CSMC 412

Operating Systems

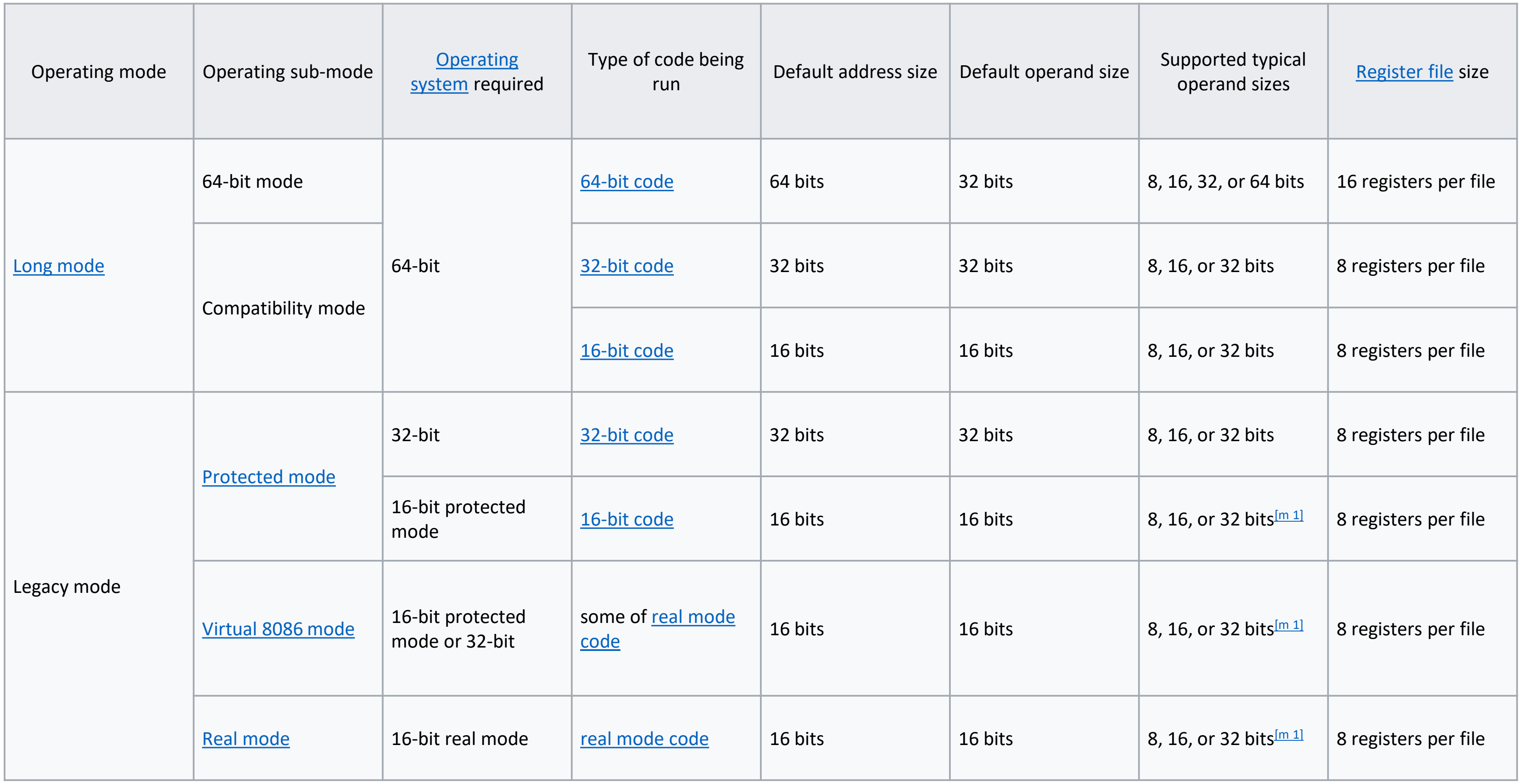
Prof. Ashok K Agrawala

Intel x86 Architecture

Contents

- Intel x86 Architecture
  - Overview
  - Register
  - Instruction
  - Memory Management
  - Interrupt and Exception
  - Task Management
  - Input/Output
  - Stack Manipulation
  - Summary

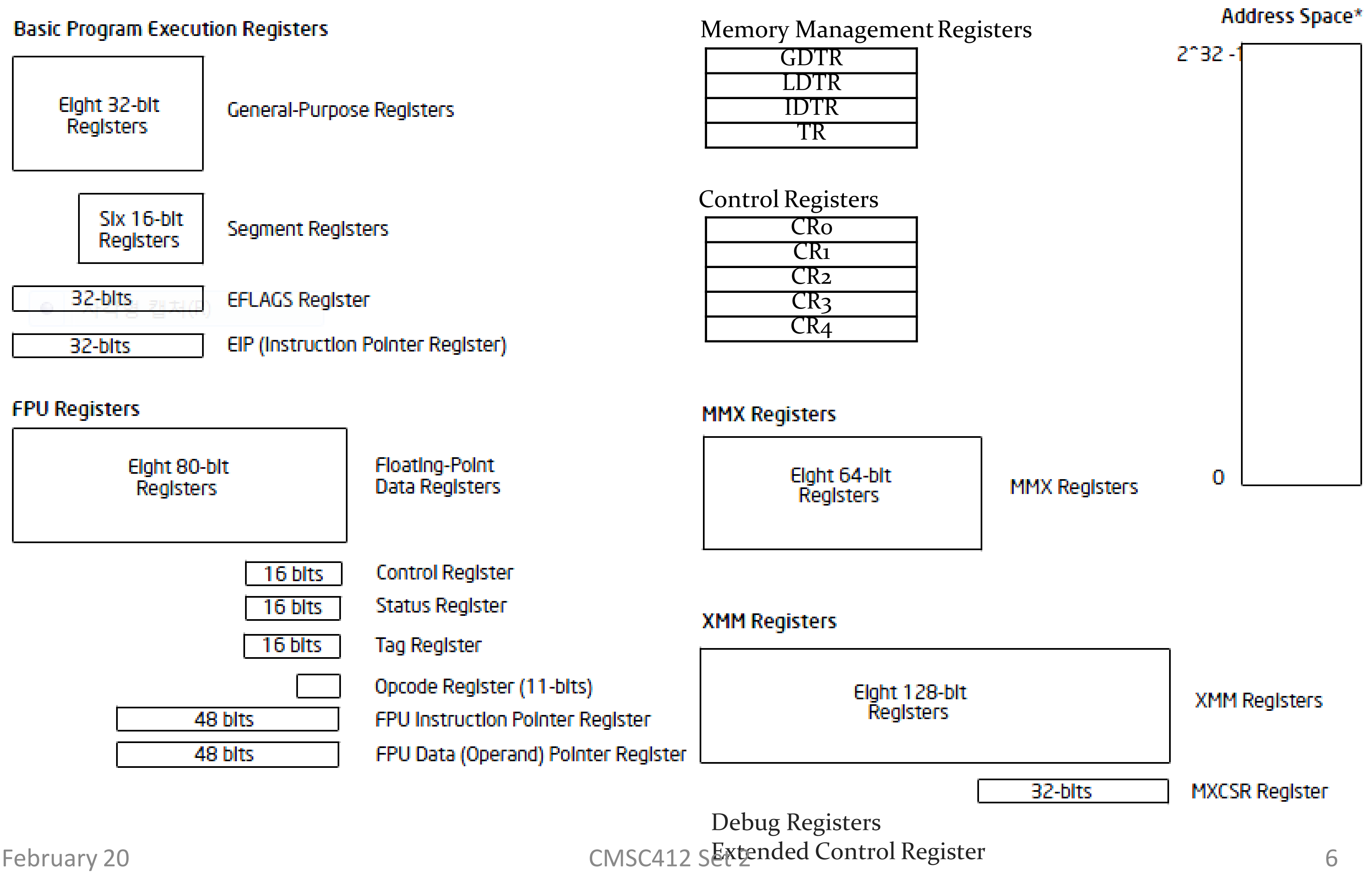
X 86 vs x64



Operating mode	Operating sub-mode	Operating system required	Type of code being run	Default address size	Default operand size	Supported typical operand sizes	Register file size
Long mode	64-bit mode	64-bit	64-bit code	64 bits	32 bits	8, 16, 32, or 64 bits	16 registers per file
	Compatibility mode		32-bit code	32 bits	32 bits	8, 16, or 32 bits	8 registers per file
			16-bit code	16 bits	16 bits	8, 16, or 32 bits	8 registers per file
Legacy mode	Protected mode	32-bit	32-bit code	32 bits	32 bits	8, 16, or 32 bits	8 registers per file
		16-bit protected mode	16-bit code	16 bits	16 bits	8, 16, or 32 bits[m 1]	8 registers per file
	Virtual 8086 mode	16-bit protected mode or 32-bit	some of real mode code	16 bits	16 bits	8, 16, or 32 bits[m 1]	8 registers per file
	Real mode	16-bit real mode	real mode code	16 bits	16 bits	8, 16, or 32 bits[m 1]	8 registers per file

Intel x86 Architecture
: Overview

Basic Execution Environment.



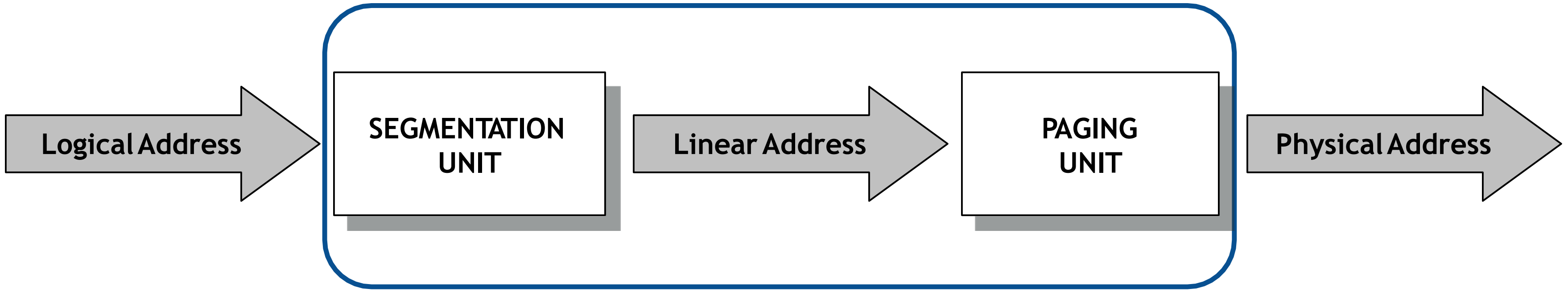
Operation Mode

- Protected mode
  - This mode is the native state of the processor.
  - Support virtual-8086 mode to execute “real-address mode” 8086 software in a protected, multi-tasking environment.
  - Segmentation, 32bit addressing

- Real mode
  - This mode implements the programming environment of the Intel 8086 processor with extensions (such as the ability to switch to protected or system management mode).
  - The processor is placed in real-address mode following power-up or a reset.
  - 16bit mode, Segmentation, 20bit addressing

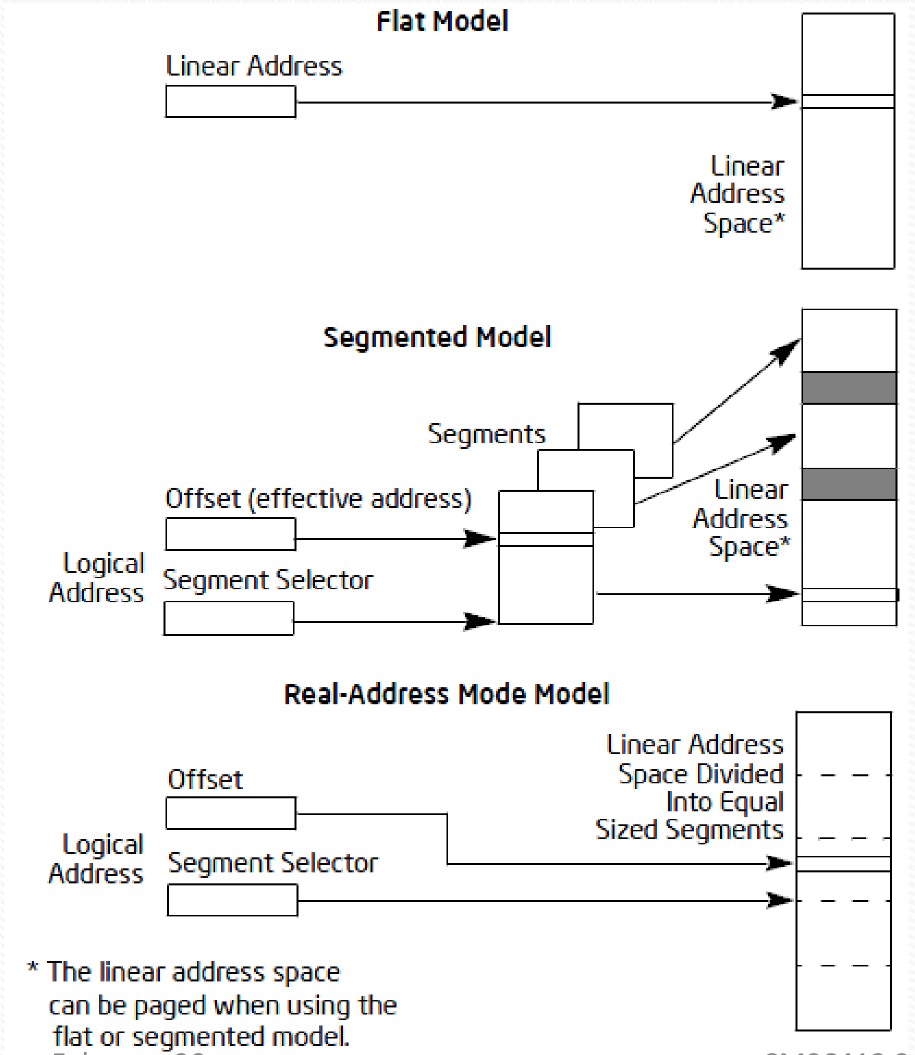
Memory Addresses

- Logical Address
  - Included in the machine language instruction
  - the address of an operand or of an instruction
  - Consists of segment(16bit) and offset(32bit)
    - offset - distance from the start of the segment to the actual address
- Linear Address (known as virtual address)
  - A single 32-bit unsigned integer
  - Range: 0x00000000~0xffffffff(4GB)
- Physical Address
  - Used to address memory cells included in memory chips
  - Represented as 32-bit unsigned integer



