

Microcomputers: x86 Architecture Basics

**Dr. Konstantin Levit-Gurevich,
Intel Israel**

Spring Semester 2006

Administration

● Requirements

- ◆ Familiarity with C programming language

● Grading

- ◆ 65% exam, 35% exercises
- ◆ Exercises are mandatory! No exercises => no final grade
- ◆ Bonus: special project

● Web page: <http://www.ee.technion.ac.il/courses/044800/>

- ◆ Latest version of slides
- ◆ Pointers to additional info

● Materials

● Contacts

- ◆ Konstantin Levit-Gurevich
`konstantin.levit-gurevich@intel.com`
- ◆ Isask'har Walter
`zigi@tx.technion.ac.il`

Special Project

●Monitoring of Real System

- ◆ Monitoring and statistics gathering of different events happening in real Multi-Processor (MP) OS
- ◆ Displaying the data in GUI
- ◆ Driver development
Windows, Linux
- ◆ 1-2 people for each OS

●Local APIC usage

- ◆ Sending Inter-Processor Interrupts (IPIs) between two CPUs
- ◆ Driver development
Windows, Linux
- ◆ 1-2 people for each OS

Special Project

●Future Instruction Set Architecture (ISA) Emulator

- ◆ Development of Invalid Opcode exception handler emulating new instructions behavior
- ◆ Development the test including “new instructions”
- ◆ Driver development
Windows, Linux
- ◆ 1-2 people for each OS

●Simple Debugger

- ◆ Source Level or Assembler Level debugger
- ◆ Break Points, Single Step, Step Into, Step Over, Continue features
- ◆ Test case
- ◆ Driver development
Windows, Linux
- ◆ 1-2 people for each OS

Special Project

●x86 Simulator

- ◆ Development of a simple CPU simulator
- ◆ Main features, easy extensible
- ◆ To be used by the students in Microcomputers course for exercises and homework.
- ◆ 1-2 people

●Bonus 1: exemption from the final exam and SW exercises

●Extension - HW part

- ◆ Cache Policy Monitoring
- ◆ Small “Device Manager”
Examining PCI devices existing in the real system

●Bonus 2: exemption from the final exam and all exercises

Objectives

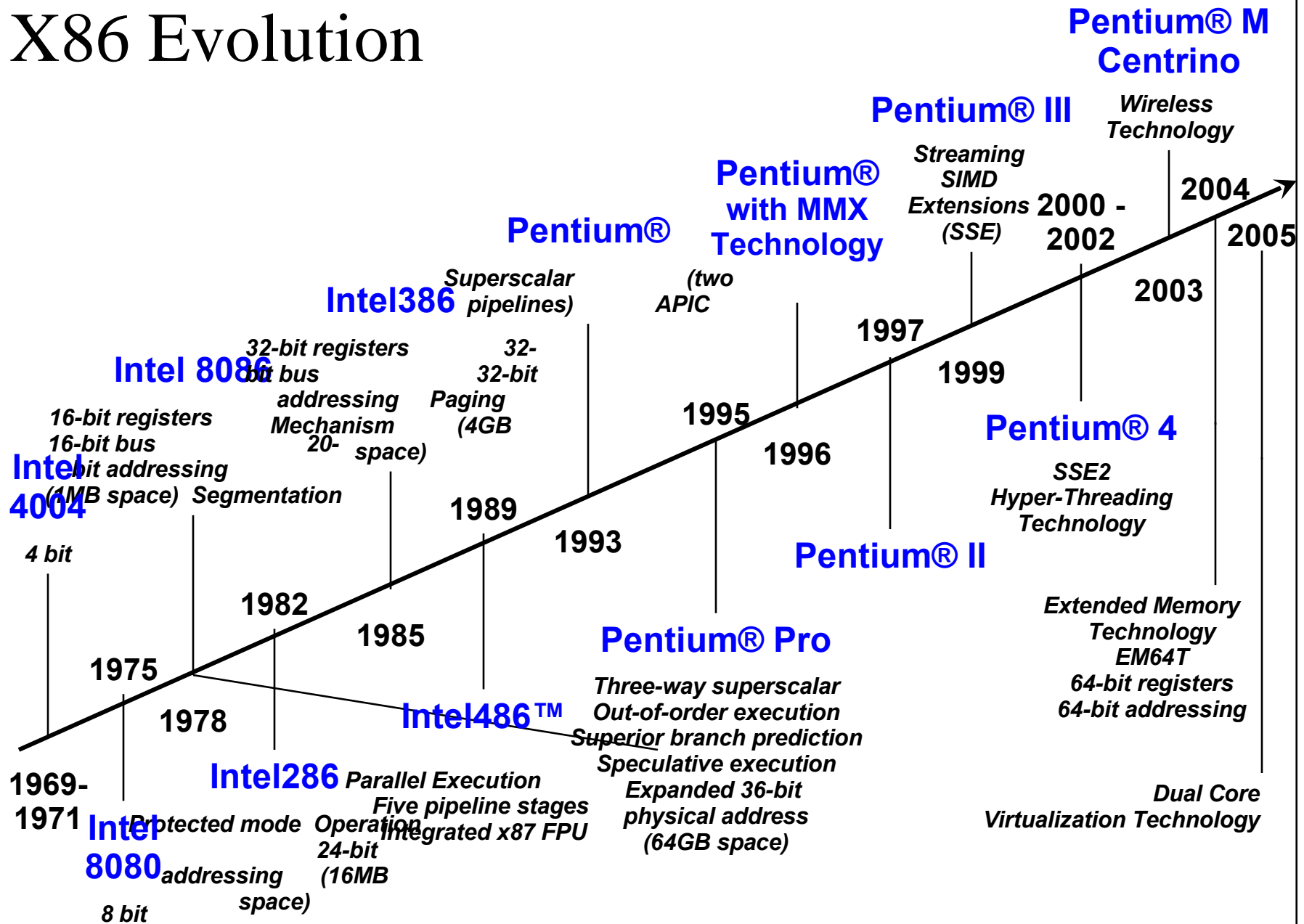
- Get acquainted with the X86 architecture and system components
 - ◆ Historical perspective
 - ◆ Major features
 - ◆ SW/HW aspects
- What you will not get here
 - ◆ Expertise in X86 programming
 - ◆ Full details of everything
- Expect
 - ◆ Varying level of details
 - ◆ Hopefully: taste of more

Agenda

- Introduction
- Basic Architecture
- Segmentation
- Paging
- Protection
- Interrupt and Exception Handling
- Local Advanced Programmable Interrupt Controller (APIC)
- EM64T Architecture
- Virtualization Technology
- Floating Point and MMX

History, Trends, & Terminology

X86 Evolution



Trends

●Phase-I (-1970): “Enter the game”

- ◆ Technology progress

●Phase-II (1970-1990): “Imitate the big guys”

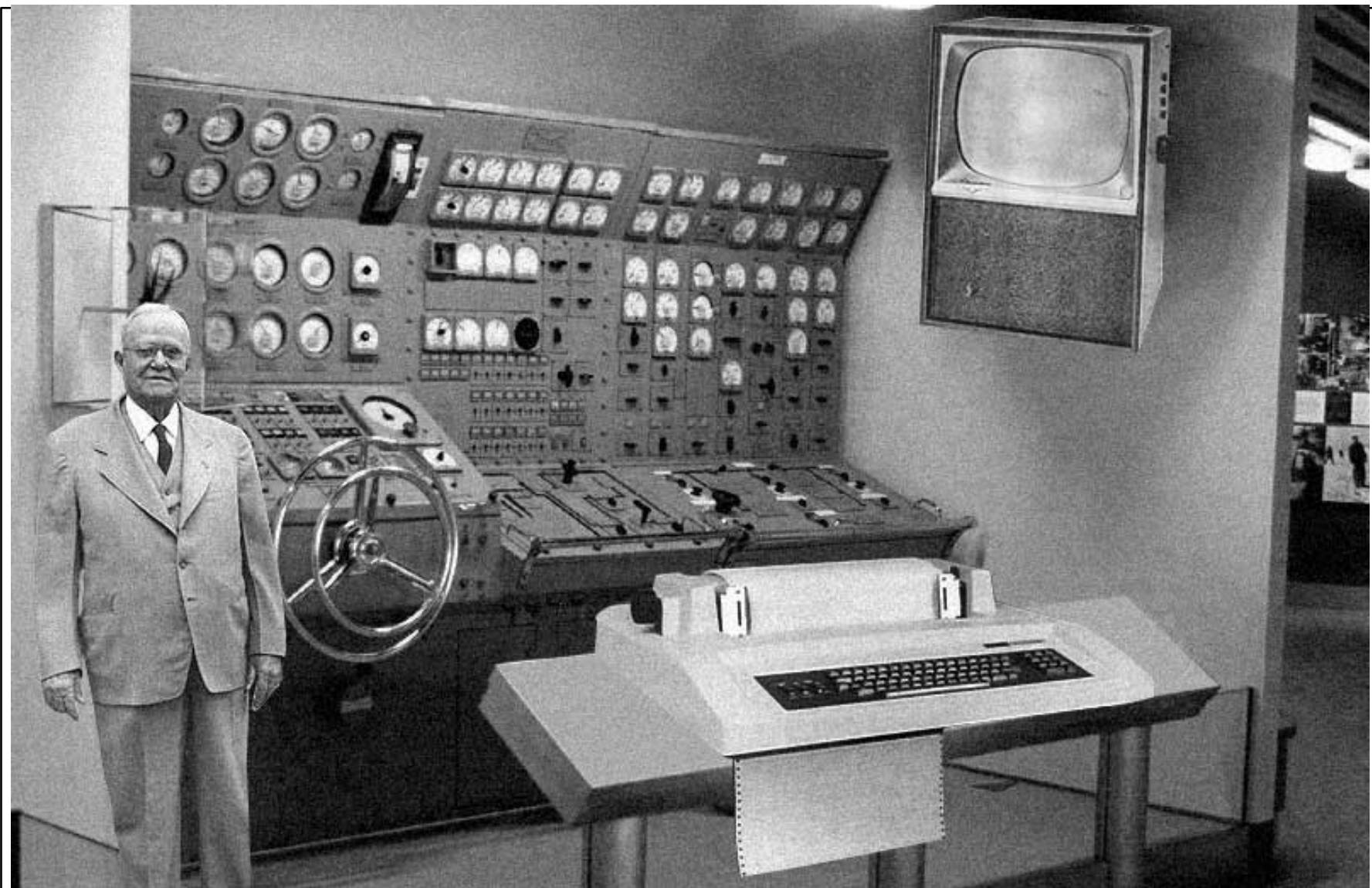
Use technology and copy ideas from mainframes

- ◆ Enlarge word size
- ◆ FP hardware
- ◆ Paging, Protection, Caching
- ◆ High integration

●Phase-III (1990-): “Lead the game”

Innovation unique to microcomputers:

- ◆ Super-scalar, out-of-order execution
- ◆ Branch prediction
- ◆ High frequency
- ◆ Networking
- ◆ Graphics



Scientists from the RAND Corporation have created this model to illustrate how a "home computer" could look like in the year 2004. However the needed technology will not be economically feasible for the average home. Also the scientists readily admit that the computer will require not yet invented technology to actually work, but 50 years from now scientific progress is expected to solve these problems. With teletype interface and the Fortran language, the computer will be easy to use and only

Performance Observations

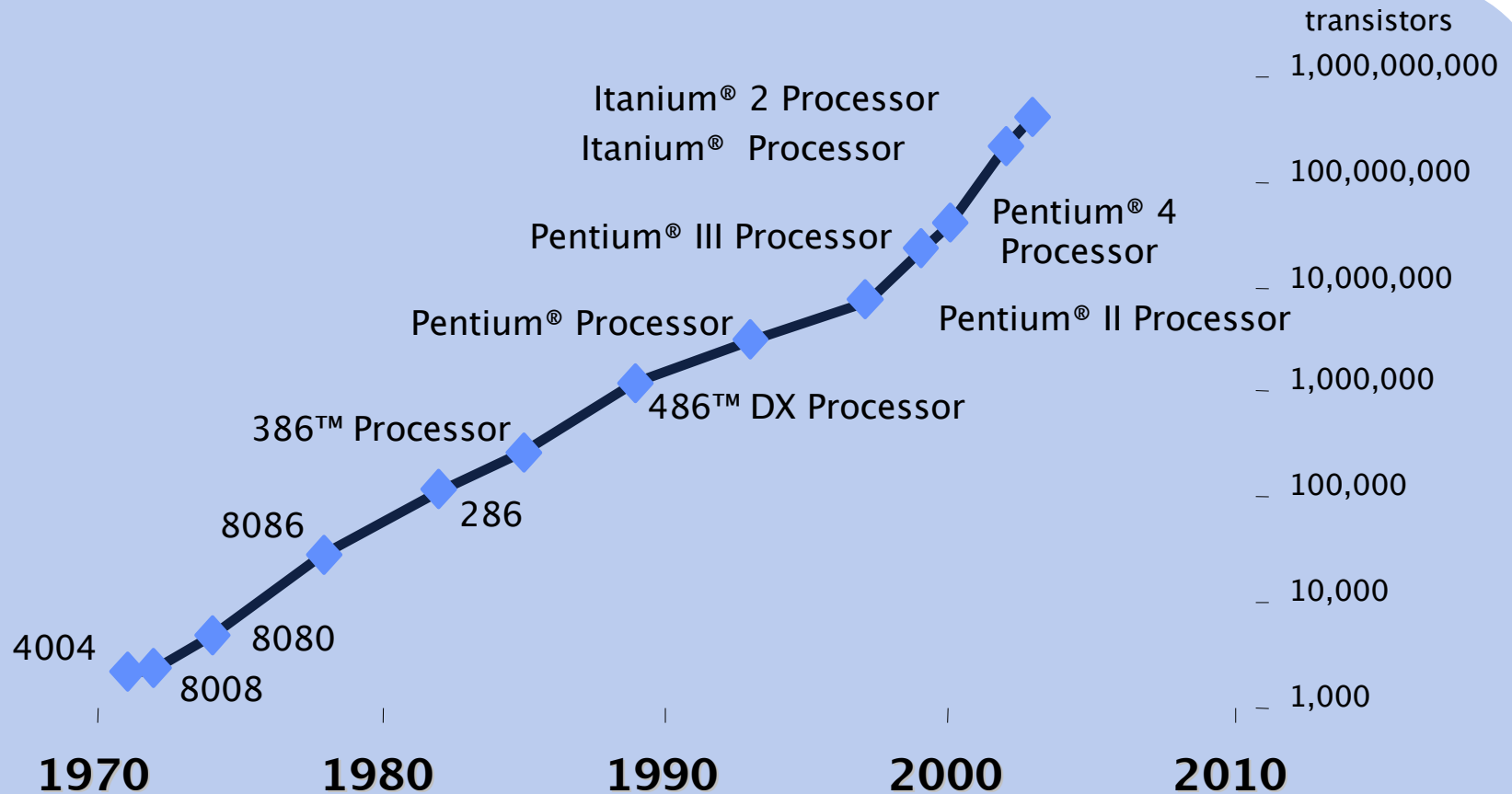
- *“The number of transistors that would be incorporated on a silicon die would double every 18 months for the next several years”.*

Gordon Moore, Intel co-founder. Mid-1960

- The “Spiral” principle (A. Grove):
New software is written to the fastest available processor. Actually, it takes the next generation of processors to run this software (reasonably).
- There will always be applications that will use all available computing power.
“My nightmare is to wake up some day and not need any more computing power.”

Gordon Moore, Intel co-founder. Feb 1998

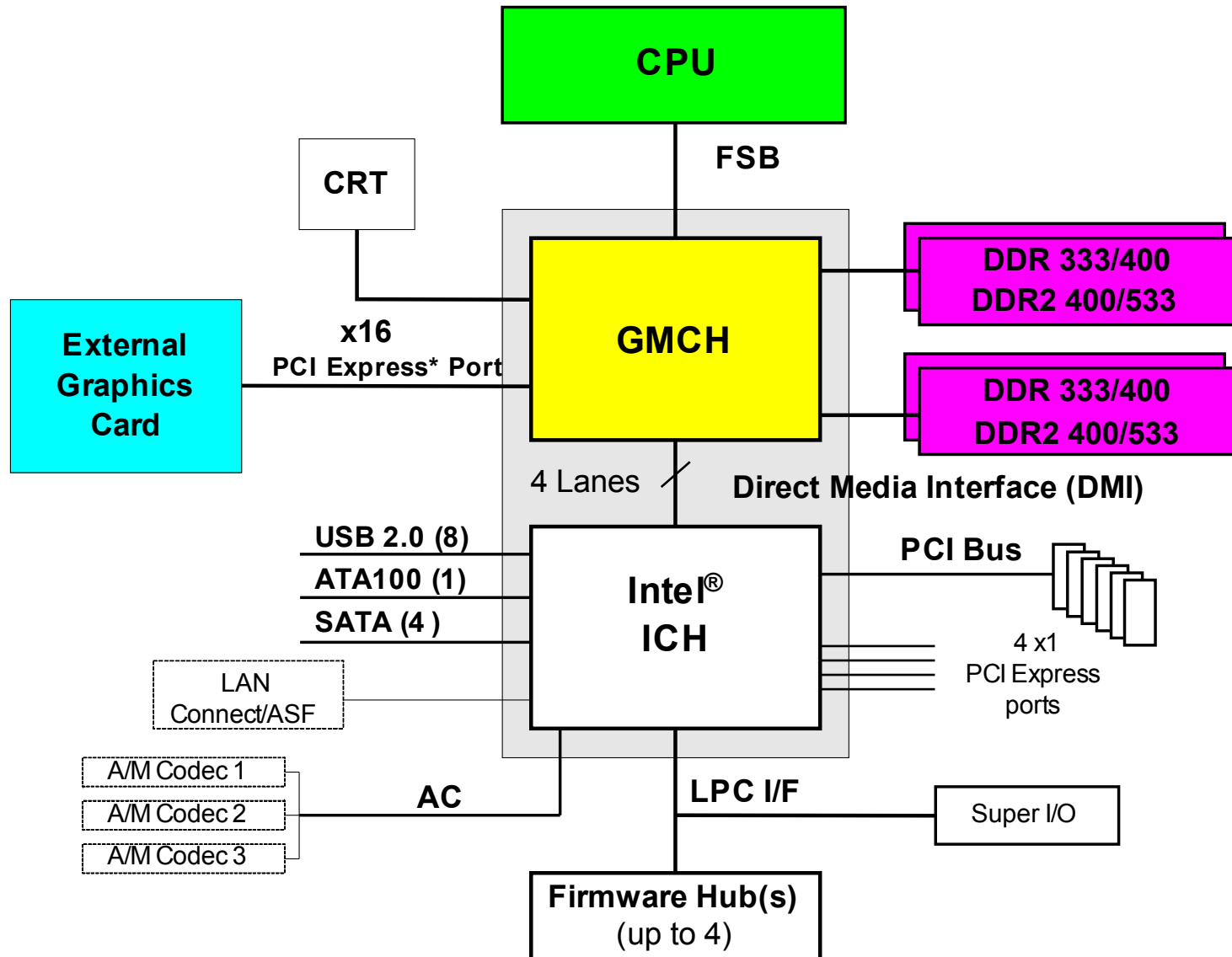
Technology Innovations: Moore's Law Continues



X86 Evolution - Detailed

Date	Processor	Transistors	Bits	Bus	Max Physical Memory	Frequency
1978	8086	29 K	16	16	1 MB	8 MHz
1979	8088	29 K	16	8	1 MB	8 MHz
1982	80286	134 K	16	16	16 MB	12.5 MHz
1985	80386	275 K	32	32	4 GB	20 MHz
1989	Intel 486	1.2 M	32	32	4 GB	25 MHz
1993	Pentium	3.1 M	32	64	4/64 GB	60 MHz
1995	Pentium Pro	5.5 M	32	64	4/64 GB	200 MHz
1997	Pentium II	7 M	32	64	4/64 GB	266 MHz
1999	Pentium III	8.2 M	32	64	4/64 GB	500 MHz
2000	Pentium 4	42 M	32	64	4/64 GB	1.5 GHz
2002	Pentium 4	55 M	32	64	4/64 GB	3.0 GHZ
2003	HT Pentium M	77 M	32	64	4/64 GB	1.6 GHz

Computer System



Architecture & μ Architecture

● Architecture

“The science, art, or profession of designing and constructing building, bridges, etc...”

- ◆ Focus on how it looks externally
- ◆ Consider how it can be built and how good it performs!

● Construction

“The way in which something is built, formed or devised by fitting parts or elements together systematically”

- ◆ Focus on how it is built/operated
- ◆ Make sure it works and works correctly!

● Definition applicable also for buildings, cars, computers, etc...
In computers: construction == μ Architecture

Architecture & μ Architecture

●Architecture

The collection of features of a processor (or a system) as they are seen by the “user”

- ◆ User: a binary executable running on that processor, or
- ◆ assembly level programmer

● μ Architecture

The collection of features or way of implementation of a processor (or a system) do not affect the user.

- Features which change Timing/performance are considered microarchitecture.

Architecture & μ Architecture Elements

● Architecture

- ◆ Registers data width (8/16/32/64)
- ◆ Instruction set
- ◆ Addressing modes
- ◆ Addressing methods (Segmentation, Paging, etc...)
- ◆ Protection
- ◆ etc...

● μ Architecture

- ◆ Bus width
- ◆ Physical memory size
- ◆ Caches & Cache size
- ◆ Number of execution units
- ◆ Execution Pipelines, Number & type execution units
- ◆ Branch prediction
- ◆ TLB
- ◆ etc...

● Timing is considered μ Architecture (*though it is user visible!*)

Compatibility

- Processor A (or system A) is compatible to processor B (or system B) if it preserves the same architecture features as seen by a user of B
- SW compatible - most important for us!
 - ◆ The ability to run existing SW on new HW!
- Architecture Extension
 - ◆ addition of data-types, instructions, etc...to enable better support of new/existing applications.
e.g. MMX
- Trends:
 - ◆ 1985-1995: no/minimal architecture changes
 - ◆ 1995- : we see them coming (MMX, SSE, EM64T, VT, etc.)

Philosophical Basis of First Wave Architecture

● HW / Instruction level support for high-level languages

- ◆ Functions – `main(), f(int x)`
- ◆ Loops – `for, while`
- ◆ Conditional jumps – `if-else`
- ◆ Unconditional jumps – `goto, continue, break`
- ◆ Increments and decrements – `i++, i--`
- ◆ Support operations with immediate value – `a = 10;`
- ◆ Support Read-Modify-Write operations – `a += b;`

● Instruction Set encoding for memory efficiency

- ◆ More frequent instructions – shorter encoding
- ◆ Two explicit operands instructions
REG-REG, REG-IMM, REG-Memory, Read-Modify-Write operations
- ◆ Specialized and Implicit Register Usage
- ◆ Complex Instructions to maximize functionality / instruction fetch

Philosophical Basis of First Wave Architecture

● Memory Segmentation

- ◆ Relocatable programs / modules well supported
- ◆ Stack and Data spaces in instruction set architecture

Local and global variables

● Addressing modes suitable for high-level languages

- ◆ Pointers – `*p`
- ◆ Structure fields addressing – `struct.field`
- ◆ Array elements addressing – `array[i]`
- ◆ Special instructions addressing stack – `PUSH, POP, PUSHF, POPF`

“C” Language Code Example

```
struct Str{  
    int x, y;  
} array[100];
```

=> Structures

=> Arrays

```
void func(struct Str* s)  
{
```

=> Functions,
pointers

```
    int i;
```

=> Local variables
saved on stack

```
    for (i = 0; i < 100; i++)  
    {
```

=> Loops, index
increment

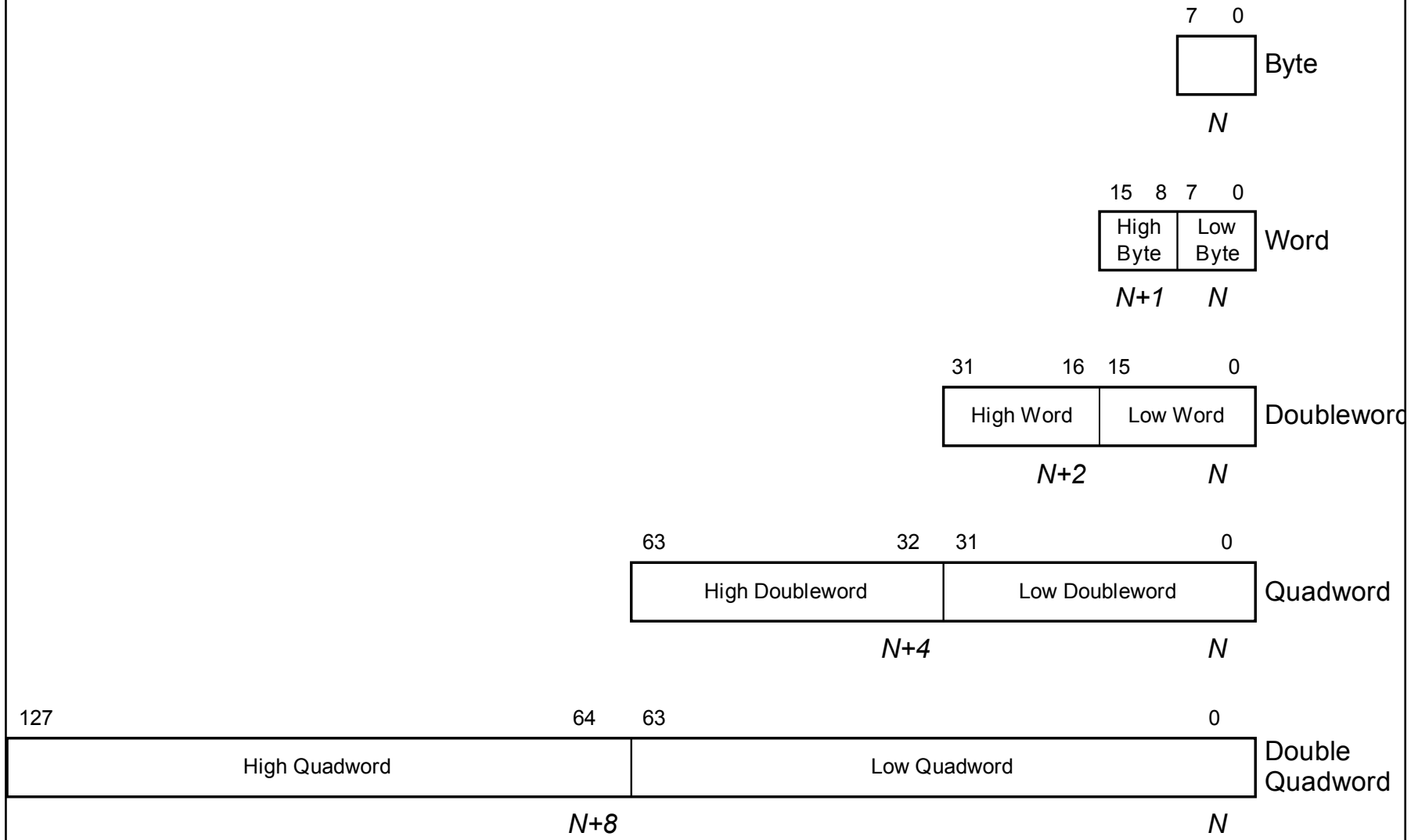
```
        array[i].x += s->x + i;  
        array[i].y += s->y + i;
```

=> Structure field
access

```
    }  
}
```

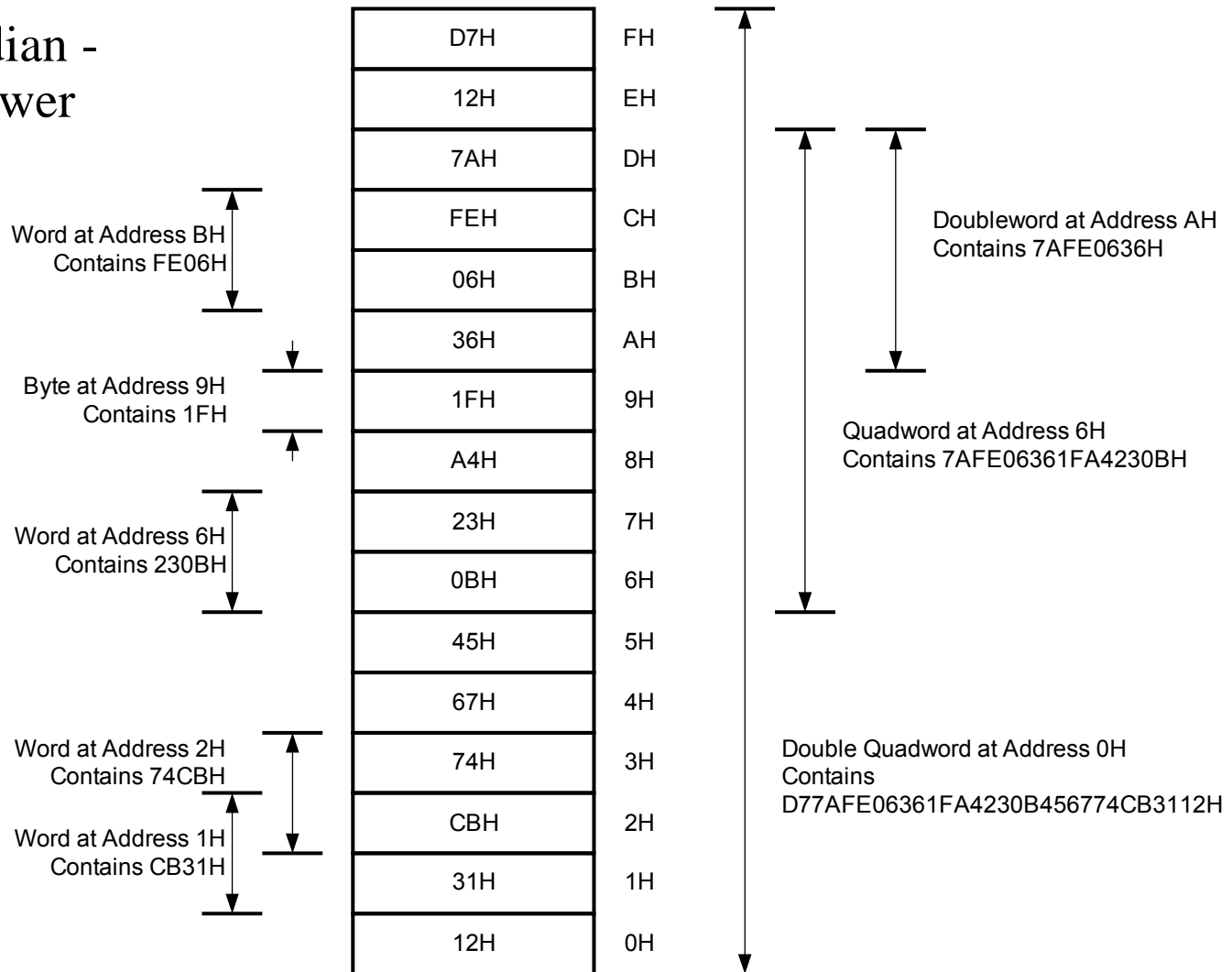
X86 Architecture Basics

Fundamental Data Types



Memory Layout (1)

- Little Endian -
LSB in lower
addresses



Memory Layout (2)

```

Brd 0 Memory [P0]
0x0018:00000000c010543d 89 f6 fb 03 8d b4 26 00 00 00 00 8d bc 27 00 '.....&.....'
0x0018:00000000c010544d 00 00 00 fb 08 ff ff ff ff ba 00 e0 ff ff 21 e2 '.....!.....'
0x0018:00000000c010545d 87 42 14 05 c0 75 0a 89 d0 83 78 14 ff f3 90 74 'B...u...x...'
0x0018:00000000c010546d f8 c3 90 55 bd 10 54 10 c0 57 56 53 e8 62 35 21 '...U..T..WVS.b5'
0x0018:00000000c010547d 00 b8 00 e0 ff ff 21 e0 c7 40 24 14 00 00 00 89 '.....!@$.....'
0x0018:00000000c010548d c7 c7 40 20 9c ff ff ff 8d 74 26 00 8d bc 27 00 '...@.....t&...'
0x0018:00000000c010549d 00 00 00 8b 1d 00 32 35 c0 8b 47 14 85 db 0f 44 '.....25..G....'
0x0018:00000000c01054ad dd 85 c0 75 17 be 00 e0 ff ff 21 e6 8d b4 26 00 '...u.....!&...'
0x0018:00000000c01054bd 00 00 00 ff d3 8b 46 14 85 c0 74 f7 e8 62 2f 01 '.....F...t..b/'
0x0018:00000000c01054cd 00 e8 dd 42 02 00 eb cb 8d 74 26 00 8d bc 27 00 '...B.....t&...'

```

```

Brd 0 Memory [P0]
0x0018:00000000c010543d 89 f6 fb 03 8d b4 26 00 00 00 00 8d bc 27 00
0x0018:00000000c010544d 00 00 00 fb 08 ff ff ff ff ba 00 e0 ff ff 21 e2
0x0018:00000000c010545d 87 42 14 05 c0 75 0a 89 d0 83 78 14 ff f3 90 74
0x0018:00000000c010546d c3f8 5590 10bd 1054 57c0 5356 62e8 2135
0x0018:00000000c010547d b800 e000 ffff e021 40c7 1424 0000 8900
0x0018:00000000c010548d c7c7 2040 ff9c ffff 748d 0026 bc8d 0027
0x0018:00000000c010549d 0000 8b00 001d 3532 8bc0 1447 db85 440f
0x0018:00000000c01054ad 85dd 75c0 be17 e000 ffff e621 b48d 0026
0x0018:00000000c01054bd 0000 ff00 8bd3 1446 c085 f774 62e8 012f
0x0018:00000000c01054cd e800 42dd 0002 cbeb 748d 0026 bc8d 0027
0x0018:00000000c01054dd 0000 5700 5356 5c8b 1424 e4e8 0032 9c00

```

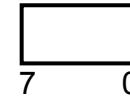
```

Brd 0 Memory [P0]
0x0018:00000000c010543d fbf689c3 20b48dc3 00000000 0027bc8d
0x0018:00000000c010544d fb000000 ffffffb8 e000baff e221ffff
0x0018:00000000c010545d 5590c3f8 105410bd 535657c0 213562e8
0x0018:00000000c010546d 5590c3f8 105410bd 535657c0 213562e8
0x0018:00000000c010547d e000b800 e021ffff 142440c7 89000000
0x0018:00000000c010548d 2040c7c7 ffffffff9c 0026748d 0027bc8d
0x0018:00000000c010549d 8b000000 3532001d 14478bc0 440fdb85
0x0018:00000000c01054ad 75c085dd e000be17 e621ffff 0026b48d
0x0018:00000000c01054bd ff000000 14468bd3 f774c085 012f62e8
0x0018:00000000c01054cd 42dde800 cbeb0002 0026748d 0027bc8d
0x0018:00000000c01054dd 57000000 5c8b5356 e4e81424 9c000032

```

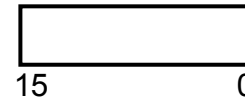
Numeric Data Types - Integers

0 – 255



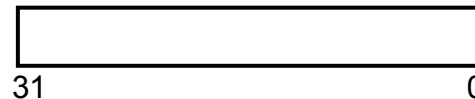
Byte Unsigned Integer

0 – 65,535



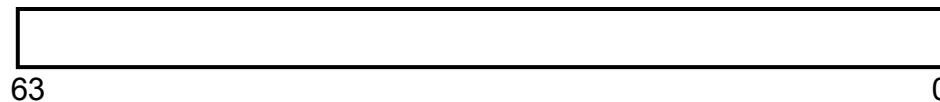
Word Unsigned Integer

0 – 4,294,967,295



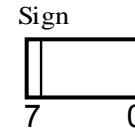
Doubleword Unsigned Integer

0 – $2^{64} - 1$



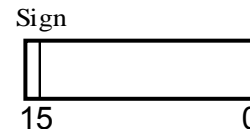
Quadword Unsigned Integer

-128 – +127



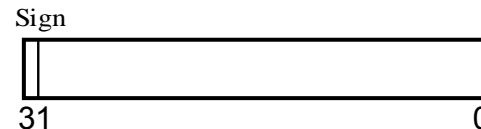
Byte Signed Integer

-32,768 – +32,767



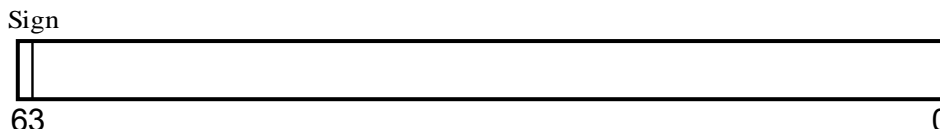
Word Signed Integer

$-2^{31} - +2^{31} - 1$



Doubleword Signed Integer

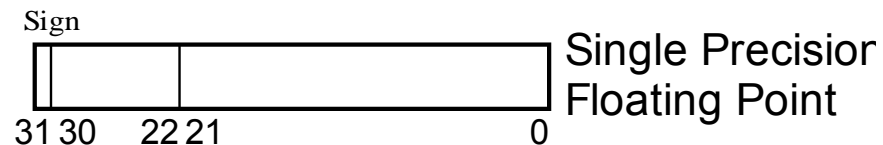
$-2^{63} - +2^{63} - 1$



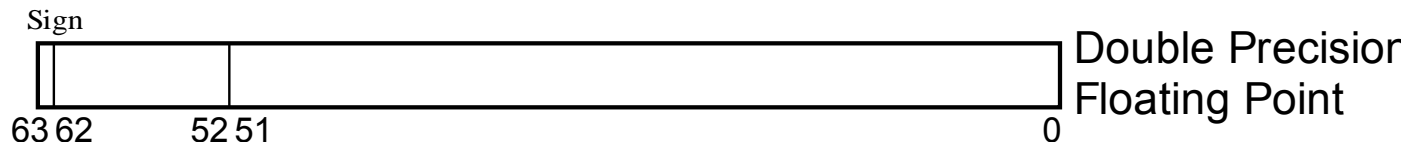
Quadword Signed Integer

Numeric Data Types – Floating Point

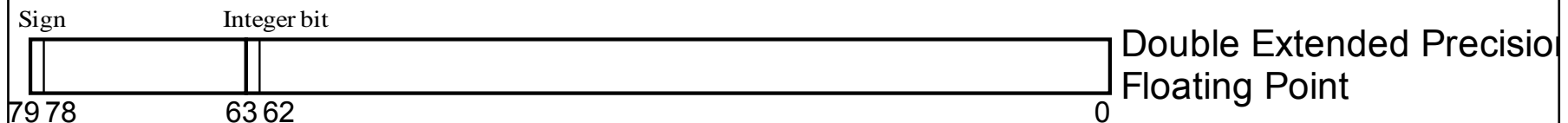
Approximate Normalized Range: 1.18×10^{-38} to 3.40×10^{38}



Approximate Normalized Range: 2.23×10^{-308} to 1.79×10^{308}

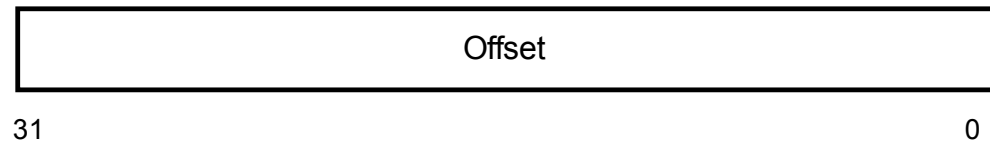


Approximate Normalized Range: 3.37×10^{-4932} to 1.18×10^{4932}

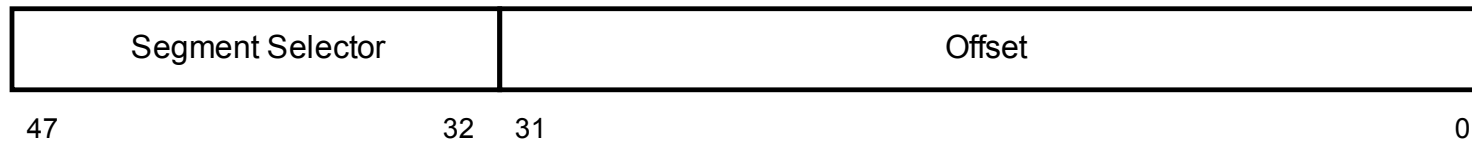


Pointer Data Types

Near Pointer



Far Pointer or Logical Address



Basic Registers

General-Purpose Registers

	31	16	15	8	7	0	16-bit	32-bit
Accumulator					AH	AL	AX	EAX
Base					BH	BL	BX	EBX
Count					CH	CL	CX	ECX
Data					DH	DL	DX	EDX
Source Index					SI			ESI
Destination Index					DI			EDI
Base Pointer					BP			EBP
Stack Pointer					SP			ESP

Special Registers

31	0	
Program Status and Control Register		
31	0	EFLAGS
Instruction Pointer		
		EIP

15	0	Segment Registers
Code Segment		CS
Data Segment		DS
Stack Segment		SS
Extra Segment		ES
		FS
		GS

Flags

- Affected by ALU operations, e.g.,:

- ADD,
- CMP

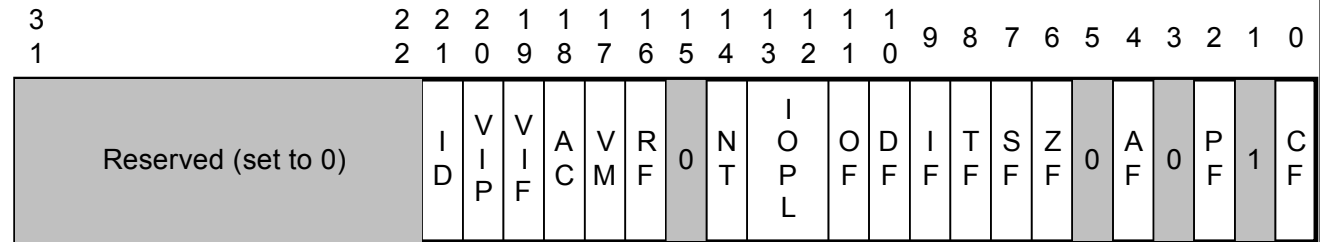
- Used mainly for control flow:

- JCC

- But also by other instructions

- ADC, RLC, LOOP, etc...

