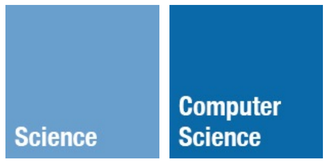
x86 & xv6 overview



CS 450: Operating Systems

Michael Lee <lee@iit.edu>

Agenda

- Motivation
- x86 ISA
- PC architecture
- UNIX
- xv6

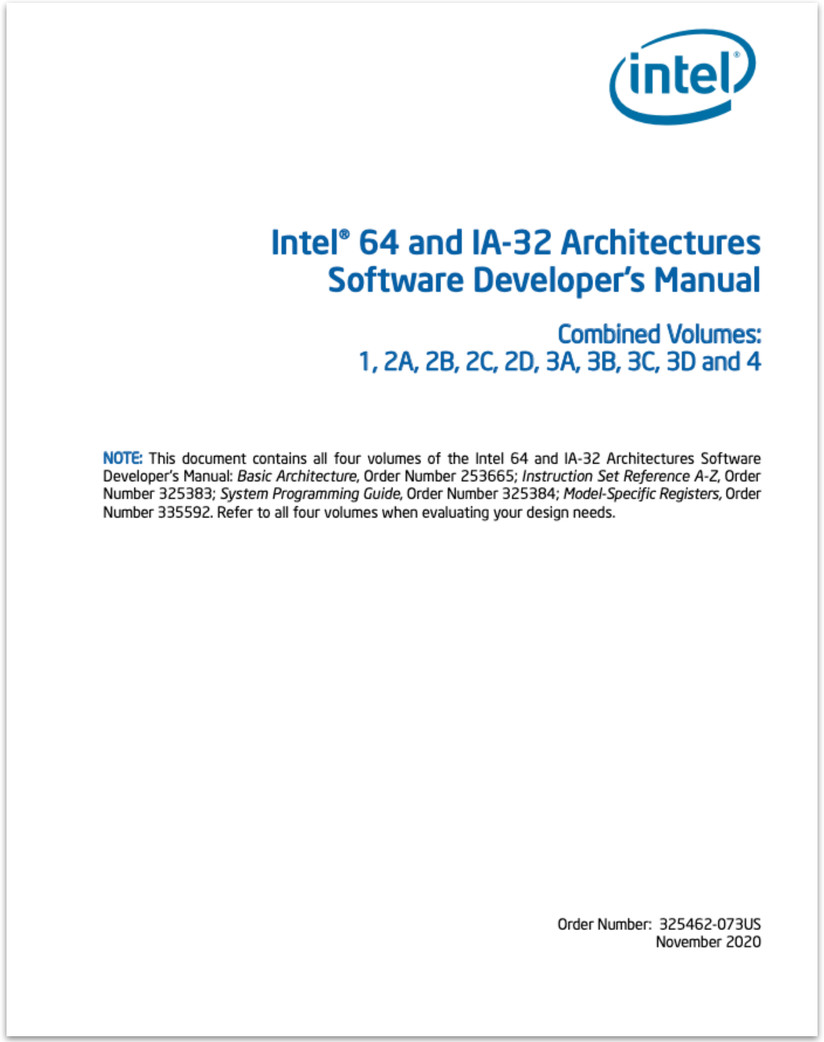
Motivation

- OS relies on many low-level hardware mechanisms to do its job
- To work on an OS kernel, we must be intimately familiar with the underlying ISA and PC hardware
  - Hardware may dictate what is or isn't possible, and influence how we represent and manage system-level structures
- We focus on x86, but all modern ISAs support the mechanisms we need
  - e.g., xv6 has been ported to ARM already

§ x86

Documentation

- Intel IA-32 Software Developer's Manuals are complete references
  - Volume 1: Architectural Overview
  - Volume 2: Instruction Set Reference
  - **Volume 3: Systems Programming Guide**
- Many diagrams in slides taken from them



x86 coverage

- Timeline
- Syntax
- Registers
- Instruction operands
- Instructions and sample usage
- Processor modes
- Interrupt & Exception handling

Timeline

- **1978**: Intel released 8086, a 16-bit CPU
- **1982**: 80186 and 80286 (still 16-bit)
- **1985**: 80386 was the first 32-bit x86 CPU (aka i386/IA-32)
- **2000**: AMD created x86-64: 64-bit ISA compatible with x86
- **2001**: Intel released IA-64 “Itanium” ISA, *incompatible* with x86
  - End-of-life announced in 2019 (i.e., official failure)

x86 ISA

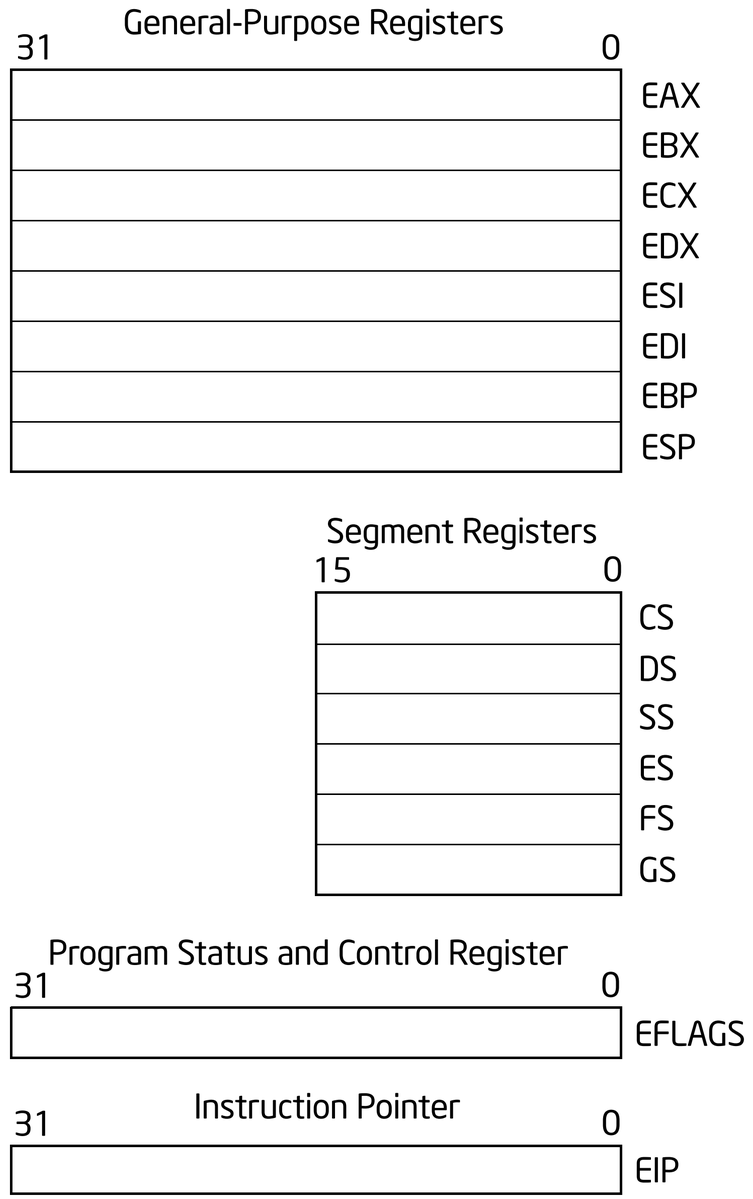
- xv6 uses the IA-32 ISA
  - But we can still build/run it on x86-64!
- x86 is a CISC ISA, so we have:
  - Memory operands for non-load/store instructions
  - Complex addressing modes
  - Relatively large number of instructions

Syntax / Formatting

- Two common variants: Intel and AT&T syntax
- *Intel syntax* common in Windows world
  - e.g., `mov DWORD PTR [ebp-4], 10 ; format: OP DST, SRC`
- *AT&T syntax* common in UNIX world (default GCC output)
  - e.g., `movl $10, -4(%ebp) # format: OP SRC, DST`
  - We will use this syntax

Registers

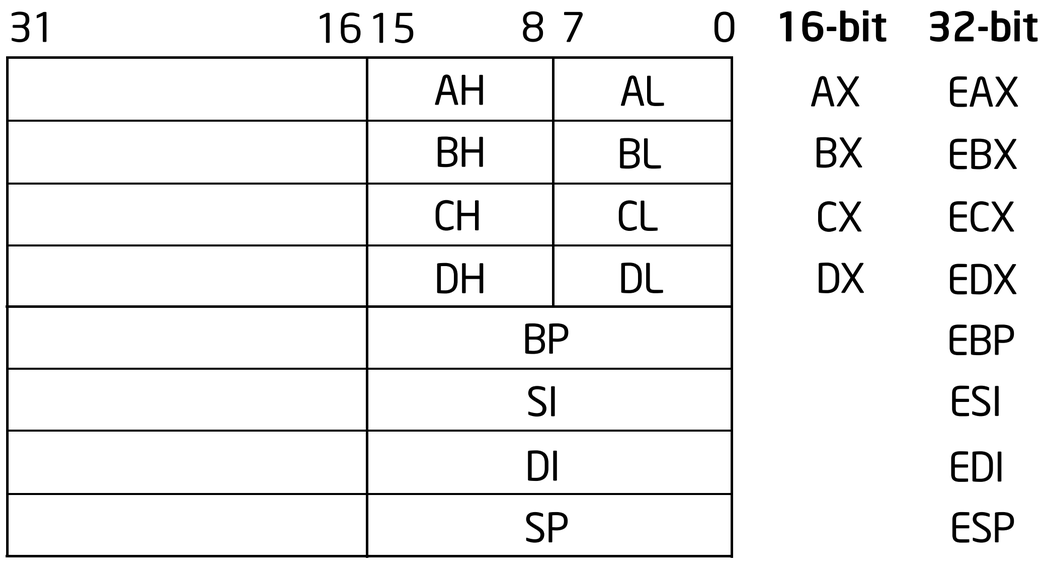
- 8 general-purpose registers
- 6 segment registers for addressing
- Status & Control register
- Program counter / Instruction pointer
- (Many others — including control registers — coming up later)



General purpose registers

- Can be directly manipulated, but some have special applications
- Most can be accessed as full 32-bit values, or as 16/8-bit subvalues
- Each register is, by convention, *volatile* or *non-volatile*
  - A *volatile* register may be clobbered by a function call; i.e., its value should be saved — maybe on the stack — if it must be preserved
  - A *non-volatile* register is preserved (by callees) across function calls