MMU(Memory Management Unit)

Memory Models



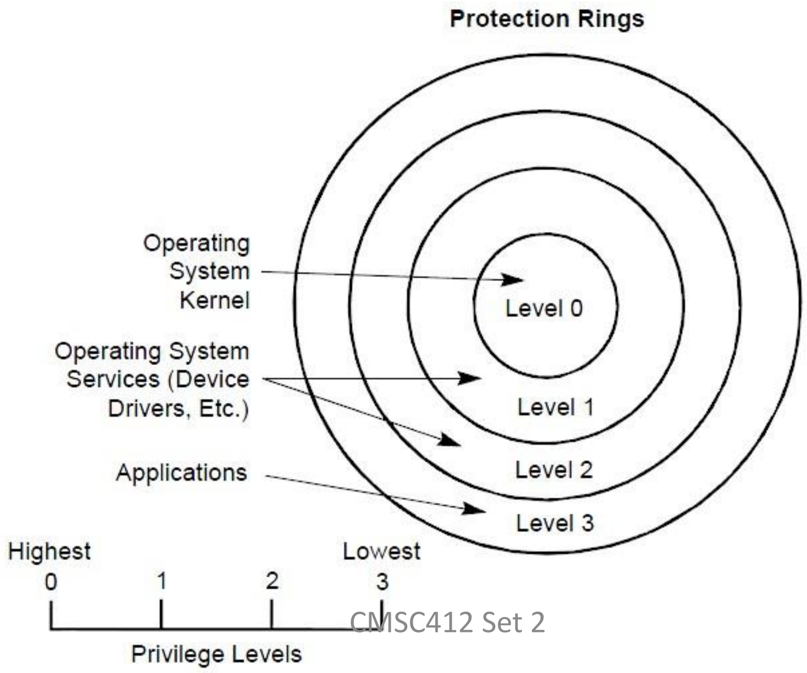
•No segmentation
•Code, Data, stacks are all contained in this address space.
• 32 bit addressing

•Code, Data, stacks are typically contained in separate segments for better isolation.
•32 bit addressing (32 bit offset, 16 bit seg. selector)

•Compatibility mode for 8086 processor.
•20 bit addressing (16 bit offset, 16 seg. selector)

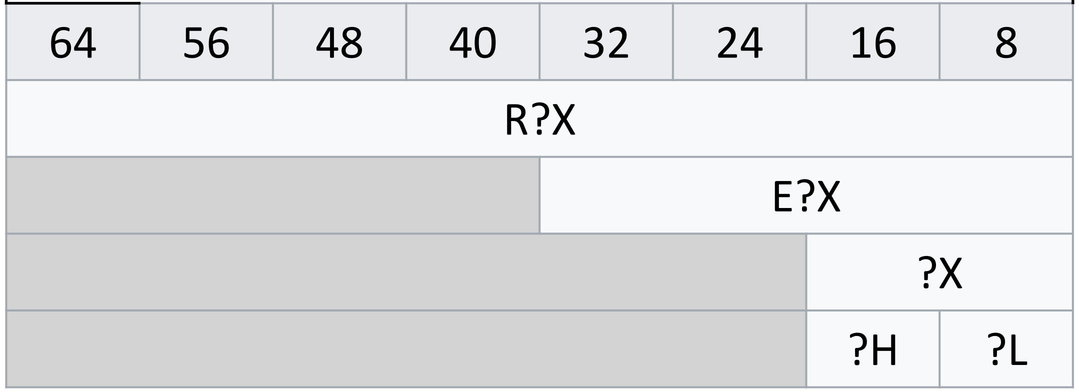
Privilege Level

- Code modules in lower privilege segments can only access modules operating at higher privilege segments by means of a tightly controlled and protected interface called **gate**.
- Attempts to access higher privilege segments without going though a protection gate and without having sufficient access rights causes a general-protection exception(#GP) to be generated.

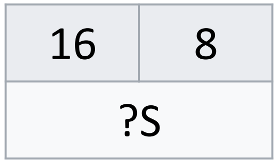


Intel x86 Architecture
: Register

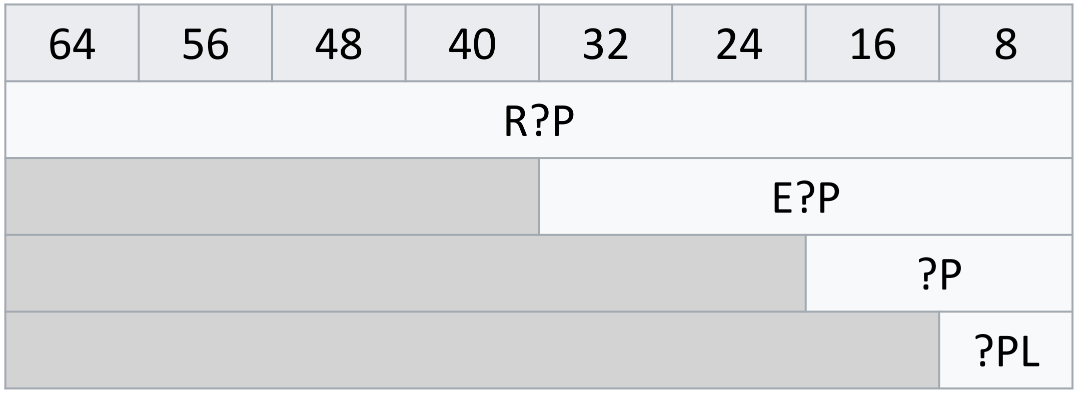
General Purpose Registers (A, B, C and D)



Segment Registers
C,D,S,E,F and G



Pointer Registers (S and B)



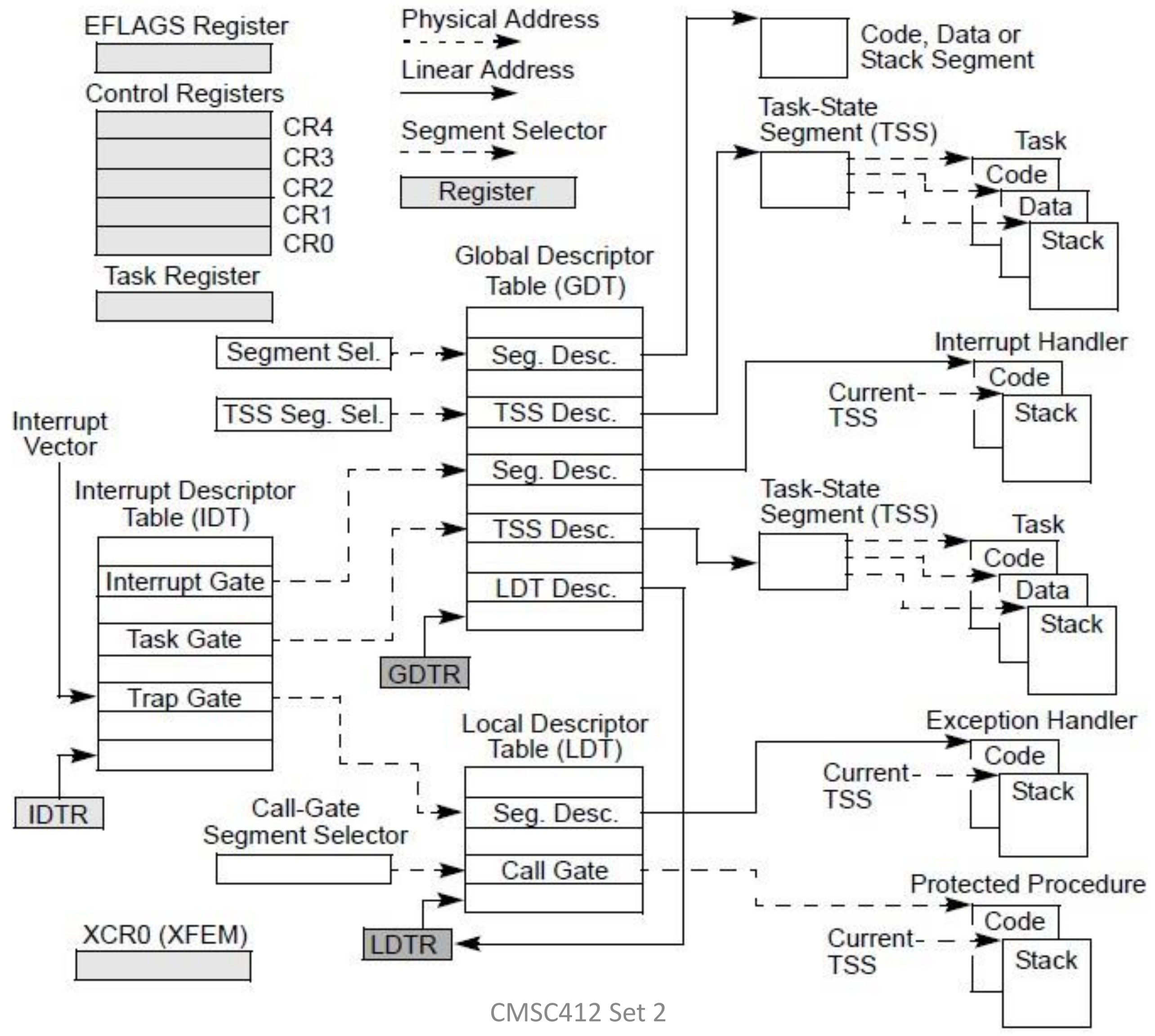
Index Registers (S and D)

64	56	48	40	32	24	16	8
R?I							
▨	▨	▨	▨	E?I			
▨	▨	▨	▨	▨	▨	?I	
▨	▨	▨	▨	▨	▨	▨	?IL

Instruction Pointer Register (I)

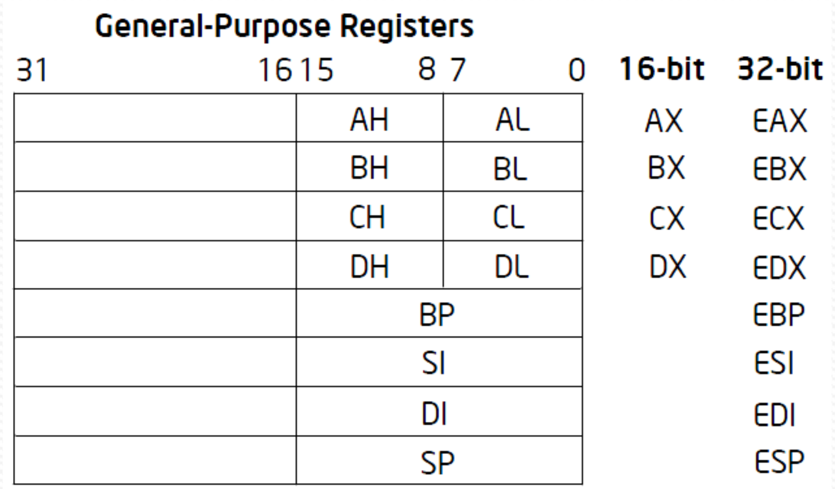
64	56	48	40	32	24	16	8
RIP							
▨	▨	▨	▨	EIP			
▨	▨	▨	▨	▨	▨	IP	

System Level Registers and Data Structures



Basic Program Execution Registers

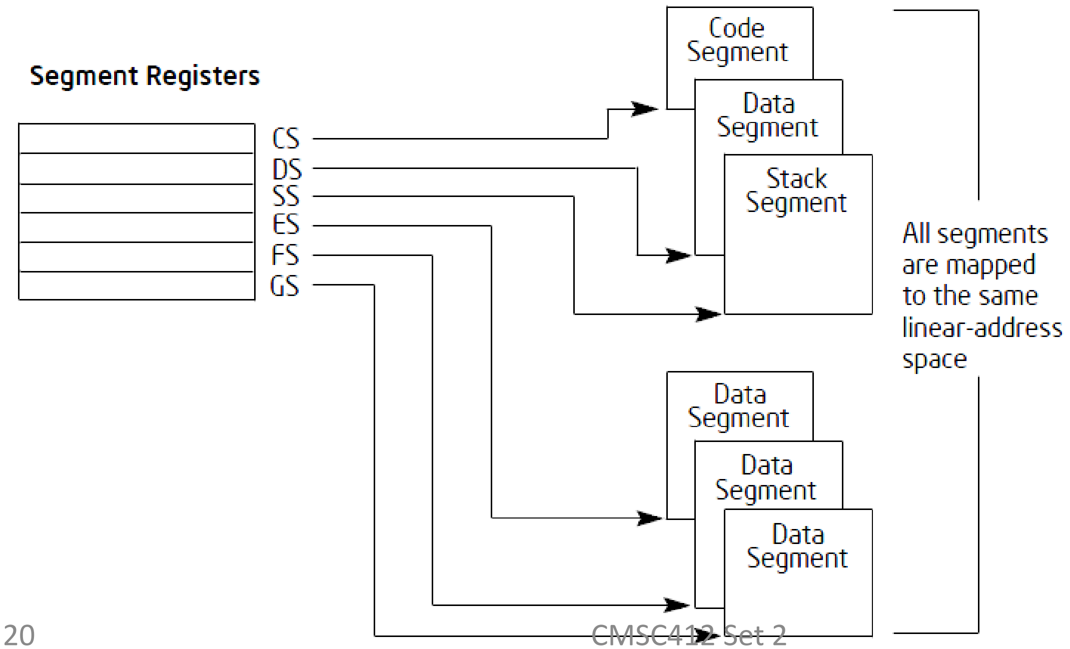
- General-Purpose Registers
  - For storing operands and pointers
  - ESP – Stack pointer in the SS segment
  - EBP – Frame pointer on the stack
  - ECX – Counter for string and loop operations
  - ESI – Source pointer for string operations
  - EDI – Destination pointer for string operations.