Status Flags



ID - Identification Flag

VIP - Virtual Interrupt Pending

VIF - Virtual Interrupt Flag

AC - Alignment Check

VM - Virtual-8086 Mode

RF - Resume Flag

NT - Nested Task Flag

IOPL - I/O Privilege Level

IF - Interrupt Enable Flag

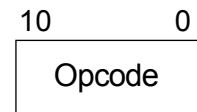
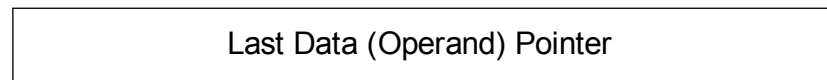
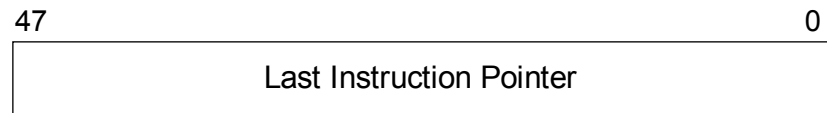
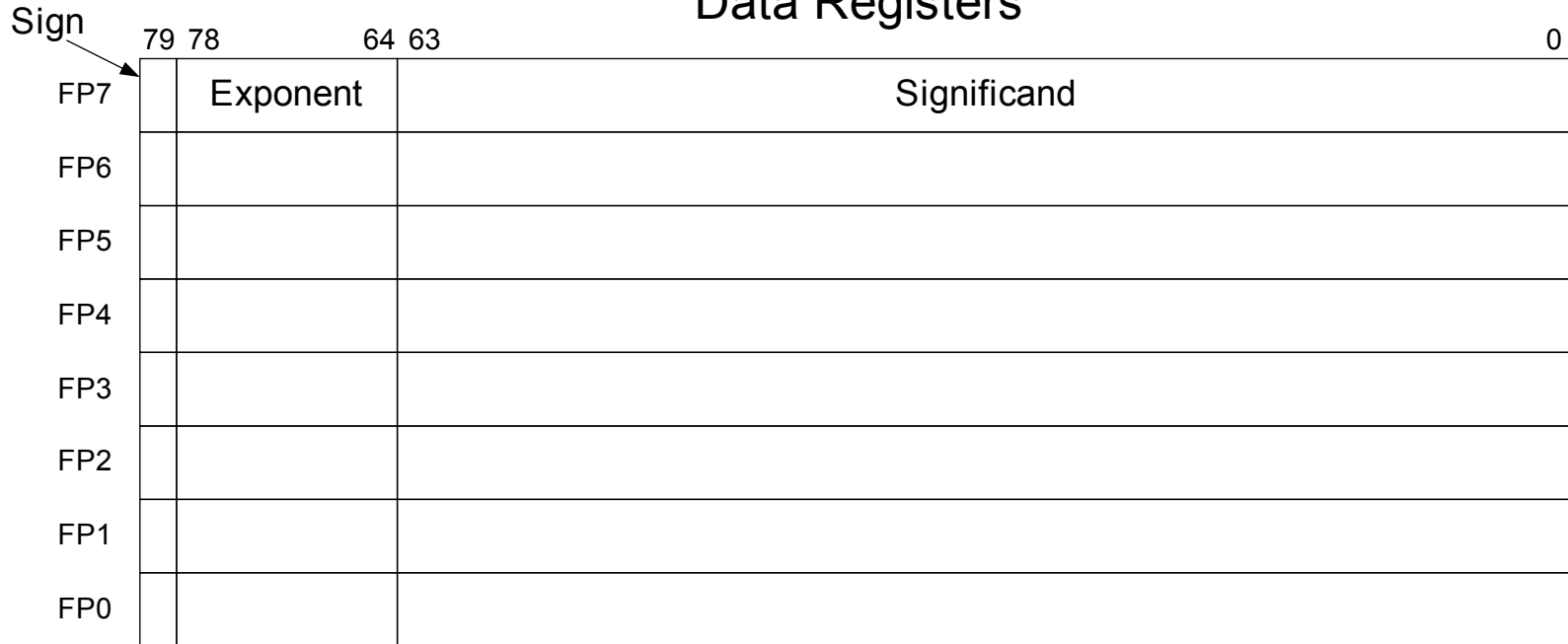
TF - Trap Flag

EFLAGS Register

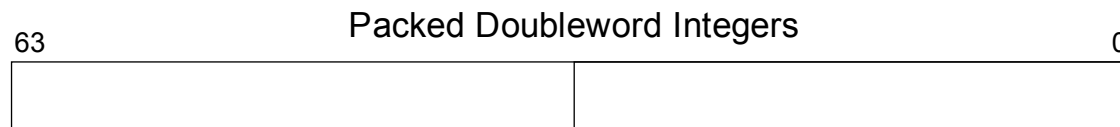
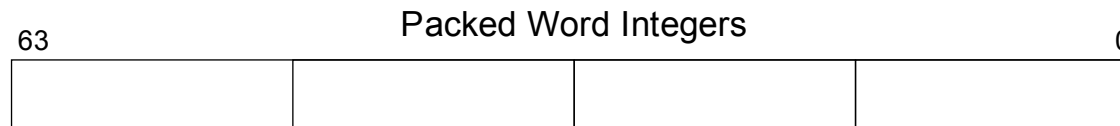
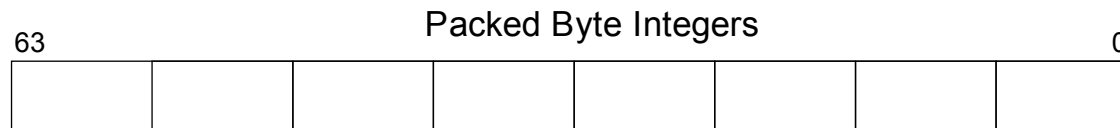
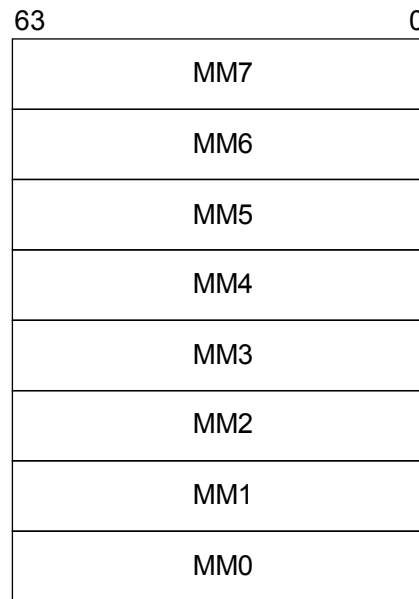
NAME	PURPOSE	CONDITIONS REPORTED
OF	Overflow Flag	Result exceeds positive or negative limit of numbers range
SF	Sign Flag	Result is negative (less then zero)
ZF	Zero Flag	Result is zero
AF	Auxiliary Carry	Carry out of bit position 3 (used for BCD)
PF	Parity Flag	Low byte of result had even parity (even number of set bits)
CF	Carry Flag	Carry out of most significant bit of result

Floating Point

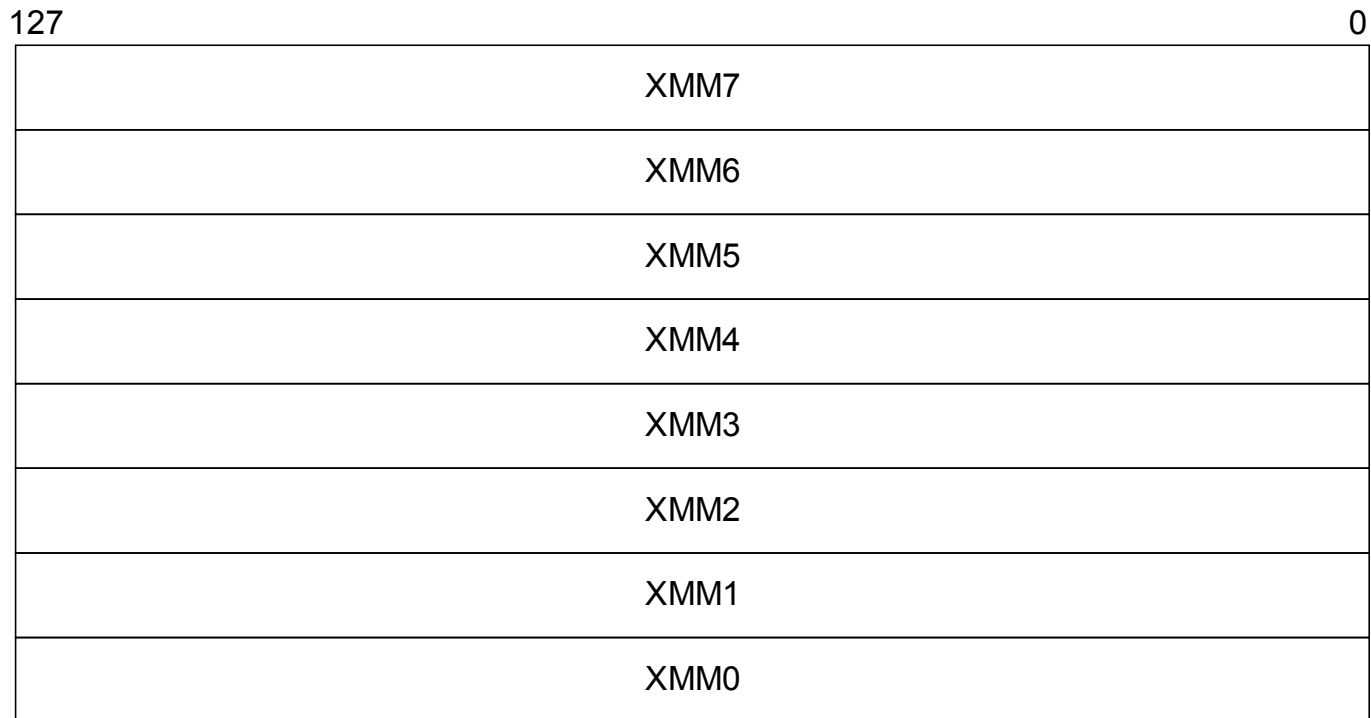
Data Registers



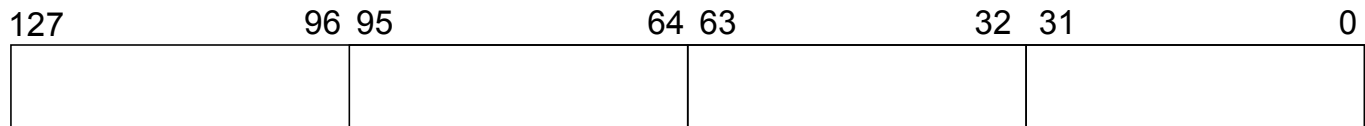
MMX



Streaming SIMD Extensions



Packed Single Precision Floating-Point



Stack

● Related instructions:

- ◆ PUSH/POP,
PUSHF/POPF,
PUSHA/POPA,
CALL/RET,
IRET

Local Variables
for Calling
Procedure

● Used in interrupt handling

Parameters
Passed to
Called
Procedure

Frame Boundary

Stack Segment

Bottom of Stack
(Initial ESP Value)

The Stack Can Be
16 or 32 Bits Wide

The EBP register
is typically set to
point to the return
instruction pointer

Return Instruction
Pointer

EBP Register

ESP Register

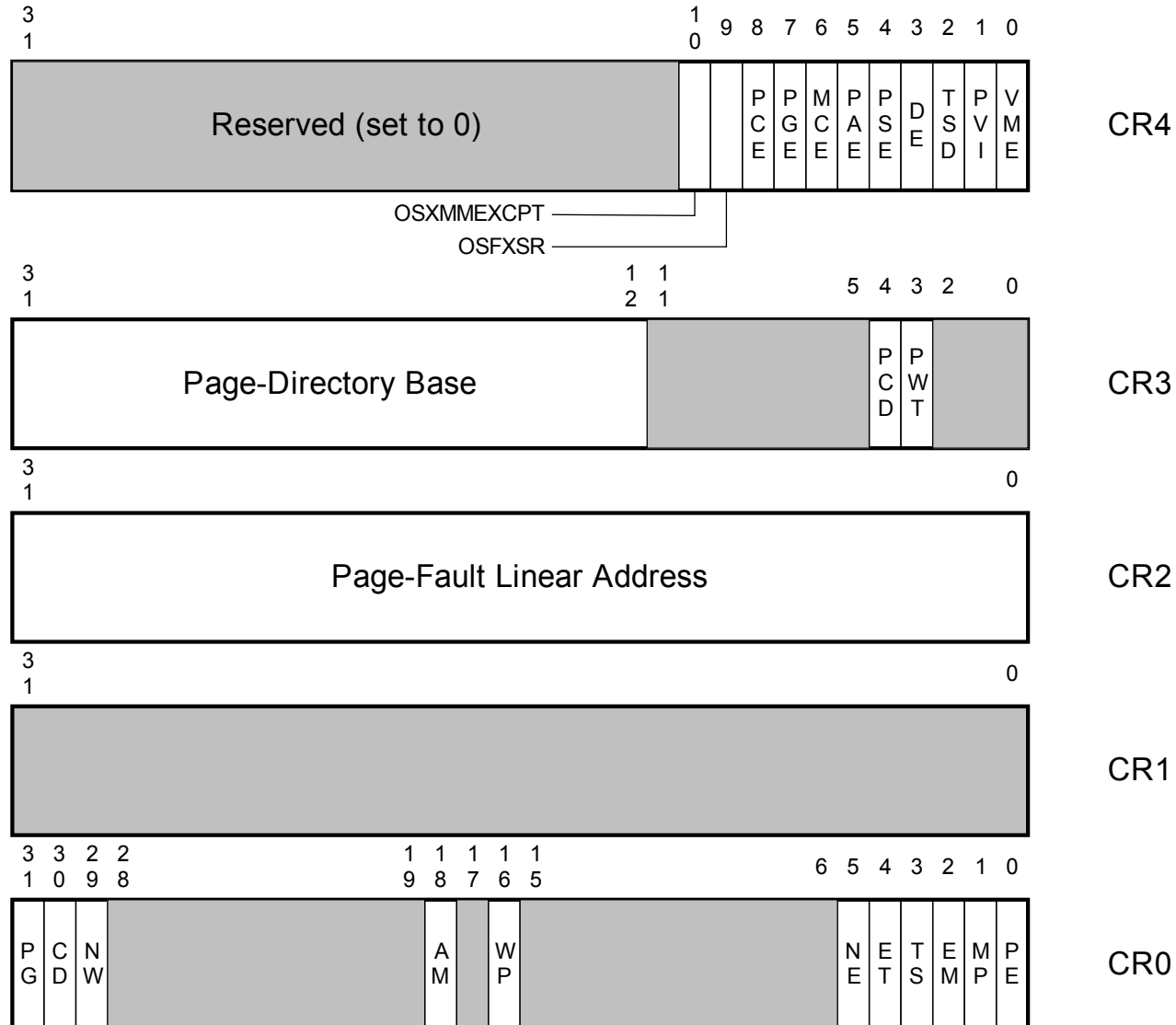
Top of Stack

Pushes Move the
Top of Stack to
Lower Addresses



Pops Move the
Top of Stack to
Higher Addresses

Control Registers



Other Registers

- Segment Registers

- ◆ CS, SS, DS, ES, FS, GS

- System Registers

- ◆ GDTR, IDTR, LDTR

- Task Register

- ◆ TR

- Debug Registers

- ◆ DR0-DR7 – not discussed here

- Model Specific Registers

- ◆ MSR, MTRR – not discussed here

System Registers Example - Linux

```

Brd 0 IA-32 Registers [P0]
eax=00000000  esi=c030c000  cr0=80050033  dr0=00000000
ebx=c0105410  edi=c030c000  cr2=40018000  dr1=00000000
ecx=c030c000  ebp=c0105410  cr3=37297000  dr2=00000000
edx=00000000  esp=c030dfc8  cr4=00000650  dr3=00000000
eip=c010543d  eflags=00000246  dr6=ffff0ff0
dr7=00000400

idtbas=c034b000  idtlim=ffff
gdtbas=c02f1b40  gdtlim=049f
ldtbas=f8a82000  ldtlim=0000ffff  ldtr=82  ldtr=00a8
tssbas=c034b800  tsslim=00000067  tssar=8b  tr=00a0

csbas=00000000  csLim=ffffffff  csar=9b  cs=0010
dsbas=00000000  dsLim=ffffffff  dsar=93  ds=0018
ssbas=00000000  ssLim=ffffffff  ssar=93  ss=0018
esbas=00000000  esLim=ffffffff  esar=93  es=0018
fsbas=00000000  fsLim=ffffffff  fsar=93  fs=0018
gsbas=00000000  gsLim=ffffffff  gsar=93  gs=0018

```

Brd 0 eflags [P0]																																												
3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0											
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	0	0											
												i	v	v	a	v	r			i																								
												d	p	f	c	m	f			n	t	l																						
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	1	0	0	0	0	0										

Memory Organization

● Physical Memory

- ◆ The memory that the processor addresses on its bus.
- ◆ Each byte is assigned with the unique address, called a **Physical Address**.
- ◆ Maximal size is 64 GB (in IA32).

● Programs do not directly access the physical memory. Instead, they access memory using any of three following memory models:

◆ Real-Address Mode

Uses the memory model for the Intel 8086 processor.

Uses a specific implementation of segmented memory in which the linear address space for the program and the operating system consists of an array of segments of 64 KB in size each.

◆ Flat Mode

Memory appears to the program as a single, continuous address space, called **Linear Address Space**.

Code, data and a procedure stack are all contained in this address space.

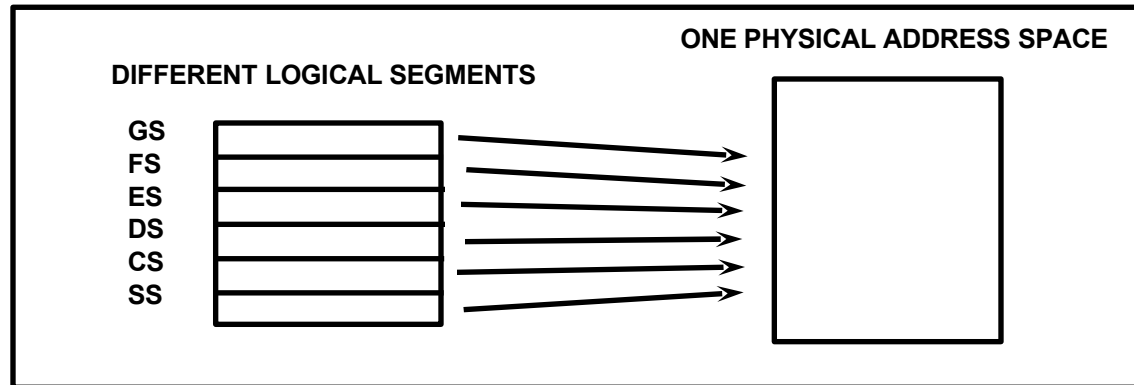
Size is 4 GB.

◆ Segmented Mode

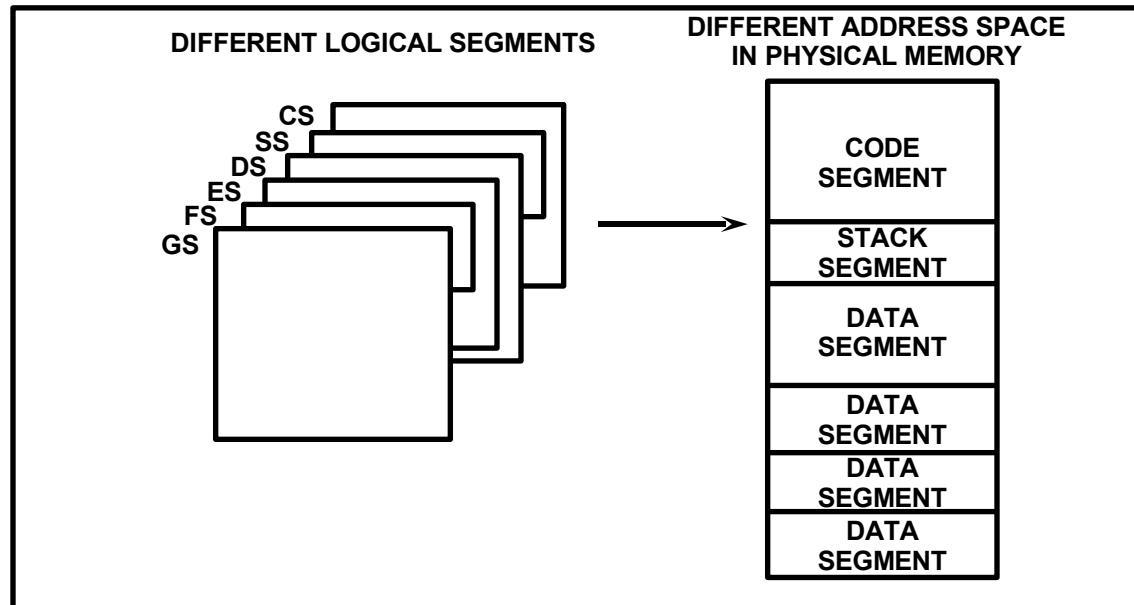
Segmentation

- Problem: use “large” memory with short word width
 - ◆ 16 bits can cover only 64KB
- Solution
 - ◆ address space divided into “segments”
- **Logical address** is *segment:offset*
- Segmentation: **mapping from Logical Address to Linear Address**
 - ◆ Segment base = f(segment register)
 - ◆ offset < segment size: $\leq 64\text{K}$ (8086/286), $\leq 4\text{G}$ (386+)
 - ◆ ***linear address = segment_base + offset***
- At any time. there are 4 (8086) or 6 (386 and on) active segment registers
 - ◆ **CS**: code, **DS**: data, **SS**: stack
 - ◆ **ES**: extra, **FS**, **GS**: new extra
- Rough analogy - phone directory: region and a number within

Segmentation

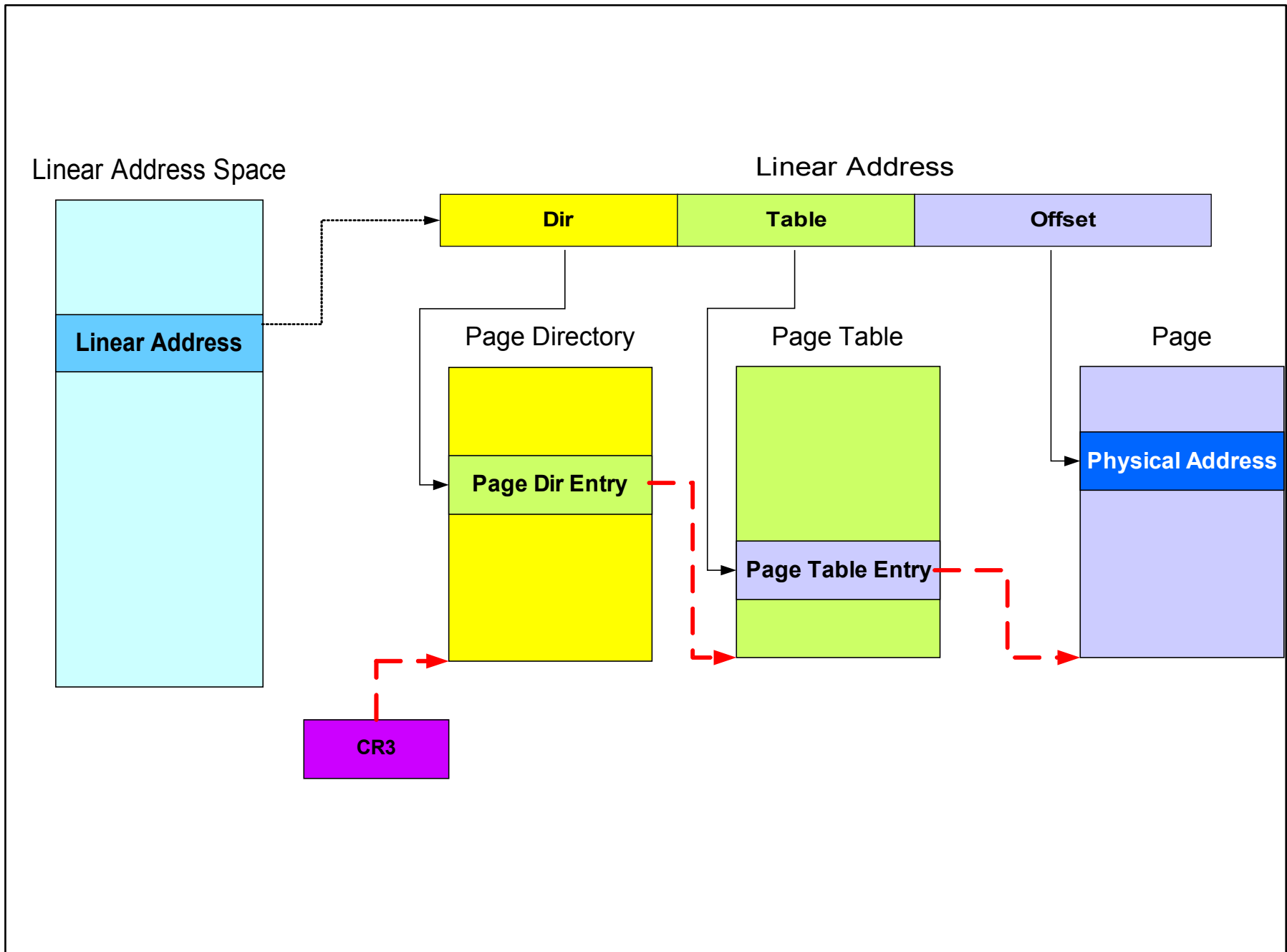


An Unsegmented Memory



A Segmented Memory

System Architecture Overview



Operation Modes

●Real-address Mode

- ◆ Implements the programming environment of the Intel 8086 processor with a few extensions. Processor is placed in real-address mode following power-up or a reset.

●Protected Mode

- ◆ Native state of the processor. All instructions and architectural features are available. This is the recommended mode for applications and operating systems

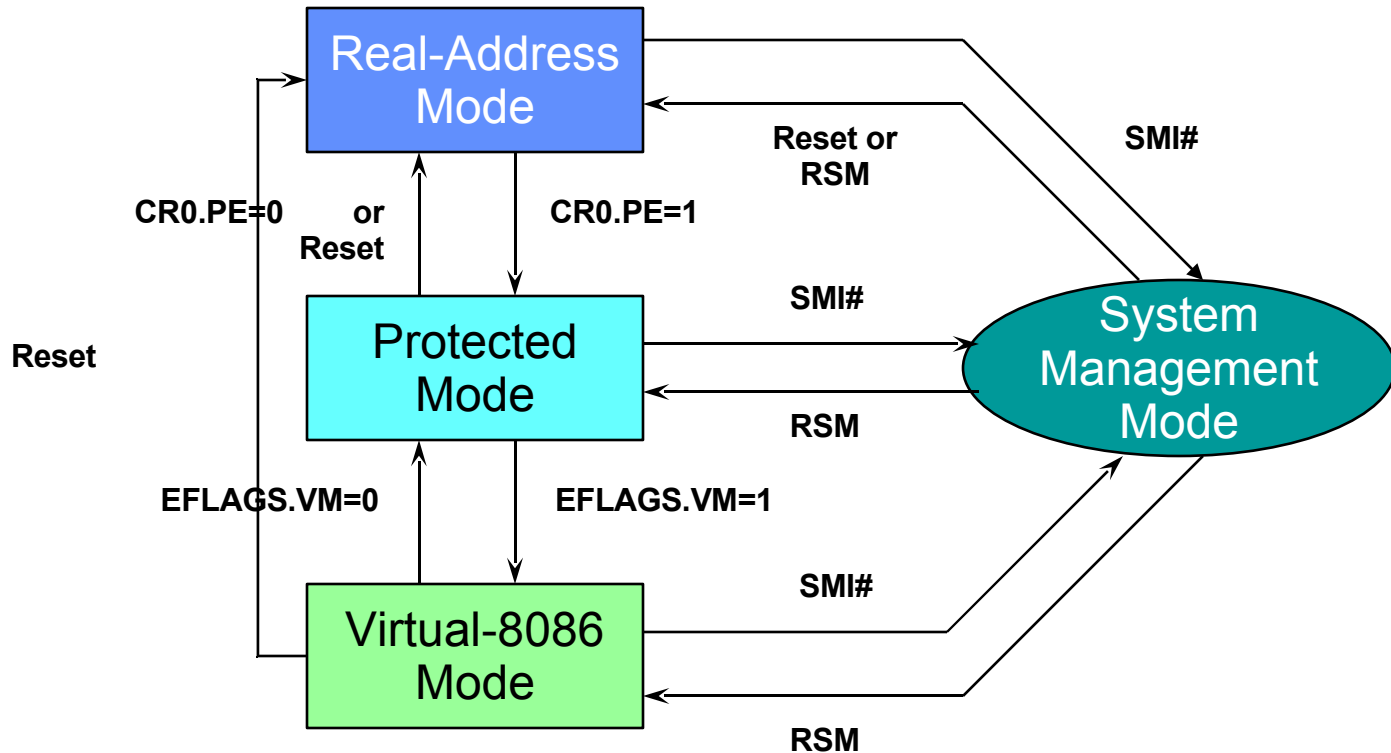
●Virtual-8086 mode

- ◆ Sub-mode of the Protected mode. Allows to execute “real-address mode” 8086 software in a protected, multi-tasking environment. (Not covered here)

●System Management Mode (SMM)

- ◆ This mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security. (Not covered here)

Operation Modes



● 16 bit

- ◆ Real Mode (8086 and up)
- ◆ Protected mode (286 and up)
- ◆ Virtual 8086 mode (386 and up)

● 32 bit

- Protected mode (386 and up)
- Paging enabled/disabled

● 64 bit modes are not shown here

32-Bit vs. 16-Bit Address and Operand Sizes

- The processor can be configured for 32-bit and 16-bit address and operand sizes.
- With 16-Bit address and operand sizes
 - ◆ The maximum segment offset is 0xFFFF
 - ◆ Operand sizes are typically 8-bits, and 16-bits
- With 32-Bit address and operand sizes
 - ◆ The maximum linear address or segment offset is 0xFFFFFFFF
 - ◆ Operand sizes are typically 8-bits, and 32-bits
- Logical Address (far pointer):
 - ◆ 16-bit addressing: 16-bit selector : 16-bit offset
 - ◆ 32-bit addressing: 16-bit selector : 32-bit offset
- In Real-Address Mode the default address and operand sizes are 16-bit
- In Protected Mode the default address and operand sizes are determined by the attributes of the segment registers
- Instruction prefixes allow temporary overrides of the default address and/or operand sizes from within a program

Instruction Set Addressing & Instruction Encoding

Instruction Set Architecture

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");

    return 0;
}
```

Linux Style

```
pushl    %ebp
movl     %esp,%ebp
subl     $0x4,%esp
movl     $0x8043210, (%esp,1)
call     0x8060325 <printf>
movl     $0x0,%eax
leave
ret
```

Windows Style

```
push     ebp
mov      ebp,esp
push     0042201c

call     00401060
xor      eax,eax
mov      esp,ebp
pop      ebp
ret
```

Instruction Set Architecture

- Data transfer
- Common ALU operations
- Control flow
- String
- FP, MMX and SSE instructions
- OS support
- I/O instructions
- “Exotic” instructions (DAA, XLAT, etc.)

Data Transfer Instructions

● MOV Byte / Word / Dword, register to / from register / memory

- ◆ Also Zero/Sign extended moves

● STACK OPERATIONS

◆ PUSH [SOURCE]

$SP = SP - 2$; (SS : SP) = SOURCE ; (16-bit)

$ESP = ESP - 4$; (SS : ESP) = SOURCE ; (32-bit)

◆ POP [DEST]

DEST = (SS : SP) ; $SP = SP + 2$; (16-bit)

DEST = (SS : ESP) ; $ESP = ESP + 4$; (32-bit)

- ◆ PUSHA / POPA, PUSHF / POPF

● I/O instructions

- ◆ IN (I/O port specified by 8-bit immediate or DX)

EAX / AX / AL = Dword / Word / Byte from I/O port

- ◆ OUT (I/O port specified by 8-bit immediate or DX)

I/O Port = Dword / Word / Byte from EAX / AX / AL

ALU and Shift Instructions

●ALU

- ◆ Usual ALU ops on Byte / Word / Dword operands
- ◆ Integer Multiply and Divide Instructions
 - Multiply and Divide use implicit register pairs
- ◆ Chained add / subtract for multi-precision arithmetic
- ◆ Conversion ops for packed and unpacked decimal math

●SHIFTS and ROTATES

- ◆ Logical and arithmetic shifts
- ◆ Rotates through / not through C bit
- ◆ By constant (1) or by count in CL

Later extended to immediate operand in 80386

Control Flow

● Assortment of conditional jumps

- ◆ **Jcc** Byte/Word/Dword displacement
- ◆ CS not modified, $EIP = EIP + \text{signed displacement}$

● **JMP**

- ◆ **SHORT** - one byte displacement
- ◆ **NEAR** - two/four byte displacement
- ◆ **FAR** - Across Segments (Loads CS : EIP)

● **CALL**

- ◆ Like **JMP**
- ◆ Pushes next EIP in **SHORT** and **NEAR** forms
- ◆ Pushes CS : EIP in **FAR** forms

● **RET** (Return)

- ◆ Pops into EIP in **NEAR** form
- ◆ Pops into CS : EIP in **FAR** form
- ◆ **RET immediate** form – pops N bytes from the stack

MOV, ALU, JMP Example

```
int array[100];
int sum = 0;
void sum_array()
{
    int i;
    for(i = 0; i < 100; i++)
    {
        sum += array[i];
    }
}
```

```
COMM    _array:DWORD:064H
_sum     DD      00H DUP (?)
mov      eax, OFFSET FLAT:_array
$L28:    mov      ecx, DWORD PTR [eax]
add      eax, 4
add      DWORD PTR _sum, ecx
cmp      eax, OFFSET FLAT:_array+400
jl       SHORT $L28
```

String Instructions

● String operation components

- ◆ Opcode specifying *ES:EDI op DS:ESI*
 - ◆ Typically preceded by REP (Repeat) Prefix
- ```
for (;Repeat Condition; ECX--)
{
 ES : EDI op DS : ESI;
 EDI = EDI + d*operand_size;
 ESI = ESI + d*operand_size;
}
```

## ● d = 1 or -1

- ◆ Value selected by Direction Flag EFLAGS.DF

## ● INS, OUTS, MOVS, LODS, STOS, CMPS, SCAS.

- ◆ Operand size is one of 8/16/32

## ● Repeat Condition is test of Zero Flag

- ◆ Can be test if set or test if clear
- ◆ Zero Flag set by  $ECX == 0$  or by op (CMPS)

# Repeat MOVs Example

```
struct str
{
 char c[1000];
} src, dst;
```

```
void move()
{
 dst = src;
}
```

*Save registers  
on stack*

*Copy*

*Restore registers  
from stack*

*Return from function*

```
COMM _src:BYTE:03e8H
COMM _dst:BYTE:03e8H
...
```

```
push ebp
mov ebp, esp
push esi
push edi
push ecx
mov ecx, 250
mov esi, OFFSET FLAT:_src
mov edi, OFFSET FLAT:_dst
rep movsd
pop ecx
pop edi
pop esi
pop ebp
ret
```

# Assembly Version

## ●MS/Intel

*“programming tool”*

- ◆ Case insensitive
- ◆ Operand size implicit
- ◆ Destination on left

## ●Unix

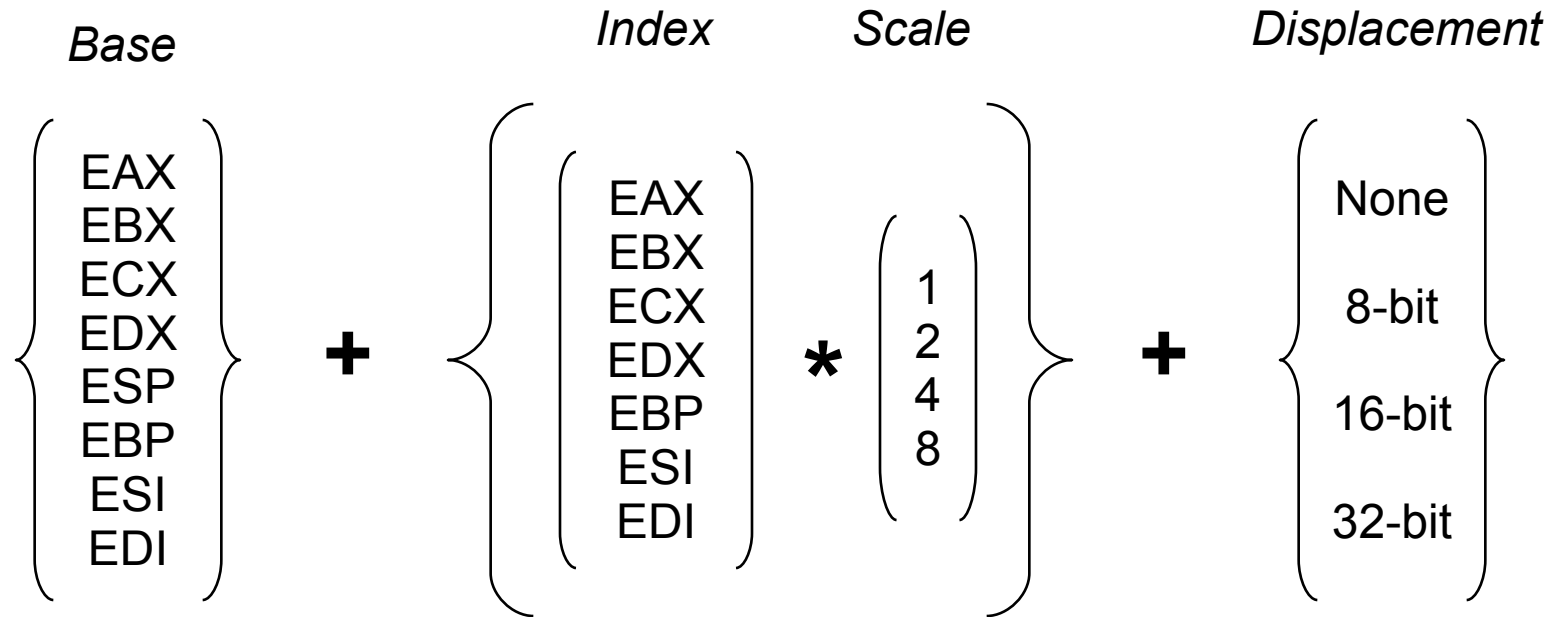
*“compiler back-end”*

- ◆ Case sensitive
- ◆ Operand size explicit
- ◆ Destination on right

- Assembler should be smart enough to handle prefix when 16 bit operand is used in 32 bit code...

# Addressing Modes

# Effective Address Computation



$$\text{Offset} = \text{Base} + (\text{Index} * \text{Scale}) + \text{Displacement}$$

# Addressing Mode Examples (1)

## ● Displacement

- ◆ Direct (uncomputed) offset to the operand

```
int a, b;
```

```
b = a;
```

```
mov eax, DWORD PTR _a
```

```
mov DWORD PTR _b, eax
```

## ● Base

- ◆ Indirect offset to the operand

```
int a, *p;
```

```
a = *p;
```

```
mov eax, DWORD PTR _p
```

```
mov ebx, DWORD PTR [eax]
```

```
mov DWORD PTR _a, ebx
```

# Addressing Mode Examples (2)

## ●Base + Displacement

- ◆ Indexing into an array when the element size is not 2, 4, or 8 bytes.
- ◆ Work with stack

```
void func(int a)
{
 int b = a;
}
```

```
mov ecx, DWORD PTR [ebp+8]
mov DWORD PTR [ebp-4], ecx
```

- ◆ Accessing a field of a record

```
struct {int x, y;} *str_p;
int z = str_p->y;
```

```
mov edx, DWORD PTR _str_p
mov eax, DWORD PTR [edx+4]
```

# Addressing Mode Examples (3)

## ●(Index \* Scale) + Displacement

- ◆ Efficient indexing into array when the element size is 2, 4, or 8 bytes.