General-Purpose Registers



16-bit	32-bit
AX	EAX
BX	EBX
CX	ECX
DX	EDX
	EBP
	ESI
	EDI
	ESP

Register	Purpose
`%eax`	Return value
`%ebx`	—
`%ecx`	Counter
`%edx`	—
`%ebp`	Frame/Base pointer
`%esi`	Source index (for arrays)
`%edi`	Destination index (for arrays)
`%esp`	Stack pointer

`%eax`, **`%ecx`**, **`%edx`** are volatile registers

Instruction operands

Mode	Example(s)	Meaning
Immediate	`$0x42`, `$0xd00d`	Literal value
Register	`%eax`, `%esp`	Value found in register
Direct	`0x4001000`	Value found in address
Indirect	`(%esp)`	Value found at address in register
Base-Displacement	`8(%esp)`, `-24(%ebp)`	Given `D(B)`, value found at address `D+B` (i.e., address in base register `B` + numeric offset `D`)
Scaled Index	`8(%esp,%esi,4)`	Given `D(B,I,S)`, value found at address `D+B+I×S` `S` ∈ {`1`,`2`,`4`,`8`}; `D` and `I` default to `0` if left out, `S` defaults to `1`

Memory references

Instructions

- Instructions have 0-3 operands
  - For many 2 operand instructions, one operand is both read and written
    - e.g., `addl $1, %eax  # %eax = %eax + 1`
- Instruction suffix indicates width of operands (`l`/`w`/`b` → 32/16/8 bits)
- Arithmetic operations populate `EFLAGS` register bits, including `ZF` (zero result), `SF` (signed/neg result), `CF` (carry-out of MSB occurred), `OF` (overflow occurred)
  - Used by subsequent conditional instructions (e.g., jump if result = zero)

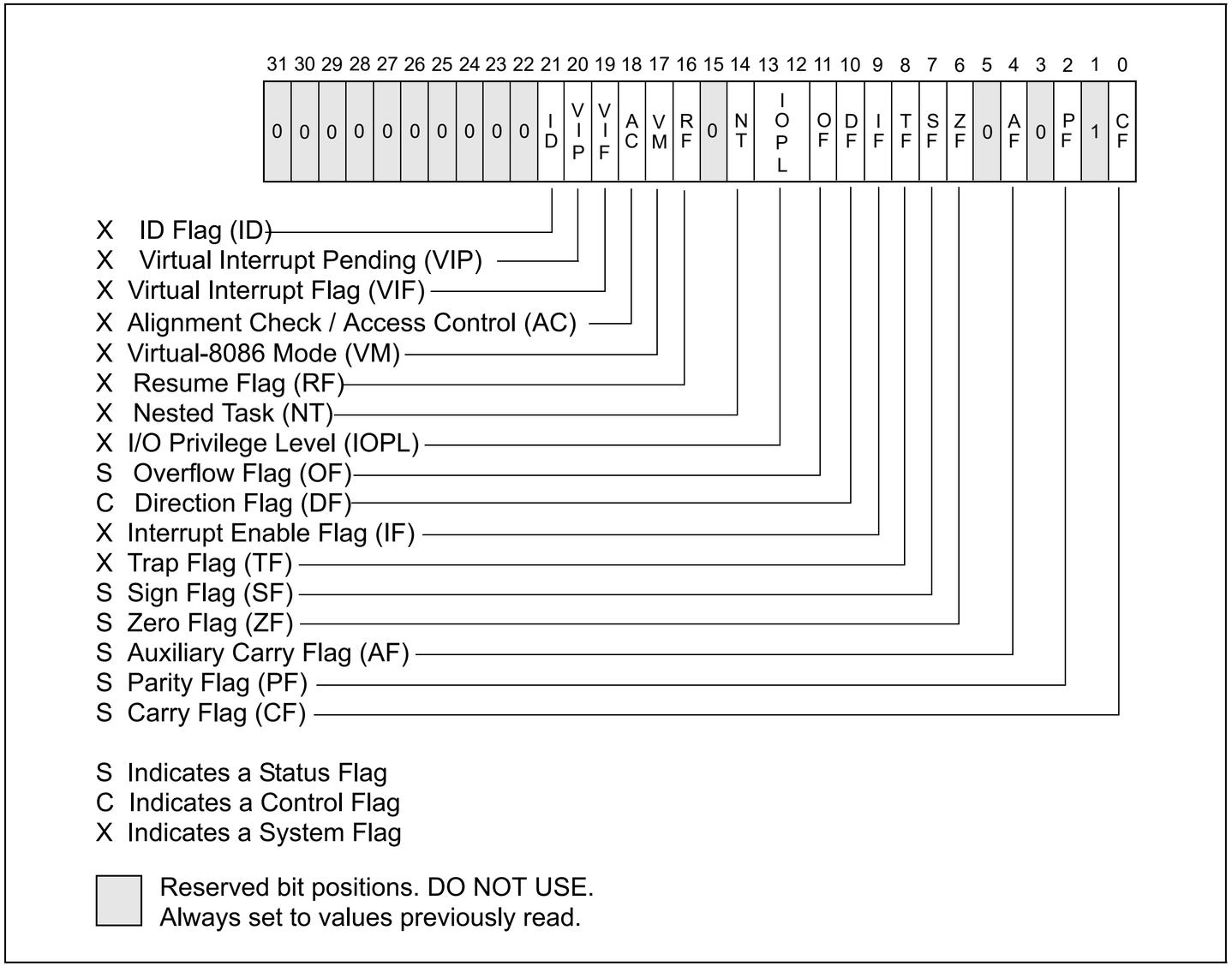


Figure 3-8. EFLAGS Register

Arithmetic

Instruction(s)	Description
`{add,sub,imul}` ***`src, dst`***	dst = dst {+,–,×} src
`neg` ***`dst`***	dst = –dst
`{inc,dec}` ***`dst`***	dst = dst {+,–} 1
`{sal,sar,shr}` ***`src, dst`***	dst = dst {<<,>>,>>>} src (arithmetic & logical shifts)
`{and,or,xor}` ***`src, dst`***	dst = dst {&,|,^} src (bitwise)
`not` ***`dst`***	dst = ~dst (bitwise)

`src` can be an immediate, register, or memory operand; ***`dst`*** can be a register or memory operand.
But at most one memory operand!

Conditions and Branches

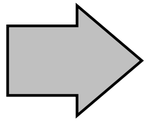
Instruction(s)	Description
`cmp` *`src, dst`*	dst – src (discard result but set flags)
`test` *`src, dst`*	dst & src (discard result but set flags)
`jmp` *`target`*	Unconditionally jump to target (change `%eip`)
`{je,jne}` *`target`*	Jump to target if dst equal/not equal src (ZF=1 / ZF=0)
`{jl,jle}` *`target`*	Jump to target if dst </≤ src (SF≠OF / ZF=1 or SF≠OF)
`{jg,jge}` *`target`*	Jump to target if dst >/≥ src (ZF=0 and SF=OF / SF=OF)
`{ja,jb}` *`target`*	Jump to target if dst above/below src (CF=0 and ZF=0 / CF=1)

conditional jump often follows `cmp` (or `test`)

`target` is usually an address encoded as an immediate operand (e.g., `jmp $0x4001000`), but addresses may be stored in a register or memory, in which case *indirect addressing* is required, which uses the `*` symbol.
E.g., `jmp *%eax` (jump to address in `%eax`), `jmp *0x4001000` (jump to address found at address 0x4001000)

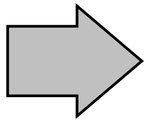
E.g., basic control structures

```
if (cond) {
    // if-clause
} else {
    // else-clause
}
...
```



```
    testl %eax, %eax  # %eax = cond
    je ELSE
    # if-clause
    jmp ENDIF
ELSE:
    # else-clause
ENDIF:
    # ...
```

```
while (cond) {
    // loop-body
}
...
```



```
    testl %eax, %eax  # %eax = cond
    je ENDLOOP
LOOP:
    # loop-body
    testl %eax, %eax
    jne LOOP
ENDLOOP:
    # ...
```

Data movement

Instruction(s)	Description
`mov src, dst`	Copy data from src to dst (memory→memory moves not possible)
`movzbl src, dst`	Copy 8-bit value to 32-bit target (& other variants), using zero-fill
`movsbl src, dst`	Copy 8-bit value to 32-bit target (& other variants), using sign-extension
`{cmove/ne} src, dst`	Move data from src to dst if ZF=1 / ZF=0
`{cmovg/ge/l/le/a/b/…}`	Conditionally move data from src to dst (per jump naming conventions)

Address computation

`lea address, dst`	dst = address (no memory access! just computes value of address)

Functions and Call stack

Instruction(s)	Description
`push` *`src`*	Push src onto stack
`pop` *`dst`*	Pop top of stack into dst
`call` *`target`*	Push current `%eip` (address of instruction after call) onto stack, jump to target
`leave`	Restore frame pointer (`%ebp`) and clears stack frame
`ret`	Pop top of stack into `%eip`

All instructions above implicitly adjust `%esp` and access the stack.

