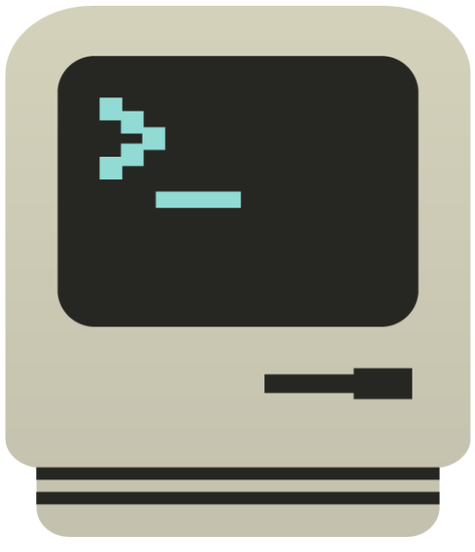
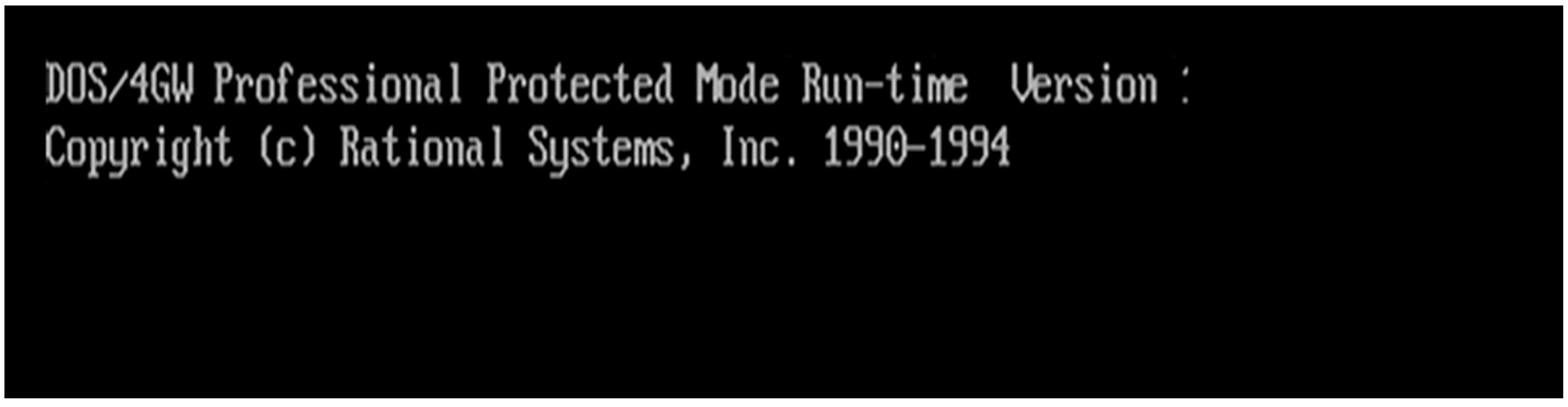