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

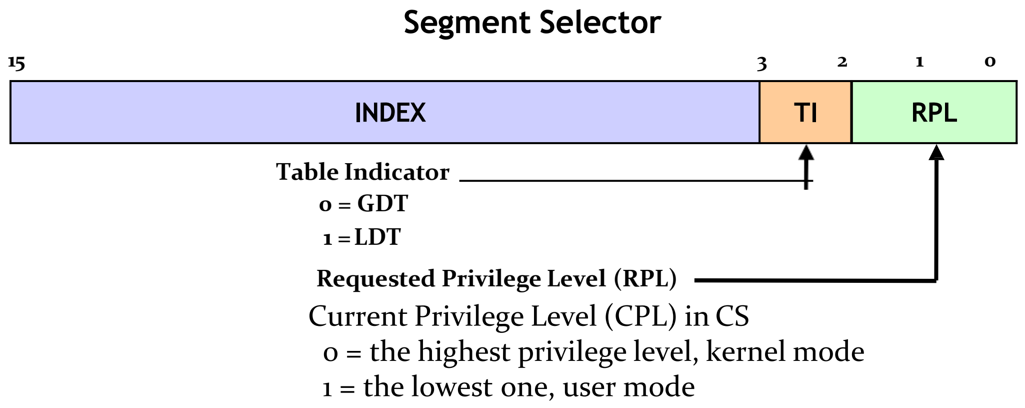
Basic Program Execution Registers (cont'd)

- Segment Registers
  - It holds 16-bit **segment selectors**. A segment selector is a special pointer that identifies a segment in memory.
  - To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register.



Basic Program Execution Registers (cont'd)

- Segment Selectors(16bit)
  - Index(13bit) – Segment Descriptor entry in GDT, LDT
  - TI (Table Indicator)(1bit)
    - 0 : Segment Descriptor is stored in GDT
    - 1 : Segment Descriptor is stored in LDT
  - RPL(2bit) – Requested Privilege Level (CPL in CS)

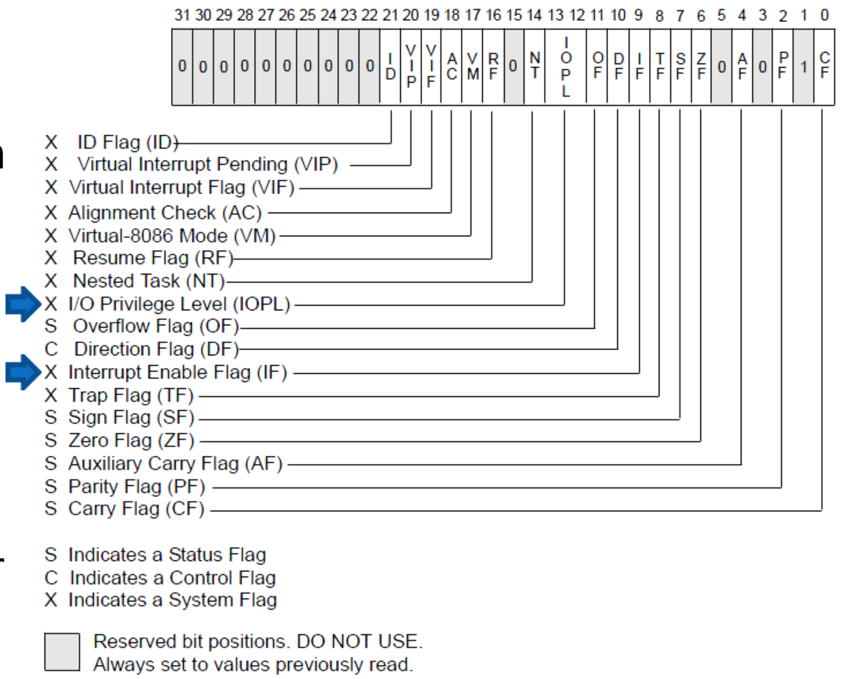

Segment Selector
15
3
2
1
0
INDEX
TI
RPL
Table Indicator
0 = GDT
1 = LDT
Requested Privilege Level (RPL)
Current Privilege Level (CPL) in CS
0 = the highest privilege level, kernel mode
1 = the lowest one, user mode

Basic Program Execution Registers (cont'd)

- Default Segment Selection Rules
  - CS : Instructions
    - All instruction fetches
  - SS : Stack
    - All stack pushes and pops. Any memory reference which uses the ESP or EBP register as a base register.
  - DS : Local Data
    - All data references, except when relative to stack or string destination.
  - ES : Destination Strings
    - Destination of string instructions, eg. MOVS.

Basic Program Execution Registers (cont'd)

- EFLAGS Register
  - The EFLAGS register report on the status of the program being executed and allows limited (application program level) control of the processor.
  - Some of the flags in the EFLAGS register can be modified directly, using special purpose instructions. There are no instructions that allow the whole register to be examined or modified directly.
  - When suspending a task, the processor automatically saves the state of the EFLAGS register in the task statement segment(TSS) for the task being suspended. When binding itself to a new task, the task processor loads the EFLAGS register with data from the new task's TSS.



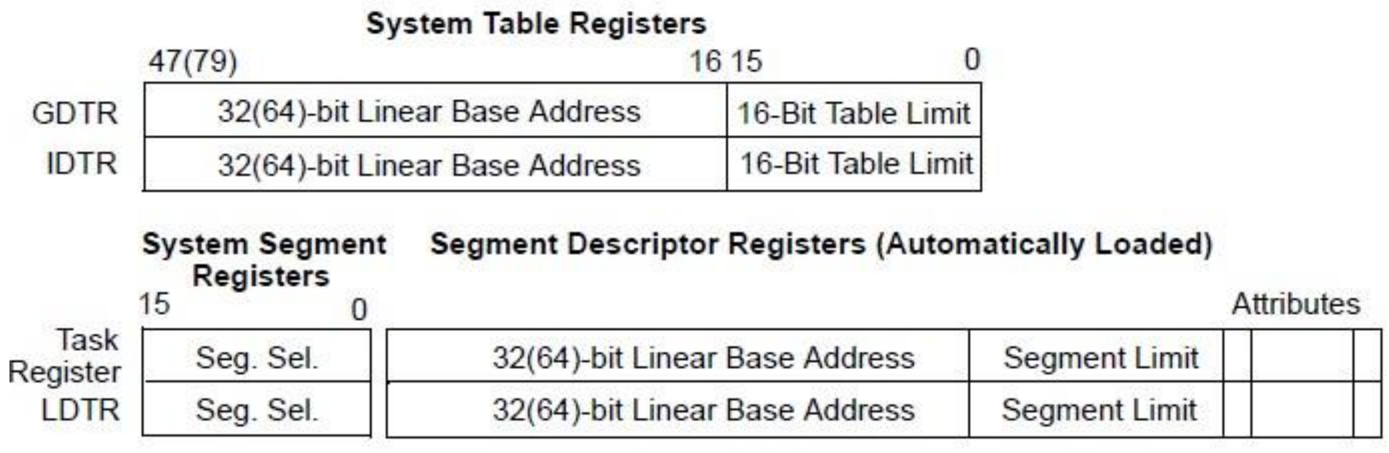
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE. Always set to values previously read.

Basic Program Execution Registers (cont'd)

- EIP (Instruction Pointer)
  - The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed.
  - It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions.

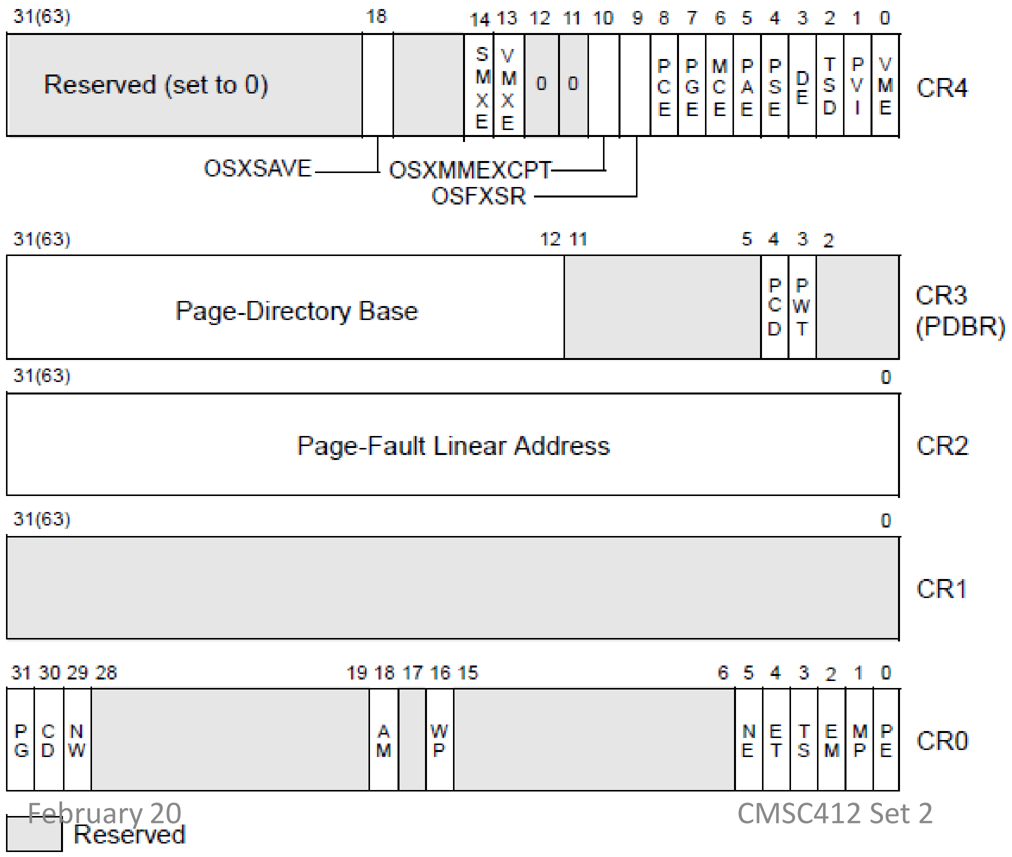
Memory Management Registers

- The processor provides four memory-management registers (GDTR, LDTR, IDTR an TR) that specify the locations of the data structures which control segmented memory management. Special instructions are provided for loading and storing these registers.



Control Registers

- Control registers determine operating mode of the processor and the characteristics of the currently running task.



- **CR4**: Contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities.

- **CR3**: physical address of the page directory

- **CR2**: page fault linear address

- **CR0**: System control flag
- PE flag
  - 0/1 : real mode/protected mode
- PG flag
  - 0: linear address == physical address
  - 1 : paging enable
- TS flag
- It causes the CPU to trap (int 7) if the floating point unit is used. It is used to restore FPU state lazily after a task switch.

Intel x86 Architecture
: Instruction

General Purpose Instructions

- The general-purpose instructions perform basic data movement, arithmetic, logic, program flow, and string operations that programmers commonly use to write application and system software to run.
  - Data Transfer Instructions
    - MOV, CMOV, PUSH, POP, XCHG, …
  - Binary Arithmetic Instructions
    - ADD, SUB, INC, DEC, …
  - Decimal Arithmetic Instructions
  - Logical Instructions
    - AND, OR, XOR, …
  - Shift and Rotate Instructions
    - SAR, SAL, ROR, ROL, …
  - Bit and Byte Instructions
    - BT, SET, TEST, …
  - Control Transfer Instructions
    - JMP, CALL, INT, RET, IRET, INTO, BOUND, …
  - String Instructions
    - MOVS, LODS, CMPS, …
  - IO Instructions
    - IN, OUT, …
  - EFLSGS Control Instructions
    - STC, CLC, …
  - Segment Register Instructions
    - LDS, LES, LFS, LGS, LSS
  - Misc. Instructions
    - NOP, …

System Instructions

- The following system instructions are used to control those functions of the processor that are provided to support for operating systems and executives.

