Supports from IA-32

- AT&T Assembly language
- Function calls
- Recap – IA-32 memory management
- I/O & Peripherals
- IA-32 Interrupt Processing

Lab0 is completely out!

- 打字游戏 (deadline: 2014/03/09 23:59:59)
- 自己的游戏 (deadline: 2014/03/23 23:59:59)
- 完整的工程和实验报告 (deadline: 2014/03/25 23:59:59)

- Have fun!

Accessing Google

- For some reasons, Google maybe not easy to access.
- Modify the "hosts" file
  - /etc/hosts
- Search "google hosts" for more details.

- AT&T assembly language

Assembly language

- a "middle-level" programming language
- the symbolization of machine language
- can operate registers and hardware directly
  - To know the state of hardware, we simply read from its control register(s)
  - To let the hardware do something, we simply write the specific "command word" to its control register(s)

Assembly language (cont.)

inb $0x60, %al

- read one byte from the port 0x60, and store it to the register "al"
  - obtain the scancode of the key pressed down

outb %al, $0x20

- write the byte storing in the register "al" to the port 0x20
  - issue a command word to the I8259A processor

Addressing

- syntax:

  mem_location[(base, [index, [scale]])]

- example:

  gcc -S -o test.S test.c

```
 1 #include "stdio.h"
 2
 3 int main(){
 4     int n = 1;
 5     int a[100];
 6     a[n + 3] = 97;
 7     printf("a[n + 3] = %d\n", a[n + 3]);
 8     return 0;
 9 }
```

```
 8 main:
 9     pushl   %ebp
10     movl    %esp, %ebp
11     andl    $-16, %esp                  #esp &= 0xfffffff0
12     subl    $432, %esp                  #esp -= 432
13     movl    $1, 428(%esp)               #n = 1
14     movl    428(%esp), %eax             #eax = n
15     addl    $3, %eax                    #eax += 3
16     movl    $97, 28(%esp,%eax,4)        #a[eax] = 97
```

Example - "Hello, world!"

-http://en.wikipedia.org/wiki/List_of_Hello_world_program_examples

Assembly language – x86-64 Linux, AT&T syntax

Main article: Assembly language

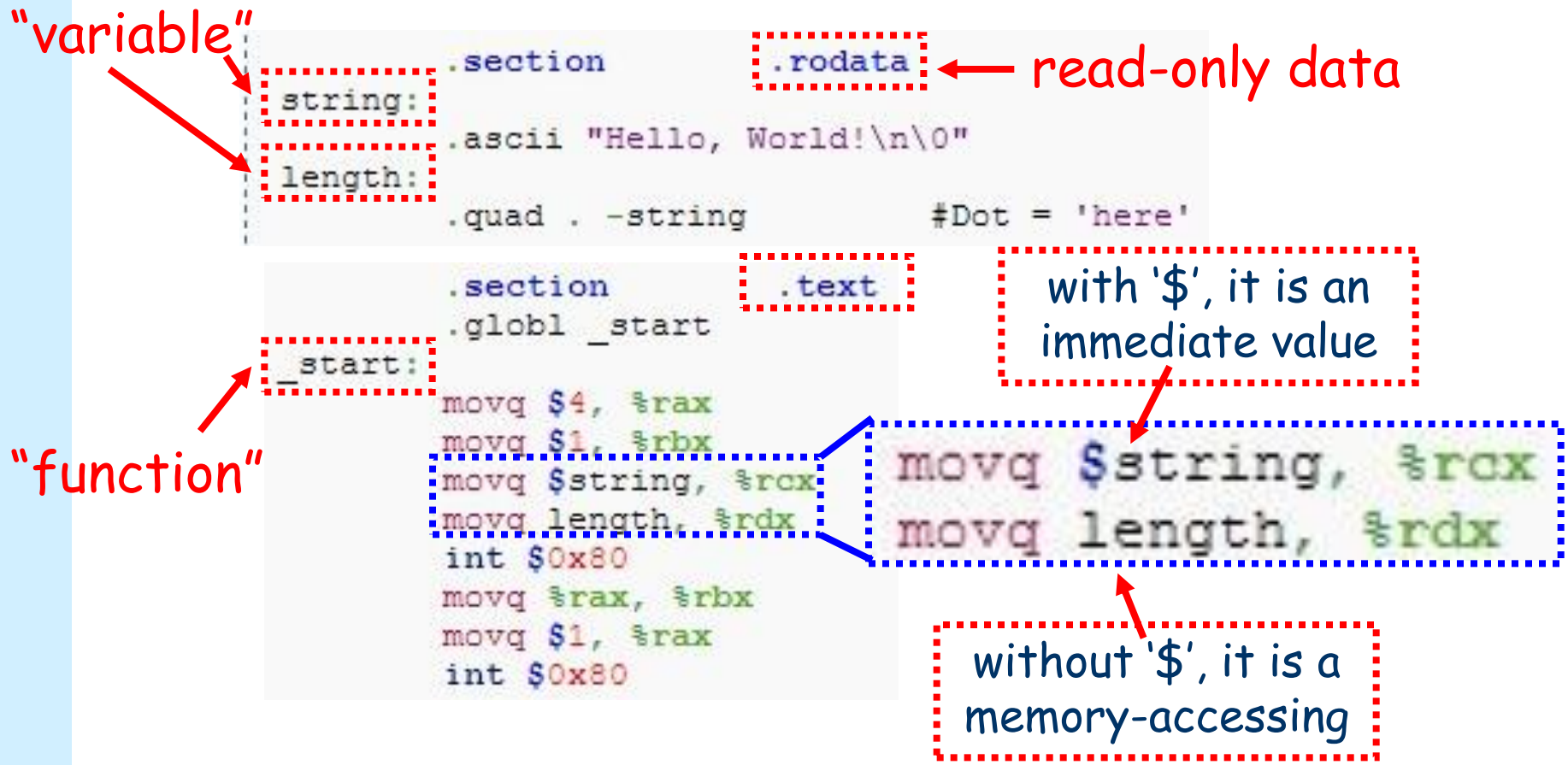
```
        .section        .rodata
string:
        .ascii "Hello, World!\n\0"
length:
        .quad . -string          #Dot = 'here'

        .section        .text
        .globl _start            #Make entry point visible to linker
_start:
        movq $4, %rax            #4=write
        movq $1, %rbx            #1=stdout
        movq $string, %rcx
        movq length, %rdx
        int $0x80                #Call Operating System
        movq %rax, %rbx          #Make program return syscall exit status
        movq $1, %rax            #1=exit
        int $0x80                #Call System Again
```

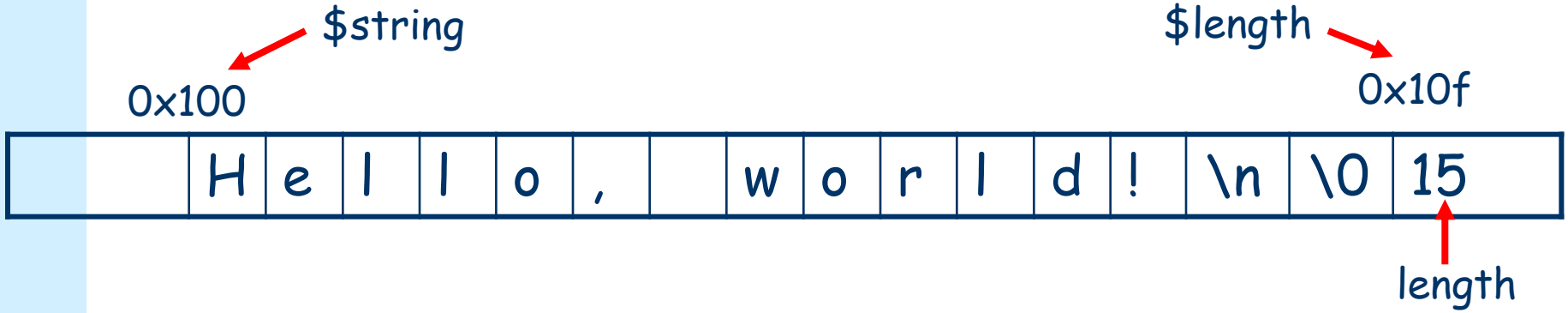
in IA-32 assembly language:

movl $4, %eax

Example - "Hello, world!" (cont.)



Example - "Hello, world!" (cont.)




```
movl $1, %eax

movl 1, %eax
```

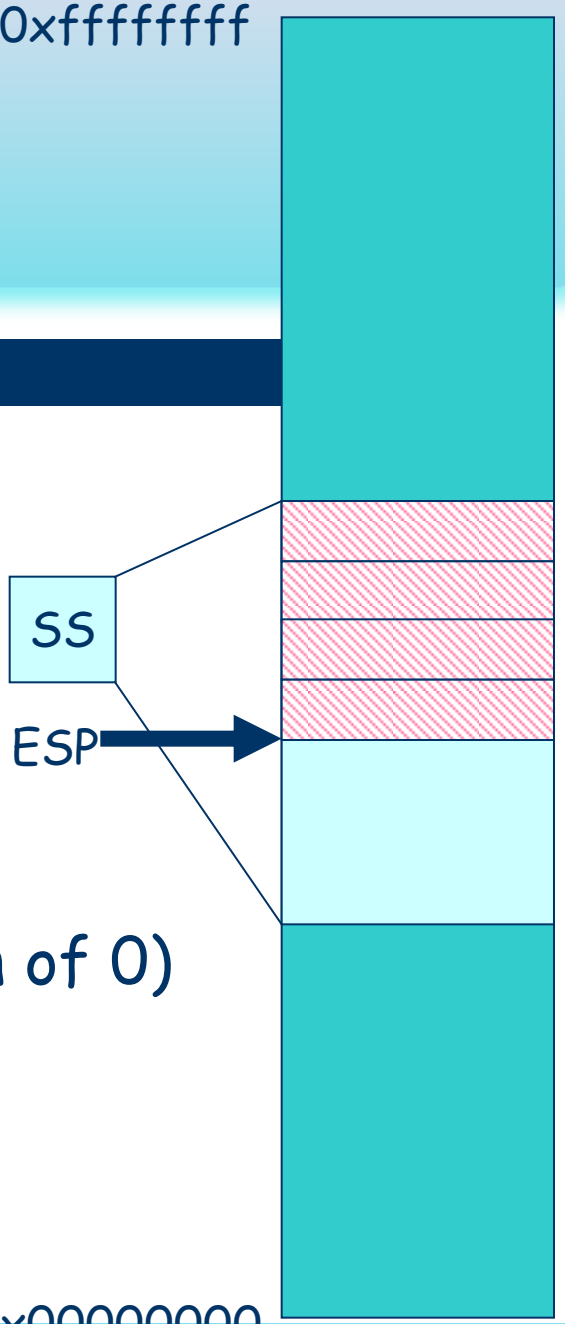
???

- Hardware stack & Function calls

- http://sock-raw.org/netsec/stack

Stack

- in data structure
  - stack = array + top pointer
- in hardware
  - stack = SS + ESP
    - SS specifies a segment of memory
    - ESP indicates the top of stack
  - grow downward (to the direction of 0)
  - 4 bytes per elements



Stack (cont.)

- operations – push, pop

```
pushl %eax
```

  - It is identical to

```
subl $4, %esp
movl %eax, (%esp)
```

  - ==========

```
popl %ebx
```

  - It is identical to

```
movl (%esp), %ebx
addl $4, %esp
```

Function calls

- Calling a function can be easy

  jmp printf

- But how to assure it can return correctly?
  - save the return address before calling
- Where to save?
  - Stack !!!
- Memories for local variables are usually allocated in stack.
  - Why?

An example

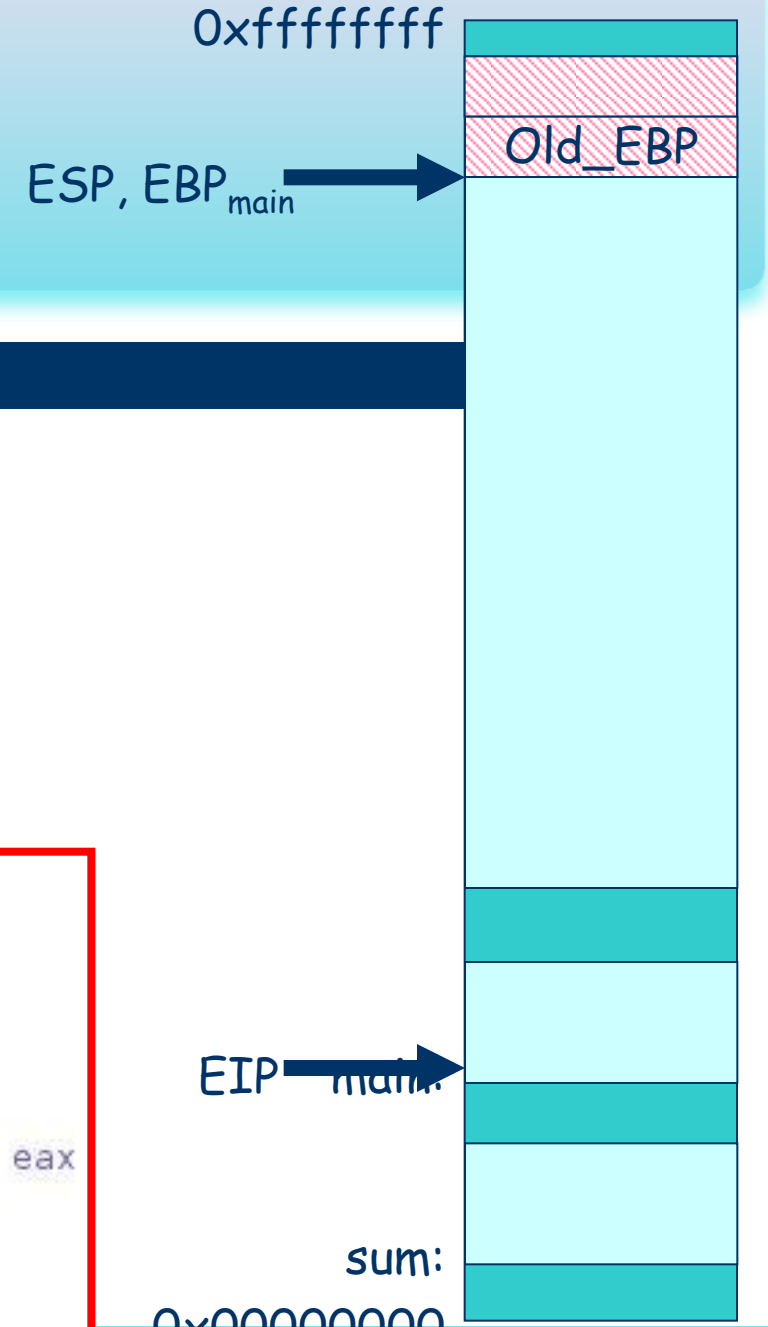
```
 1 #include "stdio.h"
 2
 3 int sum(int a, int b){
 4     int c = a + b;
 5     return c;
 6 }
 7
 8 int x = 1, y = 2, z;
 9
10 int main(){
11     z = sum(x, y);
12     printf("%d\n", z);
13     return 0;
14 }
```

```
17 .globl x
18     .data
19     .align 4
20     .type   x, @object
21     .size   x, 4
22 x:
23     .long   1
24 .globl y
25     .align 4
26     .type   y, @object
27     .size   y, 4
28 y:
29     .long   2
30     .comm   z,4,4
31     .section    .rodata
32 .LC0:
33     .string "%d\n"
```

```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

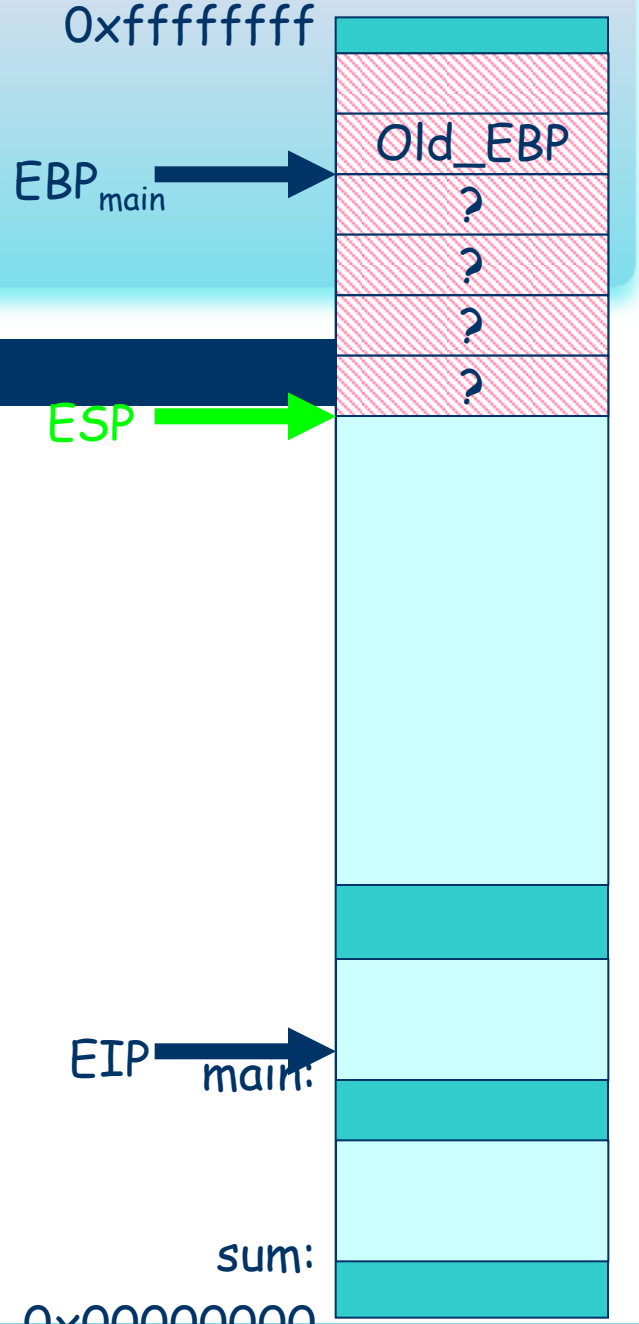
eax	edx
?	?



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

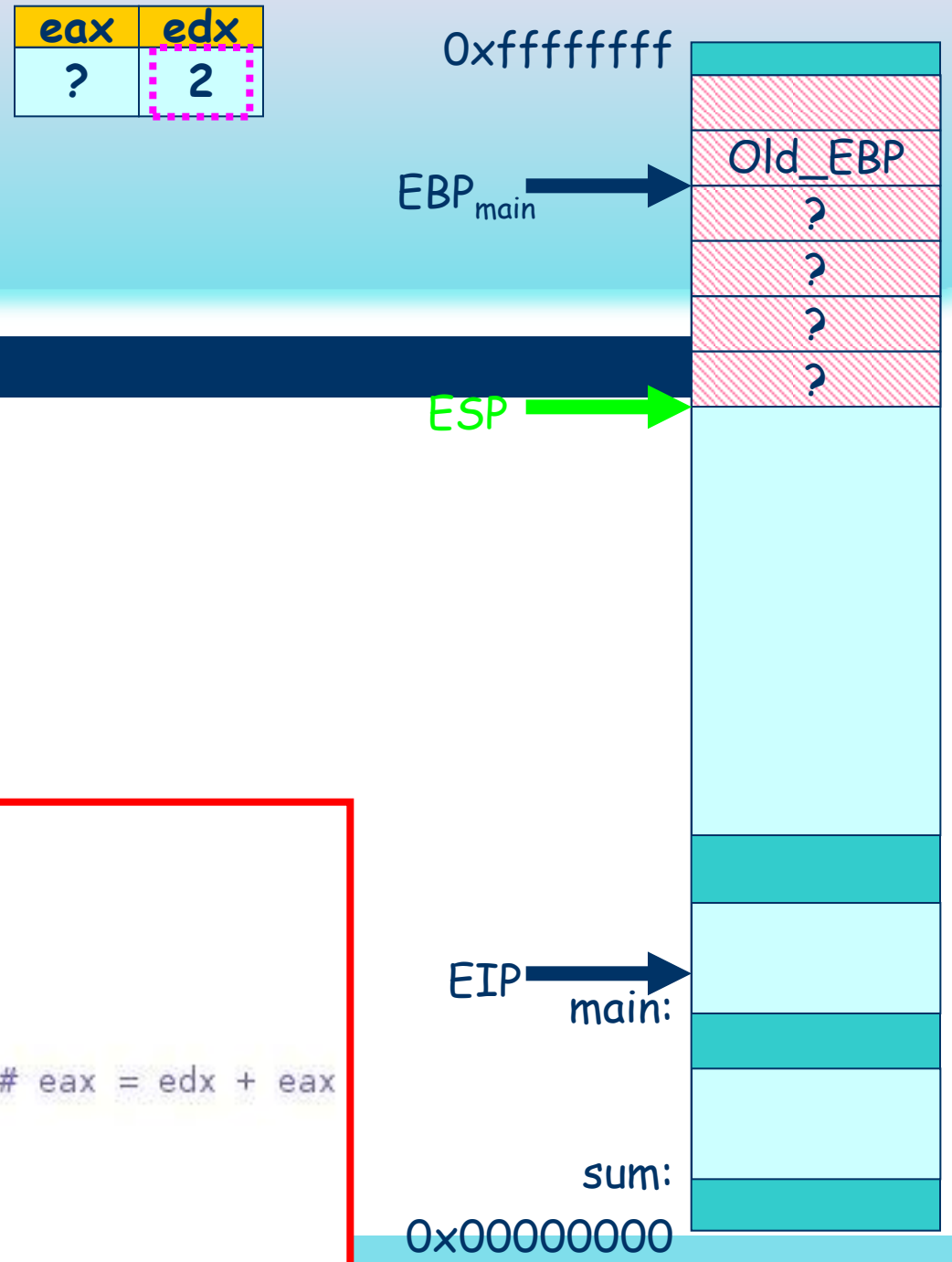
```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