Labs
Intel Paging&Interrupts

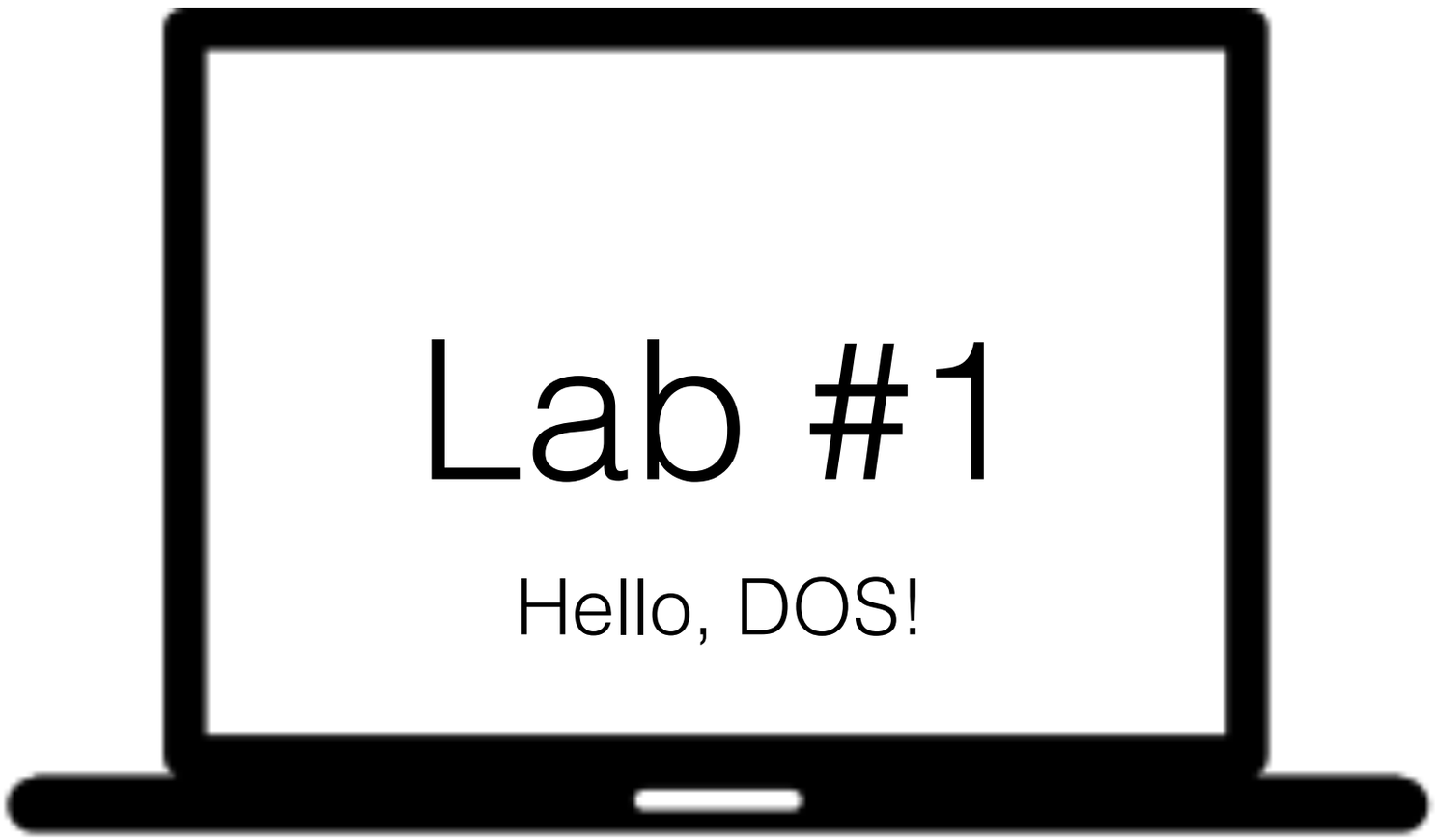
Environment



MS DOS Operating System
1 Mb Address Space
Real Mode (**No protection:**
Programs can modify System code)

DOS Extender Program for DOS
allows to use 4 GB Address Space
and Protected Mode




Lab #1
Hello, DOS!

Lab #1 «Hello, DOS!»

Description: Print «Hello, DOS!» to console

Operating system provides an access to hardware including IO access.
We use operating system (via **syscall** in modern OSes) to write into standard output (to console).
In **DOS** we can call system functions using interrupt.

Background:

- Registers
- Interrupts

Registers

General-Purpose Registers

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

Figure 3-5. Alternate General-Purpose Register Names

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Interrupts

Interrupt instruction:

```
int N // N is interrupt number
```

MS DOS: 21h (33 in decimal) is interrupt to call system function

```
int 21h
```

Function number is placed to **ah** register

Other registers content depends on function

Write string

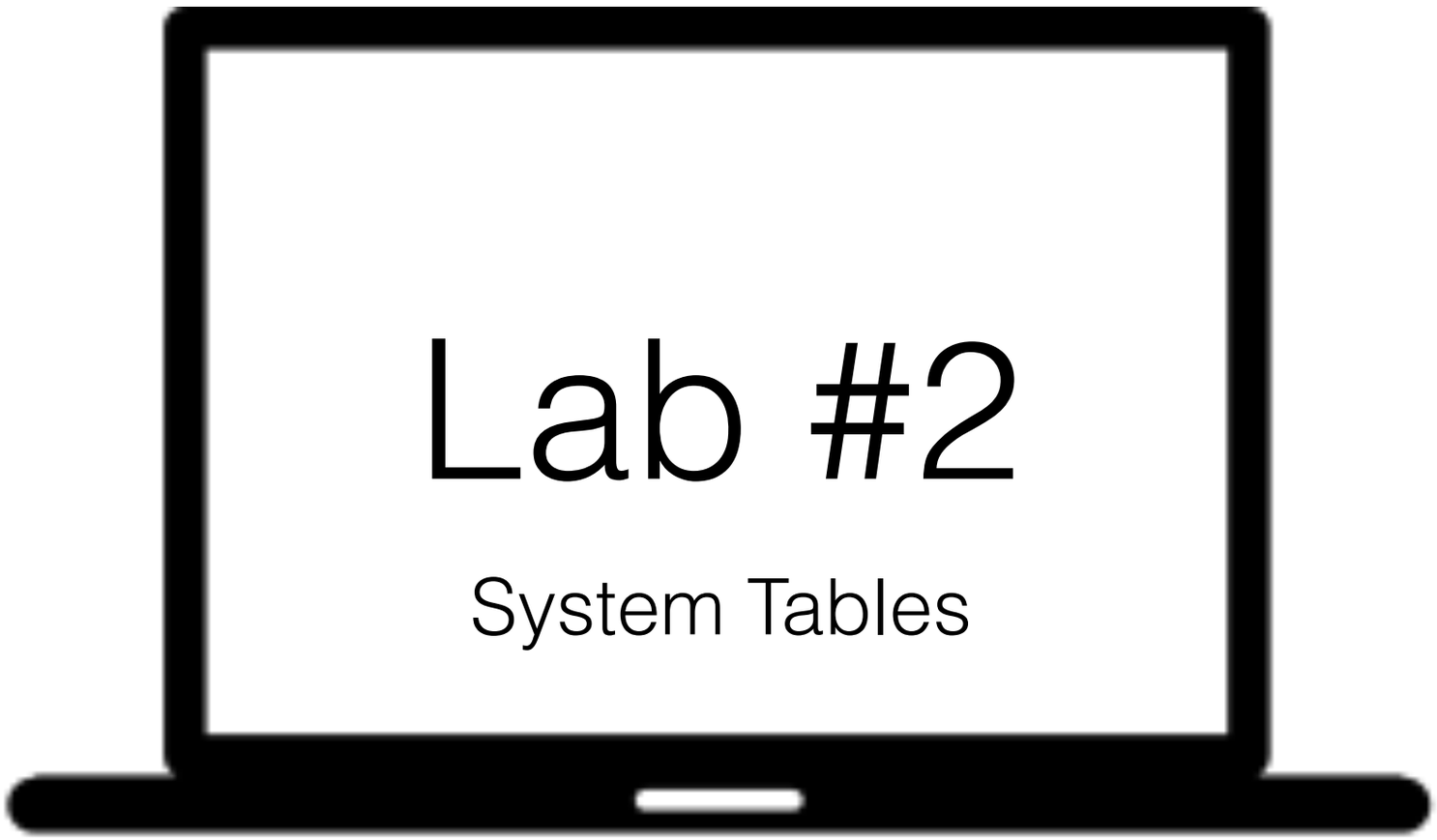
MS DOS:

AH = 09h - WRITE STRING TO STANDARD OUTPUT

Entry: DS:DX -> '$'-terminated string

1. Put `09h` to **`ah`** using **`mov`** instruction
2. Put string address into **`edx`** (DOS extender cares about DS:DX)
3. Use `int 21h` to call system function `09h`

```c
#include "stdio.h"

int main(void) {
    char *DosMsg = "Hello, DOS! \n\r$"
    /* Our string ends with $ symbol */

    /* This is assembler code block */
    __asm {
        mov ah, 09h // to write string
        mov edx, DosMsg // put str address to edx
        int 21h // interrupt 21h for system call
    }

    return 0;
}
```

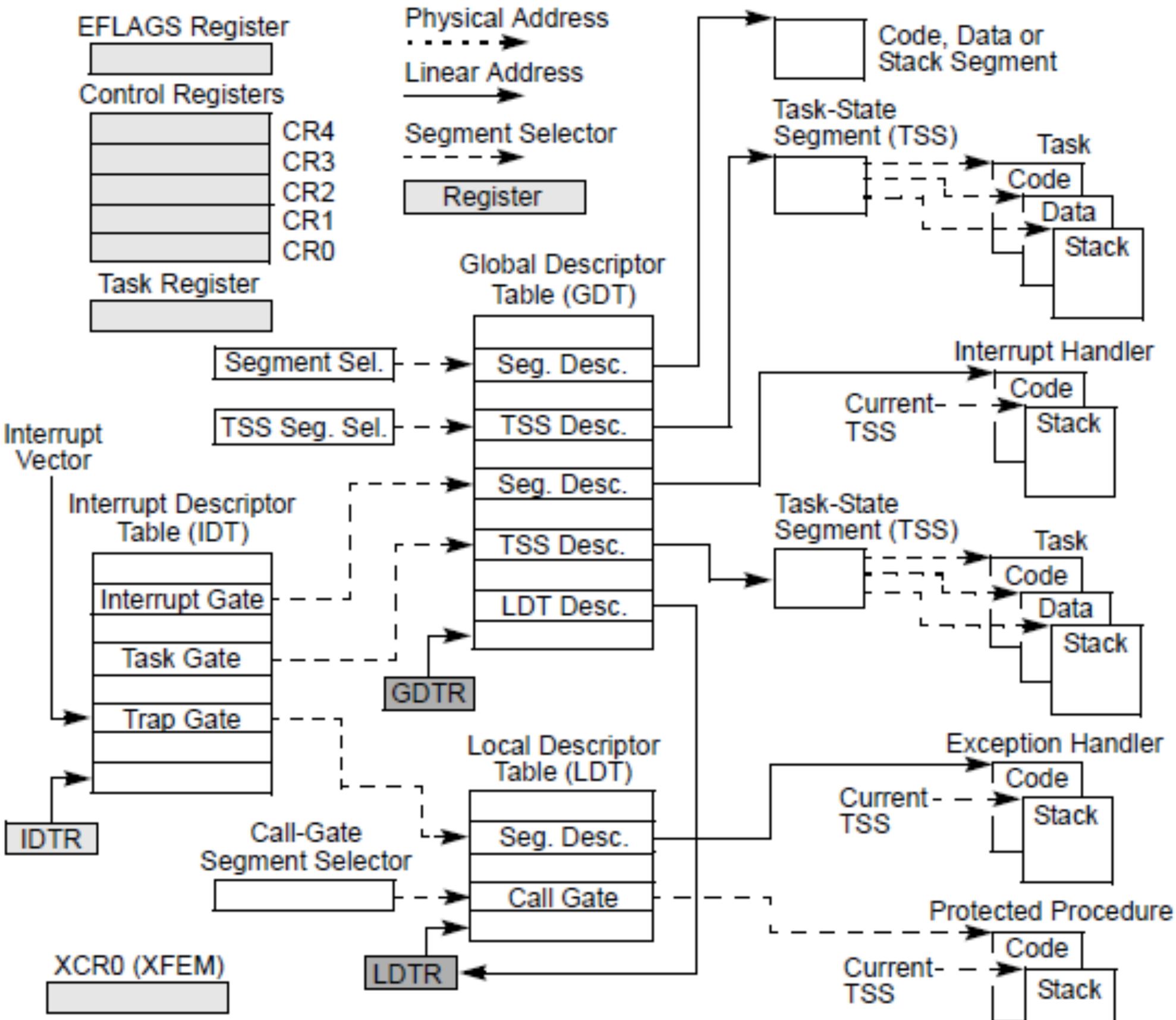

Lab #2
System Tables

Lab #2 «System Tables»

