

Operating Systems

Introduction to Lab 1

Department of Computer Science & Technology
Tsinghua University

- ◆ Hand in your homework: HOWTO
- ◆ x86 boot sequence
- ◆ C function calls implementation
- ◆ GCC inline assembly
- ◆ Interrupt handling in x86 architecture

- Know how to clone/pull/push/commit a git repository

HAND IN YOUR HOMEWORK: HOWTO

Hand in your homework: HOWTO

- ◆ Clone your copy of the code and setup personal info
 - Follow the instructions in the mail
- ◆ Complete a lab
- ◆ Stage & commit your changes
 - `git add lab1/kern/debug/kdebug.c`
 - `git commit -s -m "Solution to lab 1"`
 - `git push origin master`
- ◆ Your solution will be automatically tested on the autobuild system
 - <http://os.cs.tsinghua.edu.cn:3100/>
 - You'll receive a mail on test results

Hand in your homework: HOWTO

- ◆ **WARNING!**
 - The git service may not be always available, esp. at nights.
 - Complete the labs and hand in your solutions ASAP

Hand in your homework: HOWTO – References

- ◆ Code School on Github: <http://try.github.io>

- Understand how x86 platform boots up

X86 BOOT SEQUENCE

x86 boot sequence – initial values of registers

Table 9-1. IA-32 Processor States Following Power-up, Reset, or INIT

Register	Pentium 4 and Intel Xeon Processor	P6 Family Processor (Including DisplayFamily = 06H)	Pentium Processor
EFLAGS ¹	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H ²	60000010H ²	60000010H ²
CR2, CR3, CR4	00000000H	00000000H	00000000H
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/w, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/w, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/w, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/w, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/w, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/w, Accessed
EDX	00000FxxH	000n06xxH ³	000005xxH
EAX	0 ⁴	0 ⁴	0 ⁴
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H

x86 boot sequence – hacks for the first instruction

- ◆ CS = F000H, EIP = 0000FFF0H
- ◆ But the actual address is:
 $\text{Base} + \text{EIP} = \text{FFFF0000H} + \text{0000FFF0H} = \text{FFFFFFFF0H}$
This is where the EPROM (Erasable Programmable Read Only Memory) should reside in
- ◆ After CS is loaded with a new value, the normal address translation rule (see below) will take effect
- ◆ Usually, the first instruction executed is a long jump (both CS and EIP will be updated) to the BIOS code

x86 boot sequence – segmentation in real mode

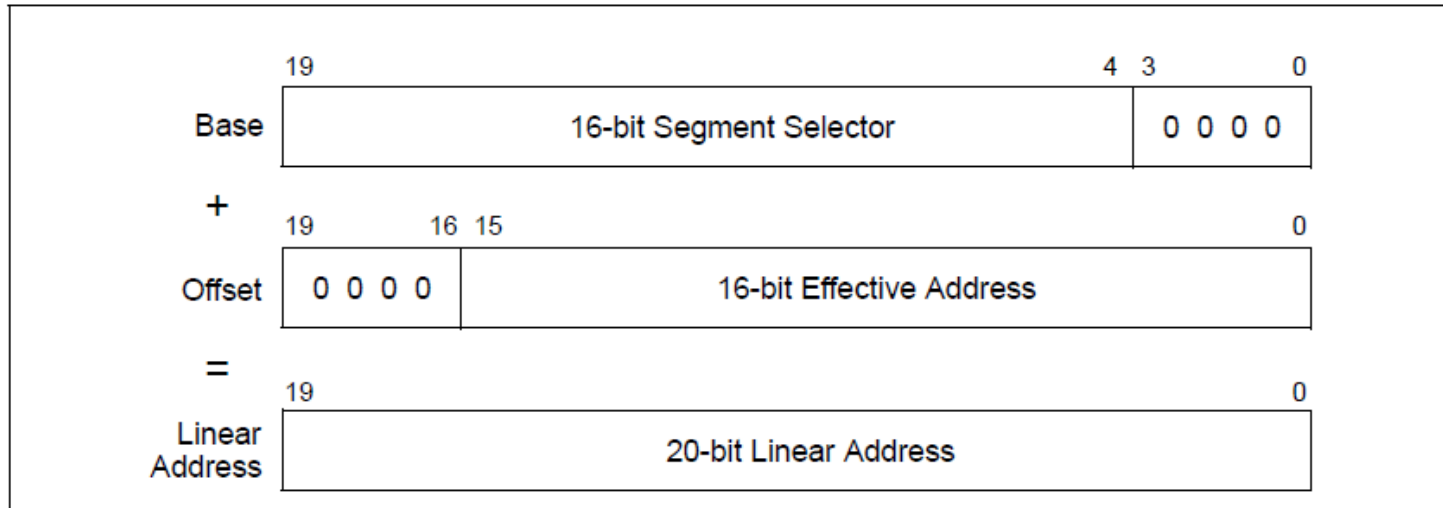


Figure 20-1. Real-Address Mode Address Translation

- ◆ Segment Selector: CS, DS, SS, ...
- ◆ Offset: saved in EIP

x86 boot sequence – from BIOS to bootloader

- ◆ BIOS load the first 512B (Master Boot Record, or MBR) of the boot device to 0x7c00 ...
 - Maybe from hard disk, USB or CD/DVD
- ◆ ... and goto the first instruction @ 0x7c00

x86 boot sequence – from bootloader to kernel

- ◆ Bootloader will:
 - enable protection mode & segment-level protection,
 - read kernel in ELF format from disk (just following MBR) and place it at the right place, and
 - jump to the entry point of the kernel