```
int array[100];
int c, i;
c = array[i];
```

```
mov edx, DWORD PTR _i
mov eax, DWORD PTR _array[edx*4]
mov DWORD PTR _c, eax
```

# Addressing Mode Examples (4)

## ● Base + (Index \* Scale) + Displacement

- ◆ Access to several instances of an array of records
- ◆ Efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

```
struct {int x, y;} arr_str[100];
int d, i;
d = arr_str[i].y;
```

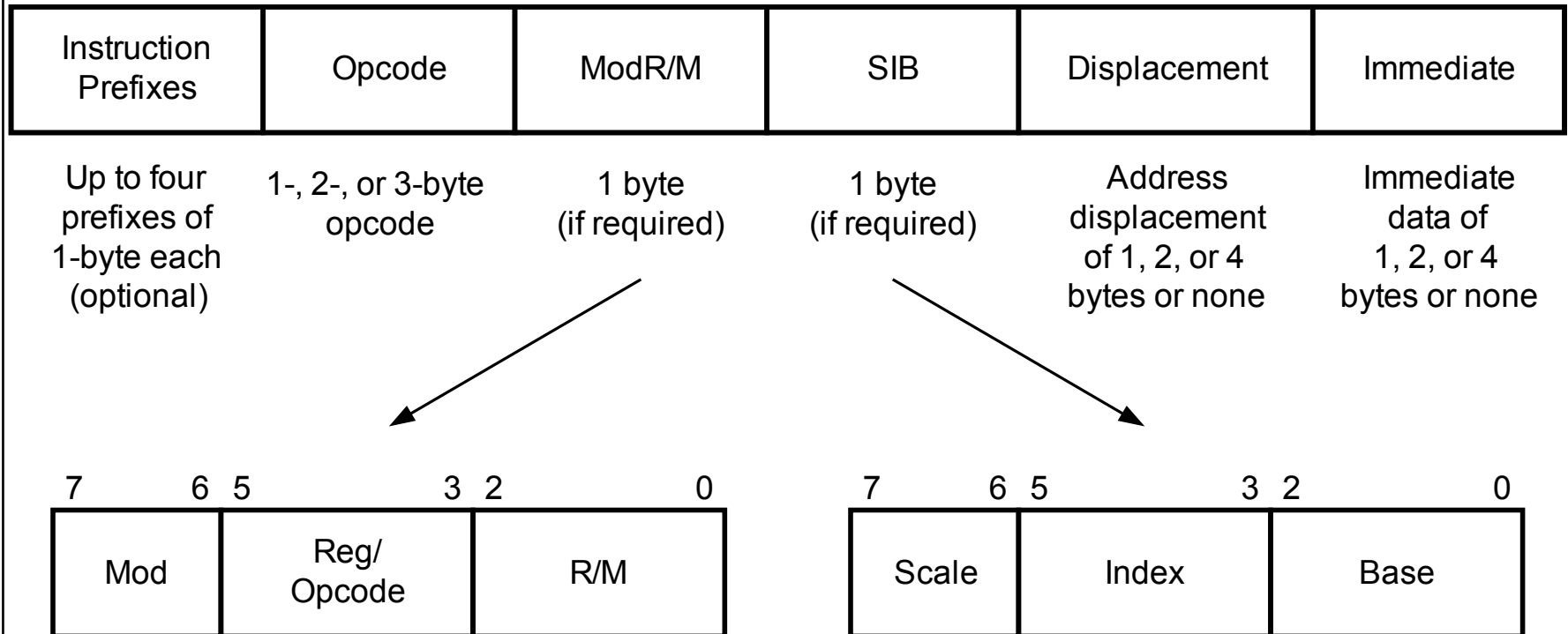
```
mov ecx, DWORD PTR _i
mov ebx, DWORD PTR _arr_str
mov edx, DWORD PTR [ebx+ecx*8+4]
mov DWORD PTR _d, edx
```

# Instruction Encoding

# Instruction Format

- Variable length instructions!
- Remember - only 2 explicit operands - source and destination may be the same
- Machine language
  - ◆ Zero or more prefixes
  - ◆ Opcode
  - ◆ Mod/Rm
  - ◆ Displacement
  - ◆ Immediate
- 8/16/32 Handling
  - ◆ Instruction encoding differs for 8-bit and 16/32 bit operands
  - ◆ Same instruction encoding is used for 16 and 32 bit operands
  - ◆ Current code segment defines if processor runs 16 or 32 bit code
  - ◆ In 16 bit code “word” operands indicate 16 bit operand,  
In 32 bit code “word” operands indicate 32 bit operand.
  - ◆ This can be toggled by the operand-size prefix (0x66)

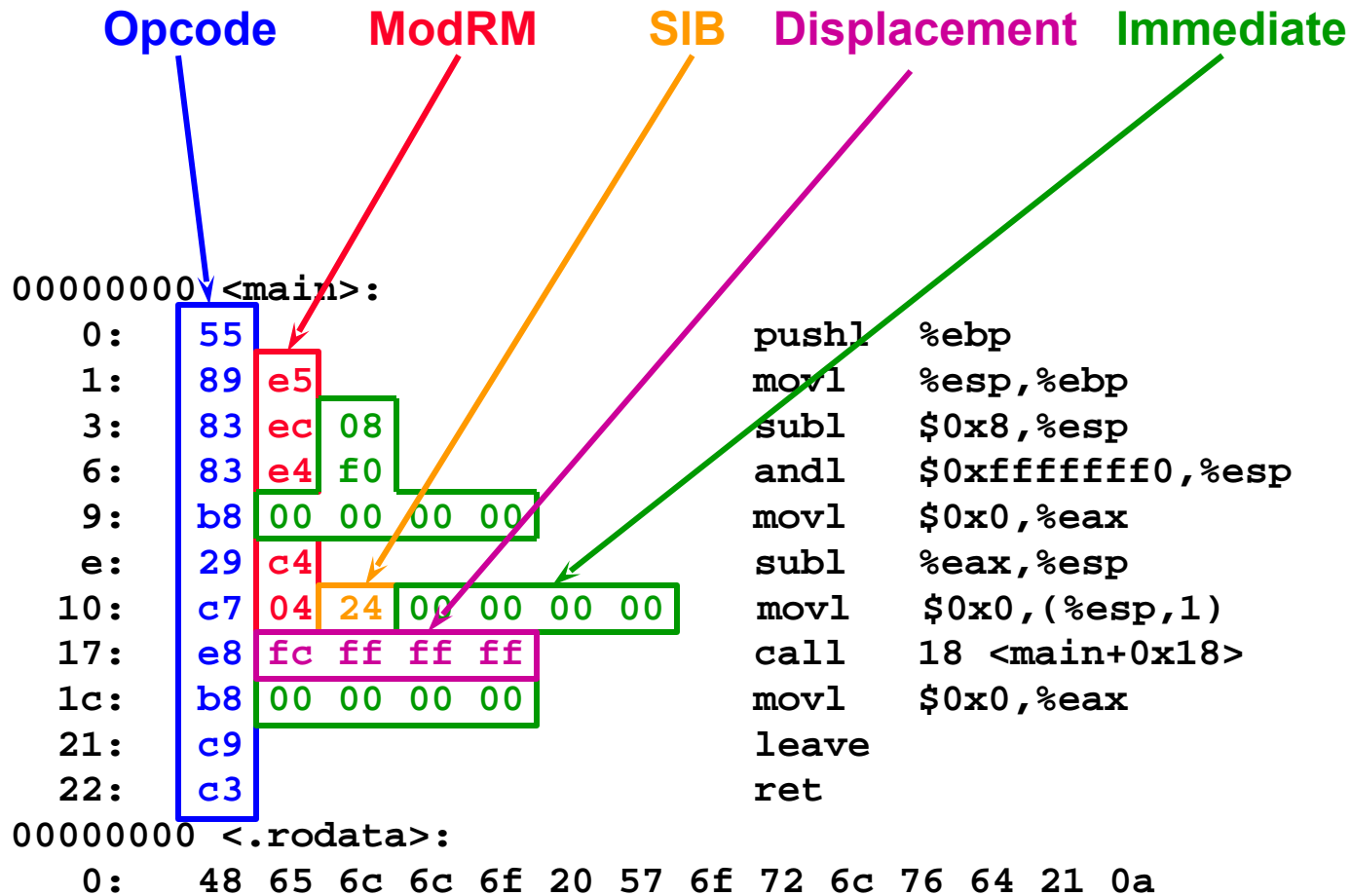
# Instruction Format



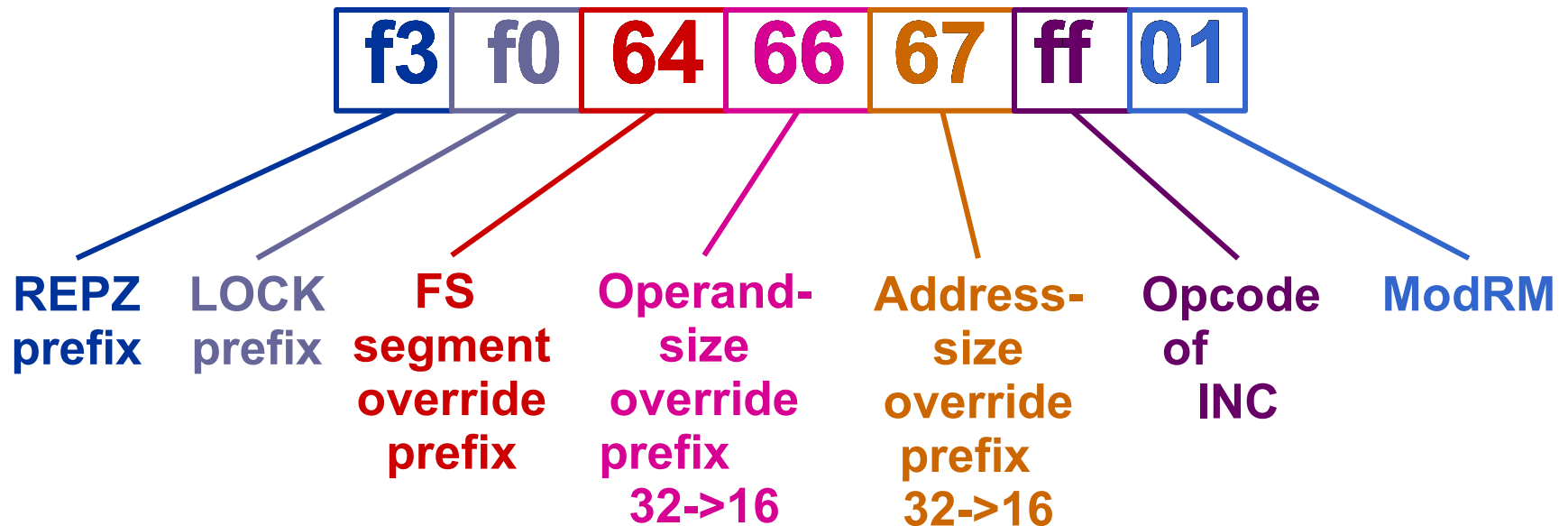
Minimal instruction size = 1 byte

Maximal instruction size = 15 bytes

# Instruction Format Example

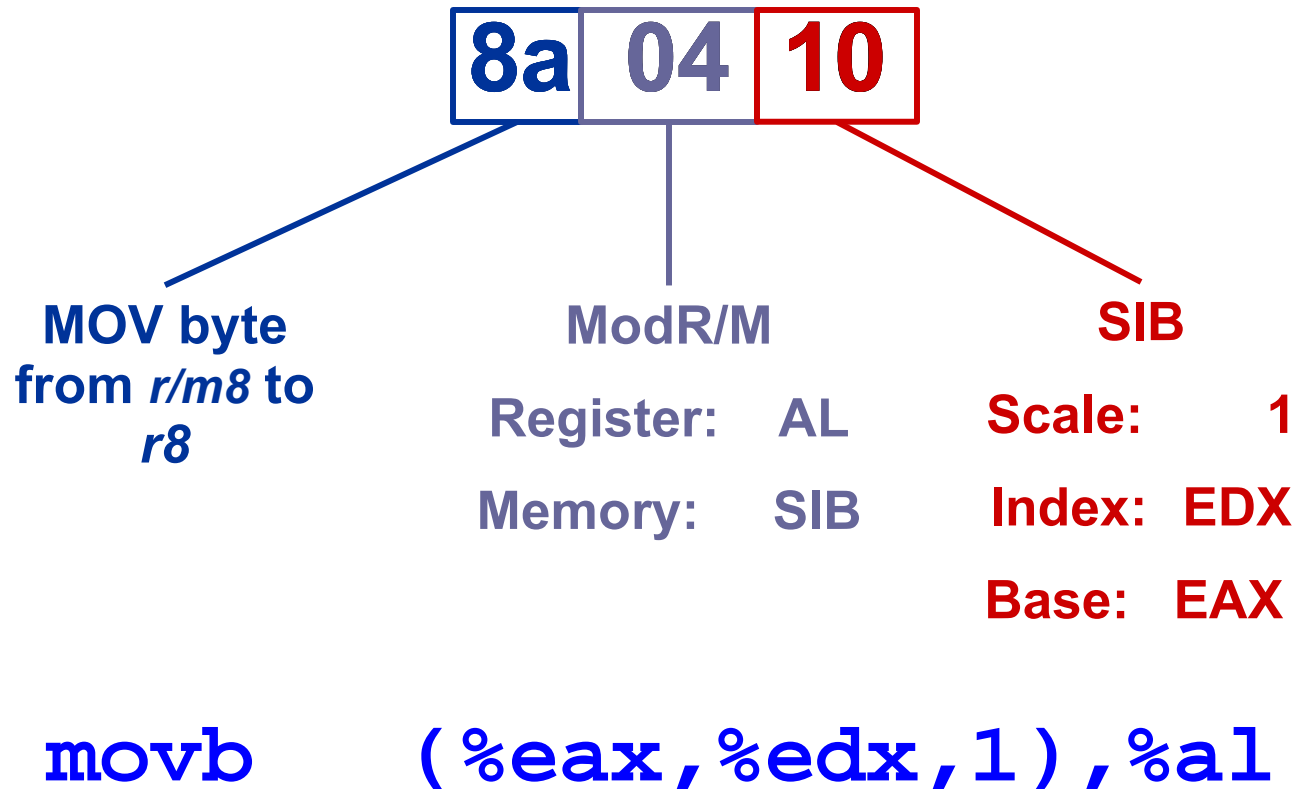


# Instruction Encoding Example (1)



`repz lock incw %fs:(%bx,%di,1)`

# Instruction Encoding Example (2)



# MOD/RM Table: 32-bit addressing table (1)

|                 |      |      |      |      |      |      |      |      |
|-----------------|------|------|------|------|------|------|------|------|
| r8(/r)          | AL   | CL   | DL   | BL   | AH   | CH   | DH   | BH   |
| r16(/r)         | AX   | CX   | DX   | BX   | SP   | BP   | SI   | DI   |
| r32(/r)         | EAX  | ECX  | EDX  | EBX  | ESP  | EBP  | ESI  | EDI  |
| mm(/r)          | MM0  | MM1  | MM2  | MM3  | MM4  | MM5  | MM6  | MM7  |
| xmm(/r)         | XMM0 | XMM1 | XMM2 | XMM3 | XMM4 | XMM5 | XMM6 | XMM7 |
| /digit (Opcode) | 0    | 1    | 2    | 3    | 4    | 5    | 6    | 7    |
| REG =           | 000  | 001  | 010  | 011  | 100  | 101  | 110  | 111  |

| Effective Address | Mod | R/M | Value of ModR/M Byte (in Hexadecimal) |    |    |    |    |    |    |    |
|-------------------|-----|-----|---------------------------------------|----|----|----|----|----|----|----|
| [EAX]             | 00  | 000 | 00                                    | 08 | 10 | 18 | 20 | 28 | 30 | 38 |
| [ECX]             |     | 001 | 01                                    | 09 | 11 | 19 | 21 | 29 | 31 | 39 |
| [EDX]             |     | 010 | 02                                    | 0A | 12 | 1A | 22 | 2A | 32 | 3A |
| [EBX]             |     | 011 | 03                                    | 0B | 13 | 1B | 23 | 2B | 33 | 3B |
| [--][--]          |     | 100 | 04                                    | 0C | 14 | 1C | 24 | 2C | 34 | 3C |
| disp32            |     | 101 | 05                                    | 0D | 15 | 1D | 25 | 2D | 35 | 3D |
| [ESI]             |     | 110 | 06                                    | 0E | 16 | 1E | 26 | 2E | 36 | 3E |
| [EDI]             |     | 111 | 07                                    | 0F | 17 | 1F | 27 | 2F | 37 | 3F |
| [EAX]+disp8       | 01  | 000 | 40                                    | 48 | 50 | 58 | 60 | 68 | 70 | 78 |
| [ECX]+disp8       |     | 001 | 41                                    | 49 | 51 | 59 | 61 | 69 | 71 | 79 |
| [EDX]+disp8       |     | 010 | 42                                    | 4A | 52 | 5A | 62 | 6A | 72 | 7A |
| [EBX]+disp8       |     | 011 | 43                                    | 4B | 53 | 5B | 63 | 6B | 73 | 7B |
| [--][--]+disp8    |     | 100 | 44                                    | 4C | 54 | 5C | 64 | 6C | 74 | 7C |
| [EBP]+disp8       |     | 101 | 45                                    | 4D | 55 | 5D | 65 | 6D | 75 | 7D |
| [ESI]+disp8       |     | 110 | 46                                    | 4E | 56 | 5E | 66 | 6E | 76 | 7E |
| [EDI]+disp8       |     | 111 | 47                                    | 4F | 57 | 5F | 67 | 6F | 77 | 7F |

# MOD/RM Table: 32-bit addressing table (2)

|                 |      |      |      |      |      |      |      |      |
|-----------------|------|------|------|------|------|------|------|------|
| r8(/r)          | AL   | CL   | DL   | BL   | AH   | CH   | DH   | BH   |
| r16(/r)         | AX   | CX   | DX   | BX   | SP   | BP   | SI   | DI   |
| r32(/r)         | EAX  | ECX  | EDX  | EBX  | ESP  | EBP  | ESI  | EDI  |
| mm(/r)          | MM0  | MM1  | MM2  | MM3  | MM4  | MM5  | MM6  | MM7  |
| xmm(/r)         | XMM0 | XMM1 | XMM2 | XMM3 | XMM4 | XMM5 | XMM6 | XMM7 |
| /digit (Opcode) | 0    | 1    | 2    | 3    | 4    | 5    | 6    | 7    |
| REG =           | 000  | 001  | 010  | 011  | 100  | 101  | 110  | 111  |

| Effective Address  | Mod | R/M | Value of ModR/M Byte (in Hexadecimal) |    |    |    |    |    |    |    |
|--------------------|-----|-----|---------------------------------------|----|----|----|----|----|----|----|
| [EAX]+disp32       | 10  | 000 | 80                                    | 88 | 90 | 98 | A0 | A8 | B0 | B8 |
| [ECX]+disp32       |     | 001 | 81                                    | 89 | 91 | 99 | A1 | A9 | B1 | B9 |
| [EDX]+disp32       |     | 010 | 82                                    | 8A | 92 | 9A | A2 | AA | B2 | BA |
| [EBX]+disp32       |     | 011 | 83                                    | 8B | 93 | 9B | A3 | AB | B3 | BB |
| [--][--]+disp32    |     | 100 | 84                                    | 8C | 94 | 9C | A4 | AC | B4 | BC |
| [EBP]+disp32       |     | 101 | 85                                    | 8D | 95 | 9D | A5 | AD | B5 | BD |
| [ESI]+disp32       |     | 110 | 86                                    | 8E | 96 | 9E | A6 | AE | B6 | BE |
| [EDI]+disp32       |     | 111 | 87                                    | 8F | 97 | 9F | A7 | AF | B7 | BF |
| EAX/AX/AL/MM0/XMM0 | 11  | 000 | C0                                    | C8 | D0 | D8 | E0 | E8 | F0 | F8 |
| ECX/CX/CL/MM1/XMM1 |     | 001 | C1                                    | C9 | D1 | D9 | E1 | E9 | F1 | F9 |
| EDX/DX/DL/MM2/XMM2 |     | 010 | C2                                    | CA | D2 | DA | E2 | EA | F2 | FA |
| EBX/BX/BL/MM3/XMM3 |     | 011 | C3                                    | CB | D3 | DB | E3 | EB | F3 | FB |
| ESP/SP/AH/MM4/XMM4 |     | 100 | C4                                    | CC | D4 | DC | E4 | EC | F4 | FC |
| EBP/BP/CH/MM5/XMM5 |     | 101 | C5                                    | CD | D5 | DD | E5 | ED | F5 | FD |
| ESI/SI/DH/MM6/XMM6 |     | 110 | C6                                    | CE | D6 | DE | E6 | EE | F6 | FE |
| EDI/DI/BH/MM7/XMM7 |     | 111 | C7                                    | CF | D7 | DF | E7 | EF | F7 | FF |

# SIB Table: 32-bit addressing table (1)

| r32          |    |       | EAX                                | ECX | EDX | EBX | ESP | [*] | ESI | EDI |
|--------------|----|-------|------------------------------------|-----|-----|-----|-----|-----|-----|-----|
| Base =       |    |       | 0                                  | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
| Base =       |    |       | 000                                | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Scaled Index | SS | Index | Value of SIB Byte (in Hexadecimal) |     |     |     |     |     |     |     |
| [EAX]        | 00 | 000   | 00                                 | 01  | 02  | 03  | 04  | 05  | 06  | 07  |
| [ECX]        |    | 001   | 08                                 | 09  | 0A  | 0B  | 0C  | 0D  | 0E  | 0F  |
| [EDX]        |    | 010   | 10                                 | 11  | 12  | 13  | 14  | 15  | 16  | 17  |
| [EBX]        |    | 011   | 18                                 | 19  | 1A  | 1B  | 1C  | 1D  | 1E  | 1F  |
| none         |    | 100   | 20                                 | 21  | 22  | 23  | 24  | 25  | 26  | 27  |
| [EBP]        |    | 101   | 28                                 | 29  | 2A  | 2B  | 2C  | 2D  | 2E  | 2F  |
| [ESI]        |    | 110   | 30                                 | 31  | 32  | 33  | 34  | 35  | 36  | 37  |
| [EDI]        |    | 111   | 38                                 | 39  | 3A  | 3B  | 3C  | 3D  | 3E  | 3F  |
| [EAX*2]      | 01 | 000   | 40                                 | 41  | 42  | 43  | 44  | 45  | 46  | 47  |
| [ECX*2]      |    | 001   | 48                                 | 49  | 4A  | 4B  | 4C  | 4D  | 4E  | 4F  |
| [EDX*2]      |    | 010   | 50                                 | 51  | 52  | 53  | 54  | 55  | 56  | 57  |
| [EBX*2]      |    | 011   | 58                                 | 59  | 5A  | 5B  | 5C  | 5D  | 5E  | 5F  |
| none         |    | 100   | 60                                 | 61  | 62  | 63  | 64  | 65  | 66  | 67  |
| [EBP*2]      |    | 101   | 68                                 | 69  | 6A  | 6B  | 6C  | 6D  | 6E  | 6F  |
| [ESI*2]      |    | 110   | 70                                 | 71  | 72  | 73  | 74  | 75  | 76  | 77  |
| [EDI*2]      |    | 111   | 78                                 | 79  | 7A  | 7B  | 7C  | 7D  | 7E  | 7F  |

# SIB Table: 32-bit addressing table (2)

| r32<br>Base =<br>Base = |    |       | EAX<br>0<br>000                    | ECX<br>1<br>001 | EDX<br>2<br>010 | EBX<br>3<br>011 | ESP<br>4<br>100 | [*]<br>5<br>101 | ESI<br>6<br>110 | EDI<br>7<br>111 |
|-------------------------|----|-------|------------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| Scaled Index            | SS | Index | Value of SIB Byte (in Hexadecimal) |                 |                 |                 |                 |                 |                 |                 |
| [EAX*4]                 | 10 | 000   | 80                                 | 81              | 82              | 83              | 84              | 85              | 86              | 87              |
| [ECX*4]                 |    | 001   | 88                                 | 89              | 8A              | 8B              | 8C              | 8D              | 8E              | 8F              |
| [EDX*4]                 |    | 010   | 90                                 | 91              | 92              | 93              | 94              | 95              | 96              | 97              |
| [EBX*4]                 |    | 011   | 98                                 | 99              | 9A              | 9B              | 9C              | 9D              | 9E              | 9F              |
| none                    |    | 100   | A0                                 | A1              | A2              | A3              | A4              | A5              | A6              | A7              |
| [EBP*4]                 |    | 101   | A8                                 | A9              | AA              | AB              | AC              | AD              | AE              | AF              |
| [ESI*4]                 |    | 110   | B0                                 | B1              | B2              | B3              | B4              | B5              | B6              | B7              |
| [EDI*4]                 |    | 111   | B8                                 | B9              | BA              | BB              | BC              | BD              | BE              | BF              |
| [EAX*8]                 | 11 | 000   | C0                                 | C1              | C2              | C3              | C4              | C5              | C6              | C7              |
| [ECX*8]                 |    | 001   | C8                                 | C9              | CA              | CB              | CC              | CD              | CE              | CF              |
| [EDX*8]                 |    | 010   | D0                                 | D1              | D2              | D3              | D4              | D5              | D6              | D7              |
| [EBX*8]                 |    | 011   | D8                                 | D9              | DA              | DB              | DC              | DD              | DE              | DF              |
| none                    |    | 100   | E0                                 | E1              | E2              | E3              | E4              | E5              | E6              | E7              |
| [EBP*8]                 |    | 101   | E8                                 | E9              | EA              | EB              | EC              | ED              | EE              | EF              |
| [ESI*8]                 |    | 110   | F0                                 | F1              | F2              | F3              | F4              | F5              | F6              | F7              |
| [EDI*8]                 |    | 111   | F8                                 | F9              | FA              | FB              | FC              | FD              | FE              | FF              |

# Instruction Encoding Example (3)

MOV EAX, [EBP + 0x4] – move the content of memory located at [EBP] + 0x4 to register EAX

8B /r MOV r32, r/m32 (Move r/m32 to r32)

r32 = EAX REG = 0

Effective Address = [EBP]+disp8 Mod = 01 R/M = 101 ModRM = 0x45

0x8B 0x45 0x4

MOV EAX, [ESP + 0x4] – move the content of memory located at [ESP] + 0x4 to register EAX

r32 = EAX REG = 0

Effective Address = [ESP]+disp8 Mod = 01 R/M = 100 ModRM = 0x44

SIB required Base (ESP) = 4, Index = none SIB = 0x24, 0x64, 0xA4, 0xE4

0x8B 0x44 0x24 0x04

0x8B 0x44 0x64 0x04

0x8B 0x44 0xA4 0x04

0x8B 0x44 0xE4 0x04

# Instruction Encoding Example (4)

MOV EBX, EAX – move the content of register EAX to register EBX

Two possibilities:

89 /r MOV r/m32, r32  
(Move r32 to r/m32)

8B /r MOV r32, r/m32  
(Move r/m32 to r32)

EBX corresponds to r/m32operand  
EAX corresponds to r32operand

EBX corresponds to r32 operand  
EAX corresponds to r/m32 operand

ModRM = 0xC3

ModRM = 0xD8

0x89 0xC3

0x8B 0xD8

# **Protected-Mode Memory Management**

# Overview

## ● Memory Management in IA-32

### ◆ Segmentation

Provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another.

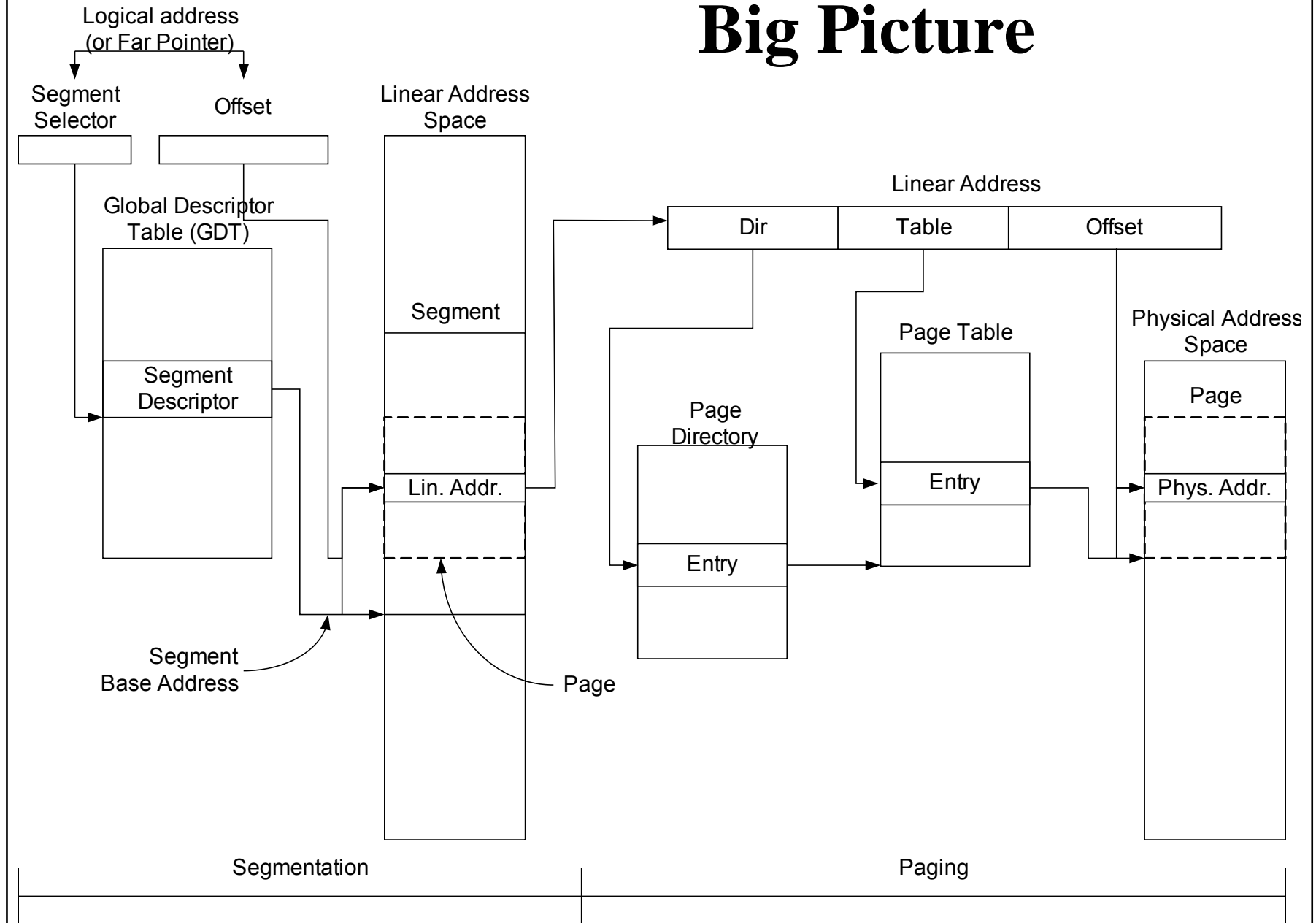
Always enabled

### ◆ 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.

Can be enabled/disabled

# Big Picture

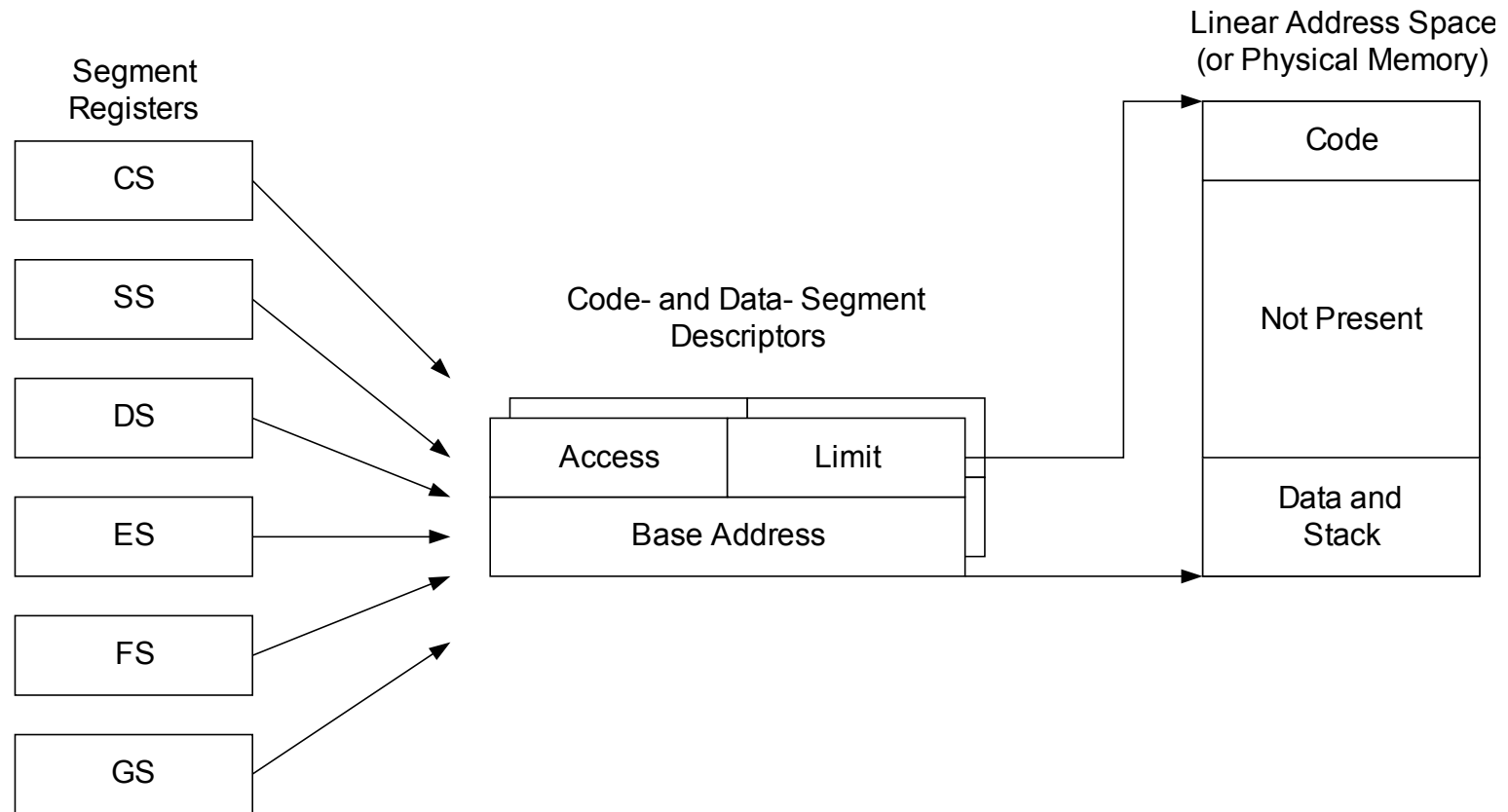


# Segmentation

# Segmentation

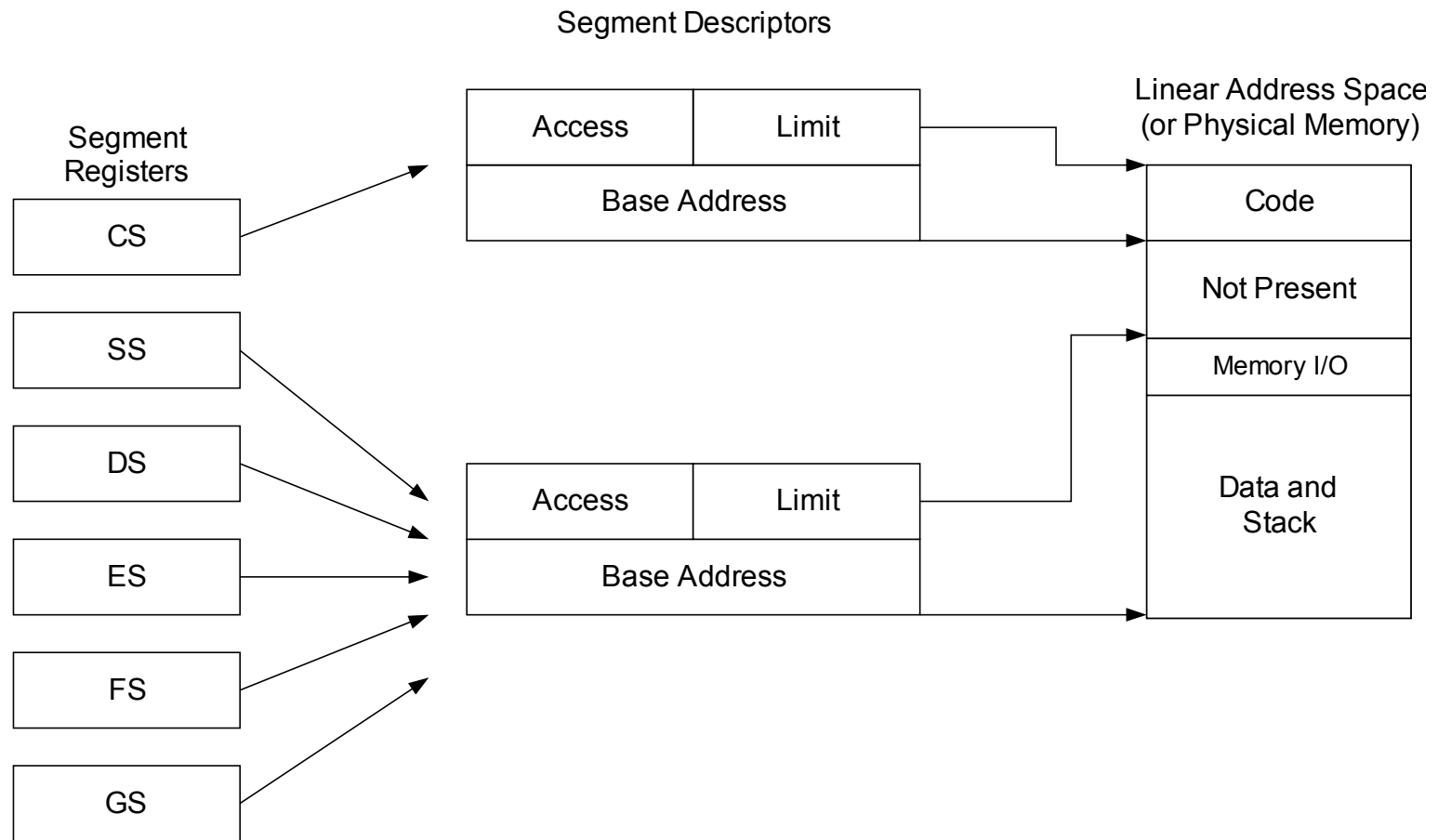
- “**Linear Address Space**” is divided into “**segments**” holding
  - ◆ Code, data and stack for programs
  - ◆ System data structures (TSS, LDT)
- **Logical Address (or far pointer)** = Segment : Offset
- **Segment Selector**
  - ◆ Provides an offset into a descriptor table to a data structure called a segment descriptor
- **Segment Descriptor**
  - ◆ Specifies the size, access rights, privilege level, type and base address.
- **Segmentation**
  - ◆ Mapping from Logical Address to Linear Address
  - ◆  $\text{Linear Address} = \text{Segment\_Base} + \text{Offset}$
- 6 active segment registers
  - ◆ **CS**: code, **DS**: data, **SS**: stack, **ES**: extra. **FS**, **GS**: new extra

# Basic Flat Model



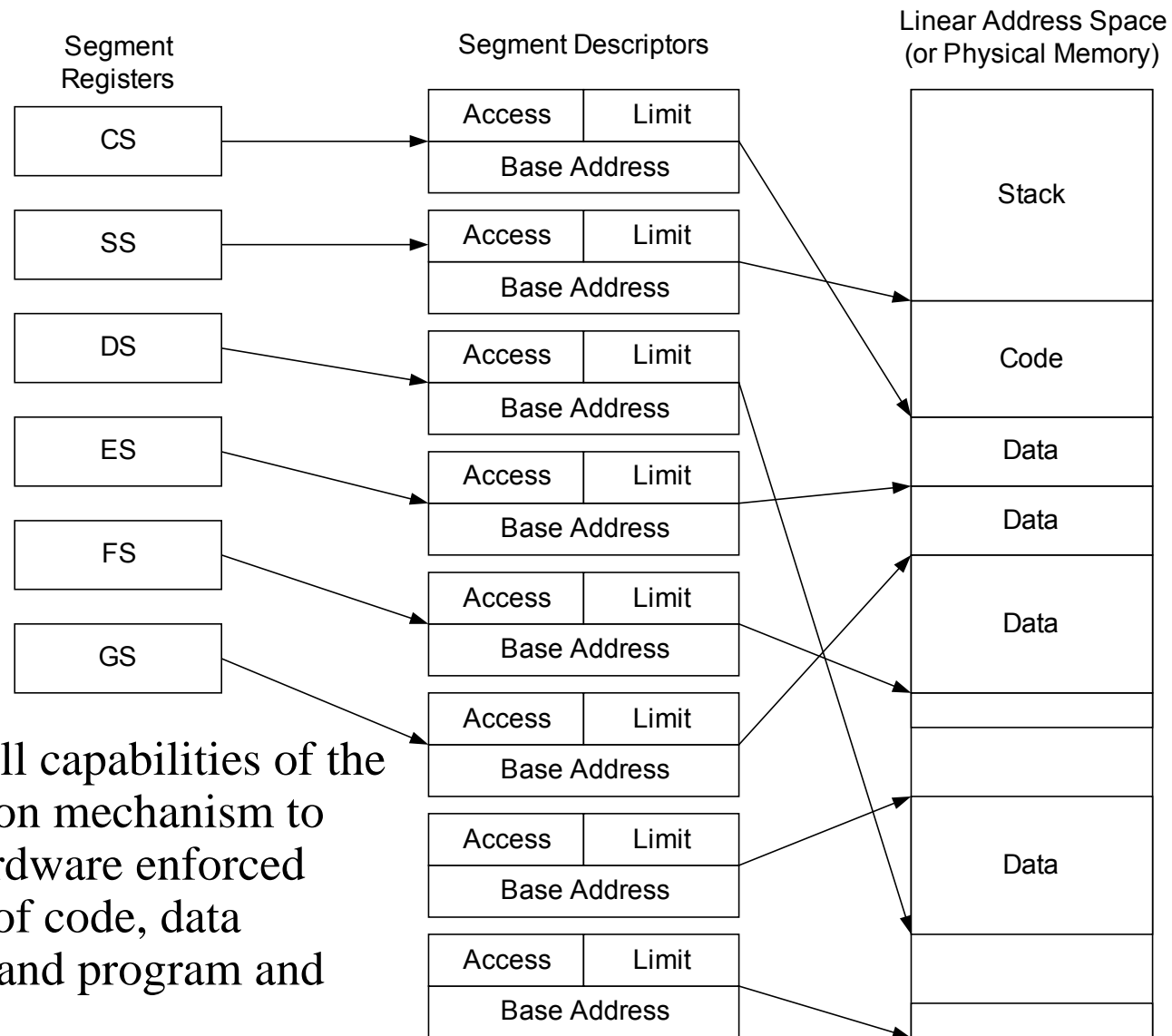
- Simplest memory model
- Operating System and application programs have access to a continuous, unsegmented address space

# Protected Flat Model



- Segment limits are set to include only the range of addresses for which physical memory actually exists.
- Minimal level of hardware protection against some kinds of program bugs.

# A General Segmentation Model



- Uses the full capabilities of the segmentation mechanism to provide hardware enforced protection of code, data structures, and program and tasks.

# Linear Address Computation

## ● Real Mode

Segment Base = segment\*16

Linear address = segment\*16 + offset

Last address:

$$0xFFFF * 16 + 0xFFFF = 0x10FFEF$$

(1M+64K-16)

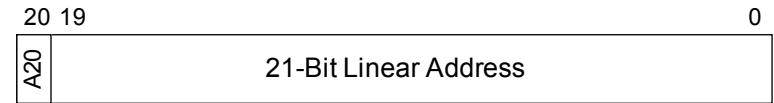
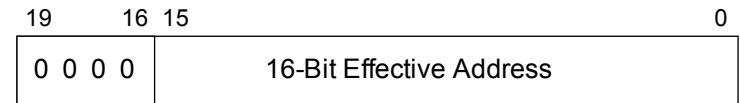
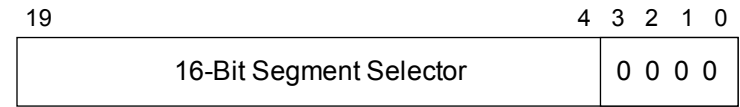
Base

+

Offset

=

Linear Address



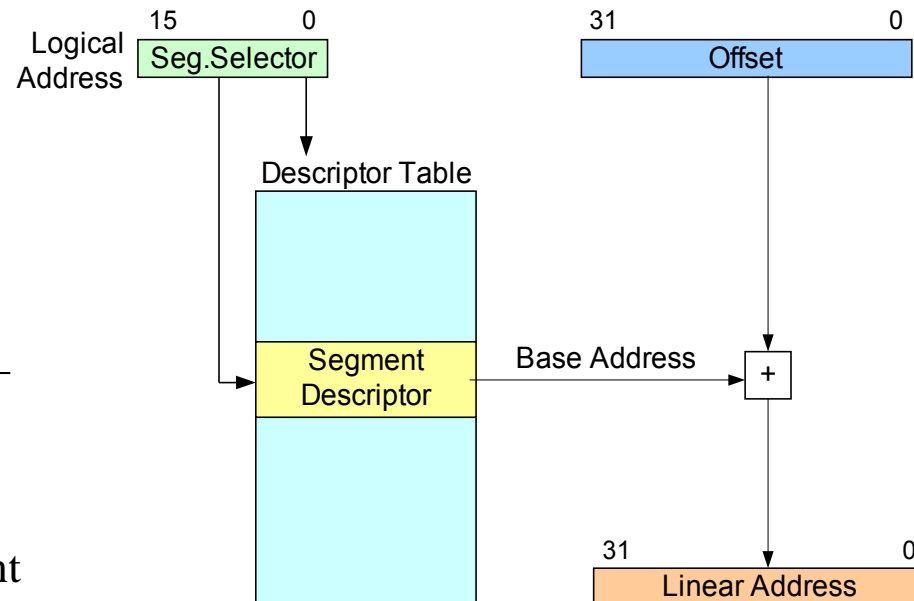
## ● Protected Mode

Linear address =

Descriptor\_Table[f(segment)].base +  
offset

Coverage

4GB address space. 1 segment is sufficient



# Segment Registers

Visible Part

Hidden Part

| Segment Selector | Base Address, Limit, Access Information |    |
|------------------|-----------------------------------------|----|
|                  |                                         | CS |
|                  |                                         | SS |
|                  |                                         | DS |
|                  |                                         | ES |
|                  |                                         | FS |
|                  |                                         | GS |

- Code-segment (CS), data-segment (DS), and stack-segment (SS) registers **must be loaded** with **valid** segment selectors
- ES, FS, and GS can be used to make additional data segments available to task.
- When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register.
- Instructions:
  - ◆ Direct load: MOV SR, POP SR, LDS, LES, LSS, LGS, and LFS instructions.
  - ◆ Implied load: far CALL, JMP, and RET; SYSENTER and SYSEXIT; IRET, INT $n$ , INTO and INT3 instructions.

# Segment Selector Format



Table Indicator

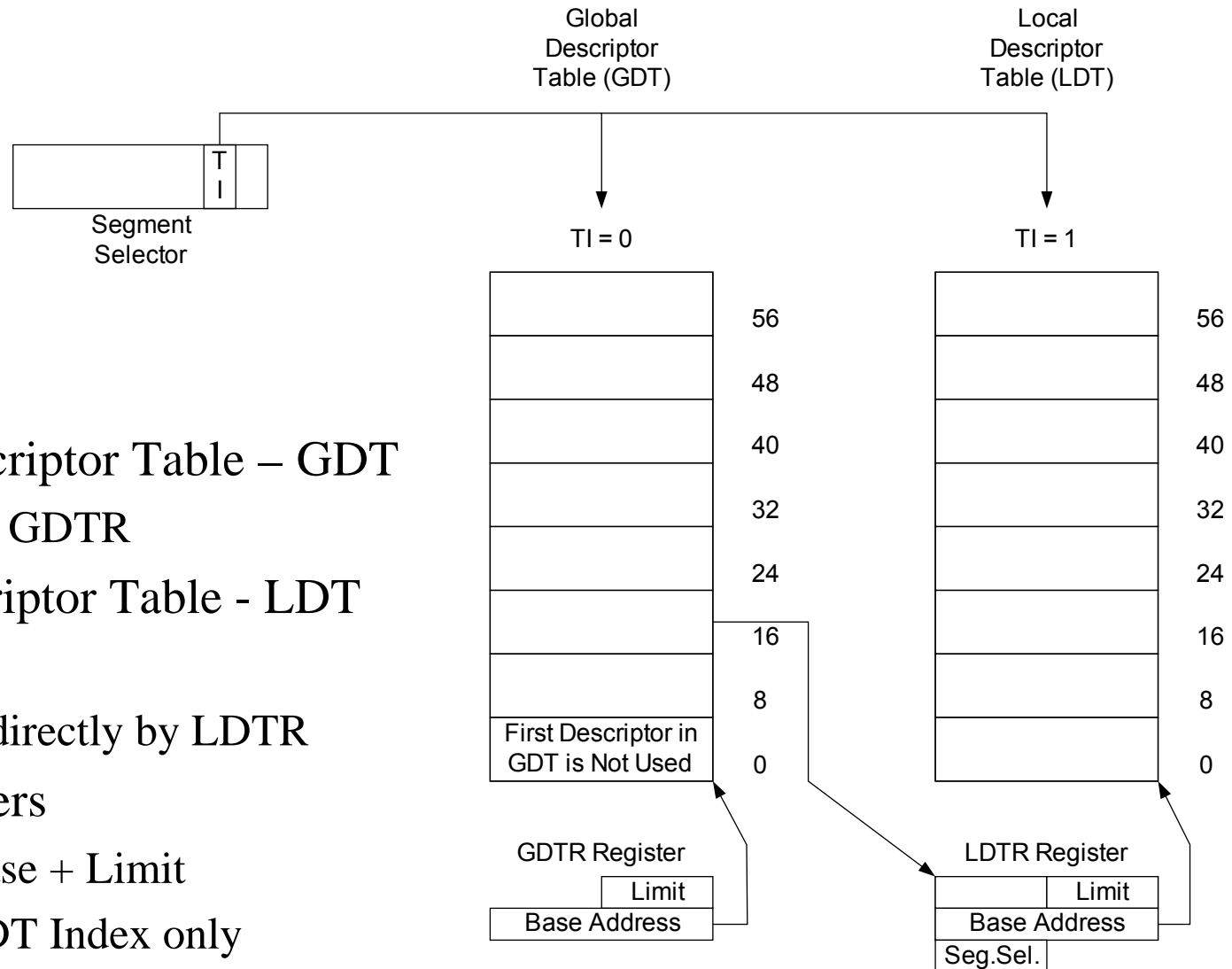
0 = GDT

1 = LDT

Requested Privilege Level (RPL)

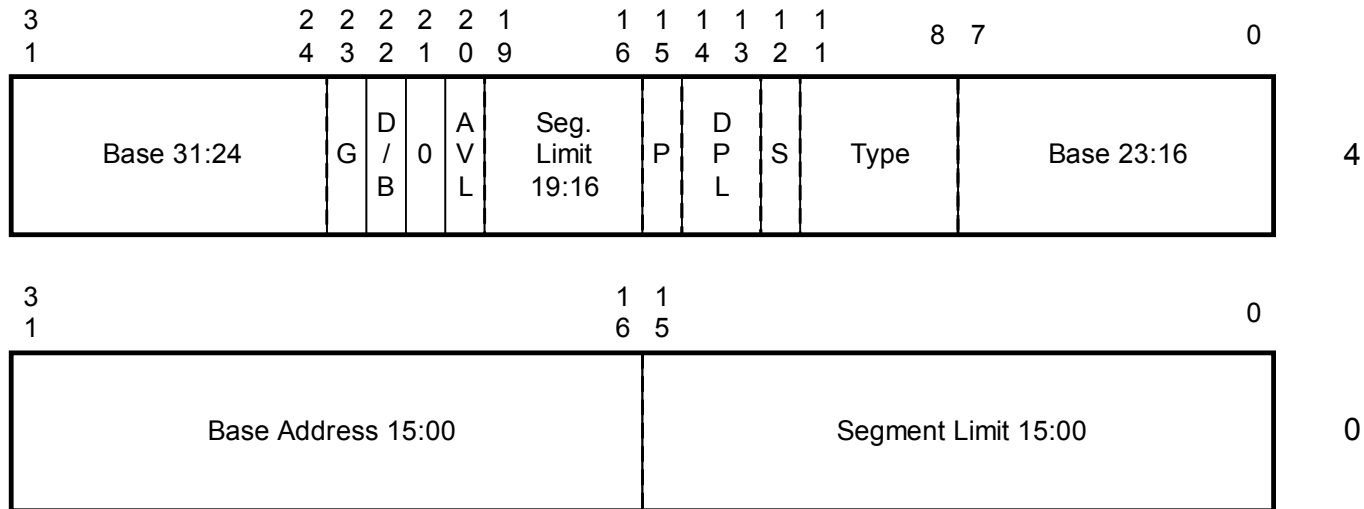
- 13 bit index (8K entries)
- 1 bit table index - GDT (0) or LDT (1)
- 2 bit “requested” privilege level – 0, 1, 2, 3

# Global and Local Descriptor Tables



- Global Descriptor Table – GDT
  - Pointed by GDTR
- Local Descriptor Table - LDT
  - Per task
  - Pointed indirectly by LDTR
- Table pointers
  - GDTR: Base + Limit
  - LDTR: GDT Index only

# Segment Descriptor Format



- 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

# Code- and Data-Segment Types

| Type Field |    |         |        |        | Descriptor<br>Type | Description                        |
|------------|----|---------|--------|--------|--------------------|------------------------------------|
| Decimal    | 11 | 10<br>E | 9<br>W | 8<br>A |                    |                                    |
| 0          | 0  | 0       | 0      | 0      | Data               | Read-Only                          |
| 1          | 0  | 0       | 0      | 1      | Data               | Read-Only, accessed                |
| 2          | 0  | 0       | 1      | 0      | Data               | Read/Write                         |
| 3          | 0  | 0       | 1      | 1      | Data               | Read/Write, accessed               |
| 4          | 0  | 1       | 0      | 0      | Data               | Read-Only, expand-down             |
| 5          | 0  | 1       | 0      | 1      | Data               | Read-Only, expand-down, accessed   |
| 6          | 0  | 1       | 1      | 0      | Data               | Read/Write, expand-down            |
| 7          | 0  | 1       | 1      | 1      | Data               | Read/Write, expand-down, accessed  |
|            |    | C       | R      | A      |                    |                                    |
| 8          | 1  | 0       | 0      | 0      | Code               | Execute-Only                       |
| 9          | 1  | 0       | 0      | 1      | Code               | Execute-Only, accessed             |
| 10         | 1  | 0       | 1      | 0      | Code               | Execute/Read                       |
| 11         | 1  | 0       | 1      | 1      | Code               | Execute/Read, accessed             |
| 12         | 1  | 1       | 0      | 0      | Code               | Execute-Only, conforming           |
| 13         | 1  | 1       | 0      | 1      | Code               | Execute-Only, conforming, accessed |
| 14         | 1  | 1       | 1      | 0      | Code               | Execute/Read, conforming           |
| 15         | 1  | 1       | 1      | 1      | Code               | Execute/Read, conforming, accessed |

# System Descriptor Types

| Type Field |    |    |   |   | Description            |
|------------|----|----|---|---|------------------------|
| Decimal    | 11 | 10 | 9 | 8 |                        |
| 0          | 0  | 0  | 0 | 0 | Reserved               |
| 1          | 0  | 0  | 0 | 1 | 16-Bit TSS (Available) |
| 2          | 0  | 0  | 1 | 0 | LDT                    |
| 3          | 0  | 0  | 1 | 1 | 16-Bit TSS (Busy)      |
| 4          | 0  | 1  | 0 | 0 | 16-Bit Call Gate       |
| 5          | 0  | 1  | 0 | 1 | Task Gate              |
| 6          | 0  | 1  | 1 | 0 | 16-Bit Interrupt Gate  |
| 7          | 0  | 1  | 1 | 1 | 16-Bit Trap Gate       |
| 8          | 1  | 0  | 0 | 0 | Reserved               |
| 9          | 1  | 0  | 0 | 1 | 32-Bit TSS (Available) |
| 10         | 1  | 0  | 1 | 0 | Reserved               |
| 11         | 1  | 0  | 1 | 1 | 32-Bit TSS (Busy)      |
| 12         | 1  | 1  | 0 | 0 | 32-Bit Call Gate       |
| 13         | 1  | 1  | 0 | 1 | Reserved               |
| 14         | 1  | 1  | 1 | 0 | 32-Bit Interrupt Gate  |
| 15         | 1  | 1  | 1 | 1 | 32-Bit Trap Gate       |

# GDT and LDT in Linux

| Brd 0 GDT [P0] |         |            |          |          |   |   |   |   |   |   |   |   |
|----------------|---------|------------|----------|----------|---|---|---|---|---|---|---|---|
| Indx           | Type    | Descriptor | Bas/Sel  | Lim/Off  | G | D | O | L | P | L | S | T |
| 0              | RESERVD | 00000000   | 00000000 | 00000    | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1              | RESERVD | 00000000   | 00000000 | 00000    | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2              | CODERA  | 00cf9b00   | 0000ffff | 00000000 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | b |
| 3              | DATAWA  | 00cf9300   | 0000ffff | 00000000 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 3 |
| 4              | CODERA  | 00cffb00   | 0000ffff | 00000000 | 1 | 1 | 0 | 0 | 1 | 3 | 1 | b |
| 5              | DATAWA  | 00cff300   | 0000ffff | 00000000 | 1 | 1 | 0 | 0 | 1 | 3 | 1 | 3 |
| 6              | RESERVD | 00000000   | 00000000 | 00000    | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7              | RESERVD | 00000000   | 00000000 | 00000    | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8              | DATAW   | c0409200   | 04000bff | c0000400 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 2 |
| 9              | CODERA  | c0409b0f   | 0000ffff | c00f0000 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | b |
| a              | CODERA  | c0009b0f   | 0000ffff | c00f0000 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | b |
| b              | DATAWA  | c0409300   | 0400ffff | c0000400 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 3 |
| c              | CODER   | 00c09a00   | 00000000 | 00000000 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | a |
| d              | CODER   | 00809a00   | 00000000 | 00000000 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | a |
| e              | DATAW   | 00809200   | 00000000 | 00000000 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 2 |
| f              | DATAW   | 00809200   | 00000000 | 00000000 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 2 |
| 10             | DATAW   | 00809200   | 00000000 | 00000000 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 2 |
| 11             | RESERVD | 00000000   | 00000000 | 00000    | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12             | RESERVD | 00000000   | 00000000 | 00000    | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13             | RESERVD | 00000000   | 00000000 | 00000    | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 14             | BTSS32  | c0008b34   | b800     |          |   |   |   |   |   |   |   |   |
| 15             | LDT     | f80082a8   | 2000     |          |   |   |   |   |   |   |   |   |
| 16             | RESERVD | 00000000   | 0000     |          |   |   |   |   |   |   |   |   |
| 17             | RESERVD | 00000000   | 0000     |          |   |   |   |   |   |   |   |   |
| 18             | RESERVD | 00000000   | 0000     |          |   |   |   |   |   |   |   |   |
| 19             | RESERVD | 00000000   | 0000     |          |   |   |   |   |   |   |   |   |
| 1a             | RESERVD | 00000000   | 0000     |          |   |   |   |   |   |   |   |   |
| 1b             | RESERVD | 00000000   | 0000     |          |   |   |   |   |   |   |   |   |

| Brd 0 LDT [P0] |         |            |          |          |   |   |   |   |   |   |   |   |
|----------------|---------|------------|----------|----------|---|---|---|---|---|---|---|---|
| Indx           | Type    | Descriptor | Bas/Sel  | Lim/Off  | G | D | O | L | P | L | S | T |
| 0              | DATAWA  | 4040f303   | 7f800420 | 40037f80 | 0 | 1 | 0 | 0 | 1 | 3 | 1 | 3 |
| 1              | DATAWA  | 4040f303   | 83a00420 | 400383a0 | 0 | 1 | 0 | 0 | 1 | 3 | 1 | 3 |
| 2              | DATAWA  | 4040f398   | cbe00420 | 4098cbe0 | 0 | 1 | 0 | 0 | 1 | 3 | 1 | 3 |
| 3              | DATAWA  | 4140f318   | cbe00420 | 4118cbe0 | 0 | 1 | 0 | 0 | 1 | 3 | 1 | 3 |
| 4              | RESERVD | 00000000   | 00000000 | 00000000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5              | RESERVD | 00000000   | 00000000 | 00000000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6              | RESERVD | 00000000   | 00000000 | 00000000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7              | RESERVD | 00000000   | 00000000 | 00000000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8              | RESERVD | 00000000   | 00000000 | 00000000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9              | RESERVD | 00000000   | 00000000 | 00000000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a              | RESERVD | 00000000   | 00000000 | 00000000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| b              | RESERVD | 00000000   | 00000000 | 00000000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# GDT in Windows NT

GDTbase = 80036000

Limit = 03FF

| Sel. | Type   | Base     | Limit    | DPL | P | Attr. |
|------|--------|----------|----------|-----|---|-------|
| 0008 | Code32 | 00000000 | FFFFFFFF | 0   | 1 | RE    |
| 0010 | Data32 | 00000000 | FFFFFFFF | 0   | 1 | RW    |
| 001B | Code32 | 00000000 | FFFFFFFF | 3   | 1 | RE    |
| 0023 | Data32 | 00000000 | FFFFFFFF | 3   | 1 | RW    |
| 0028 | TSS32  | 8000B000 | 000020AB | 0   | 1 | B     |
| 0030 | Data32 | FFDFF000 | 00001FFF | 0   | 1 | RW    |
| 003B | Data32 | 7FFDE000 | 00000FFF | 3   | 1 | RW    |
| 0043 | Data16 | 00000400 | 0000FFFF | 3   | 1 | RW    |
| 0048 | LDT    | E156C000 | 0000FFEF | 0   | 1 |       |
| 0050 | TSS32  | 80143FE0 | 00000068 | 0   | 1 |       |
| 0058 | TSS32  | 80144048 | 00000068 | 0   | 1 |       |

User Thread Environment Block

Kernel Processor Control Region

# GDT/LDT Manipulation Example (1)