  - Manipulate memory management register
    - LGDT, LLDT, LTR, LIDT, SGDT, SLDT, SIDT, STR
  - Load and store control registers
    - MOV {CR0~CR4}, CLI, STI
  - Invalidate Cache and TLB
    - INVD, WBINVD, INVLPG
  - Performance monitoring
    - RDPMC, RDTSC, RDTSCP
  - Fast System Call
    - SYSENTER, SYSEXIT
  - Pointer Validation
    - LAR, LSL, VERR, VERW, ARPL
  - Misc.
    - LOCK, CLTS, HLT

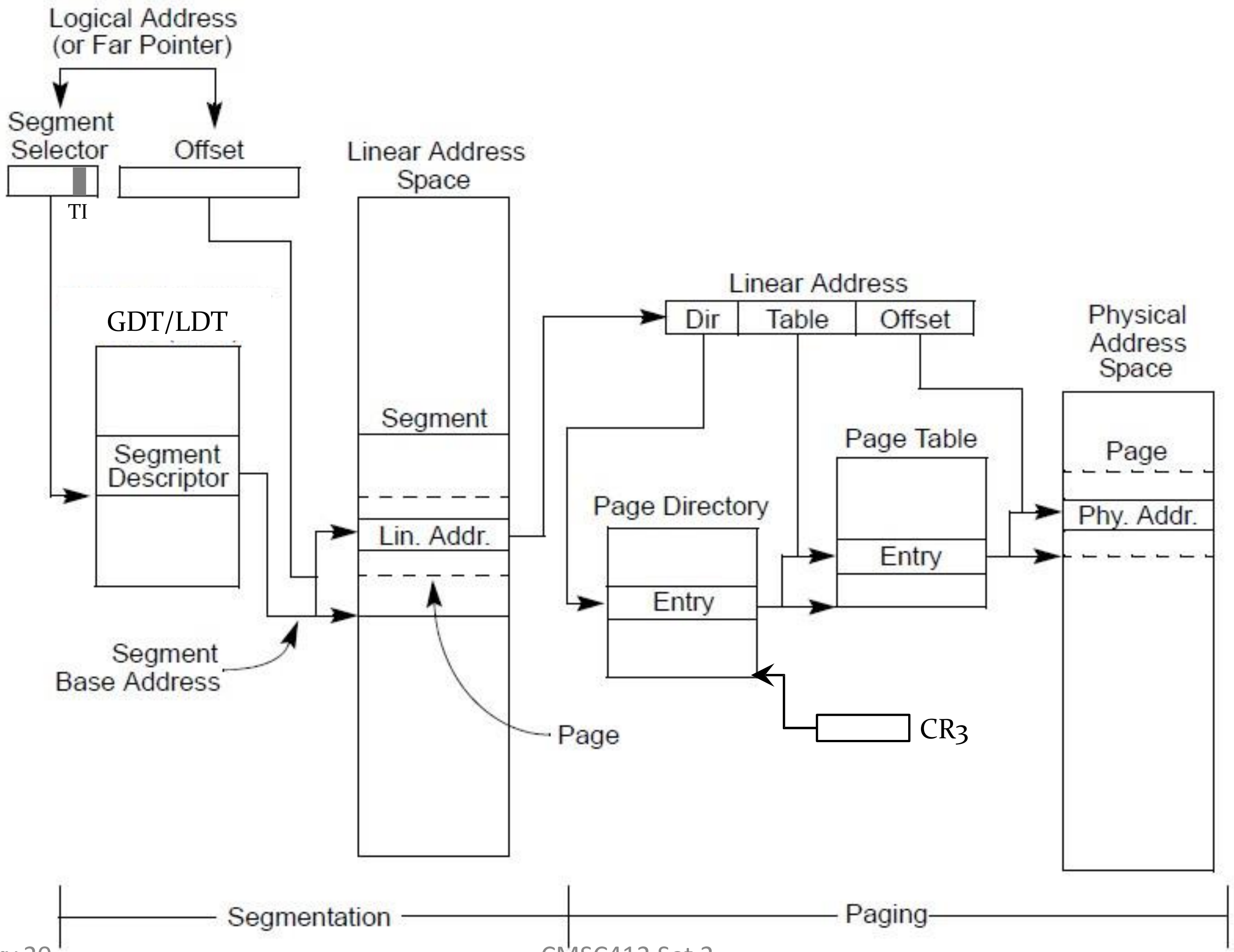
Privileged instructions in red which can be executed only in ring 0.

Intel x86 Architecture
: Memory Management

Segmentation & Paging

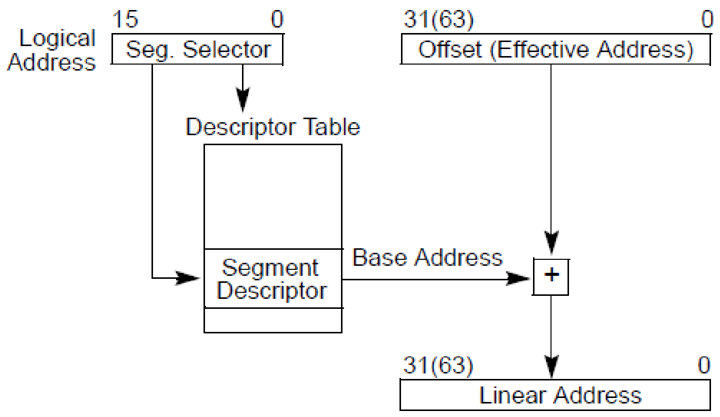
- Segmentation
  - provides a mechanism for dividing the processor’s linear address space into smaller protected address spaces called segments.
  - translate logical address to linear address
- Paging
  - provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program’s execution environment are mapped into physical memory as needed. It can also be used to provide isolation between multiple tasks.
  - translate linear address to physical address

Segmentation & Paging (cont'd)

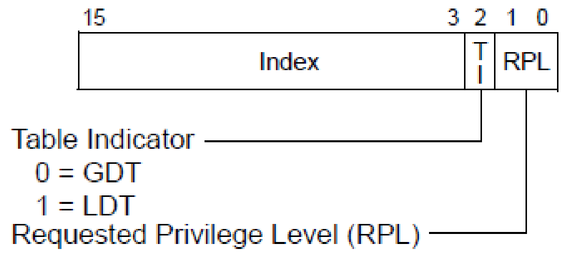


Segmentation

- Logical address to linear address translation



- Segment Selector

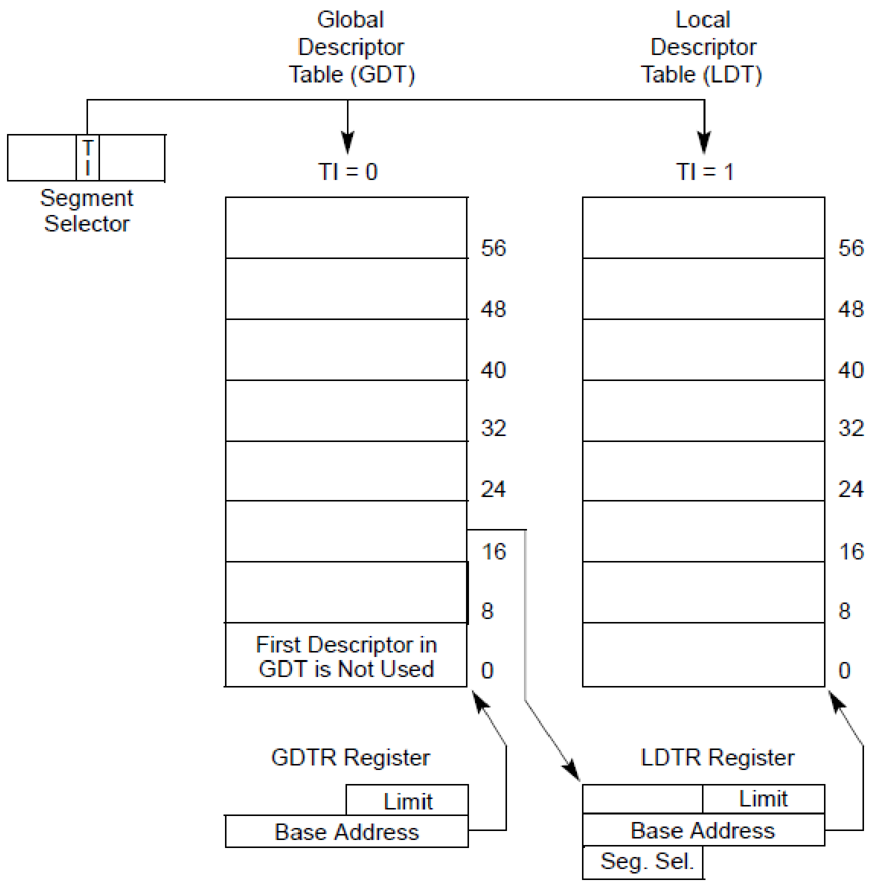


Current Privilege Level (CPL) in CS
0 = the highest privilege level, kernel mode
1 = the lowest one, user mode

•To reduce address translation time and coding complexity, the processor provides registers for holding up to 6 segment selectors.
• CS, SS, DS, ES, FS, GS

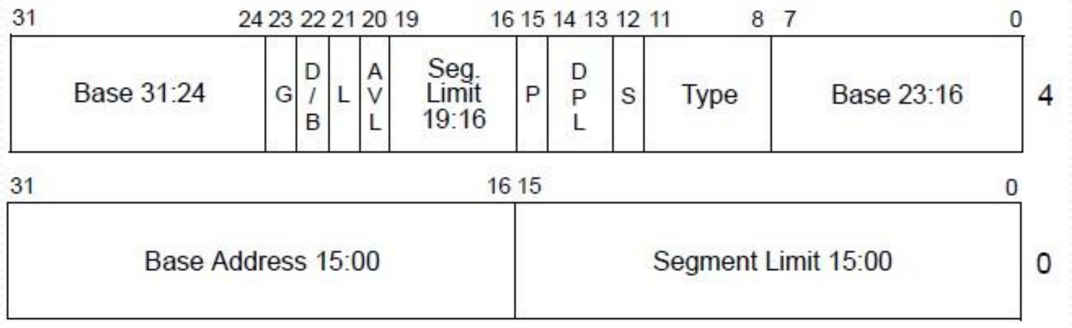
Segmentation (cont'd)

- Global and local descriptor tables



Segmentation (cont'd)

- Segment Descriptors
  - It a s data structure in a GDT or LDT that provides the processor with the size and location of a segment, as well as access control and status information.

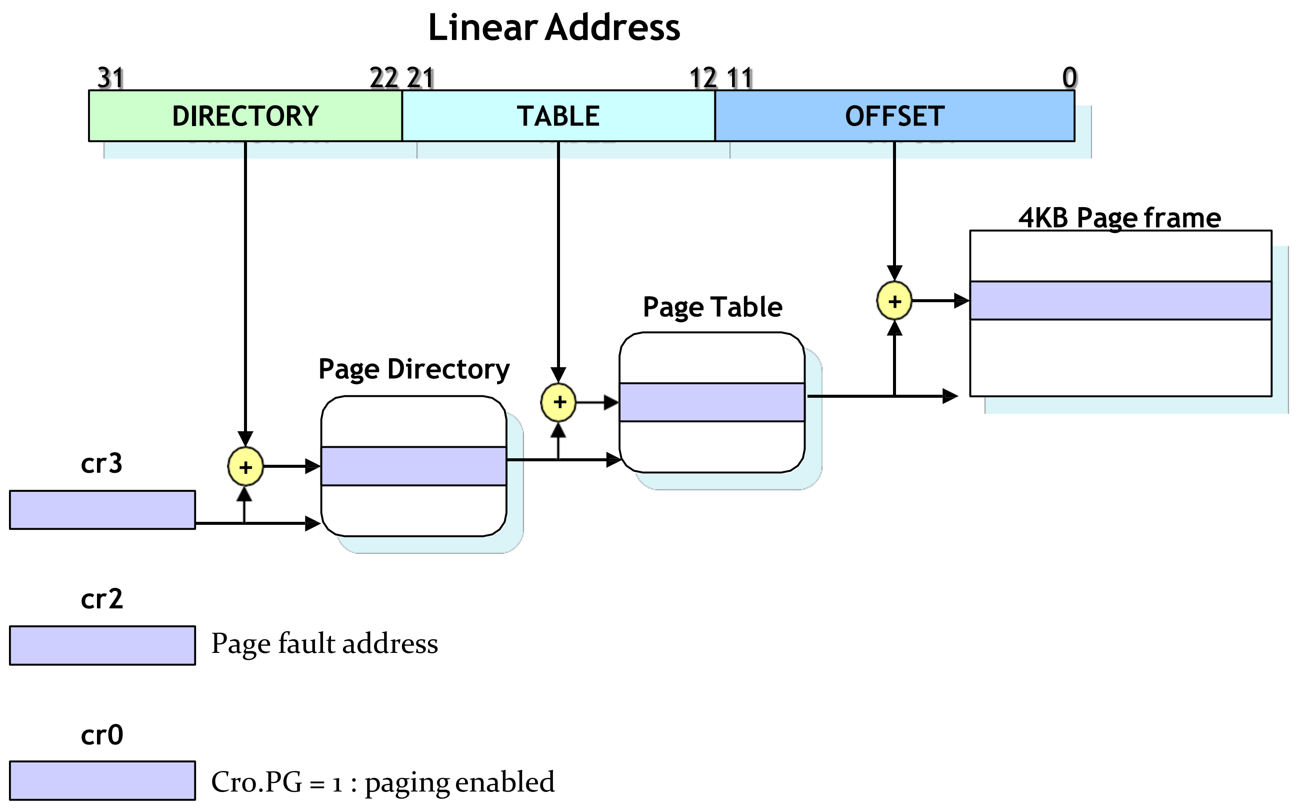


L	— 64-bit code segment (IA-32e mode only)
AVL	— Available for use by system software
BASE	— Segment base address
D/B	— Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL	— Descriptor privilege level
G	— Granularity
LIMIT	— Segment Limit
P	— Segment present
S	— Descriptor type (0 = system;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

11: Code/Data
9: Write-enable
10: Expansion-direction
8: Accessed

Paging

- Linear address to physical address translation



Paging (cont’d)

- Page Directories and Page Tables entry field
  - Available for system programmer’s use
  - Global page
  - Page size(0 indicates 4 Kbytes)
  - Reserved(set to 0) / Dirty
  - Accessed
  - Cache disabled
  - Write-through
  - User/Supervisor
  - Read/Write
  - Present

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
Page-Table Base Address	Avail.	G	PS	0	A	PCD	PWT	U/S	R/W	P

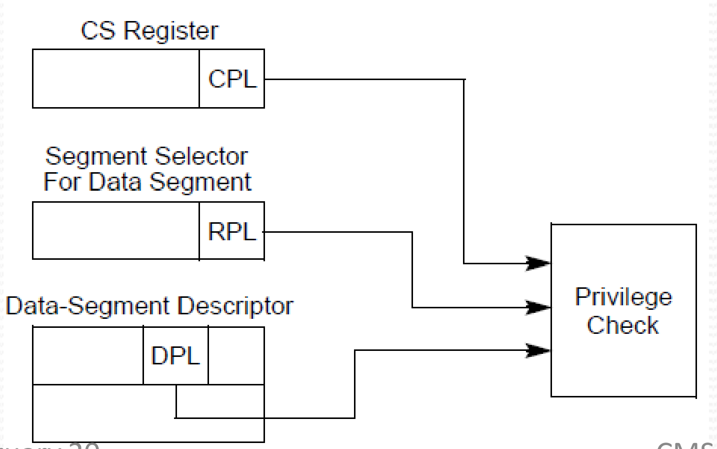
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0
Page Base Address	Avail.	G	0	D	A	PCD	PWT	U/S	R/W	P

Protection

- Privilege Level Checking
  - The segment-protection mechanism recognizes 4 privilege levels, numbers from 0 to 3. The greater numbers mean lesser privileges.
  - Privilege levels are checked when the segment selector of a segment descriptor is loaded into a segment register.
  - When the processor detects a privilege level violation, it generates a general-protection exception(#GP).