x86 boot sequence – segment-level protection

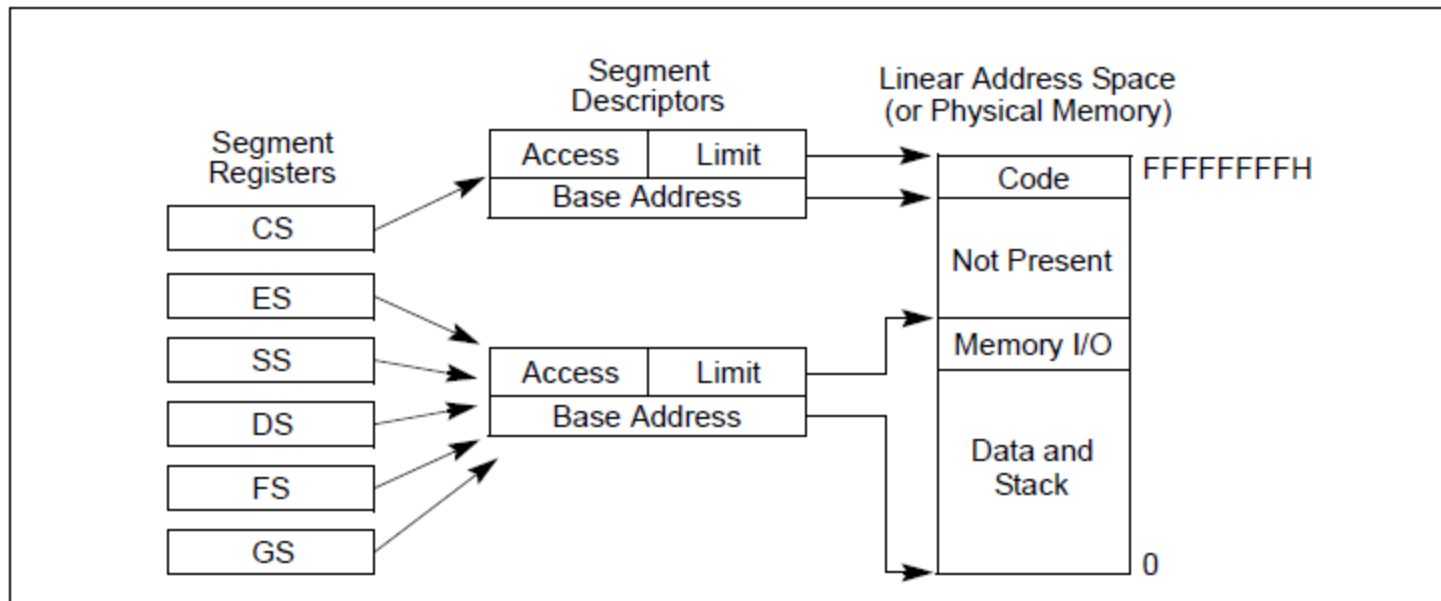


Figure 3-3. Protected Flat Model

x86 boot sequence – segment-level protection

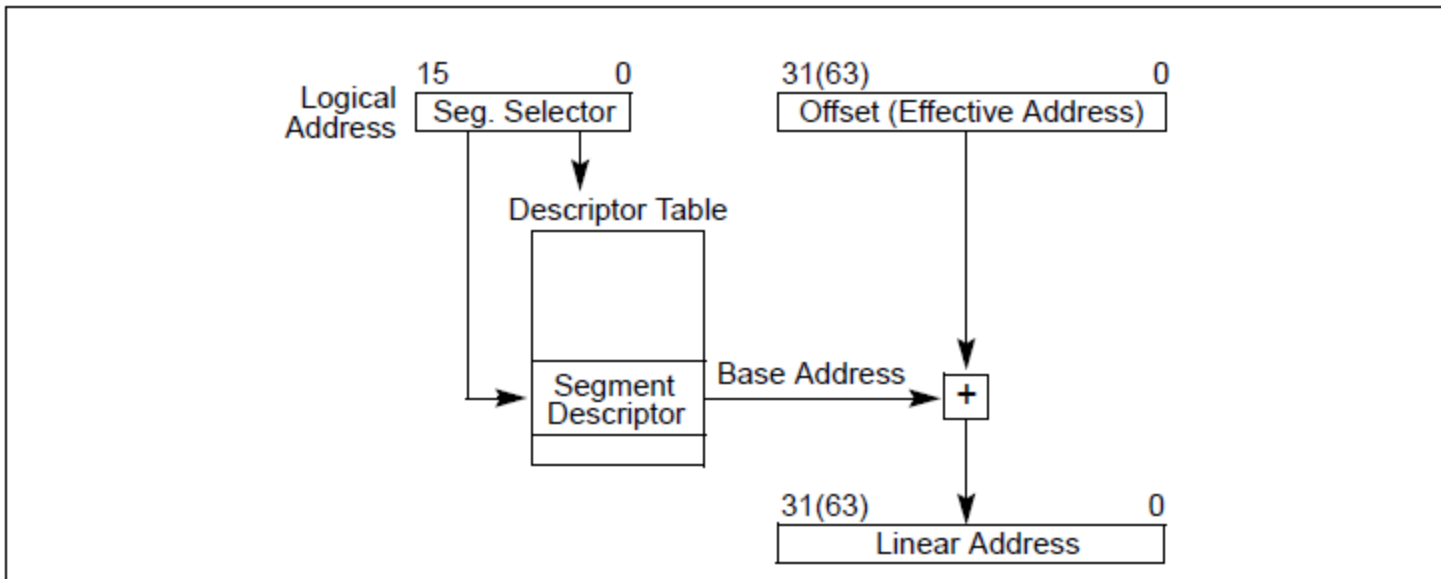


Figure 3-5. Logical Address to Linear Address Translation

x86 boot sequence – segment-level protection

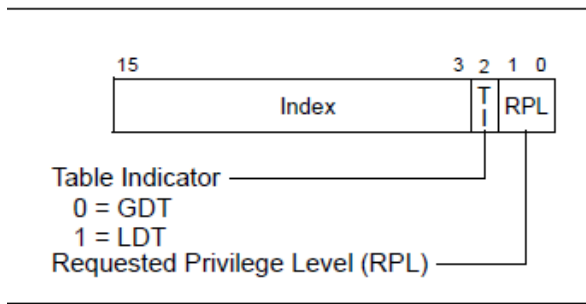


Figure 3-6. Segment Selector

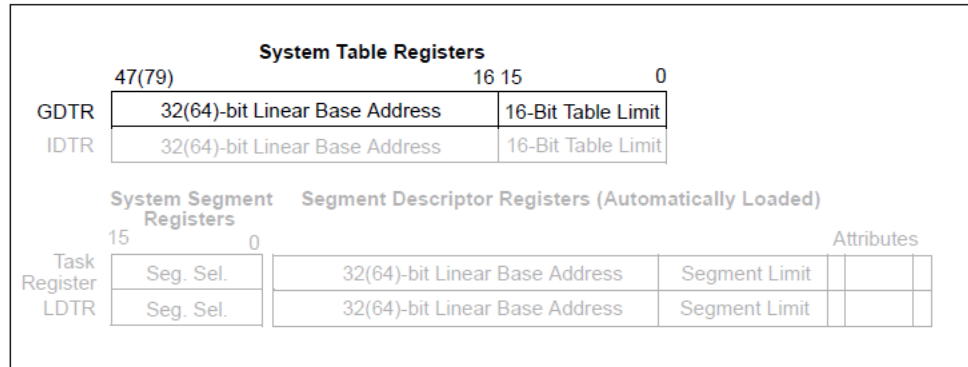


Figure 2-6. Memory Management Registers

- Loading GDT:

```
lgdt gdt desc
```

```
gdt:
    .....
```

```
gdt desc:
    .word 0x17
    .long gdt
```

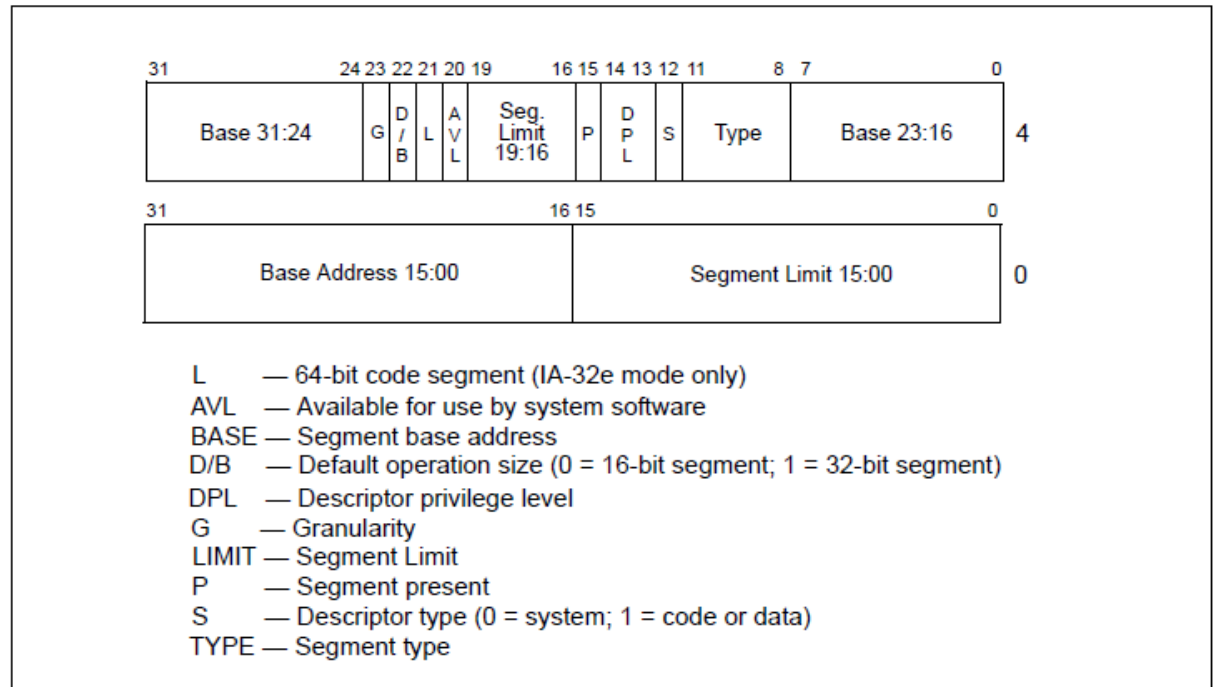


Figure 3-8. Segment Descriptor

x86 boot sequence – enable protection mode

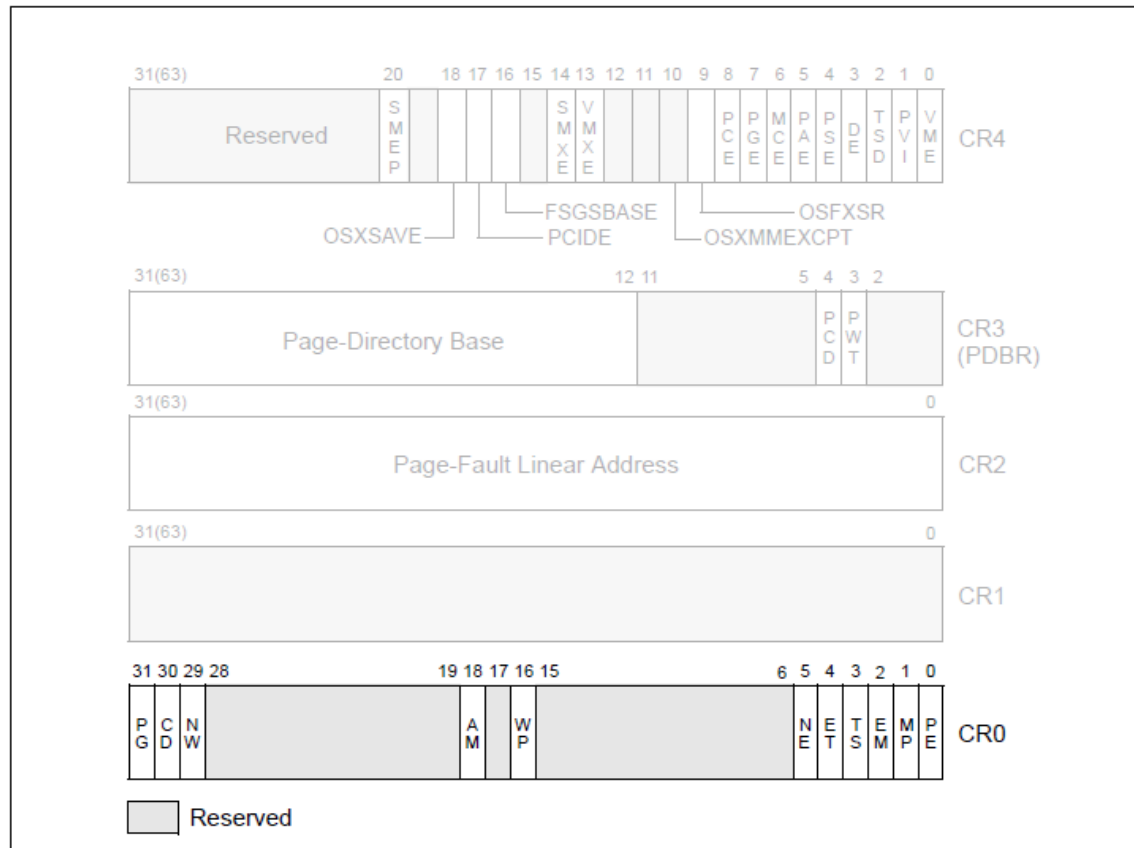


Figure 2-7. Control Registers

- ◆ To enable protection mode, OS should set bit 0 (PE) in CR0
- ◆ Segment-level protection is automatically enabled in protection mode

x86 boot sequence – what is ELF?

