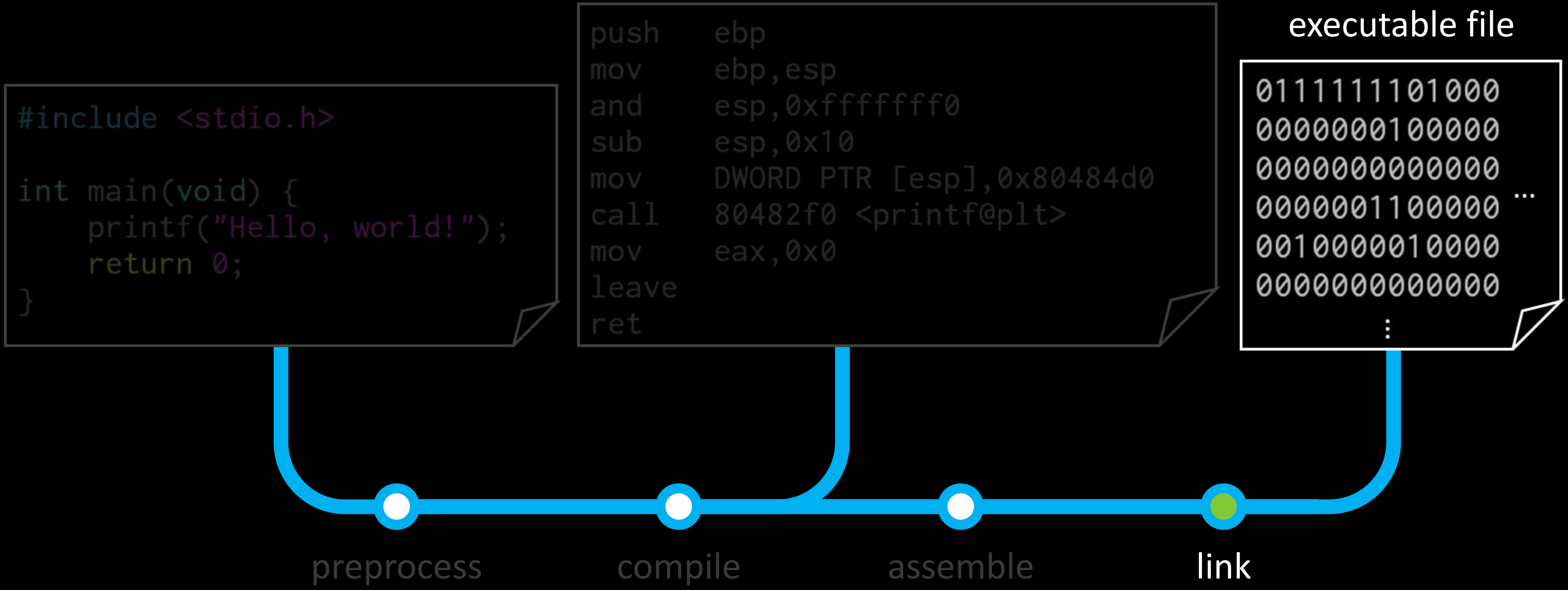
```
#include <stdio.h>

int main(void) {
    printf("Hello, world!");
    return 0;
}
```

```
push    ebp
mov     ebp,esp
and     esp,0xfffffff0
sub     esp,0x10
mov     DWORD PTR [esp],0x80484d0
call    80482f0 <printf@plt>
mov     eax,0x0
leave
ret
```

```
0111111101000
0000000100000
0000000000000
0000001100000 ...
0010000010000
0000000000000
      ⋮
```

How to create an executable file



```
#include <stdio.h>

int main(void) {
    printf("Hello, world!");
    return 0;
}
```

```
push    ebp
mov     ebp,esp
and     esp,0xfffffff0
sub     esp,0x10
mov     DWORD PTR [esp],0x80484d0
call    80482f0 <printf@plt>
mov     eax,0x0
leave
ret
```

executable file

```
01111111010000
00000001000000
00000000000000
00000001100000 ...
00100000010000
00000000000000
       ⋮
```

Intel vs. AT&T syntax

Intel	AT&T
mov eax, 1	movl $1, %eax
mov eax, [ebx+3]	movl 3(%ebx), %eax
add eax, [ebx+ecx*2h]	addl (%ebx,%ecx,0x2), %eax

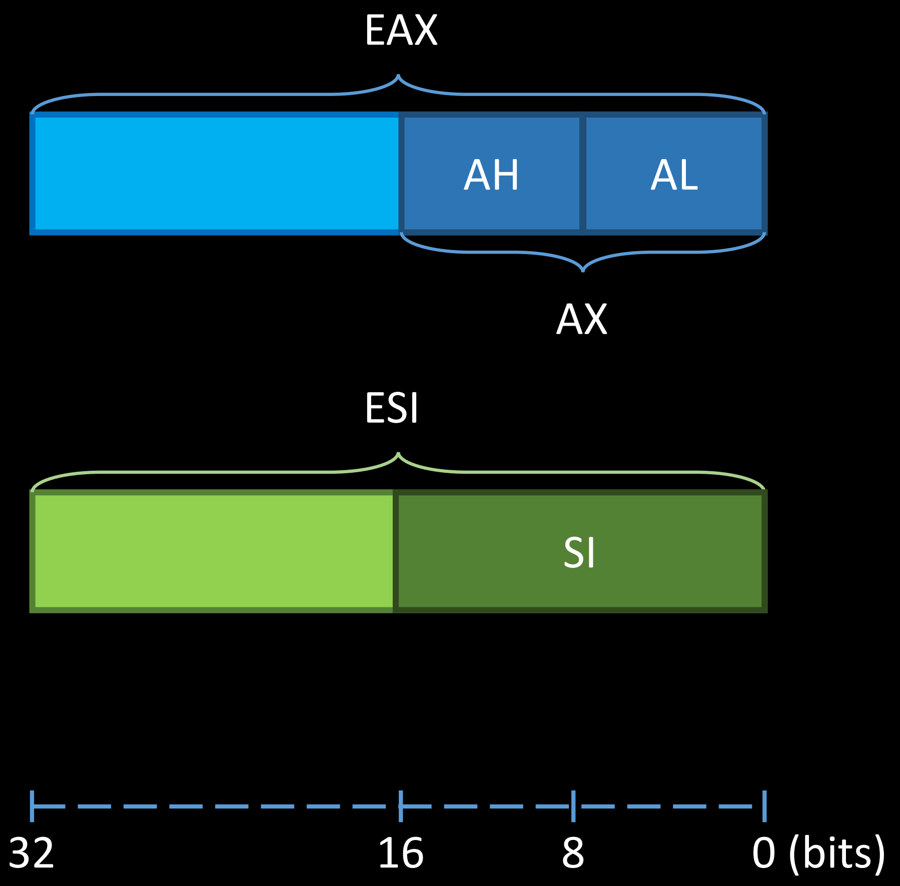
- We prefer to use Intel syntax

➢ `objdump –M intel`

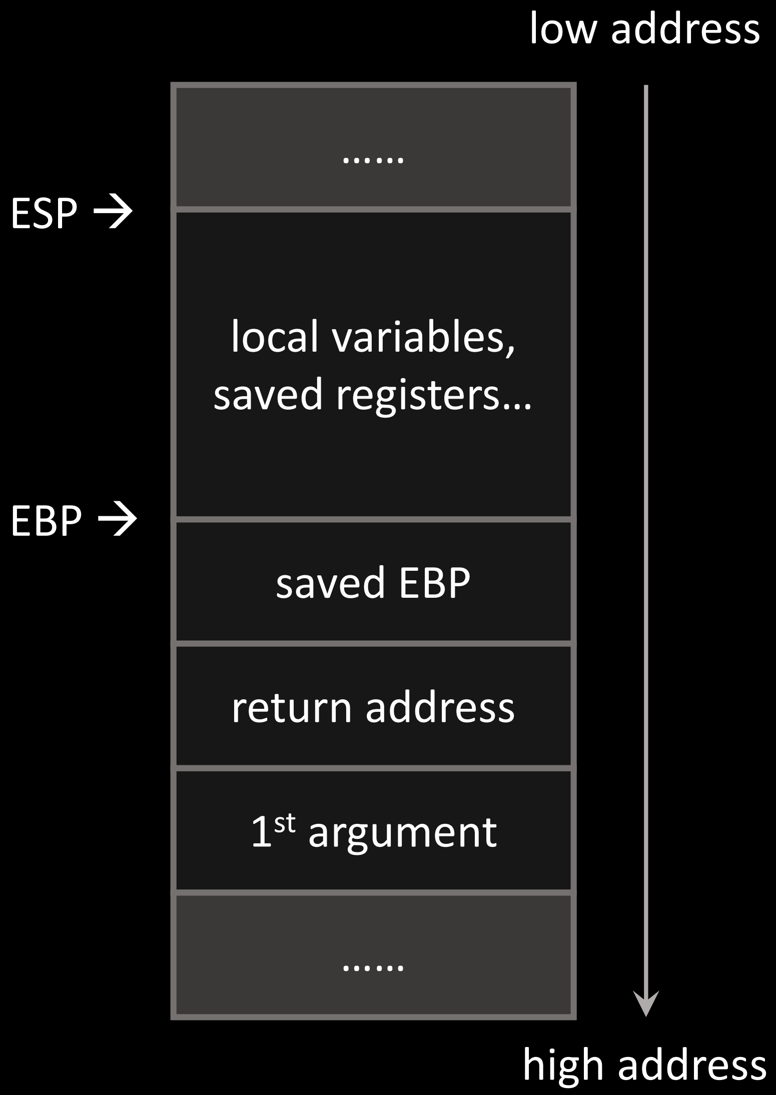
➢ `set disassembly-flavor intel`

Register

- General purpose register
  - EAX EBX ECX EDX ESI EDI ESP EBP
  - mainly used for arithmetic and data movement
  - some have special usage:
    - ✓ return value of function will be stored in EAX
    - ✓ system call number is indicated by EAX
    - ✓ ESP points to top of current stack frame
    - ✓ EBP points to base of current stack frame