eax	edx
?	?



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

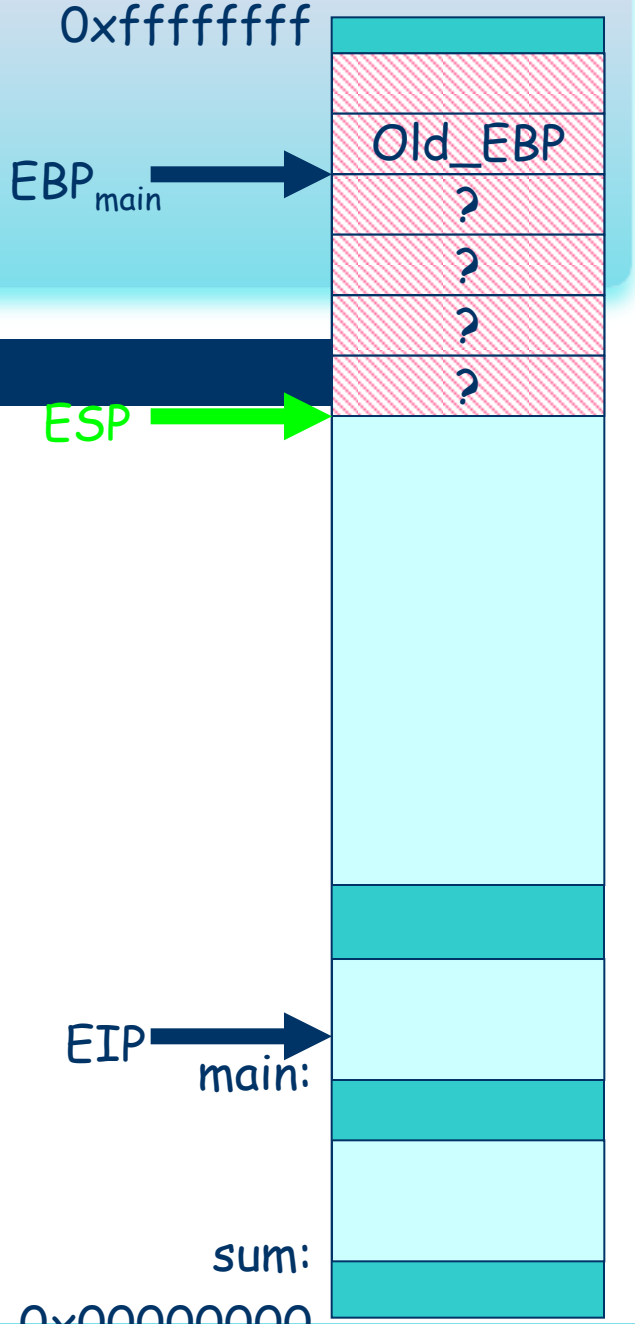
```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

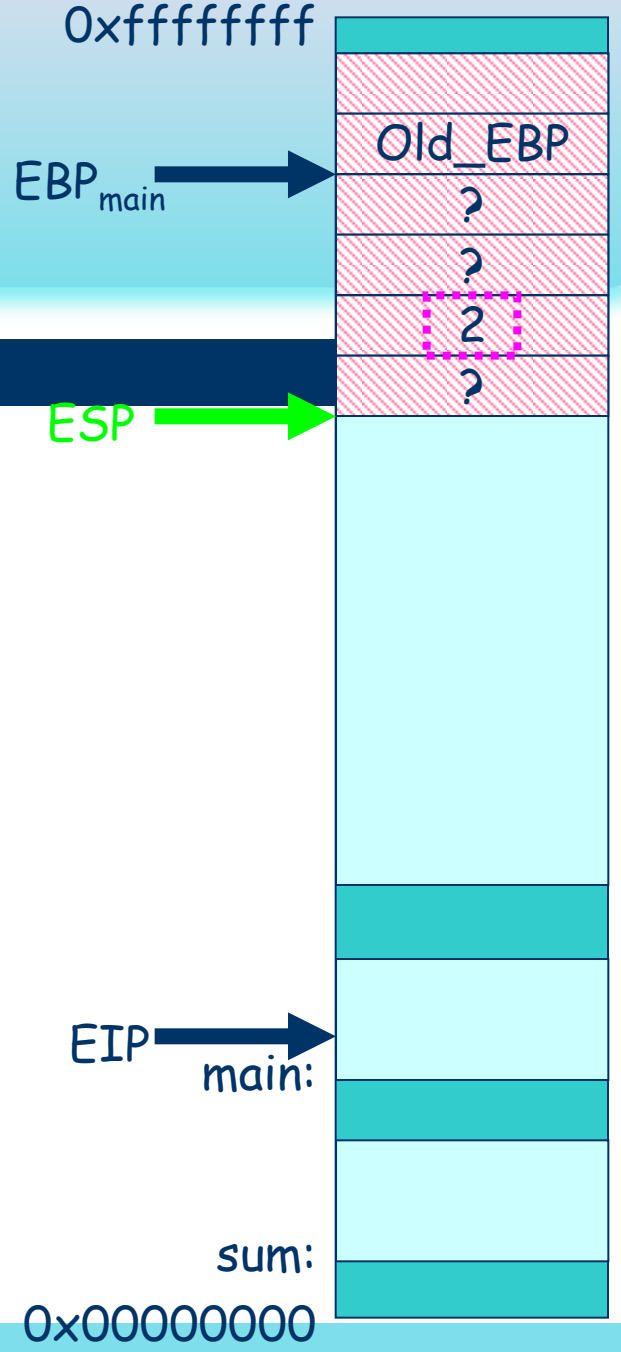
eax	edx
1	2



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

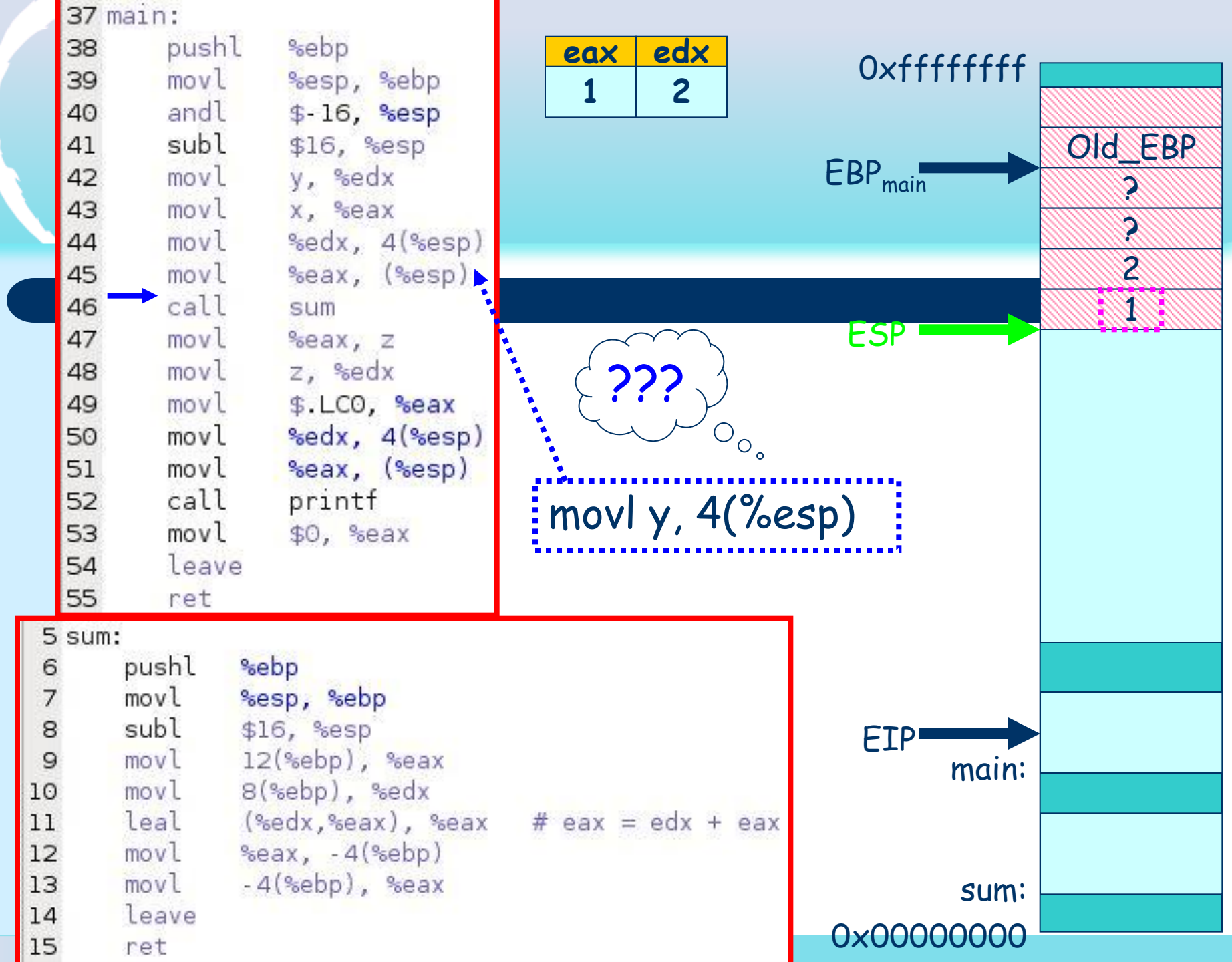
eax	edx
1	2



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

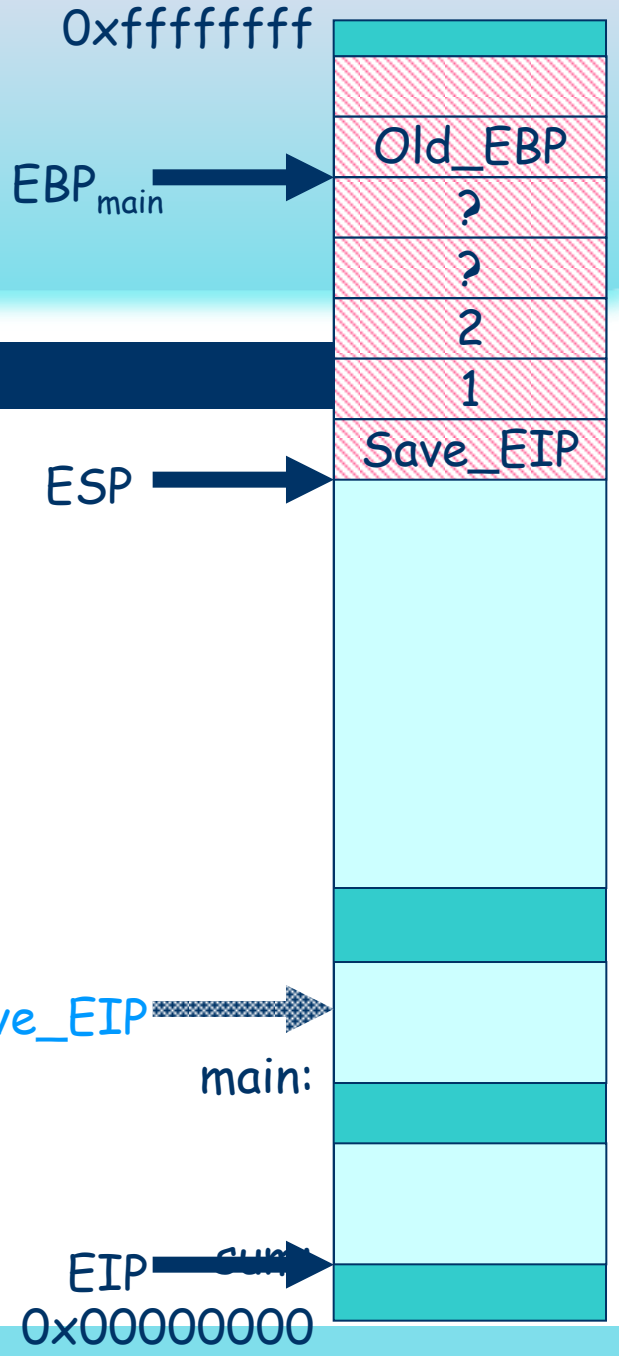
eax	edx
1	2



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

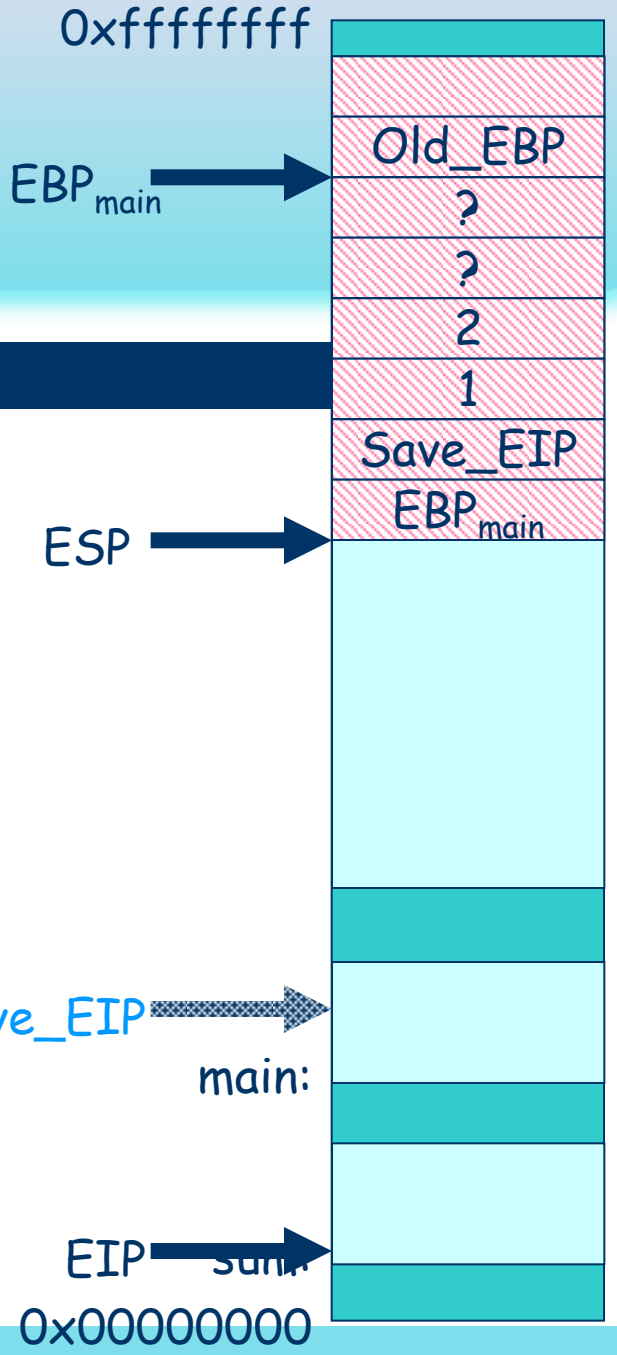
eax	edx
1	2



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

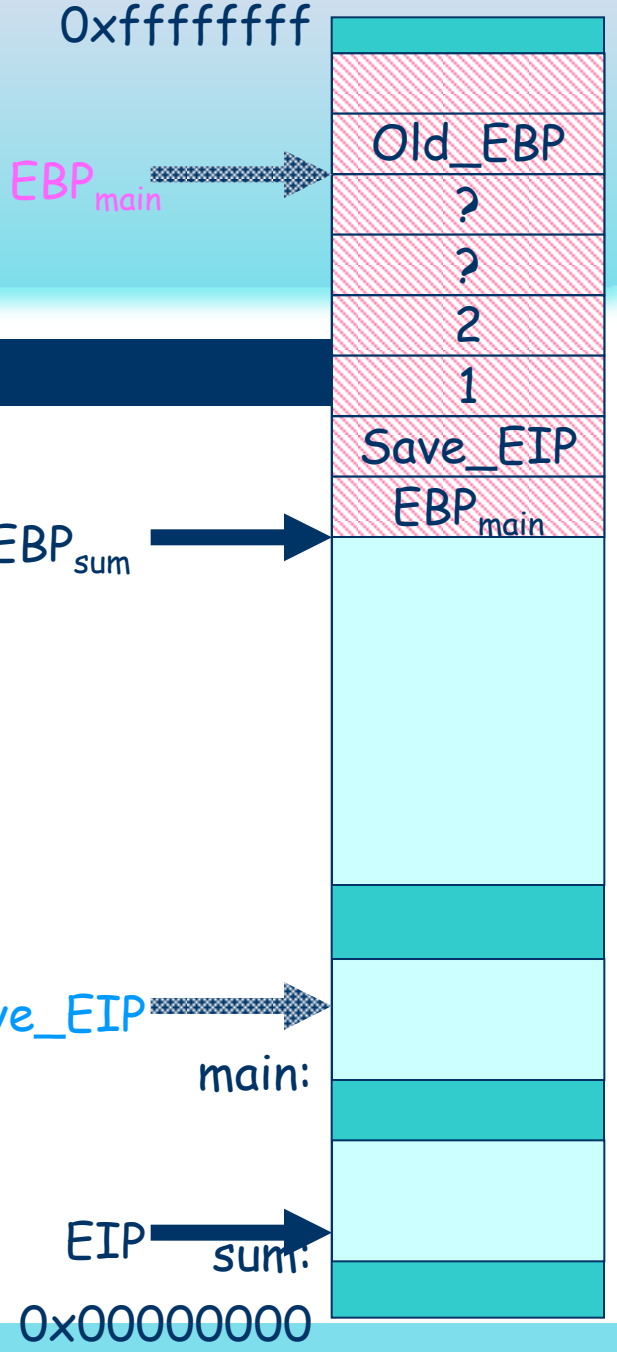
eax	edx
1	2



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

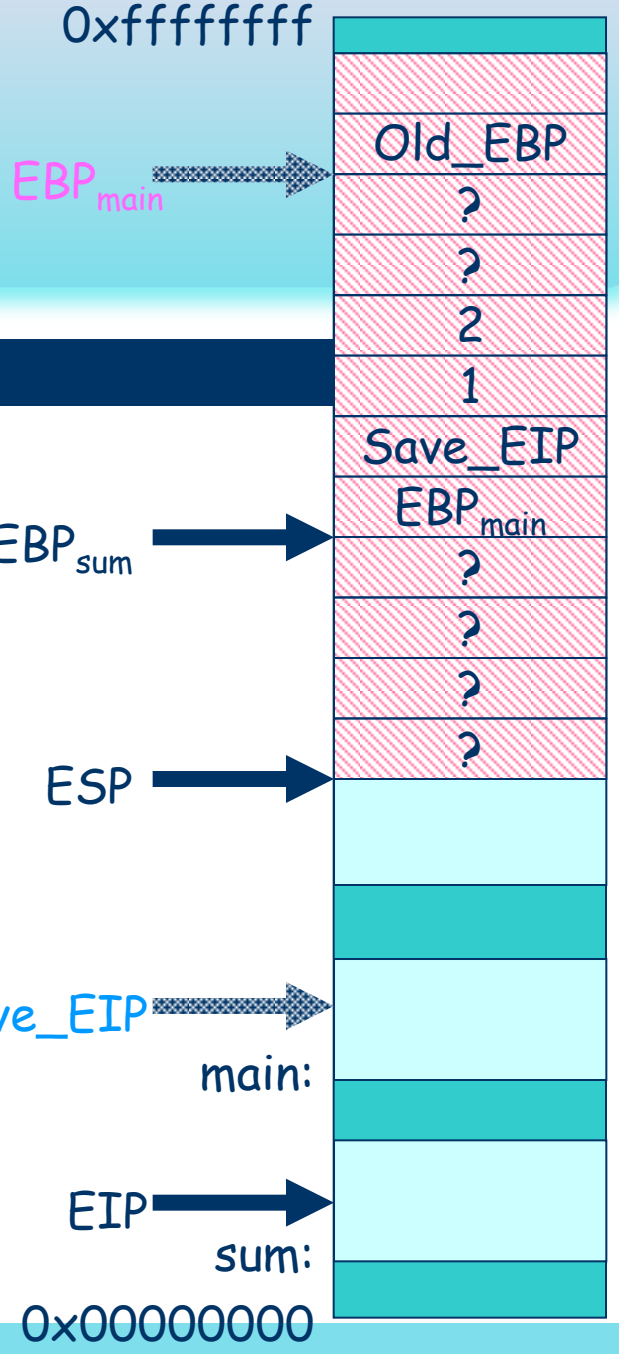
eax	edx
1	2



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

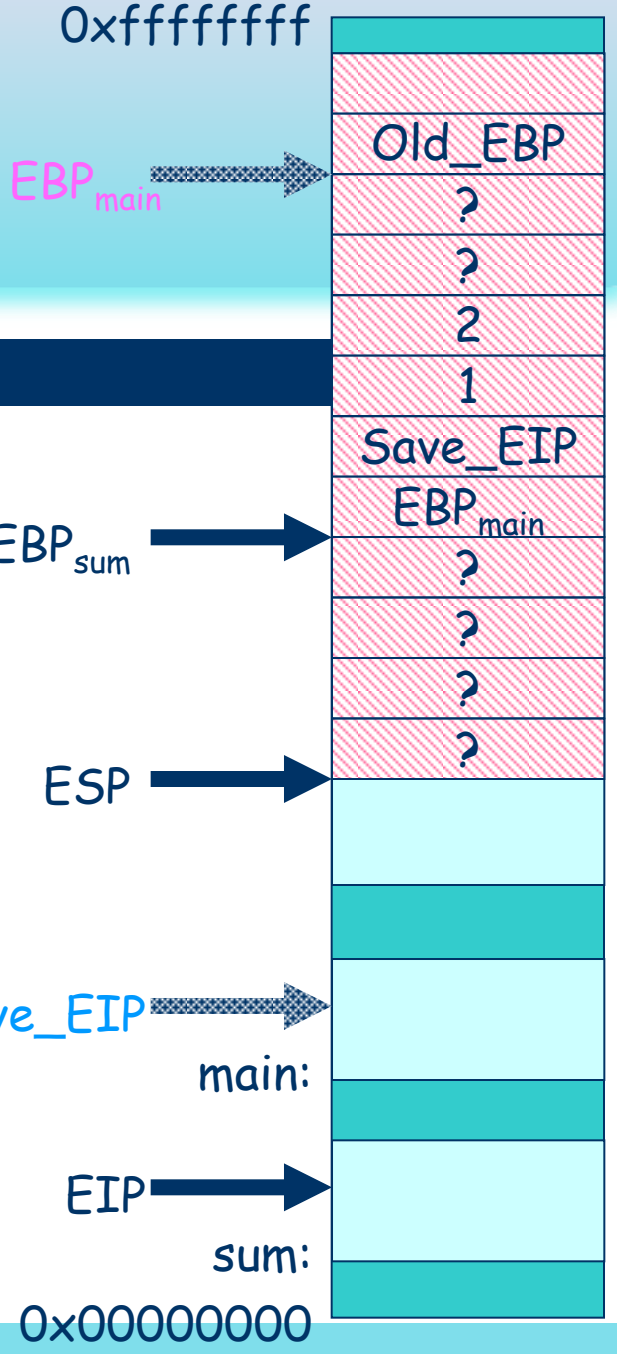
eax	edx
1	2



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax    # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

eax	edx
2	2



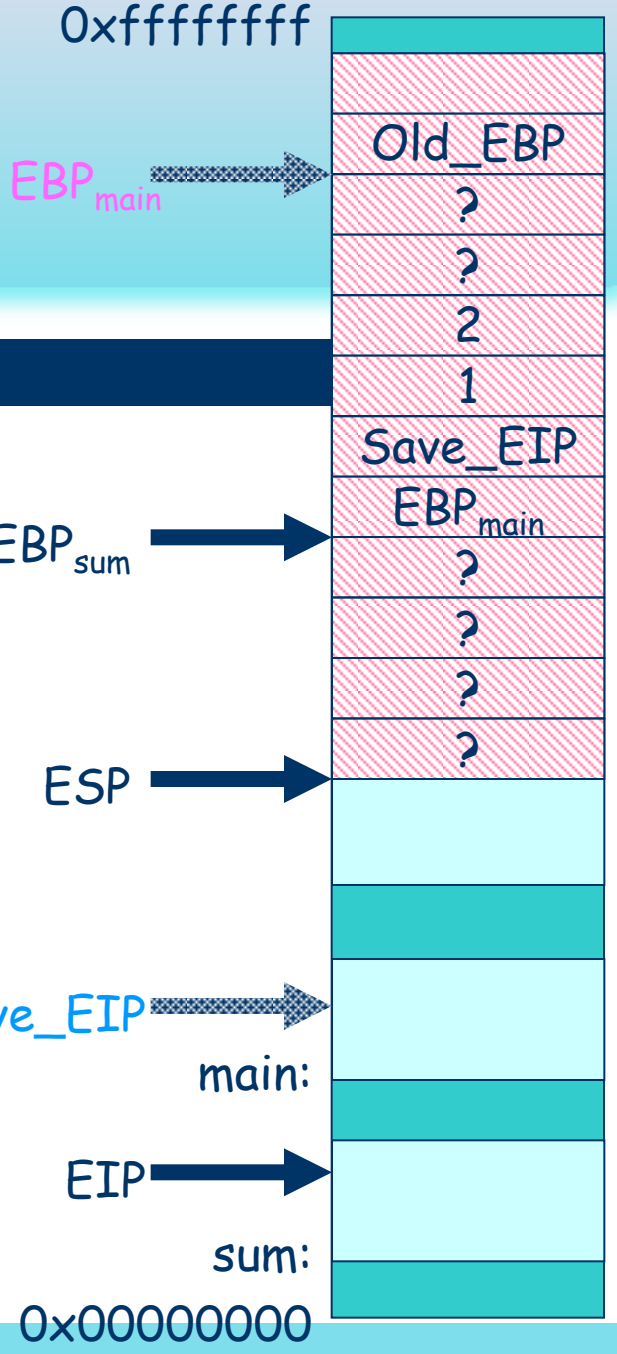
Is EBP necessary?

- Why use EBP?
  - accessing parameters
  - all EBP values saved in the stack form a trace of function calls
- Without EBP,
  - do function calls still work?
  - can you somehow obtain the trace of function calls?