- ◆ ELF = Executable and Linking Format
- ◆ E.g: *.o, *.a, *.so, *(.exe)

x86 boot sequence – loading ELF kernel

```
struct elfhdr {
    uint magic;          // must equal ELF_MAGIC
    uchar elf[12];
    ushort type;
    ushort machine;
    uint version;
    uint entry;         // program entry point (in va)
    uint phoff;        // offset of the program header tables
    uint shoff;
    uint flags;
    ushort ehsize;
    ushort phentsize;
    ushort phnum;      // number of program header tables
    ushort shentsize;
    ushort shnum;
    ushort shstrndx;
};
```

x86 boot sequence – loading ELF kernel

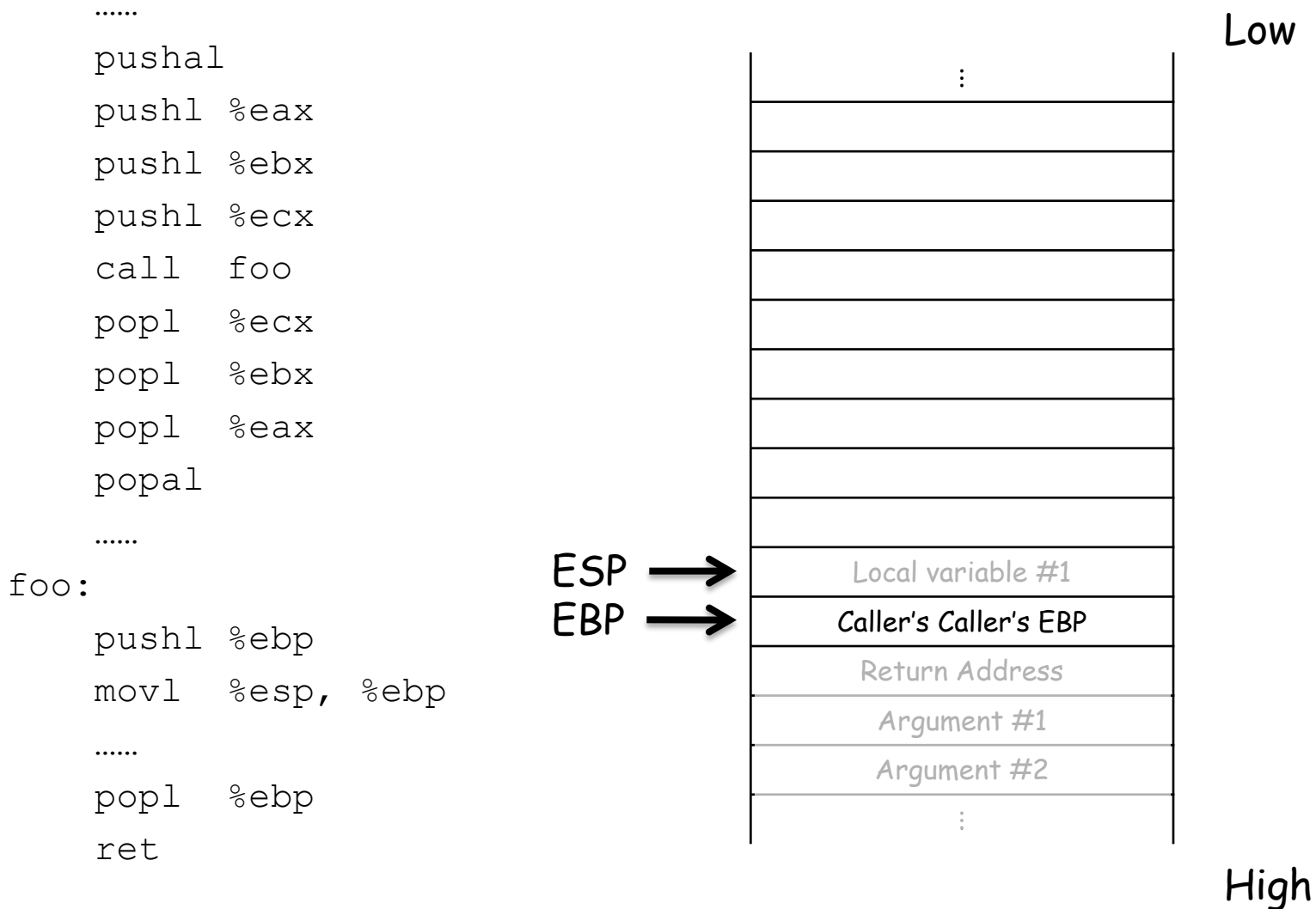
```
struct proghdr {
    uint type;           // segment type
    uint offset;        // beginning of the segment in the file
    uint va;            // where this segment should be placed at
    uint pa;
    uint filesz;
    uint memsz;         // size of the segment in byte
    uint flags;
    uint align;
};
```

- ◆ Chap. 2.5 (Control Registers)), Vol. 3, Intel® and IA-32 Architectures Software Developer's Manual
- ◆ Chap. 3 (Protected-Mode Memory Management), Vol. 3, Intel® and IA-32 Architectures Software Developer's Manual
- ◆ Chap. 9.1 (Initialization Overview), Vol. 3, Intel® and IA-32 Architectures Software Developer's Manual
- ◆ An introduction to ELF format: <http://wiki.osdev.org/ELF>

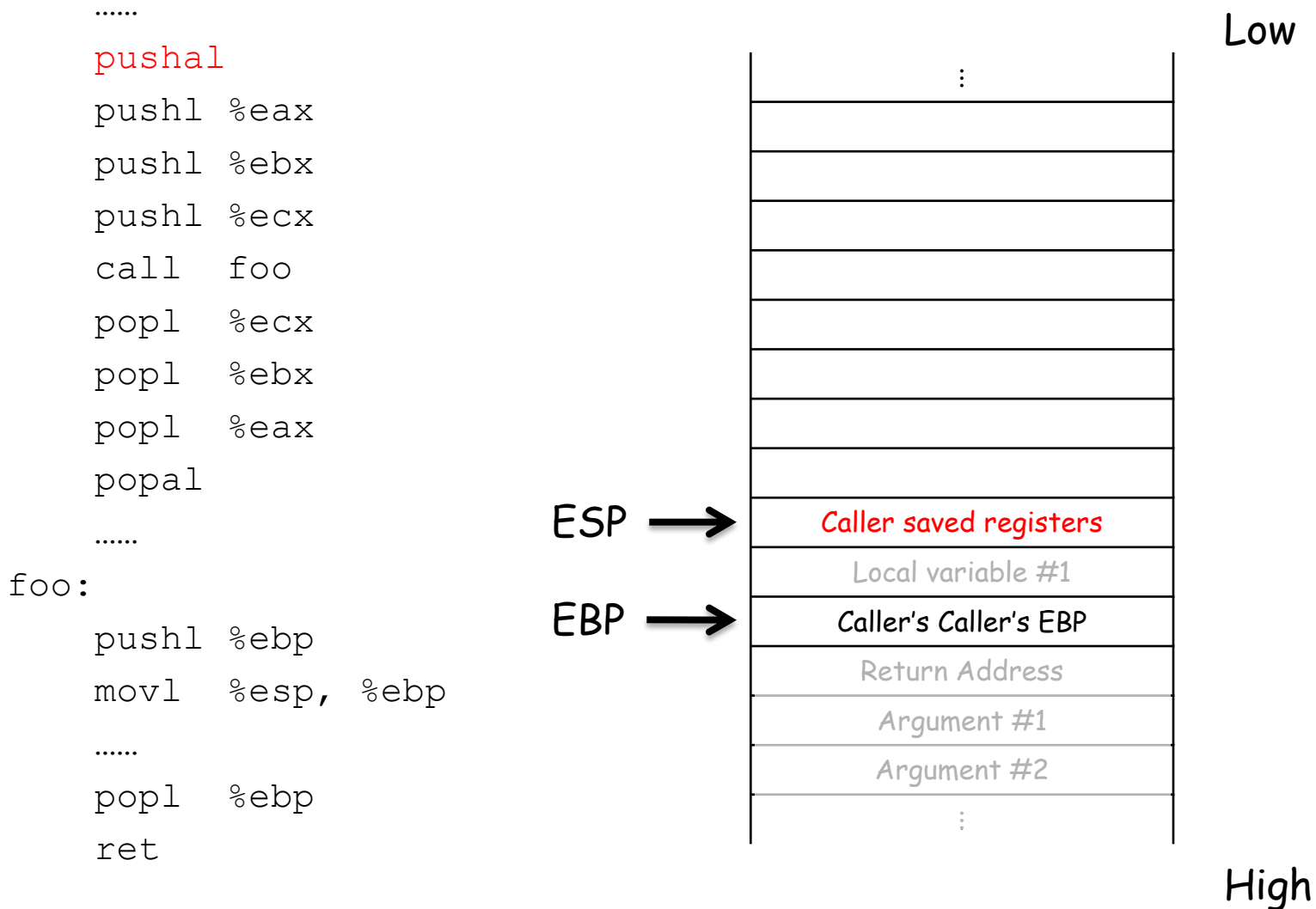
- Understand how C function calls are implemented at assembly level
- Know how to call C functions in assembly sources
- Be able to iterate stack frames using EBP