Register



low address

……

ESP →

local variables, saved registers…

EBP →

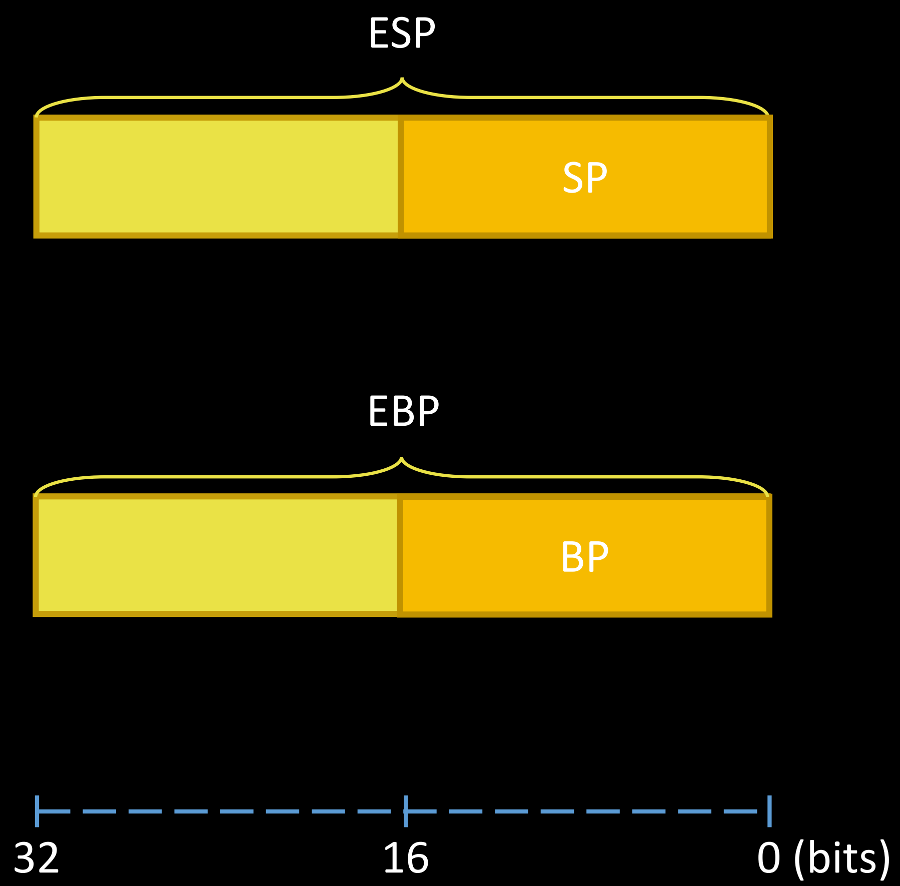
saved EBP

return address

1st argument

……

high address



ESP

SP

EBP

BP

32

16

0 (bits)

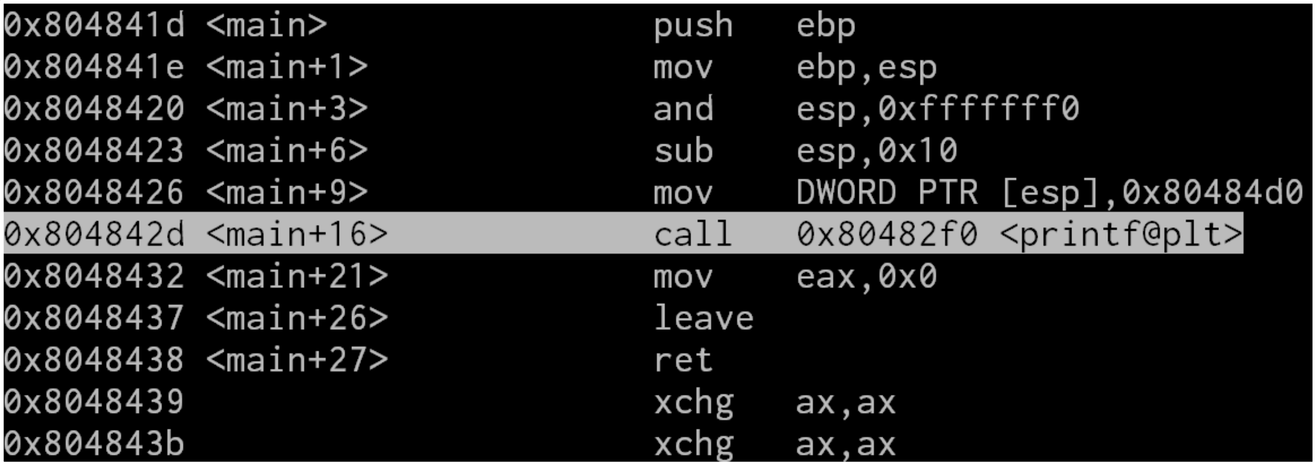
Register

- Instruction pointer

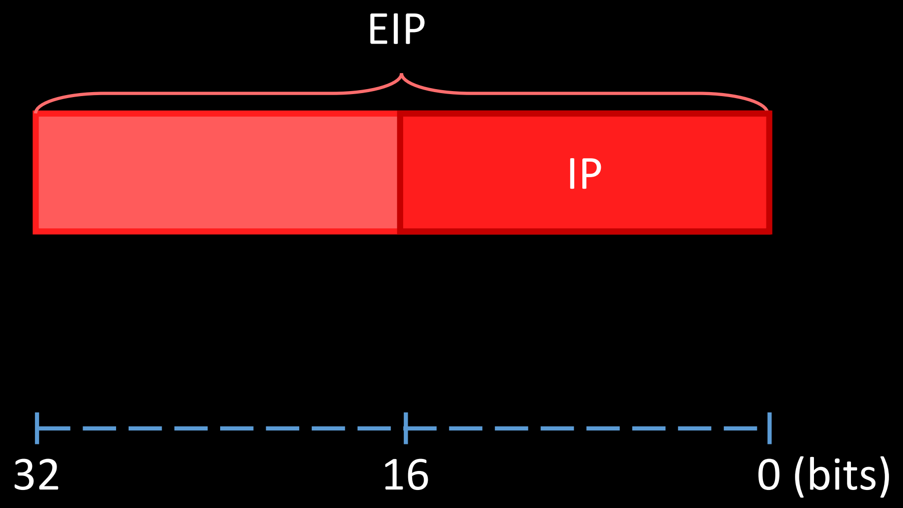
➢ EIP

➢ point to the next instruction to be executed

EIP →



```
0x804841d <main>                push   ebp
0x804841e <main+1>              mov    ebp,esp
0x8048420 <main+3>              and    esp,0xfffffff0
0x8048423 <main+6>              sub    esp,0x10
0x8048426 <main+9>              mov    DWORD PTR [esp],0x80484d0
0x804842d <main+16>             call   0x80482f0 <printf@plt>
0x8048432 <main+21>             mov    eax,0x0
0x8048437 <main+26>             leave
0x8048438 <main+27>             ret
0x8048439                       xchg   ax,ax
0x804843b                       xchg   ax,ax
```