Protection (cont'd)

- To carry out privilege-level checks between code segments and data segments, the processor recognizes the following three types of privilege levels:
  - Current Privilege Level (CPL)
    - The privilege level of the currently executing task
    - It is equal to the privilege level of the code segment from which instructions are being fetched.
  - Descriptor Privilege Level (DPL)
    - The privilege level of the segment of gate.
  - Requested Privilege Level (RPL)
    - It is an override privilege level that is assigned to segment selectors.



* Privilege check for dataaccess

Intel x86 Architecture
: Interrupt and Exception

Gate

- The architecture also defines a set of special descriptors called gates (call gates, interrupt gates, trap gates, and task gates). These provide protected gateways to system procedures and handlers that may operate at a different privilege level than application programs and most procedures.

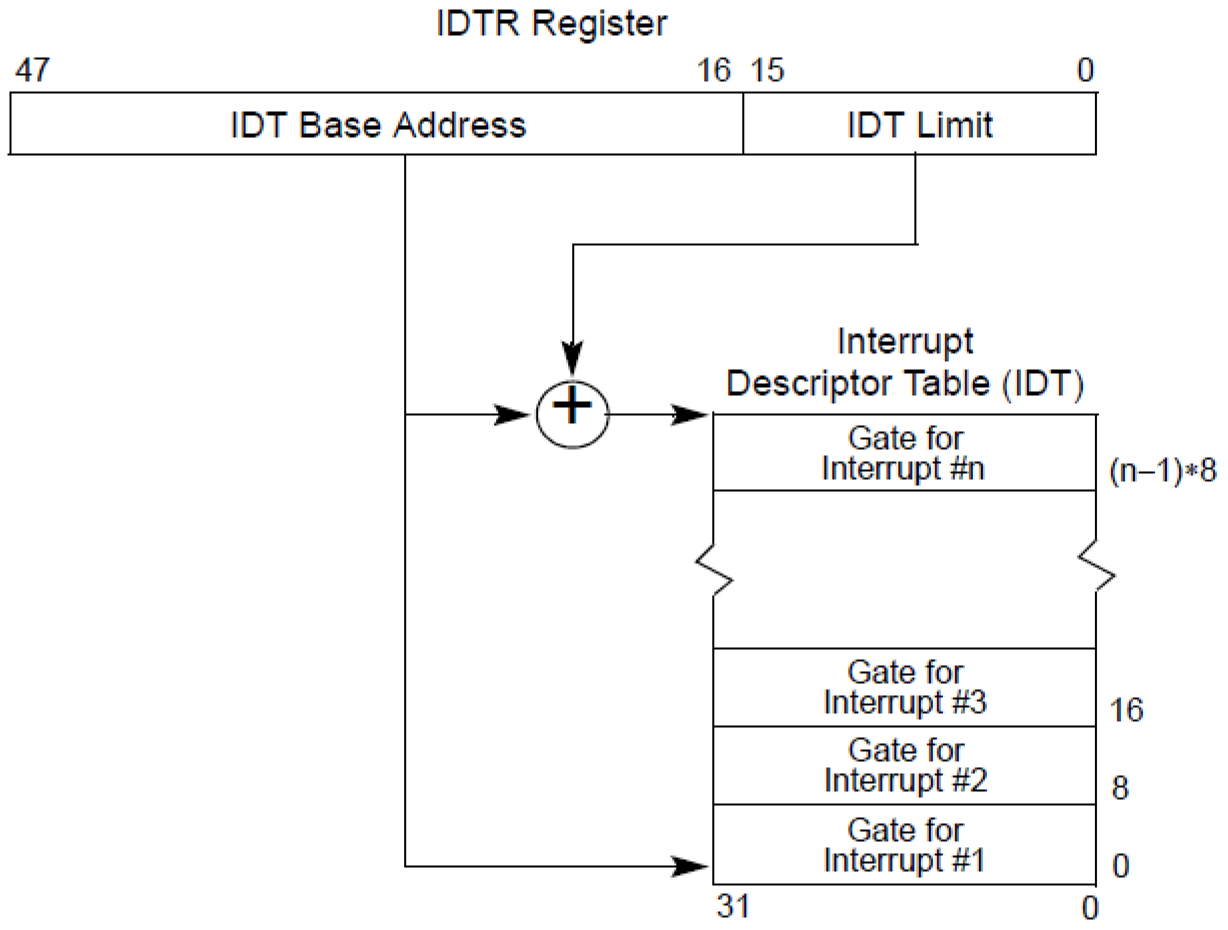
- For example, a CALL to a call gate can provide access to a procedure in a code segment that is at the same or a numerically lower privilege level (more privileged) than the current code segment. To access a procedure through a call gate, the calling procedure supplies the selector for the call gate. The processor then performs an access rights check on the call gate, comparing the CPL with the privilege level of the call gate and the destination code segment pointed to by the call gate.

- If access to the destination code segment is allowed, the processor gets the segment selector for the destination code segment and an offset into that code segment from the call gate. If the call requires a change in privilege level, the processor also switches to the stack for the targeted privilege level. The segment selector for the new stack is obtained from the TSS for the currently running task. Gates also facilitate transitions between 16-bit and 32-bit code segments, and vice versa.

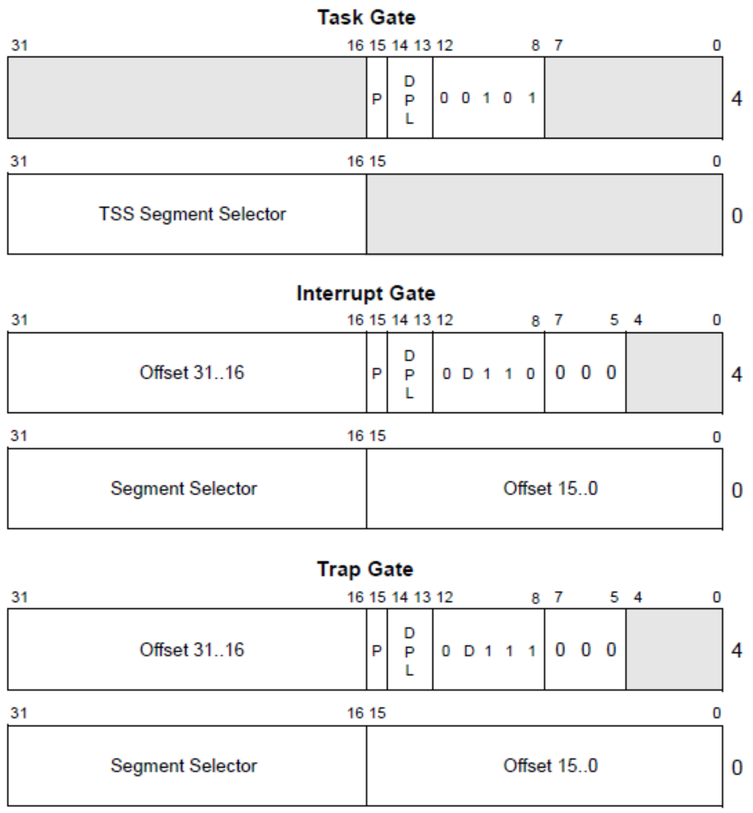
Interrupt and Exception handling

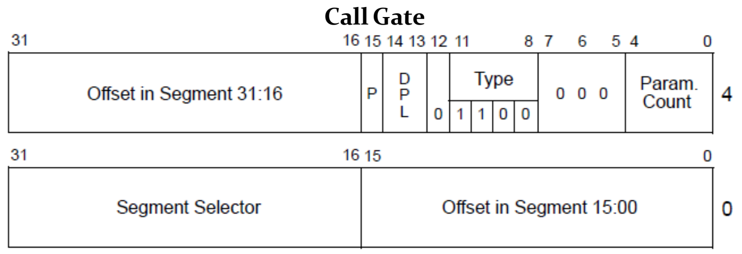
- External interrupts, software interrupts and exceptions are handled through the interrupt descriptor table (IDT). The IDT stores a collection of gate descriptors that provide access to interrupt and exception handlers. The linear address for the base of the IDT is contained in the IDT register (IDTR).
- Gate descriptors in the IDT can be interrupt, trap, or task gate descriptors. To access an interrupt or exception handler, the processor first receives an interrupt vector (interrupt number) from internal hardware, an external interrupt controller, or from software by means of an INT, INTO, INT 3, or BOUND instruction.
- The interrupt vector provides an index into the IDT. If the selected gate descriptor is an interrupt gate or a trap gate, the associated handler procedure is accessed in a manner similar to calling a procedure through a call gate. If the descriptor is a task gate, the handler is accessed through a task switch.

Relationship of the IDTR and IDT



Gate Descriptor





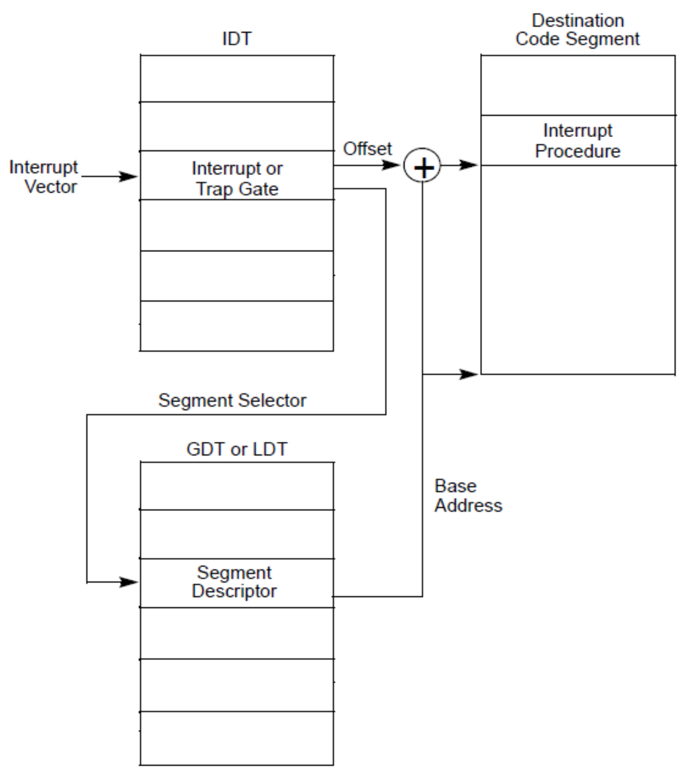
- While transferring control to the proper segment, the processor clears the EFLAGS.IF flag, thus disabling further maskable interrupts.

- While transferring control to the proper segment, the processor does not modify the EFLAGS.IF flag.

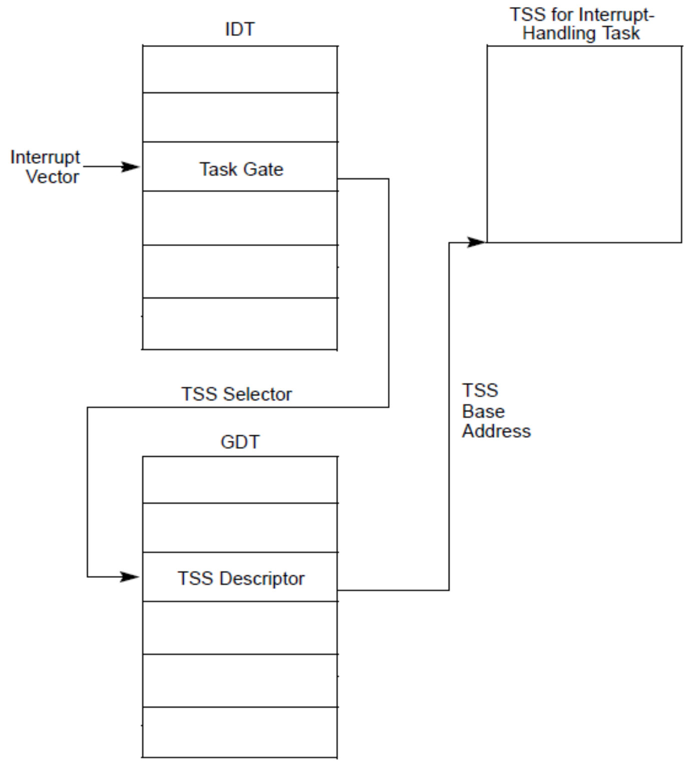
- IDT : Task Gate, Interrupt, Trap Gate
- LDT : Call Gate