C FUNCTION CALLS IMPLEMENTATION

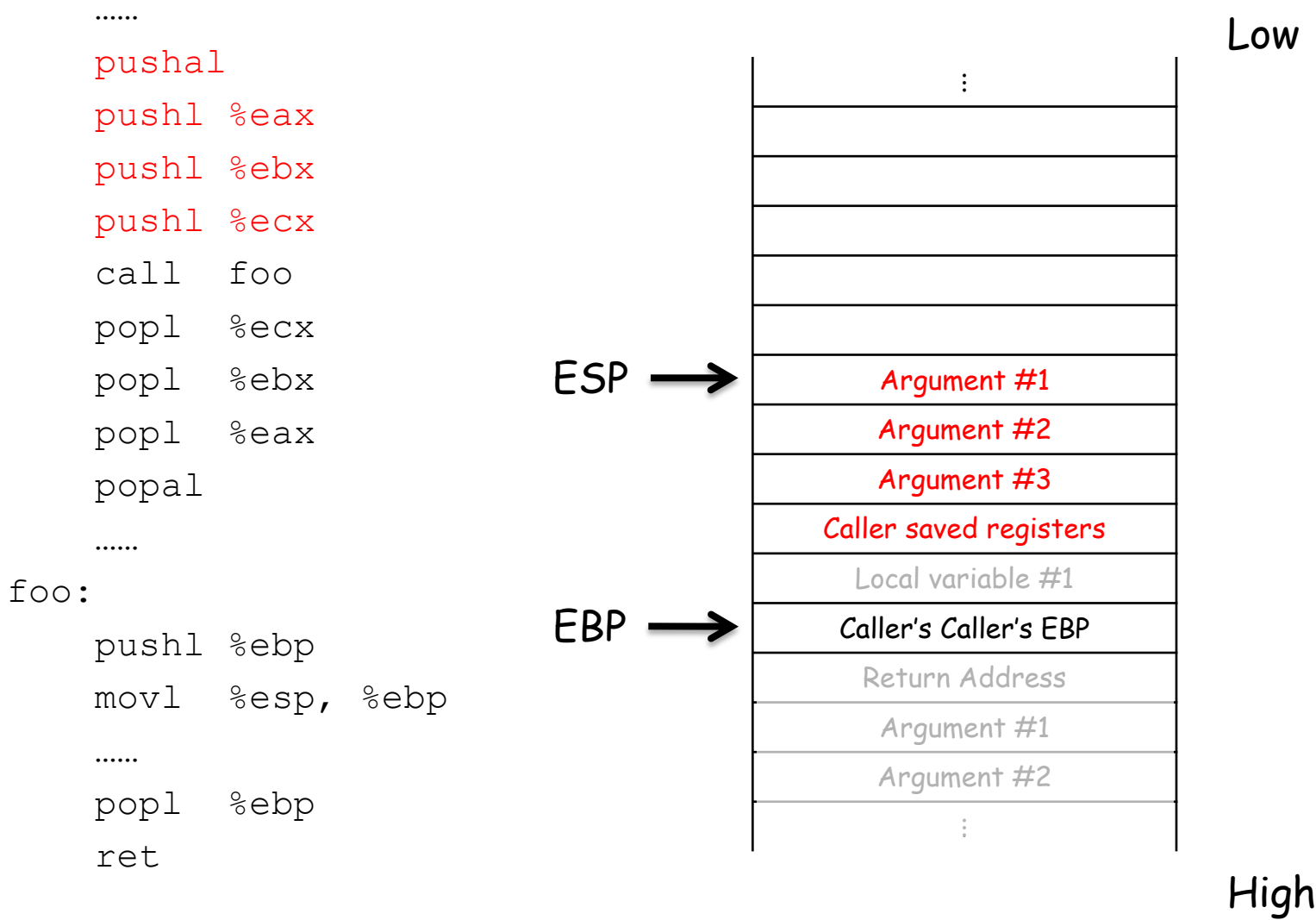
C function calls implementation



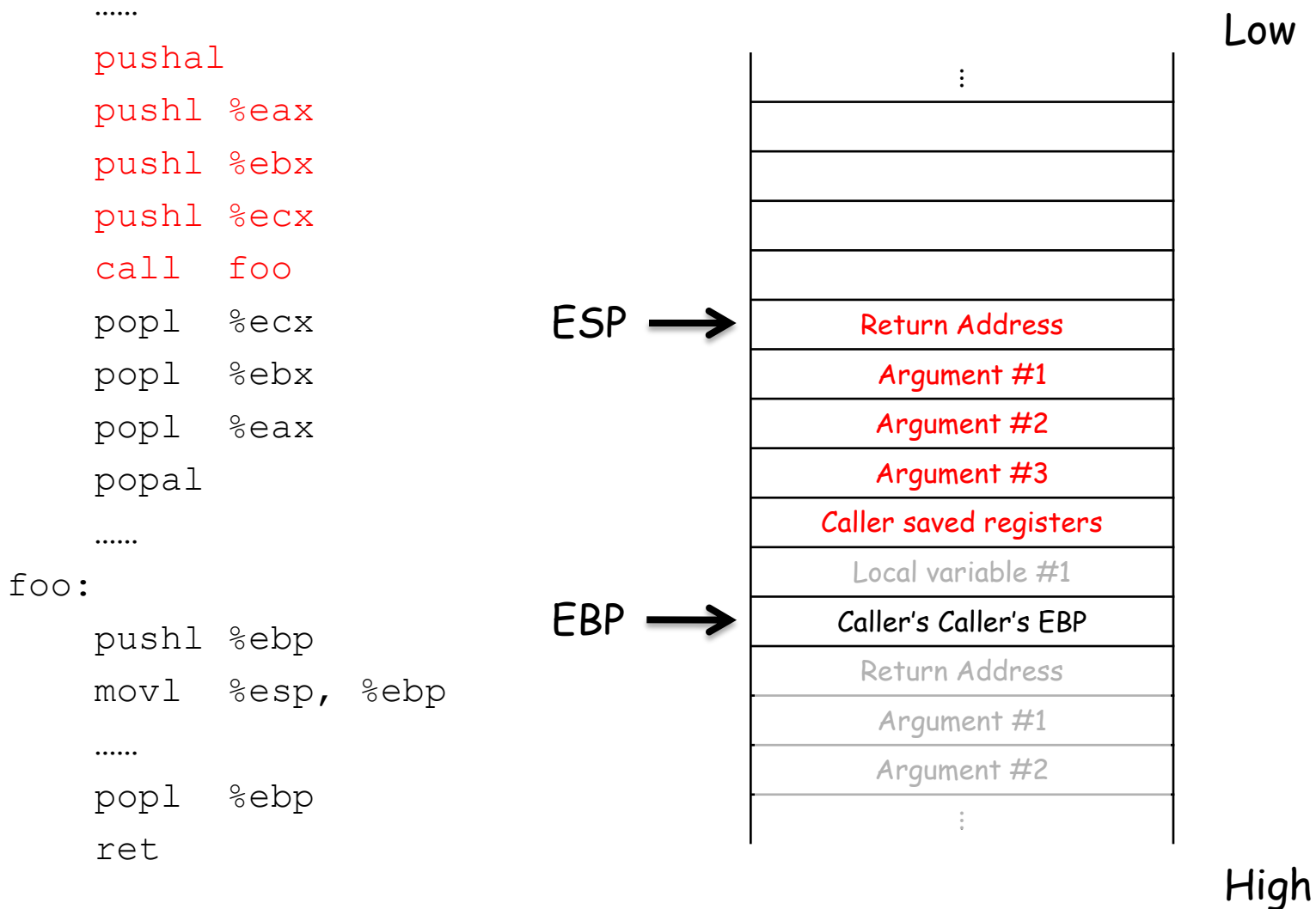
C function calls implementation



C function calls implementation



C function calls implementation

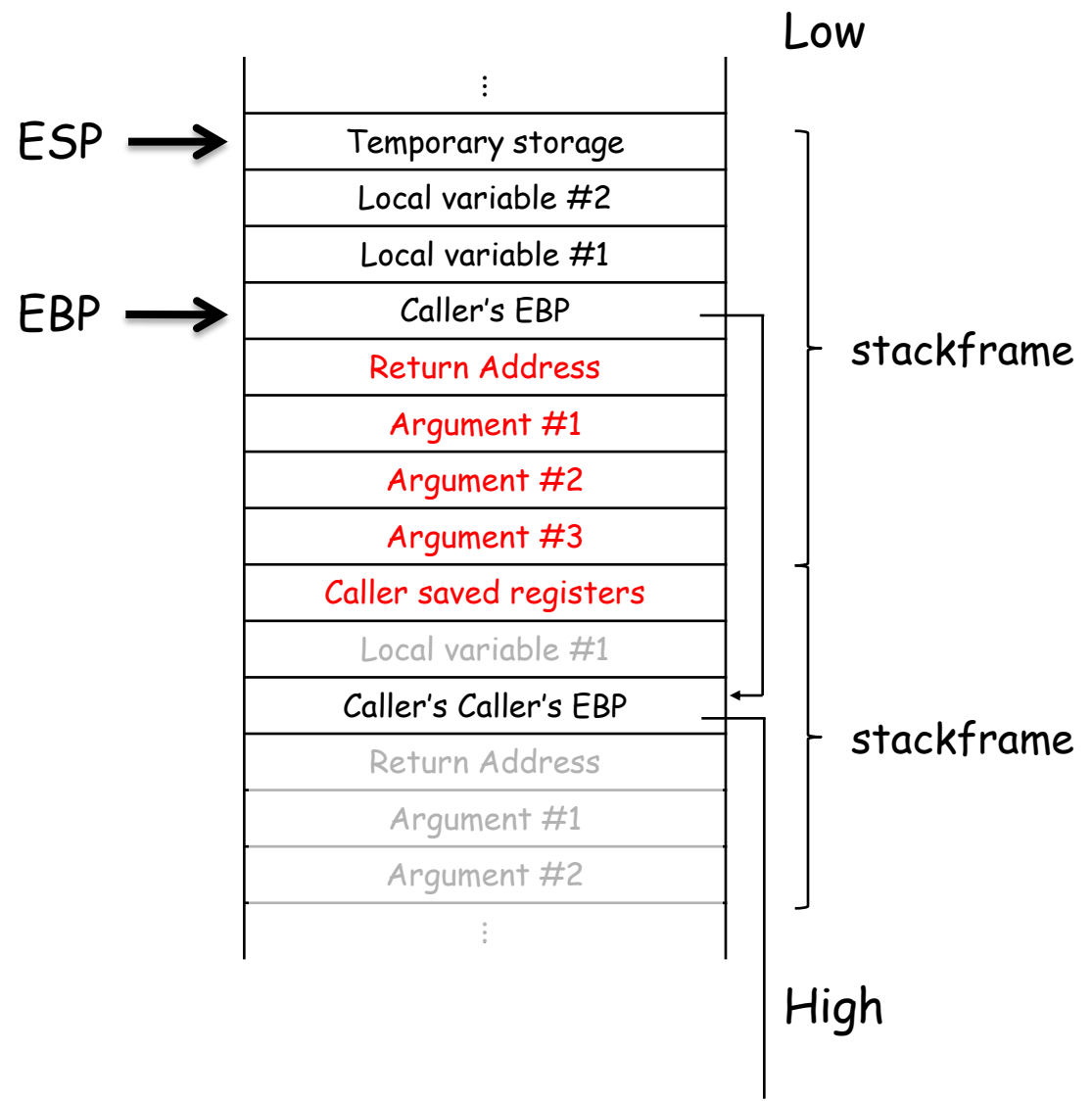


C function calls implementation

```

.....
pushal
pushl %eax