```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax    # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

eax	edx
2	1



Why "lea", but not "add"?

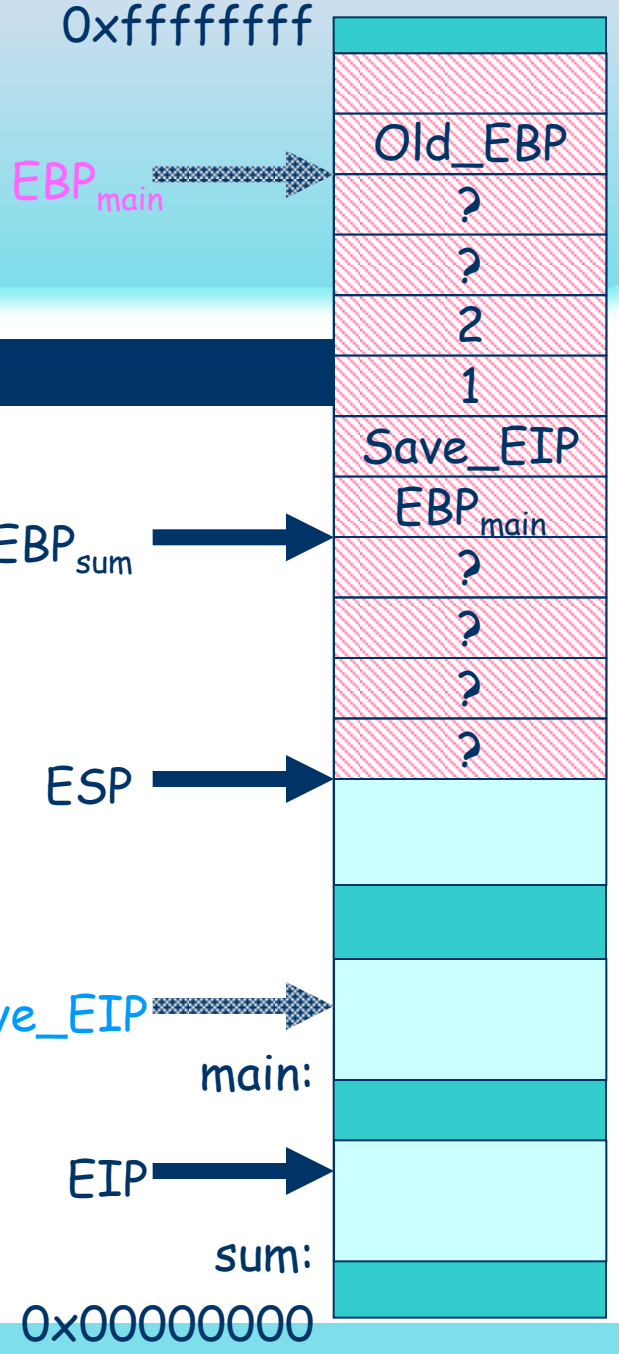
- leal (%edx, %eax), %eax
  - lea = load effective address
  - R[eax] = R[edx] + R[eax]
- Which is faster? Why?