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
▨	Reserved

Executing a handler



* Exception or Interrupt Procedure call



* Interrupt Task Switch

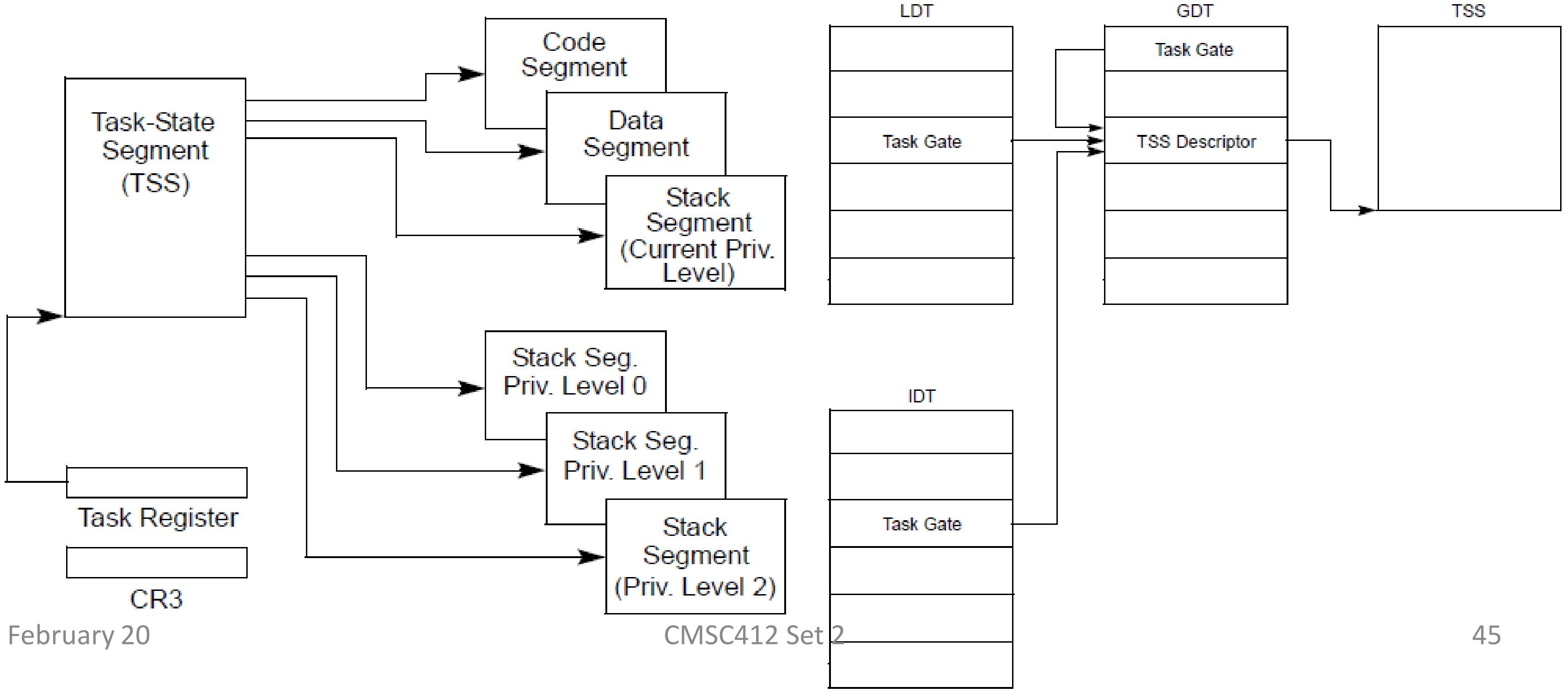
Interrupt and Exception Vectors

- 0 ~ 31 (fixed)
  - Exceptions and nonmaskable interrupts
    - 6: Invalid Opcode
    - 13 : general protection exception
    - 14 : page fault
- 32 ~ 47
  - Maskable interrupts
  - Interrupts caused by IRQs
- 48 ~ 255
  - S/W interrupts
  - Linux uses only one of them,
    - 128 : to implement system calls

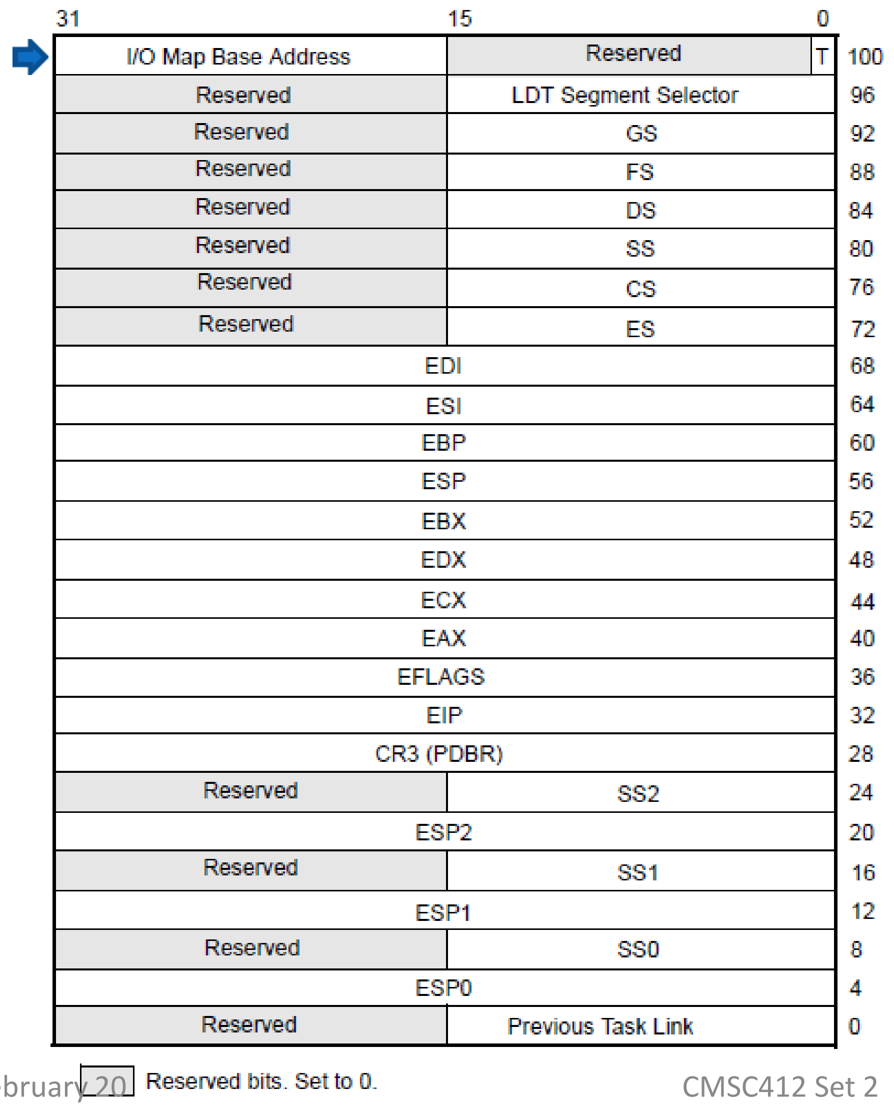
Intel x86 Architecture
: Task Management

Task Structure

- A task is made up of two parts: a task execution space and a task-state segment(TSS).
- A task is identified by the segment selector for its TSS. When a task is loaded into the processor for execution, the segment selector, base address, limit, and segment descriptor attributes for TSS are loaded into the task register.



Task State Segment



31	15	0	
I/O Map Base Address	Reserved	T	100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
Reserved	SS2		24
ESP2			20
Reserved	SS1		16
ESP1			12
Reserved	SS0		8
ESP0			4
Reserved	Previous Task Link		0

▨ Reserved bits. Set to 0.

- •SS0, SS1, SS2
  - Stack Segment for ring 0, 1, 2
- •ESP0, ESP1, ESP2
  - Stack pointer for ring 0, 1, 2

H/W Task Switching

- The processor transfers execution to another task in one of following cases
  - JMP or Call instruction to a procedure located in a different task using far pointer
    - to a TSS descriptor in the GDT.
    - to a task-gate descriptor in the GDT or the current LDT.
  - An interrupt or exception vector points to a task-gate descriptor in the IDT.
  - The current task executes an IRET when the NT flag in the EFLAGS register is set.

H/W Task Switching (cont'd)

- The processor performs the following operations when switching to a new task
  - Obtains the TSS segment selector for the new task.
  - Check that the current (old) task is allowed to switch to the new task. (CPL/DPL/RPL)
  - Saves the state of the current (old) task in the current task's TSS.
  - Loads the task register with the segment selector and descriptor for the new task's TSS.
  - The TSS state is loaded into the processor. This includes the LDTT, CR3, EFLAGS, EIP, the general purpose registers, and the segment selectors.
  - The descriptor associated with the segment selectors are loaded and qualified.

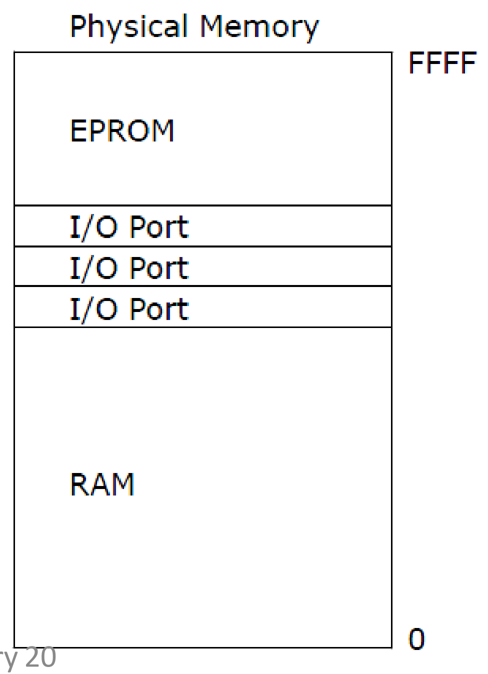
Intel x86 Architecture
: Input/Output

I/O Port Addressing

- The processor permits applications to access I/O ports in either of two ways:
  - Through a separate I/O address space
    - Handled though a set of I/O instructions and a special I/O protection mechanism
    - Writes to I/O ports are guaranteed to be completed before the next instruction in the instruction stream is executed.
  - Through memory-mapped I/O
    - Accessing I/O ports through memory-mapped I/O is handled with the processors general-purpose move and string instructions, with protection provided through segmentation or paging.

I/O Address Space

- The processor's I/O address space is separate and distinct from the physical-memory address space.
- The I/O address space consists of 2^{16} (64K) individually addressable 8-bit I/O ports, numbered 0 through FFFFH.
- I/O port addresses 0F8H through 0FFH are reserved.



I/O port protection

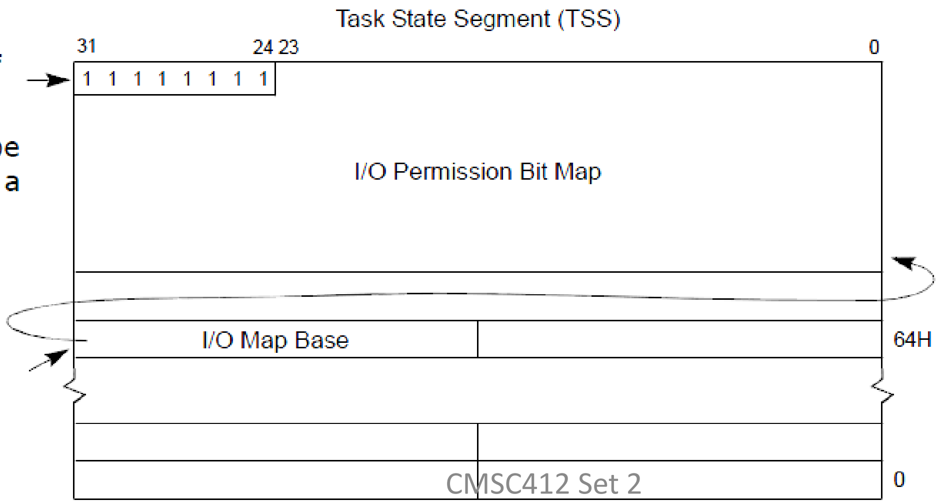
- When accessing I/O ports through the I/O address space, two protection devices control access:
  - I/O instructions can be executed only if the current privilege level (CPL) of the program or task currently executing is less than or equal to the IOPL.
  - Any attempt by a less privileged program or task to use an I/O sensitive instruction results in a general-protection exception (#GP) being signaled.
  - The I/O permission bit map in the TSS can be used to modify the effect of the IOPL on I/O sensitive instructions, allowing access to some I/O ports by less privileged programs or tasks.
- When accessing memory-mapped I/O ports,
  - the normal segmentation and paging protection also affect access to I/O ports.

I/O port protection (cont'd)

- The I/O permission bit map is a device for permitting limited access to I/O ports by less privileged programs or tasks.
  - If in protected mode and the CPL is less than or equal to the current IOPL, the processor allows all I/O operations to proceed.
  - If the CPL is greater than the IOPL, the processor checks the I/O permission bit map to determine if access to a particular I/O port is allowed.
- The I/O permission bit map is located in the TSS for the currently running task or program.
  - Each bit in the map corresponds to an I/O port byte address.

Last byte of bit

map must be followed by a



I/O map base must not exceed DFFFH.