pushl %ebx
pushl %ecx
call foo
popl %ecx
popl %ebx
popl %eax
popal
.....
foo:
pushl %ebp
movl %esp, %ebp
.....
popl %ebp
ret

```

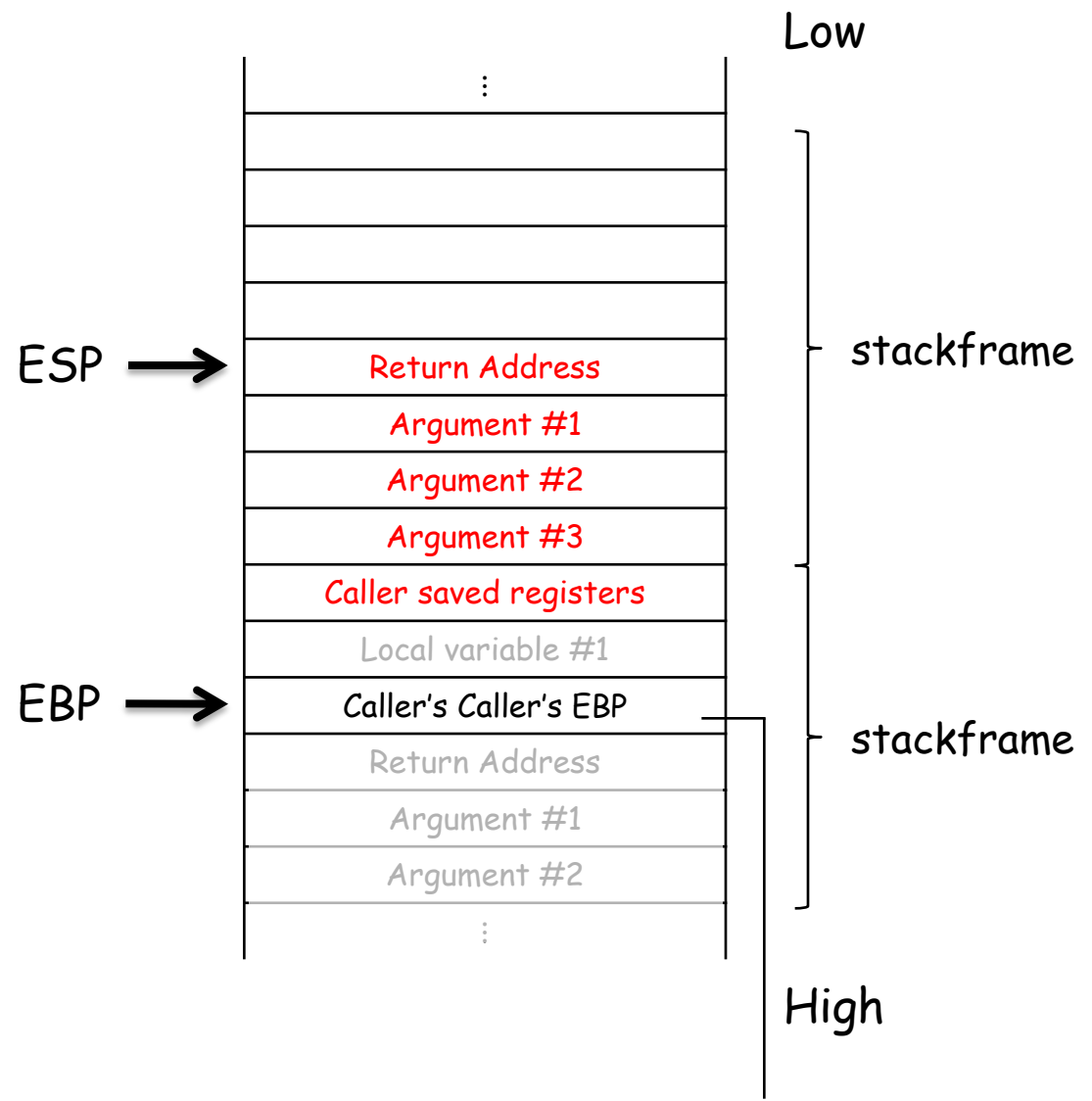


C function calls implementation

```

.....
pushal
pushl %eax
pushl %ebx
pushl %ecx
call  foo
popl  %ecx
popl  %ebx
popl  %eax
popal
.....
foo:
pushl %ebp
movl  %esp, %ebp
.....
popl  %ebp
ret