- Design of machine-dependent optimization in a compiler requires deep insight into computer architecture.
  - This is systematic view.

```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

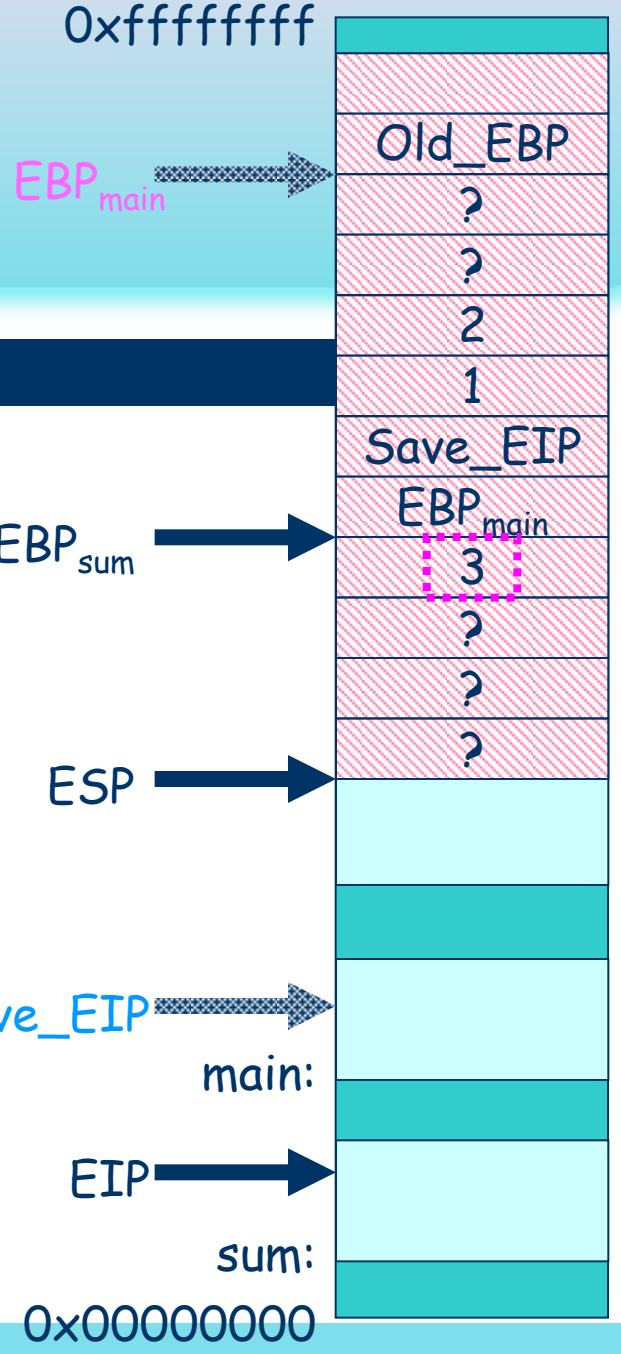
eax	edx
3	1



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

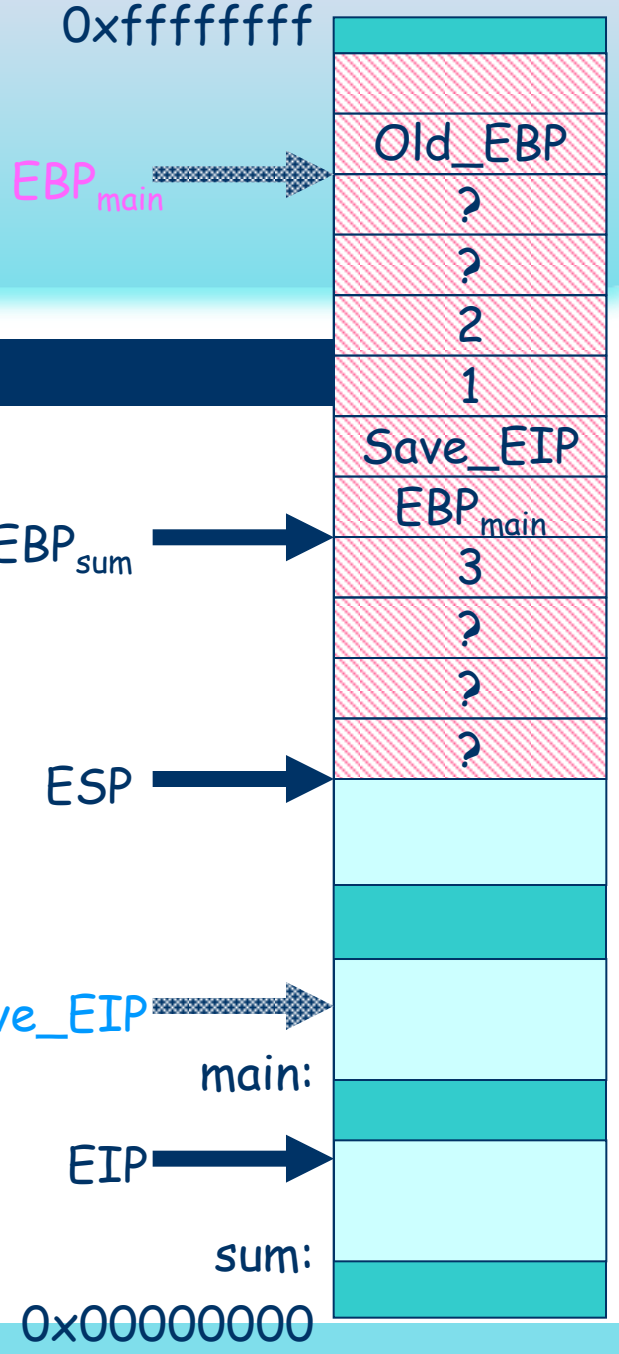
eax	edx
3	1



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

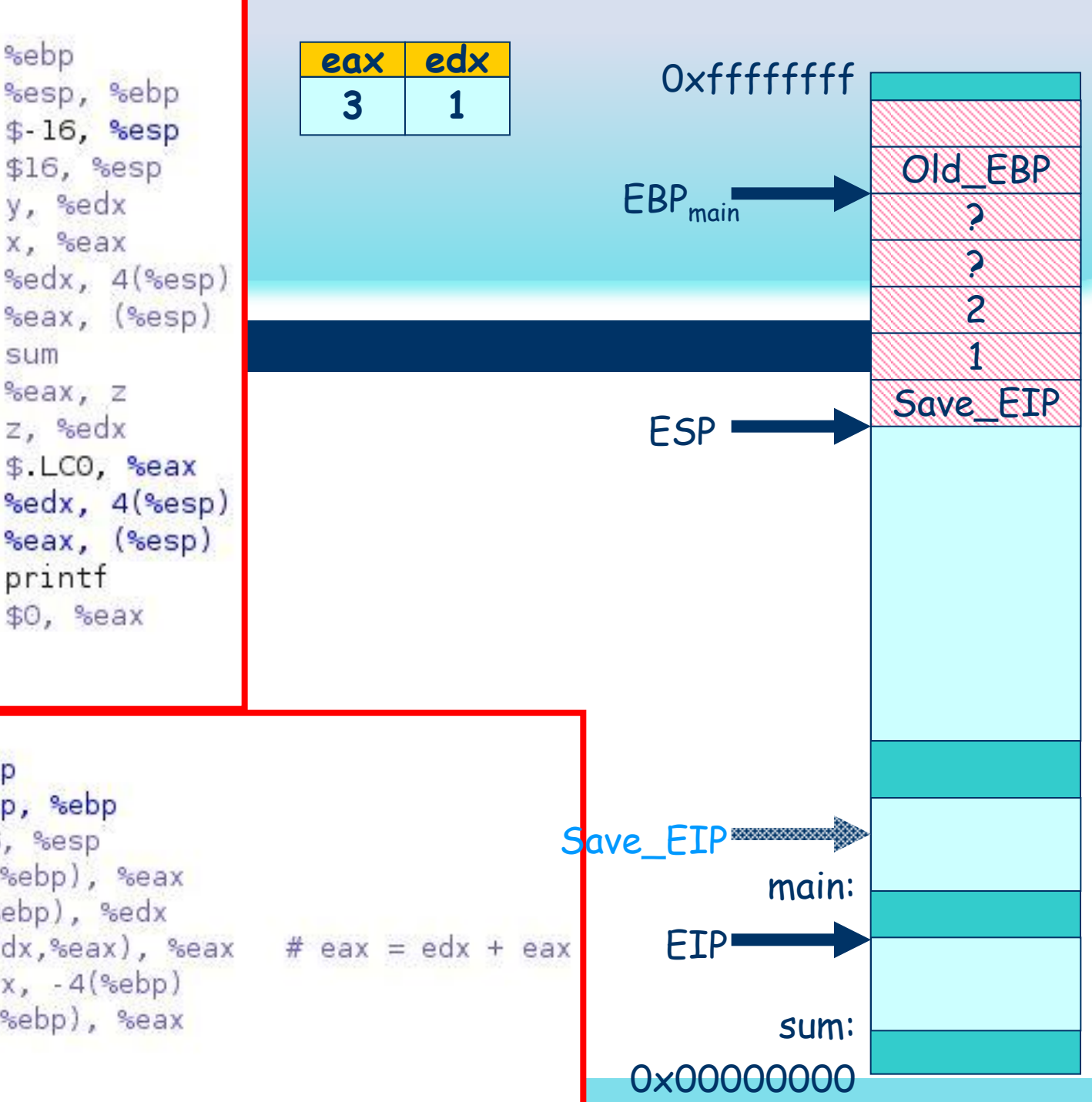
eax	edx
3	1



```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

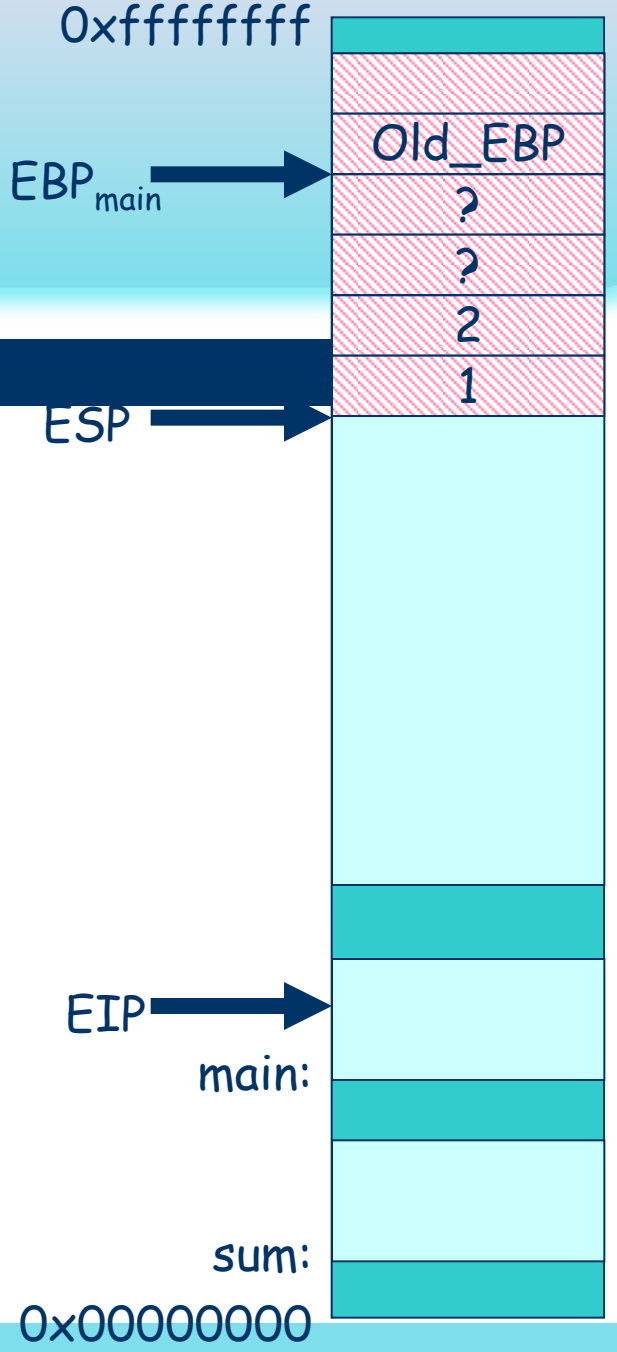
eax	edx
3	1



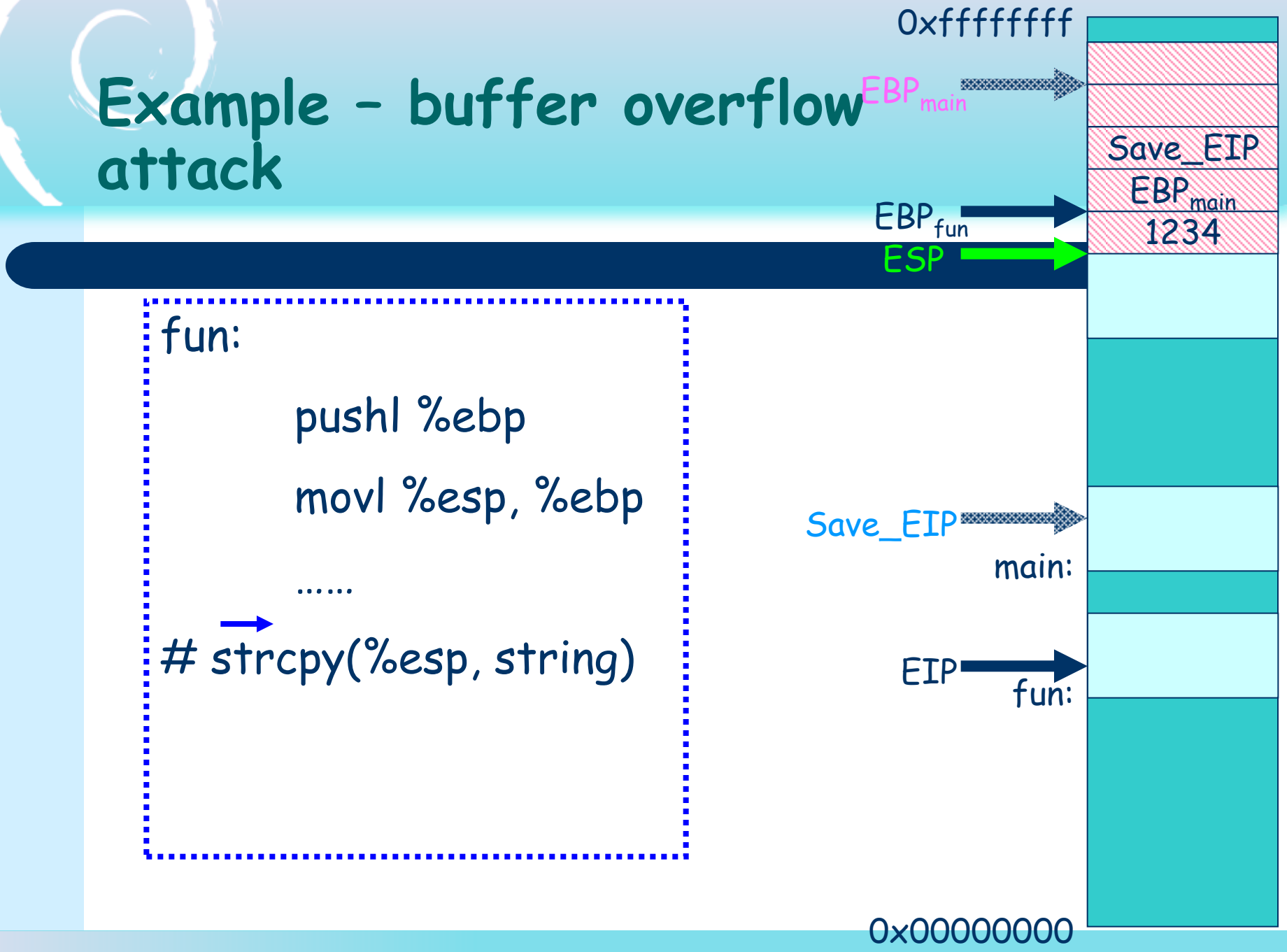
```
37 main:
38     pushl   %ebp
39     movl    %esp, %ebp
40     andl    $-16, %esp
41     subl    $16, %esp
42     movl    y, %edx
43     movl    x, %eax
44     movl    %edx, 4(%esp)
45     movl    %eax, (%esp)
46     call    sum
47     movl    %eax, z
48     movl    z, %edx
49     movl    $.LC0, %eax
50     movl    %edx, 4(%esp)
51     movl    %eax, (%esp)
52     call    printf
53     movl    $0, %eax
54     leave
55     ret
```

```
 5 sum:
 6     pushl   %ebp
 7     movl    %esp, %ebp
 8     subl    $16, %esp
 9     movl    12(%ebp), %eax
10     movl    8(%ebp), %edx
11     leal    (%edx,%eax), %eax   # eax = edx + eax
12     movl    %eax, -4(%ebp)
13     movl    -4(%ebp), %eax
14     leave
15     ret
```

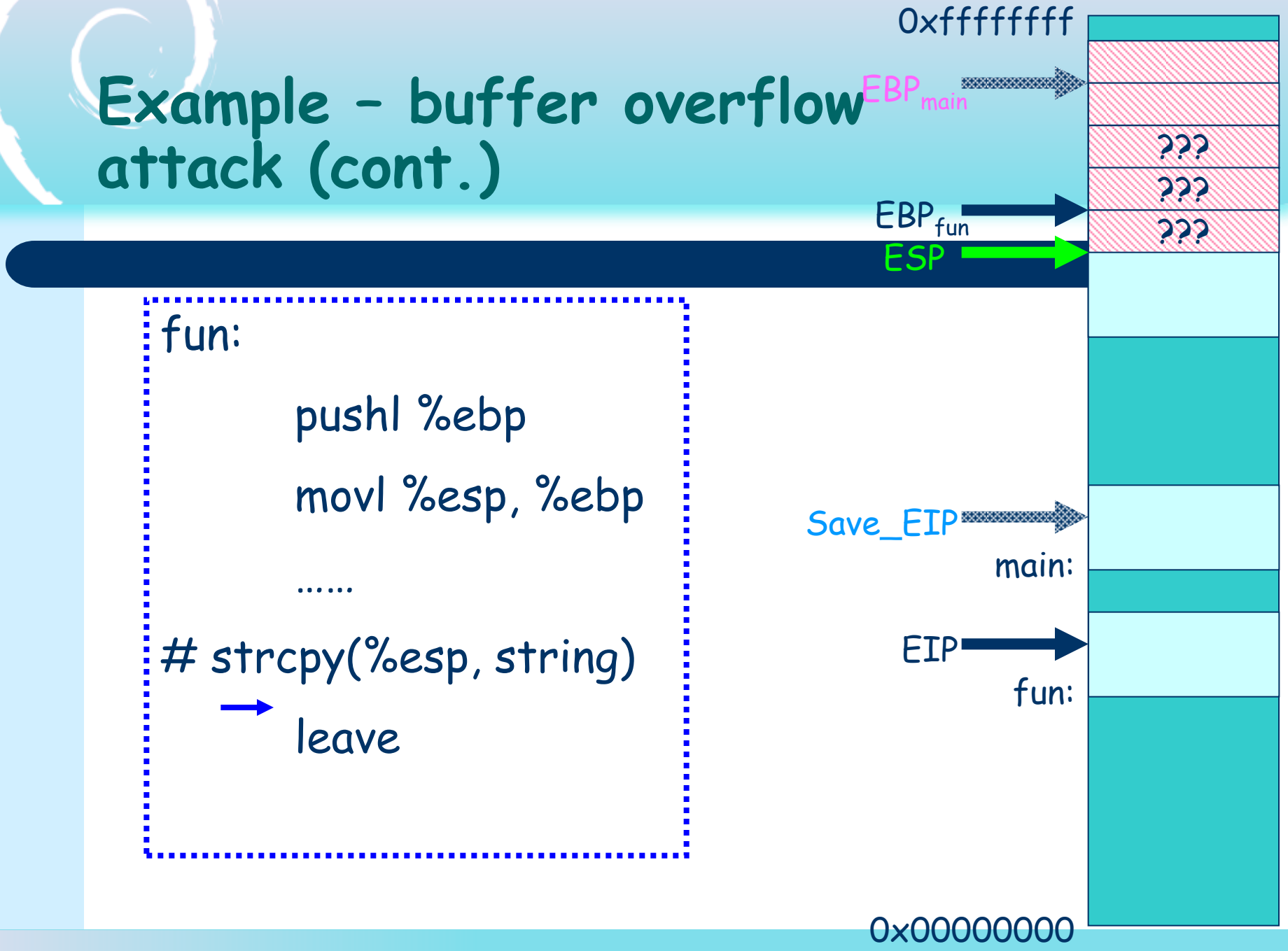
eax	edx
3	1



Example – buffer overflow attack

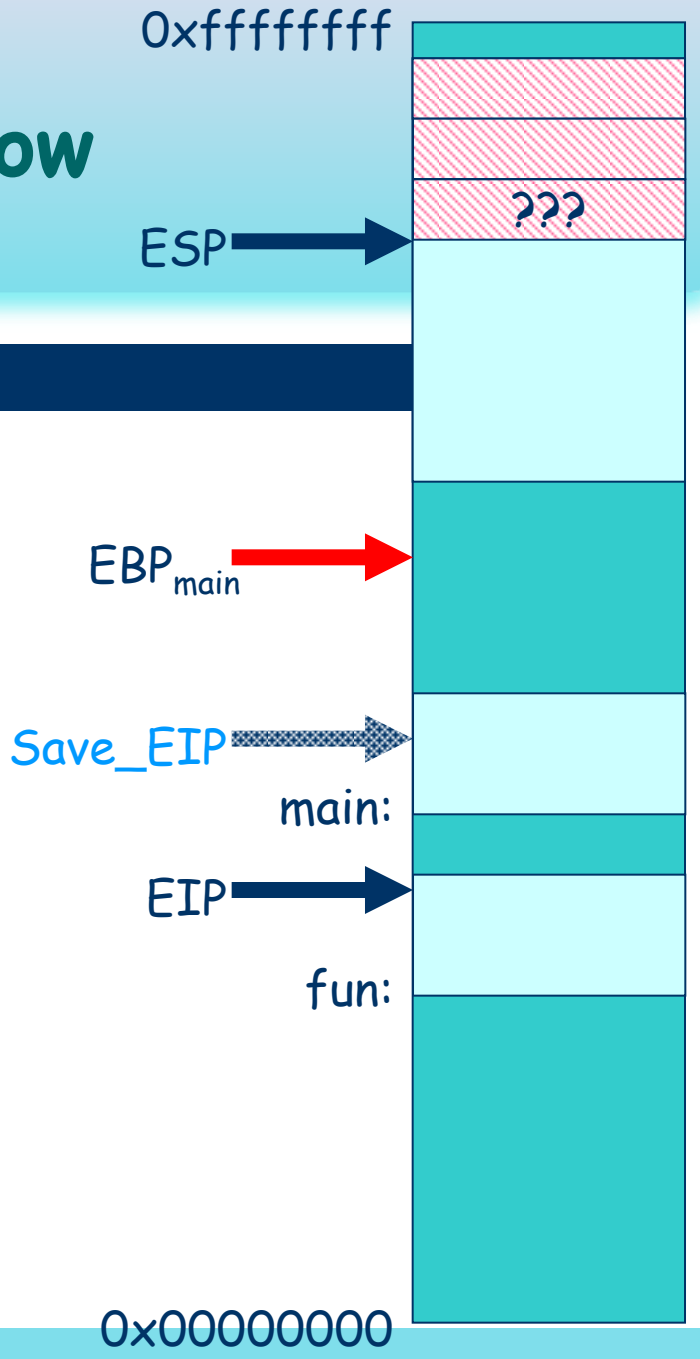


Example – buffer overflow attack (cont.)