Register

- EFLAGS register

➢ EFLAGS

➢ contain a collection of flags indicating state of processor

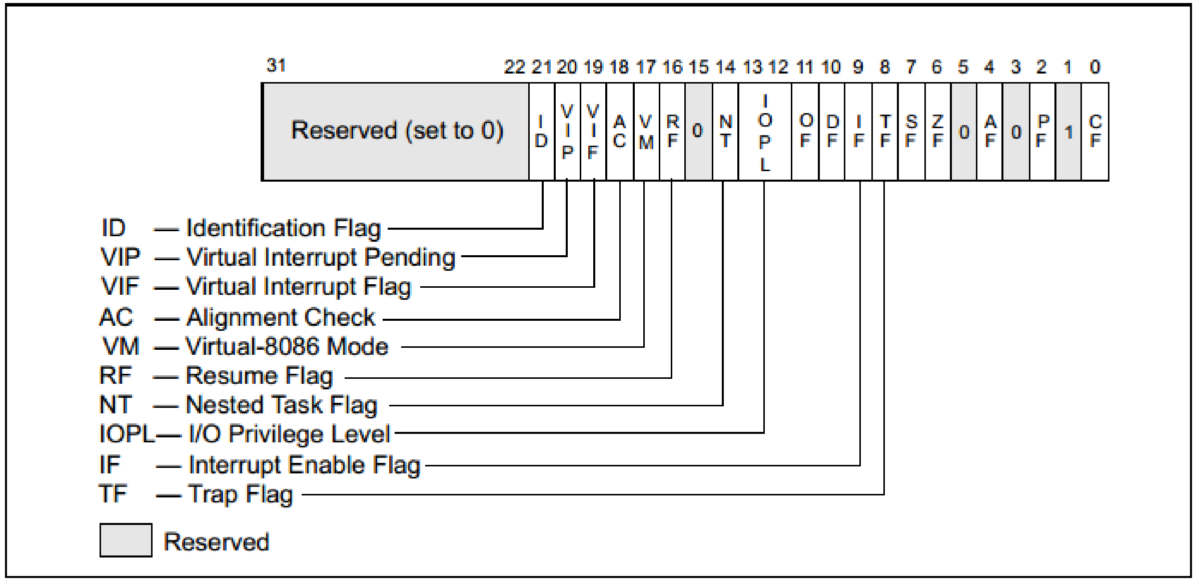


Figure 2-4. System Flags in the EFLAGS Register

mov

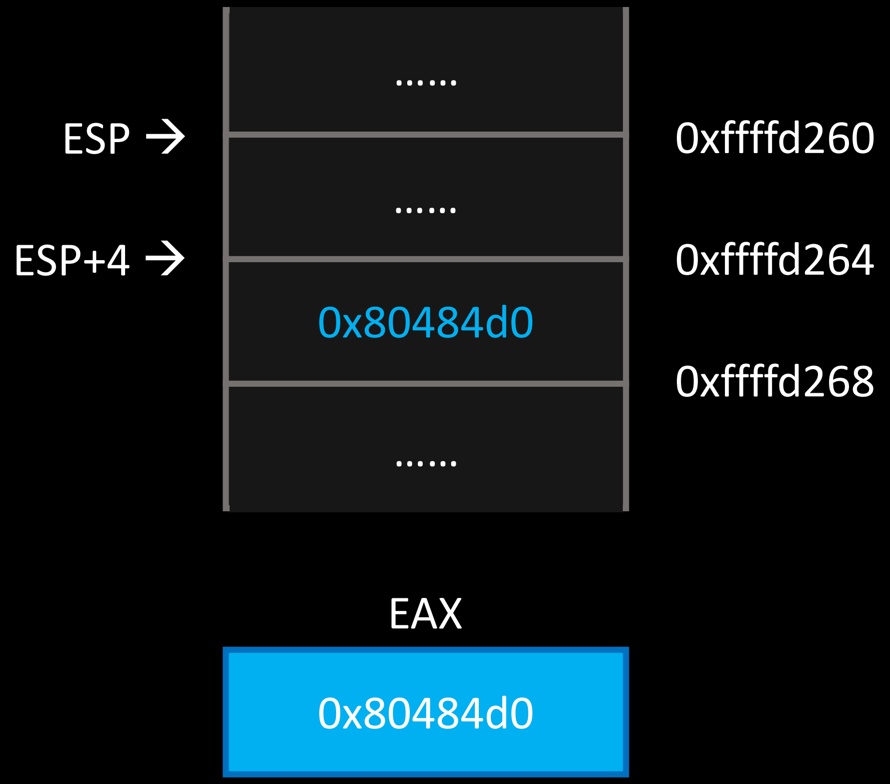
- assign value of src. operand to dest. operand
- size of operands must be the same

➢ `mov dest. src.`

➢ `mov ebp, esp`

➢ `mov BYTE PTR [edi], 0x41`

➢ `mov eax, DWORD PTR [esp+4]`

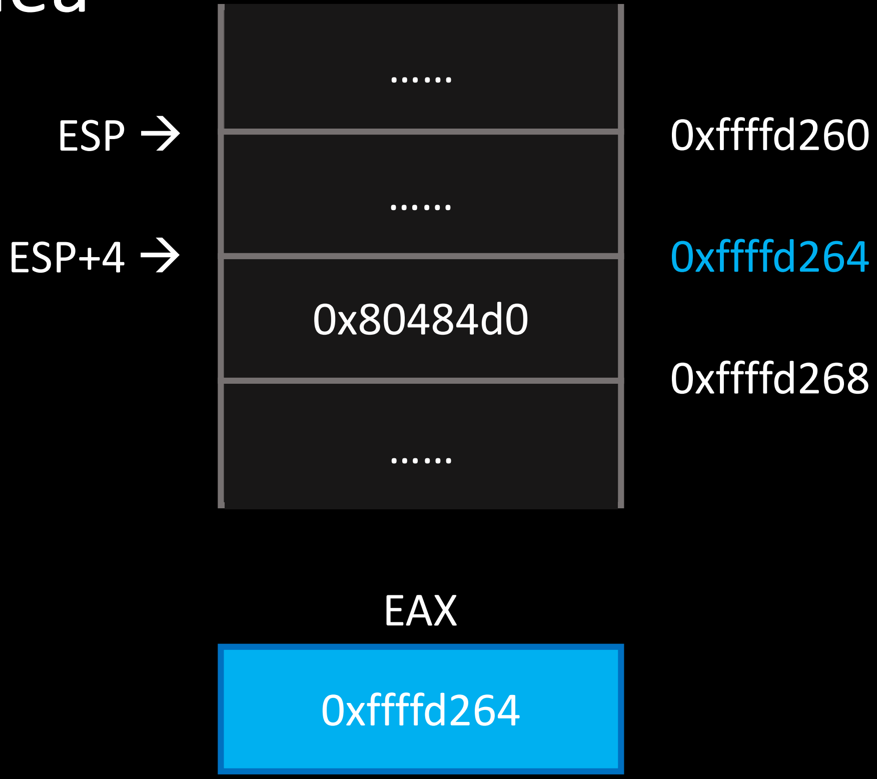


lea

- assign effective address of src. operand to dest. operand
- note the difference between mov and lea

➢ **lea dest. src.**

➢ `lea eax, DWORD PTR [esp+4]`



add / sub

- store the sum / difference of two operands to dest. operand

➢ **`add/sub dest. src.`**

➢ `sub esp, 0x18`

➢ `add DWORD PTR [edi], edx`

➢ `cmp eax, 1       ; check whether eax is 1`

and / or / xor

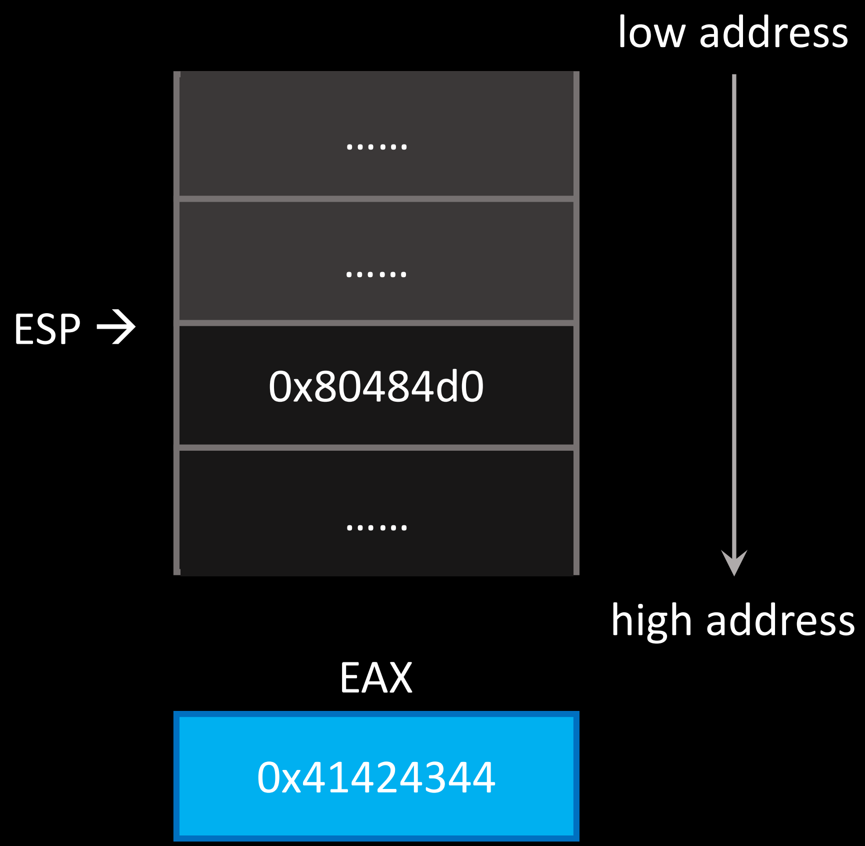
- store the result of bitwise operation of two operands to dest.