Intel x86 Architecture
: Stack Manipulation

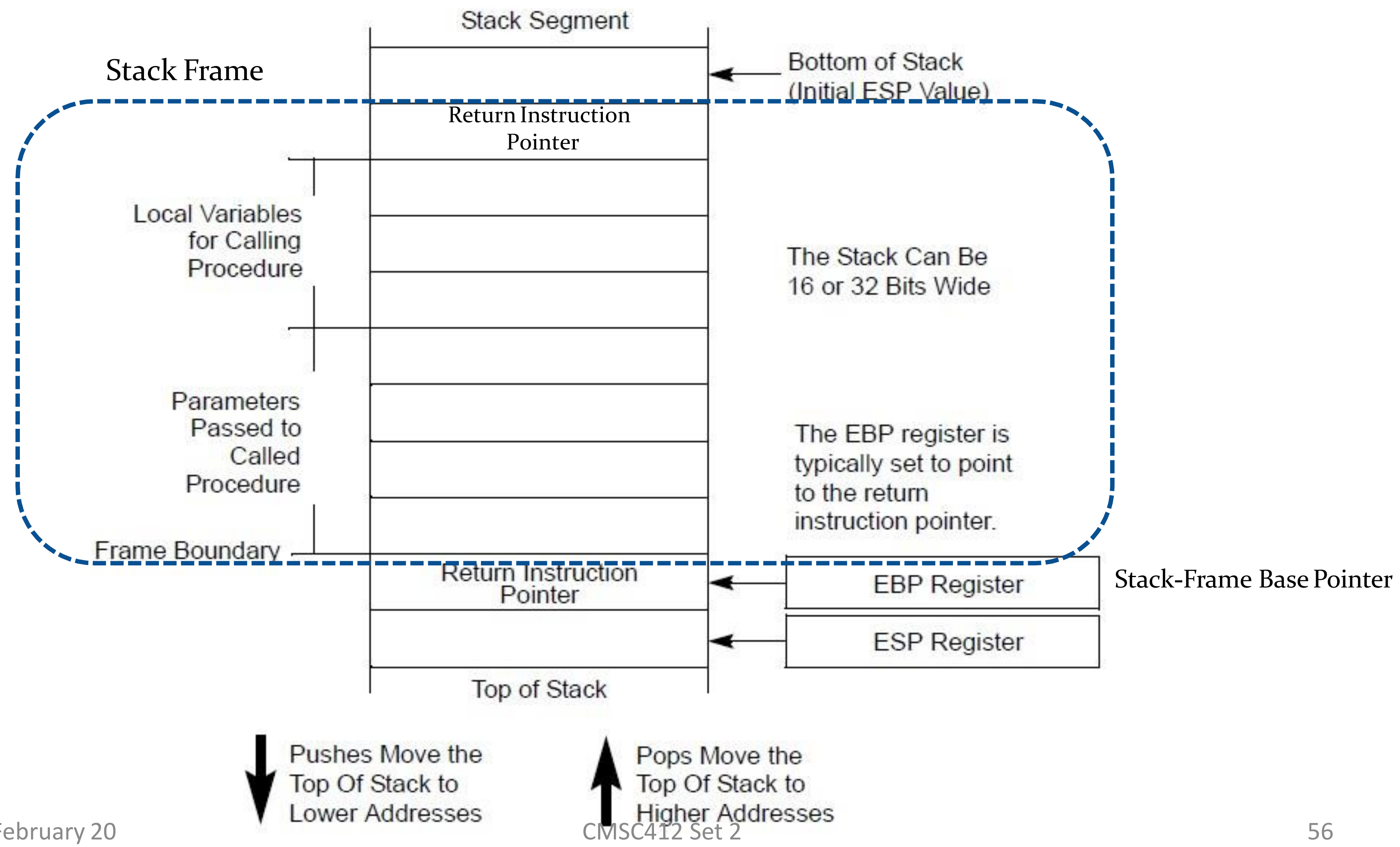
Stack

- The stack is a contiguous array of memory locations. It is contained in a segment and identified by the segment selector in the SS register.

- Items are placed on the stack using the PUSH instruction and removed from the stack using the POP instruction.
  - When an item is pushed onto the stack, the processor decrements the ESP register, then writes the item at the new top of stack. When an item is popped off the stack, the processor reads the item from the top of stack, then increments the ESP register.

- The processor references the SS register automatically for all stack operations. For example, when the ESP register is used as a memory address, it automatically points to an address in the current stack. Also, the CALL, RET, PUSH, POP, ENTER, and LEAVE instructions all perform operations on the current stack.

Stack (cont'd)

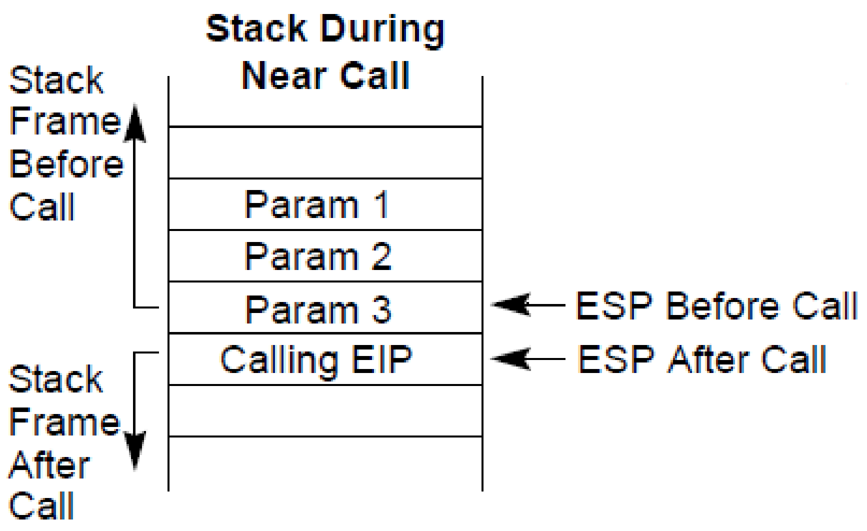

Stack Segment
Stack Frame
Bottom of Stack (Initial ESP Value)
Return Instruction Pointer
Local Variables for Calling Procedure
The Stack Can Be 16 or 32 Bits Wide
Parameters Passed to Called Procedure
The EBP register is typically set to point to the return instruction pointer.
Frame Boundary
Return Instruction Pointer
EBP Register
Stack-Frame Base Pointer
ESP Register
Top of Stack
Pushes Move the Top Of Stack to Lower Addresses
Pops Move the Top Of Stack to Higher Addresses

Procedure Call (CALL/RET)

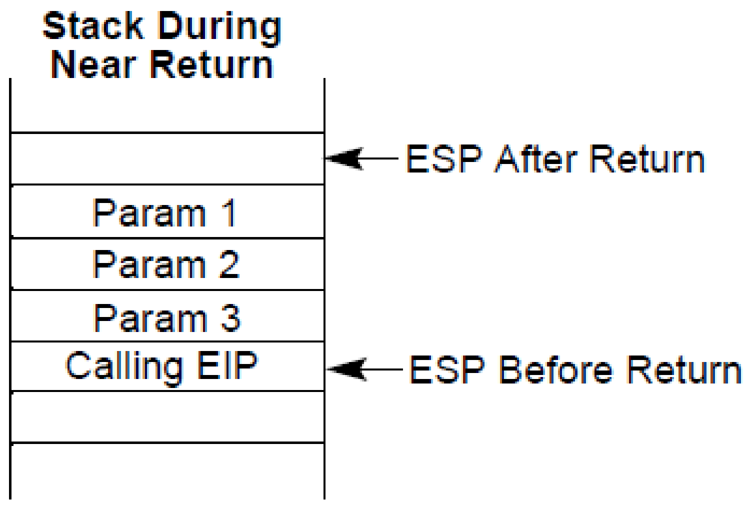
- When executing a call, the processor does the following
  - Pushes the current value of the EIP register on the stack.
  - Loads the offset of the called procedure in the EIP register.
  - Begins execution of the called procedure.

- When executing a near return, the processor performs these actions:
  - Pops the top-of-stack value (the return instruction pointer) into the EIP register.
  - If the RET instruction has an optional n argument, increments the stack pointer by the number of bytes specified with the n operand to release parameters from the stack.
  - Resumes execution of the calling procedure.

Procedure Call (CALL/RET) (cont'd)



CALL *addr*



RET *n*

Interrupt and Exceptions

- When an interrupt or exception is signaled, the processor halts execution of the current program or task and switches to a handler procedure that has been written specifically to handle the interrupt or exception condition.
- The processor accesses the handler procedure through an entry in the interrupt descriptor table (IDT).
- When the handler has completed handling the interrupt or exception, program control is returned to the interrupted program or task.
- If the code segment for the handler procedure has the same privilege level as the currently executing program or task, the handler procedure uses the current stack; if the handler executes at a more privileged level, the processor switches to the stack for the handler's privilege level.
- A return from an interrupt or exception handler is initiated with the IRET instruction. The IRET instruction is similar to the far RET instruction, except that it also restores the contents of the EFLAGS register for the interrupted procedure.

Interrupt and Exceptions (cont’d)

- If no stack switch occurs, the processor does the following when calling an interrupt or exception handler
  - Pushes the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack.
  - Pushes an error code (if appropriate) on the stack.
  - Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
  - If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
  - Begins execution of the handler procedure.

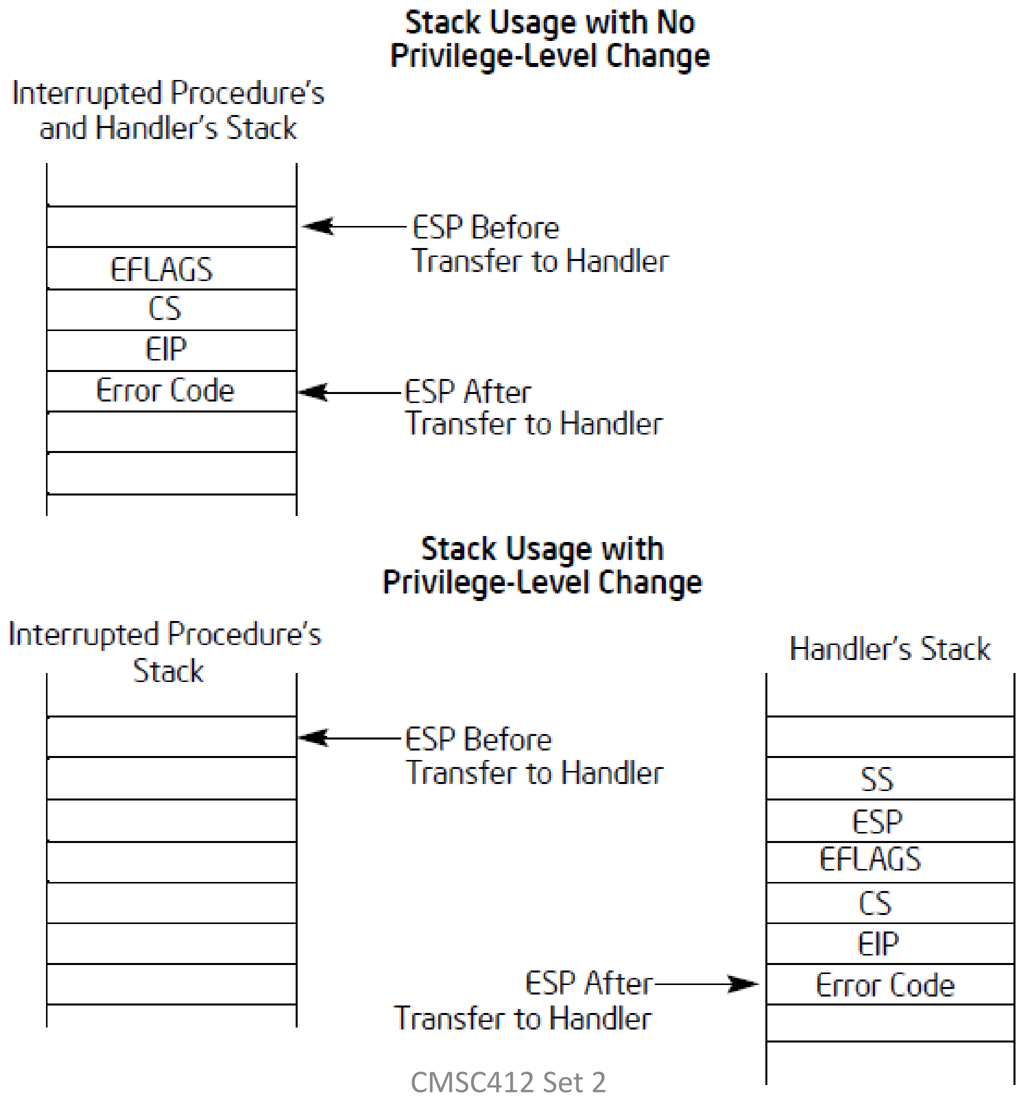
- When executing a return from an interrupt or exception handler from the same privilege level as the interrupted procedure, the processor performs these actions:
  - Restores the CS and EIP registers to their values prior to the interrupt or exception.
  - Restores the EFLAGS register.
  - Increments the stack pointer appropriately.
  - Resumes execution of the interrupted procedure.

Interrupt and Exceptions (cont'd)

- If a stack switch does occur, the processor does the following:
  - Temporarily saves (internally) the current contents of the SS, ESP, EFLAGS, CS, and EIP registers.
  - Loads the segment selector and stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
  - Pushes the temporarily saved SS, ESP, EFLAGS, CS, and EIP values for the interrupted procedure's stack onto the new stack.
  - Pushes an error code on the new stack (if appropriate).
  - Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
  - If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
  - Begins execution of the handler procedure at the new privilege level.

- When executing a return from an interrupt or exception handler from a different privilege level than the interrupted procedure, the processor performs these actions:
  - Performs a privilege check.
  - Restores the CS and EIP registers to their values prior to the interrupt or exception.
  - Restores the EFLAGS register.
  - Restores the SS and ESP registers to their values prior to the interrupt or exception, resulting in a stack switch back to the stack of the interrupted procedure.
  - Resumes execution of the interrupted procedure.