```
0x0100: 0008002F 00000000 00000000 00000000
0x0110: 0000FFFF 00CF9A00 0000FFFF 00CF9200
0x0120: 0000FFFF 00CFFA00 0000FFFF 00CFF200
0x0130: 0138000F 00408200 F0001FFF FF4092DF
0x0140: E0000FFF 7F40F2FD
```

**Assume Protected 16-bit mode here**

```
0x1028: mov eax, 0x0
0x102D: lgdt fword ptr [eax] Load GDT
0x1030: jmp 0x08:0x00001037 Load new CS
0x1037: mov eax, 0x10 Protected 32 mode
0x103C: mov ds, ax Load data and stack
0x103F: mov ss, ax segments from GDT
0x1042: mov eax, 0x28
0x1047: lldt ax Load LDT
0x104A: mov eax, 0x7 Load data segments
0x104F: mov fs, ax from LDT
0x1052: mov eax, 0x0F
0x1057: mov gs, ax
```

# GDT/LDT Manipulation Example (2)

## Exercise 1

Assuming working in flat data segment write the following system call in C programming language

● **void print\_gdt()**

- ◆ This function presents the unpacked context of all present segment descriptors of the current GDT

To present a segment descriptor context you can use the following function:

```
void show_descriptor(__int32 index,
 __int32 base,
 __int32 size,
 __int32 type,
 __int32 dpl,
 bool system,
 bool db)
```

```

typedef struct GDTR_s {
 __int64 limit : 16,
 base : 32,
 res : 16;
} GDTR_t;

typedef struct Segment_Descriptor_s {
 __int64 limit_15_00 : 16,
 base_15_00 : 16,
 base_23_16 : 8,
 type : 4,
 system : 1,
 dpl : 2,
 present : 1,
 limit_19_16 : 4,
 avl : 1,
 res : 1,
 db : 1,
 granularity : 1,
 base_31_24 : 4;
} Segment_Descriptor_t;

```

```

void print_gdt()
{
 unsigned number_of_entries, i, base, size;
 Segment_Descriptor_t* gdt;
 GDTR_t gdtr;

 // Get GDTR
 __asm
 {
 sgdt gdtr
 }

 // Calculate the number of descriptors within GDT
 number_of_entries = (gdtr.limit + 1) >> 3;

 // Assign the base pointer
 gdt = (Segment_Descriptor_t*)gdtr.base;

```

```

// Go through all descriptors
for (i = 0; i < number_of_entries; i++) {
 // Proceed if present
 if (gdt[i].present) {
 // Calculate segment base
 base = (gdt[i].base_31_24 << 24) |
 (gdt[i].base_23_16 << 16) |
 gdt[i].base_15_00;

 // Calculate segment size
 size = (gdt[i].limit_19_16 << 16) |
 gdt[i].limit_15_00;

 // Incorporate granularity
 if (gdt[i].granularity) {
 size = (size * 0x1000) + 0xFFF;
 }

 // Present the descriptor
 show_descriptor(i, base, size, gdt[i].type,
 gdt[i].dpl, gdt[i].system,
 gdt[i].db);
 }
}
}

```

# GDT/LDT Manipulation Example

## Exercise 2 - Homework

Assuming working in flat data segment write the following system call in C programming language

● **void print\_ldt()**

- ◆ This function presents the unpacked context of all present segment descriptors of the current LDT. Assume LDT exists if LDTR != 0.

To present a segment descriptor context you can use the following function:

```
void show_descriptor(__int32 index,
 __int32 base,
 __int32 size,
 __int32 type,
 __int32 dpl,
 bool system,
 bool db)
```

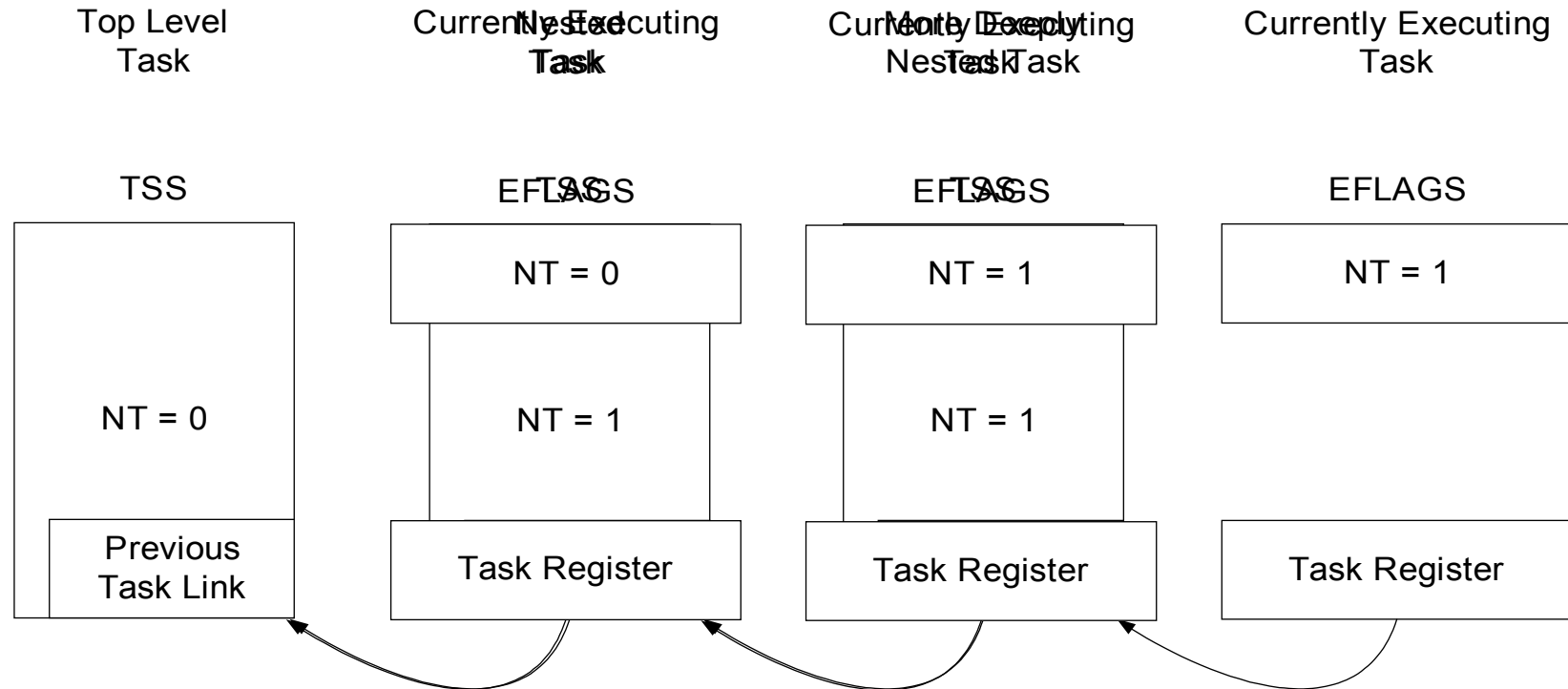
# Task State Segment (TSS)

- HW mechanism to allow multi-tasking
- Contains task state:
  - ◆ Dynamic: Integer registers, segment registers, flags...
  - ◆ Static: CR3, per level stack pointer, etc...
- Task switch occurs on
  - ◆ JMP/Call to a TSS descriptor or task gate
  - ◆ Interrupt indexed to a task gate
  - ◆ Certain IRET
- On switch:
  - ◆ Current TSS content is updated
 

For example, general registers are saved by the CPU!
  - ◆ New task's values are retrieved from TSS
- Contains initial stack values for level 0-2
- System should have at least one TSS.
- Task Register (TR) is loaded by LTR instruction, or during a task switch.

| 31                   | 15                   | 0 |     |
|----------------------|----------------------|---|-----|
| I/O Map Base Address |                      | T | 100 |
|                      | LDT Segment Selector |   | 96  |
|                      | GS                   |   | 92  |
|                      | DS                   |   | 88  |
|                      | DS                   |   | 84  |
|                      | SS                   |   | 80  |
|                      | CS                   |   | 76  |
|                      | 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  |
|                      | SS2                  |   | 24  |
|                      | ESP2                 |   | 20  |
|                      | SS1                  |   | 16  |
|                      | ESP1                 |   | 12  |
|                      | SS0                  |   | 8   |
|                      | ESP0                 |   | 4   |
|                      | Previous Task Link   |   | 0   |

# Task Linking



- Upon CALL instruction, an interrupt, or an exception causing a task switch
  - ◆ CPU copies the segment selector of the current TSS into the previous task link field of the TSS for the new task and sets EFLAGS.NT = 1
- Upon IRET instruction suspending the new task:
  - ◆ CPU uses the value in the previous task link field and the NT flag to return to the previous task.

# Paging

# Motivation

## ●The problem:

- ◆ Register can cover 4GB, but actual memory is smaller.
- ◆ Application should be independent of actual memory location
  - avoid memory size and location concerns
  - allow using fixed, pre-defined addresses
  - allow easier code/data sharing

## ●Solution: Paging

- ◆ memory is divided to equal sized chunks - pages (4KB)
- ◆ A layer on top of segmentation. Performs linear to physical mapping.
- ◆ Address “processing”

**Logical (Seg:Offset) ==> Linear (segment base+offset) ==> Physical**

- ◆ Page tables map linear address to physical. Keeping track of:
  - Valid pages, not existing pages, page attributes

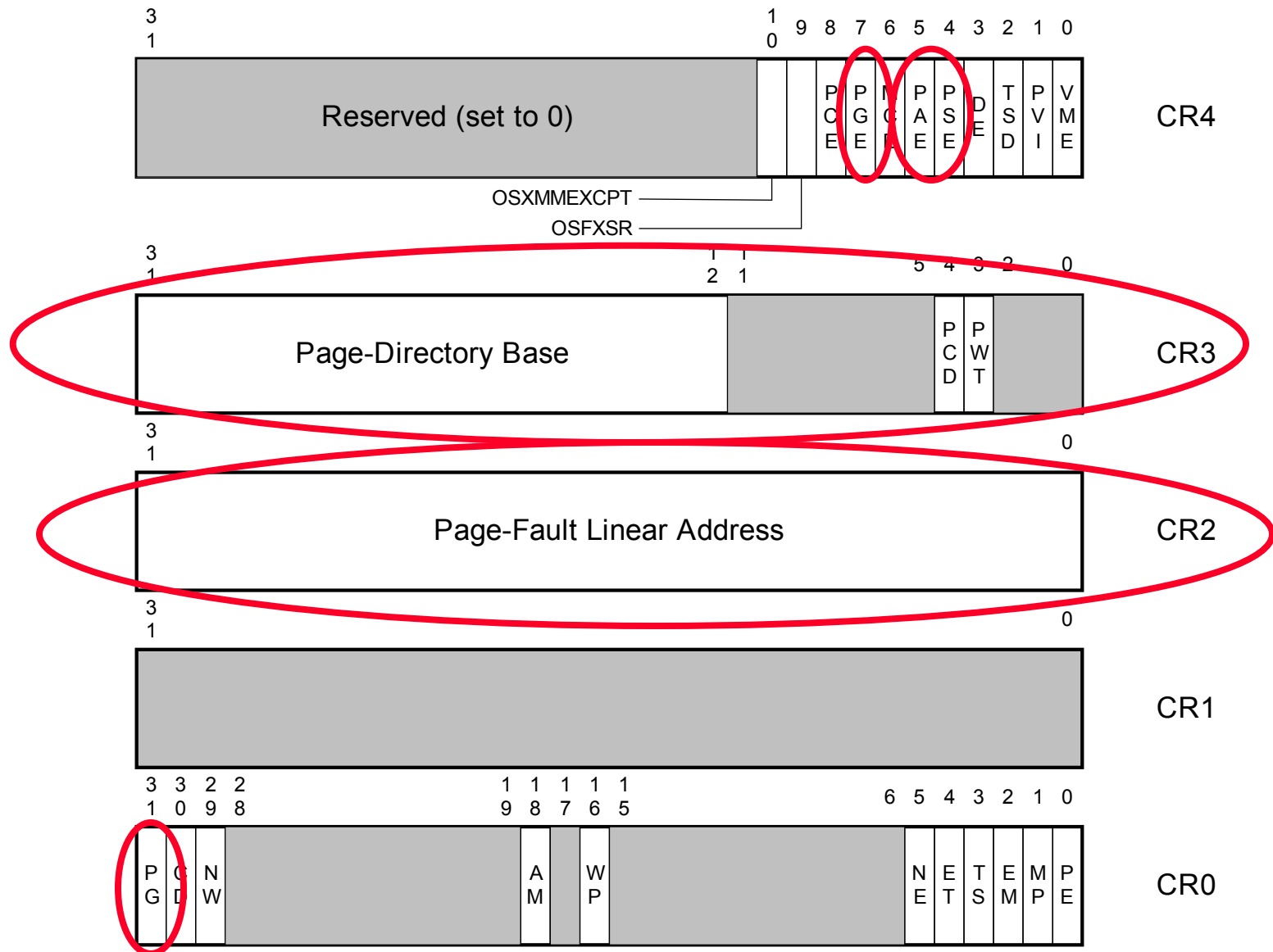
## ●Analogy:

- ◆ **Big library, good catalog, reading room, storage room**

# Paging - Virtual Memory

- A page can be
  - ◆ New, not yet loaded
  - ◆ Loaded
  - ◆ Paged out to disk
- A loaded page can be
  - ◆ Clean
  - ◆ Dirty
- When a page is not loaded (P bit clear) => a page fault occurs
  - ◆ It may require removing a loaded page to insert the new one
    - OS prioritizes removal by LRU and dirty/clean/avail bits
    - Dirty page should be written to disk. Clean ones need not
  - ◆ New page is either loaded from disk or “initialized”
  - ◆ CPU sets page “access” flag when accessed, “dirty” when written
- Copy-on-write strategy for forked Unix processes

# Paging Related Bits in Control Registers



# Paging options

## ●Paging is controlled by three flags in CPU's control registers:

### ◆ CR0.PG - paging

enables the page translation mechanism when set.

can be set if CR0.PE flag is set

### ◆ CR4.PSE - page size extension

enables large page sizes when set

4MB

### ◆ CR4.PAE - physical address extension

enables 36-bit physical addresses.

can only be used when paging is enabled.

Enables 2MB pages

# Page Tables and Directories

## ●Page Directory

- ◆ an array of 1024 32-bit Page-Directory Entries (PDE) contained in 4KB page.

## ●Page Table

- ◆ an array of 1024 32-bit Page-Table Entries (PTE) contained in 4KB page.

## ●Page

- ◆ a 4KB, 2MB or 4MB flat address space.

## ●Page-Directory-Pointer Table

- ◆ an array of four 64-bit entries, each of which points to a page directory.
- ◆ only used when the physical address extension is enabled (CR4.PAE=1).

# Page Translation Mechanism (4KB)

## ●2-level hierarchical mapping

- ◆ using page directory and page tables
- ◆ all pages and page tables are 4KB

## ●Linear address divided to:

- ◆ Dir: 10 bits - index into page directory
- ◆ Table: 10 bits - index into page table
- ◆ Offset: 12 bits - offset in the page

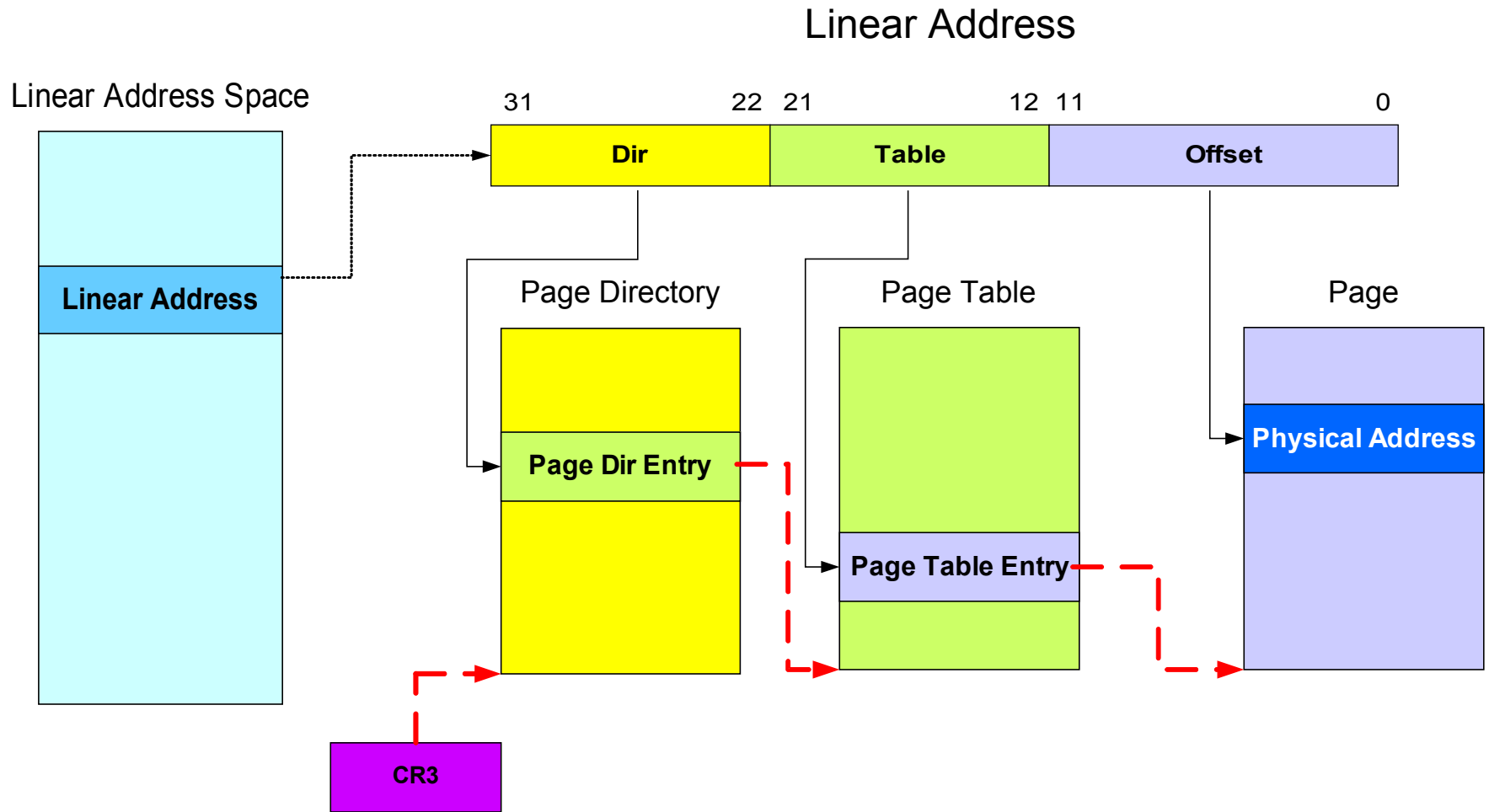
## ●CR3 points to current Page Directory

- ◆ may be changed per process

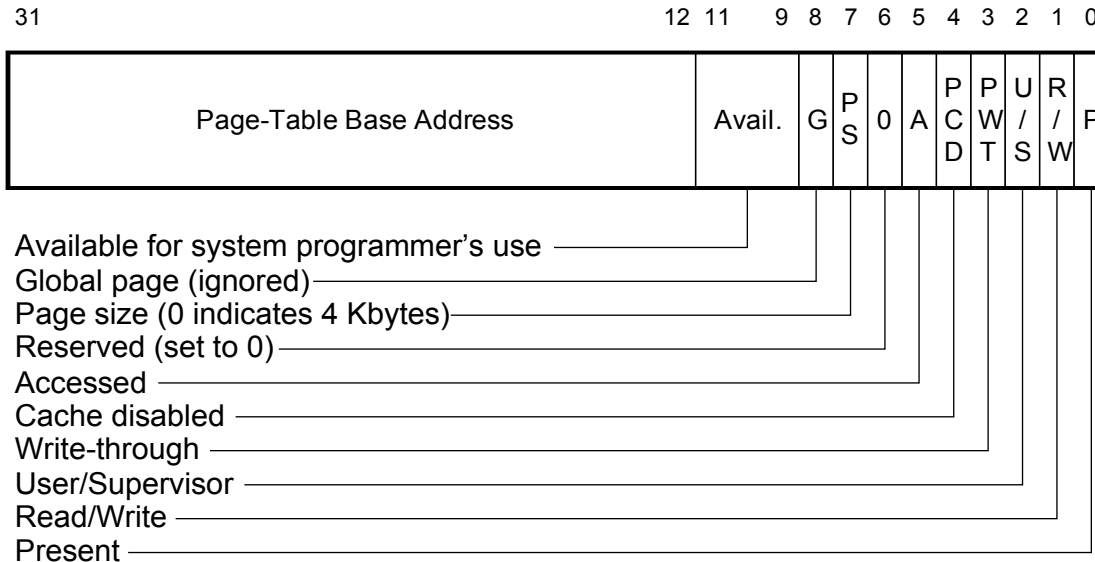
## ●Sharing

- ◆ different linear to same physical
- ◆ pages from different processes to same physical

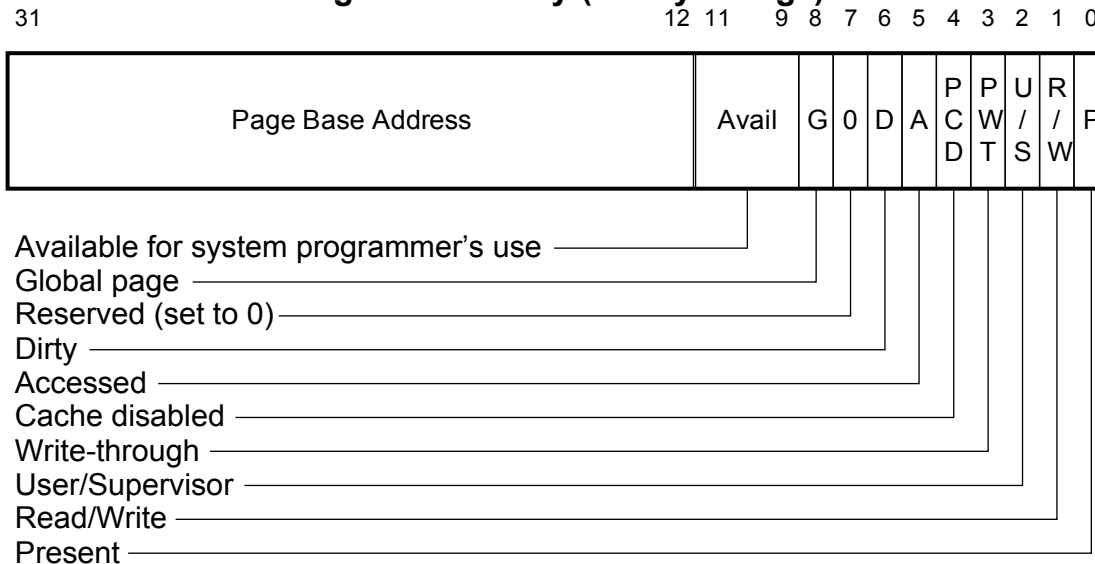
# Page Translation Mechanism (4KB)



## Page-Directory Entry (4-KByte Page Table)



## Page-Table Entry (4-KByte Page)



# Enabling Paging Example

Physical Memory at 0x1000:      0x00002003   0x00000000  
Physical Memory at 0x2000:      0x00000003   0x00001003

Virtual Address

=====

|            |     |                |                                                       |
|------------|-----|----------------|-------------------------------------------------------|
| 0x00000C34 | mov | eax, 1000h     |                                                       |
| 0x00000C3A | mov | cr3, eax       | Load CR3                                              |
| 0x00000C3E | mov | eax, cr0       |                                                       |
| 0x00000C42 | or  | eax, 80000000h |                                                       |
| 0x00000C48 | mov | cr0, eax       | Enable paging                                         |
| 0x00000C4C | jmp | 08h:00h        | Make sure the<br>next instruction<br>is mapped 1-to-1 |

# Page Walk Example

## Exercise

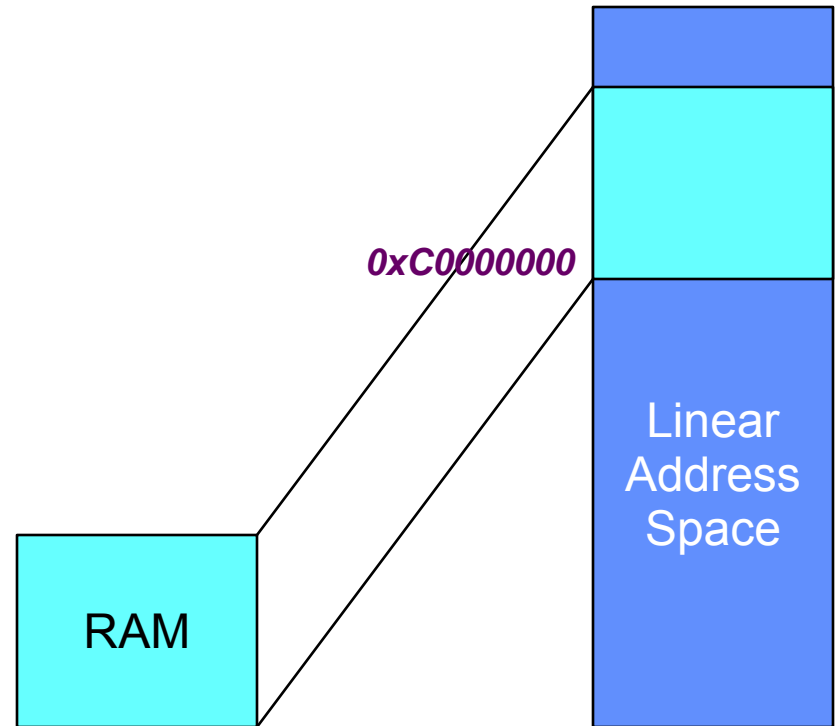
Assume that the Operating System operates in flat memory model and maps the whole RAM 1-to-1 starting from linear address **0xC0000000**.

Assume also that OS uses both 4KB and 4MB pages (CR4.PSE=1).

Write the following kernel function in C:

● **unsigned translate\_linear\_to\_physical(unsigned la)**

- ◆ Input:           **unsigned la**       – Linear Address
- ◆ Output:          **unsigned**           – Corresponding Physical Address or 0xFFFFFFFF if not mapped



```

unsigned translate_linear_to_physical(unsigned la)
{
 unsigned cr3_val;
 unsigned pdir_index = la >> 22;
 unsigned ptable_index = (la & 0x003FFFFFF) >> 12;
 unsigned *pdir, *ptable, pde, pte;

 // Get CR3
 __asm {
 mov eax, cr3
 mov cr3_val, eax
 }

 // Calculate linear address of the Page Directory
 pdir = (unsigned*)((cr3_val & 0xFFFFF000) + 0xC0000000);

 // Calculate the corresponding Page Directory Entry
 pde = pdir[pdir_index];

 // Check if Page Directory Entry is present
 if (pde & 0x1 == 0x0) return 0xFFFFFFFF;
}

```

```

// Check if the Page is 4MB size
if (pde & 0x80 != 0x0)
{
 return (pde & 0xFFC00000) + (la & 0x003FFFFFF);
}

// Calculate linear address of the Page Table
ptable = (unsigned*)((pde & 0xFFFFF000) + 0xC0000000);

// Calculate the corresponding Page Table Entry
pte = ptable[ptable_index];