- and/or/xor dest. src.
- and dl, 11101100b   ; clear 1st, 2nd and 5th bits of dl
- or dl, 00100000b   ; set 6th bit of dl
- xor eax, eax   ; set eax to 0
- test eax, eax   ; check whether eax is 0

push

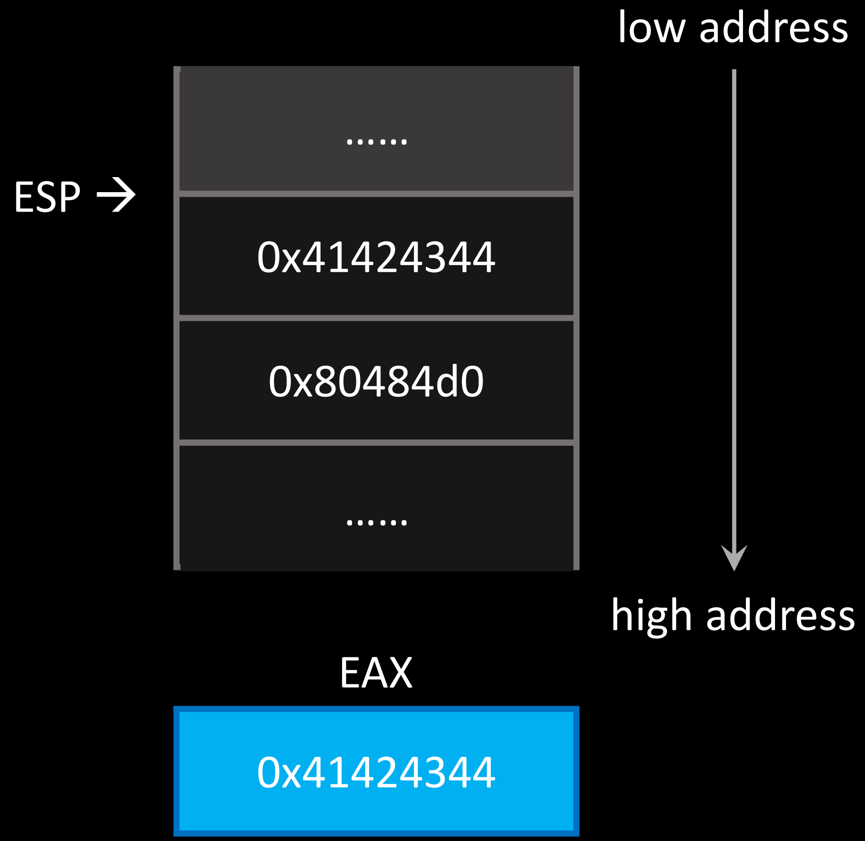
- push sth to top of stack

➢ push eax



push

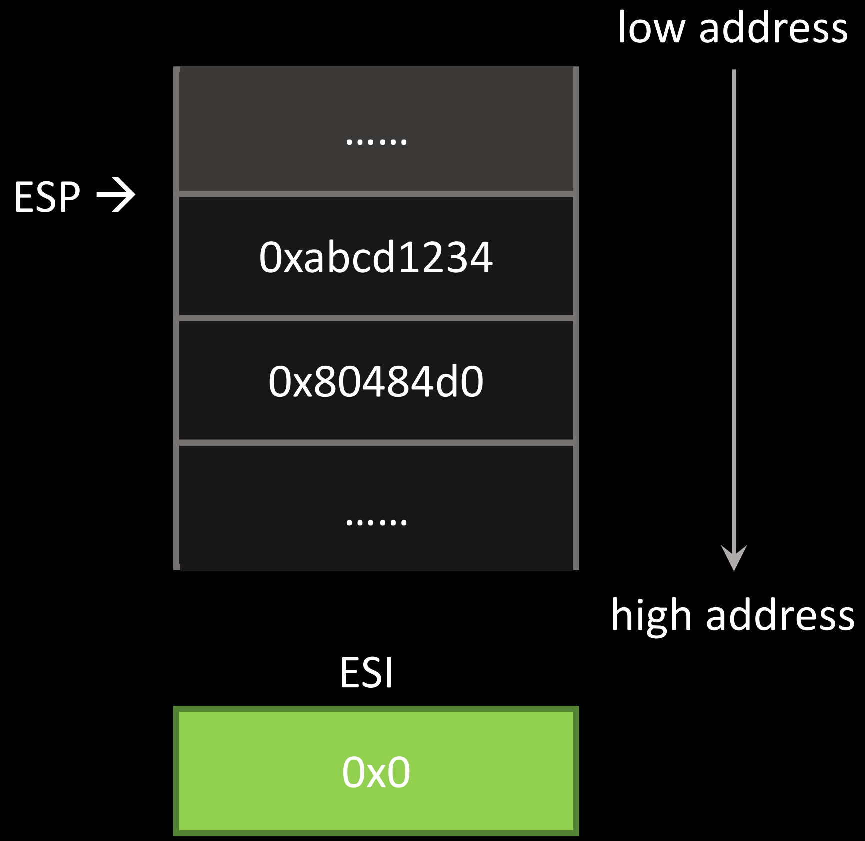
- push operand to top of stack
  - push eax



pop

- pop value off top of stack and store to operand

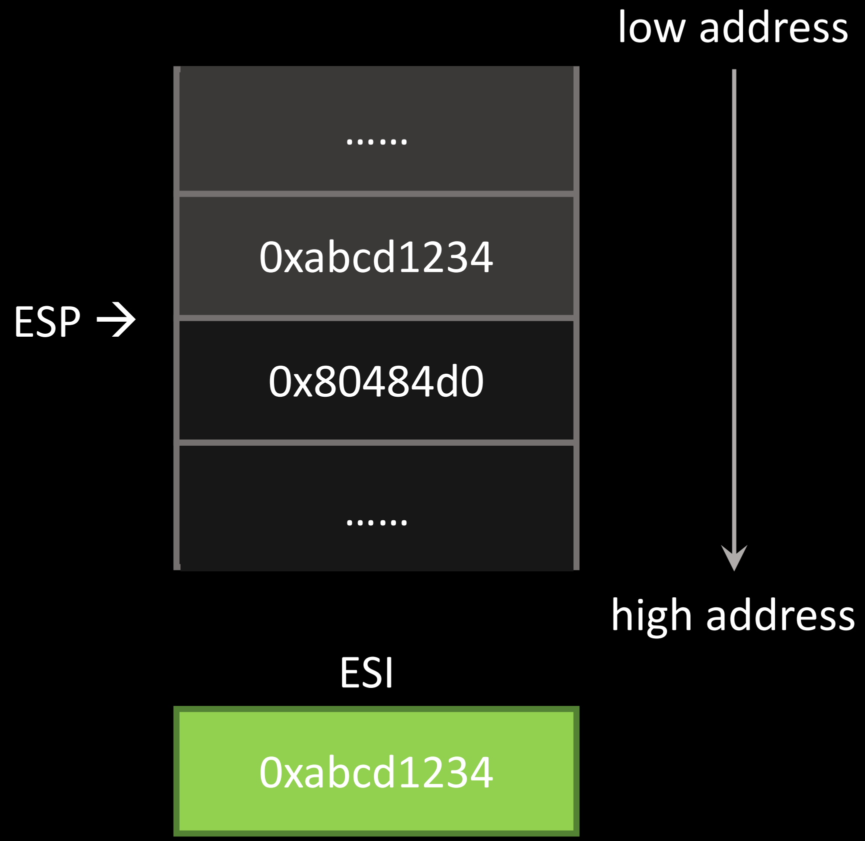
➢ `pop esi`



pop

- pop value off top of stack and store to operand

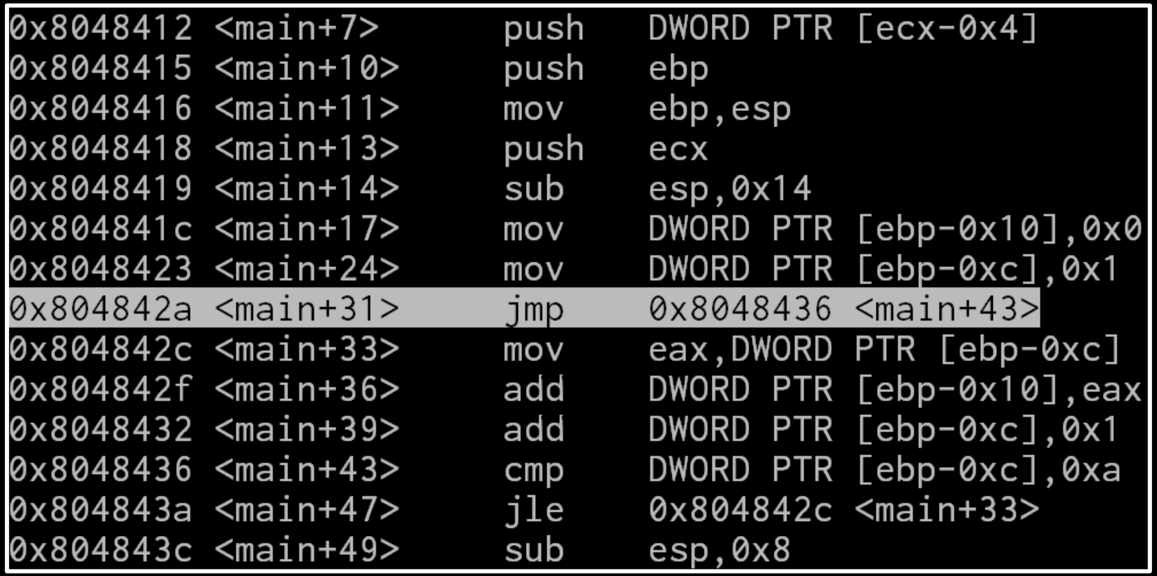
➢ `pop esi`



jmp

- jump to specified location to continue execution
- there are a variety of conditional jump instructions