`target` may use *indirect addressing* as well, e.g., `call *%eax` (call function whose address is in `%eax`)

Function calls

- Functions make extensive use of the call stack — leads to convention-driven *prologue* and *epilogue* blocks in assembly code
- Typical function prologue:
  - Save old frame pointer and establish new frame pointer
  - Save non-volatile register values we might clobber (“callee-saved”)
  - Load needed parameters from prior stack frame
  - Allocate stack space for any local data

Function calls

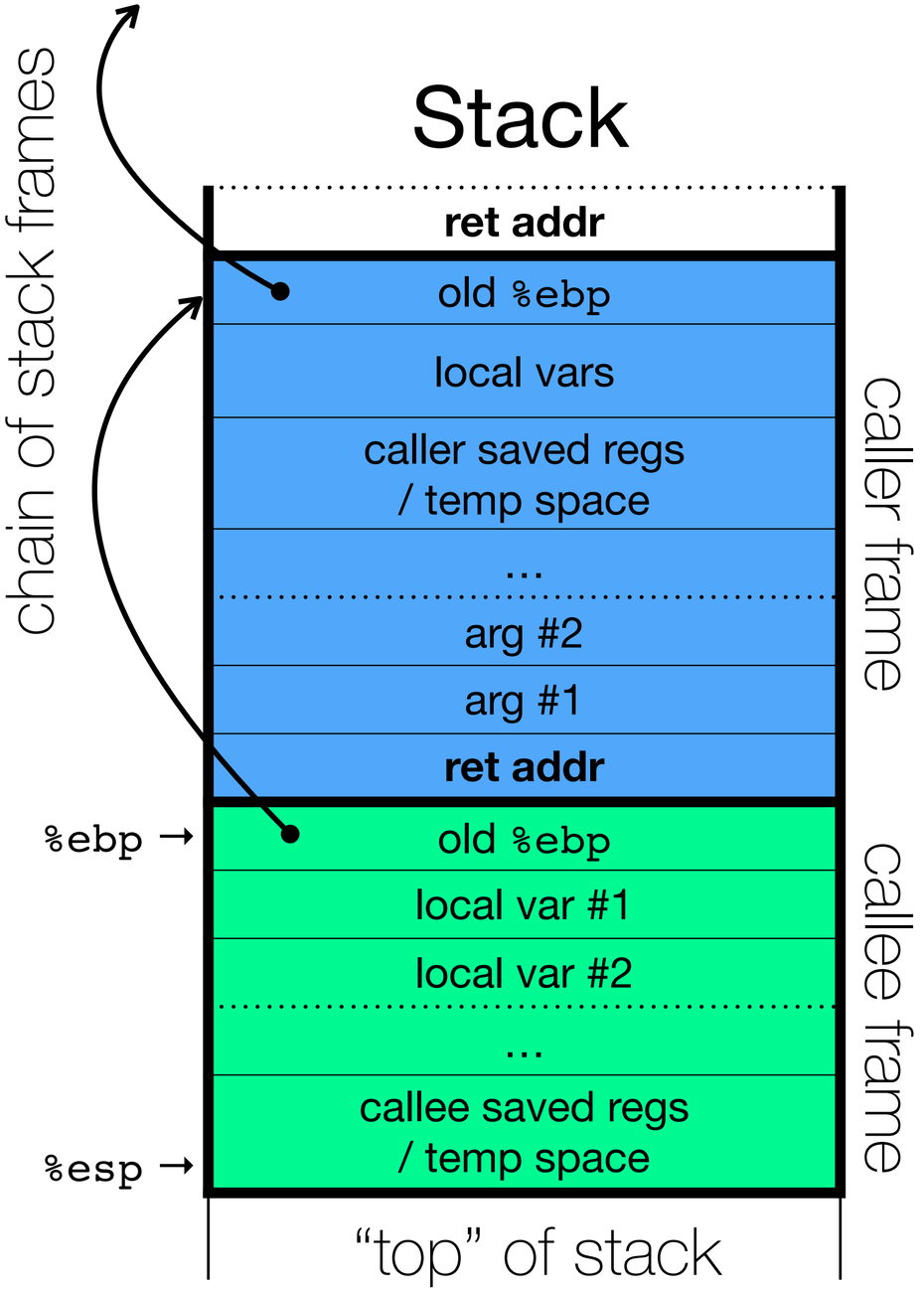
- Typical function epilogue:
  - Place return value in `%eax`
  - Deallocate any space used for local data
  - Restore/Pop any clobbered non-volatile register values
  - Restore/Pop old frame pointer
  - Return

Function calls (Optimization)

- Many of these steps may be optimized (simplified or neglected altogether) by the compiler!
  - Prefer registers to stack-based args or local vars (regs vs. memory)
  - `%esp` doesn't always reflect the top of the stack (only need to do this if calling another function)
  - `lea` often used in surprising ways (addressing modes as arithmetic)

Call Stack

- Maintains dynamic state and context of executing program
- Saved frame pointers (previous values of `%ebp`) create a chain of stack frames
  - Useful to navigate for debugging and tracing! (e.g., gdb “backtrace”)



E.g., function calls

```
int main() {
    int x=10, y=20;
    sum(x, y);
    return 0;
}

int sum(int a, int b) {
    int ret = a + b;
    return ret;
}
```

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $10, -4(%ebp)
    movl    $20, -8(%ebp)
    movl    -4(%ebp), %edi
    movl    -8(%ebp), %esi
    call    sum
    movl    $0, %eax
    addl    $16, %esp
    popl    %ebp
    ret
```

```
sum: # unoptimized
    pushl   %ebp
    movl    %esp, %ebp
    movl    %edi, -4(%ebp)
    movl    %esi, -8(%ebp)
    movl    -4(%ebp), %eax
    addl    -8(%ebp), %eax
    movl    %eax, -12(%ebp)
    movl    -12(%ebp), %eax
    popl    %ebp
    ret