// Check if Page Table Entry is present
if (pte & 0x1 == 0x0)
{
 return 0xFFFFFFFF;
}

return (pte & 0xFFFFF000) + (la & 0x00000FFF);
}

```

# Page Fault

- #PF - Page Fault Exception occurs if

- ◆ Present bit (P) of page-directory or page-table entries is clear
- ◆ A privilege level violation (User / Supervisor) occurs
- ◆ An access violation (Read-Only / Read-Write) occurs
- ◆ Reserved bits are set

- CPU loads CR2 with the faulting address

- If the processor generates a page-fault exception due to not present page, the OS must carry out the following operations in the following order:

- ◆ Copy the page from disk storage into physical memory, if needed.
- ◆ Load the page address into the page-table or page-directory entry and set its present flag. Other bits, such as the dirty and accessed flags, may also be set at this time.
- ◆ Invalidate the current page-table entry in the TLB.
- ◆ Return from the page-fault handler to restart the interrupted program or task.

# Page-Level Protection

- Can be used alone (flat memory model)

- ◆ allows supervisor (ring0,1,2) code and data to be protected from user (ring3) code and data
- ◆ write protection

- Can be applied to segments

- ◆ CPU evaluates segment protection first
- ◆ Page-level protection cannot be used to override segment-level protection.

For example: code segment is by definition not writable. Setting its pages to Read-Write does not make the pages writable. Attempts to write into the pages will be blocked by segment-level protection checks.

# Mapping 4M pages, 36-bit Physical Addresses

## ●PSE (page size extension) flag bit 4 of CR4

- ◆ The PSE enables 4MByte. Those pages are mapped directly from the page directory, without using page tables.
- ◆ If PDE.PS=1 the page is considered to be 4MB. Linear to physical address translation then is:

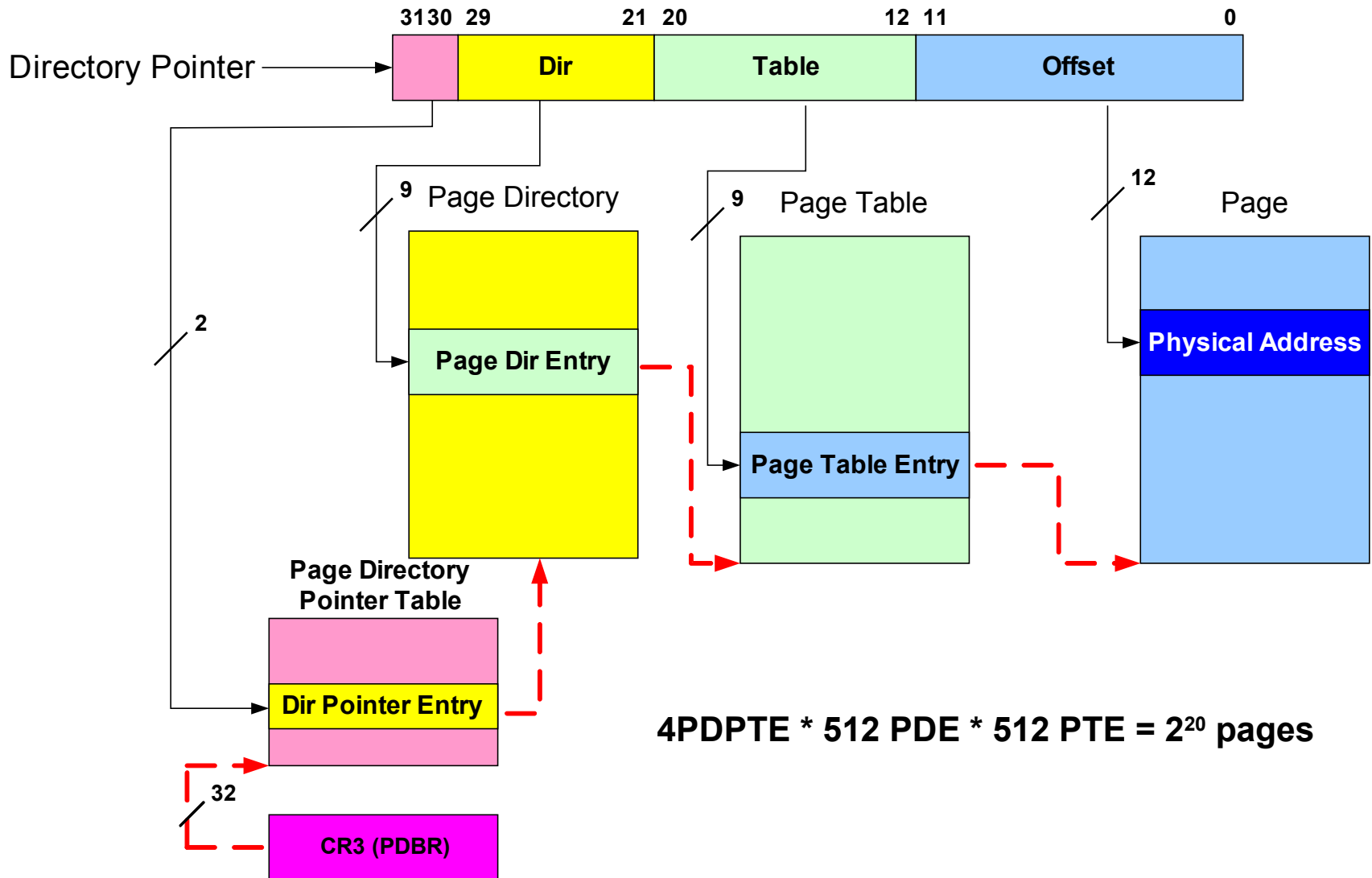
$$PA = (PDE \& 0xFFC00000) + (LA \& 0x003FFFFFFF)$$

## ●PAE (physical address extension) flag bit 5 of CR4

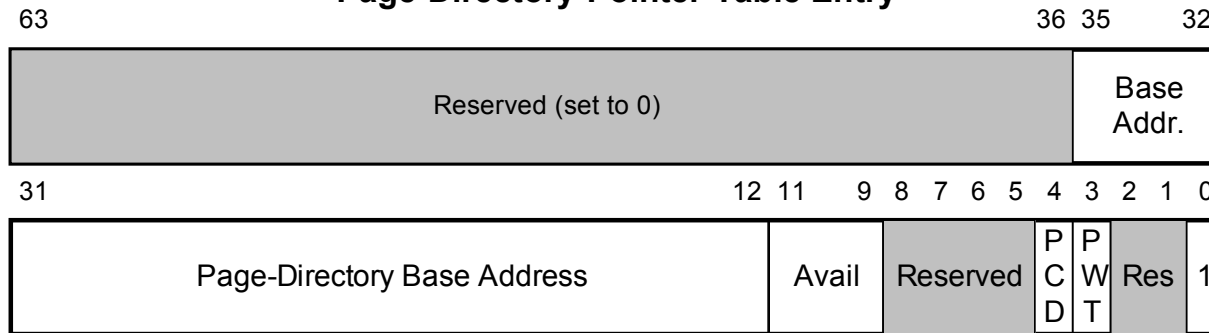
- ◆ The PAE flag enables 36 bit physical addresses
- ◆ If CR4.PAE=1 the architecture supports page sizes of 4KB and 2MB regardless of the value of CR4.PSE.

# Page Translation Mechanism (4KB) with Physical Address Extension

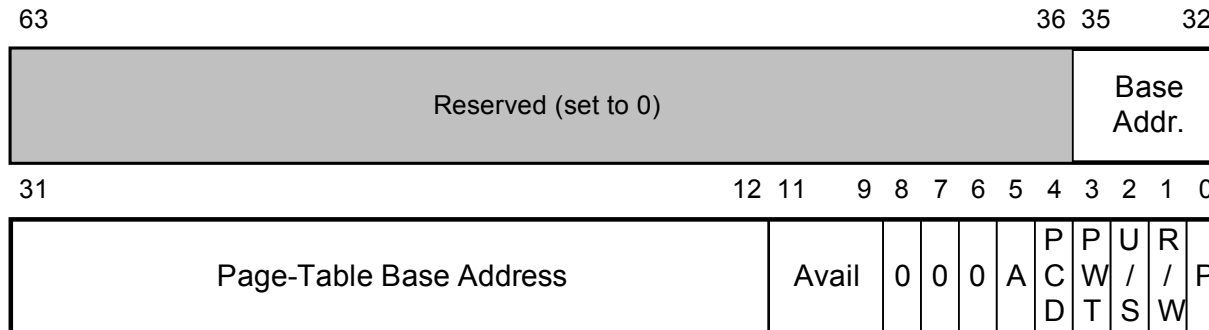
Linear Address



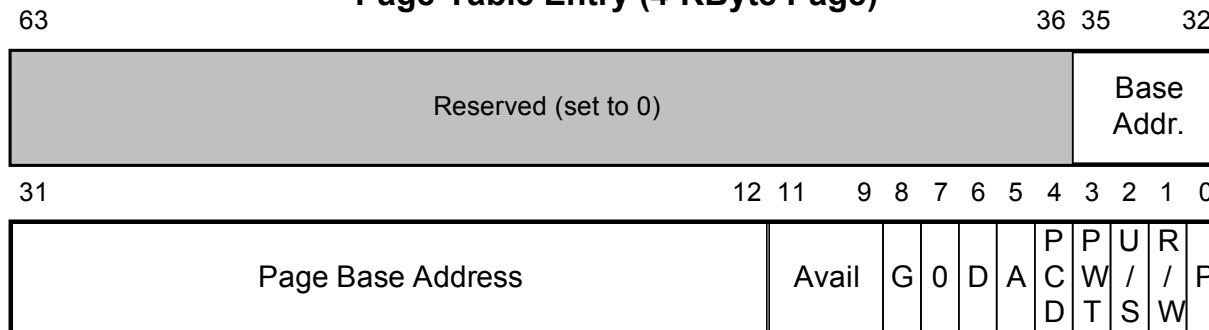
### Page-Directory-Pointer-Table Entry



### Page-Directory Entry (4-KByte Page Table)



### Page-Table Entry (4-KByte Page)



# Enabling Physical Address Extension

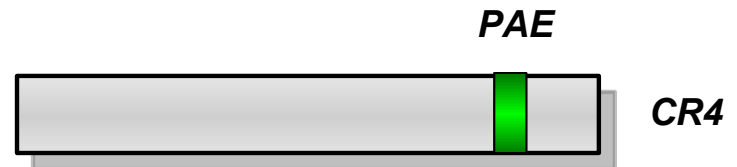
- Start from Paging Enabled and Physical Address Extension disabled  
(CR0.PG=1, CR4.PAE=0)

- Disable Paging  
(CR0.PG=0, CR4.PAE=0)

- Enable physical address extension  
(CR0.PG=0, CR4.PAE=1)

- Load CR3 with the physical address of the Page Directory Pointer Table

- Enable Paging  
(CR0.PG=1, CR4.PAE=1)



# Paging/Segmentation Example

●GDTR: 0x00002000 (linear)

●CR3: 0x00003000 (physical)

●Logical address:

◆ DS: 0x23

◆ Offset: 0x3456

●DS: 0010 0011 ==> index = 4, TI = GDT, PL = 3

$GDT(4) = GDTR + 4 * 8 = 0x2000 + 0x20 = 0x2020$

●Assume protection validation has passed...and

$GDT(4).base = 0x00c03000$  (linear)

●Linear address =  $0x00c03000 + 0x3456 = 0x00c06456$

0000 0000 1100 0000 0110 0100 0101 0110

◆ PDE\_index = 0x003

◆ PTE\_index = 0x006

◆ Offset = 0x456

# Paging/Segmentation Example (cont)

- Physical address of PDE is

$$\text{PDE} = \text{CR3} + 4 * \text{PDE\_index} = 0\text{x}3000 + 4 * 3 = 0\text{x}300\text{c}$$

$$\text{M}(0\text{x}300\text{c}) = 0\text{x}4000$$

- Physical address of PTE is

$$\text{PTE} = * \text{PDE} + 4 * \text{PTE\_index} = 0\text{x}4000 + 4 * 6 = 0\text{x}4018$$

$$\text{M}(0\text{x}4018) = 0\text{x}5000$$

- Physical address corresponding to linear address is

$$\text{Physical Addr.} = * \text{PTE} + \text{offset} = 0\text{x}5000 + 0\text{x}456 = 0\text{x}5456$$

$$\text{M}(0\text{x}5456) = 0\text{x}12345678$$

- Finished? Not exactly...

# Paging/Segmentation Example (cont)

- Where do we find the GDT?

- ◆ GDTR points to 0x2000 linear, need to look at the physical address to see GDT(4)!

- Linear address = 0x00002020 =

0000 0000 0000 0000 0010 0000 0010 0000

- ◆ PDE = 0x000

- ◆ PTE = 0x002

- ◆ Offset = 0x020

$$\text{PDE} = \text{CR3} + 4 * \text{PDE\_index} = 0\text{x}3000 + 4 * 0 = 0\text{x}3000$$

$$\text{M}(0\text{x}3000) = 0\text{x}6000$$

$$\text{PTE} = * \text{PDE} + 4 * \text{PTE\_index} = 0\text{x}6000 + 4 * 2 = 0\text{x}6008$$

$$\text{M}(0\text{x}6008) = 0\text{x}5000$$

$$\text{Physical Addr.} = * \text{PTE} + \text{offset} = 0\text{x}5000 + 0\text{x}20 = 0\text{x}5020$$

$$\text{M}(0\text{x}5020) = 0\text{x}\underline{0010}\text{f0}\underline{\text{c03000}}\text{fedc}$$

# Page Tables in Linux

Brd 0 Page Directory [P0]

| Indx | Entry | Frame addr | A P P P |   |   |   | P P P P |   |   |   |   |   |   |   |   |
|------|-------|------------|---------|---|---|---|---------|---|---|---|---|---|---|---|---|
|      |       |            | L       | V | A | T | G       | S | D | A | D | T | U | W | P |
| +    | 2fd   | 00000000   | 0       | * | 0 | 0 | 0       | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| +    | 2fe   | 00000000   | 0       | * | 0 | 0 | 0       | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| +    | 2ff   | 372ac067   | 0       | * | 0 | 0 | 1       | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
|      | 300   | 000001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 301   | 004001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 302   | 008001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 303   | 00c001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 304   | 010001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 305   | 014001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 306   | 018001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 307   | 01c001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 308   | 020001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 309   | 024001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 30a   | 028001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 30b   | 02c001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 30c   | 030001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 30d   | 034001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 30e   | 038001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 30f   | 03c001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 310   | 040001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 311   | 044001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 312   | 048001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 313   | 04c001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 314   | 050001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 315   | 054001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 316   | 058001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 317   | 05c001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 318   | 060001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 319   | 064001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 31a   | 068001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 31b   | 06c001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 31c   | 070001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 31d   | 074001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 31e   | 078001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 31f   | 07c001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 320   | 080001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 321   | 084001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 322   | 088001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 323   | 08c001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 324   | 090001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 325   | 094001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
|      | 326   | 098001e3   | 0       | 0 | 1 | 1 | 1       | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |

Brd 0 Page Directory [2ff] -> Page Table [P0]

| Indx | Entry    | Frame addr | A |   |   |   | P |   |   |   | P |   |   |  |
|------|----------|------------|---|---|---|---|---|---|---|---|---|---|---|--|
|      |          |            | L | G | D | V | A | D | T | U | W | P |   |  |
| 3dd  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3de  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3df  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3e0  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3e1  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3e2  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3e3  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3e4  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3e5  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3e6  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3e7  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3e8  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3e9  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3ea  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3eb  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3ec  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3ed  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3ee  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3ef  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3f0  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3f1  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3f2  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3f3  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3f4  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3f5  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3f6  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3f7  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3f8  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3f9  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3fa  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3fb  | 00000000 | 00000000   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3fc  | 00028500 | 00028000   | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3fd  | 00019b00 | 00019000   | 5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| 3fe  | 3e52c067 | 3e52c000   | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |  |
| 3ff  | 3e519025 | 3e519000   | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |  |

# Windows NT Paging Management

- Reserves 4MB of the system address space for Page Tables:

0xC0000000-0xC03FFFFFFF

- Page Directory functions:

- ◆ Page Directory for 4GB space
- ◆ Page Table for pages representing the Page Tables

- Page Directory virtual address:

0xC0300000

1100000000 1100000000 00000000000000

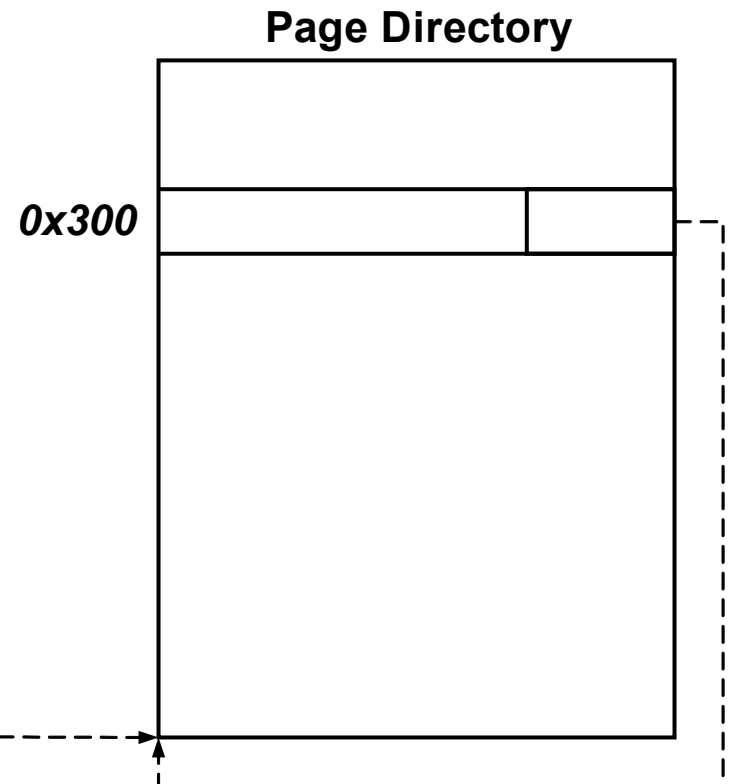
0x300

0x300

DIR

TABLE

CR3



# Windows NT Paging Management

## Exercise

Assuming working in Windows NT, write the following functions in “C”:

● **bool is\_present(unsigned int la)**

- ◆ Input:            **unsigned int la**            – Linear Address
- ◆ Output:          **TRUE** if present, **FALSE** otherwise

● **void map\_page(unsigned int la,  
                  unsigned int pa,  
                  bool is\_4kb\_page,  
                  unsigned attr)**

- ◆ Input:            **unsigned int la**            – Linear Address to be mapped
- unsigned int pa**            – Corresponding Physical Address
- bool is\_4kb\_page**            – **TRUE** if 4KB, **FALSE** if 4MB
- unsigned attr**            – page attributes (R/W, U/S, etc)

● **unsigned int virtual\_to\_physical(unsigned int la)**

- ◆ Input:            **unsigned int la**            – Linear Address
- ◆ Output:          **unsigned int**            – Corresponding Physical Address

```

bool is_present(unsigned la)
{
 unsigned pdir_index = la >> 22;
 unsigned* ptables = (unsigned*)0xC0000000;
 unsigned* pdir = (unsigned*)0xC0300000;

 // Check if Page Directory Entry is present
 if (pdir[pdir_index] & 0x1 == 0x0) {
 return false;
 }

 // Check if 4MB page
 if (pdir[pdir_index] & 0x80) {
 return true;
 }

 // Check if Page Table Entry is present
 if (ptables[la >> 12] & 0x1) {
 return true;
 }

 return false;
}

```

```

unsigned virtual_to_physical(unsigned la)
{
 unsigned pdir_index = la >> 22;
 unsigned* ptables = (unsigned*)0xC0000000;
 unsigned* pdir = (unsigned*)0xC0300000;

 // Check if Page Directory Entry is present
 if (pdir[pdir_index] & 0x1 == 0x0)
 return 0xFFFFFFFF;

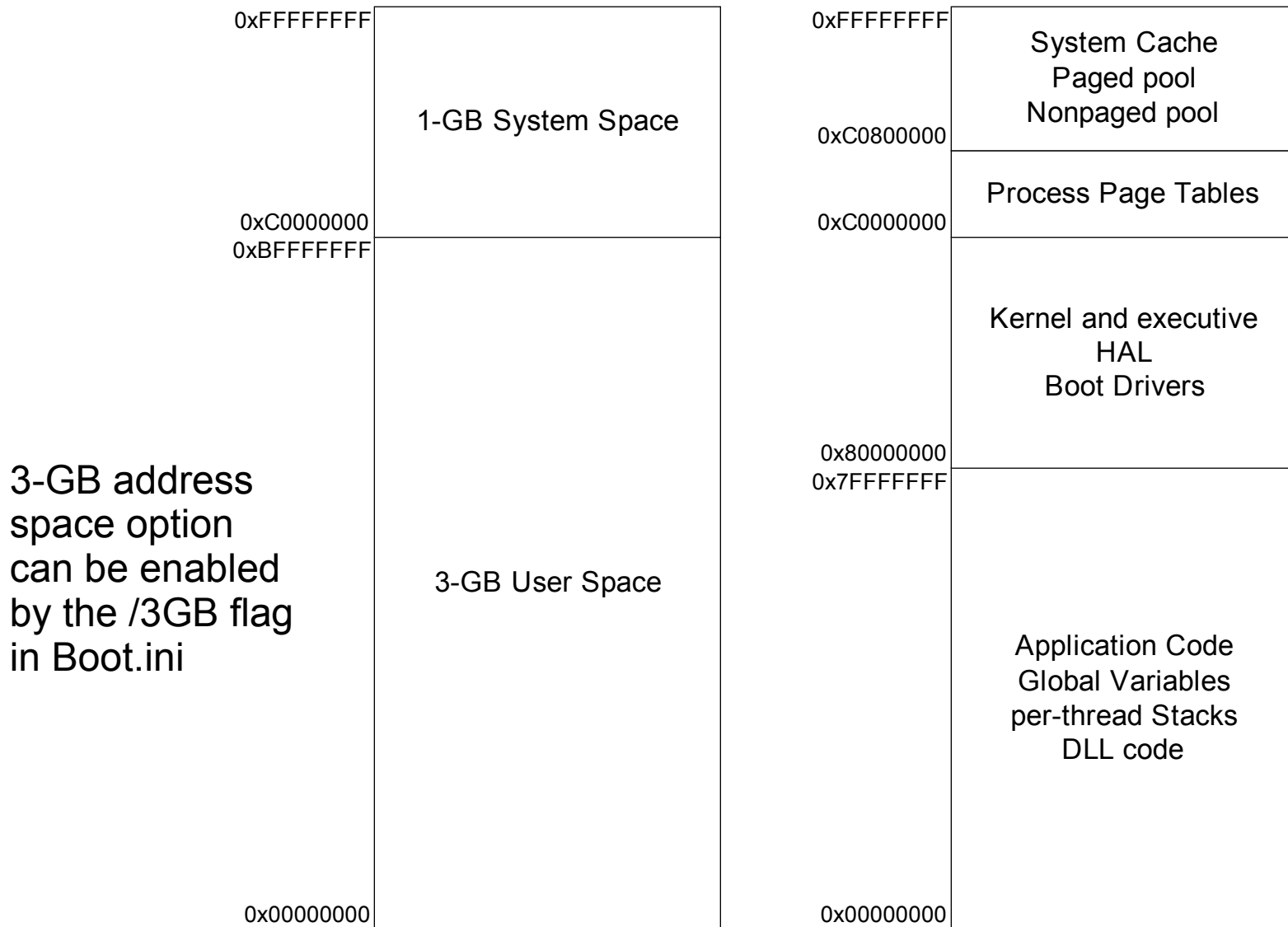
 // Check if 4MB page
 if (pdir[pdir_index] & 0x80)
 return (pdir[pdir_index] & 0xFFC00000) +
 (la & 0x003FFFFFF);

 // Check if Page Table Entry is present
 if (ptables[la >> 12] & 0x1)
 return (ptables[la >> 12] & 0xFFFFF000) +
 (la & 0x00000FFF);