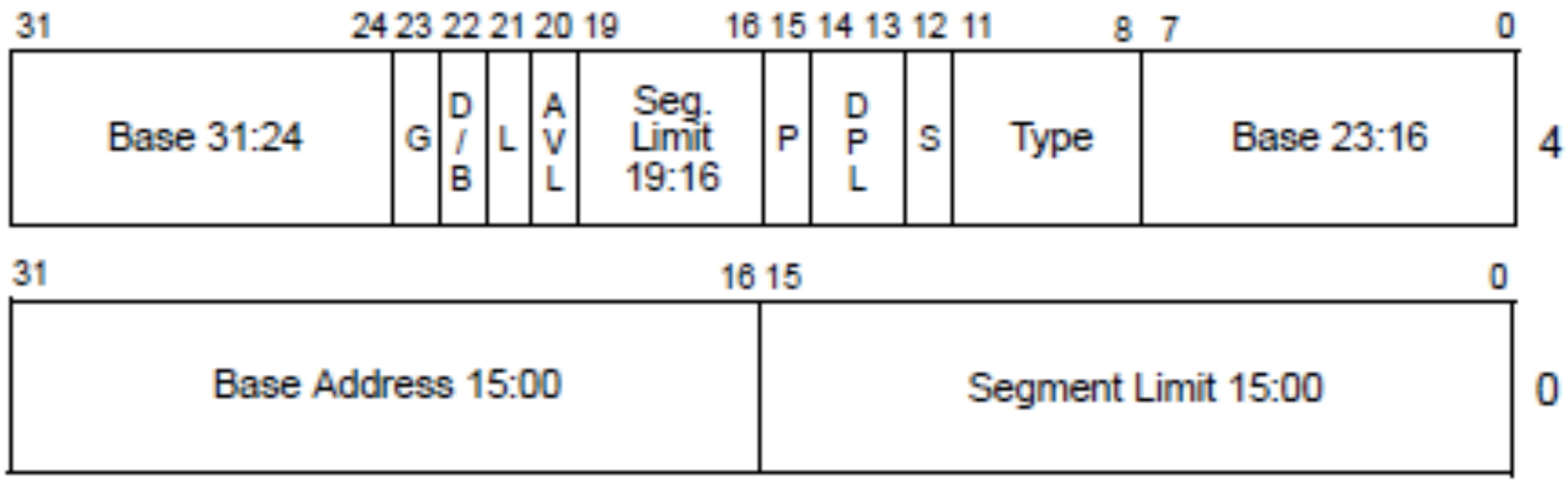
Description: Print Global Description Table (GDT) entries and Interrupt Descriptor Table (IDT) entries

Background:

- GDT stores system descriptors (for segments and system tables)
- IDT stores interrupt gates (including handler pointers)



Intel® 64 and IA-32 Architectures Software Developer's Manual



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 3-8. Segment Descriptor

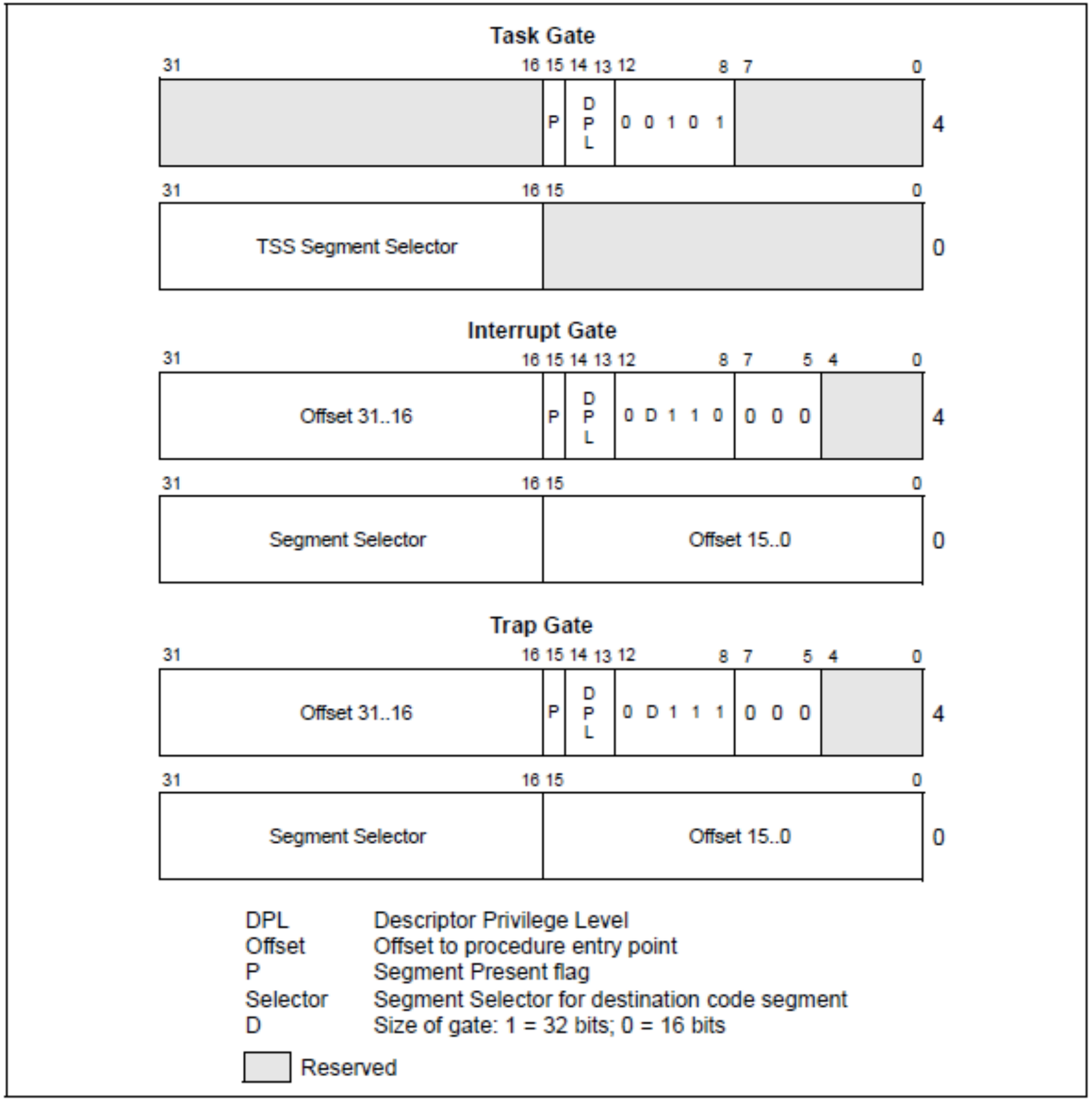


Figure 6-2. IDT Gate Descriptors

```
/* Descriptor table register */
typedef struct _DTR {
    union {
        struct {
            u32_t dw0;
            u32_t dw1;
        } raw;
        struct {
            u16_t Limit; // size of table in bytes
            u32_t Base; // address of table
            u16_t Padding;
        };
    };
} DTR, *PDTR;

DTR _gdtr, _idtr;
```