```

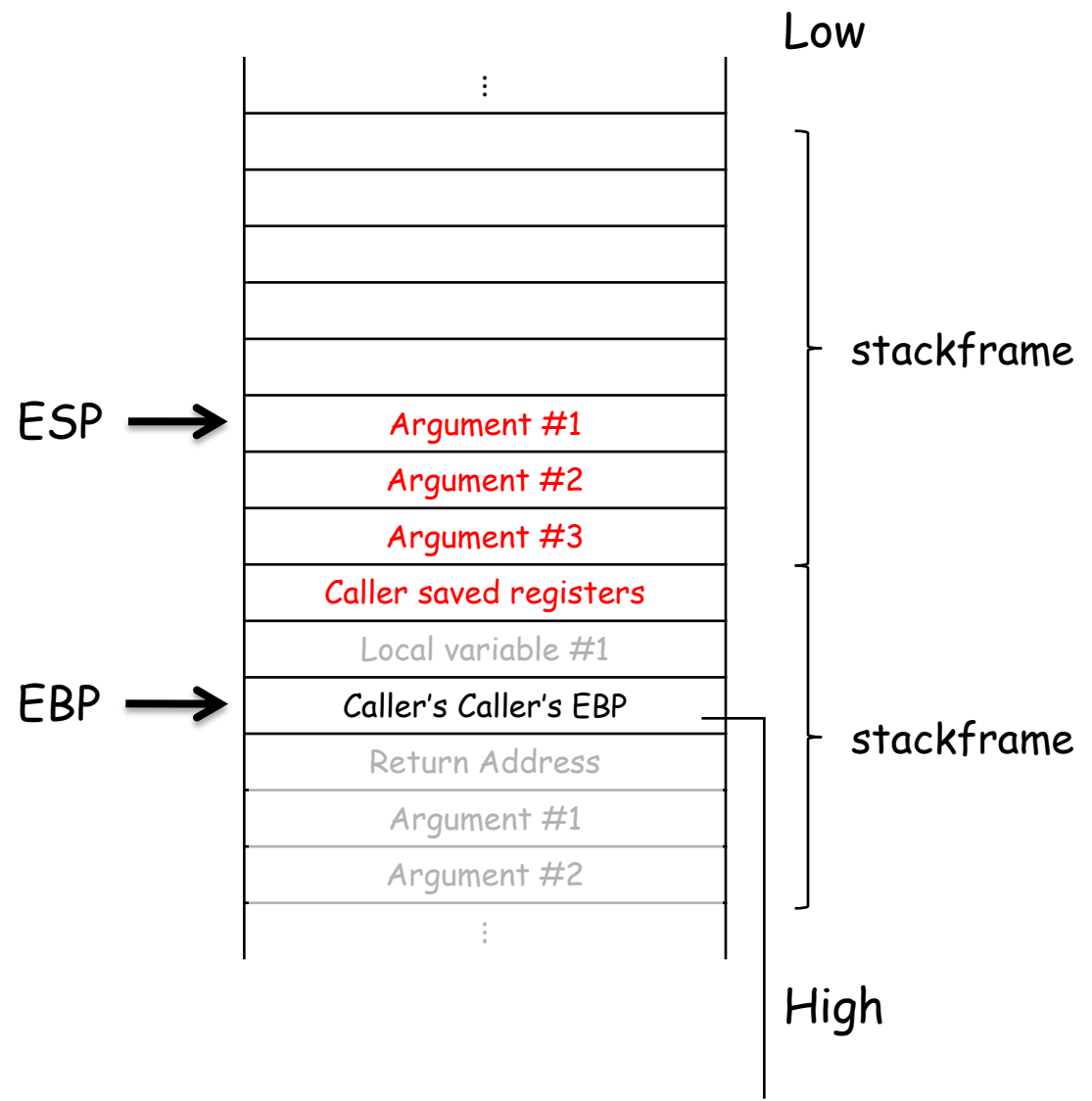


C function calls implementation

```

.....
pushal
pushl %eax
pushl %ebx
pushl %ecx
call  foo
popl  %ecx
popl  %ebx
popl  %eax
popal
.....
foo:
pushl %ebp
movl  %esp, %ebp
.....
popl  %ebp
ret

```



C function calls implementation

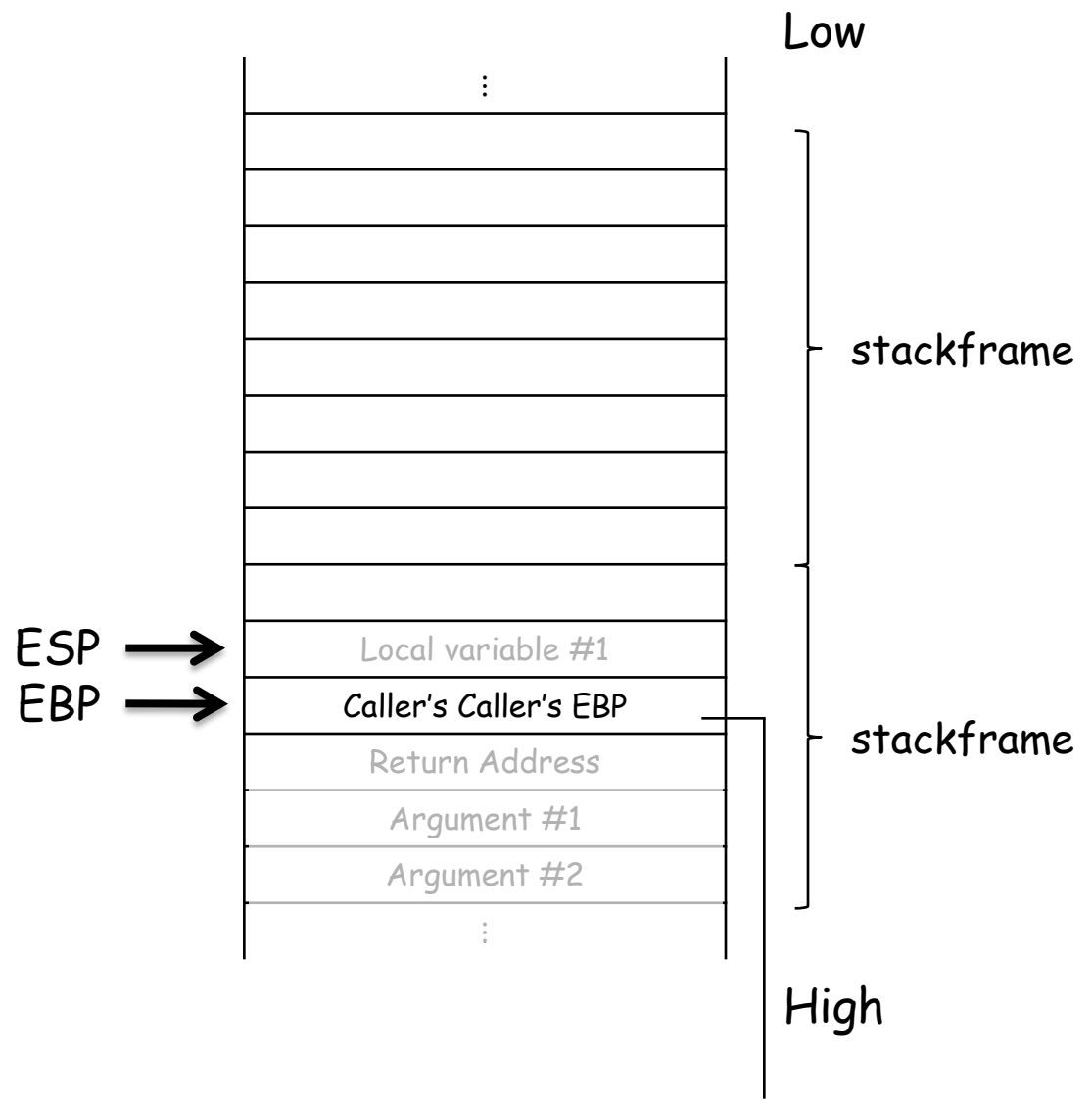
```

.....
pushal
pushl %eax
pushl %ebx
pushl %ecx
call  foo
popl  %ecx
popl  %ebx
popl  %eax
popal

.....
foo:
pushl %ebp
movl  %esp, %ebp

.....
popl  %ebp
ret

```



C function calls implementation

◆ More notes

- Parameters & return values can either be passed by registers or by stack
- No need to save/restore all registers all the time
- There may be a few hardware instructions available (e.g. **enter/leave** on x86)

C function calls implementation – References

- ◆ Understanding the Stack:
<http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html>

- Be able to read inline assembly instructions

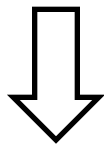
GCC INLINE ASSEMBLY

- ◆ What is inline assembly?
 - An GCC extension to C language
 - Inserting assembly instructions among C statements
- ◆ What it is for?
 - Invoking instructions not supported by C
 - Hand-optimizing hot spots
- ◆ How it works?
 - Generate assembly instructions based on the given template & constraints
 - Insert instructions generated into the target assembly source

GCC inline assembly – Example 1

Assembly (*.S):

```
movl $0xffff, %eax
```



Inline assembly (*.c):

```
asm ("movl $0xffff, %%eax\n")
```

asm (assembler template

: output operands (optional)

: input operands (optional)

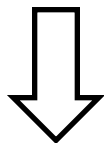
: clobbers (optional, you may skip this for now)

);

GCC inline assembly – Example 2

Inline assembly (*.c):

```
uint32_t cr0;  
asm volatile ("movl %%cr0, %0\n" : "=r" (cr0));  
cr0 |= 0x80000000;  
asm volatile ("movl %0, %%cr0\n" :: "r" (cr0));
```



Generated assembly code (*.s):

```
movl %cr0, %ebx  
movl %ebx, 12(%esp)  
orl $-2147483648, 12(%esp)  
movl 12(%esp), %eax  
movl %eax, %cr0
```