➢ `jmp 0x8048436`



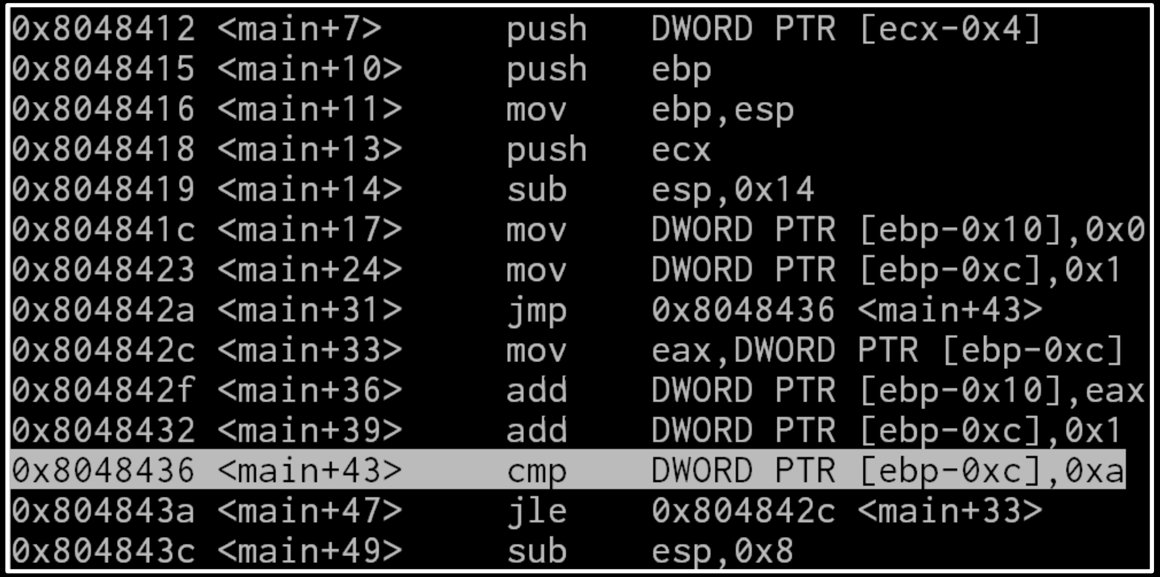
EIP →

```
0x8048412 <main+7>       push   DWORD PTR [ecx-0x4]
0x8048415 <main+10>      push   ebp
0x8048416 <main+11>      mov    ebp,esp
0x8048418 <main+13>      push   ecx
0x8048419 <main+14>      sub    esp,0x14
0x804841c <main+17>      mov    DWORD PTR [ebp-0x10],0x0
0x8048423 <main+24>      mov    DWORD PTR [ebp-0xc],0x1
0x804842a <main+31>      jmp    0x8048436 <main+43>
0x804842c <main+33>      mov    eax,DWORD PTR [ebp-0xc]
0x804842f <main+36>      add    DWORD PTR [ebp-0x10],eax
0x8048432 <main+39>      add    DWORD PTR [ebp-0xc],0x1
0x8048436 <main+43>      cmp    DWORD PTR [ebp-0xc],0xa
0x804843a <main+47>      jle    0x804842c <main+33>
0x804843c <main+49>      sub    esp,0x8
```

jmp

- jump to specified location to execute instruction
- there are a variety of conditional jump instructions

➢ `jmp 0x8048436`



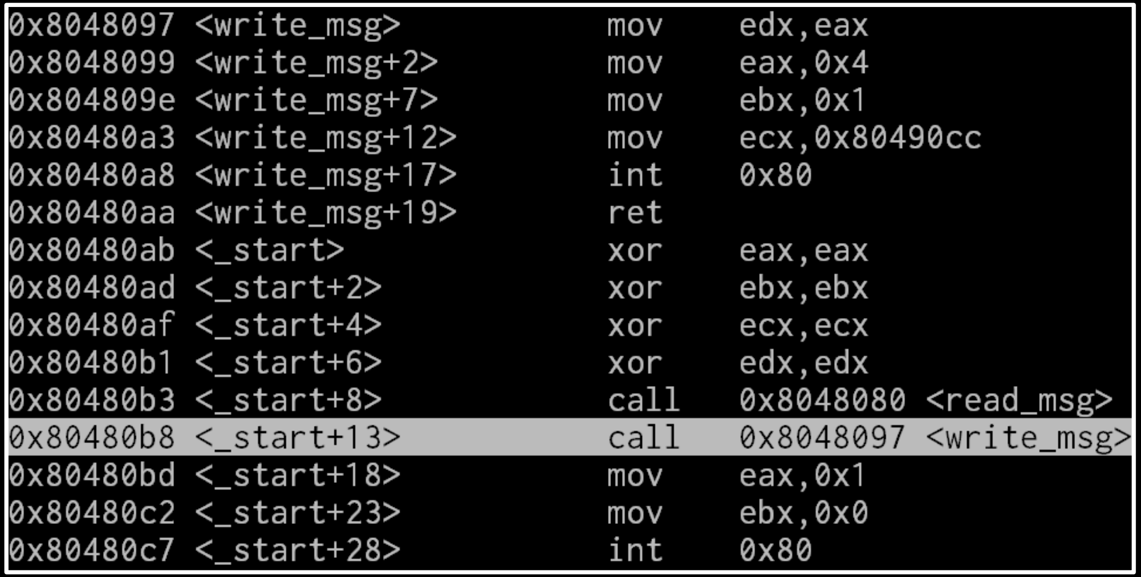
```
0x8048412 <main+7>       push   DWORD PTR [ecx-0x4]
0x8048415 <main+10>      push   ebp
0x8048416 <main+11>      mov    ebp,esp
0x8048418 <main+13>      push   ecx
0x8048419 <main+14>      sub    esp,0x14
0x804841c <main+17>      mov    DWORD PTR [ebp-0x10],0x0
0x8048423 <main+24>      mov    DWORD PTR [ebp-0xc],0x1
0x804842a <main+31>      jmp    0x8048436 <main+43>
0x804842c <main+33>      mov    eax,DWORD PTR [ebp-0xc]
0x804842f <main+36>      add    DWORD PTR [ebp-0x10],eax
0x8048432 <main+39>      add    DWORD PTR [ebp-0xc],0x1
0x8048436 <main+43>      cmp    DWORD PTR [ebp-0xc],0xa
0x804843a <main+47>      jle    0x804842c <main+33>
0x804843c <main+49>      sub    esp,0x8
```

EIP →

call

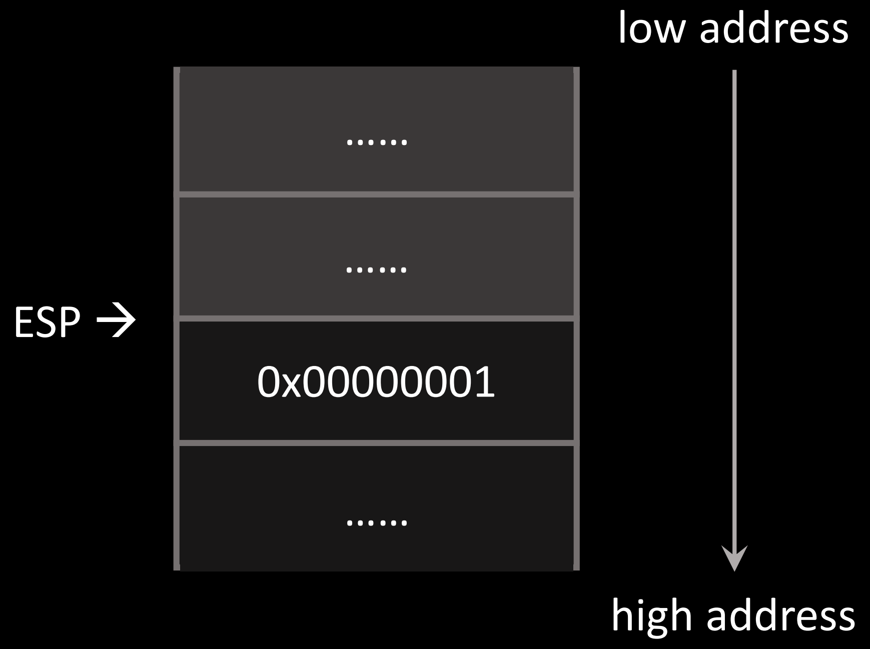
- push address of next consecutive instruction into stack as return address, and jump to execute target function

➢ `call write_msg`



EIP →

```
0x8048097 <write_msg>      mov    edx,eax
0x8048099 <write_msg+2>    mov    eax,0x4
0x804809e <write_msg+7>    mov    ebx,0x1
0x80480a3 <write_msg+12>   mov    ecx,0x80490cc
0x80480a8 <write_msg+17>   int    0x80
0x80480aa <write_msg+19>   ret
0x80480ab <_start>         xor    eax,eax
0x80480ad <_start+2>       xor    ebx,ebx
0x80480af <_start+4>       xor    ecx,ecx
0x80480b1 <_start+6>       xor    edx,edx
0x80480b3 <_start+8>       call   0x8048080 <read_msg>
0x80480b8 <_start+13>      call   0x8048097 <write_msg>
0x80480bd <_start+18>      mov    eax,0x1
0x80480c2 <_start+23>      mov    ebx,0x0
0x80480c7 <_start+28>      int    0x80
```