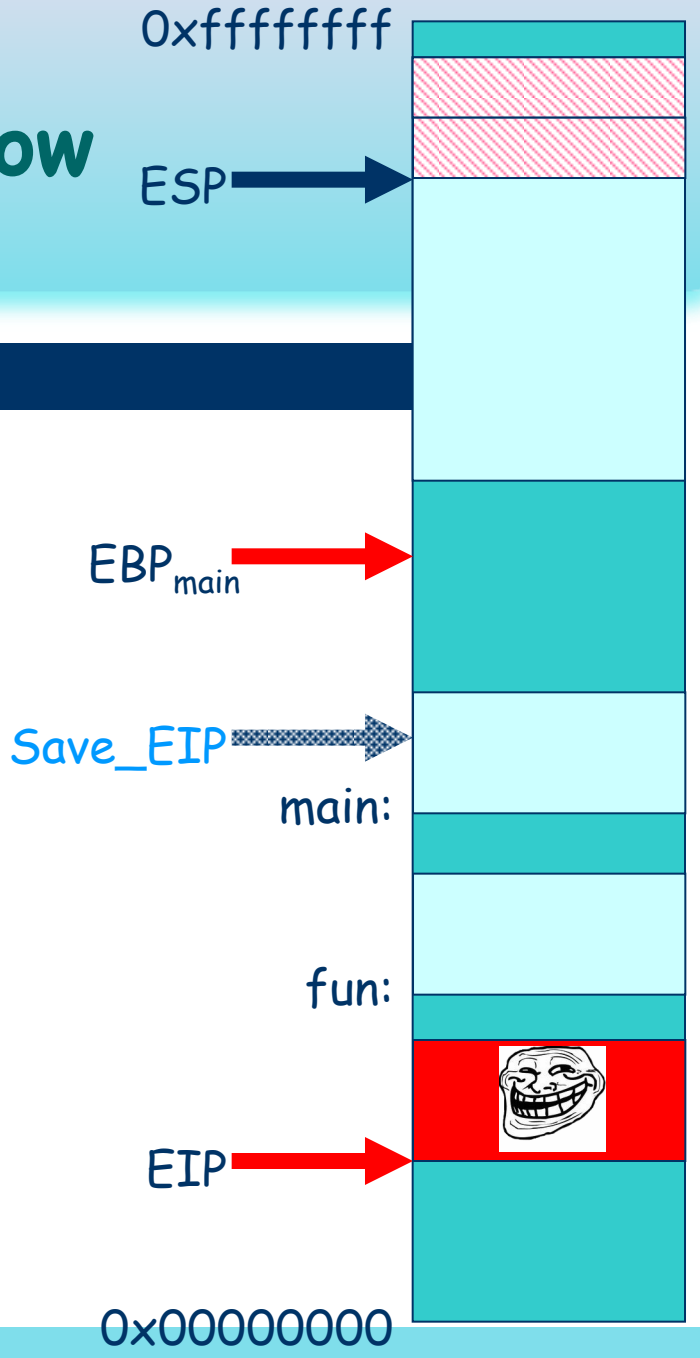
Example – buffer overflow attack (cont.)

```
fun:
        pushl %ebp
        movl %esp, %ebp
        ......
# strcpy(%esp, string)
        leave
        ret
```



Example – buffer overflow attack (cont.)

```
fun:
        pushl %ebp
        movl %esp, %ebp
        .....
# strcpy(%esp, string)
        leave
        ret
```

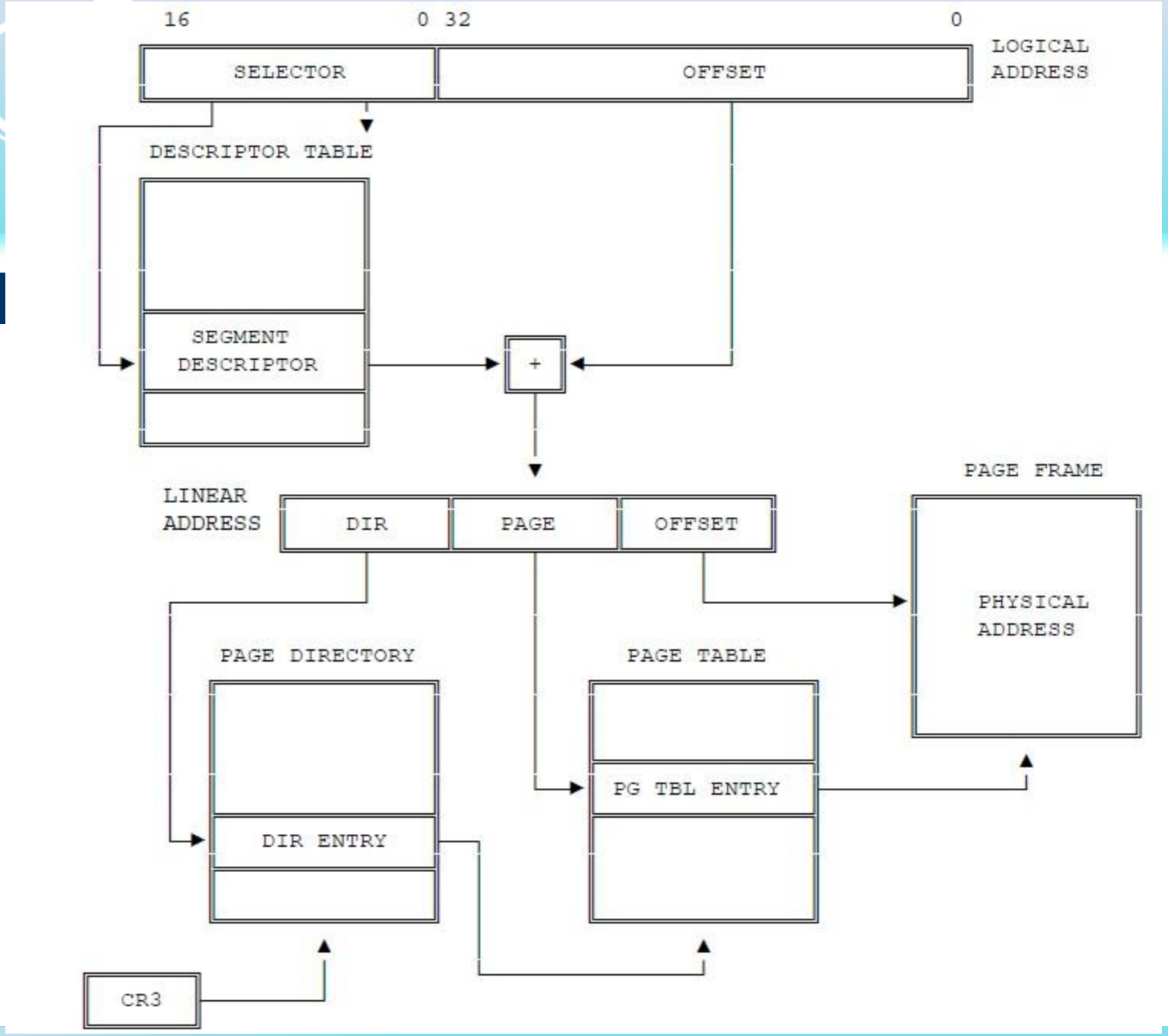


Try to attack this program!

```
 1 #include "stdio.h"
 2 #include "string.h"
 3
 4 void hacker(void) {
 5     printf("being hacked\n");
 6 }
 7
 8 void outputs(char *str) {
 9     char buffer[16];
10     strcpy(buffer,str);
11     printf("%s\n", buffer);
12 }
13
14 int main(int argc, char *argv[]) {
15     outputs(argv[1]);
16     return 0;
17 }
```

Hint: x86 is little-endian

- Recap - IA-32 memory management


16
0 32
0
SELECTOR
OFFSET
LOGICAL ADDRESS
DESCRIPTOR TABLE
SEGMENT DESCRIPTOR
+
LINEAR ADDRESS
DIR
PAGE
OFFSET
PAGE FRAME
PHYSICAL ADDRESS
PAGE DIRECTORY
DIR ENTRY
PAGE TABLE
PG TBL ENTRY
CR3

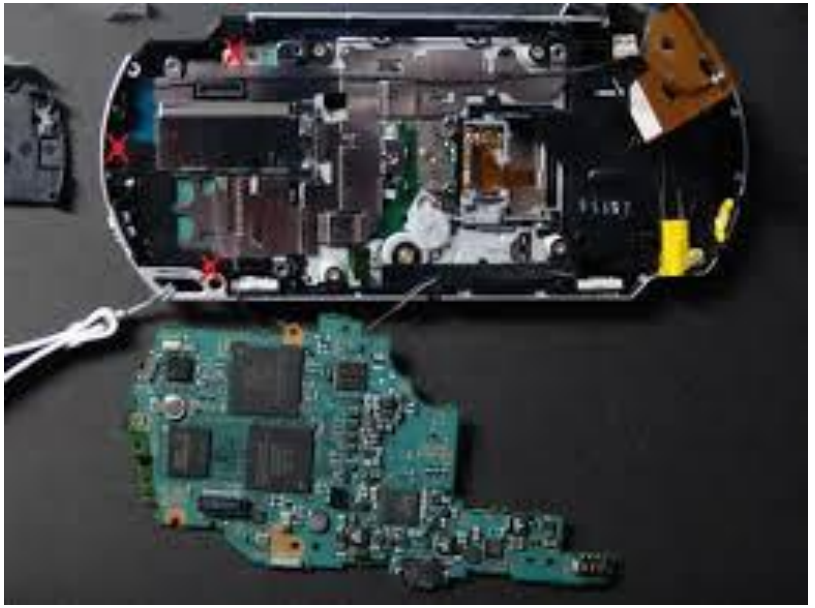
- I/O & Peripherals

Peripherals

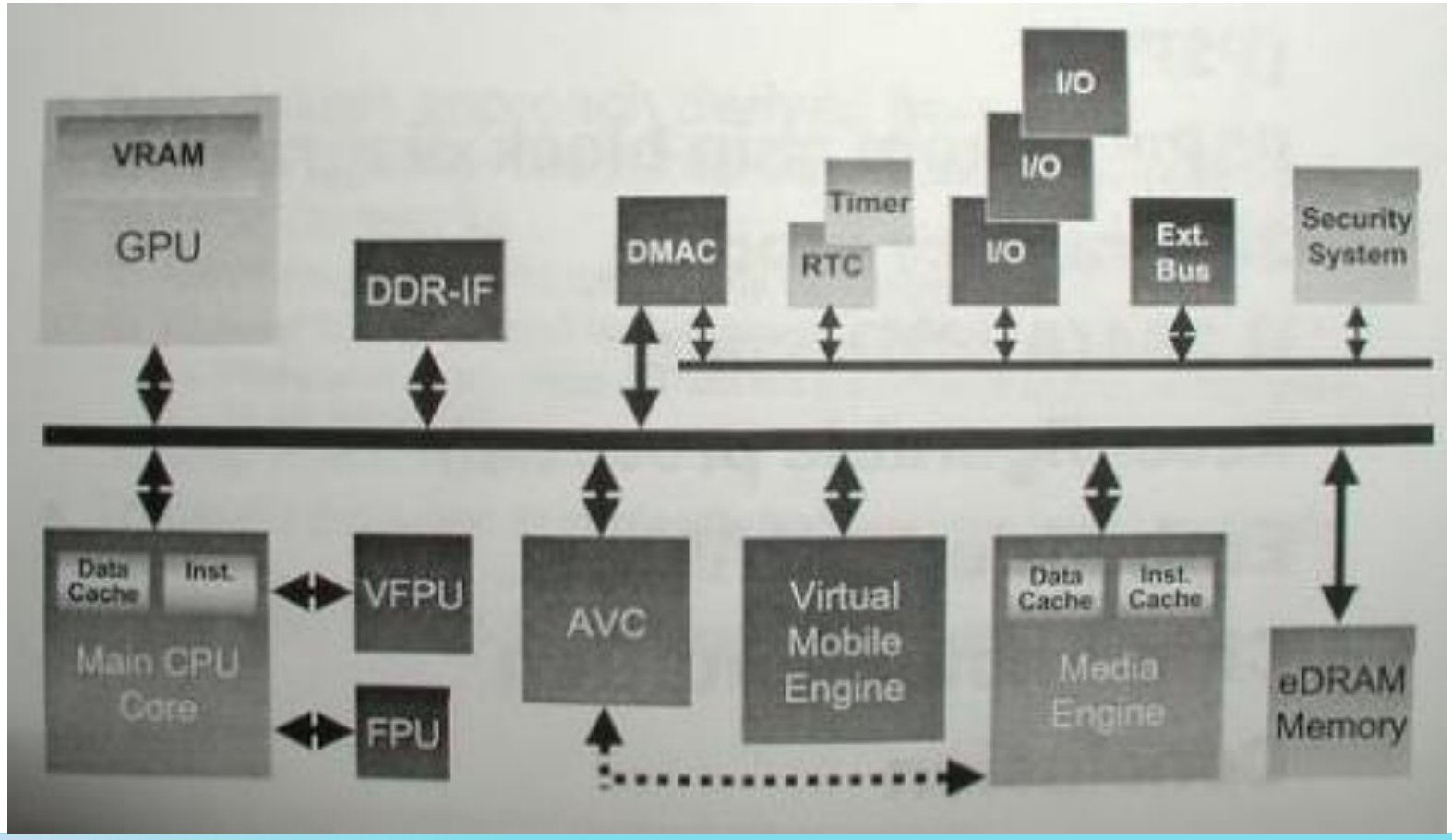
- Why peripherals?
  - CPU can only execute instructions.
  - Peripherals make computer more powerful.
- But how do they communicate with each other to finish a task?

- That is the topic of I/O system.

Example - SONY PSP



SONY PSP block diagram



Some processing units

Game processing unit

- CPU Core
  - MIPS R4000 32bit Core:1-333MHz
  - FPU, VFPU (Vector Unit)
- 3D graphics
  - 'Rendering Engine' + 'Surface Engine'
  - 2MByte eDRAM(VRAM):512bit/166MHz bus I/F
- eDRAM
  - Total capacity of 4Mbytes
  - One half for the game processing unit
  - The other half for the media processing unit
- Security block
  - Protect game/ audio-video contents.
  - Protect the PSP™ as a system.
- I/O
  - Mobile DDR I/F
  - USB 2.0(Device)
  - Memory Stick™

Media processing unit

- Media Engine
  - MIPS R4000 32bit Core with FPU
  - 1-333MHz
- H.264 codec engine
  - H.264 hardware accelerator
- VME(Virtual Mobile Engine)
  - A reconfigurable processor to decode audio/video codec

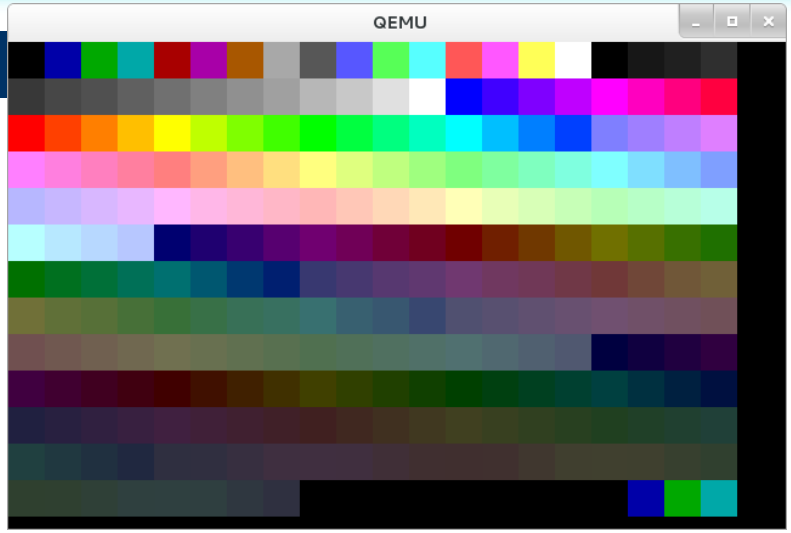
I/O addressing

- port-mapped I/O
  - use special instructions
  - separate address space with memory
  - more control signals for CPU and address bus
- memory-mapped I/O
  - use memory-accessing instructions
  - the same address space with memory
  - no extra control signals for CPU
  - some control signals for address bus