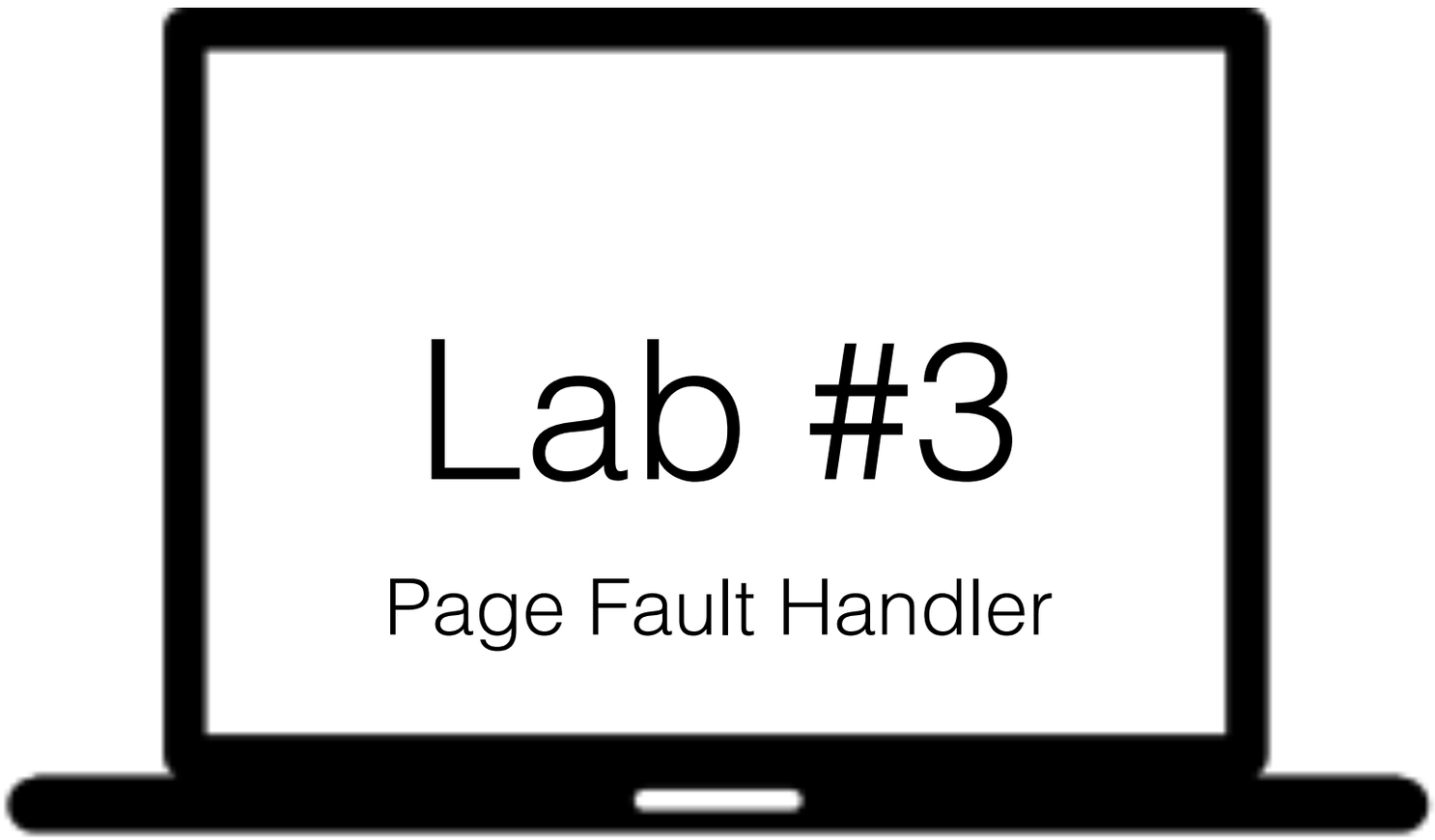
Table 2-3. Summary of System Instructions

Instruction	Description	Useful to Application?
LLDT	Load LDT Register	No
SLDT	Store LDT Register	No
LGDT	Load GDT Register	No
SGDT	Store GDT Register	No
LTR	Load Task Register	No
STR	Store Task Register	No
LIDT	Load IDT Register	No
SIDT	Store IDT Register	No
MOV CR*n*	Load and store control registers	No

```
// IDT entry
typedef struct _IDT_ENTRY {
    u16_t offset_l;
    u16_t seg_sel;
    u8_t  zero;
    u8_t  flags;
    u16_t offset_h;
} IDT_ENTRY, *PIDT_ENTRY;

PIDT_ENTRY idt;
```

```
/* We set memory with zeros */
memset(&_idtr, 0, sizeof(_idtr));

__asm {
    sidt _idtr // read IDT register
}

idt = (PIDT_ENTRY)_idtr.Base; // pointer to IDT
```

```
typedef union _GDT_ENTRY {
    struct {
        uint32 low;
        uint32 high;
    } raw;
    struct {
        uint16 limit_low;
        uint16 base_low;
        uint8  base_mid;
        uint8  type:4;
        uint8  s:1; // S bit is 0 for segments
        uint8  dpl:2;
        uint8  p:1;
        uint8 limit_high:4;
        uint8 avl:1;
        uint8 rsrvd:1; // L bit (only in 64-bit)
        uint8 db:1;
        uint8 g:1;
       uint8 base_high;
    } desc; // Segment descriptor NB! GDT can contain other entities
} GDT_ENTRY, *PGDT_ENTRY;

/* We set memory with zeros */
memset(&_gdtr, 0, sizeof(_gdtr));

__asm {
    sgdt _gdtr // read GDT register
}

gdt = (PDESCRIPTOR)_gdtr.Base; // pointer to GDT
```