Interrupt and Exceptions (cont'd)

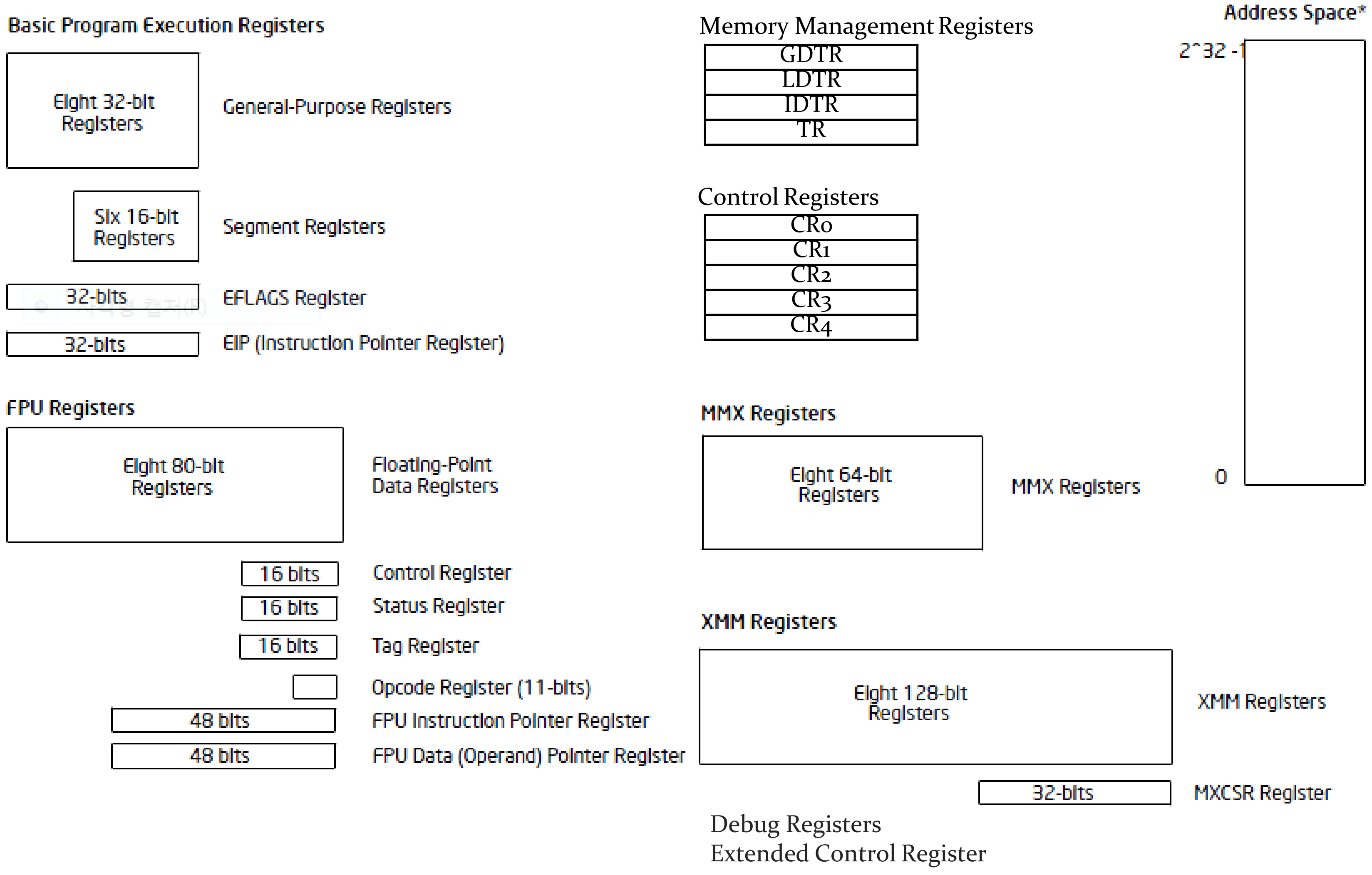

Stack Usage with No Privilege-Level Change
Interrupted Procedure's and Handler's Stack
EFLAGS
CS
EIP
Error Code
ESP Before Transfer to Handler
ESP After Transfer to Handler
Stack Usage with Privilege-Level Change
Interrupted Procedure's Stack
Handler's Stack
ESP Before Transfer to Handler
SS
ESP
EFLAGS
CS
EIP
Error Code
ESP After Transfer to Handler

Intel x86 Architecture
: Summary

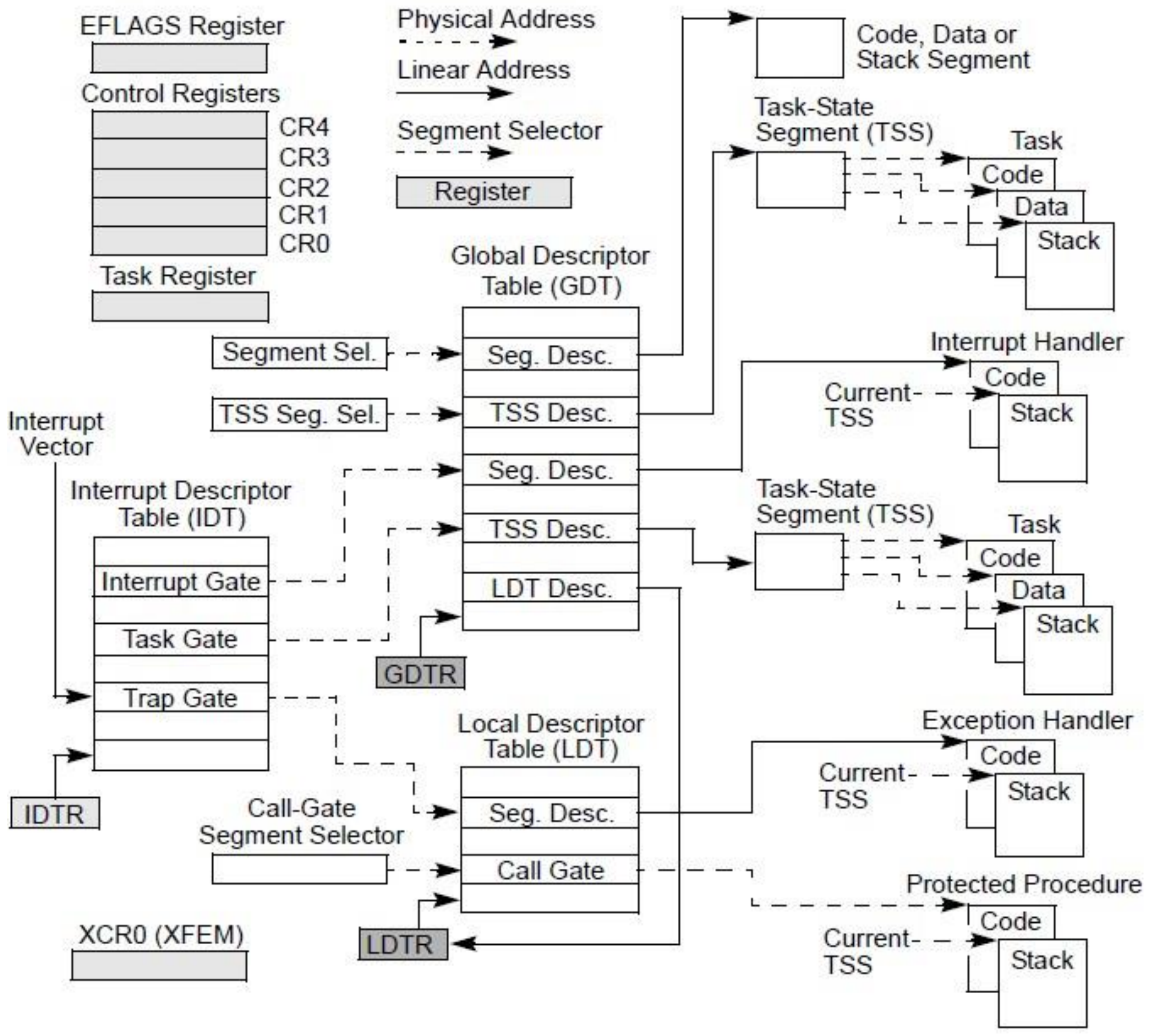
Major differences with ARM processor

- CISC Architecture
  - Many, many instructions
  - More functional processor
- Segmentation
  - Logical address -> linear(virtual) address -> physical address
- Protection
  - Ring 0 ~ 4
  - Gate
  - IO Bitmap

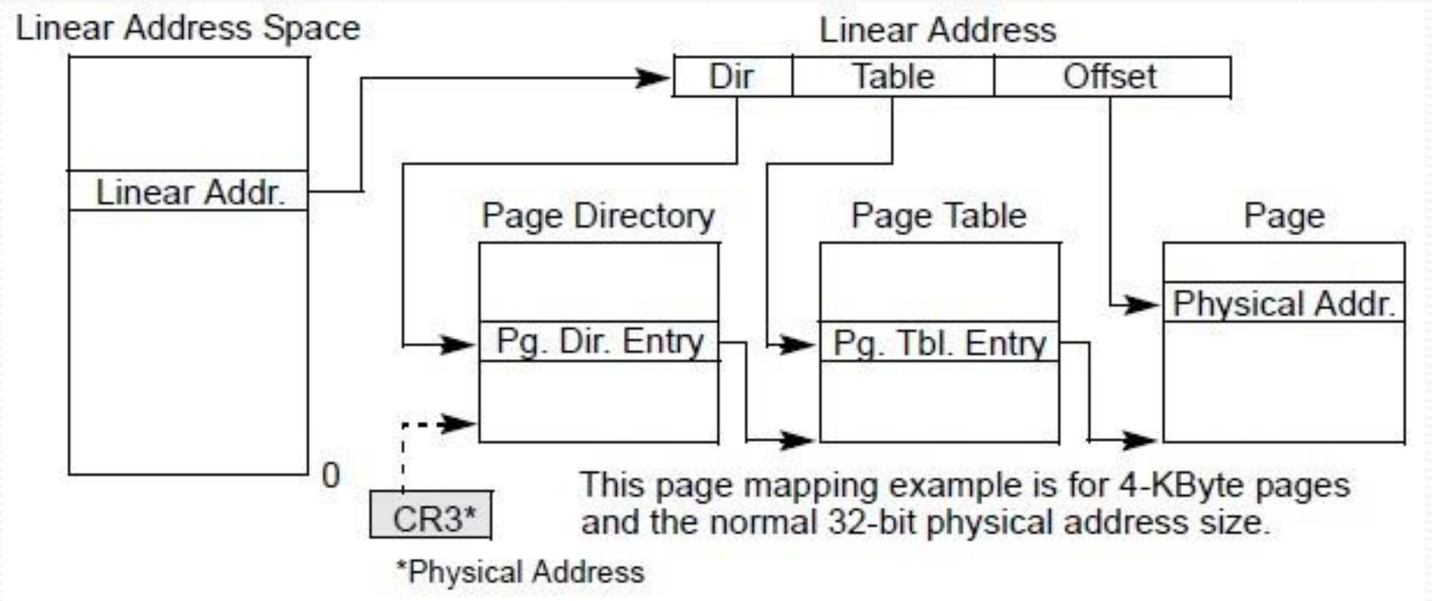
Basic Execution Environment.



System Level Registers and Data Structures



System Level Registers and Data Structures (cont'd)



GCC Assembler

GCC Assembler Syntax

- *mov %eax, %ebx*
  - Transfer the contents of *eax* to *ebx*
    - AT&T Assembly syntax
  - Register names are prefixed by %
  - Source and destination ordering
- *movb, movw, movl*
  - Specifying size of operand by suffix: byte, word, long
- *mov $0xffff, %eax*
  - An immediate operand is specified by using $.
- movb (%esi), %al
  - Any indirect references to memory are done by using ().

GCC Assembler Syntax (cont'd)

- Addressing mode
  - Register Addressing
    - *mov %eax, %ebx*
  - Immediate Addressing
    - *add $-45, %esi*
  - Register Indirect Addressing
    - *mov $45, (%ebx)* *# *ebx = 45*
  - Register Indexed Addressing
    - *mov $45, 12(%ebx)* *# *(ebx + 12) = 45*
  - Base Indexed Addressing
    - *mov $45, 12(%ebx, %esi)* *# *(ebx + esi + 12) = 45*
  - Direct Addressing
    - *mov x, %eax* *# eax = *x*

GCC Assembler Syntax (cont’d)

- Data Definition Pseudo-opcodes

```
x: .long 12
y: .word 34
b: .byte 1
ch: .byte 'A'
myArray: .space 20 # allocates an array of 5 longs
Message: .ascii "Hello World\0"
Message: .asciz "Hello World“
```

- Segment Definition Directives

```
.text
.data
```

Basic Inline Assembly

- *asm ("movb %bh (%eax)");*
  - "_ asm__" is interchangeable with "asm".
- *__asm__("movl %eax, %ebx\n\t"*
  "movl $56, %esi\n\t" …
  - If we have 2 or more instructions, each should be suffixed by "₩n₩t".
- *asm volatile (…)*
  - It must execute where we put it. Ask optimizer not to move the code.

Extended Inline Assembly

- You can specify operands and leave register allocation to GCC just specifying constraints.

```
asm ( assembler template
        : output operands    /* optional */
        : input operands /* optional */
        : list of modified, clobbered, registers /* optional */  );
```

- Template
  - Delimiter: *\ n \ t* and *;*
  - *%0*: Operands corresponding to C-expr are prefixed by *%* in assembler template.
    - The first output operand is numbered 0, continuing in increasing order, and the last input operand is numbered n-1.
    - The order is the same as the order in the operand lists.
  - Operands are prefixed by option letters to describe constraints.

Extended Inline Assembly (cont’d)

- Option Letters
  - = : this operand is assigned to.
  - r: register
  - m: memory
  - i: immediate value
  - g: general effective address(r+m+i)

Example (cont'd)

```
int cas_2w(volatile unsigned long long *p, unsigned long long o,
unsigned long long v)
{
        unsigned long long r;
        unsigned long *n32 = (unsigned long *)&v;
        unsigned long tmp;

__asm_____volatile__(
                "movl %%ebx,%2\n\t"
                "movl %5,%%ebx\n\t"
                "lock\n\t cmpxchg8b %1\n\t"
                "movl %2,%%ebx\n\t"
                : "=A"(r), "=m"(*p), "=m"(tmp)
                : "m"(*p), "o"(o), "g"(n32[0]), "c"(n32[1])
                : "memory"
        );

        return r == o;
}
```

More online resources on GCC inline assembly

- GCC inline assembly
- GCC inline assembly howto
- GCC inline assembly guide (Korean)