GCC inline assembly – Example 2

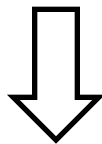
Inline assembly (*.c):

```
uint32_t cr0;  
asm volatile ("movl %%cr0, %0\n" : "=r" (cr0));  
cr0 |= 0x80000000;  
asm volatile ("movl %0, %%cr0\n" :: "r" (cr0));
```

- **volatile**
No reordering; No elimination
- **%0**
The first constraint following
- **r**
A constraint; GCC is free to use any register

GCC inline assembly – Example 3

```
long __res, arg1 = 2, arg2 = 22, arg3 = 222, arg4 = 233;
__asm__ volatile ("int $0x80"
                  : "=a" (__res)
                  : "0" (11), "b" (arg1), "c" (arg2), "d" (arg3), "S" (arg4));
```



```
movl    $11, %eax
movl    8(%ebp), %ebx
movl    12(%ebp), %ecx
movl    16(%ebp), %edx
movl    20(%ebp), %esi
movl    %eax, %edi
movl    %edi, %eax
int     $0x80
movl    %eax, %edi
movl    %edi, -16(%ebp)
```

- **Constraints**

a = %eax

b = %ebx

c = %ecx

d = %edx

S = %esi

D = %edi

0 = same as the first

GCC inline assembly - References

- ◆ GCC Manual 6.41 – 6.43
- ◆ Inline assembly for x86 in Linux:
<http://www.ibm.com/developerworks/library/l-ia/index.html>

- Know all kinds of interrupt sources in x86
- Know how CPU handles an interrupt
- Be able to fill IDT

INTERRUPT HANDLING IN X86 ARCHITECTURE

Interrupt handling in x86 architecture – Interrupt sources

◆ Interrupts

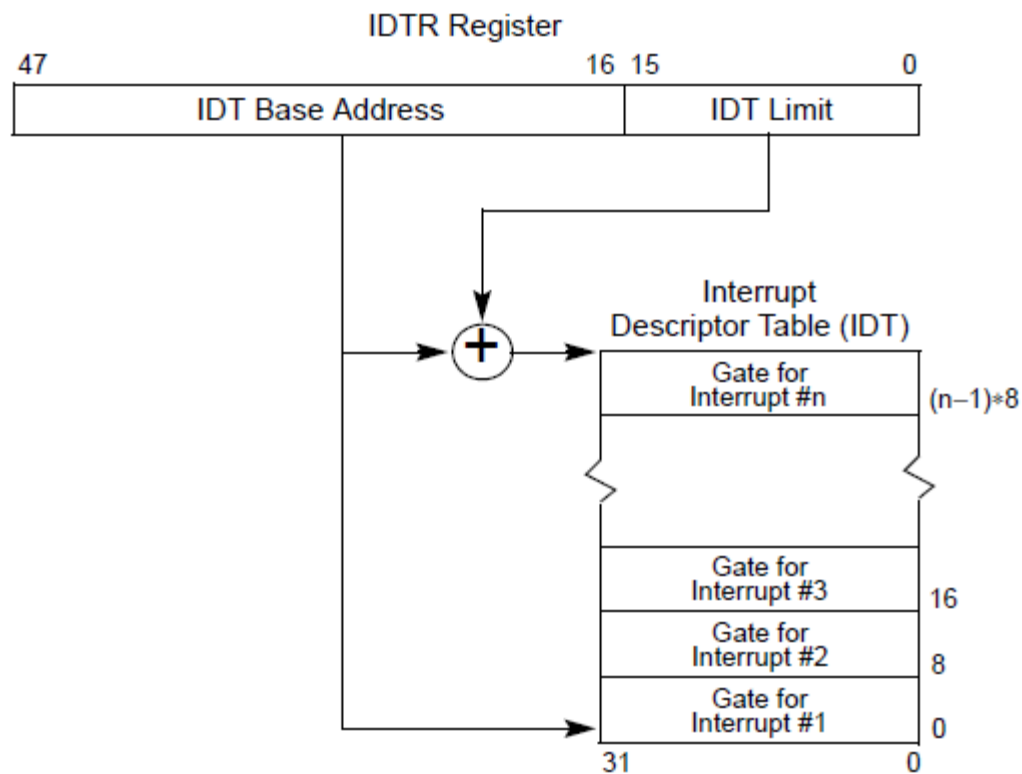
- External (hardware generated) interrupts
From serial port, disk, GPU or other devices.
- Software generated interrupts
The **INT** *n* instruction

◆ Exceptions

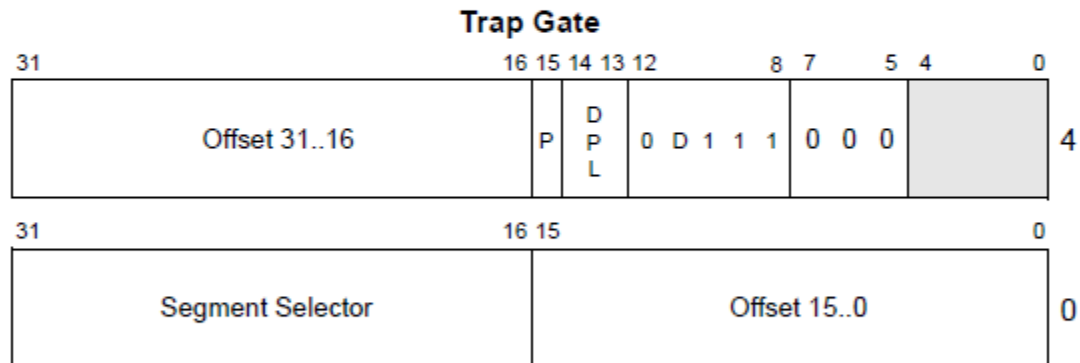
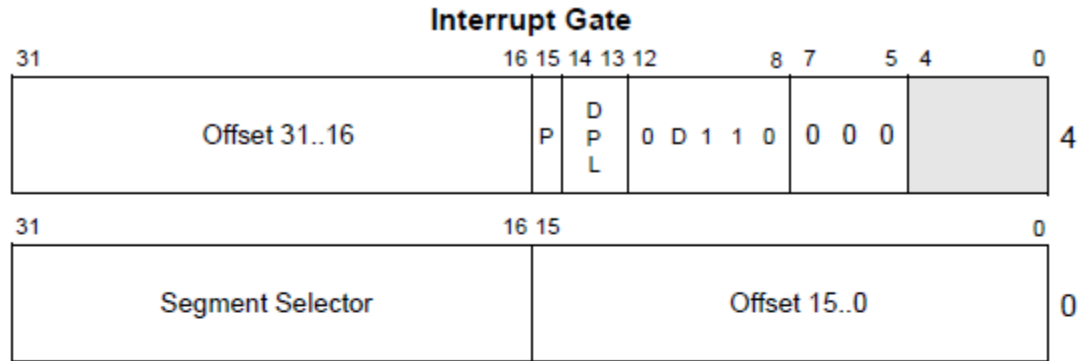
- Program error
- Software generated exceptions
INTO, **INT 3** and **BOUND**
- Machine check exceptions

Interrupt handling in x86 architecture – Locating ISRs

- ◆ Each interrupt or exception is associated with an Interrupt Service Routine (ISR) by the Interrupt Descriptor Table (IDT).
- ◆ The address and size of the IDT is stored in IDTR



Interrupt handling in x86 architecture – Locating ISRs



- DPL Descriptor Privilege Level
- Offset Offset to procedure entry point
- P Segment Present flag
- Selector Segment Selector for destination code segment
- D Size of gate: 1 = 32 bits; 0 = 16 bits
- Reserved