Lab #3
Page Fault Handler

Lab #3 «Page Fault Handler»

Description: Replace system page fault handler with your own

1. Enable paging
2. Create gap in mapping and generate BSOD
3. Create page fault interrupt handler to close the gap
4. Replace system PF handler

Background:

- Paging
- Interrupts

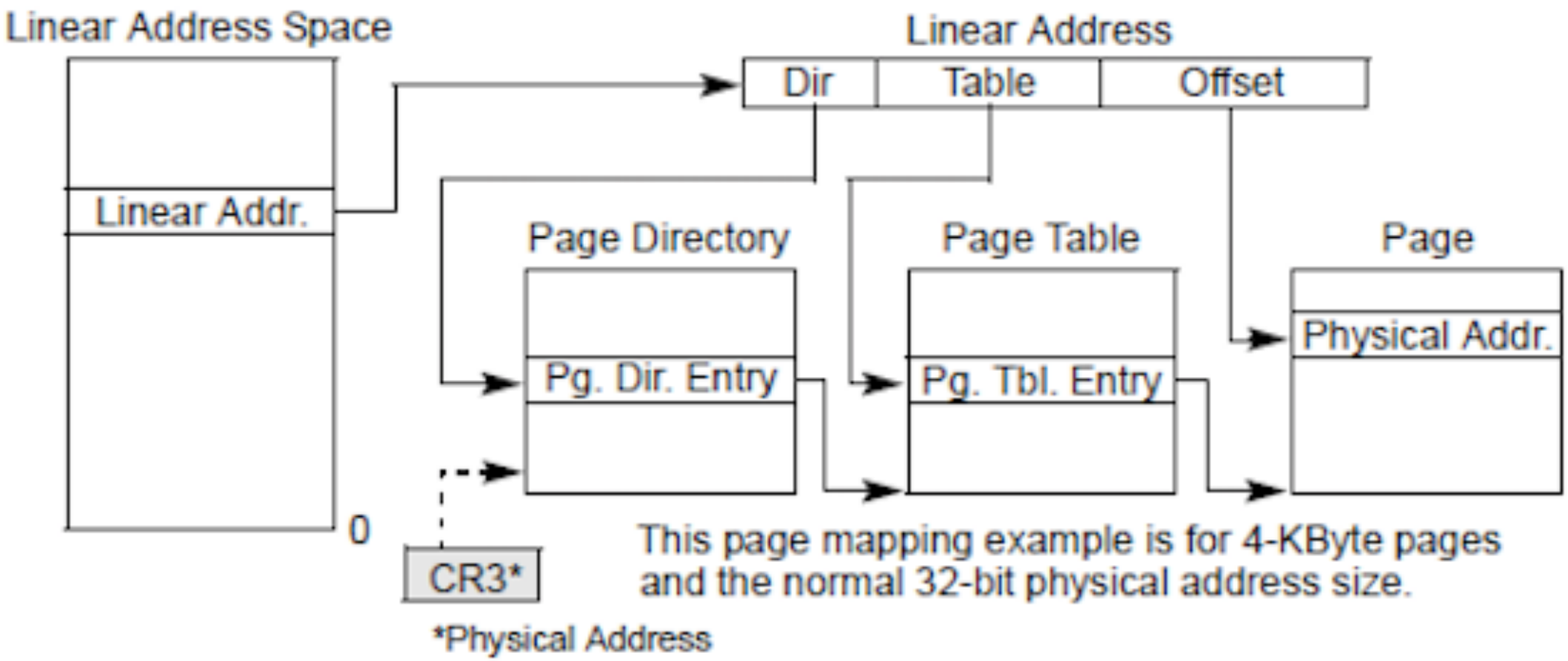


Figure 2-1. IA-32 System-Level Registers and Data Structures

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored				CR3
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)						Bits 39:32 of address[2]			PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

```
#define NP 512 // This page will be marked as not present
#define PAGE_SIZE 4096
#define PTE_SIZE 4
#define PTE_PER_PAGE (PAGE_SIZE/PTE_SIZE)

/* Page table we create should contain aligned pages */
u32_t *pt_aligned;

u32_t *np_page_ptr; // not preset page

void page_table_create() {
    char *p = (char *)malloc(8*1024*1024); // 8Mb

/* 4Mb aligned pointer got from allocated 8Mb region */
    pt_aligned = (u32_t *)(( ((u32_t)p) & 0xffc00000) + 0x400000);
```

```
    for (int i = 0; i < PTE_PER_PAGE; i++) {
// page directory entry
        u32_t pde = i*0x400000 // 4Mb page
                  + 0x87; // Present, RW, US, PS bits on
        pt_aligned[i] = pde;
        if (i == NP) {
            pt_aligned[i] = pt_aligned[i] & 0xFFFFFFFE;
            p_np_pde = &pt_aligned[i];
        }
    }

    printf("Page %d is not present now\n", NP);
}
```