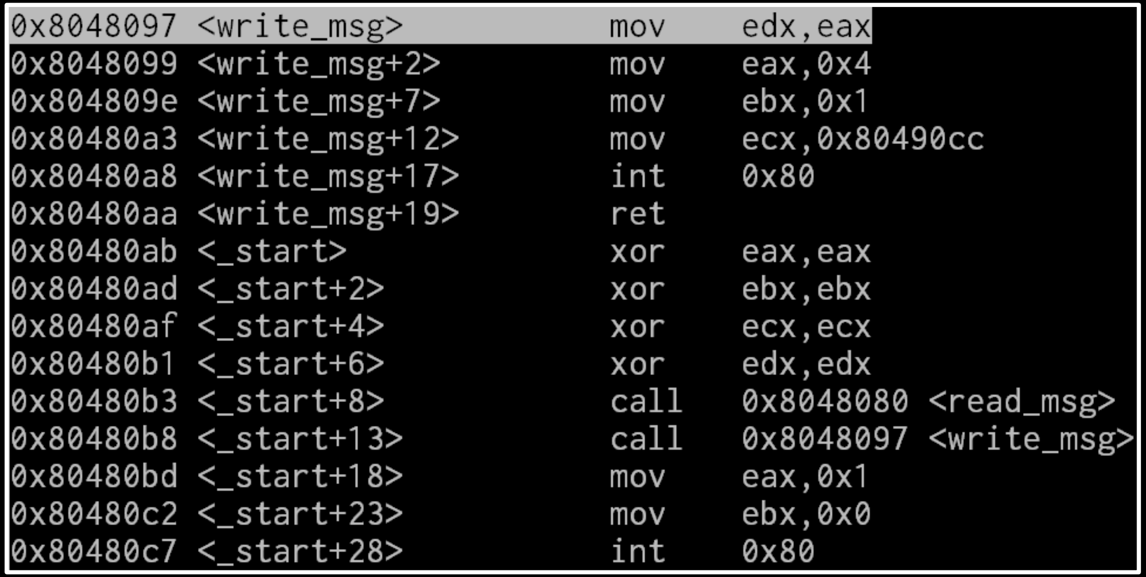


call

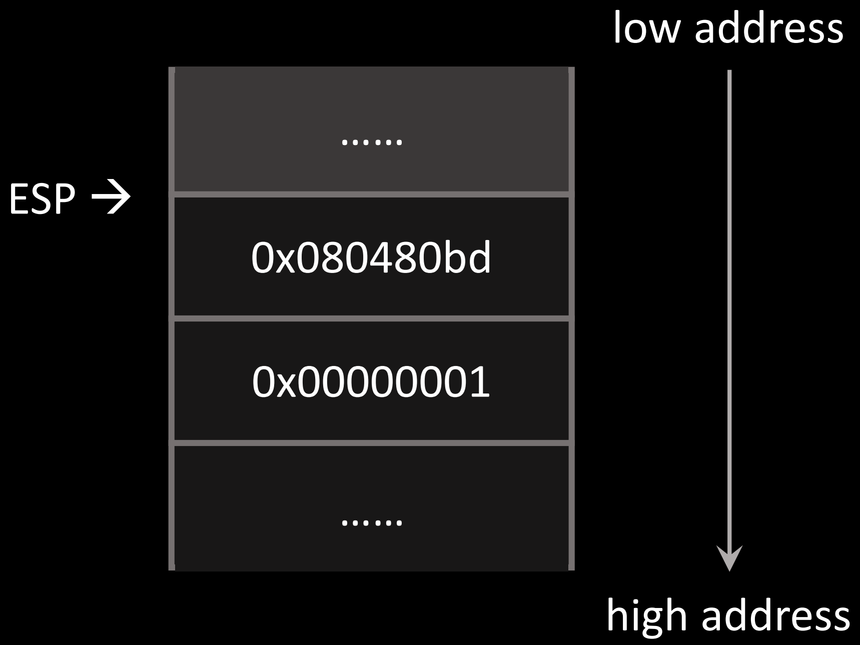
- push address of next consecutive instruction on stack as return address, and jump to execute target function

➢ `call write_msg`

EIP →



```
0x8048097 <write_msg>          mov    edx,eax
0x8048099 <write_msg+2>        mov    eax,0x4
0x804809e <write_msg+7>        mov    ebx,0x1
0x80480a3 <write_msg+12>       mov    ecx,0x80490cc
0x80480a8 <write_msg+17>       int    0x80
0x80480aa <write_msg+19>       ret
0x80480ab <_start>             xor    eax,eax
0x80480ad <_start+2>           xor    ebx,ebx
0x80480af <_start+4>           xor    ecx,ecx
0x80480b1 <_start+6>           xor    edx,edx
0x80480b3 <_start+8>           call   0x8048080 <read_msg>
0x80480b8 <_start+13>          call   0x8048097 <write_msg>
0x80480bd <_start+18>          mov    eax,0x1
0x80480c2 <_start+23>          mov    ebx,0x0
0x80480c7 <_start+28>          int    0x80
```

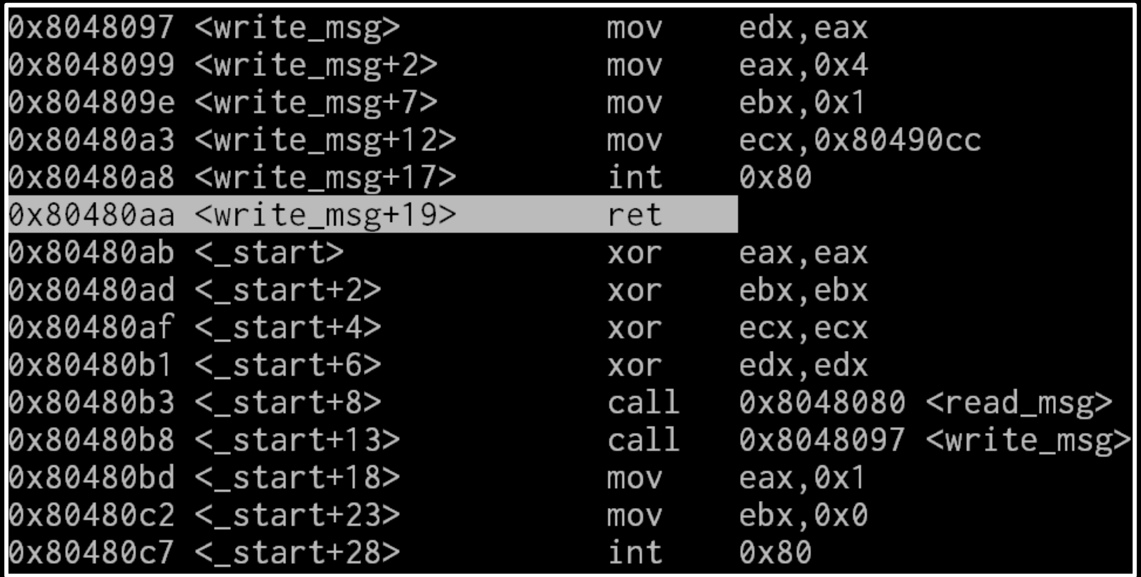


ret

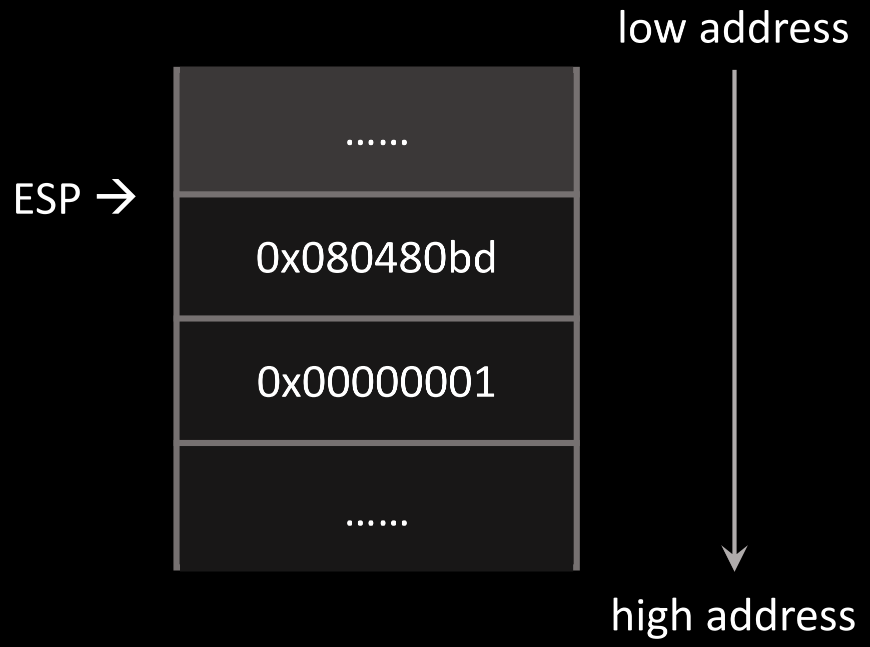
- retrieve return address from stack and jump back to it