 return 0xFFFFFFFF;
}

```

# Windows 2000 Memory Layout



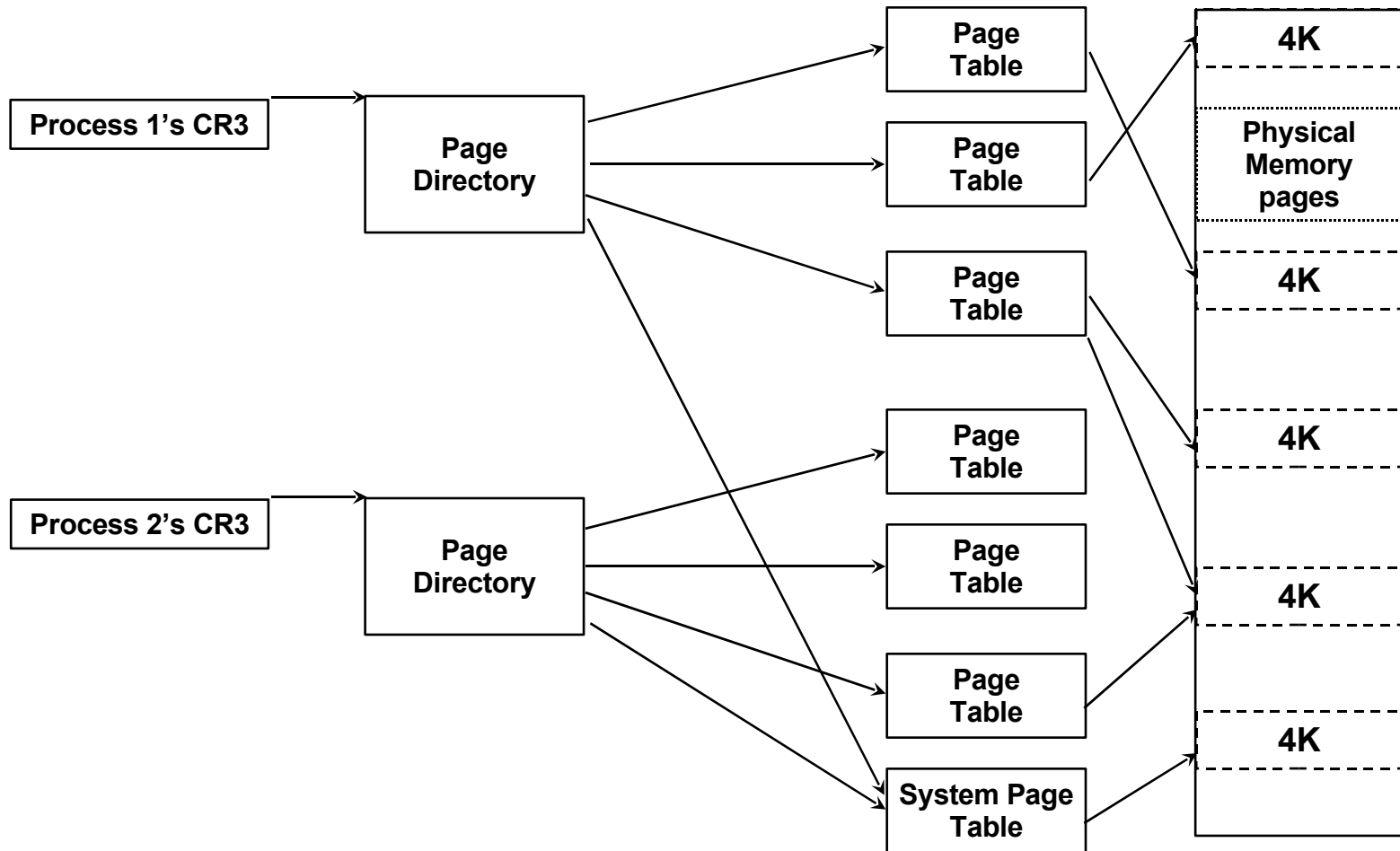
# System Memory Map for Windows NT

|              |                   |                                                                                   |
|--------------|-------------------|-----------------------------------------------------------------------------------|
| 0xFFFFFFFF   | Processor Control | <i>Processor Control Region</i>                                                   |
| 0xFFDFF000   |                   |                                                                                   |
| 0xFFDFEFFF   | Non Paged System  | <i>System Drivers, Kernel Thread Stacks, NonPageable Pool, Pageframe Database</i> |
| 0xFB000000   |                   |                                                                                   |
| 0xE57FFFFFFF | Paged Pool        | <i>Pageable Pool</i>                                                              |
| 0xE1000000   |                   |                                                                                   |
| 0xD8FFFFFFF  | System Cache      | <i>Windows NT Cache Manager Mapping Area</i>                                      |
| 0xC1000000   |                   |                                                                                   |
| 0xC0FFFFFFF  | System Tables     | <i>Page Directory and Page Tables</i>                                             |
| 0xC0000000   |                   |                                                                                   |
| 0xBFFFFFFF   | System View Space | <i>Win32 GDI Object Table, Win32 USER Object Table</i>                            |
| 0xA0000000   |                   |                                                                                   |
| 0x9FFFFFFF   | System Code       | <i>Boot Drivers, NT OS Kernel, HAL, TSS, GDT, IDT</i>                             |
| 0x80000000   |                   |                                                                                   |

# Memory Model in Windows NT

- Minimize usage of segments
  - ◆ Use paging for address and access check
- Address error:  
Memory access beyond program limits is checked by the OS following page fault exception
- Access error:  
Memory Access is checked for read/write & privilege level in the page table and the address segment attribute
- There are different privilege levels for the OS and the user context (need to remember this during interrupt time)
- The FS segment register is used as a Task pointer to the OS database
- WinNT Still supports 16-bit and real mode segmentation applications with Virtual X86 (need to remember during Interrupt service)

# Windows NT Page Tables



# Translation Lookaside Buffers (TLBs)

- The processor stores the most recently used page-directory and page-table entries in on-chip caches called TLBs.
- The P6 family and Pentium processors have separate TLBs for the data and instruction caches.
- Also, P6 family processors maintain separate TLBs for 4-Kbyte and 4-Mbyte page sizes.
- Translates the linear address to physical address for all memory load and store.
- Lookups in a cache array for the physical address of the page being accessed.
- Caches page attributes with the physical address, and uses this information to check for page protection faults and other paging related exceptions.
- Also caches the effective memory type with each page address.

# Invalidating the TLBs

- **INVLPG** instruction - invalidates the TLB for a specific page (is not affected by the state of the G flag in PDE or PTE).
- The following operations invalidate all TLB entries except global entries:
  - ◆ Writing into control register CR3.
  - ◆ A task switch that changes control register CR3.
- The following operations invalidate all TLB entries, irrespective of the setting of the G flag:
  - ◆ Asserting or de-asserting the FLUSH# pin.
  - ◆ Writing to an MTRR (with a WRMSR instruction).
  - ◆ Writing to control register CR0 to modify the PG or PE flag.
  - ◆ Writing to control register CR4 to modify the PSE, PGE, or PAE flag.

# Paging Pros and Cons

- Independence on physical memory size
  - No need to preserve continuous memory ranges
  - Physical memory is allocated only if accessed
  - Can map same linear address to different physical address
  - Can share data/code among processes
  - Pages have predefined size - easy memory management
  - Simple protection mechanism
- 
- Overhead of tables (insignificant)
  - Internal fragmentation (insignificant)
  - Translation time - reduced by TLB

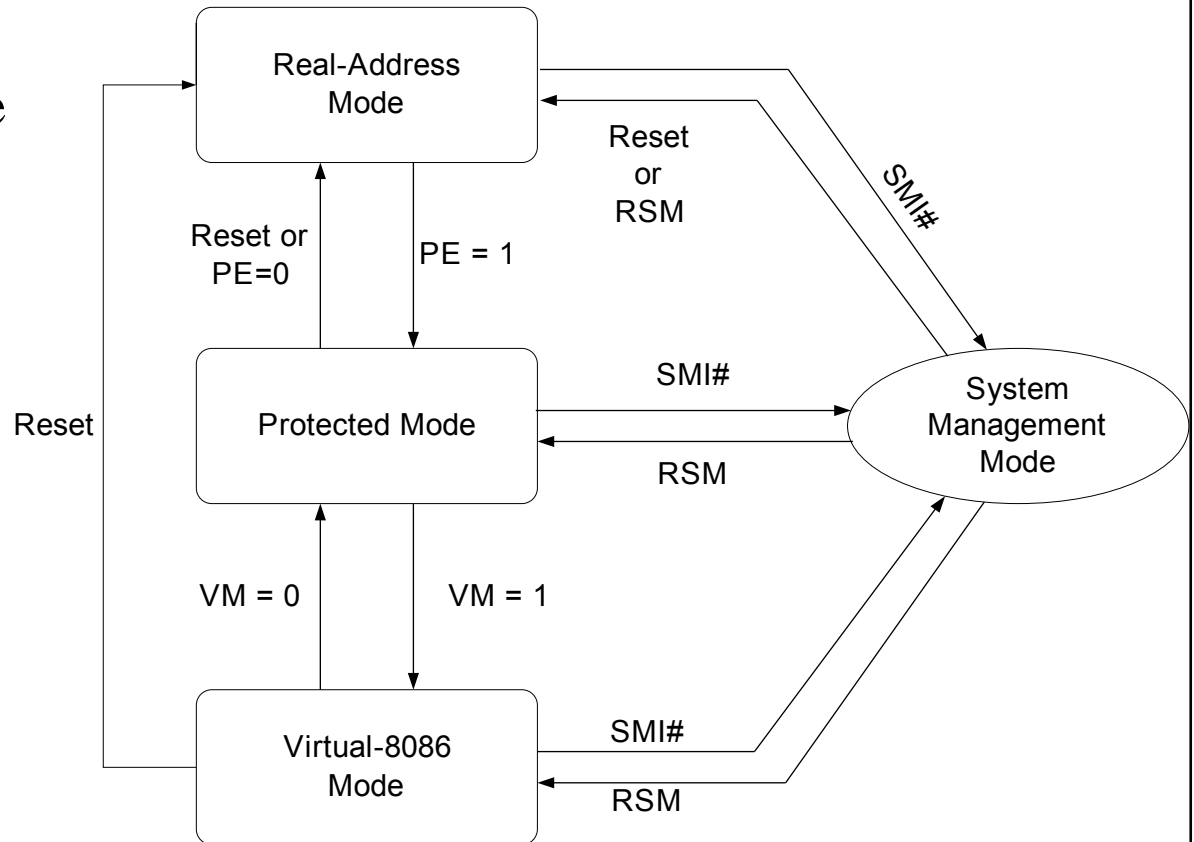
# Protection

# Protected Mode

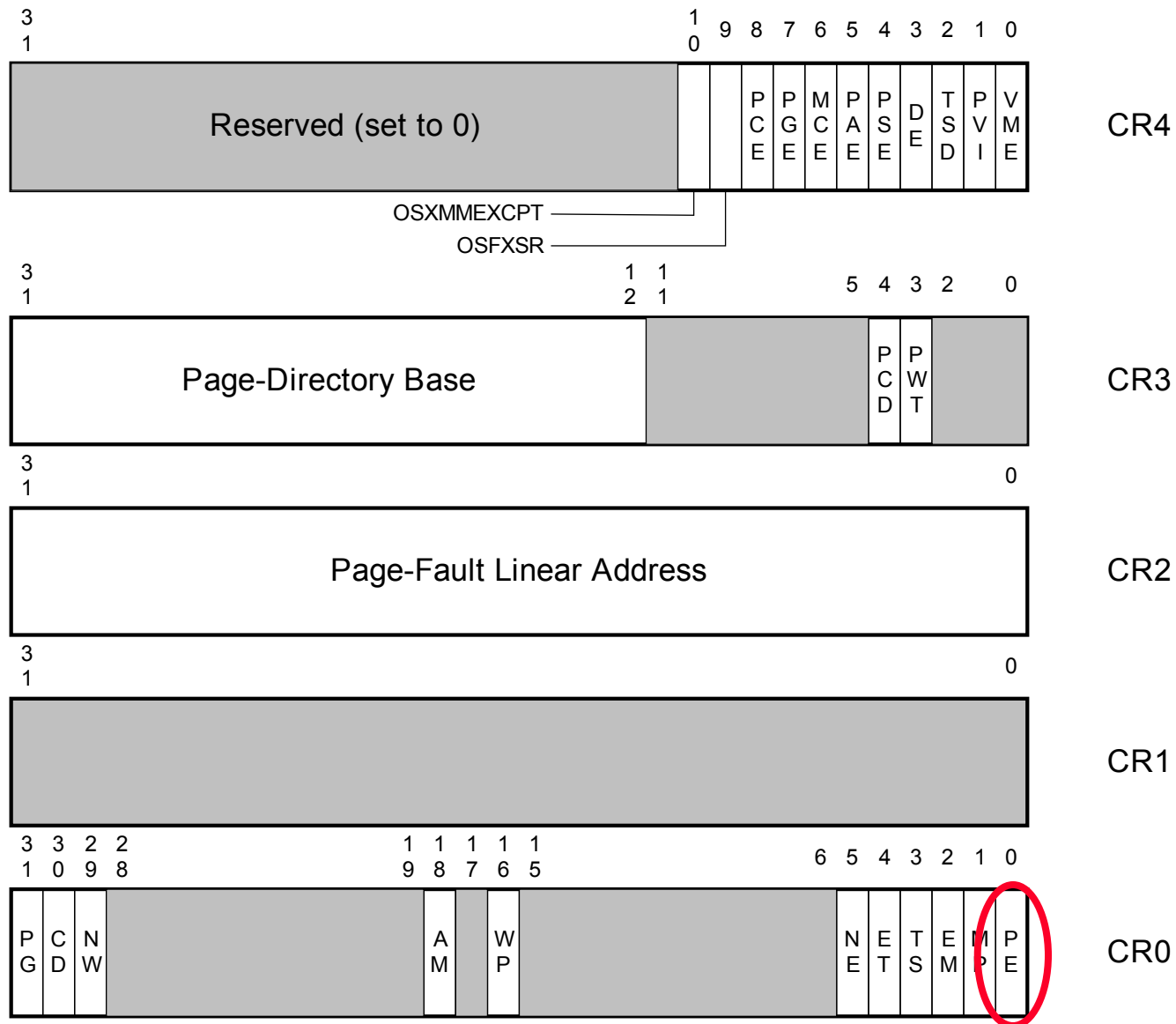
- Designed to provide a mechanism protecting Operating System code and data from being corrupted by user applications.

- Elements:

- ◆ Segmentation
- ◆ Paging
- ◆ Privilege levels
- ◆ Privilege instructions



# Protected Mode Bit in Control Registers



# Switching to Protected Mode

- Real Mode
- Set CR0.PE=1
  - ◆ Protected 16-bit mode
- Load CS with a 32-bit code segment
  - ◆ Protected 32-bit mode
- Set CR0.PG=1
  - ◆ Protected Mode with Paging
  - ◆ Make sure that the current page is mapped 1-to-1 in page tables

# Switching to Protected Mode

|         |            |                 |            |                                                  |
|---------|------------|-----------------|------------|--------------------------------------------------|
| 0x9008: | 0x90100018 | 0x00000000      | 0x00000000 | 0x00000000                                       |
| 0x9018: | 0x0000ffff | 0x00cf9b00      | 0x0000ffff | 0x00cf9300                                       |
| 0x9028: | cli        |                 | <==        | Disable interrupts                               |
| 0x9029: | movw       | \$0x0, %ax      |            |                                                  |
| 0x902c: | movw       | %ax, %ds        | <==        | Load DS with 16-bit segment based at 0x0000      |
| 0x902e: | leaw       | 0x9008, %si     |            |                                                  |
| 0x9032: | lgdt       | (%si)           | <==        | Load Global Descriptor Table                     |
| 0x9036: | movl       | %cr0, %eax      |            |                                                  |
| 0x9039: | orb        | \$0x1, %al      |            |                                                  |
| 0x903b: | movl       | %eax, %cr0      | <==        | Enable Protection                                |
| 0x903e: | ljmp       | \$0x8, \$0x9043 | <==        | Long JMP loads CS with 32-bit flat code segment  |
| 0x9043: | movw       | \$0x10, %ax     | <==        | CPU is in Protected 32-bit Mode                  |
| 0x9047: | movw       | %ax, %ds        | <==        | Load Segment Registers with 32-bit flat segments |
| 0x9049: | movw       | %ax, %es        |            |                                                  |
| 0x904b: | movw       | %ax, %gs        |            |                                                  |
| 0x904d: | movw       | %ax, %fs        |            |                                                  |
| 0x904f: | movw       | %ax, %ss        |            |                                                  |

# Protection Checks

- Loading Segment Register

- ◆ Privilege level checks
- ◆ Segment type checks

LLDT

LTR

- Memory Accesses

- ◆ Segment level protection
- ◆ Page level protection

- Instruction Execution

- ◆ Restriction of instruction set

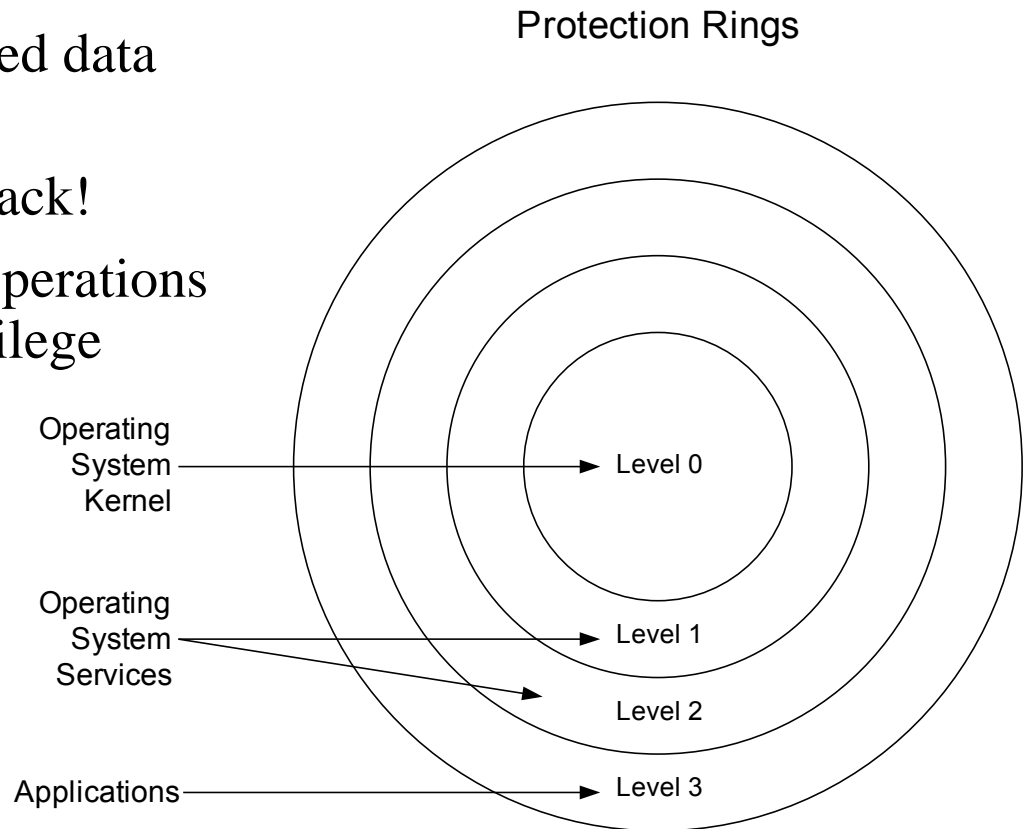
- Transfer Execution Control

- ◆ When transforming control between code segments
- ◆ Type checks
  - Far CALL and JMP
- ◆ Limit checks

- All protection violation results in an exception being generated.

# Privilege Levels

- Lower number => higher privilege
- Code can access data of equal/lower privilege levels only
- Code can call more privileged data via “call gates”
  - Each level has its own stack!
- Some instructions and I/O operations are restricted to certain privilege levels
- Also referred to as:  
“Rings”



# Privilege Level Checks

## ●CPL – Current Privilege Level

- Privilege level of the currently executed program or task
- Stored in bits 0 and 1 of the CS and SS segment registers

## ●DPL – Descriptor Privilege Level

- Privilege level of a segment or gate
- Stored in the DPL field of the segment or gate descriptor

## ●RPL – Requested Privilege Level

- Override privilege level that is assigned to segment selectors
- Stored in bits 0 and 1 of the segment selector
- Used by OS to protect against user Trojan Horses – user code calls OS code and passes a pointer to a privileged data area

# Loading Data Segment Register

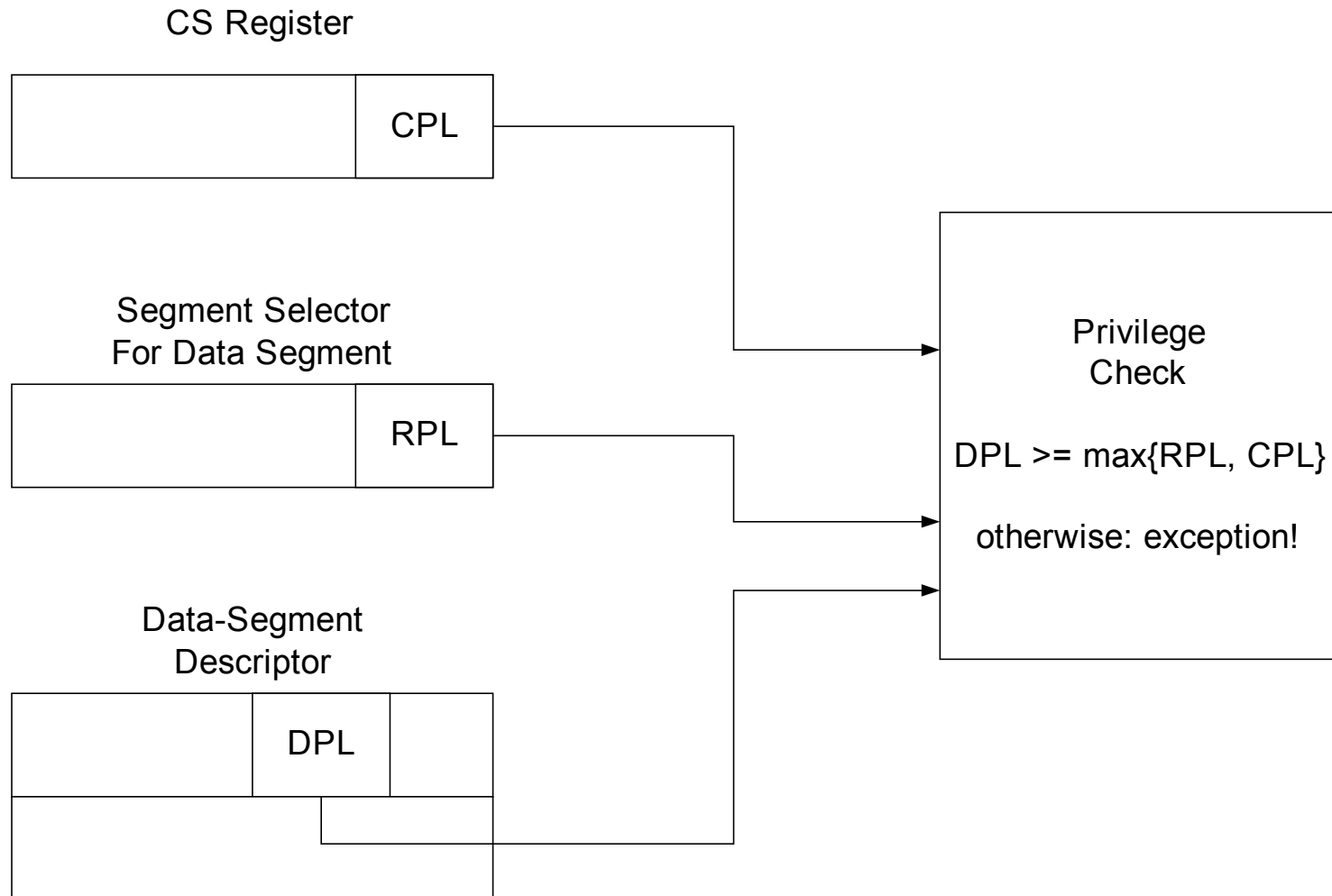
## ● Stack Segment (SS)

- Segment Selector is not Null (  $\geq 0x3$  )
- Segment index is within descriptor table limit
- Segment Descriptor is marked as Present
- Segment is writable data-segment
- CPL = RPL = DPL

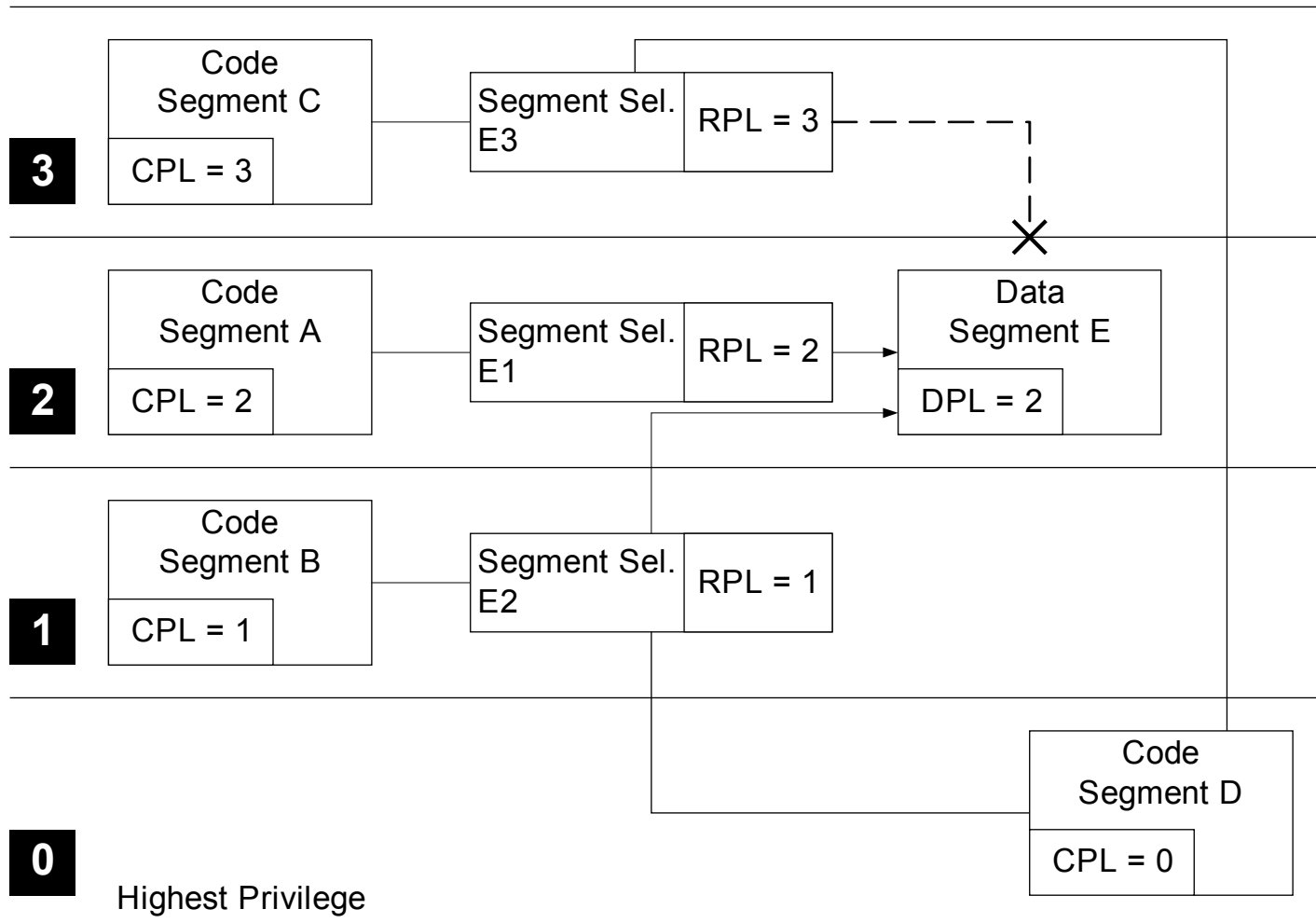
## ● Data Segments (DS, ES, FS, GS)

- Segment Descriptor is marked as Present
- Segment index is within descriptor table limit
- Segment is a data or readable code segment
- Not both RPL and CPL are greater than DPL
- Null selector can be loaded into DS, ES, FS, and GS without causing exception
- Memory access with a Null segment causes exception

# Privilege Check for Data Access



# Examples of Accessing Data Segments



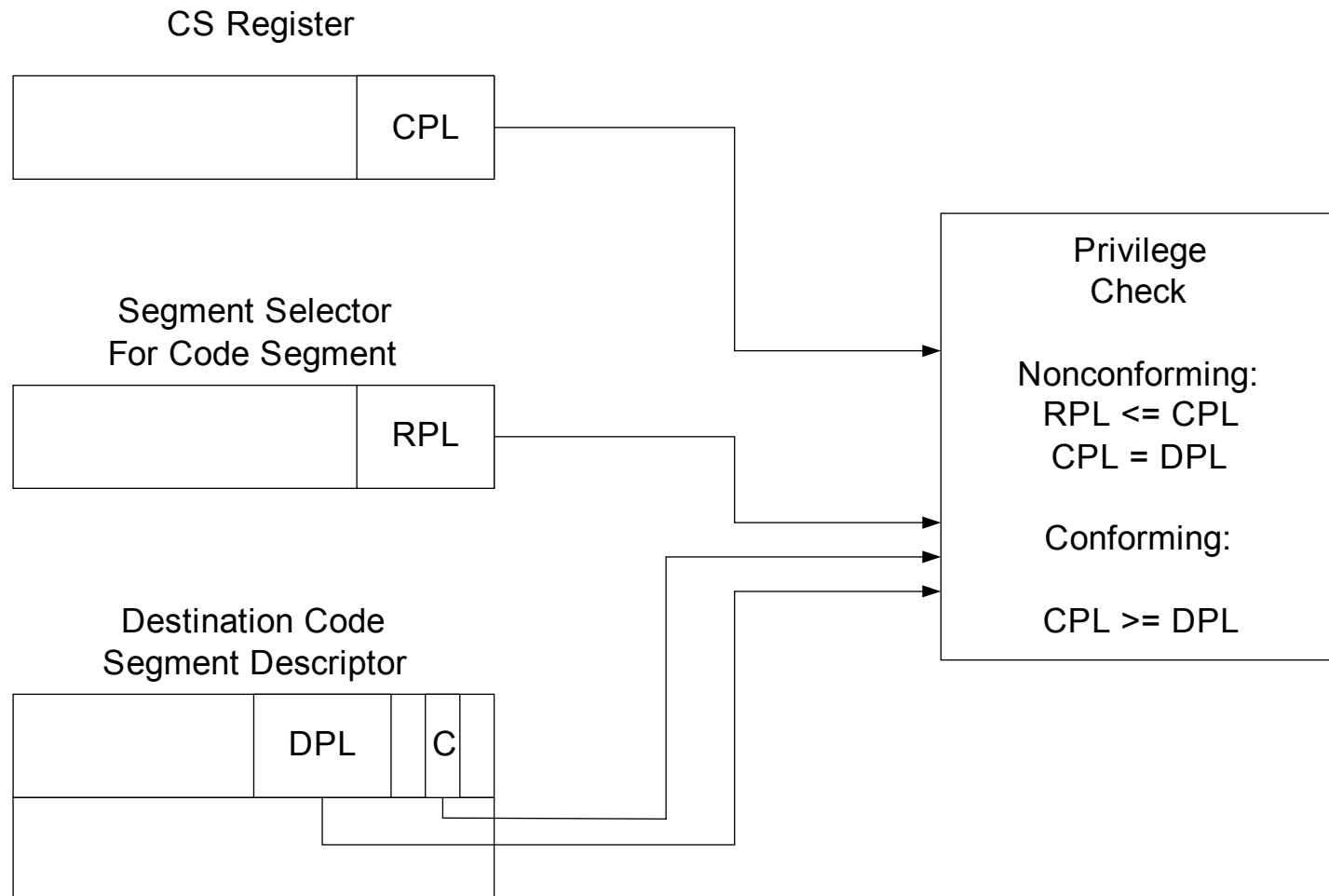
# Loading Code Segment Register

- To transfer program control from one code segment to another, the segment selector for the destination code segment must be loaded into the code segment register (CS).
- Program control transfers are carried out with
  - JMP, CALL, RET, INT n, IRET, SYSENTER and SYSEXIT instructions
  - Exception and interrupt mechanism
- As part of CS loading process, the CPU examines the segment descriptor for the destination code segment and performs various limit, type, and privilege checks.

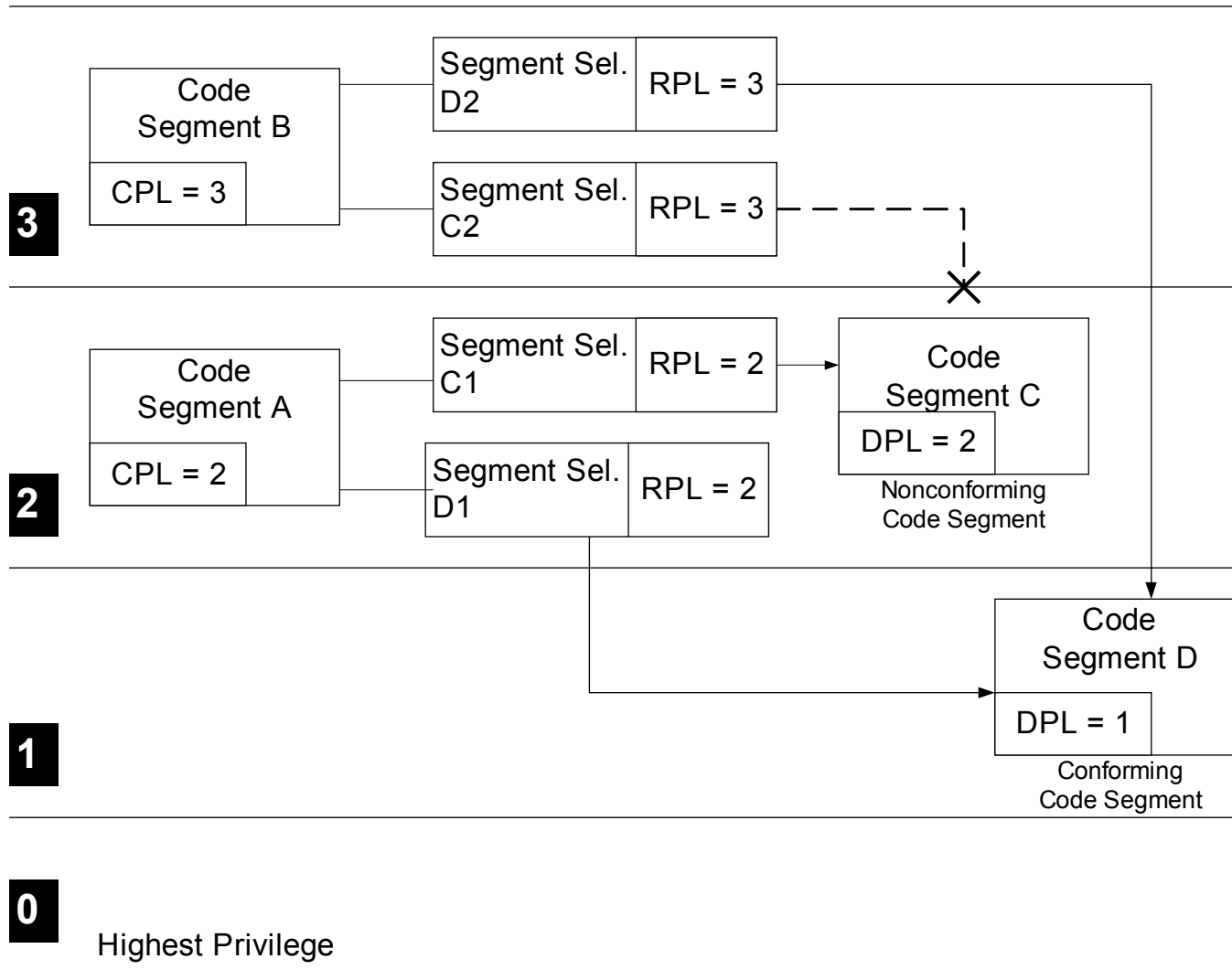
# Loading Code Segment Register

- JMP and CALL instructions can reference another code segment in any of four ways:
  - Target operand contains the segment selector for the target code segment
  - Target operand points to a call-gate descriptor, which contains the segment selector for the target code segment
  - Target operand points to a TSS, which contains the segment selector for the target code segment
  - Target operand points to a task gate, which points to a TSS, which in turn contains the segment selector for the target code segment

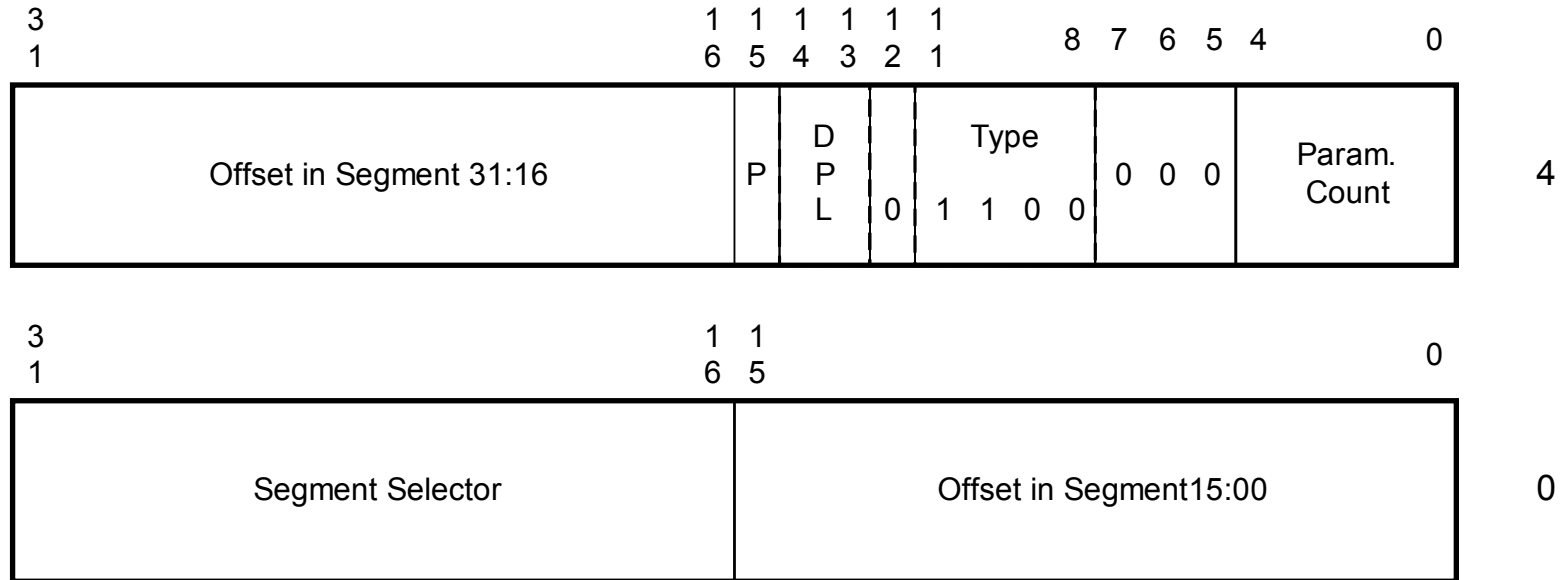
# Direct Calls and Jumps to Code Segment



# Examples of Accessing Code Segments



# Call Gate

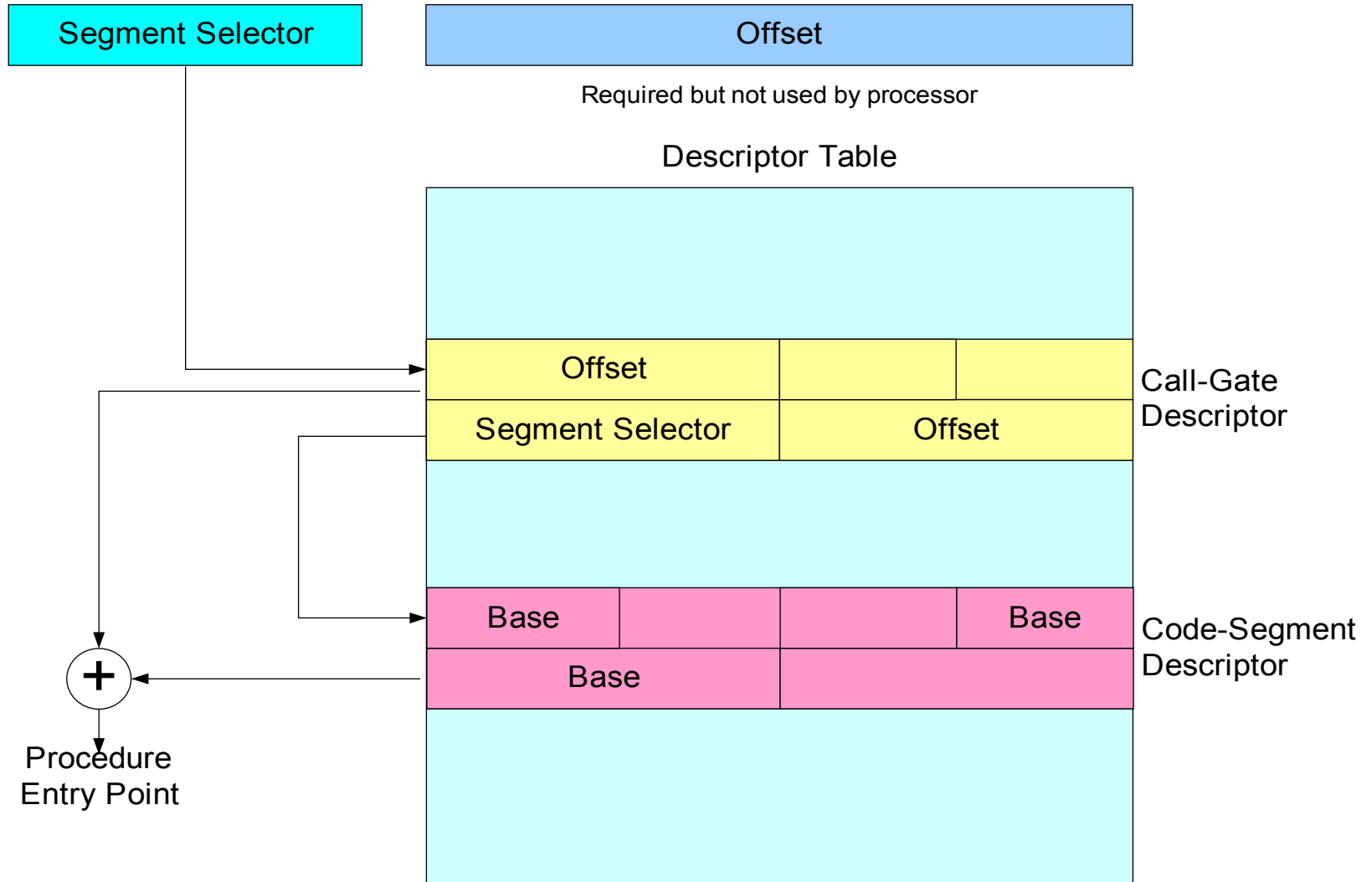


## ● Call Gate specifies

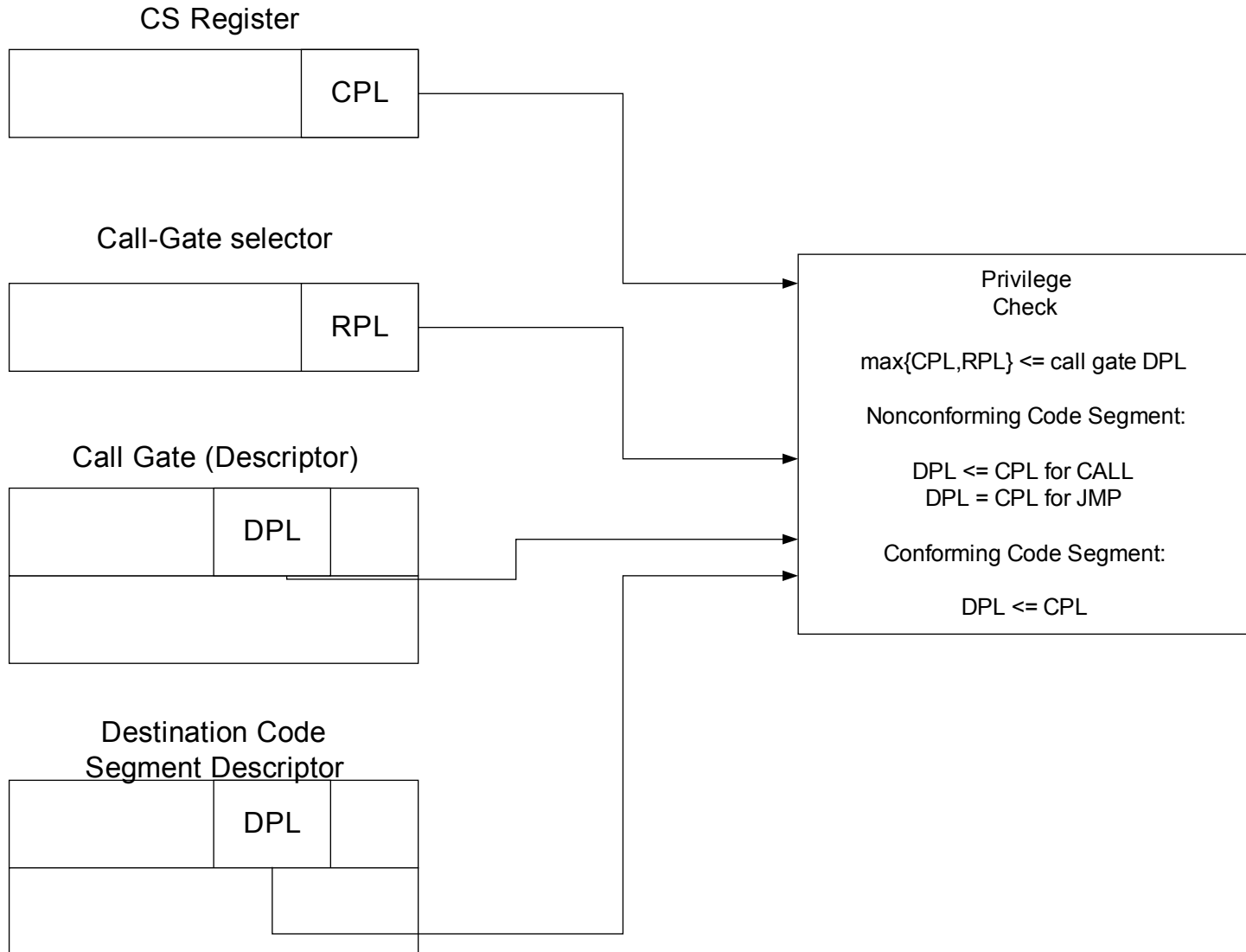
- ◆ The code segment to be accessed
- ◆ An entry point for a procedure in the specified code segment
- ◆ The privilege level required for a caller trying to access the procedure
- ◆ If a stack switch occurs, the number of optional parameters to be copied between stacks
- ◆ The size of values to be pushed onto the target stack: 16-bit or 32-bit
- ◆ Whether the call-gate descriptor is valid

# Accessing a Code Segment Through a Call Gate

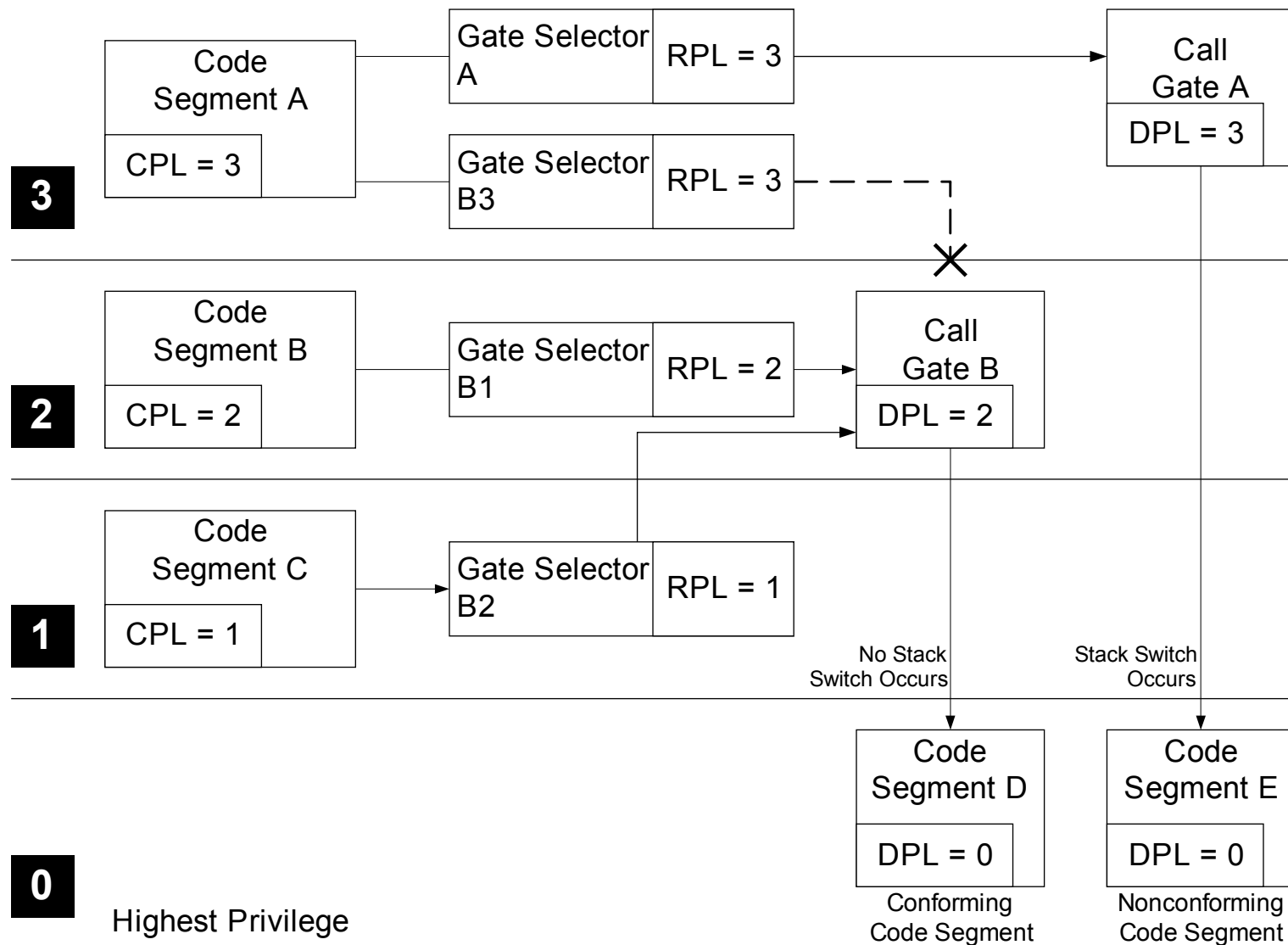
Far Pointer to Call Gate



# Privilege Check Rules for Call Gate



# Examples of Accessing Call Gates



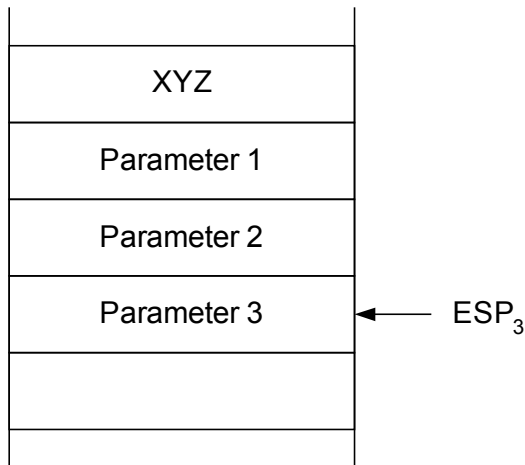
# Inter Level Call - Stack Switching

● On inter-level call: CPL changes, Control transferred, Stack switched

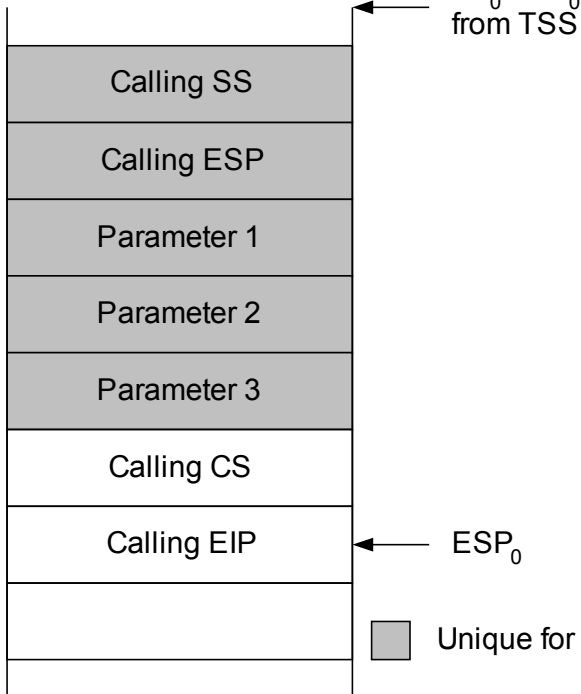
● Stack Switch

- ◆ New (privileged) SS:ESP is taken from the TSS
- ◆ Old SS:ESP is stored on new stack
- ◆ Parameters are copied to new stack

Calling Procedure's Stack  
Before Call

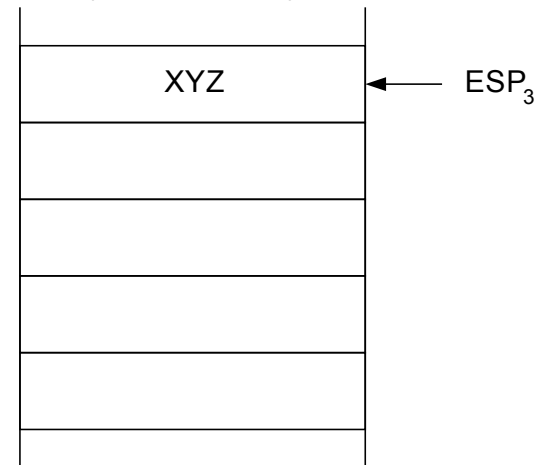


Called Procedure's Stack  
Before Return



SS<sub>0</sub>:ESP<sub>0</sub>  
from TSS

Calling Procedure's Stack  
After Return  
(far RETn n = 3)



Unique for interlevel calls

# Privilege Instructions

- An execution of privilege instructions in privilege level other than 0 causes exception

- ◆ **LGDT** – Load GDT register
- ◆ **LLDT** – Load LDT register
- ◆ **LTR** – Load task register
- ◆ **LIDT** – Load IDT register
- ◆ **MOV CR** – Load and store control registers
- ◆ **LMSW** – Load machine status word
- ◆ **CLTS** – Clear task-switch flag in register CR0
- ◆ **MOV DR** – Load and store debug registers
- ◆ **INVD** – Invalidate cache without write-back
- ◆ **WBINVD** – Invalidate cache with write-back
- ◆ **INVLPG** – Invalidate TLB entry
- ◆ **HLT** – Halt processor
- ◆ **RDMSR/WRMSR** – Read/Write Model-Specific Registers
- ◆ **RDPMSR** – Read Performance-Monitoring Counters
- ◆ **RDTSC** – Read Time-Stamp Counter

# Fast System Call

- SYSENTER instruction
  - ◆ Provides maximum performance for ring3 → ring0 privilege level change
  - ◆ Introduced in Pentium-III processor
- The following Model Specific Registers should be set prior execution:
  - ◆ **SYSENTER\_CS\_MSR** — Contains the segment selector for the privilege level 0 code segment.
  - ◆ **SYSENTER\_EIP\_MSR** — Contains the offset into the privilege level 0 code segment to the first instruction of the selected operating procedure or routine.
  - ◆ **SYSENTER\_ESP\_MSR** — Contains the stack pointer for the privilege level 0 stack.
- SYSENTER operation:
  1. CS ← SYSENTER\_CS\_MSR
  2. EIP ← SYSENTER\_EIP\_MSR
  3. SS ← SYSENTER\_CS\_MSR + 8
  4. ESP ← SYSENTER\_ESP\_MSR
  5. Switches to privilege level 0.
  6. EFLAGS.VM = 0

# Fast Return to User Level

- SYSEXIT instruction
  - ◆ Provides maximum performance for ring0 ring3 privilege level change
  - ◆ Introduced in Pentium-II processor
- Players:
  - ◆ **SYSENTER\_CS\_MSR** — Contains the segment selector for the privilege level 0 code segment.
  - ◆ **EDX** — Contains the offset into the privilege level 3 code segment to the first instruction to be executed in the user code.
  - ◆ **ECX** — Contains the stack pointer for the privilege level 3 stack.
- SYSEXIT operation:
  1. CS     $\text{SYSENTER\_CS\_MSR} + 16$
  2. EIP   EDX
  3. SS     $\text{SYSENTER\_CS\_MSR} + 24$
  4. ESP   ECX
  5. Switches to privilege level 3.
- **Note:** SYSENTER/SYSEXIT do not constitute a CALL/RET. No data is passed on the stack.

# Protected Mode

## Implications and Observations (1)

### ● Software conversion

- ◆ Straightforward if software does not assume linear =  $\text{seg} * 16 + \text{offset}$

### ● Real mode “tricks” do not work. Usually for the better:

- ◆ Cannot write on code
- ◆ Cannot access segment 0
- ◆ Cannot jump/write into the OS

### ● New issues

- ◆ How to write on code (self-modifying code, SMC) - use aliasing
- ◆ How to use OS services - via call gates
- ◆ How to access a known linear address -  
Map a segment to a known base. e.g., use 4G at base 0.
- ◆ How to protect between processes (applications) -  
Use LDT per process. Reload LDTR at context switch

### ● Additional related mechanisms exist - not discussed

# Protected Mode

## Implications and Observations (2)

### ● Segmentation performance impact

Issues:

- ◆ Indirect vs direct addressing
- ◆ Protection check

Solution: information is cached for each active segment

- ◆ Almost no overhead on memory reference... But
- ◆ Change of a segment register is costly!

### ● Segment based virtual memory

- ◆ Possible - through use of “segment not present” bit
- ◆ Unattractive - due to variable length of segments

# Protected Mode in Practice

- Value of segmentation/protected mode
  - ◆ Only option for a 16-bit OS
  - ◆ 32-bit OS prefers flat model, page based protection
- Call gates used for system calls by Linux
  - ◆ Not by Windows NT
- Task switch mechanism rarely if ever used
- OS/2 uses 32-bit segmentation
- Exotic uses for segment registers
  - ◆ FS used by NT for multi-threading

# Interrupts and Exceptions

# What is it?

*“A signal from hardware or software, such as a keystroke, that demands immediate attention and takes priority over other operations”*

● Analogy:

- ◆ Alarm clock
- ◆ Pain

# What Causes It?

## ●Interrupts

- ◆ External to the CPU
- ◆ Peripheral: I/O: mouse move/click, keyboard, timer

## ●Exceptions

- ◆ Internal to the CPU
- ◆ Exceptions: Processor detected:

**Fault:** restart from the causing instruction (e.g., page fault)

**Trap:** restart from the next instruction (e.g., overflow)

**Abort:** cannot restart (double fault)

## ●Programmed Exceptions (“software interrupts”):

- ◆ e.g.: INTO, INT3, INT n, BOUND

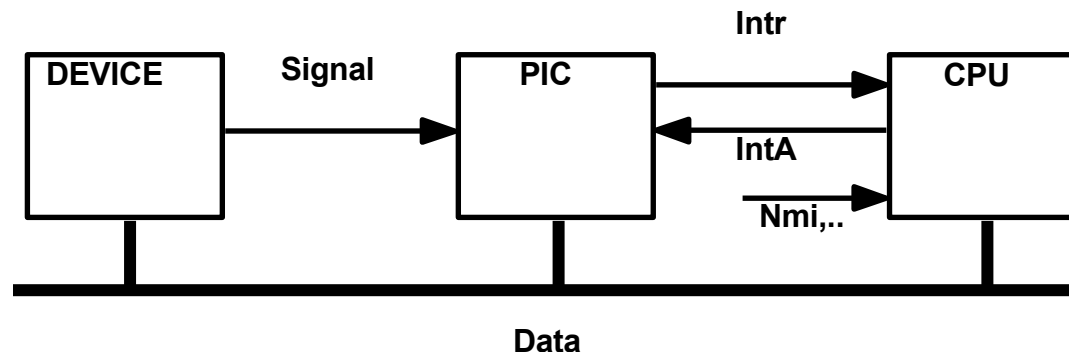
## ●The terms “interrupts” & “exceptions” are frequently intermixed

# Why Is It Important?

- Inevitable: *only way to implement a feature*
  - ◆ Cannot implement timers, page faults, ^break etc. without it
- Difficult without: *alternatives imply huge program overhead*
  - ◆ e.g., polling to identify keyboard stroke
- Convenience: *cheaper way to implement a feature*
  - ◆ e.g., zero divide, INTO, BOUND, etc..
- Conventions: *Application/OS communication*
  - ◆ System call. Allow using procedures without linkage

# How Does It Work?

- Internal interrupts: handled completely by CPU + SW
- External interrupt involves CPU + SW + peripherals
  - ◆ External device delivers “INTR” signal (or NMI, SMI)
  - ◆ CPU acknowledges with “INTA” bus cycle
  - ◆ Device sends interrupt# on Data bus
  - ◆ Usually requires a PIC (Programmable Interrupt Controller) in system
- External interrupts (except NMI/SMI) can be masked (IF=0)



# Additional Players

## ●Flags

- ◆ **Interrupt Flag** EFLAGS.IF flag: masks external interrupts  
Set/Clear by STI/CLI instructions
- ◆ **Trap Flag** EFLAGS.TF flag: is the CPU in TRAP mode (single-step)  
Set/Clear by EFLAGS manipulation
- ◆ **Resume Flag** EFLAGS.RF flag: when set disables debug-exception

## ●Instruction

- ◆ **IRET**: “return from interrupt” = RET + few more things
- ◆ Software interrupts: **INT n, INT3, INTO (4), BOUND (interrupt 5)**

## ●Signals (for external interrupts)

- ◆ INTR: interrupt request
- ◆ INTA: interrupt acknowledge (bus cycle) =  $IO\# * C * R\#$
- ◆ Data Bus: pass interrupt #
- ◆ NMI, SMI: additional interrupt lines
  - NMI - non maskable (timer, power fail)
  - SMI - system management (OS independent power management)

# Interrupt Handling

## ● Hardware side

- ◆ Ensure transparency to the affected task

Will return to the point of interruption w/ the right flags

Does the minimum - only what software cannot do!

## ● Interrupt handlers

- ◆ Transparent: preserve all used registers!

- ◆ Short as possible. If cannot - enable interrupts during handling

Do not disable interrupts for a long time!

- ◆ Avoid nesting of same interrupts, or make sure the software can handle that!

- ◆ Some of the information can/should be extracted from the Stack (old CS:EIP, old SS:ESP, EFLAGS, error code)

## ● Interrupt handler writers must fully understand the relevant HW interaction (PIC + actual peripheral)

# Exception and Interrupt Vectors

- Each exception and interrupt is associated with an identification number, called a **vector**.
  - ◆ Vectors 0-31 are assigned to the exceptions and NMI interrupt  
9,10,11,12,13,14,15,16,17,18,19,20-31 – Intel Reserved
  - ◆ Vectors 32-255 are designated and user defined interrupts.
  
- Exception Classification
  - ◆ **Fault**

An exception that can generally be corrected and then, once corrected, allows the program to be restarted with no loss of continuity
  - ◆ **Trap**

An exception that is reported immediately following the execution of the trapping instruction. Traps allow execution of a program or task to be continued without loss of program continuity
  - ◆ **Abort**

An exception that does not always report the precise location of the instruction causing the execution and does not allow restart of the program or task that caused the execution

# Protected-Mode Exceptions

| No. | Mne-<br>monic | Description              | Type      | Error<br>Code | Source                                |
|-----|---------------|--------------------------|-----------|---------------|---------------------------------------|
| 0   | #DE           | Divide by Zero           | Fault     | No            | DIV / IDIV instructions               |
| 1   | #DB           | Debug                    | Fault/Tra | No            | DR conditions / INT1 instruction.     |
| 2   | -             | NMI Interrupt            | p         | No            | Nonmaskable external interrupt        |
| 3   | #BP           | Breakpoint               | Interrupt | No            | INT3 instruction                      |
| 4   | #OF           | Overflow                 | Trap      | No            | INTO instruction                      |
| 5   | #BR           | BOUND Range Exceeded     | Trap      | No            | BOUND instruction                     |
| 6   | #UD           | Invalid Opcode           | Fault     | No            | UD2 instruction / reserved opcode     |
| 7   | #NM           | Device Not available     | Fault     | No            | FP or WAIT/FWAIT instructions         |
| 8   | #DF           | Double Fault             | Fault     | Yes           | Any exception, interrupt, NMI         |
| 10  | #TS           | Invalid TSS              | Abort     | (Zero)        | Task switch or TSS access             |
| 11  | #NP           | Segment Not Present      | Fault     | Yes           | Loading SR / accessing system         |
| 12  | #SS           | Stack-Segment Fault      | Fault     | Yes           | segment                               |
| 13  | #GP           | General Protection Fault | Fault     | Yes           | Stack operations / SS register loads  |
| 14  | #PF           | Page Fault               | Fault     | Yes           | Memory references / protection checks |
| 16  | #MF           | Floating-Point Error     | Fault     | Yes           | Any memory reference                  |
| 17  | #AC           | Alignment Check          | Fault     | No            | FP or WAIT/FWAIT instructions         |
| 18  | #MC           | Machine Check            | Fault     | Yes           | Any data reference in memory          |
| 19  | #XE           | Streaming SIMD           | Abort     | (Zero)        | Model dependent                       |

Extensions

Fault

No

SIMD floating-point instructions.

# Exceptions and Interrupts Priority

| Priority    | Descriptions                                                                                        |
|-------------|-----------------------------------------------------------------------------------------------------|
| 1 (Highest) | Hardware Reset and Machine Checks                                                                   |
| 2           | Trap on Task Switch                                                                                 |
| 3           | External Hardware Interventions<br>-FLUSH, STOPCLK, SMI, INIT                                       |
| 4           | Traps on the Previous Instruction<br>- Breakpoints, Debug Traps                                     |
| 5           | External Interrupts<br>- NMI, Maskable Hardware Interrupts                                          |
| 6           | Faults from Fetching Next Instruction<br>- Code breakpoint, code-segment violation, Code page fault |
| 7           | Faults from Decoding the Next Instruction<br>- Instruction length > 15, illegal Opcode              |
| 8 (Lowest)  | Faults on Executing an Instruction                                                                  |

# Interrupt Descriptor Table

## ● Interrupt Descriptor Table (IDT)

- ◆ Located in virtual memory
- ◆ Each entry contains 8-bytes Interrupt Descriptor

## ● Interrupt Descriptor Table Register (IDTR)

- ◆ Contains:

IDT Linear Base Address

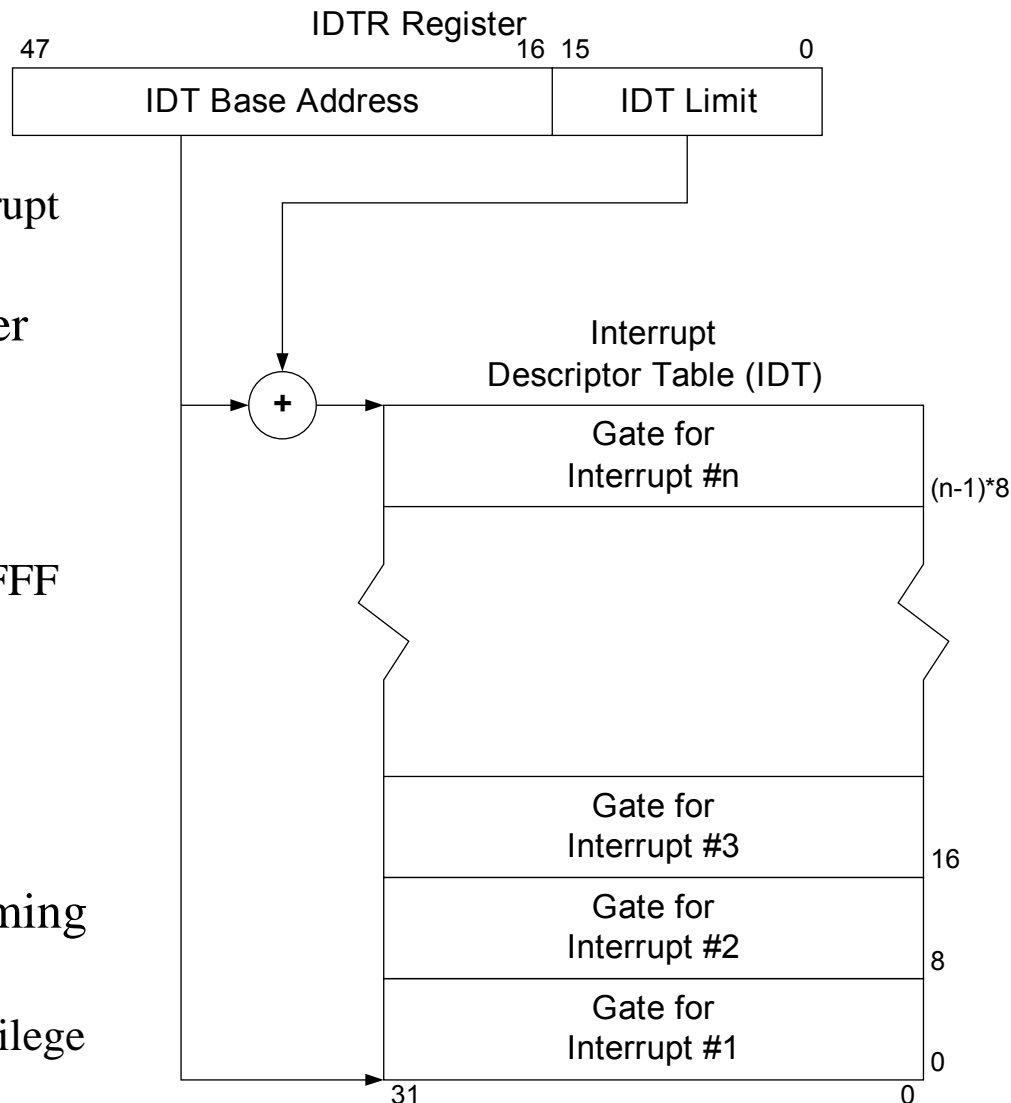
IDT Limit in bytes, max 0xFFFF

## ● Operating System loads IDTR by executing LIDT instruction

- ◆ LIDT is a privilege instruction
- ◆ Can be executed only in ring0

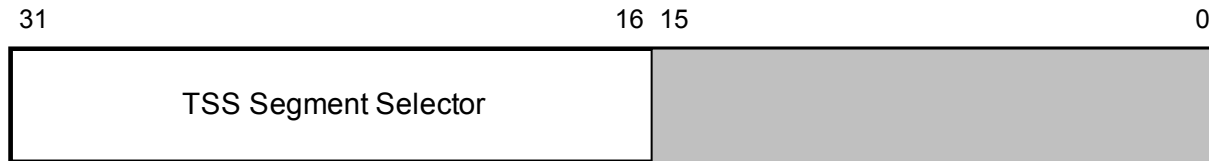
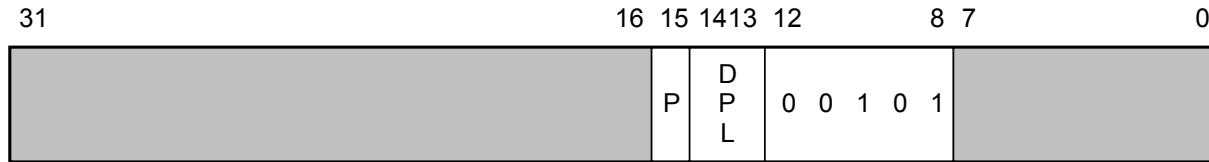
## ● One can observe IDTR by performing SIDT instruction

- ◆ SIDT can be executed in any privilege level

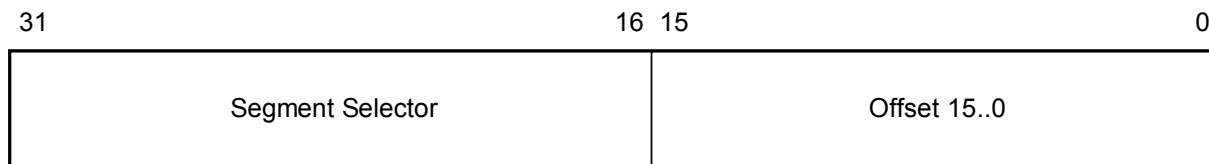
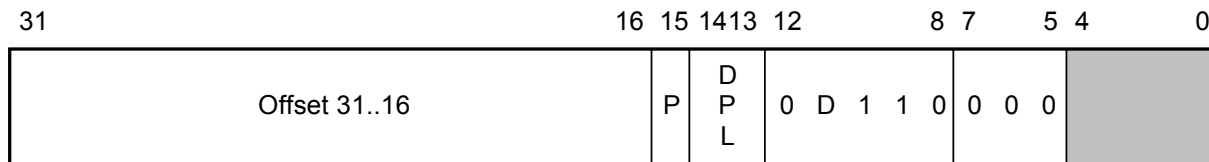


# IDT Descriptors

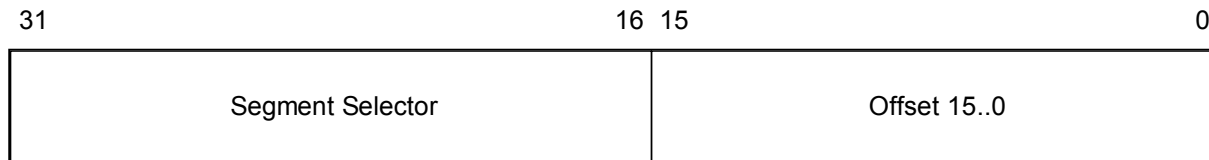
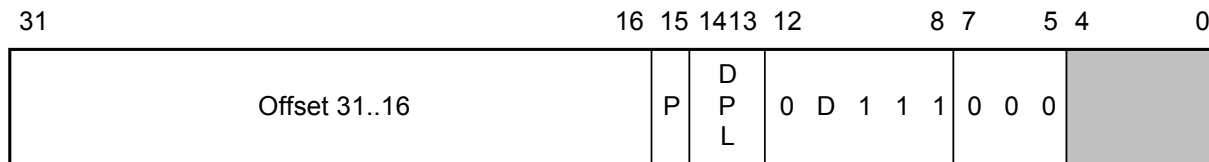
## Task gate



## Interrupt Gate



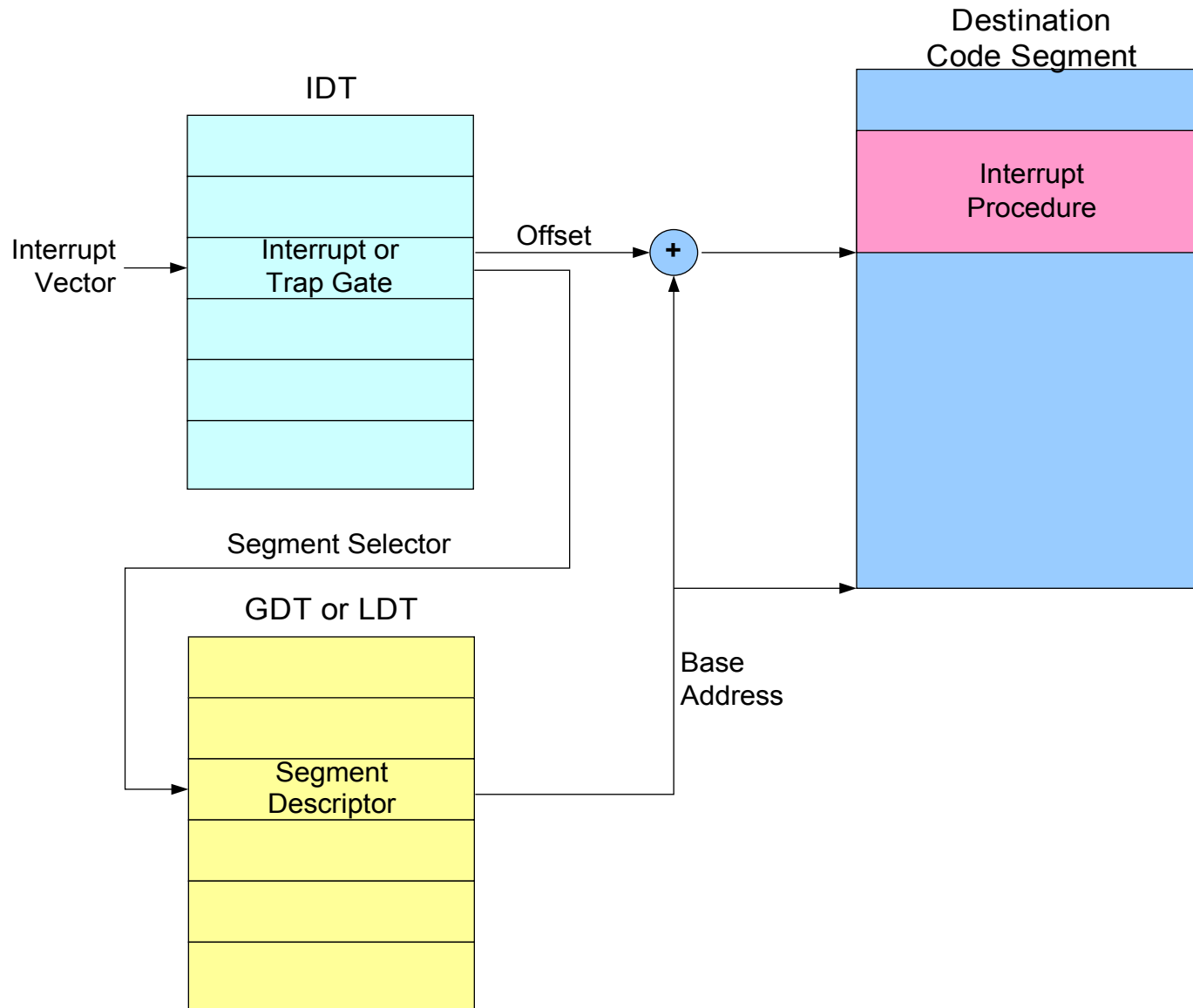
## Trap gate



# Linux IDT - Example

| Brd 0 IDT [P0] |         |                   |         |          |   |   |   |   |   | A | D | C | T |
|----------------|---------|-------------------|---------|----------|---|---|---|---|---|---|---|---|---|
| Indx           | Type    | Descriptor        | Bas/Sel | Lim/Off  | G | D | O | L | P | V | P | N | Y |
| 0              | TRAPG32 | c0108f00 00107268 | 0010    | c0107268 |   |   |   |   | 1 | 0 |   | 0 | f |
| 1              | TRAPG32 | c0108f00 00107308 | 0010    | c0107308 |   |   |   |   | 1 | 0 |   | 0 | f |
| 2              | INTG32  | c0108e00 00107314 | 0010    | c0107314 |   |   |   |   | 1 | 0 |   | 0 | e |
| 3              | TRAPG32 | c010ef00 0010734c | 0010    | c010734c |   |   |   |   | 1 | 3 |   | 0 | f |
| 4              | TRAPG32 | c010ef00 00107358 | 0010    | c0107358 |   |   |   |   | 1 | 3 |   | 0 | f |
| 5              | TRAPG32 | c010ef00 00107364 | 0010    | c0107364 |   |   |   |   | 1 | 3 |   | 0 | f |
| 6              | TRAPG32 | c0108f00 00107370 | 0010    | c0107370 |   |   |   |   | 1 | 0 |   | 0 | f |
| 7              | TRAPG32 | c0108f00 001072c8 | 0010    | c01072c8 |   |   |   |   | 1 | 0 |   | 0 | f |
| 8              | TRAPG32 | c0108f00 00107388 | 0010    | c0107388 |   |   |   |   | 1 | 0 |   | 0 | f |
| 9              | TRAPG32 | c0108f00 0010737c | 0010    | c010737c |   |   |   |   | 1 | 0 |   | 0 | f |
| a              | TRAPG32 | c0108f00 00107394 | 0010    | c0107394 |   |   |   |   | 1 | 0 |   | 0 | f |
| b              | TRAPG32 | c0108f00 001073a0 | 0010    | c01073a0 |   |   |   |   | 1 | 0 |   | 0 | f |
| c              | TRAPG32 | c0108f00 001073ac | 0010    | c01073ac |   |   |   |   | 1 | 0 |   | 0 | f |
| d              | TRAPG32 | c0108f00 001073b8 | 0010    | c01073b8 |   |   |   |   | 1 | 0 |   | 0 | f |
| e              | INTG32  | c0108e00 001073d0 | 0010    | c01073d0 |   |   |   |   | 1 | 0 |   | 0 | e |
| f              | TRAPG32 | c0108f00 001073e8 | 0010    | c01073e8 |   |   |   |   | 1 | 0 |   | 0 | f |
| 10             | TRAPG32 | c0108f00 001072b0 | 0010    | c01072b0 |   |   |   |   | 1 | 0 |   | 0 | f |
| 11             | TRAPG32 | c0108f00 001073c4 | 0010    | c01073c4 |   |   |   |   | 1 | 0 |   | 0 | f |
| 12             | TRAPG32 | c0108f00 001073dc | 0010    | c01073dc |   |   |   |   | 1 | 0 |   | 0 | f |
| 13             | TRAPG32 | c0108f00 001072bc | 0010    | c01072bc |   |   |   |   | 1 | 0 |   | 0 | f |
| 14             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 15             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 16             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 17             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 18             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 19             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 1a             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 1b             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 1c             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 1d             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 1e             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 1f             | INTG32  | c0108e00 00100240 | 0010    | c0100240 |   |   |   |   | 1 | 0 |   | 0 | e |
| 20             | INTG32  | c0228e00 0010be08 | 0010    | c022be08 |   |   |   |   | 1 | 0 |   | 0 | e |
| 21             | INTG32  | c0228e00 0010be10 | 0010    | c022be10 |   |   |   |   | 1 | 0 |   | 0 | e |
| 22             | INTG32  | c0228e00 0010be18 | 0010    | c022be18 |   |   |   |   | 1 | 0 |   | 0 | e |
| 23             | INTG32  | c0228e00 0010be20 | 0010    | c022be20 |   |   |   |   | 1 | 0 |   | 0 | e |
| 24             | INTG32  | c0228e00 0010be28 | 0010    | c022be28 |   |   |   |   | 1 | 0 |   | 0 | e |

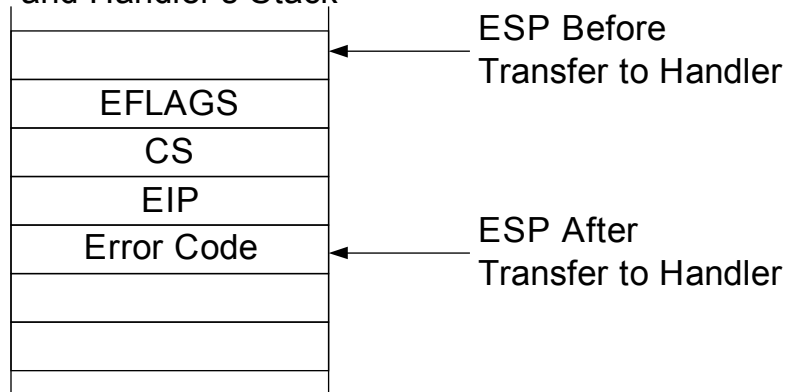
# Interrupt Procedure Call



# Stack Usage on Interrupt Transfer

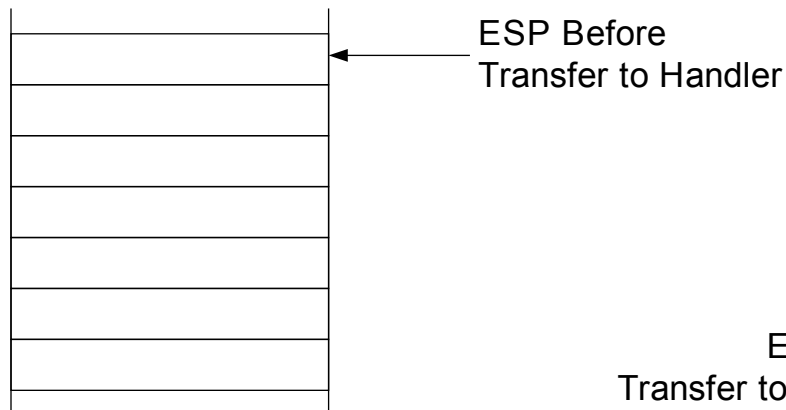
## Stack Usage with No Privilege-Level Change

Interrupted Procedure's  
and Handler's Stack

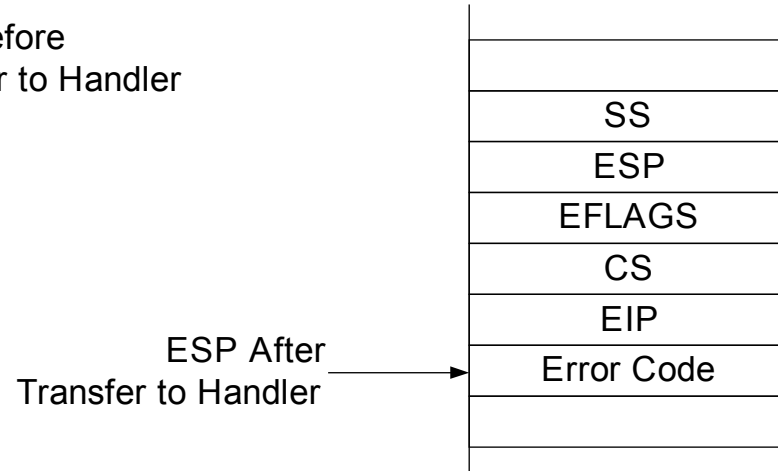


## Stack Usage with Privilege-Level Change

Interrupted Procedure's  
Stack

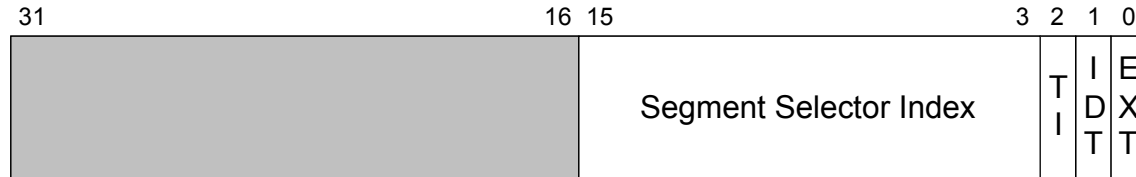


Handler's Stack



# Exception Error Codes

## General Error Code



Index refers to descriptor in GDT (0) or LDT (1) \_\_\_\_\_

Index refers to descriptor in IDT (1) or GDT/LDT (0) \_\_\_\_\_

External (1) or internal (0) event \_\_\_\_\_

## Page-Fault Error Code



Reserved Bits Violation (1) \_\_\_\_\_

System (0) or user (1) code \_\_\_\_\_

Read (0) or Write (1) operation \_\_\_\_\_

Nonpresent page (0) or page-level protection violation (1) \_\_\_\_\_

# Interrupt and Exception Classes

| Class                                  | Vector Number | Description                    |
|----------------------------------------|---------------|--------------------------------|
| <b>Benign Exception and Interrupts</b> | 1             | Debug Exception                |
|                                        | 2             | NMI Interrupt                  |
|                                        | 3             | Breakpoint                     |
|                                        | 4             | Overflow                       |
|                                        | 5             | BOUND Range Exceeded           |
|                                        | 6             | Invalid Opcode                 |
|                                        | 7             | Device Not Available           |
|                                        | 9             | Coprocessor Segment Overrun    |
|                                        | 16            | Floating-Point Error           |
|                                        | 17            | Alignment Check                |
|                                        | 18            | Machine Check                  |
|                                        | 19            | SIMD floating-point extensions |
|                                        | All           | Software Interrupts INT n      |
| <b>Contributory Exceptions</b>         | All           | External Interrupts INTR       |
|                                        | 0             | Divide Error                   |
|                                        | 10            | Invalid TSS                    |
|                                        | 11            | Segment Not Present            |
|                                        | 12            | Stack Fault                    |
| <b>Page Faults</b>                     | 13            | General Protection             |
|                                        | 14            | Page Fault                     |

# Conditions for Generating a Double Fault

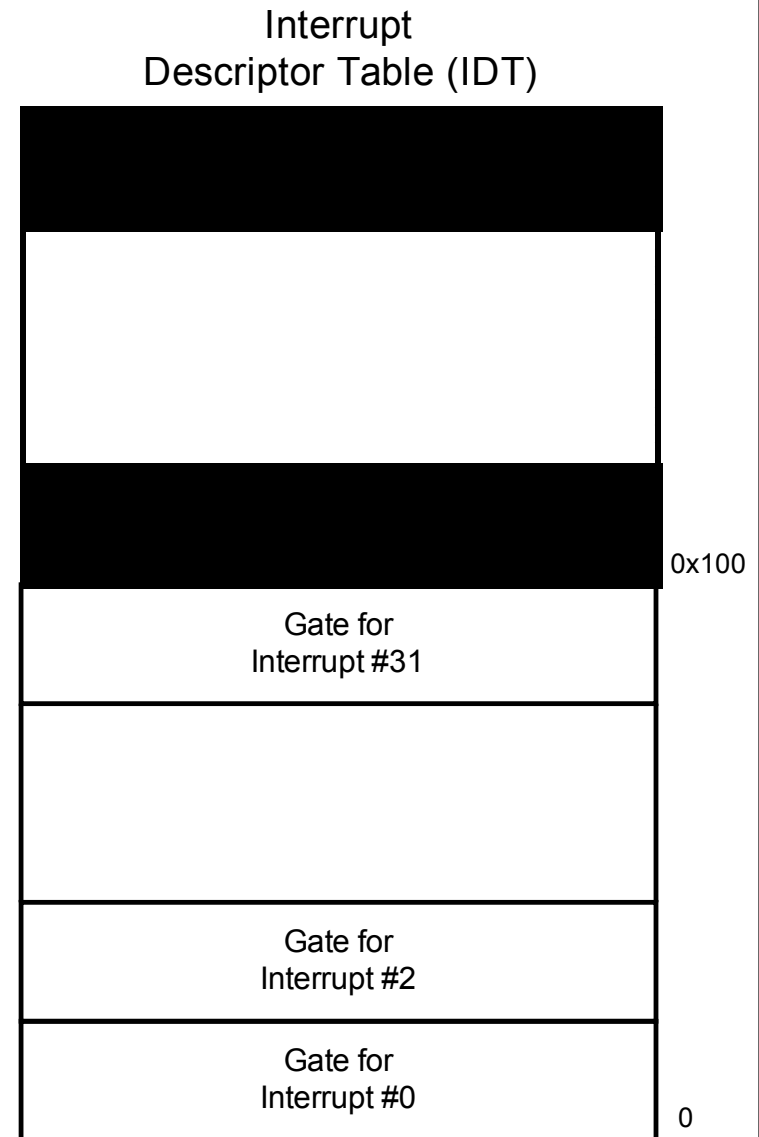
| First Exception     | Second Exception           |                            |                            |
|---------------------|----------------------------|----------------------------|----------------------------|
|                     | Benign                     | Contributory               | Page Fault                 |
| <b>Benign</b>       | Handle Exceptions Serially | Handle Exceptions Serially | Handle Exceptions Serially |
| <b>Contributory</b> | Handle Exceptions Serially | Generate a Double Fault    | Handle Exceptions Serially |
| <b>Page Fault</b>   | Handle Exceptions Serially | Generate a Double Fault    | Generate a Double Fault    |

# Triple Fault and Shutdown State

- **Triple fault** is a case when another exception occurs while attempting to call the double-fault handler, the processor enters **shutdown** mode.
- Shutdown mode is similar to the state following execution of an HLT instruction.
- In this mode the processor stops executing instructions until an NMI interrupt, SMI interrupt, hardware reset, or INIT# is received.

# Example – Exercise

- Load IDT with the limit equal to 0xFF (32 descriptors)
  - ◆ All External Interrupts descriptors (32-255) exceed IDT limit
- When external interrupt is being delivered the processor does the following steps:
  - ◆ Trying to access descriptor according to interrupt vector
  - ◆ Detecting that the descriptor exceeds IDT limit
  - ◆ Generating of a general-protection fault #GP(vector)
  - ◆ Error code contains the vector number and IDT bit is set
- #GP handler can check the IDT and EXT bits of the error code and retrieve the external interrupt vector in the selector part of the error code
- Write an example of such #GP handler.