Port-mapped I/O example

```
 3 #define PORT_PIC_MASTER 0x20
 4 #define PORT_PIC_SLAVE  0xA0
 5 #define IRQ_SLAVE       2
 6
 7 /* 初始化8259中断控制器:
 8  * 硬件中断IRQ从32号开始, 自动发送EOI */
 9 void
10 init_intr(void) {
11     out_byte(PORT_PIC_MASTER + 1, 0xFF);
12     out_byte(PORT_PIC_SLAVE + 1 , 0xFF);
13     out_byte(PORT_PIC_MASTER, 0x11);
14     out_byte(PORT_PIC_MASTER + 1, 32);
15     out_byte(PORT_PIC_MASTER + 1, 1 << 2);
16     out_byte(PORT_PIC_MASTER + 1, 0x3);
17     out_byte(PORT_PIC_SLAVE, 0x11);
18     out_byte(PORT_PIC_SLAVE + 1, 32 + 8);
19     out_byte(PORT_PIC_SLAVE + 1, 2);
20     out_byte(PORT_PIC_SLAVE + 1, 0x3);
21
22     out_byte(PORT_PIC_MASTER, 0x68);
23     out_byte(PORT_PIC_MASTER, 0x0A);
24     out_byte(PORT_PIC_SLAVE, 0x68);
25     out_byte(PORT_PIC_SLAVE, 0x0A);
26 }
```

Memory-mapped I/O example




```
11 #define VMEM_ADDR  ((uint8_t*)0xA0000)
12
13 extern uint8_t *vmem;
14
15 static inline void
16 draw_pixel(int x, int y, int color) {
17     assert(x >= 0 && y >= 0 && x < SCR_HEIGHT && SCR_WIDTH);
18     vmem[(x << 8) + (x << 6) + y] = color;
19 }
```

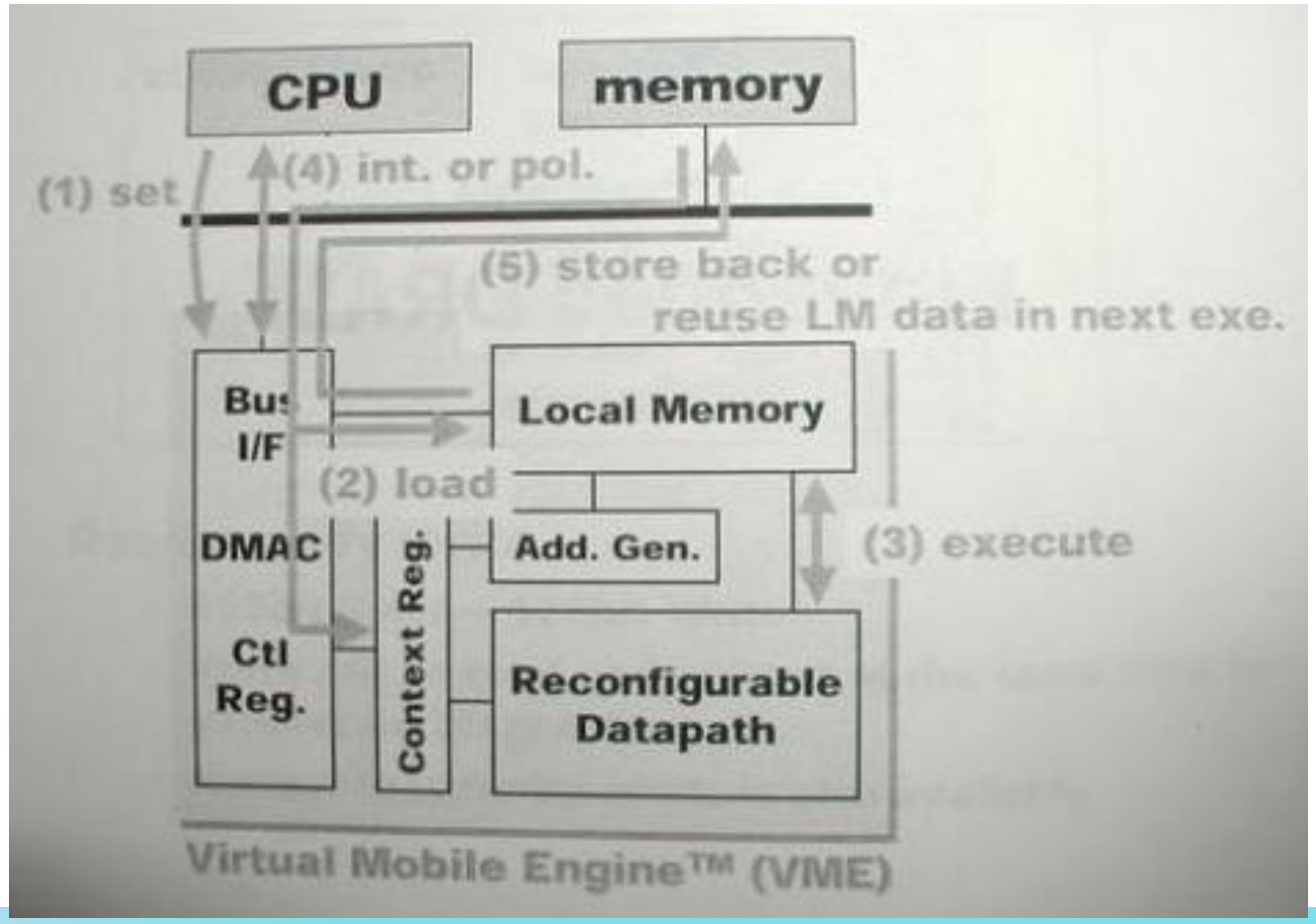
I/O communication

- Polling
  - CPU check the I/O status register repeatedly
  - until the peripheral is ready
- Interrupt
  - when peripheral is ready, or something happen to it, it sends a signal to CPU
  - it is a notification mechanism
- Is interrupt always more efficient then polling?

Polling the serial port

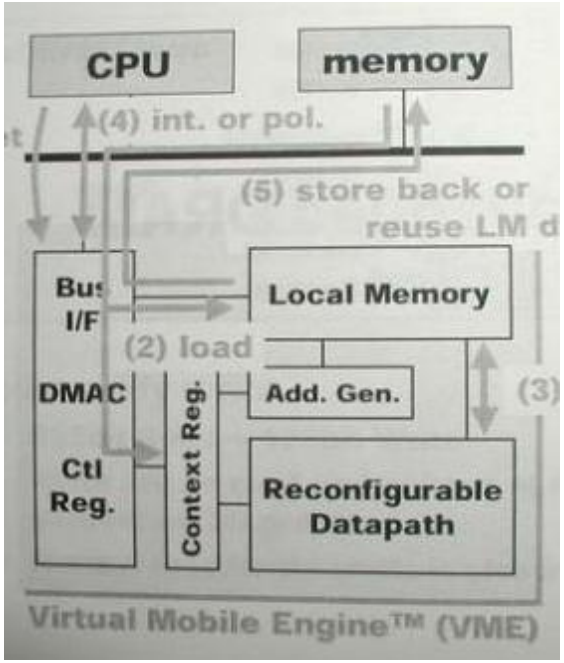
```
16 static inline int
17 serial_idle(void) {
18     return (in_byte(SERIAL_PORT + 5) & 0x20) != 0;
19 }
20
21 void
22 serial_printc(char ch) {
23     while (serial_idle() != TRUE);
24     out_byte(SERIAL_PORT, ch);
25 }
```

Typical VME operation in SONY PSP



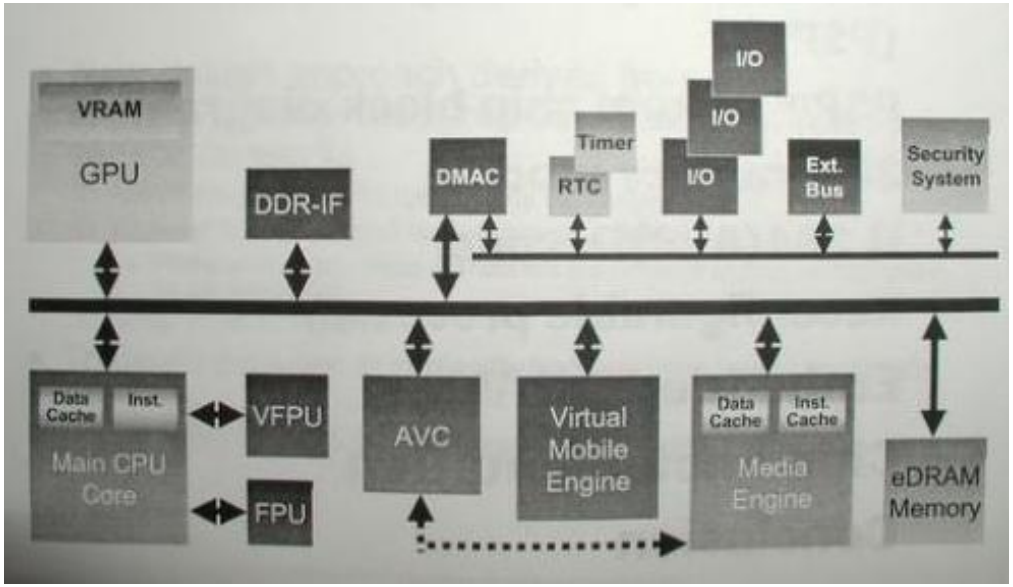
Hardware is not mysterious!

- CPU = registers + controller + exectuion unit
- CPU controller = finite state machine
- So do peripherals!
  - have their own instructions
    - command words from CPU's view
  - generate their own control signals
  - use their own private registers



Hardware is not mysterious! (cont.)

- Use bus to connect them together
  - bus = arbiter + MUX + DEMUX
  - arbiter generates the control signals for MUX and DEMUX
- This is the world of hardware!



- IA-32 Interrupt Processing

Interrupt

- A signal to inform the processor about something that must be handled immediately

Types

- hardware(external) interrupt
  - sent by a device
    - pressing a key
    - timer
- software(internal) interrupt
  - exception
    - divided by 0
    - page fault
  - trap instruction
    - int $0x80

What to do?

- CPU must give response to the interrupt source as soon as possible
  - Why?
- What should CPU do?
  - save the current state
  - call the interrupt handler
  - restore the "current state"

Save the current state

- What should be saved?
  - EFLAGS
    - the IF bit in EFLAGS will be cleared if it is an hardware interrupt
    - other flags should be saved, too
  - CS
    - the handler may be in another code segment
  - EIP
    - return address
- They are done by hardware sequentially

Call the interrupt handler

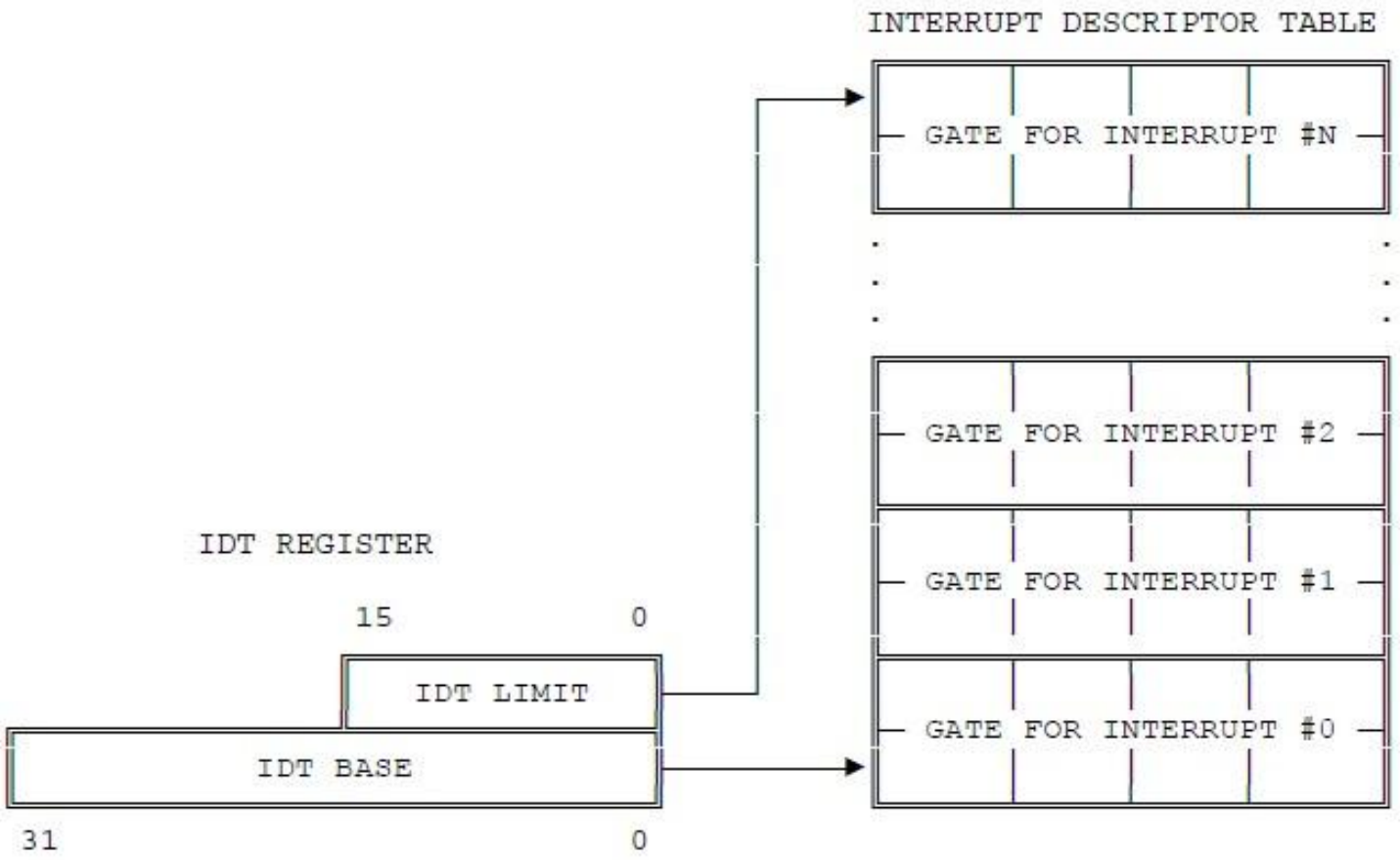
- How to find the entry point of handler?

- IA-32
  - uses gate descriptors to indicate the entry point of an interrupt handler.
  - uses IDT to manage gate descriptors.

IDT

- interrupt descriptor table
  - an array of gate descriptors
  - pointed by IDTR
- There are 3 types of gate descriptor
  - interrupt gate
    - IF in EFLAGS will be cleared – disable interrupt
  - trap gate
    - IF in EFLAGS remains unchanged
  - system gate
    - not used in Nanos

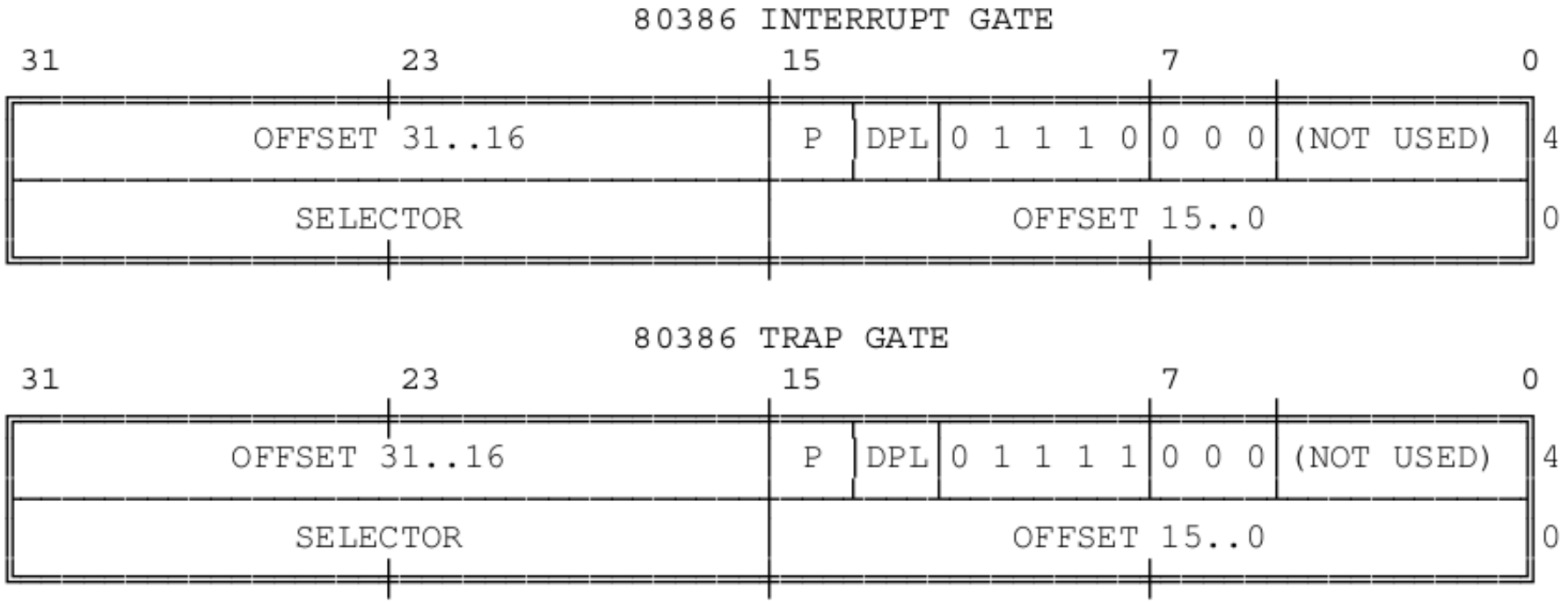
IDT (cont.)