Interrupt handling in x86 architecture – Locating ISRs

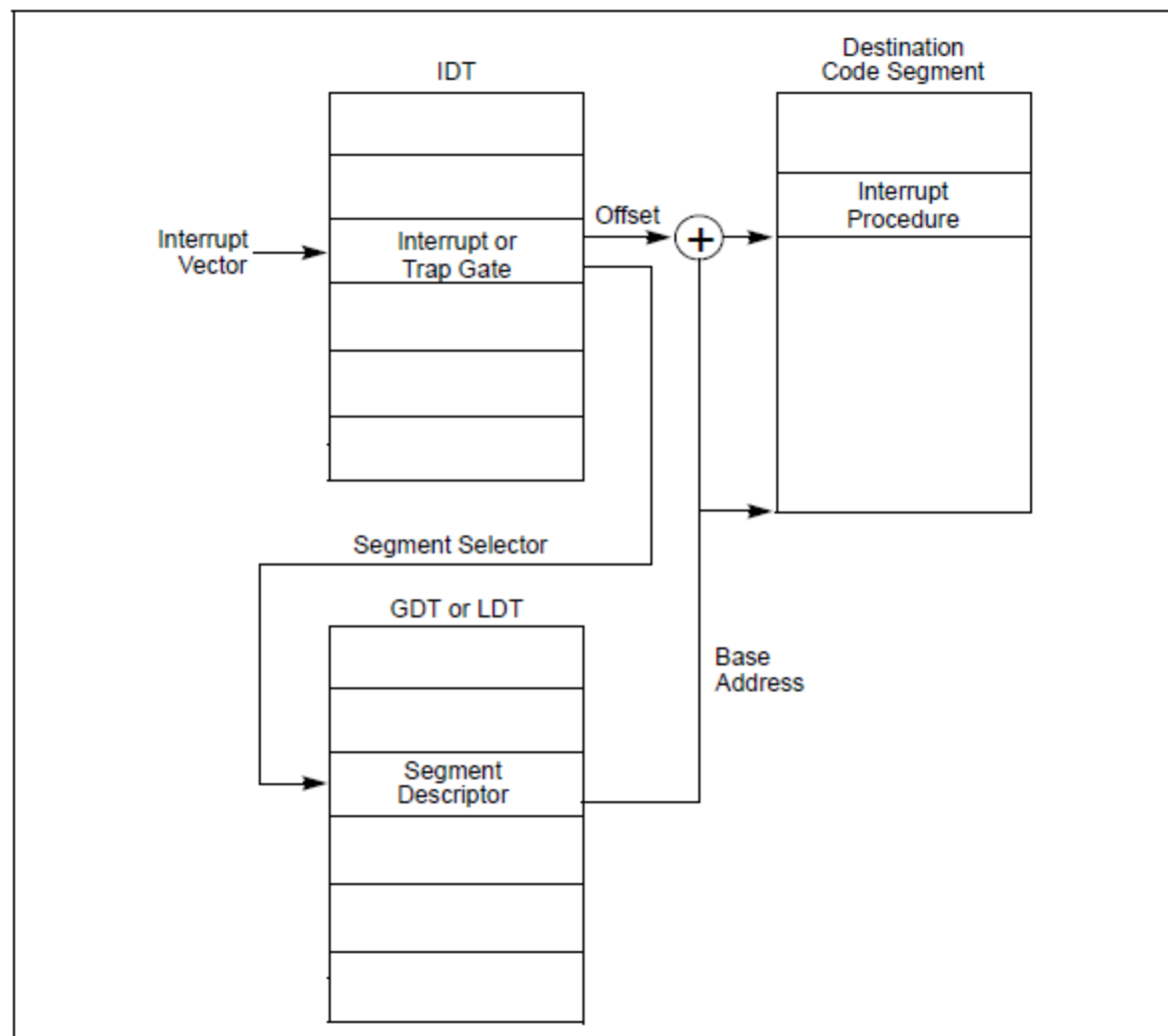
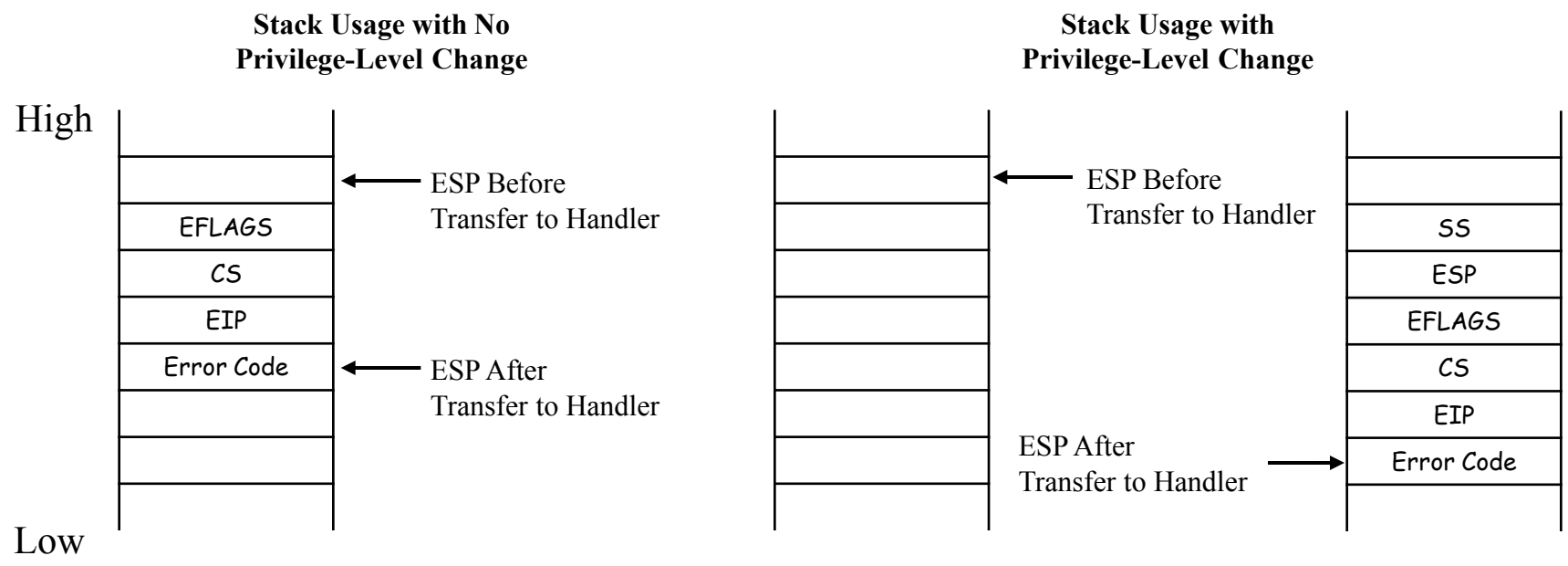


Figure 6-3. Interrupt Procedure Call

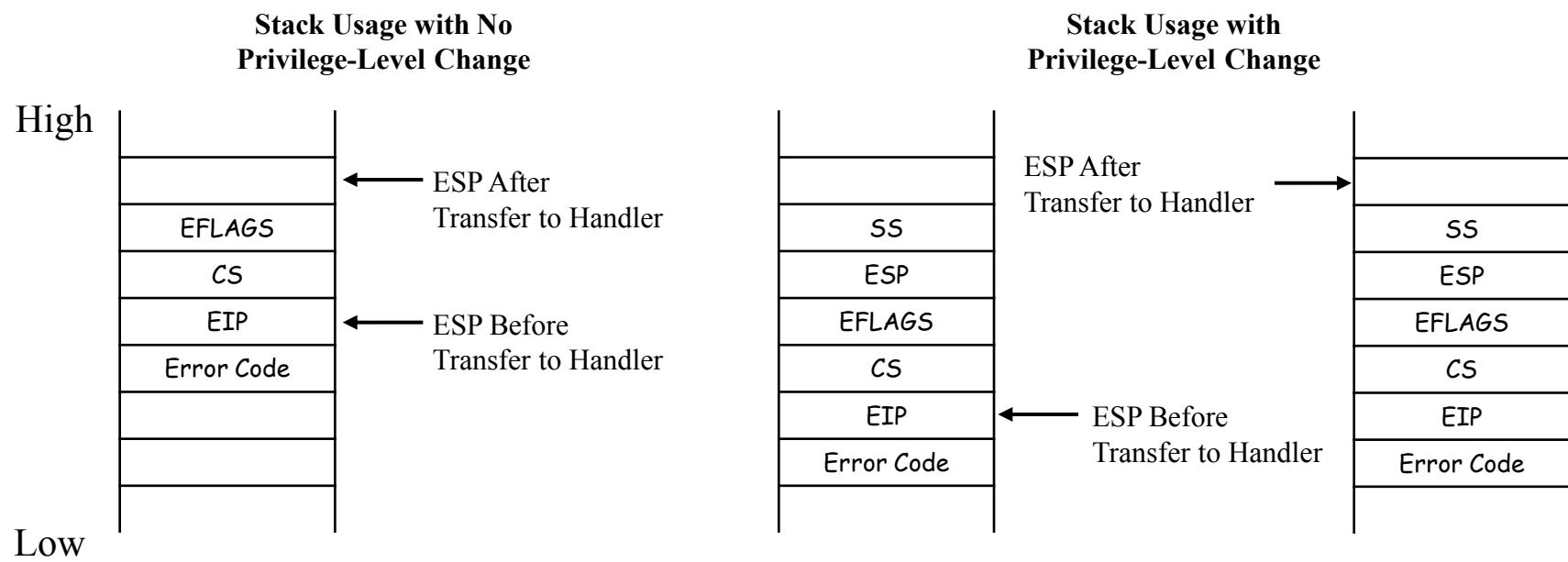
Interrupt handling in x86 architecture – Switch to ISR

- ◆ Interrupt vs. trap: IF in EFLAGS is cleared for interrupt gates but not for trap gates



Interrupt handling in x86 architecture – Return from ISR

- ◆ **ret vs. iret:** iret pops EFLAGS and SS/ESP if needed, while ret doesn't



Interrupt handling in x86 architecture – System calls

- ◆ User land applications ask for kernel services via system calls.
- ◆ How to implement
 - Software generated interrupt with predefined interrupt number
 - Special instructions (**SYSENTER/SYSEXIT**)

Interrupt handling in x86 architecture – References

- ◆ Chap. 6, Vol. 3, Intel® and IA-32 Architectures Software Developer's Manual

Last for all: About lab2

- ◆ We'll discuss lab2 this Thursday morning
 - x86 privilege levels
 - x86 hardware MMU
- ◆ Please preview the code before class

That's all. Thanks!