```

typedef struct IDTR_s {
 __int64 limit : 16,
 base : 32,
 res : 16;
} IDTR_t;

typedef struct Interrupt_Gate_s {
 __int64 offset_15_00 : 16,
 seg_sel : 16,
 res : 12,
 system : 1,
 dpl : 2,
 present : 1,
 offset_31_16 : 16;
} Interrupt_Gate_t;

unsigned handler;
unsigned original_gp_handler;
extern void exception_13(void);

```

```

void get_handler(unsigned error_code)
{
 Interrupt_Gate_t* idt;
 IDTR_t idtr;

 // Get IDTR
 __asm
 {
 sidt idtr
 }

 // Assign the base pointer
 idt = (Interrupt_Gate_t*)idtr.base;

 // Get the required handler offset
 handler = (idt[error_code >> 3].offset_31_16 << 16) |
 idt[error_code >> 3].offset_15_00;
}

```

```

void install_our_gp_handler()
{
 Interrupt_Gate_t* idt;
 IDTR_t idtr;

 // Get IDTR
 __asm
 {
 sidt idtr
 }

 // Assign the base pointer
 idt = (Interrupt_Gate_t*)idtr.base;

 // Save the original GP handler
 original_gp_handler = (idt[0xd].offset_31_16 << 16) |

 idt[0xd].offset_15_00;

```

```

// Replace the GP handler with our handler
idt[0xd].offset_15_00 = exception_13 & 0x0000FFFF;
idt[0xd].offset_31_16 = exception_13 >> 16;

// Cut the IDT limit
idtr.limit = 0xFF

// Load IDTR with new limit
__asm
{
 lidt idtr
}
}

```

```

// Actual GP handler to be invoked upon any GP or External
// Interrupt
exception_13: cli // Disable
 // interrupts

 push eax //
 mov eax, [esp+4] // Check
 and eax, 7h // Error
 cmp 3h, eax // Code
 pop eax //
 jne NATIVE_GP //