➢ `ret`

EIP →



```
0x8048097 <write_msg>          mov    edx,eax
0x8048099 <write_msg+2>        mov    eax,0x4
0x804809e <write_msg+7>        mov    ebx,0x1
0x80480a3 <write_msg+12>       mov    ecx,0x80490cc
0x80480a8 <write_msg+17>       int    0x80
0x80480aa <write_msg+19>       ret
0x80480ab <_start>             xor    eax,eax
0x80480ad <_start+2>           xor    ebx,ebx
0x80480af <_start+4>           xor    ecx,ecx
0x80480b1 <_start+6>           xor    edx,edx
0x80480b3 <_start+8>           call   0x8048080 <read_msg>
0x80480b8 <_start+13>          call   0x8048097 <write_msg>
0x80480bd <_start+18>          mov    eax,0x1
0x80480c2 <_start+23>          mov    ebx,0x0
0x80480c7 <_start+28>          int    0x80
```



low address

ESP →

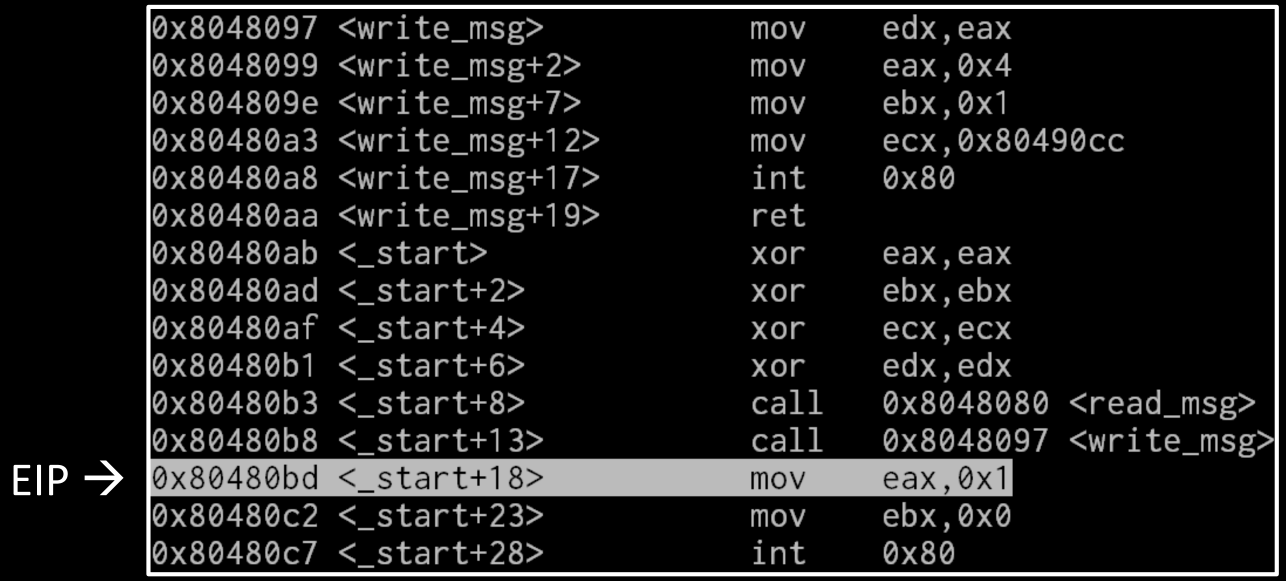
......
0x080480bd
0x00000001
......

high address

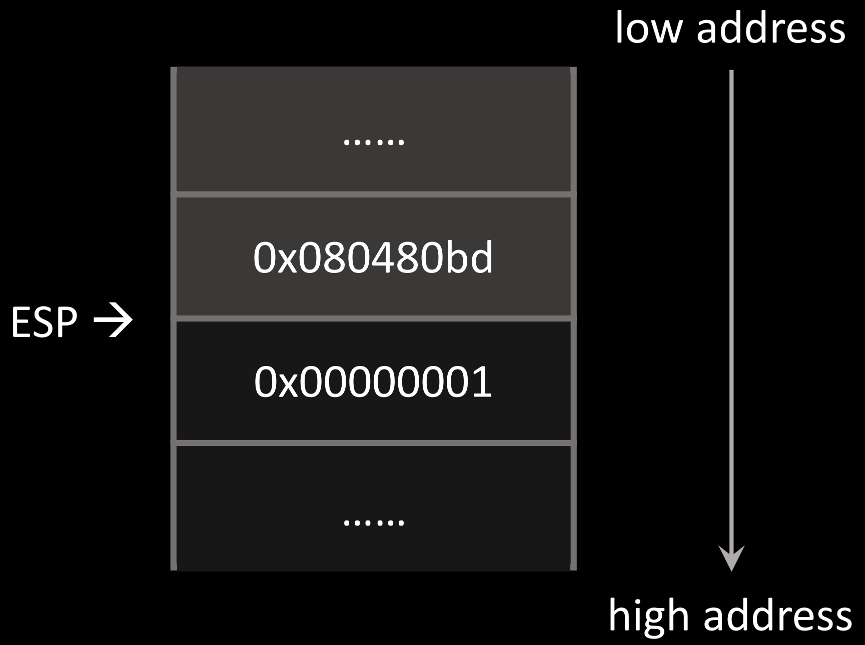
ret

- retrieve return address from stack and jump back to it

➢ `ret`



```
0x8048097 <write_msg>           mov    edx,eax
0x8048099 <write_msg+2>         mov    eax,0x4
0x804809e <write_msg+7>         mov    ebx,0x1
0x80480a3 <write_msg+12>        mov    ecx,0x80490cc
0x80480a8 <write_msg+17>        int    0x80
0x80480aa <write_msg+19>        ret
0x80480ab <_start>              xor    eax,eax
0x80480ad <_start+2>            xor    ebx,ebx
0x80480af <_start+4>            xor    ecx,ecx
0x80480b1 <_start+6>            xor    edx,edx
0x80480b3 <_start+8>            call   0x8048080 <read_msg>
0x80480b8 <_start+13>           call   0x8048097 <write_msg>
0x80480bd <_start+18>           mov    eax,0x1
0x80480c2 <_start+23>           mov    ebx,0x0
0x80480c7 <_start+28>           int    0x80
```



System Call

- Program requests services from kernel of OS
- System call number stores in EAX
- Arguments order: EBX ECX EDX ESI EDI
- Return value also stores in EAX
- int 0x80 is used to trigger system call under Linux (x86)
- Reference: x86 x86_64

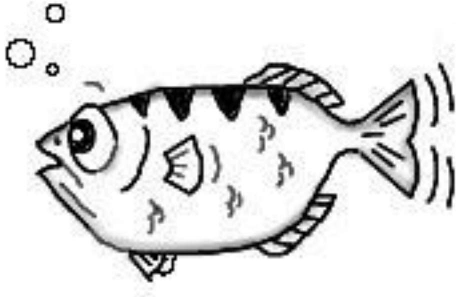
Common used system call

Usually set to 0 (stdin)

Usually set to 1 (stdout)

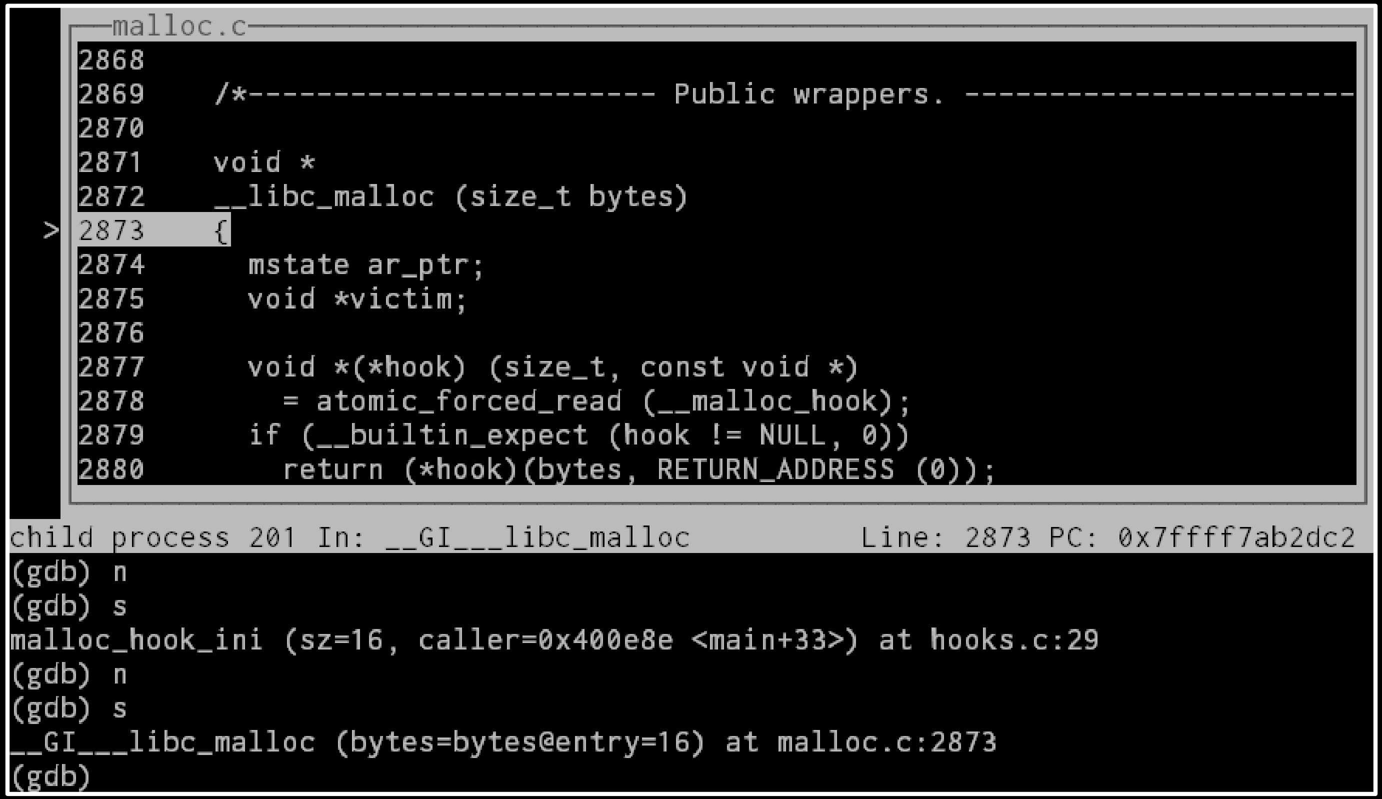
Name	EAX	EBX	ECX	EDX
read	3	fd	buf	count
write	4	fd	buf	count
open	5	filename	flags	mode
execve	11	filename	argv	envp