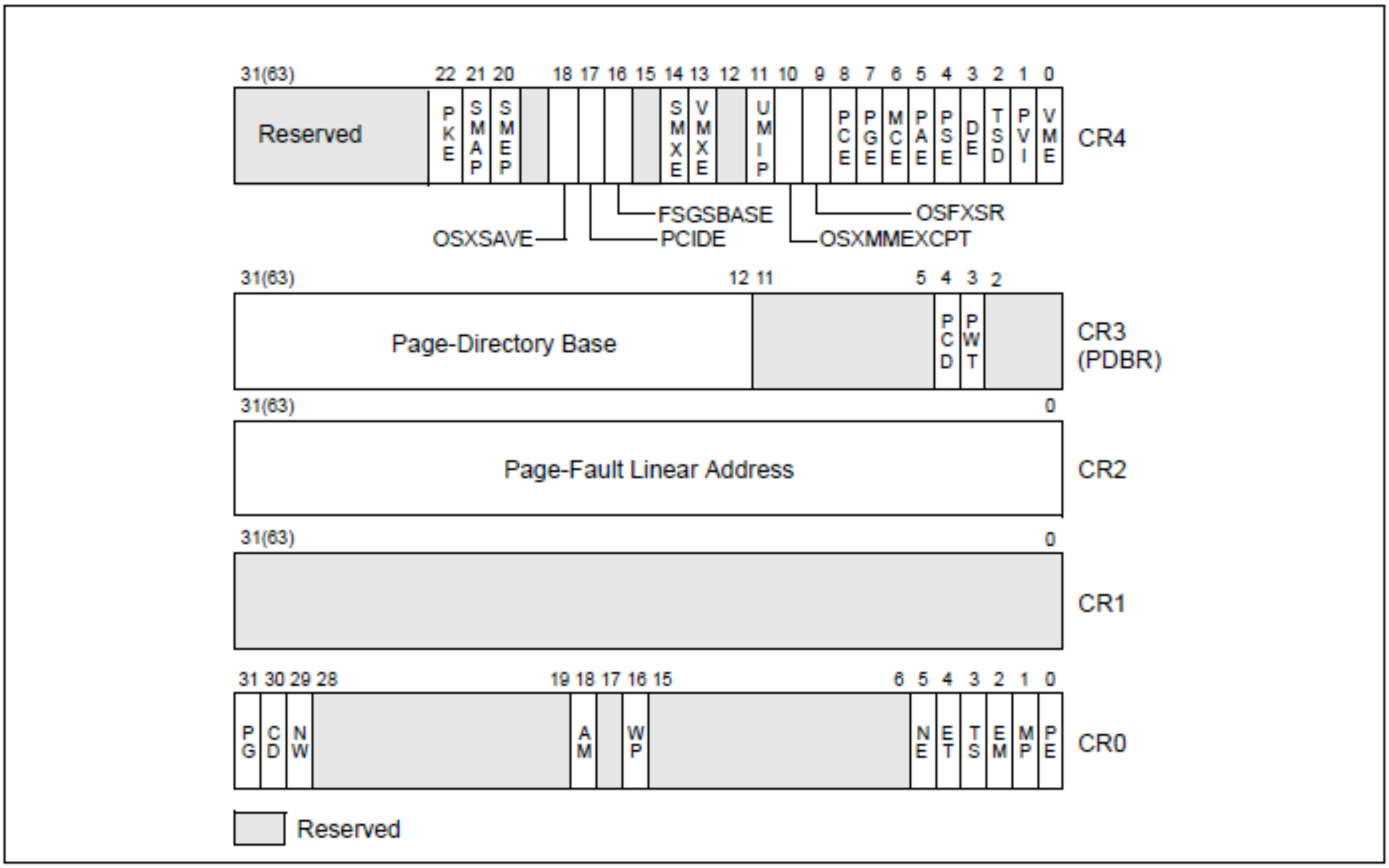
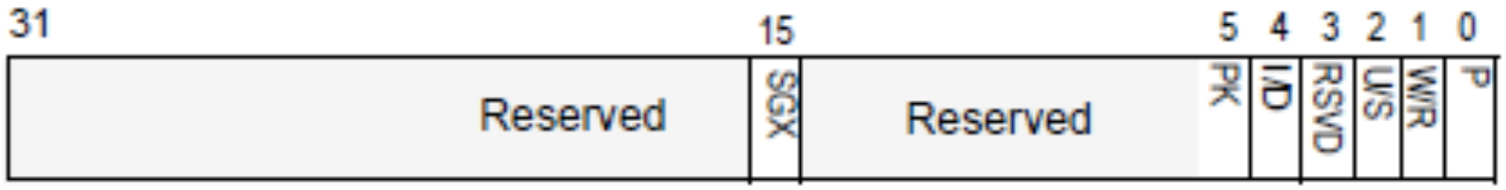


Figure 2-7. Control Registers

```
#define PF_ADDR 0x80000000 // page #512 (NP*4Mb)

__asm {
    cli // disable interrupts
    mov eax, pt_aligned
    mov cr3, eax // Put our page table address into CR
    mov eax, cr4
    or eax, 90h // PGE, PSE bits on
    mov cr4, eax
    mov eax, cr0
    or eax, 80000000h // PE bit on
    mov cr0, eax // Enable paging
    sti // enable interrupts
}

addr = (u32_t *)PF_ADDR;

printf("Memory %p: %d\n", addr, *addr); // BSOD
```



P	0	The fault was caused by a non-present page.
	1	The fault was caused by a page-level protection violation.
W/R	0	The access causing the fault was a read.
	1	The access causing the fault was a write.
U/S	0	A supervisor-mode access caused the fault.
	1	A user-mode access caused the fault.
RSVD	0	The fault was not caused by reserved bit violation.
	1	The fault was caused by a reserved bit set to 1 in some paging-structure entry.
I/D	0	The fault was not caused by an instruction fetch.
	1	The fault was caused by an instruction fetch.
PK	0	The fault was not caused by protection keys.
	1	There was a protection-key violation.
SGX	0	The fault is not related to SGX.
	1	The fault resulted from violation of SGX-specific access-control requirements.