INTERRUPT DESCRIPTOR TABLE
GATE FOR INTERRUPT #N
GATE FOR INTERRUPT #2
GATE FOR INTERRUPT #1
GATE FOR INTERRUPT #0
IDT REGISTER
15
0
IDT LIMIT
IDT BASE
31
0

IDT initialization

```
54 void init_idt() {
55     int i;
56     /* 为了防止系统异常终止，所有irq都有处理函数(irq_empty)。 */
57     for (i = 0; i < NR_IRQ; i ++) {
58         set_trap(idt + i, SEG_KERNEL_CODE, (uint32_t)irq_empty, DPL_KERNEL);
59     }
60
61     /* 设置异常的中断处理 */
62     set_trap(idt + 0, SEG_KERNEL_CODE, (uint32_t)vec0, DPL_KERNEL);
63     set_trap(idt + 1, SEG_KERNEL_CODE, (uint32_t)vec1, DPL_KERNEL);
64     set_trap(idt + 2, SEG_KERNEL_CODE, (uint32_t)vec2, DPL_KERNEL);
65     set_trap(idt + 3, SEG_KERNEL_CODE, (uint32_t)vec3, DPL_KERNEL);
66     set_trap(idt + 4, SEG_KERNEL_CODE, (uint32_t)vec4, DPL_KERNEL);
67     set_trap(idt + 5, SEG_KERNEL_CODE, (uint32_t)vec5, DPL_KERNEL);
68     set_trap(idt + 6, SEG_KERNEL_CODE, (uint32_t)vec6, DPL_KERNEL);
69     set_trap(idt + 7, SEG_KERNEL_CODE, (uint32_t)vec7, DPL_KERNEL);
70     set_trap(idt + 8, SEG_KERNEL_CODE, (uint32_t)vec8, DPL_KERNEL);
71     set_trap(idt + 9, SEG_KERNEL_CODE, (uint32_t)vec9, DPL_KERNEL);
72     set_trap(idt + 10, SEG_KERNEL_CODE, (uint32_t)vec10, DPL_KERNEL);
73     set_trap(idt + 11, SEG_KERNEL_CODE, (uint32_t)vec11, DPL_KERNEL);
74     set_trap(idt + 12, SEG_KERNEL_CODE, (uint32_t)vec12, DPL_KERNEL);
75     set_trap(idt + 13, SEG_KERNEL_CODE, (uint32_t)vec13, DPL_KERNEL);
76
77     /* 设置外部中断的处理 */
78     set_intr(idt + 32, SEG_KERNEL_CODE, (uint32_t)irq0, DPL_KERNEL);
79
80     /* 写入IDT */
81     save_idt(idt, sizeof(idt));
82 }
```

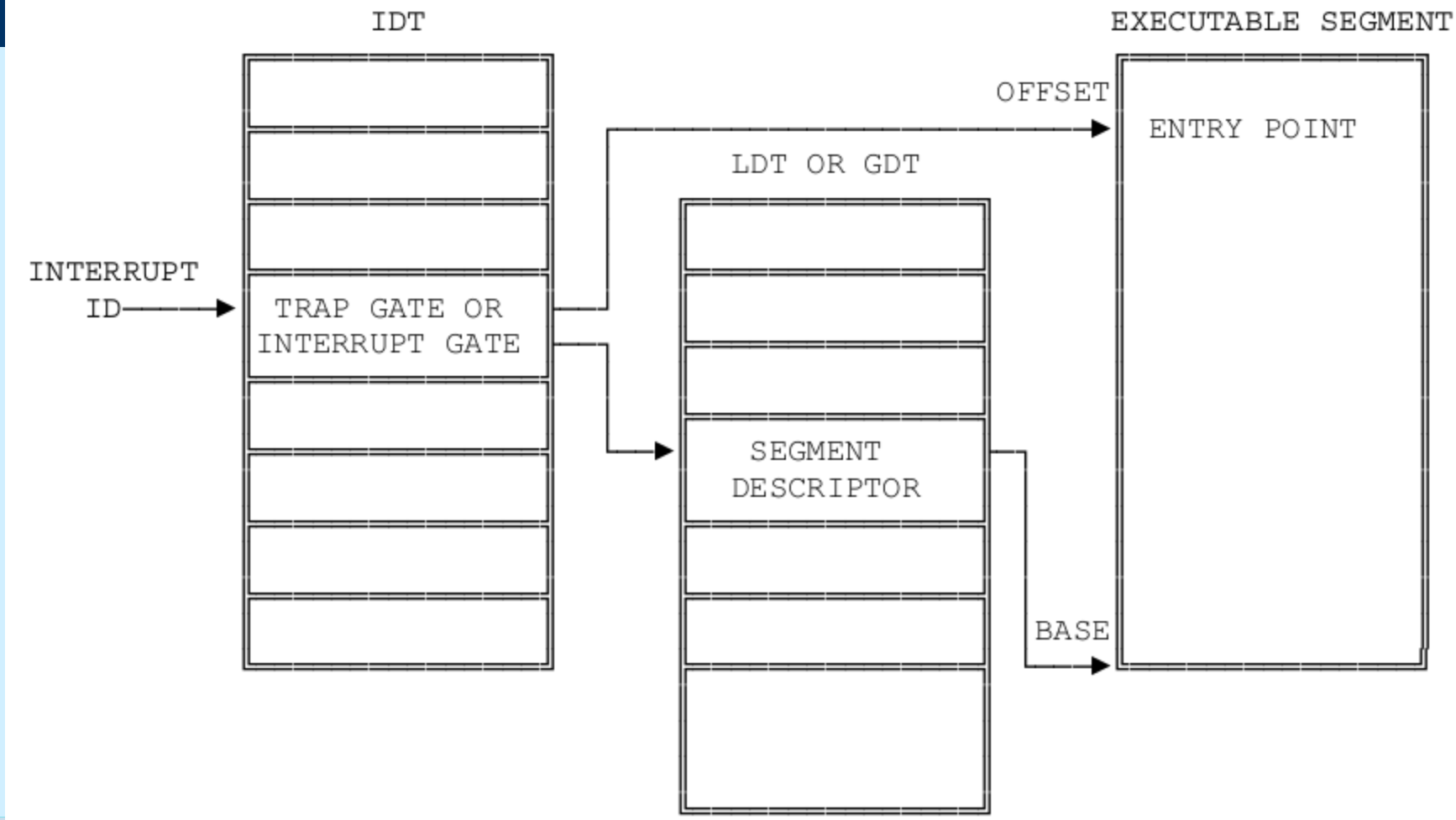
Gate descriptors



Call the interrupt handler (cont.)

- How to find the desired gate descriptor in IDT?
  - use an #id for each interrupt to index in IDT
- Where does #id come from?
  - hardware interrupt: I8259A PIC
  - software interrupt:
    - exception: CPU
    - trap instruction
      - int $0x80

Call the interrupt handler (cont.)



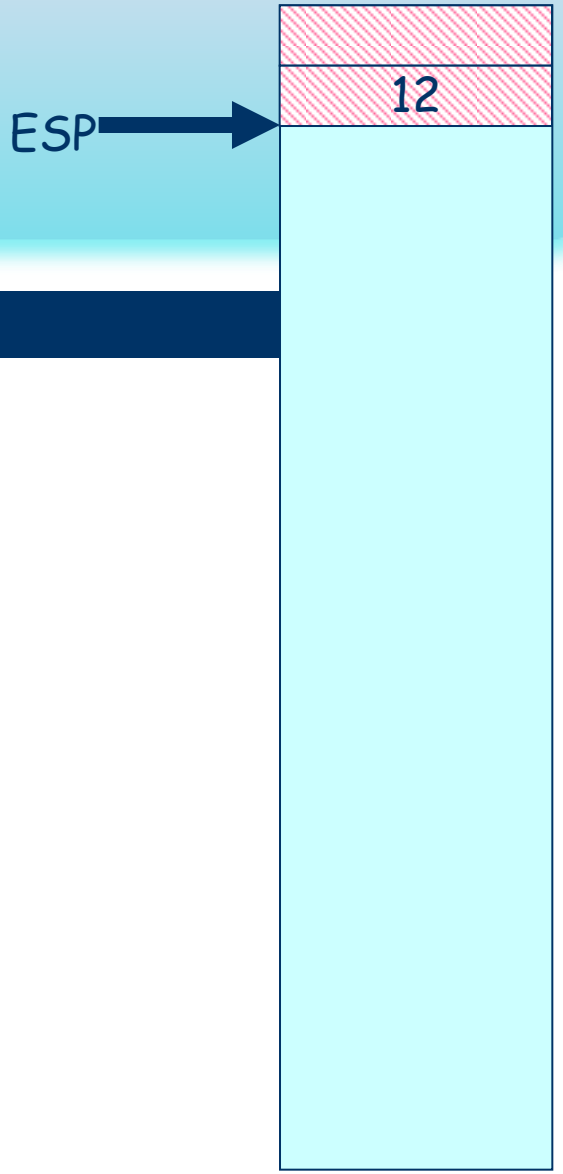
Restore the "current state"

- Use the "iret" instruction to return from interrupt handling
  - restore EIP
  - pop CS
  - restore EFLAGS

More information is needed

- The "current state" saved by hardware is not sufficient
- GPRs should be saved
  - interrupt is unpredictable for CPU
  - why not save them in a function call?
- #irq should be saved
  - to implement interrupt distribution
  - in Lab0, external(hardware) interrupts have #irq >= 1000
  - easy to distinguish

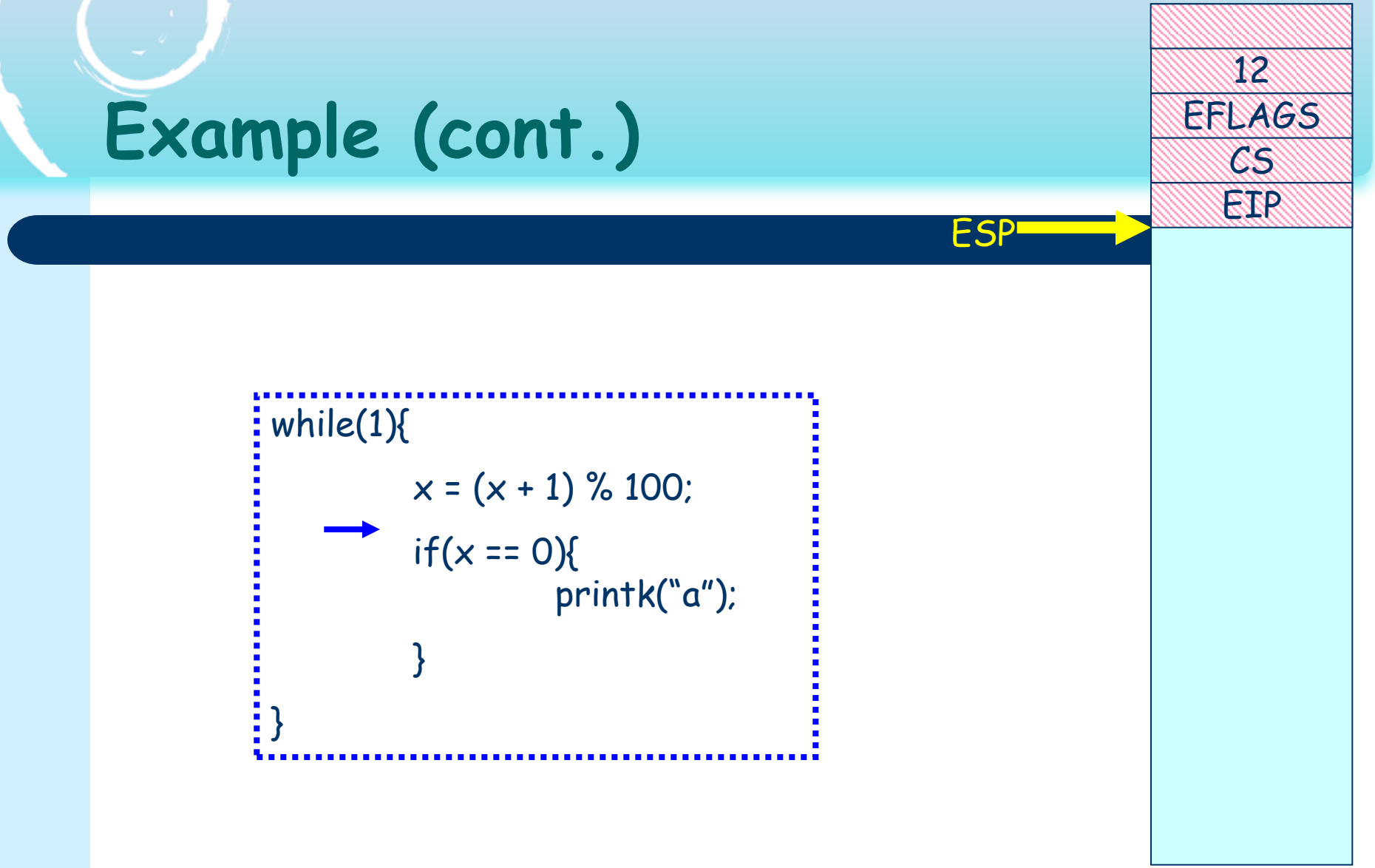
Example



time expires

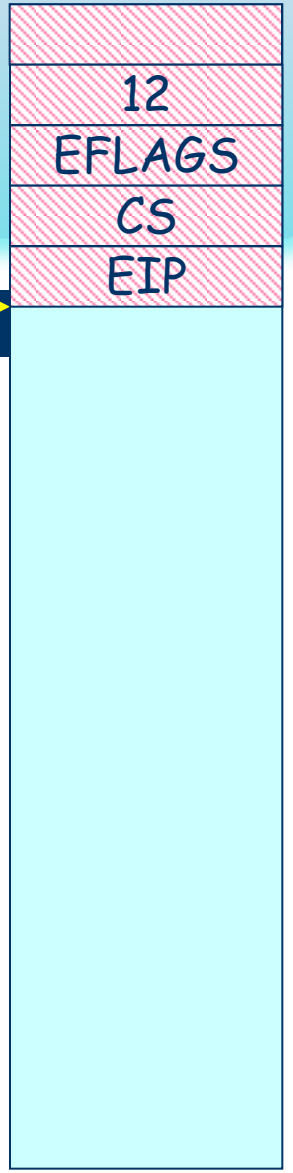
```
while(1){
        x = (x + 1) % 100;
        if(x == 0){
                printk("a");
        }
}
```

Example (cont.)



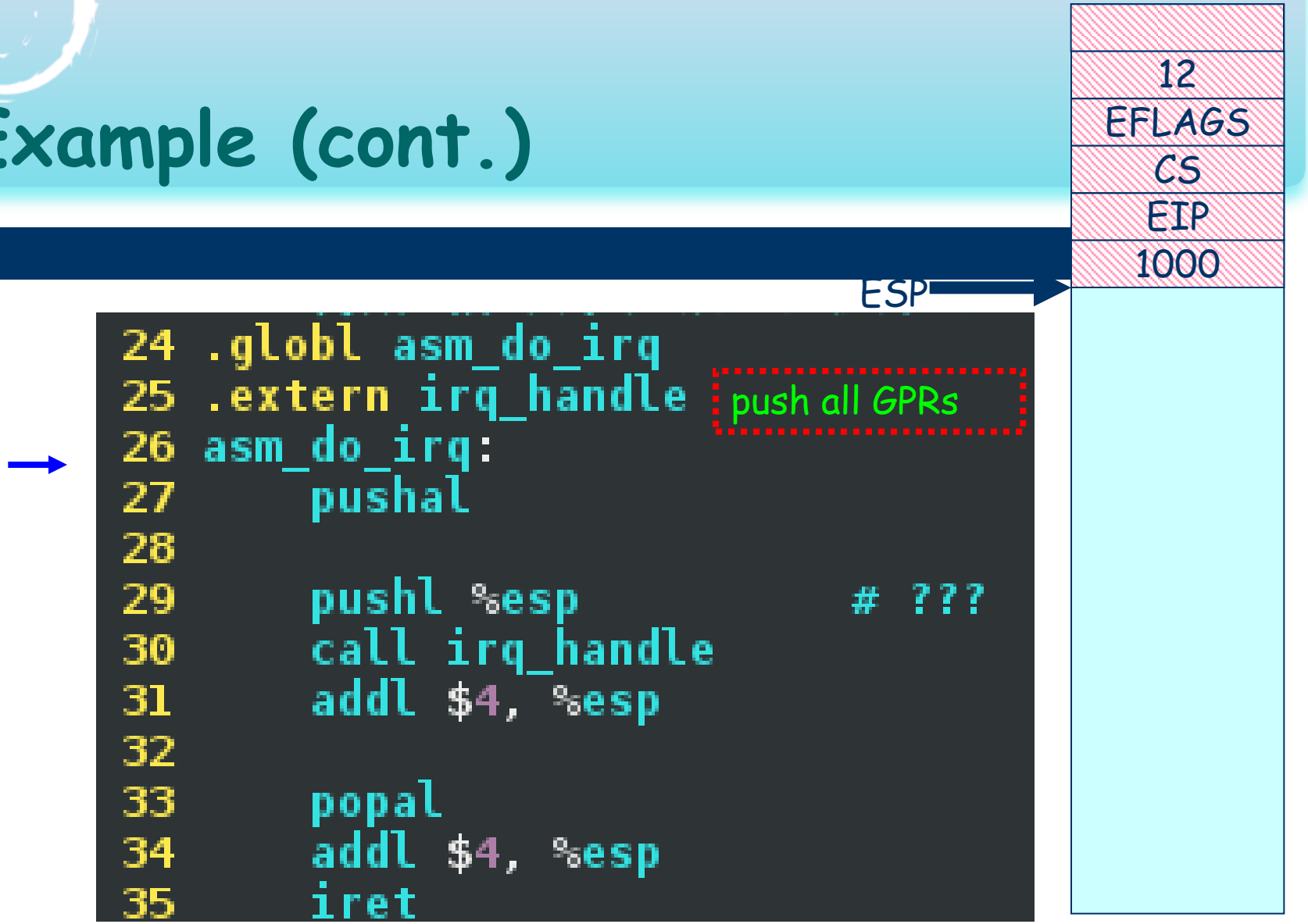

```
while(1){
        x = (x + 1) % 100;
        if(x == 0){
                printk("a");
        }
}
```

Example (cont.)




```
 1 # 中断和异常处理函数的入口
 2 # 中断处理函数会在IDT中为相应的中断/异常设置处理程序
 3 # 中断/异常的行为参见i386手册
 4 .globl vec0; vec0: pushl $0; jmp asm_do_irq
 5 .globl vec1; vec1: pushl $1; jmp asm_do_irq
 6 .globl vec2; vec2: pushl $2; jmp asm_do_irq
 7 .globl vec3; vec3: pushl $3; jmp asm_do_irq
 8 .globl vec4; vec4: pushl $4; jmp asm_do_irq
 9 .globl vec5; vec5: pushl $5; jmp asm_do_irq
10 .globl vec6; vec6: pushl $6; jmp asm_do_irq
11 .globl vec7; vec7: pushl $7; jmp asm_do_irq
12 .globl vec8; vec8: pushl $8; jmp asm_do_irq
13 .globl vec9; vec9: pushl $9; jmp asm_do_irq
14 .globl vec10; vec10: pushl $10; jmp asm_do_irq
15 .globl vec11; vec11: pushl $11; jmp asm_do_irq
16 .globl vec12; vec12: pushl $12; jmp asm_do_irq
17 .globl vec13; vec13: pushl $13; jmp asm_do_irq
18
19 .globl irq0; irq0: pushl $1000; jmp asm_do_irq
20
21 .globl irq_empty; irq_empty: pushl $-1; jmp asm_do_irq
```

Example (cont.)