Usually set to 0



GNU Debugger (GDB)

Text User Interface (TUI)



```
  ┌─malloc.c──────────────────────────────────────────────────────────────────┐
  │2868                                                                       │
  │2869    /*----------------------- Public wrappers. ----------------------  │
  │2870                                                                       │
  │2871    void *                                                             │
  │2872    __libc_malloc (size_t bytes)                                       │
 >│2873    {                                                                  │
  │2874      mstate ar_ptr;                                                   │
  │2875      void *victim;                                                    │
  │2876                                                                       │
  │2877      void *(*hook) (size_t, const void *)                             │
  │2878        = atomic_forced_read (__malloc_hook);                          │
  │2879      if (__builtin_expect (hook != NULL, 0))                          │
  │2880        return (*hook)(bytes, RETURN_ADDRESS (0));                     │
  └───────────────────────────────────────────────────────────────────────────┘
child process 201 In: __GI___libc_malloc         Line: 2873 PC: 0x7ffff7ab2dc2
(gdb) n
(gdb) s
malloc_hook_ini (sz=16, caller=0x400e8e <main+33>) at hooks.c:29
(gdb) n
(gdb) s
__GI___libc_malloc (bytes=bytes@entry=16) at malloc.c:2873
(gdb)
```

- lay src / asm / reg
- Ctrl-x + a

Basic operation

- break -- set break point

```
➢ b main        # set a breakpoint at main()
➢ b 4           # break at line 4
➢ b *0xffffd1d0 # break at instruction at 0xffffd1d0
```

- run [argument(s)] -- start the program
- continue -- continue program execution until next breakpoint or termination

Basic operation

- attach -- attach to a process

  ➢ `at 1337          # attach to process whose pid is 1337`

  ✓ `echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope`

Basic operation

- step -- run to next source line, can trace into function
- next -- run to next source line without tracing into function
- stepi -- instruction version of step
- nexti -- instruction version of next

Basic operation

- record -- record the execution process, which can be observe in replay mode later
- reverse-next
- reverse-step
- reverse-nexti
- reverse-stepi

Basic operation

- print -- print the value

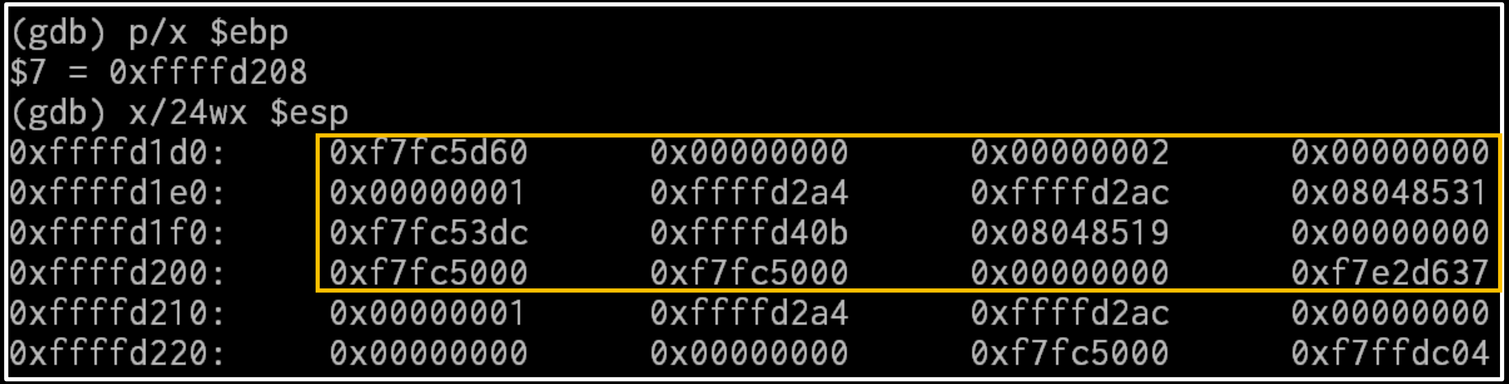
```
p/d var1    # print var1 in sign decimal
p/x $esp    # print ESP in hexadecimal format
```

- x -- examine memory content

```
x/32wx $esp        # examine content from ESP to ESP+128
x/s 0x80484d0      # print content in 0x80484d0 as string
x/4i 0x080482be    # print 4 instructions from 0x080482be
```

Basic operation

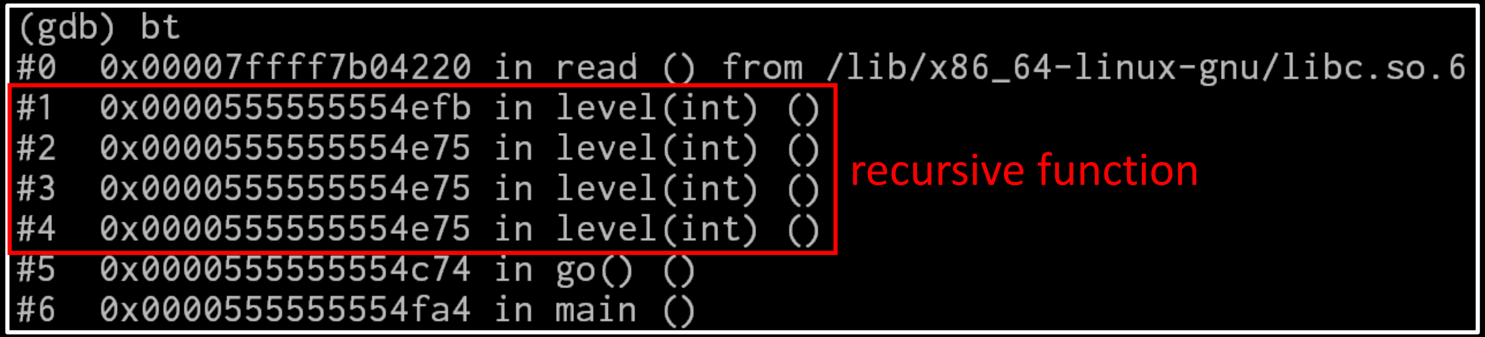
- Examine current stack frame



```
(gdb) p/x $ebp
$7 = 0xffffd208
(gdb) x/24wx $esp
0xffffd1d0:     0xf7fc5d60      0x00000000      0x00000002      0x00000000
0xffffd1e0:     0x00000001      0xffffd2a4      0xffffd2ac      0x08048531
0xffffd1f0:     0xf7fc53dc      0xffffd40b      0x08048519      0x00000000
0xffffd200:     0xf7fc5000      0xf7fc5000      0x00000000      0xf7e2d637
0xffffd210:     0x00000001      0xffffd2a4      0xffffd2ac      0x00000000
0xffffd220:     0x00000000      0x00000000      0xf7fc5000      0xf7ffdc04
```

Basic operation

- info -- list some useful information
  - ➢ `i b                # list current breakpoints`
  - ➢ `i r                # list value of each register`
  - ➢ `i proc map         # print memory mapping`
- backtrace – show stack frames information



```
(gdb) bt
#0  0x00007ffff7b04220 in read () from /lib/x86_64-linux-gnu/libc.so.6
#1  0x0000555555554efb in level(int) ()
#2  0x0000555555554e75 in level(int) ()
#3  0x0000555555554e75 in level(int) ()
#4  0x0000555555554e75 in level(int) ()
#5  0x0000555555554c74 in go() ()
#6  0x0000555555554fa4 in main ()
```

recursive function

Reference

- Binary exploitation - AIS3
- Intel and AT&T Syntax
- x86 Instruction Set Reference
- NASM document
- GDB online document
- 用 Python 拓展 GDB