sum: # optimized
    leal    (%edi,%esi), %eax
    ret
```

Processor modes

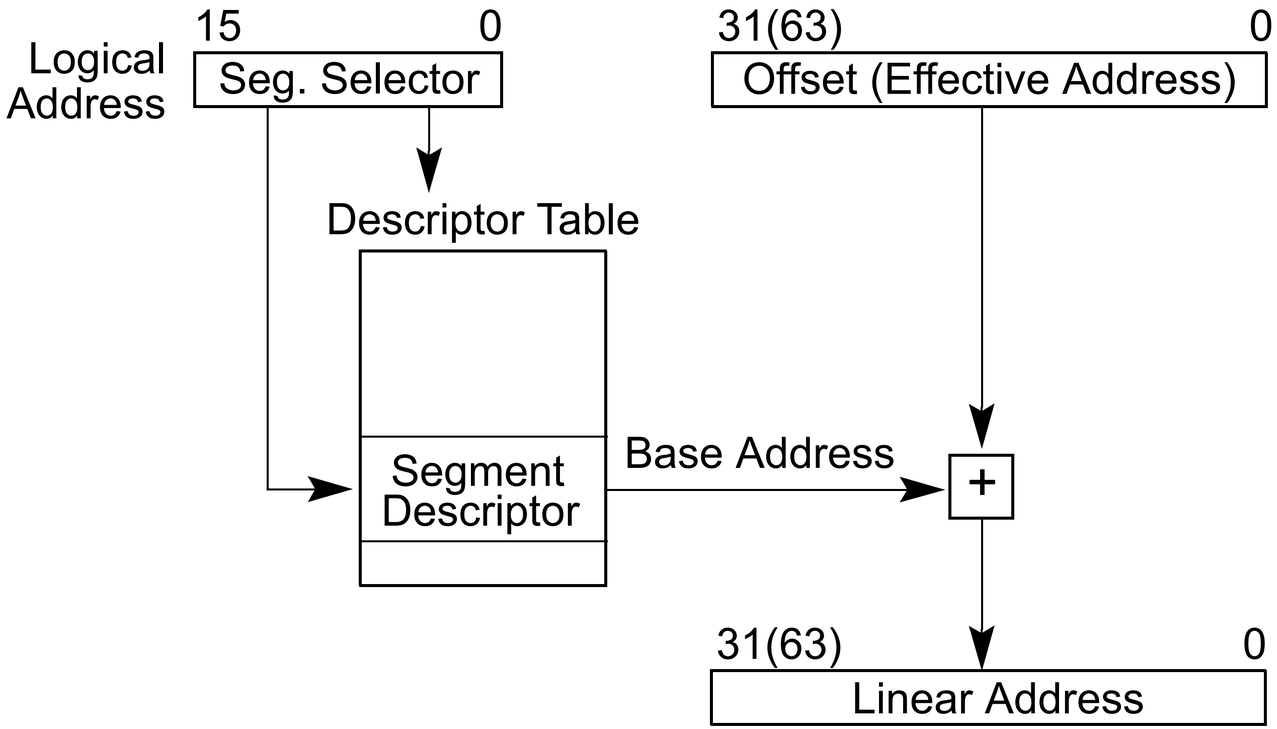
- When an x86 system first boots up, it runs in **16-bit real mode** (8086 compatible) — all addresses reference “real” memory locations
- **16/32-bit protected modes** add privilege levels, virtual memory, and other mechanisms useful to the OS (e.g., for multitasking)
- **64-bit long mode** removes some instructions and adds 64-bit registers and addressing

Real mode addressing

- Only 16-bit registers, but support for **20-bit** addresses (1MB address space) through the use of segment registers: `CS`, `DS`, `ES`, `SS`
  - Left-shift segment number by 4 (i.e., ×16) to obtain *base address*, and add to *offset* to compute 20-bit physical address
- Code (via `IP`) and Stack (via `SP` and `BP`) accesses automatically use `CS` (code segment) and `SS` (stack segment) to compute addresses
  - e.g., if `IP`=0x4000 and `CS`=0x1100, `CS:IP` refers to physical address 0x1100×16 + 0x4000 = 0x15000

Protected mode

- Segment registers (expanded to `CS`, `DS`, `SS`, `ES`, `FS`, `GS`) no longer hold base addresses, but *selectors*
  - Selectors are used to load *segment descriptors* from a descriptor table which describe location/size/status/etc. of segments
- `CS` selector contains a 2-bit `CPL` in addition to selector value
  - Recall: privileged instructions are only available when `CPL=0`



- Segments allow complex memory mapping and access control (e.g., restricted access), among other things

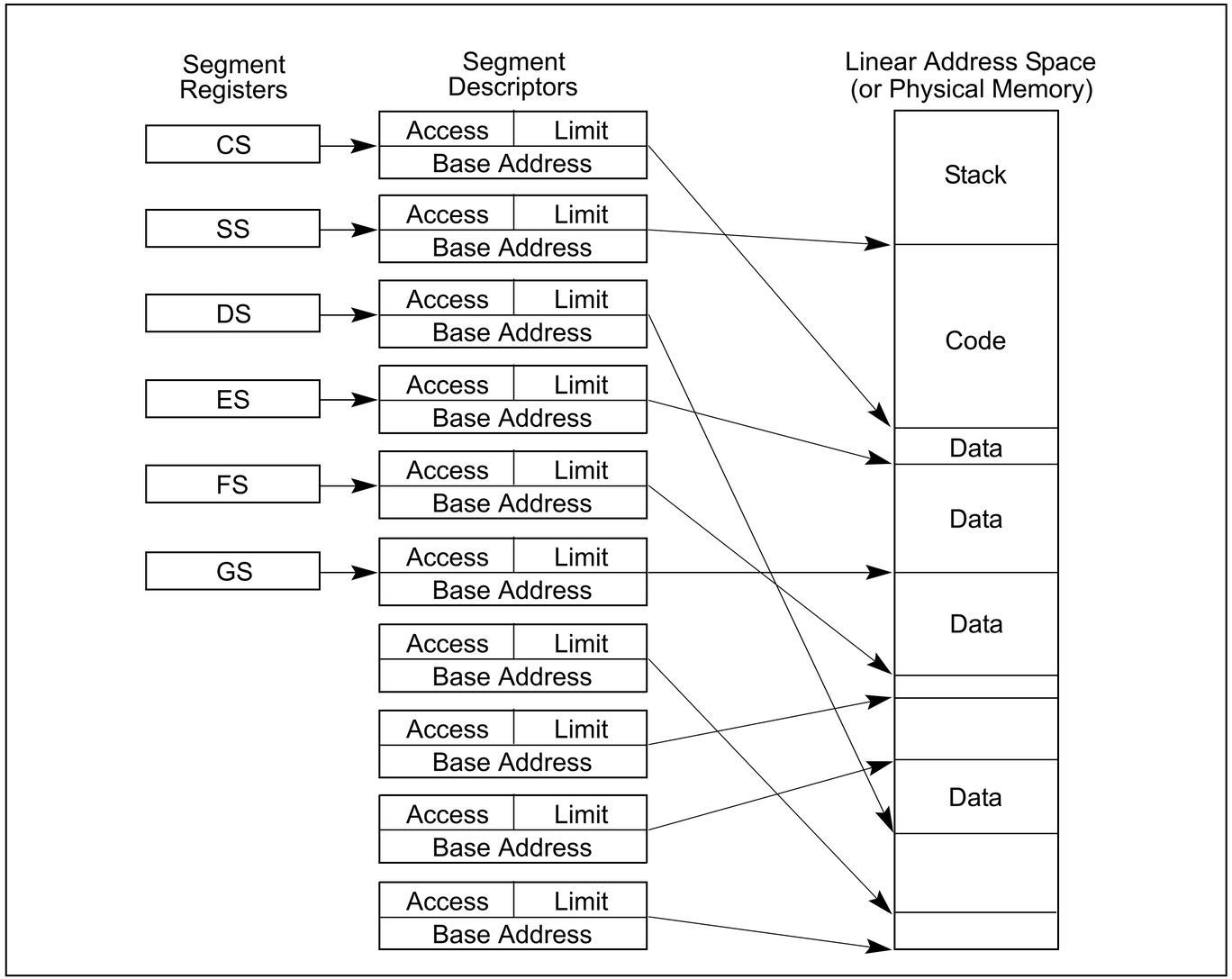
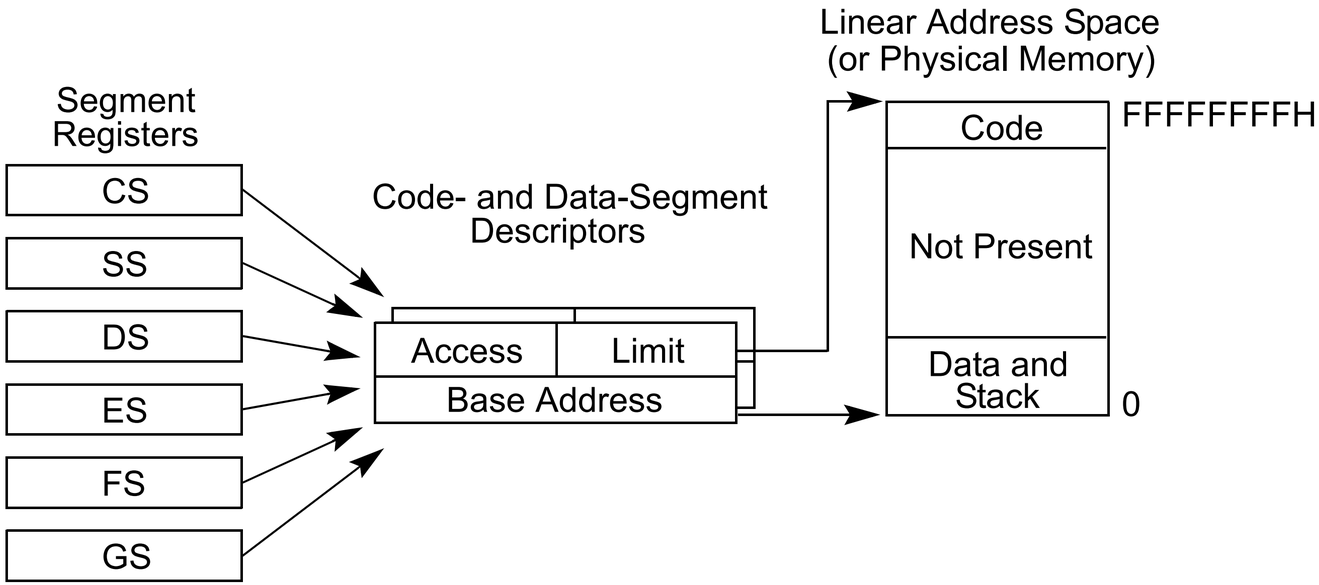
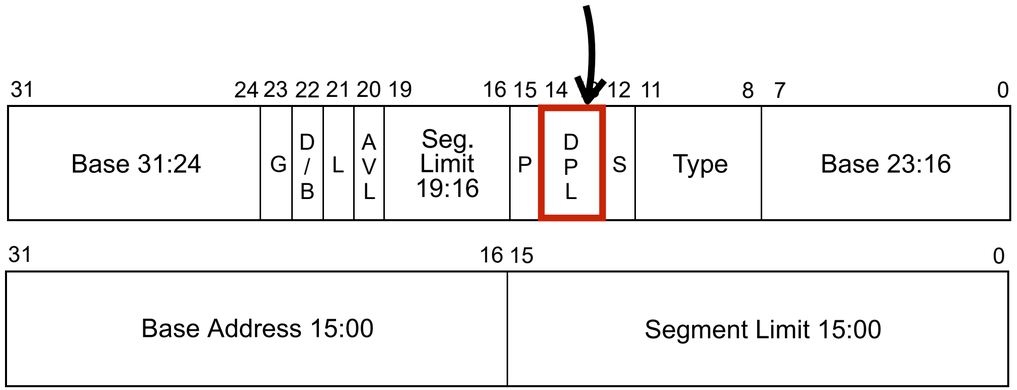


Figure 3-4. Multi-Segment Model



- In practice, a *flat model* is used by most OSes, and more granular memory mapping & protection is carried out via *paging* (coming up)
  - But segment descriptors are still used for privilege level based restrictions

DPL is loaded as CPL (in CS register) if (successful) jump occurs to this segment



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

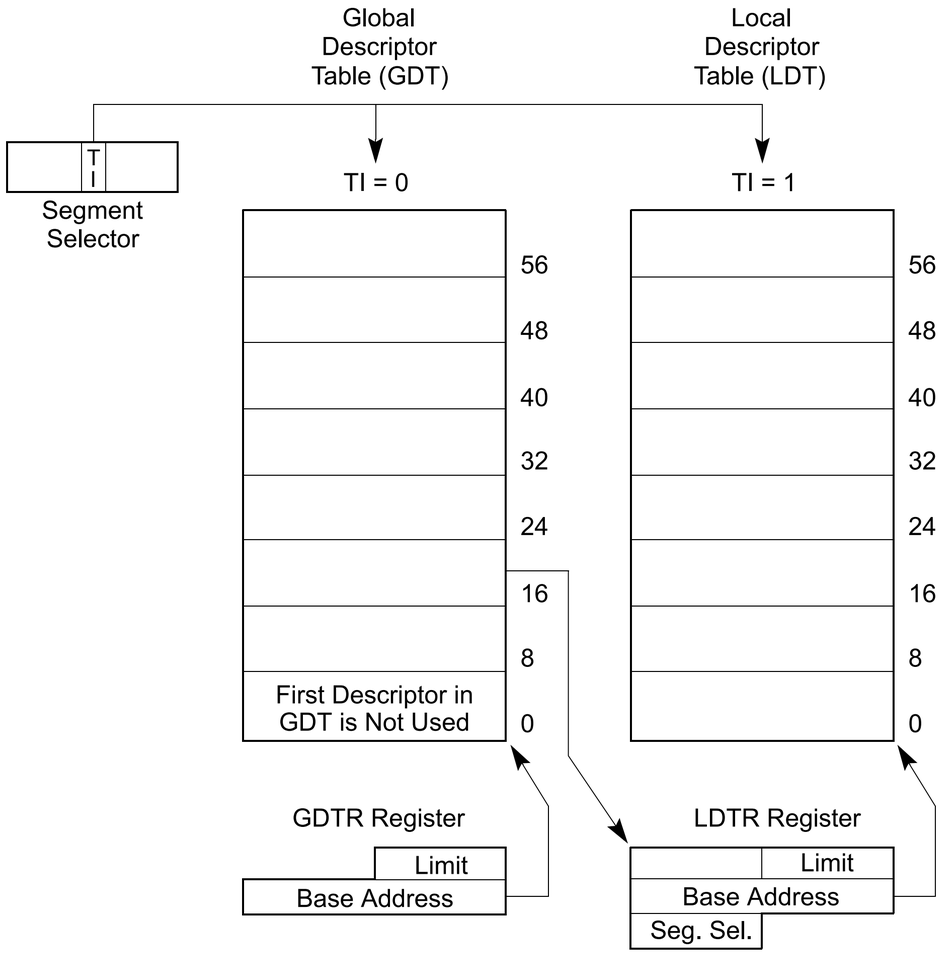
```c
// Segment Descriptor
struct segdesc {
  uint lim_15_0 : 16;  // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4;       // Segment type (see STS_ constants)
  uint s : 1;          // 0 = system, 1 = application
  uint dpl : 2;        // Descriptor Privilege Level
  uint p : 1;          // Present
  uint lim_19_16 : 4;  // High bits of segment limit
  uint avl : 1;        // Unused (available for software use)
  uint rsv1 : 1;       // Reserved
  uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
  uint g : 1;          // Granularity: limit scaled by 4K when set
  uint base_31_24 : 8; // High bits of segment base address
};

// Normal segment
#define SEG(type, base, lim, dpl) (struct segdesc)    \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }

#define SEG16(type, base, lim, dpl) (struct segdesc)  \
{ (lim) & 0xffff, (uint)(base) & 0xffff,              \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
#endif
```

Privilege check

- When loading segments, hardware ensures that CPL ≤ DPL (actually a bit more complicated, but this is very close to the truth!)
  - I.e., privilege level can only *stay the same or be lowered*
- Prevents user code from transitioning directly to kernel code
  - To elevate privilege, must do so by way of interrupts/traps!

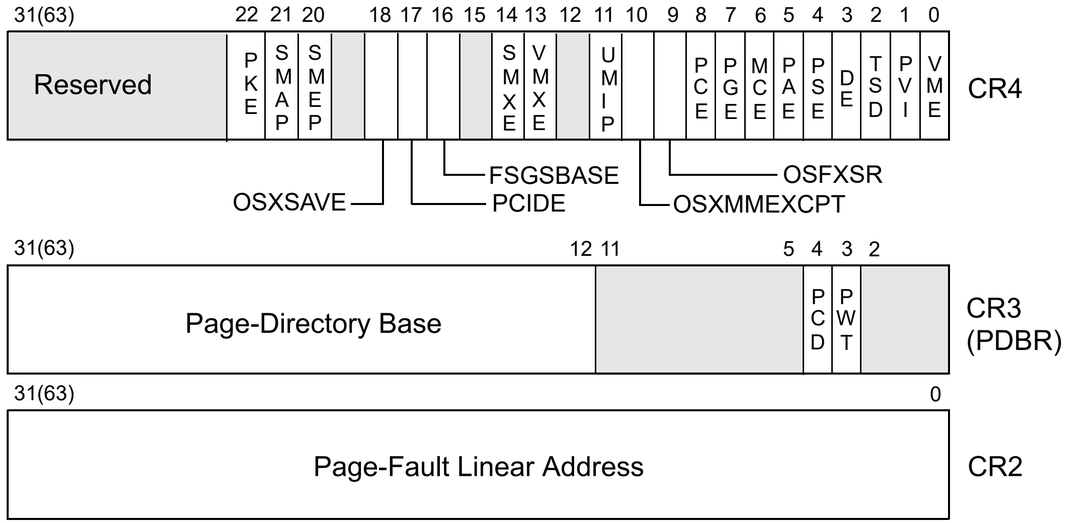
Segment descriptor tables

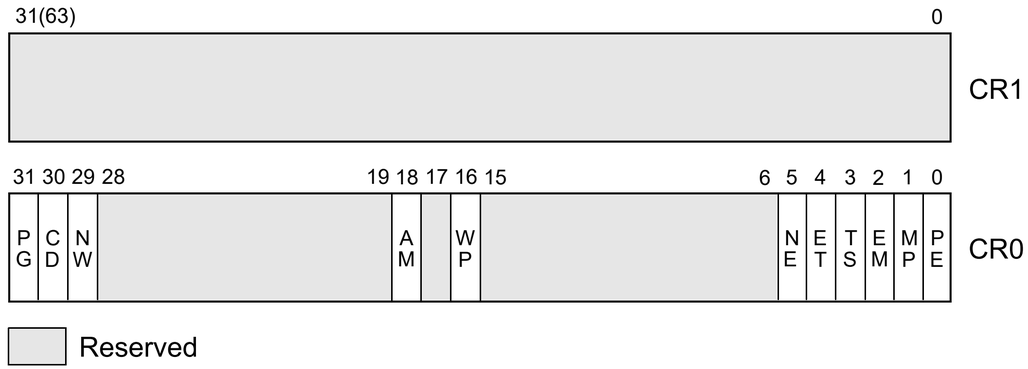
- Kernel is responsible for maintaining descriptor tables
  - System wide (Global)
  - Task-specific (Local)
- Must be set up before transitioning to protected mode



Control & System registers

- Transitioning between real & protected mode, and activating/controlling other hardware features are governed by control & system register flags





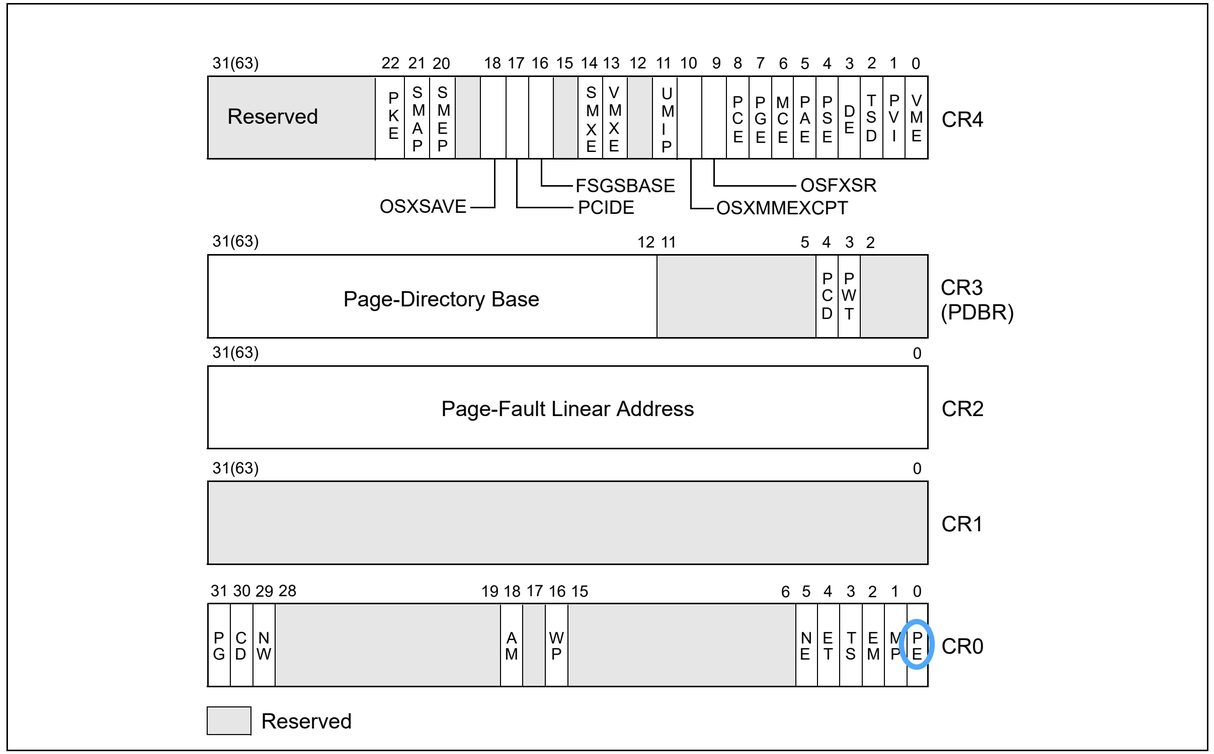


Figure 2-7. Control Registers

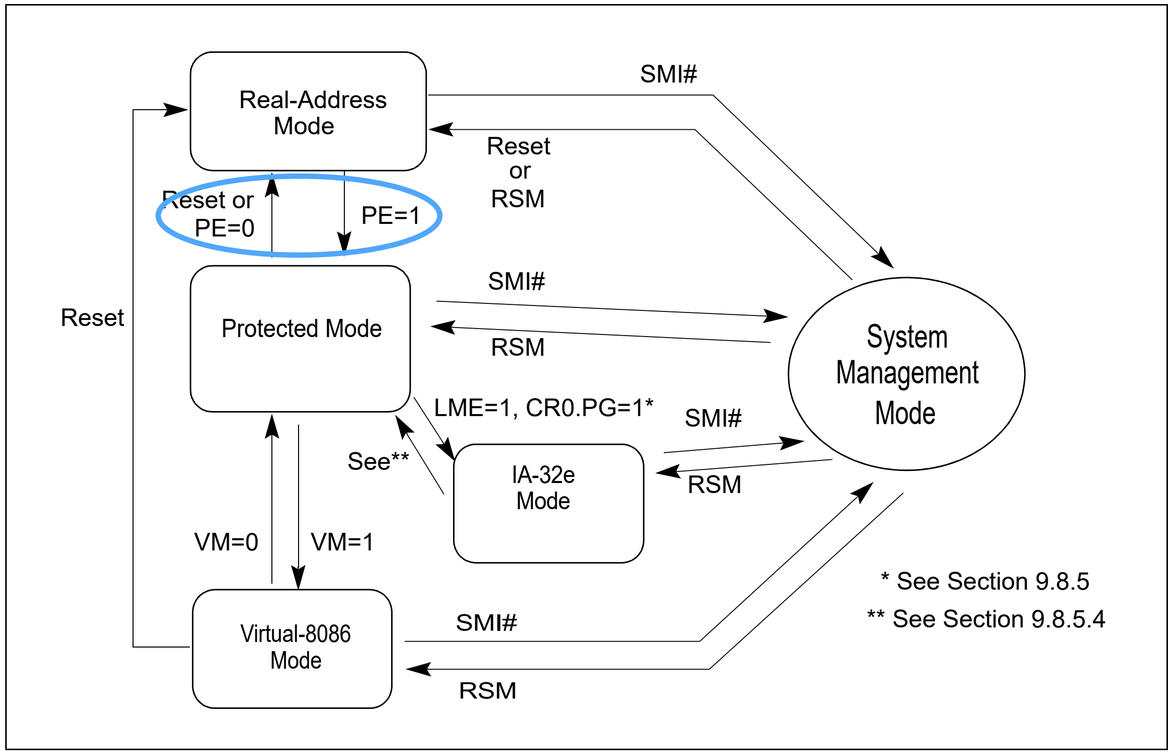


Figure 2-3. Transitions Among the Processor's Operating Modes

xv6 bootstrap code

```
.code16                         # Assemble for 16-bit mode
.globl start
start:
  cli                           # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax               # Set %ax to zero
  movw    %ax,%ds               # -> Data Segment
  ...

  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

  ljmp    $(SEG_KCODE<<3), $start32

.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  ...