Figure 4-12. Page-Fault Error Code

```
void __declspec(naked) pf_handler(void)
{
    __asm {
        push eax
        push edx
        mov edx, cr2
        cmp edx, PF_ADDR
        jnz pf
        mov eax, p_np_pde
        or dword ptr[eax], 1h // set present bit
        invlpg [eax] // flush TLB cache entry
        lea eax, incr // increment our counter
        add [eax], 1
        jmp done
pf:
        pop edx
        pop eax
        push old_segment // call default PF-handler
        push old_offset
        retf
done:
        pop edx
        pop eax
        add esp, 4 // pop error code
        iretd // 32-bit return from interrupt!
    }
}
```

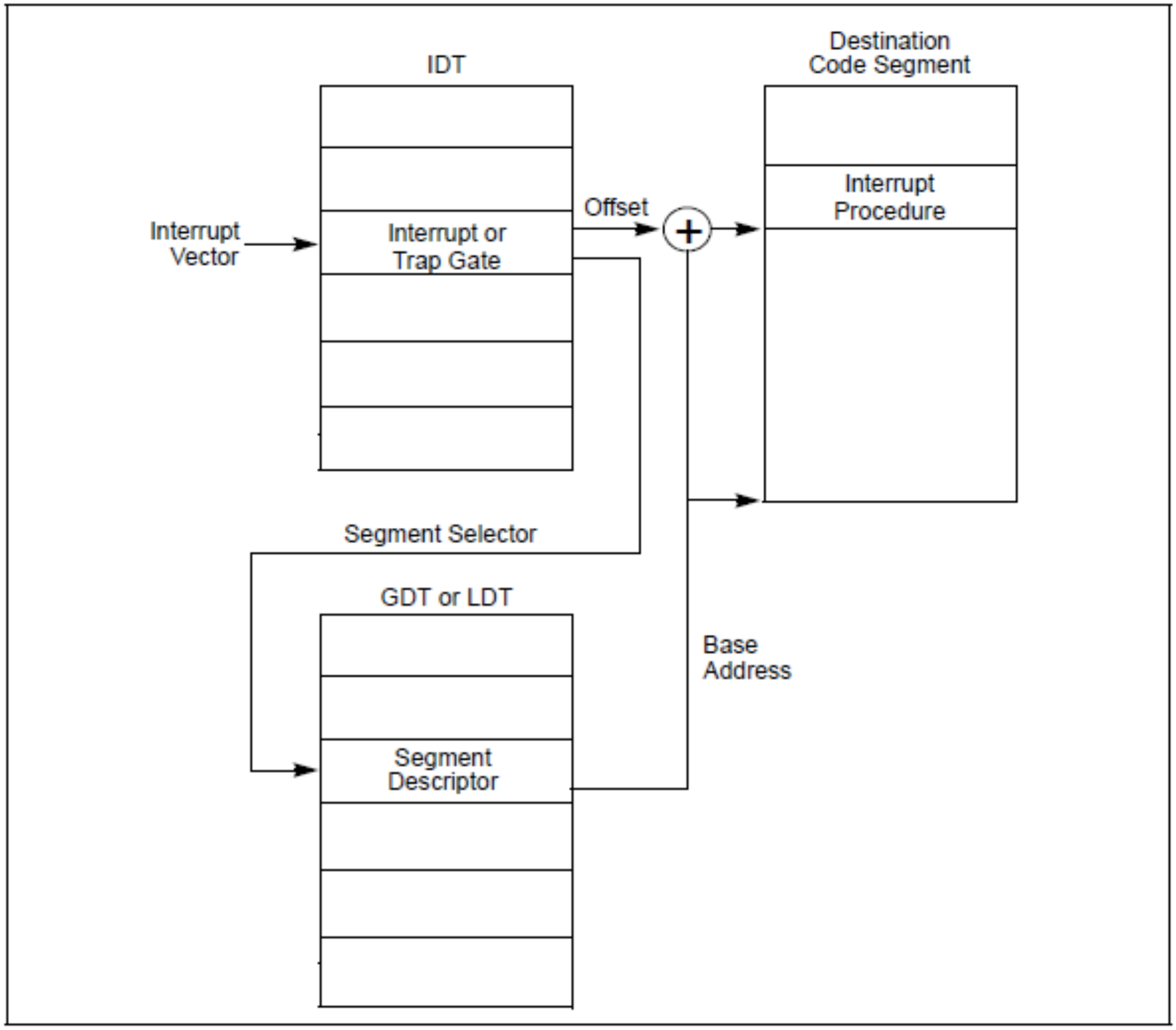


Figure 6-3. Interrupt Procedure Call

```
#define PF_NUM 14 // Number of PF handler gate in IDT

// Get offset and segment to put into IDT
__asm {
    mov edx, offset pf_handler
    mov new_offset, edx
    mov ax, seg pf_handler
    mov new_segment, ax
}

// Store default PF-handler
old_offset = idt[PF_NUM].offset_l
           | (idt[PF_NUM].offset_h << 16);
old_segment = idt[PF_NUM].seg_sel;

// Replace default handler with our handler
idt_set_gate(PF_NUM,
    (u32_t)new_offset,
    new_segment,
    idt[PF_NUM].flags);
```

```
// IDT entry
typedef struct _IDT_ENTRY {
    u16_t offset_l;
    u16_t seg_sel;
    u8_t  zero;
    u8_t  flags;
    u16_t offset_h;
} IDT_ENTRY, *PIDT_ENTRY;

PIDT_ENTRY idt;
```

```
void idt_set_gate(u8_t num,
    u32_t offset,
    u16_t seg_sel,
    u8_t flags
) {
    idt[num].offset_l = offset & 0xFFFF;
    idt[num].offset_h = (offset >> 16) & 0xFFFF;
    idt[num].seg_sel = seg_sel;
    idt[num].zero = 0;
    idt[num].flags = flags;
}
```

```
addr = (u32_t *)PF_ADDR;


/* Access to the memory
that we marked as not present
should generate page fault exception */

// to see default page fault
printf("Memory %p: %d\n", addr, *(addr + 4));

printf("Memory %p: %d\n", addr, *addr); // to recover
page
```


Labs
Thanks, DOS!