Example (cont.)

```
24 .globl asm_do_irq
25 .extern irq_handle
26 asm_do_irq:
27     pushal
28
29     pushl %esp                # ???
30     call irq_handle
31     addl $4, %esp
32
33     popal
34     addl $4, %esp
35     iret
```


trap frame

ESP

Example (cont.)

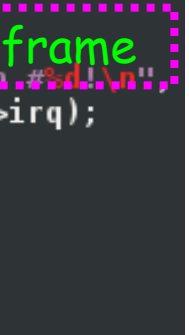
```
24 .globl asm_do_irq
25 .extern irq_handle
26 asm_do_irq:
27     pushal
28
29     pushl %esp              # ???
30     call irq_handle
31     addl $4, %esp
32
33     popal
34     addl $4, %esp
35     iret
```

trap frame

ESP

Example (cont.)

```
18 void
19 irq_handle(struct TrapFrame *tf) {
20     if(tf->irq < 1000) {
21         if(tf->irq == -1) {
22             printk("%s, %d: Unhandled exception!\n",
23                     __FUNCTION__, __LINE__);
24         }
25         else {
26             printk("%s, %d: Unexpected exception #%d!\n",
27                     __FUNCTION__, __LINE__, tf->irq);
28         }
29         assert(0);
30     }
31
32     if (tf->irq == 1000) {
33         do_timer();
34     } else if (tf->irq == 1001) {
35         uint32_t code = in_byte(0x60);
36         uint32_t val = in_byte(0x61);
37         out_byte(0x61, val | 0x80);
38         out_byte(0x61, val);
39
40         printk("%s, %d: key code = %x\n",
41                 __FUNCTION__, __LINE__, code);
42         do_keyboard(code);
43     } else {
44         assert(0);
45     }
46 }
```



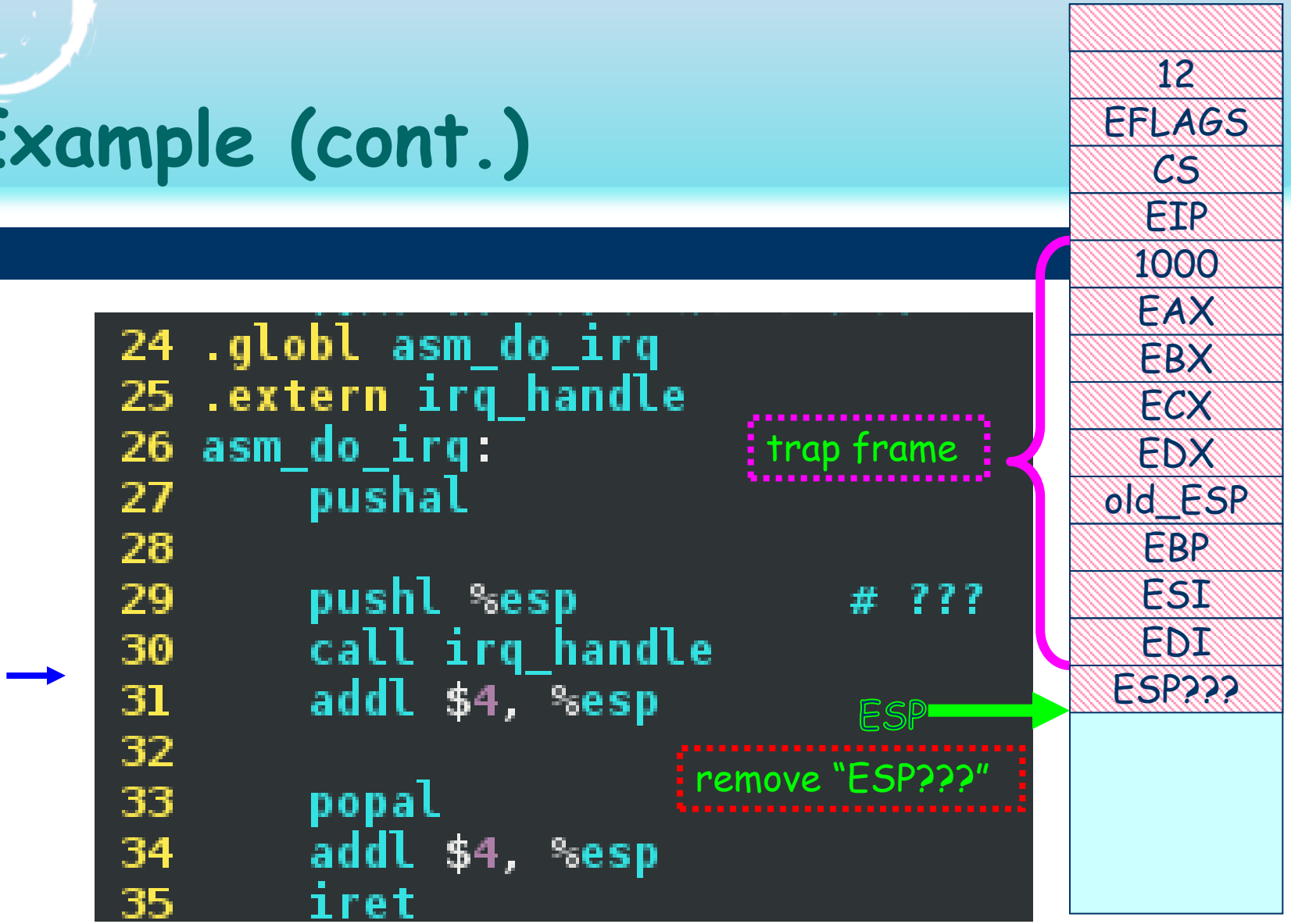
Example (cont.)

```
24 .globl asm_do_irq
25 .extern irq_handle
26 asm_do_irq:
27     pushal
28
29     pushl %esp              # ???
30     call irq_handle
31     addl $4, %esp
32
33     popal
34     addl $4, %esp
35     iret
```

trap frame

ESP

remove "ESP???"



Example (cont.)

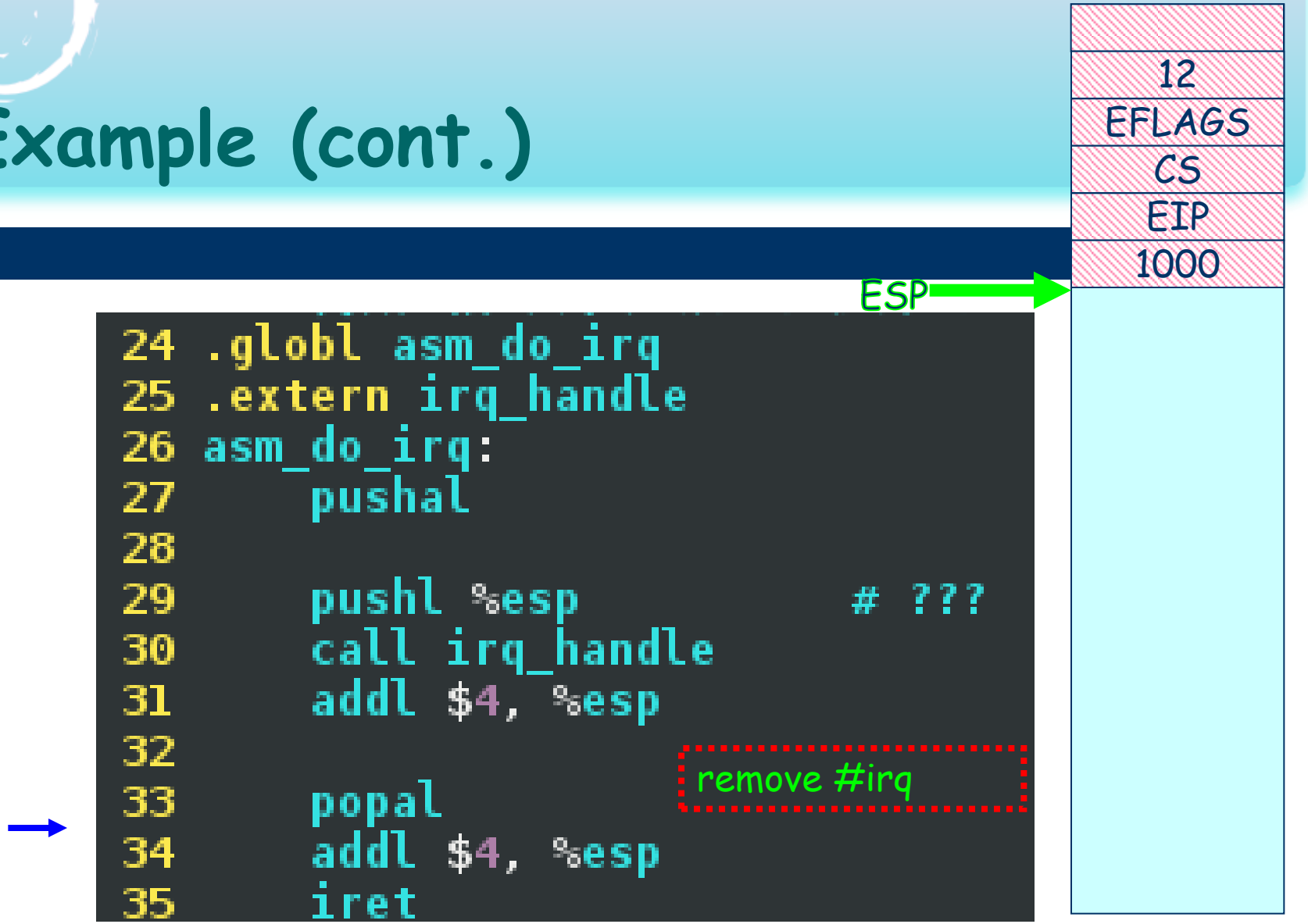
```
24 .globl asm_do_irq
25 .extern irq_handle
26 asm_do_irq:
27     pushal
28
29     pushl %esp                # ???
30     call irq_handle
31     addl $4, %esp
32
33     popal
34     addl $4, %esp
35     iret
```

trap frame

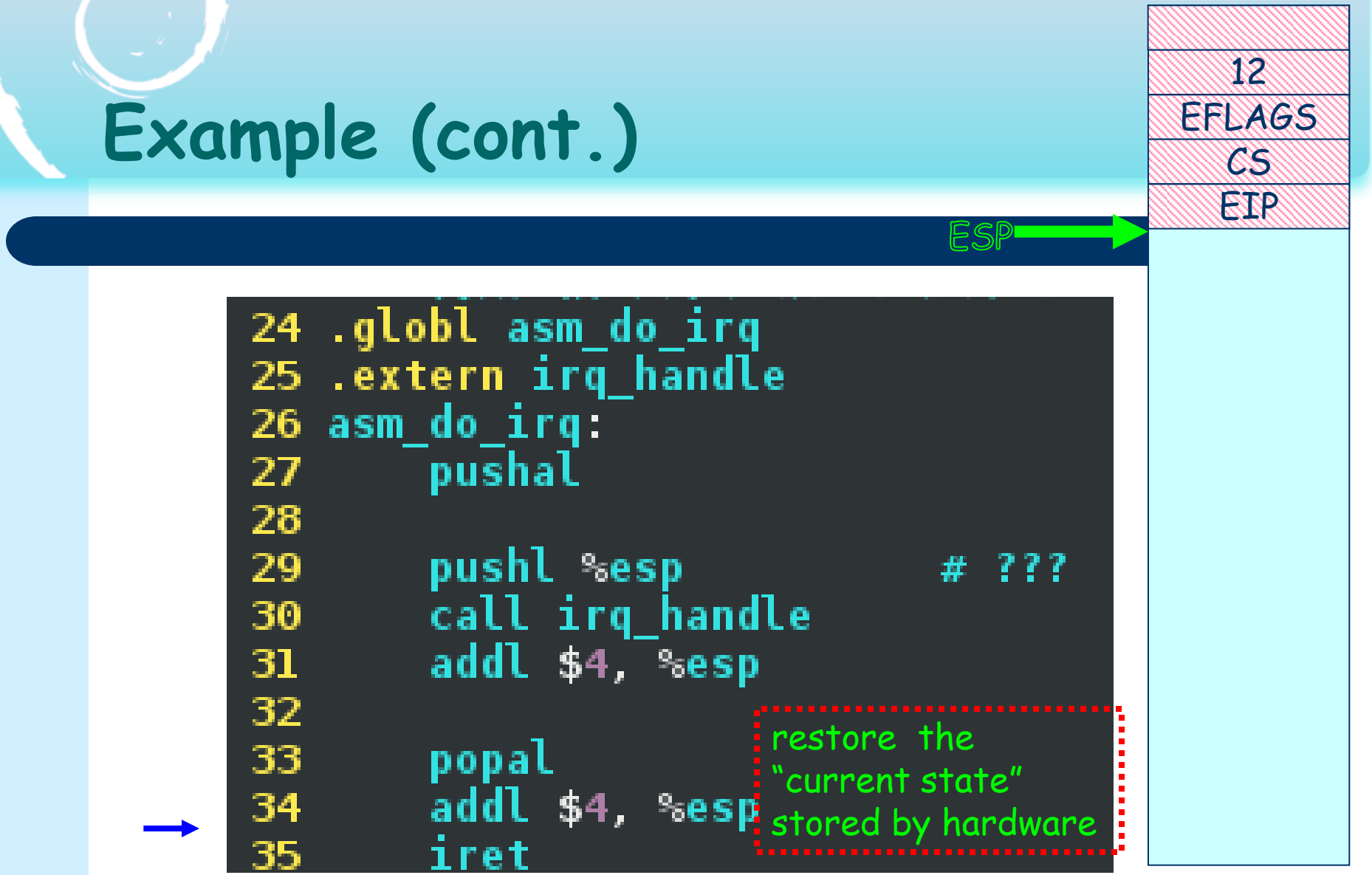
ESP

restore GPRs

Example (cont.)



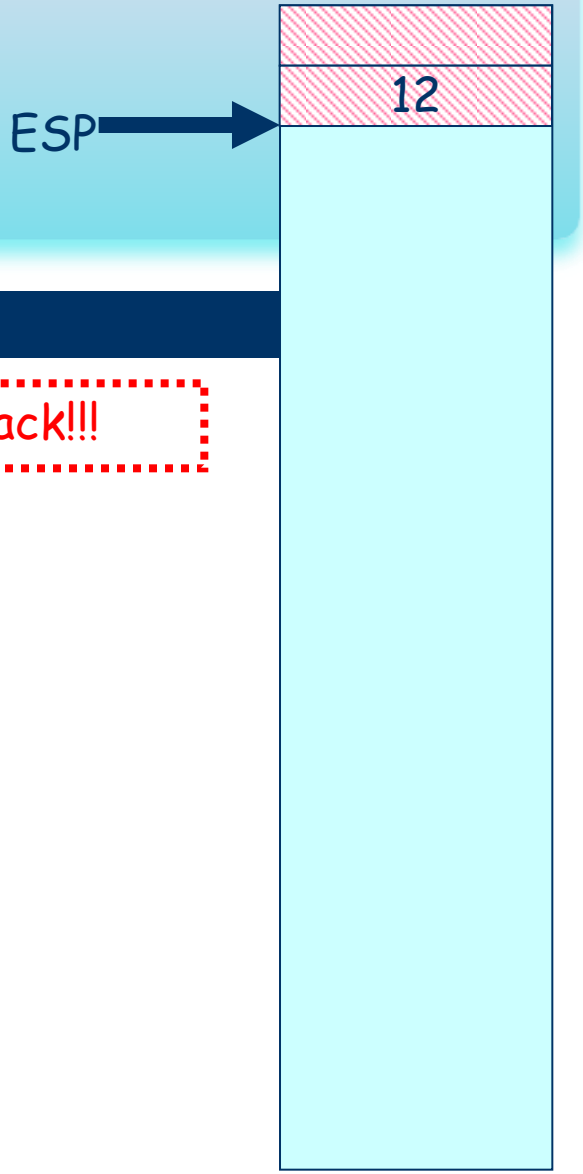
Example (cont.)




```
24 .globl asm_do_irq
25 .extern irq_handle
26 asm_do_irq:
27     pushal
28
29     pushl %esp              # ???
30     call irq_handle
31     addl $4, %esp
32
33     popal
34     addl $4, %esp
35     iret
```

restore the "current state" stored by hardware

Example (cont.)



we are back!!!

```
while(1){
        x = (x + 1) % 100;
        if(x == 0){
                printk("a");
        }
}
```

Exceptions

- They are triggered ONLY by your wrong code

```
 4 .globl vec0; vec0: pushl $0; jmp asm_do_irq
 5 .globl vec1; vec1: pushl $1; jmp asm_do_irq
 6 .globl vec2; vec2: pushl $2; jmp asm_do_irq
 7 .globl vec3; vec3: pushl $3; jmp asm_do_irq
 8 .globl vec4; vec4: pushl $4; jmp asm_do_irq
 9 .globl vec5; vec5: pushl $5; jmp asm_do_irq
10 .globl vec6; vec6: pushl $6; jmp asm_do_irq
11 .globl vec7; vec7: pushl $7; jmp asm_do_irq
12 .globl vec8; vec8: pushl $8; jmp asm_do_irq
13 .globl vec9; vec9: pushl $9; jmp asm_do_irq
14 .globl vec10; vec10: pushl $10; jmp asm_do_irq
15 .globl vec11; vec11: pushl $11; jmp asm_do_irq
16 .globl vec12; vec12: pushl $12; jmp asm_do_irq
17 .globl vec13; vec13: pushl $13; jmp asm_do_irq
```

The magic of Interrupt

- DMA
- for the game in Lab0
  - frame-based rendering
  - drive the logical time
  - get inputs from player
- for OS
  - context switching (very significant)
- refer to INTEL 80386 PROGRAMMER'S REFERENCE MANUAL for more details about interrupts