# Bootstrap GDT
gdt:
  SEG_NULLASM                               # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)     # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)           # data seg
```

Paging

- Protected mode also enables virtual memory via *paging*
- A much more granular (but potentially expensive) form of virtual memory
  - Will discuss this in detail later!
- Kernel must set up and maintain per-process structures for paging, too

Interrupts & Exceptions

- Events that require special CPU attention, typically by transferring control from the active task (kernel/user) to a kernel handler
- **Interrupts** are hardware-sourced events requesting CPU attention
  - Typically unrelated to executing instruction
  - Can also be generated by software with `int` *`N`* instruction

Exceptions

- Errors/Events arising due to the *currently executing instruction*
- Subclasses:
  - **Faults**: can be corrected — after handler, return to state prior to faulting instruction (e.g., page fault)
  - **Traps**: reported immediately after execution of instruction (e.g., debugging breakpoint, system call), regular return
  - **Abort**: severe errors; cannot return to task

Handling Interrupts/Exceptions

- **Interrupt Descriptor Table** (IDT) contains descriptors (aka “gates”) associating service routines with interrupt/exception numbers
- 255 total indices (aka vector numbers):
  - 0-31: architecture-defined
  - 32-255: user-defined; can be assigned to I/O devices

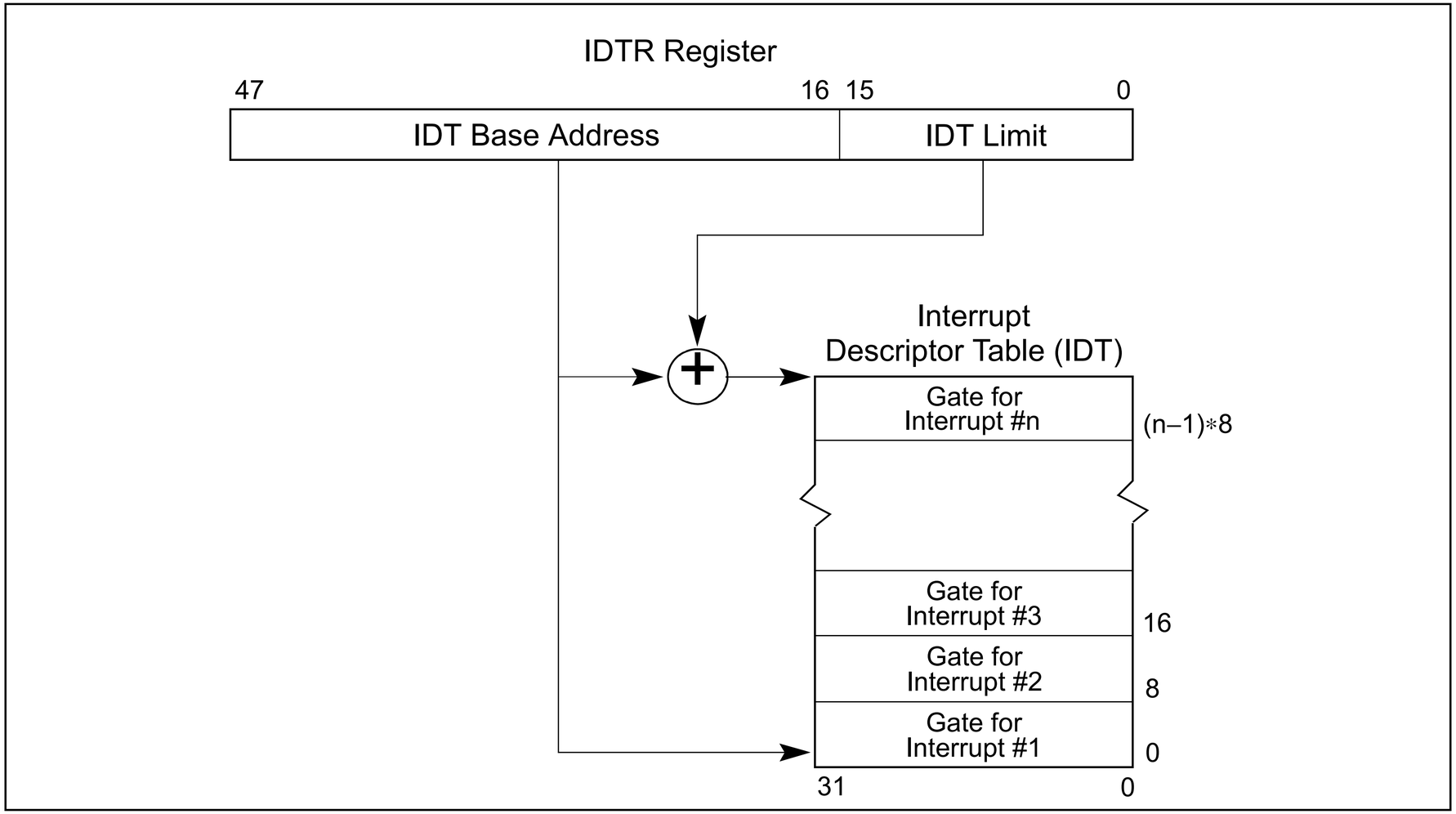


Figure 6-1. Relationship of the IDTR and IDT

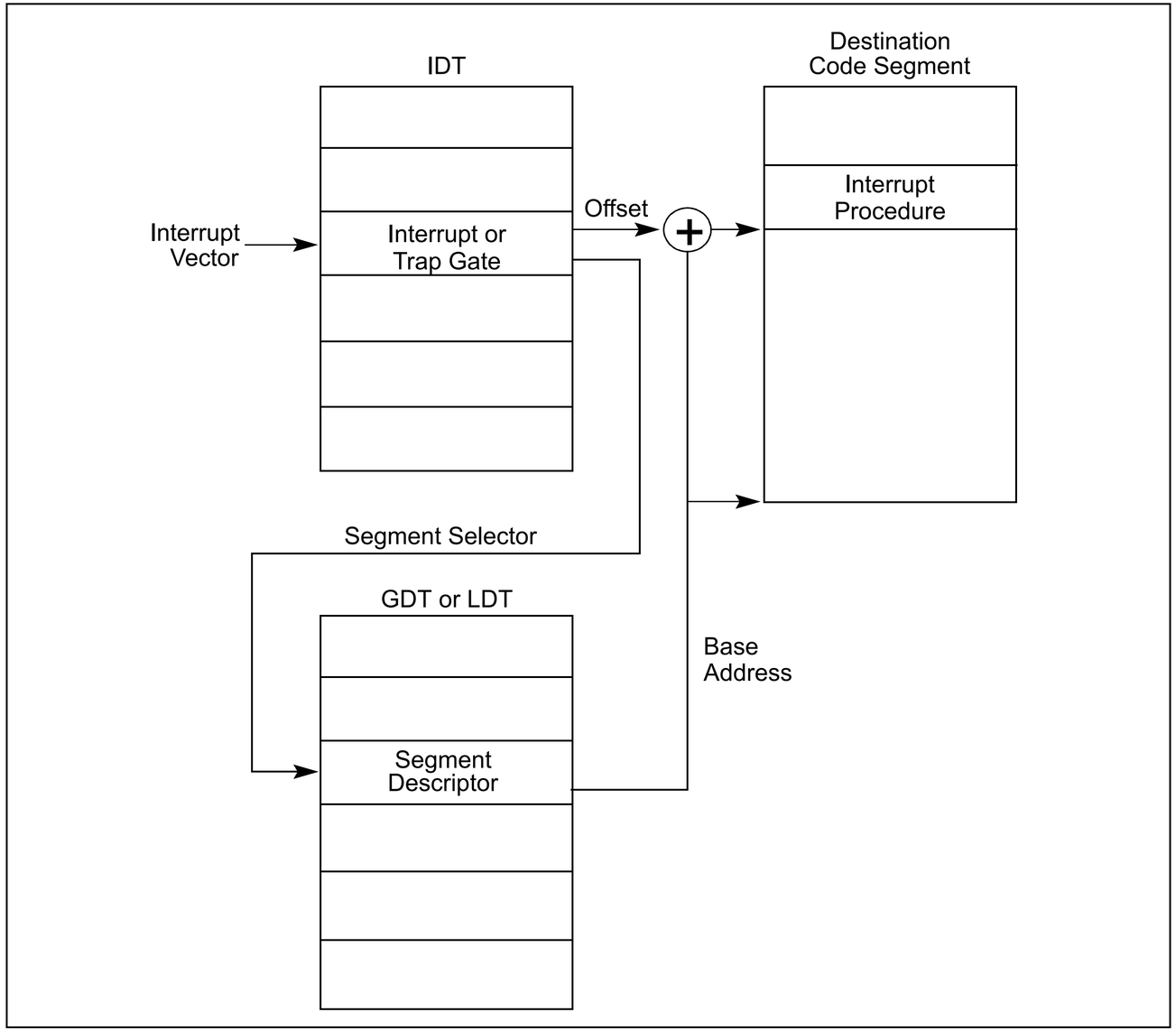


Figure 6-3. Interrupt Procedure Call

Interrupt/Exception Vectors

Table 6-1. Protected-Mode Exceptions and Interrupts

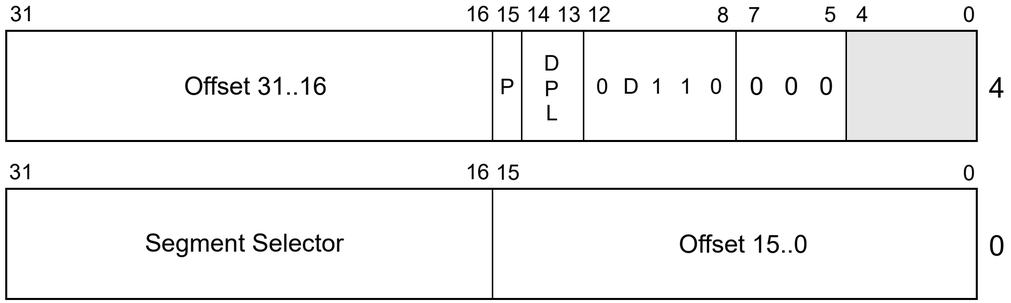
Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[1]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.

Table 6-1. Protected-Mode Exceptions and Interrupts (Contd.)

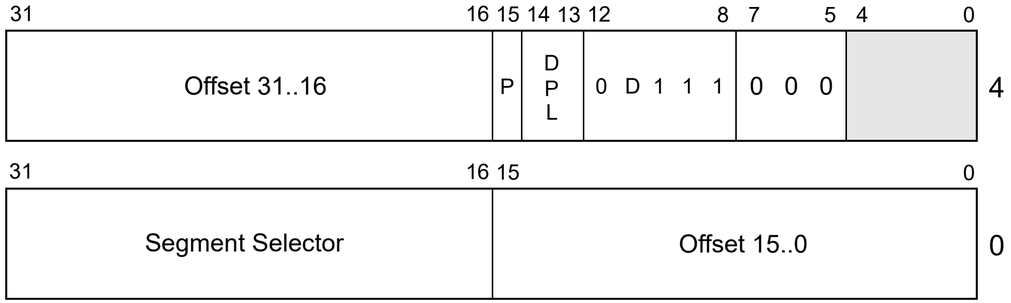
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[2]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[3]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[4]
20	#VE	Virtualization Exception	Fault	No	EPT violations[5]
21-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

Gate Descriptors

Interrupt Gate



Trap Gate



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

```
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};

#define SETGATE(gate, istrap, sel, off, d)                \
{                                                         \
  (gate).off_15_0 = (uint)(off) & 0xffff;                \
  (gate).cs = (sel);                                      \
  (gate).args = 0;                                        \
  (gate).rsv1 = 0;                                        \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
  (gate).s = 0;                                           \
  (gate).dpl = (d);                                       \
  (gate).p = 1;                                           \
  (gate).off_31_16 = (uint)(off) >> 16;                  \
}

for(i = 0; i < 256; i++)
  SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

lidt(idt, sizeof(idt));
```

Privilege check

- Three variables: `CPL`, gate `DPL`, and destination segment `DPL`
  - Destination segment `DPL` is always 0 (handler is in kernel)
- CPU guarantees that:
  - for hardware interrupts, `CPL` ≥ destination segment `DPL`
    - i.e., interrupt cannot lower privilege!
  - for software generated interrupts (via `int`), `CPL` ≤ gate `DPL`
    - i.e., can use this to allow user mode to invoke only certain interrupts
  - if assertions fail, general protection fault (#13)

Masking Interrupts

- Most external interrupts can be *masked* (i.e., ignored), by setting the `IF` (interrupt flag) in `EFLAGS`
  - `cli`/`sti` instructions: clear/set interrupt flag
- `IF` is automatically cleared when an interrupt (but not a trap) gate is taken
- How is this useful?

Interrupt-handling context

- If interrupts occur in user mode, running handler with current stack is unsafe (unpredictable state)
- TSS segment defines the currently executing task
  - General purpose registers
  - Control registers (including EFLAGS, EIP, LDTR, etc.)
  - Stack pointers *for different privilege levels*

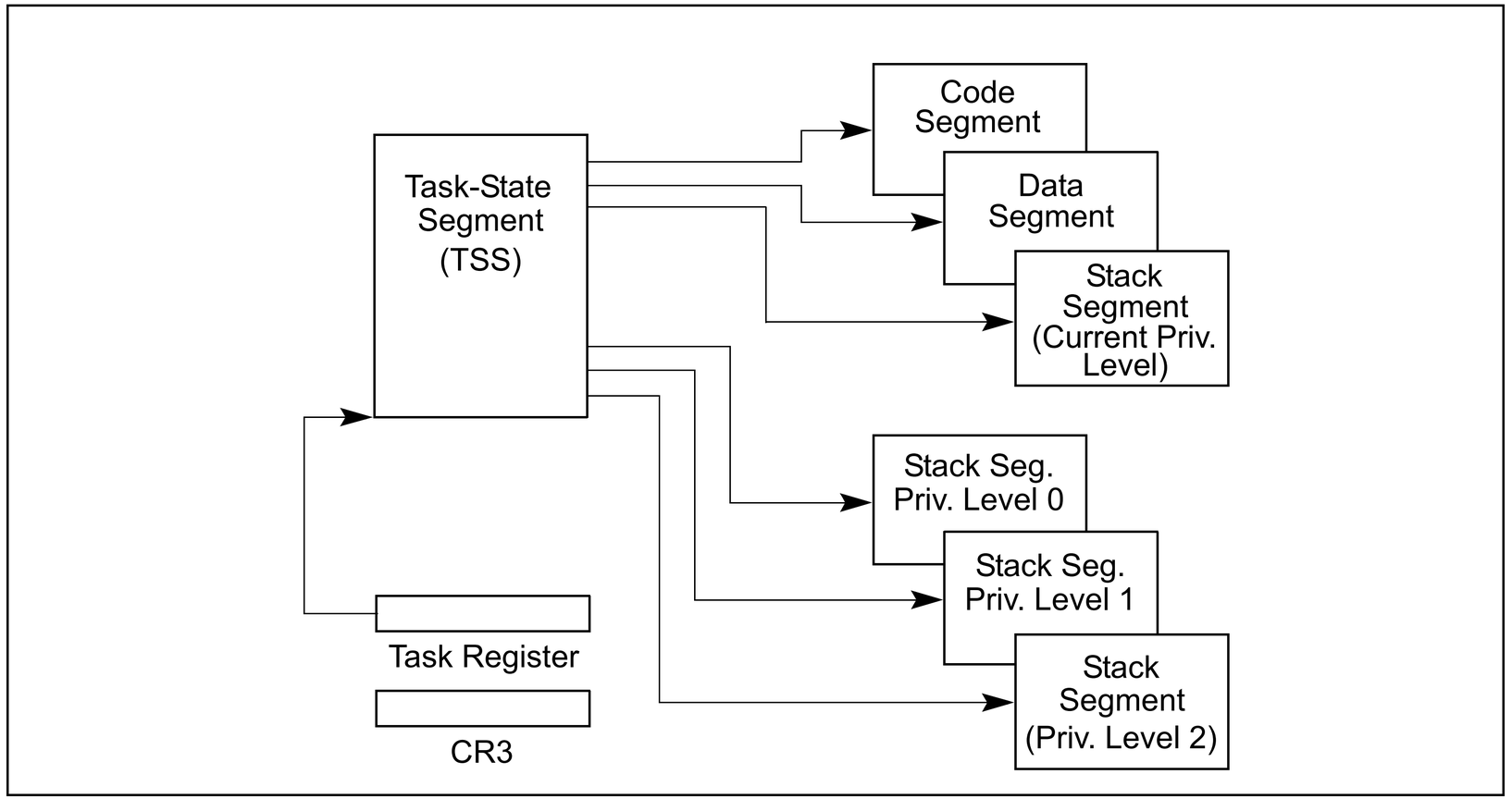


Figure 7-1. Structure of a Task

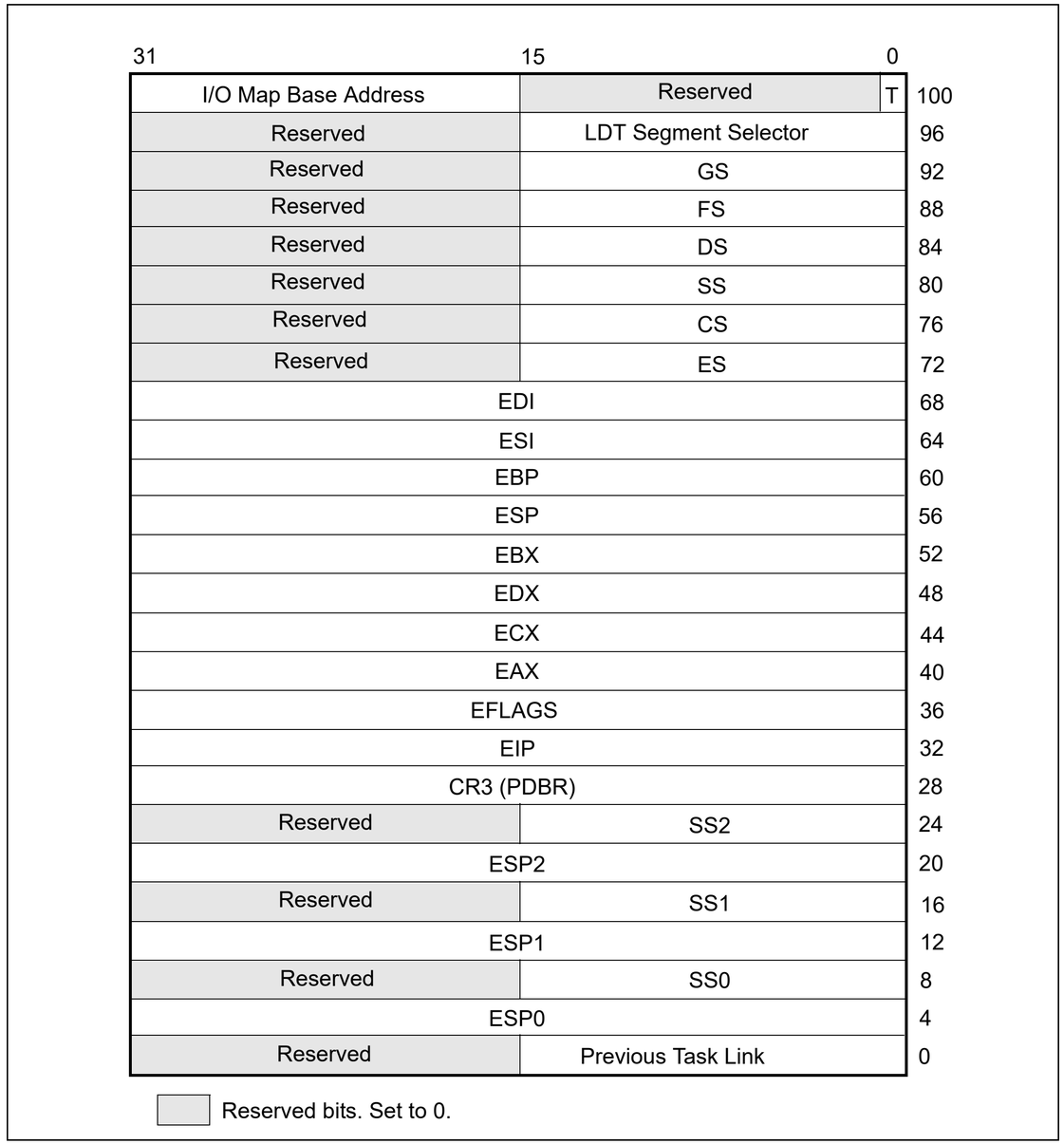


Figure 7-2. 32-Bit Task-State Segment (TSS)

Interrupt Procedure

When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be executed at a numerically lower privilege level, a stack switch occurs. When the stack switch occurs:
  a. The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task. On this new stack, the processor pushes the stack segment selector and stack pointer of the interrupted procedure.
  b. The processor then saves the current state of the EFLAGS, CS, and EIP registers on the new stack (see Figures 6-4).
  c. If an exception causes an error code to be saved, it is pushed on the new stack after the EIP value.
- If the handler procedure is going to be executed at the same privilege level as the interrupted procedure:
  a. The processor saves the current state of the EFLAGS, CS, and EIP registers on the current stack (see Figures 6-4).
  b. If an exception causes an error code to be saved, it is pushed on the current stack after the EIP value.

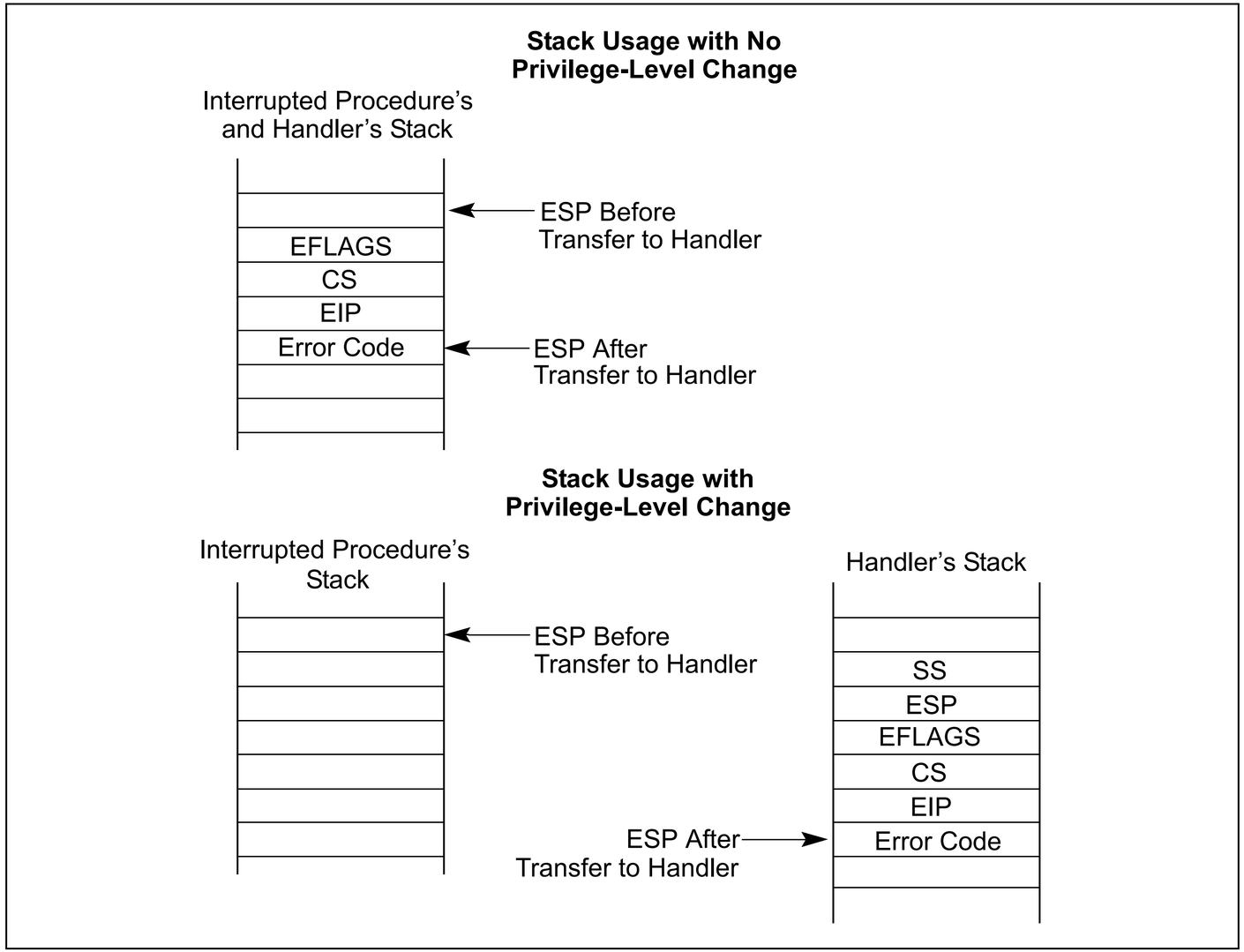
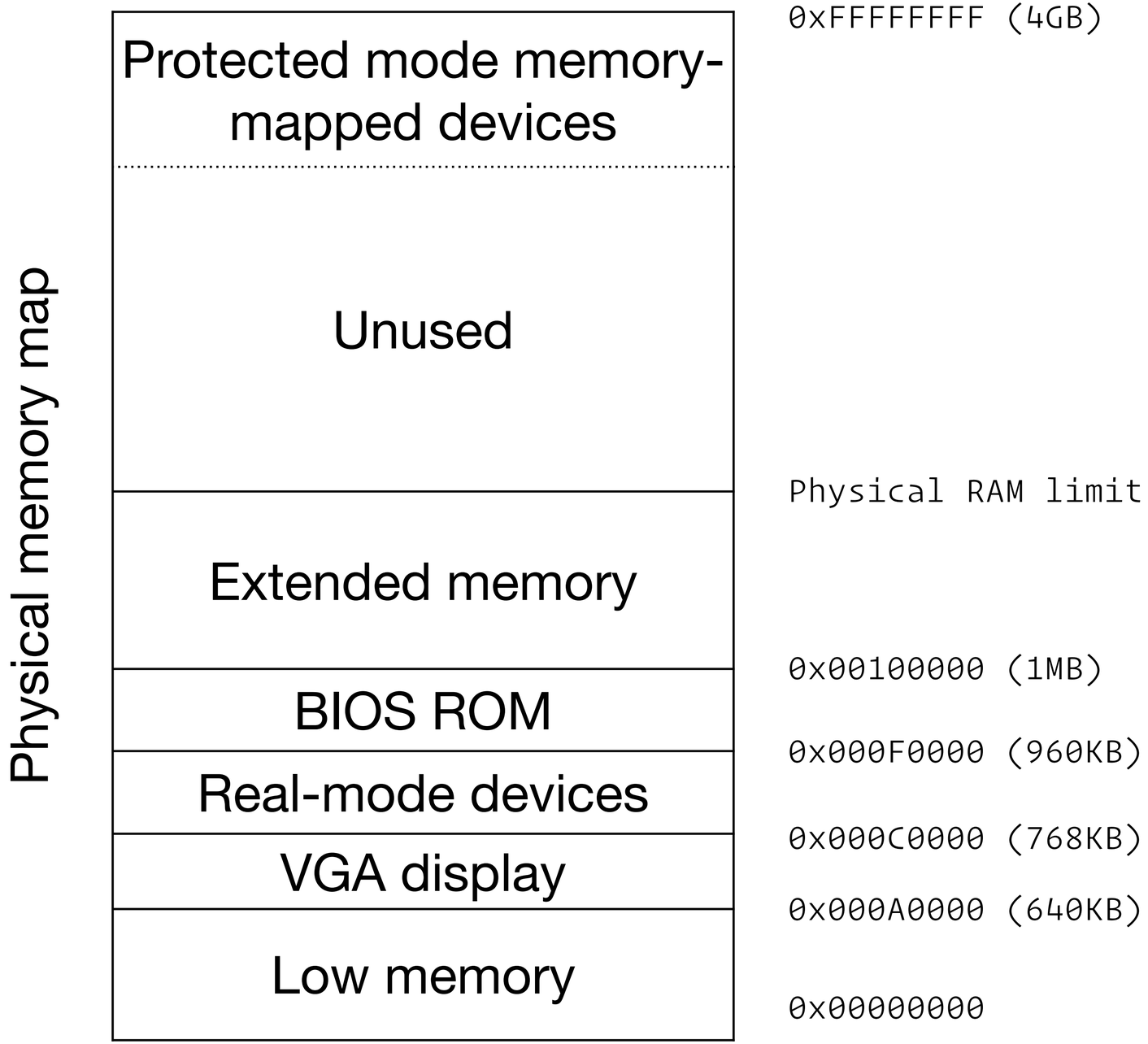


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

§ PC Architecture

What else?

- Memory + memory layout
- Persistent store (disk)
- Text/graphics display
- Keyboard/Mouse + other I/O devices and controllers
- BIOS, Clock


Physical memory map
Protected mode memory-mapped devices
Unused
Extended memory
BIOS ROM
Real-mode devices
VGA display
Low memory
0xFFFFFFFF (4GB)
Physical RAM limit
0x00100000 (1MB)
0x000F0000 (960KB)
0x000C0000 (768KB)
0x000A0000 (640KB)
0x00000000

Startup & BIOS

- On startup, transfer control to address `FFFF:0000` (real mode)
- BIOS executes power on self test, initializes video card, disk controller, and sets up basic interrupt routines for simple I/O
- If boot drive is found, load boot sector (512 bytes, tagged with ending `0x55AA` marker) from drive at address `0000:7C00`

Bootloader Responsibilities

- Set up minimal execution environment (stack, protected mode)
- Scans disk for kernel image (may load second-stage bootloader to navigate partitions, file system, executable formats, etc.)
- Load kernel image at predetermined location in memory
- Transfer control to kernel

On Bootloaders

- Bootloaders can get very complicated!
- E.g., multistage boot loaders like Linux Loader (LILO) and Grand Unified Bootloader (GRUB) understand file systems and executable file formats
  - Also have scripting support and built-in shells

§ QEMU

Full System Emulator

- Emulates the behavior of a real x86 PC in software
- Simulates physical memory map and I/O devices
- Supports up to 255 CPUs (speed dependent on host machine)
- Simple to debug, and won’t break your actual OS!
  - Can connect to GDB to “step” through instructions

The QEMU PC System emulator simulates the following peripherals:

- i440FX host PCI bridge and PIIX3 PCI to ISA bridge
- Cirrus CLGD 5446 PCI VGA card or dummy VGA card with Bochs VESA extensions (hardware level, including all non standard modes).
- PS/2 mouse and keyboard
- 2 PCI IDE interfaces with hard disk and CD-ROM support
- Floppy disk
- PCI and ISA network adapters
- Serial ports
- IPMI BMC, either and internal or external one
- Creative SoundBlaster 16 sound card
- ENSONIQ AudioPCI ES1370 sound card
- Intel 82801AA AC97 Audio compatible sound card
- Intel HD Audio Controller and HDA codec
- Adlib (OPL2) - Yamaha YM3812 compatible chip
- Gravis Ultrasound GF1 sound card
- CS4231A compatible sound card
- PCI UHCI, OHCI, EHCI or XHCI USB controller and a virtual USB-1.1 hub.

SMP is supported with up to 255 CPUs.

QEMU uses the PC BIOS from the Seabios project and the Plex86/Bochs LGPL VGA BIOS.

§ Demo