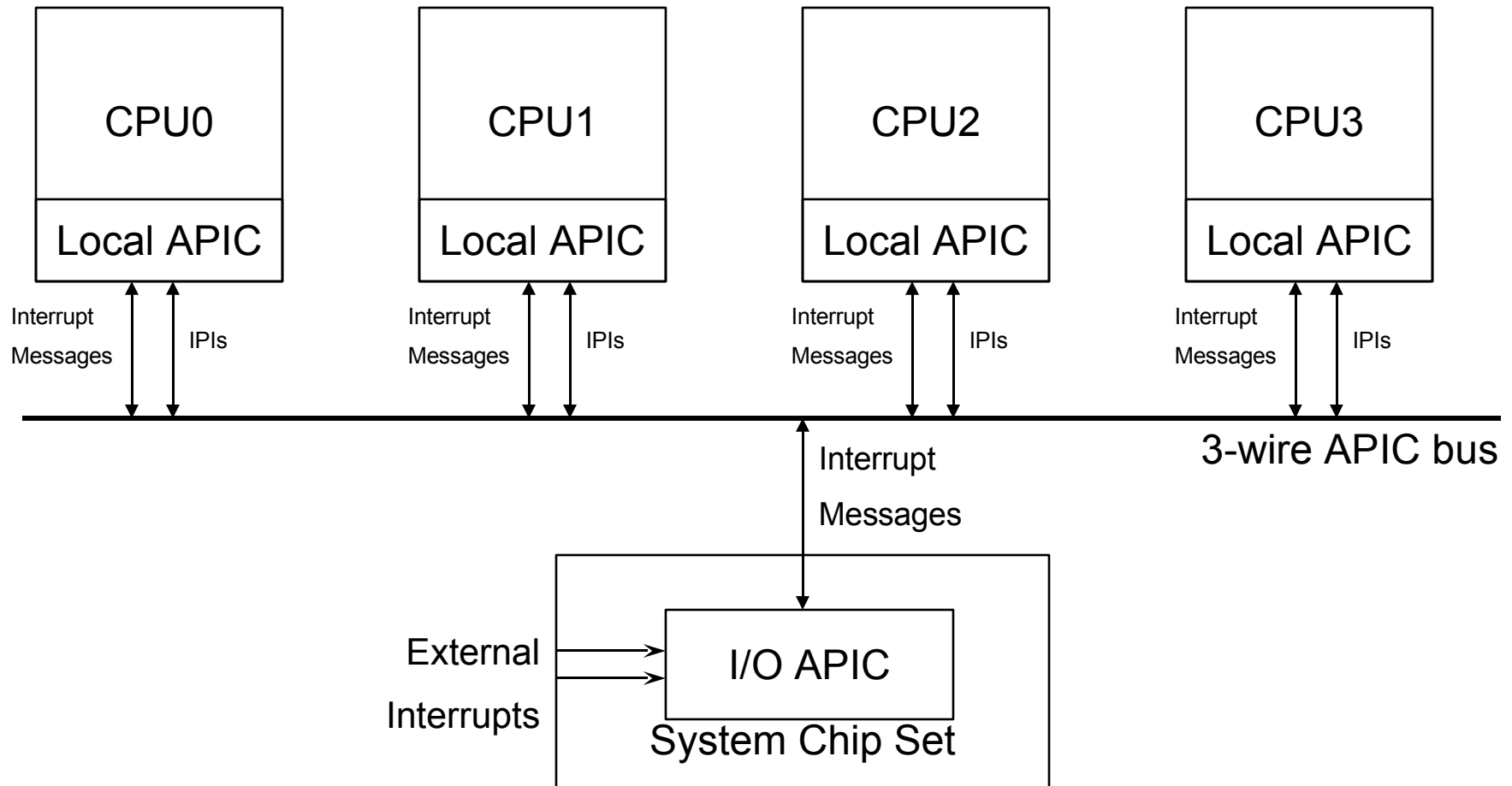
 call get_handler //
 add esp, 4h // Remove Error
 // Code from stack
 jmp handler // Jump to original
 // int. handler
NATIVE_GP: jmp original_gp_handler // Jump to
 // original GP
 // handler

```

# Advanced Programmable Interrupt Controller (I/O APIC)

- Today part of the “chip-set”
  - ◆ Full software compatibility
  - ◆ Supports multiprocessor systems
- Functions:
  - ◆ React to an external interrupt, according to priorities
  - ◆ Convert the interrupt request (IRQ) into a vector
  - ◆ Inform the peripheral that the interrupt was acknowledged
- Vector addresses can be programmed
  - ◆ Using a set of “commands”

# Local APICs and I/O APIC



# Local APIC

## ●Locally connected I/O devices.

- ◆ Interrupts from I/O devices directly connected to CPU's local interrupt pins.

## ●Externally connected I/O devices.

- ◆ Interrupts sent as I/O interrupt messages from the I/O APIC to one or more the IA-32 processors in the system.

## ●Inter-processor interrupts (IPIs).

- ◆ An IA-32 processor can use the IPI mechanism to interrupt another processor or group of processors on the system bus.

## ●APIC timer generated interrupts.

- ◆ The local APIC timer can be programmed to send a local interrupt to its associated processor when a programmed count is reached.

## ●Performance monitoring counter interrupts.

- ◆ Ability to send a interrupt to its associated processor when a performance-monitoring counter overflows.

## ●Thermal Sensor interrupts.

- ◆ Ability to send an interrupt to themselves when the internal thermal sensor has been tripped.

## ●APIC internal error interrupts.

- ◆ Ability to send an interrupt when an error is recognized within the local APIC

# Local APIC Register Address Map

| Address                          | Register Name                       | Software Read/Write            |
|----------------------------------|-------------------------------------|--------------------------------|
| 0xFEE00020                       | Local APIC ID Register              | Read/Write                     |
| 0xFEE00030                       | Local APIC Version Reg.             | Read Only                      |
| 0xFEE00080                       | Task Priority Register (TPR)        | Read/Write                     |
| 0xFEE00090                       | Arbitration Priority Register (APR) | Read Only                      |
| 0xFEE000A0                       | Processor Priority Register (PPR)   | Read Only                      |
| 0xFEE000B0                       | End Of Interrupt (EOI) Reg.         | Write Only                     |
| 0xFEE000D0                       | Logical Destination Register        | Read/Write                     |
| 0xFEE000E0                       | Destination Format Register         | Bits 0-27 RO;<br>Bits 28-31 RW |
| 0xFEE000F0                       | Spurious Interrupt Vector Register  | Bits 0-8 RW;<br>Bits 9-31 RO   |
| 0xFEE00100 through<br>0xFEE00170 | In-Service Register (ISR)           | Read Only                      |

# Local APIC Register Address Map

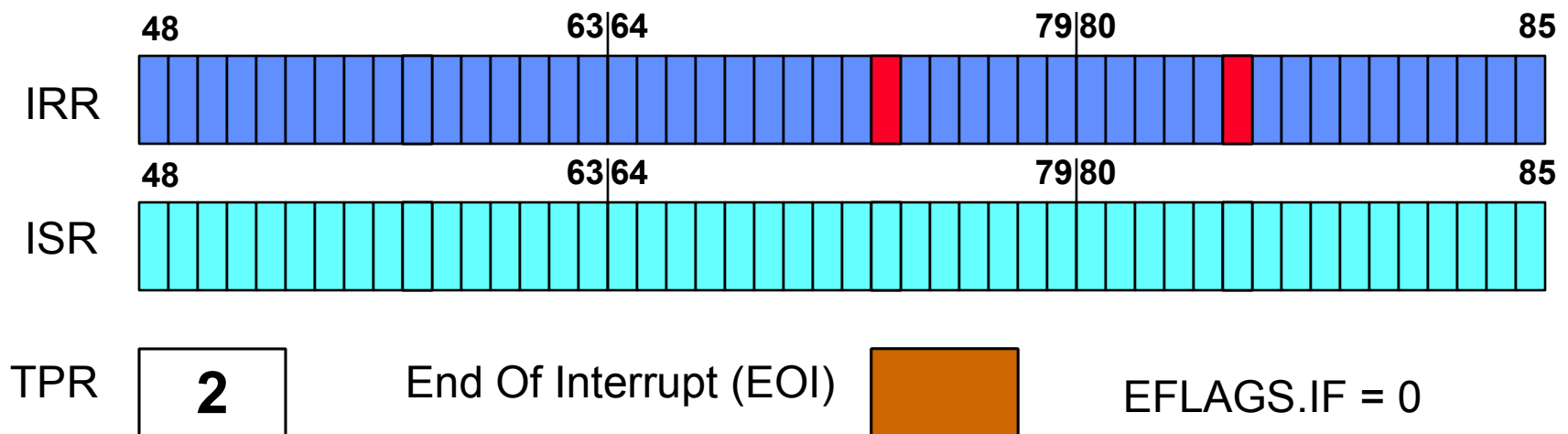
| Address                       | Register Name                            | Software Read/Write |
|-------------------------------|------------------------------------------|---------------------|
| 0xFEE00180 through 0xFEE001F0 | Trigger Mode Register (TMR)              | Read Only           |
| 0xFEE00200 through 0xFEE00270 | Interrupt Request Register (IRR)         | Read Only           |
| 0xFEE00280                    | Error Status Register                    | Read Only           |
| 0xFEE00300                    | Interrupt Command Register (ICR) [0-31]  | Read/Write          |
| 0xFEE00310                    | Interrupt Command Register (ICR) [32-64] | Read/Write          |
| 0xFEE00320                    | LVT Timer Register                       | Read/Write          |
| 0xFEE00330                    | LVT Thermal Sensor Register              | Read/Write          |
| 0xFEE00340                    | LVT Performance Monitor Counters Reg.    | Read/Write          |
| 0xFEE00350                    | LVT LINT0 Register                       | Read/Write          |
| 0xFEE00360                    | LVT LINT1 Register                       | Read/Write          |

# Local APIC Register Address Map

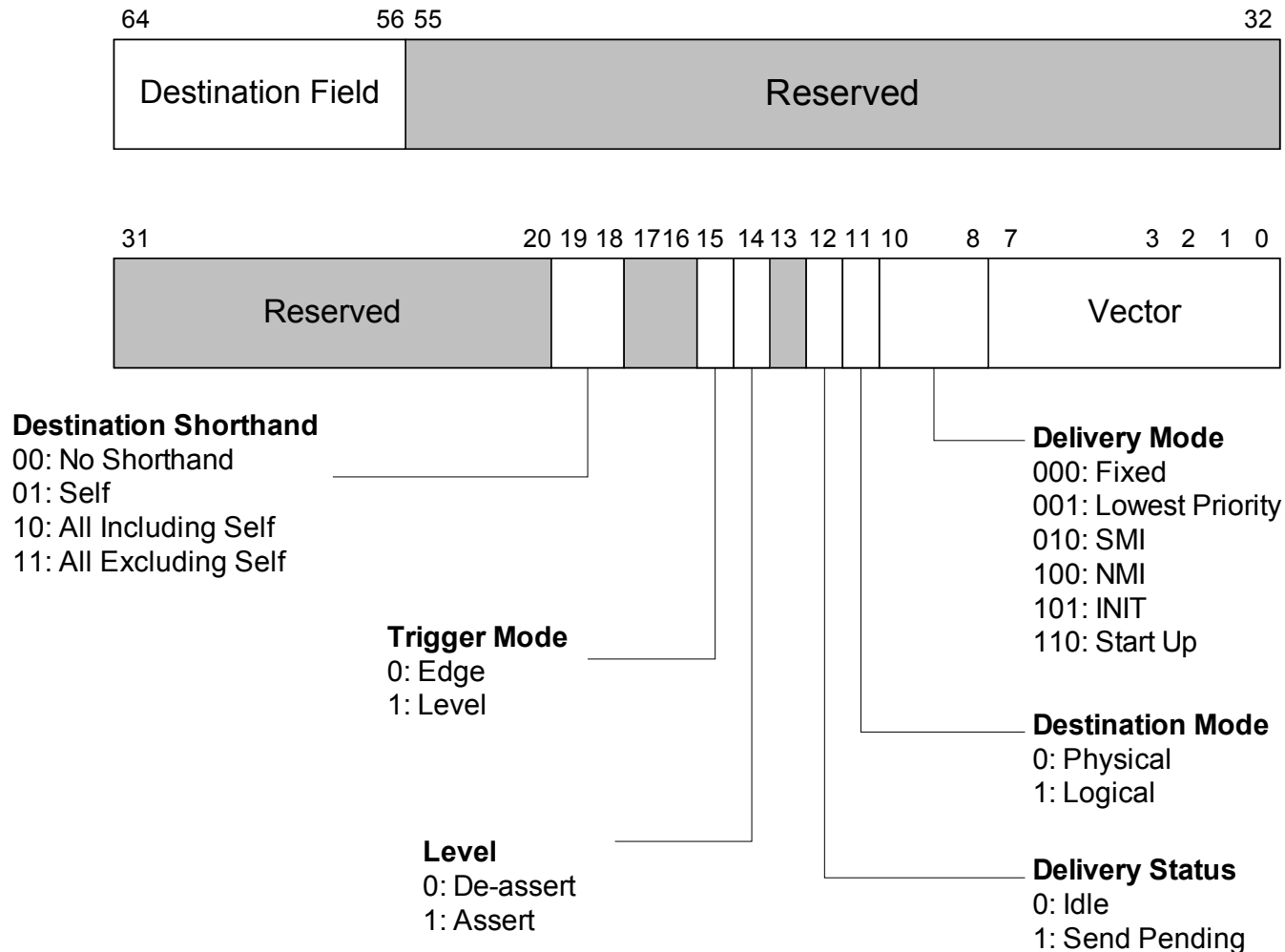
| Address    | Register Name                             | Software<br>Read/Write |
|------------|-------------------------------------------|------------------------|
| 0xFEE00370 | LVT Error Register                        | Read/Write             |
| 0xFEE00380 | Initial Timer Register (for Timer)        | Read/Write             |
| 0xFEE00390 | Current Counter Register (for Timer)      | Read Only              |
| 0xFEE003E0 | Divide Configuration Register (for Timer) | Read/Write             |

# In-Service Register (ISR) and Interrupt Request Register (IRR)

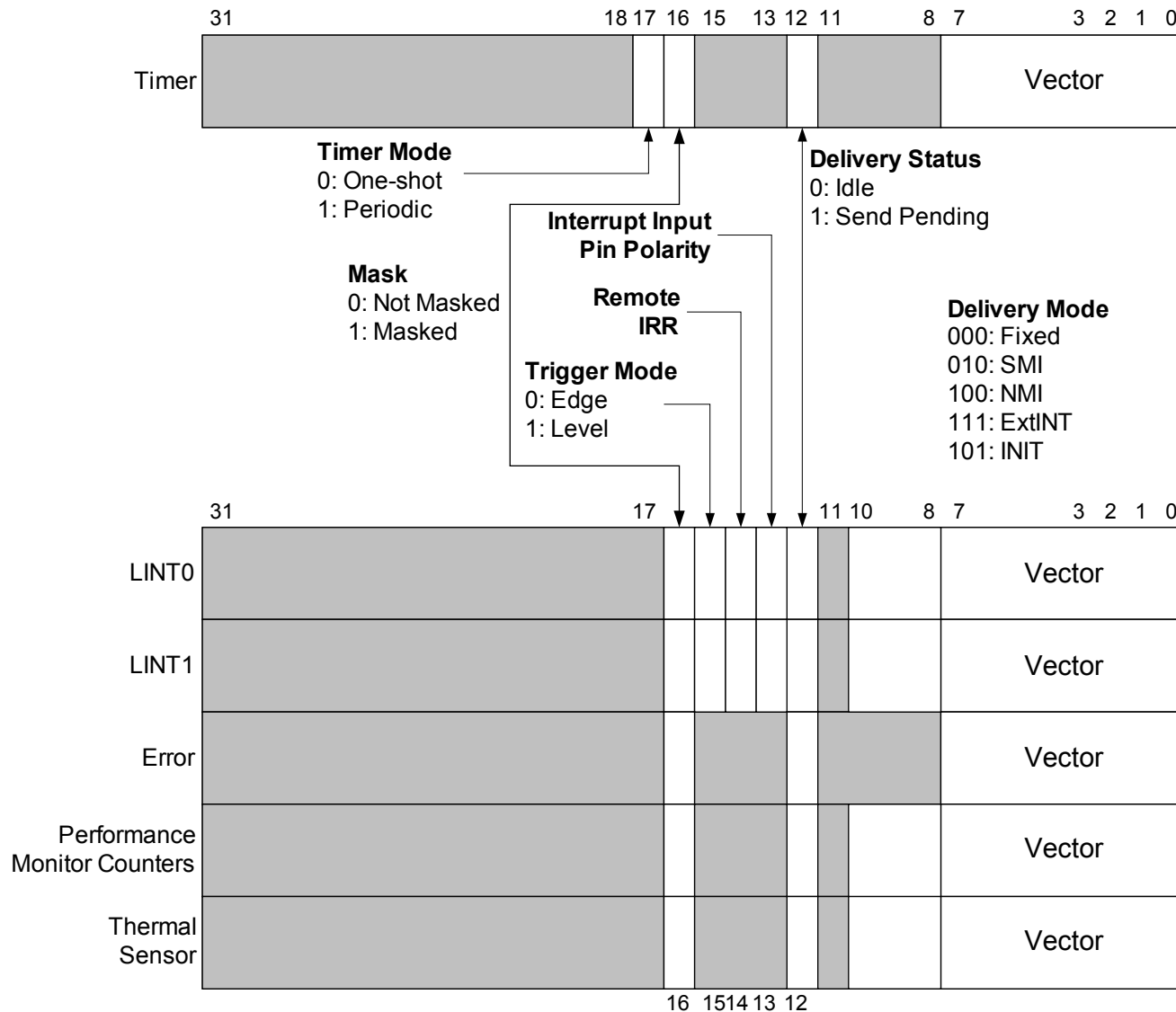
- Interrupt priority = vector / 16 (actually from 2 to 15)
- Each priority level encompasses 16 vectors
- The higher the vector number, the higher the priority within that priority level
- The task priority is a software selected value between 0 and 15 written into the Task Priority Register (TPR)
- The processor will service only those interrupts that have a priority higher than specified in TPR



# Interrupt Command Register (ICR)

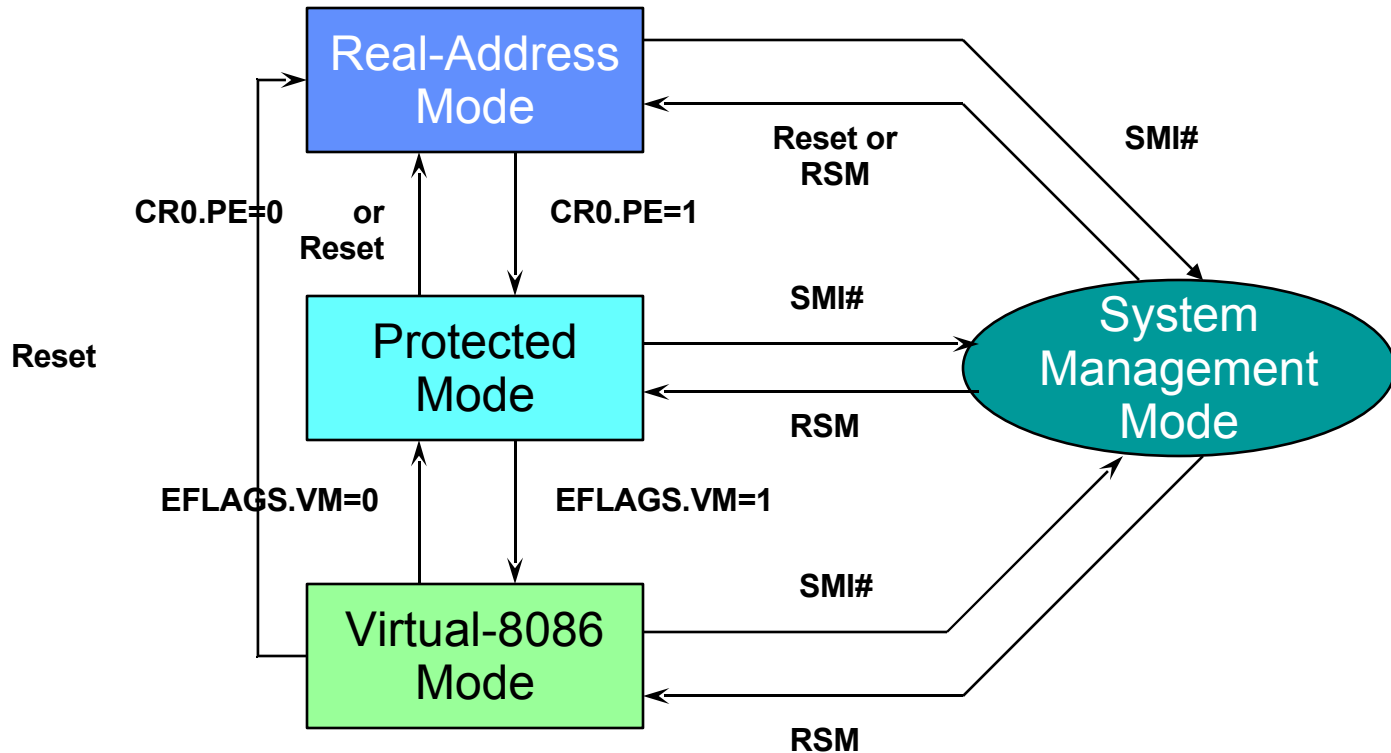


# Local Vector Table (LVT)



# System Architecture Summary

# Operation Modes



## ● 16 bit

- ◆ Real Mode (8086 and up)
- ◆ Protected mode (286 and up)
- ◆ Virtual 8086 mode (386 and up)

## ● 32 bit

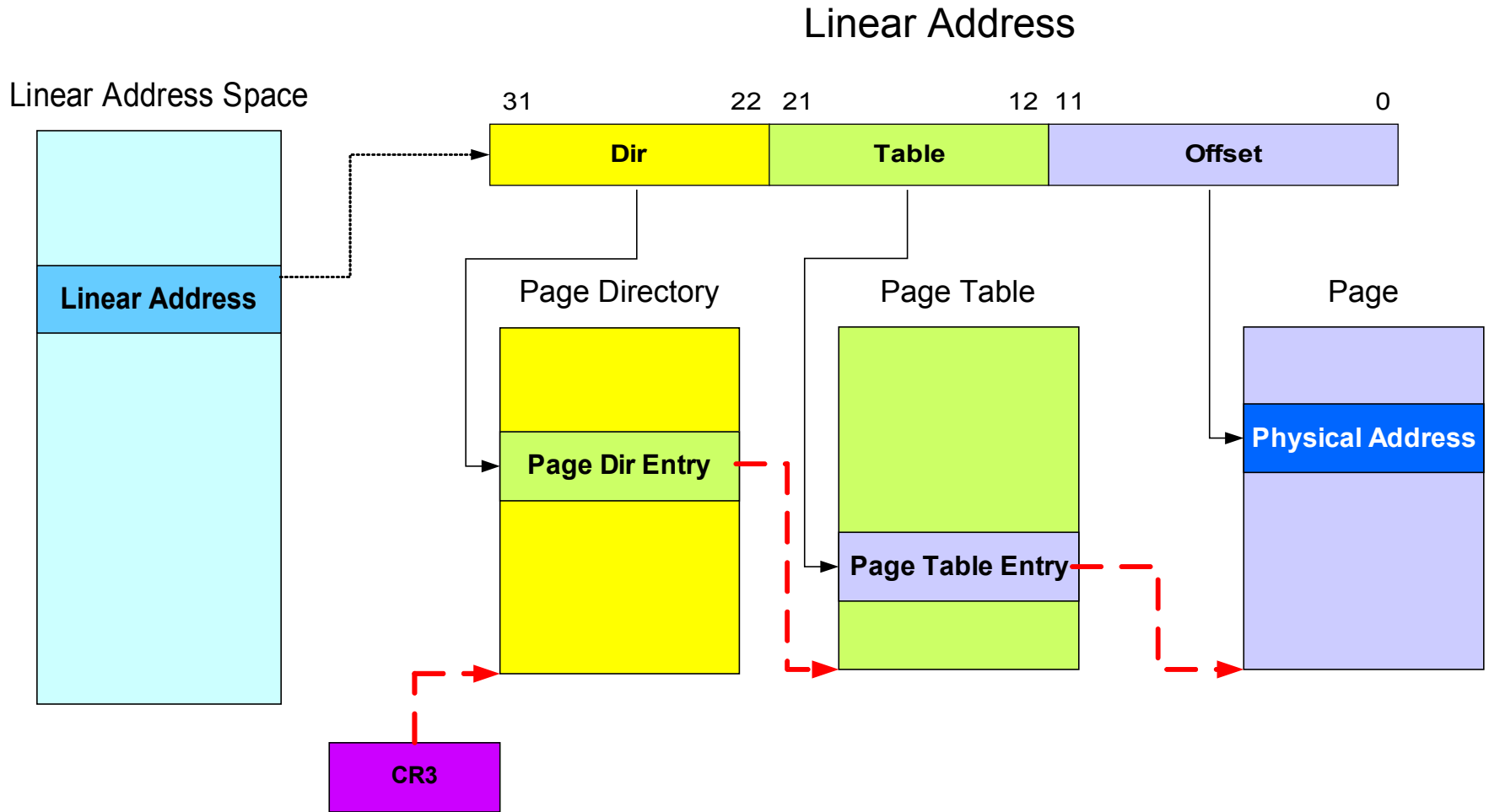
- Protected mode (386 and up)
- Paging enabled/disabled

## ● 64 bit modes are not shown here



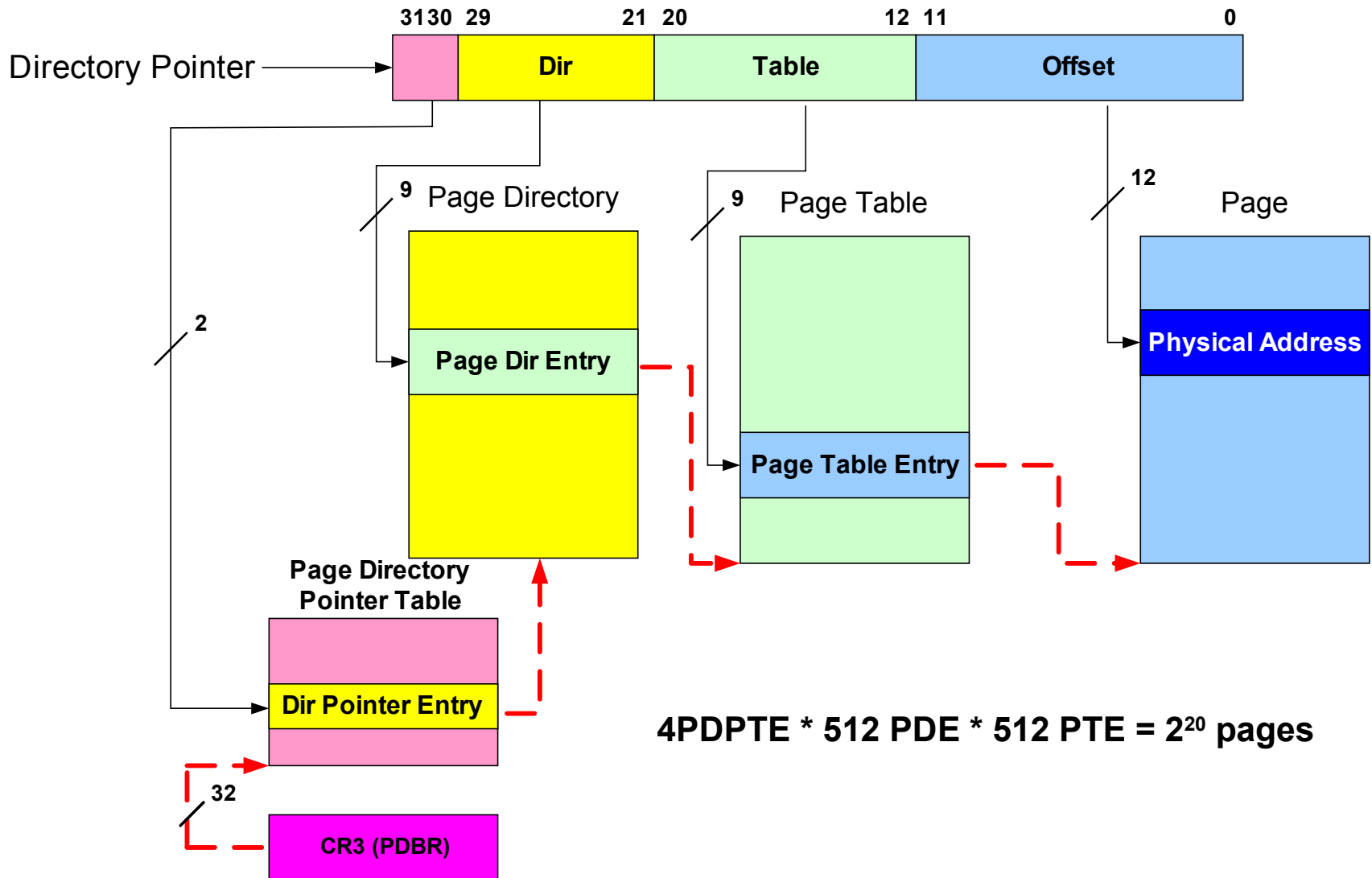
# Page Translation Mechanism (4KB)

## CR4.PAE=0

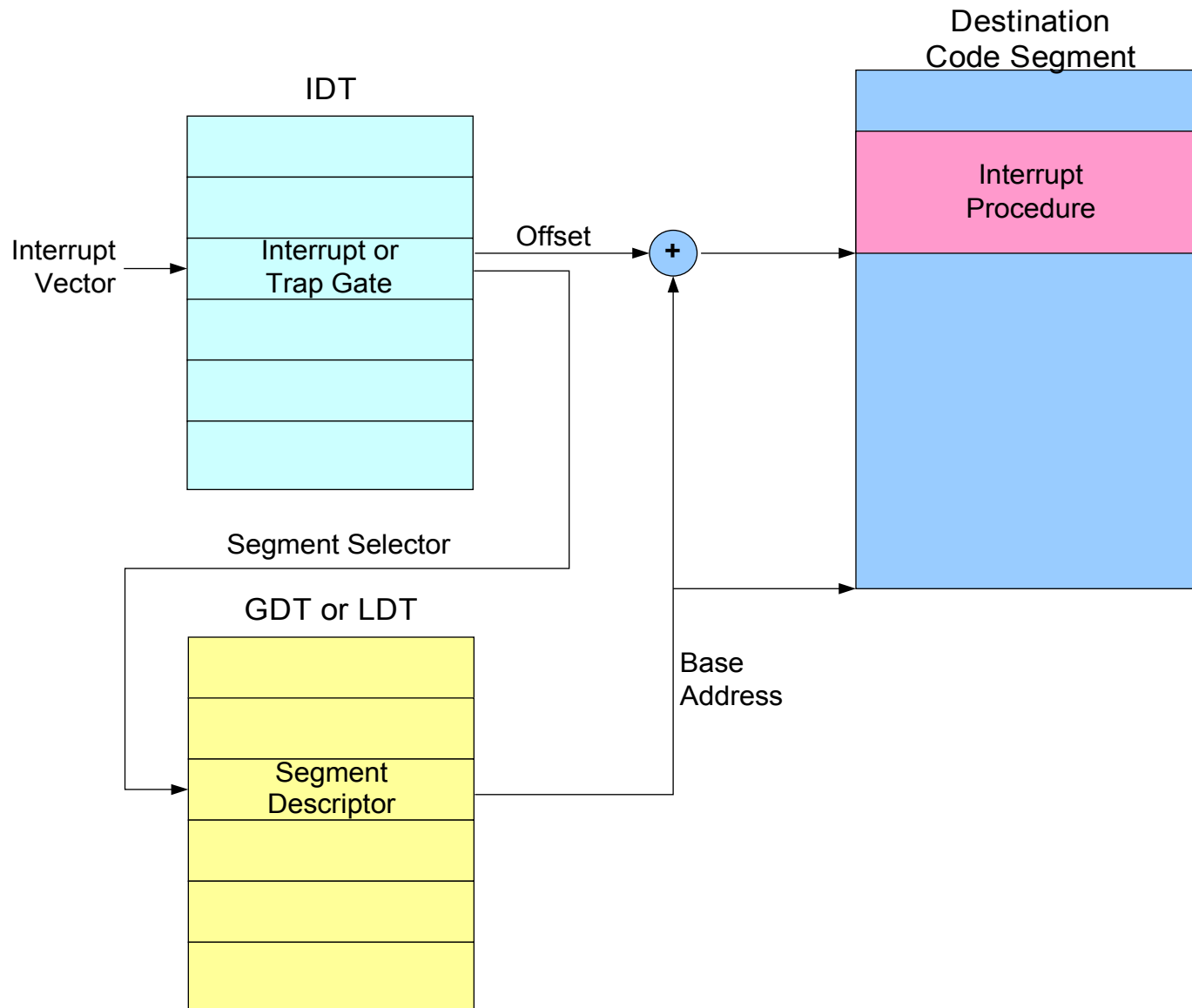


# Page Translation Mechanism (4KB) with Physical Address Extension CR4.PAE=1

Linear Address



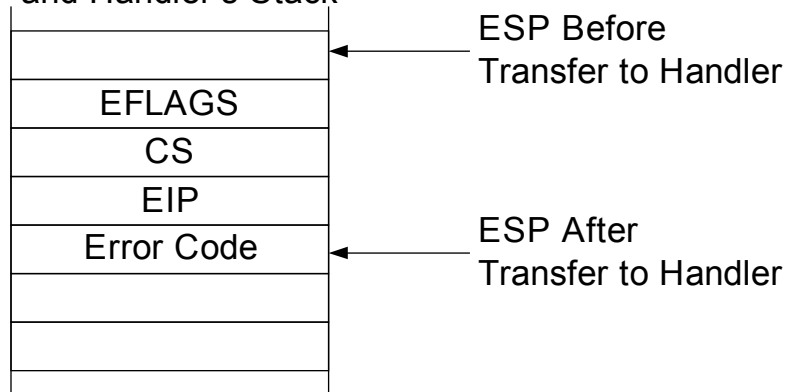
# Interrupt Procedure Call



# Stack Usage on Interrupt Transfer

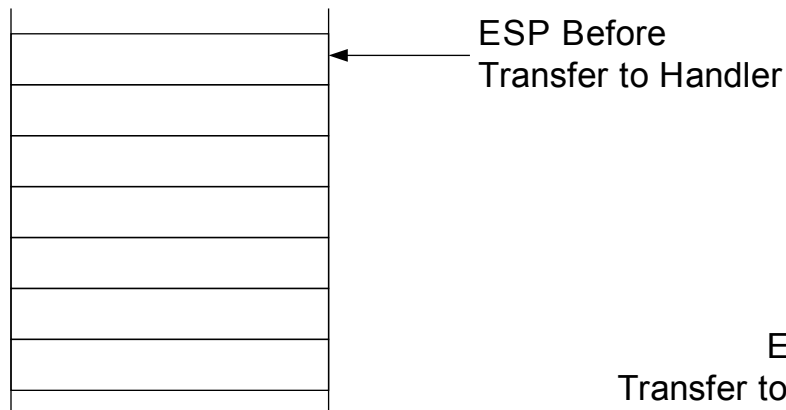
## Stack Usage with No Privilege-Level Change

Interrupted Procedure's  
and Handler's Stack

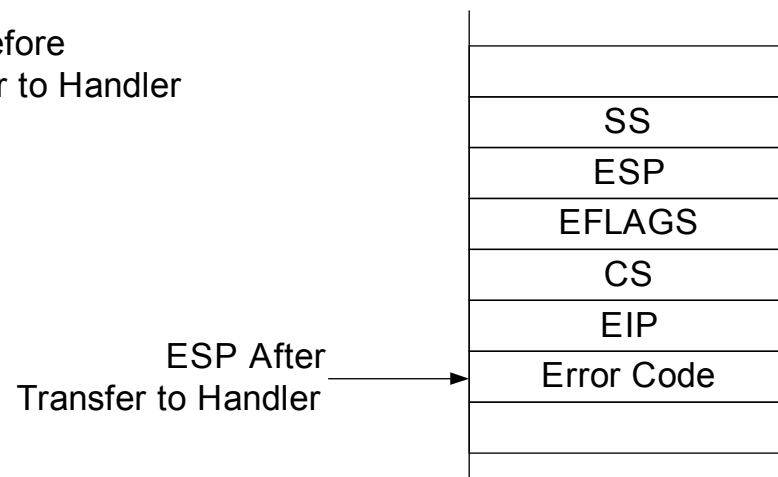


## Stack Usage with Privilege-Level Change

Interrupted Procedure's  
Stack



Handler's Stack



# Extended Memory 64-Bit Technology (EM64T)

# Introduction

## ● What is EM64T (Extended Memory 64Bit Technology) ?

- ◆ Native 64-bit support
- ◆ Similar to the previous extension from 16-bit to 32-bit

## ● Benefits of the architecture

- ◆ Extended memory Addressing

Programs can easily cross the 4GB virtual memory limit.

- ◆ Bringing 64-bit Computing into the Mainstream

Low cost 64-bit machines

- ◆ 32-bit Compatibility

32-bit and 64-bit applications coexisting under a 64-bit operating system.

No emulation is required for 32-bit applications

Migration can be performed gradually

# Introduction

## ● Benefits of the architecture (Cont')

- ◆ Ease of Porting Software from x86 to EM64T

It is not really new only an extension

- ◆ 32-bit Applications Can Run More Efficiently

32-bit applications can have access to the full 4GB address space.

In a 64-bit operating system the kernel will be moved outside the 4GB range allowing the application to use the entire 32-bit address space for code and data.

- ◆ Generate a more efficient code

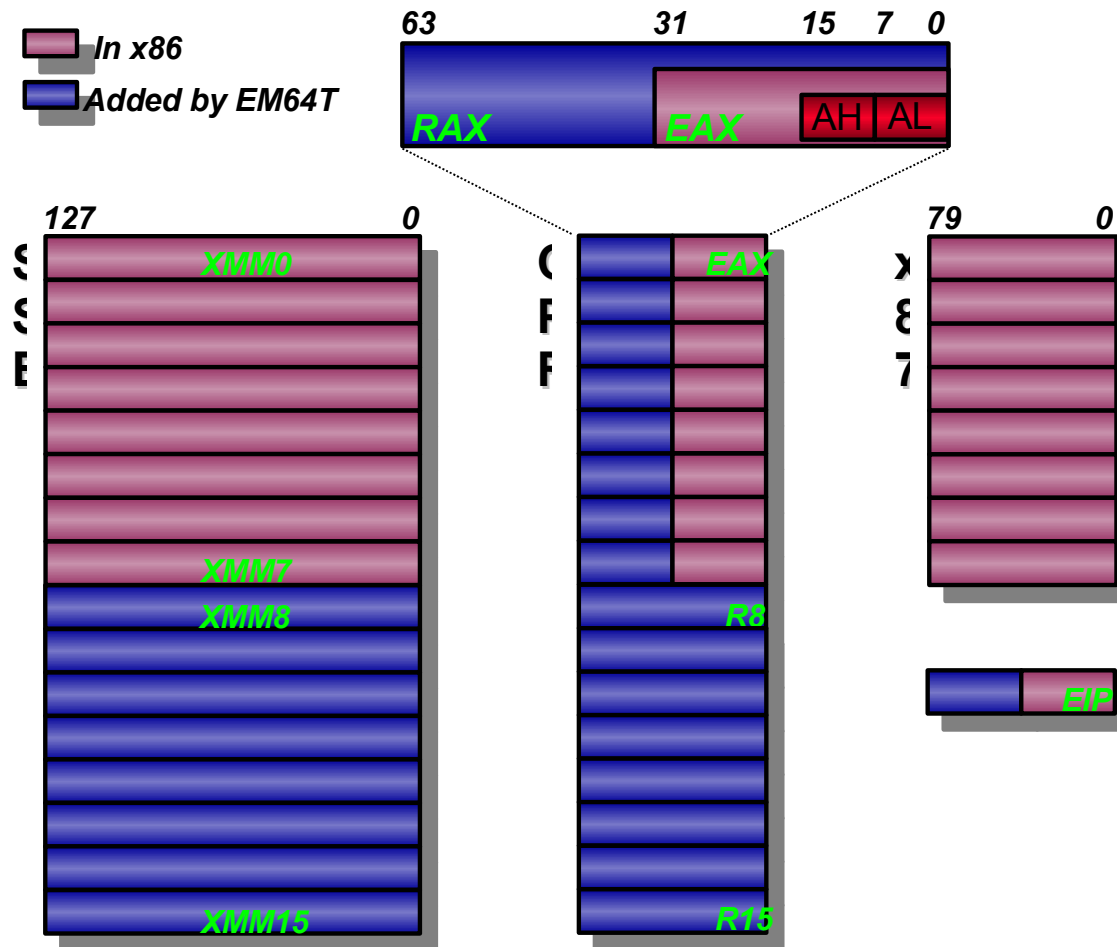
Additional Registers

IP relative addressing mode.

# Architecture Overview



- What are the main new features introduced in EM64T ?
  - ◆ Registers set changes
  - ◆ 64-bit flat linear addressing
  - ◆ New Operating Modes
  - ◆ Instruction Set Changes
  - ◆ Memory Organization.

# General Purpose Register Set Extension



# General Purpose Register Set Extension

- EM64T provides a uniform set of low-byte, low-word, low-doubleword, and quadword registers
  - ◆ Well suited for register allocation by compilers

 *In x86*  
 *Added by EM64T*

| 63         | 31 | 7 | 0   |
|------------|----|---|-----|
| <i>RSI</i> |    |   | SIL |
| <i>RDI</i> |    |   | DIL |
| <i>RSP</i> |    |   | SPL |
| <i>RBP</i> |    |   | BPL |

# System Registers Set Extension

 In x86

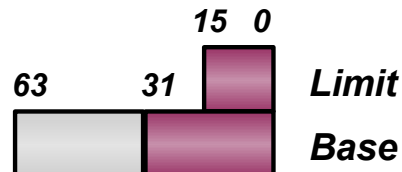
 Added by EM64T

## MSRs

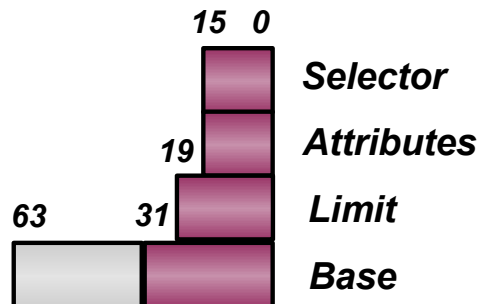


## Descriptors

### GDTR & IDTR



### LDTR & TR



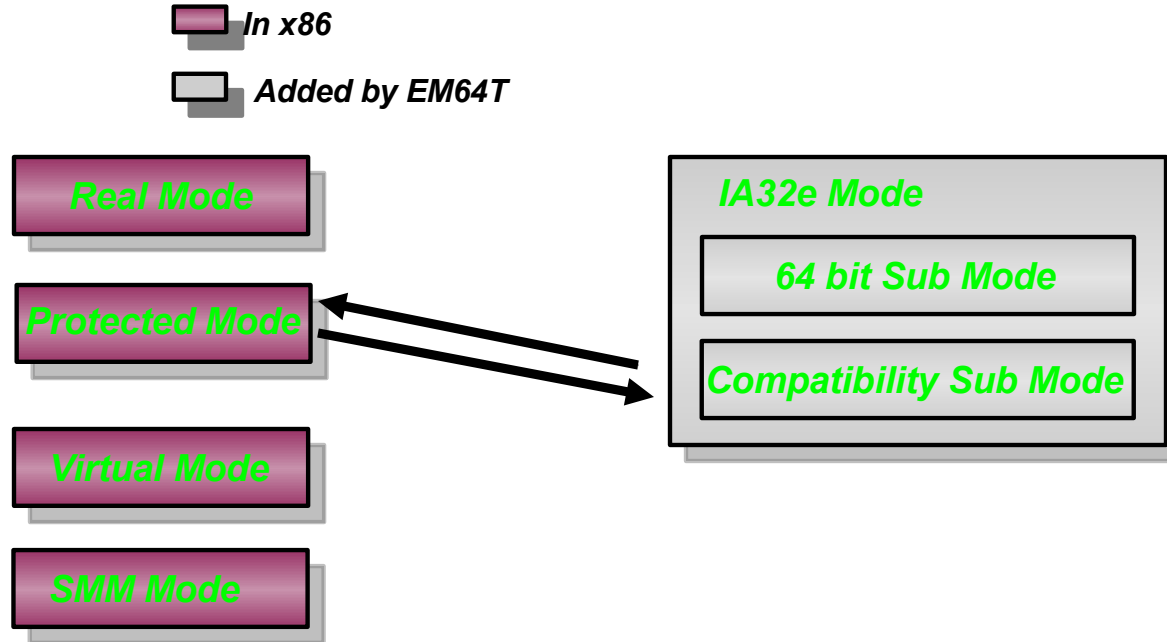
## Control



## Debug



# Operating Modes



- 64-bit and Compatibility sub modes are determined according to the Code Segment Register

# 64-bit Mode

- Used by 64-bit applications under a 64-bit OS
  - ◆ New \ Extended Registers become available.
  - ◆ Extended \ New instructions become available.
  - ◆ Architectural support for up to 64 bits of linear address
  - ◆ Physical address support of up to 52 bits
  - ◆ Can use flat address space with a single code, data and stack space
  - ◆ A new RIP-relative data addressing mode.

# Compatibility Mode

- Permits legacy 16-bit and 32-bit applications to run, without recompilation, under a 64-bit OS.
- The following elements of legacy protected mode are not available when in this mode:
  - ◆ Virtual 8086 mode.
  - ◆ Hardware task management.
  - ◆ From the OS point of view: system data structures, address translations, interrupt and exception handling all use the new 64-bit mechanisms.

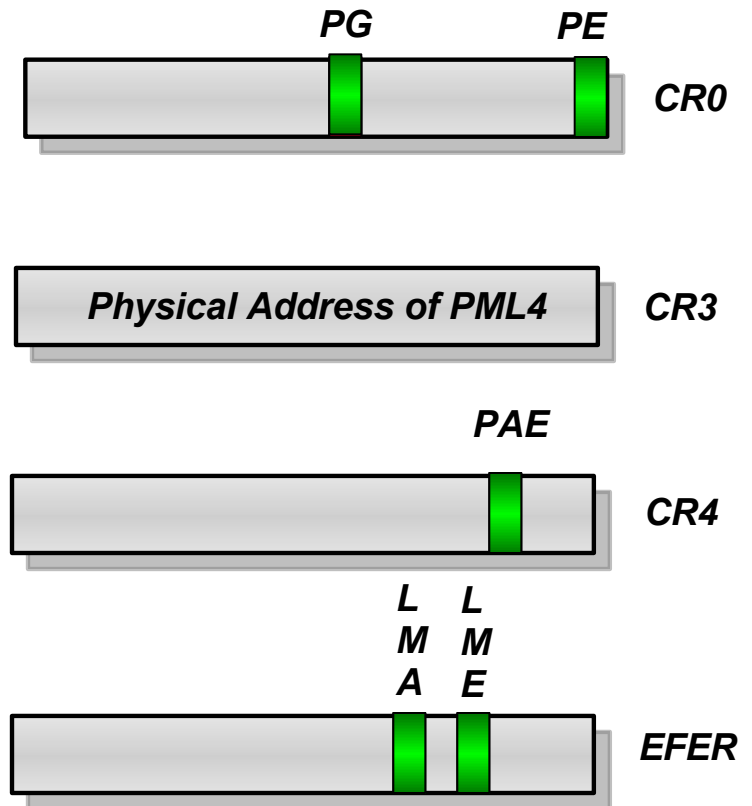
# Determining Sub Modes in IA32e

- CPU mode is determined according to its CS
  - ◆ Default Operation Size bit in the Descriptor (Legacy bit)
  - ◆ Long mode bit in the Descriptor (New bit)

| Mode   |                  | EFER.LMA | CS.L | CS.D |
|--------|------------------|----------|------|------|
| Legacy | Legacy 16        | 0        | N/A  | 0    |
|        | Legacy 32        |          |      | 1    |
| IA32e  | Compatibility 16 | 1        | 0    | 0    |
|        | Compatibility 32 |          | 0    | 1    |
|        | 64-bit           |          | 1    | 0    |

# Activating IA32e mode

- Start from Paging Enabled
- Disable Paging
- Enable physical address extension
- Load CR3 with the physical address of the Level 4 page map table
- Enable IA32e mode by setting EFER.LME bit
- Enable Paging



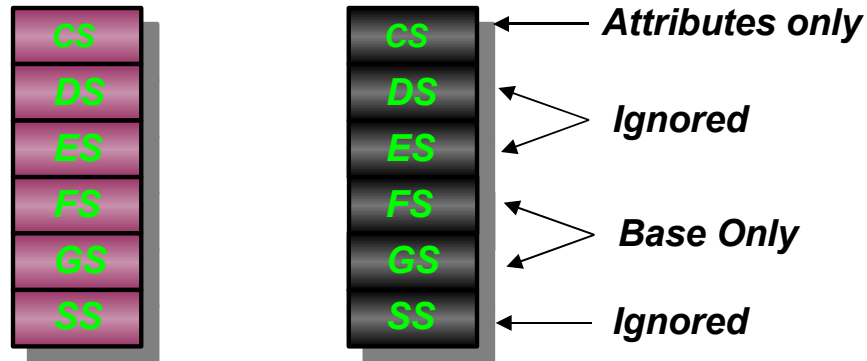
# Segmentation

- In compatibility mode segmentation functions just as it does in legacy 16-bit or 32-bit protected mode semantics.
- In 64-bit mode segmentation is almost completely disabled, creating a flat 64-bit linear address space.
  - ◆ Segment register loads still perform all legacy checks on the values. It is required since it might be a segment that is going to be used by an application in compatibility mode
  - ◆ Code Segment still defines: Operating Mode, DPL

# Segmentation

 *Legacy and Compatibility modes*

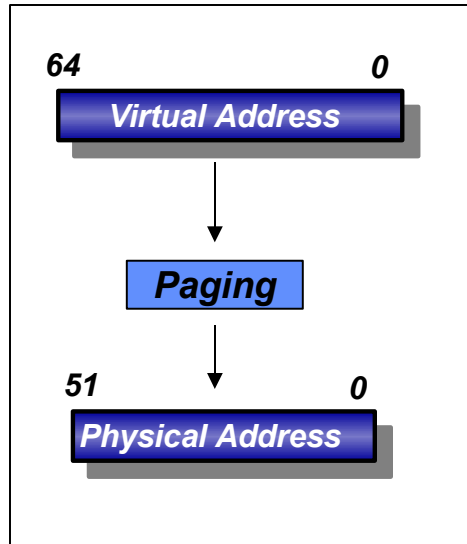
 *64-Bit mode*



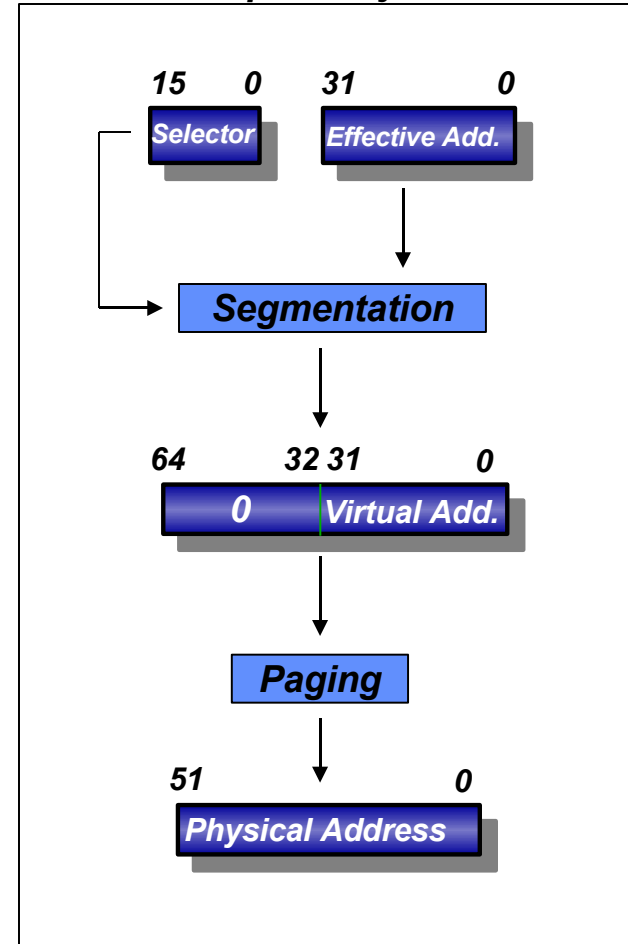
- FS, GS segment bases continue to be used. This facilitates addressing local data and certain OS structures.
- FS and GS base parts of the descriptor are mapped to MSRs (FS\_BASE & GS\_BASE).

# Segmentation

**64 – Bit Mode**



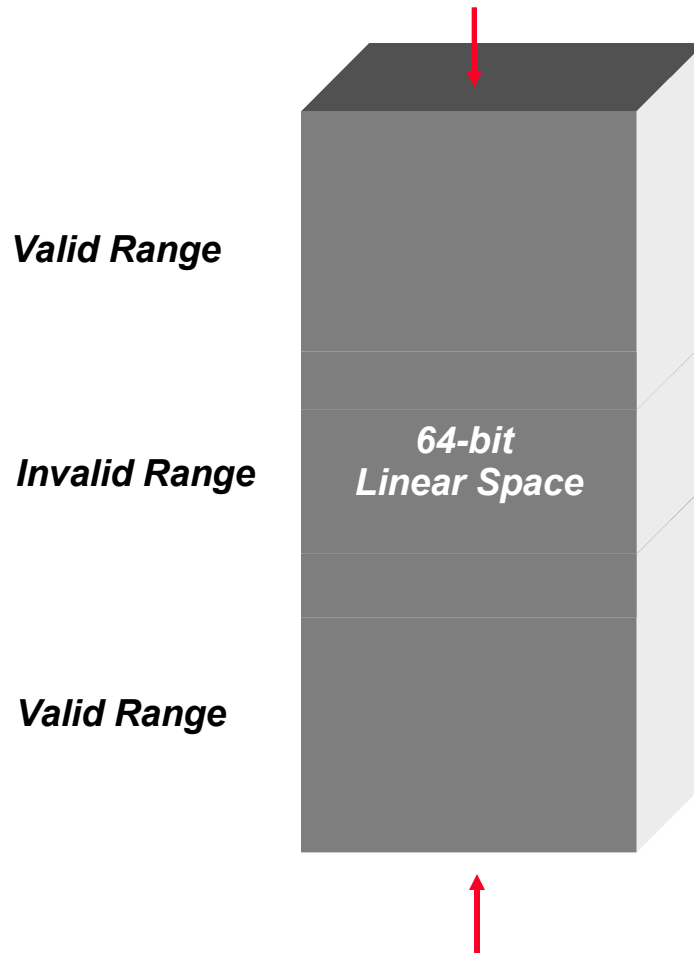
**Compatibility Mode**



# Canonical Addressing

- IA32e mode defines a 64-bits linear address.
  - ◆ Canonical Addressing mechanism allows implementations to support less.
  - ◆ the first implementation supports 48-bits of linear address.
- Definition: *“An address is in canonical form if the address bits from the most significant implemented bit up to bit 63 are set to either all ones or all zeros”*
- Linear address size derives from CPUID values
- Any linear-address reference which is not in canonical form generates an exception.

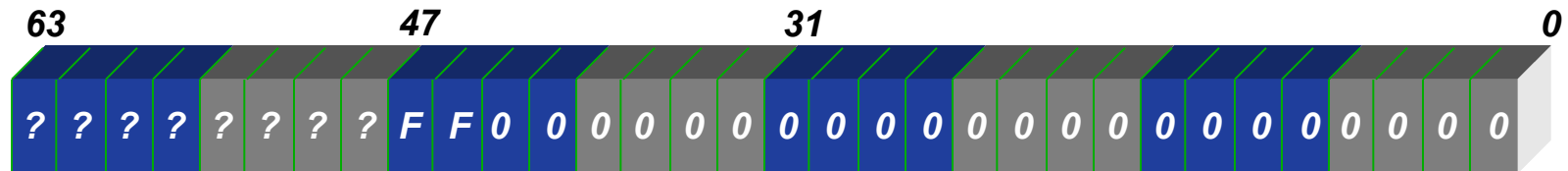
# Canonical Addressing



# Canonical Addressing

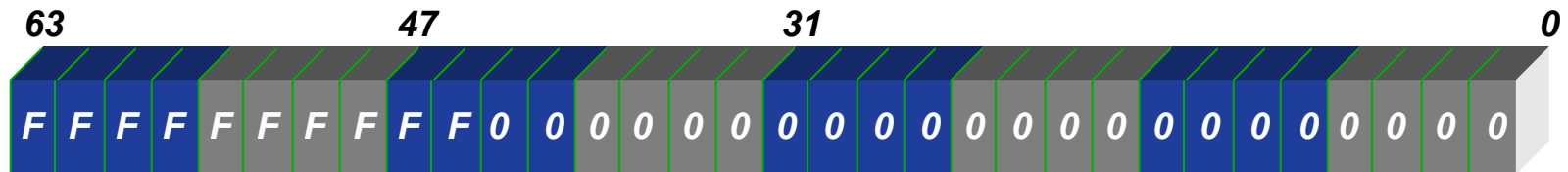
**Example:**

**Canonical address Size = 48**



**Question: What is the canonical form of this address?**

**Answer: Bit 47 is set and therefore the address is:**

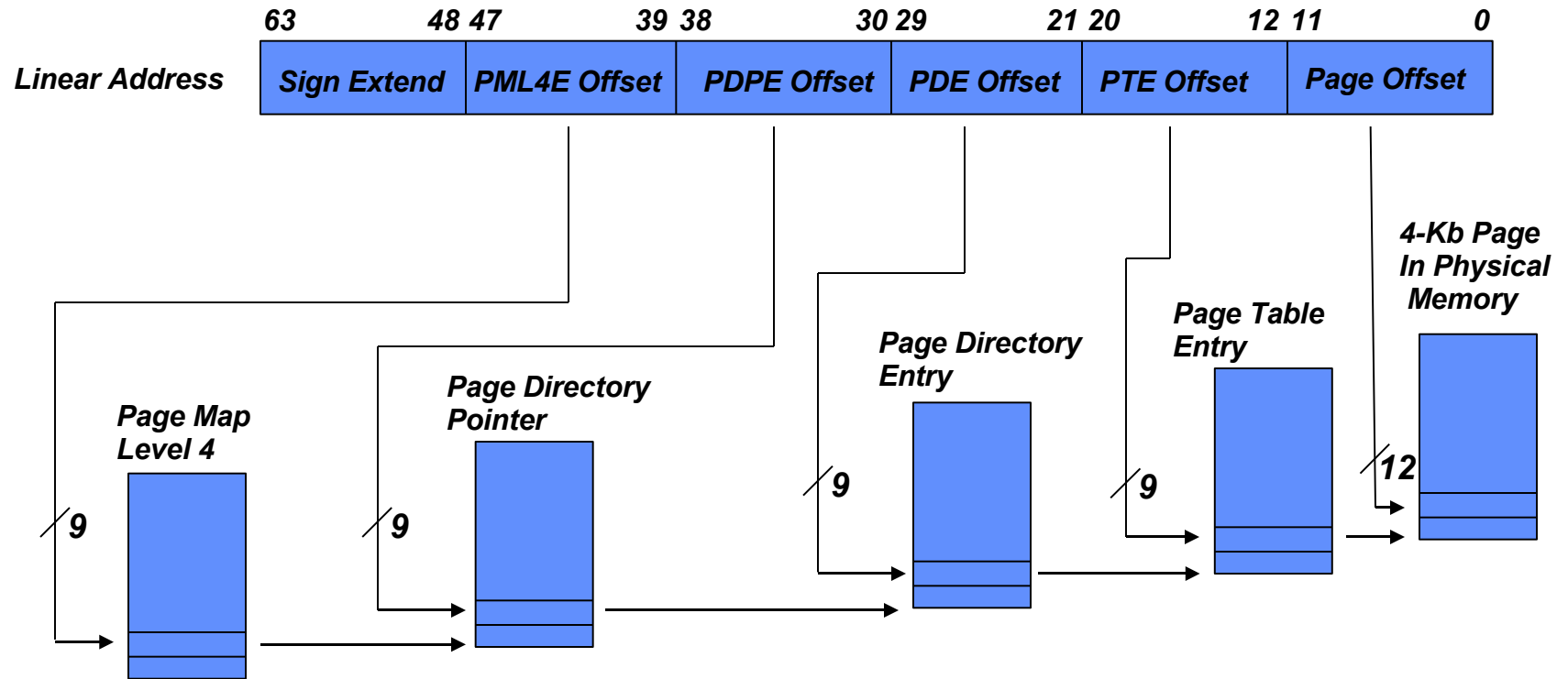


# Paging

- EM64T expands physical address extension (PAE) paging structure to potentially support mapping a 64-bit linear address into a 52-bit physical address.
  - ◆ The first implementation support translation of 48-bit linear address into 40-bit physical address.
  - ◆ Physical address size derives from CPUID values.
- Prior to activating IA32e mode CR4.PAE must be set.
  - ◆ PAE extends paging structures to support 64 bits addresses.

# Paging

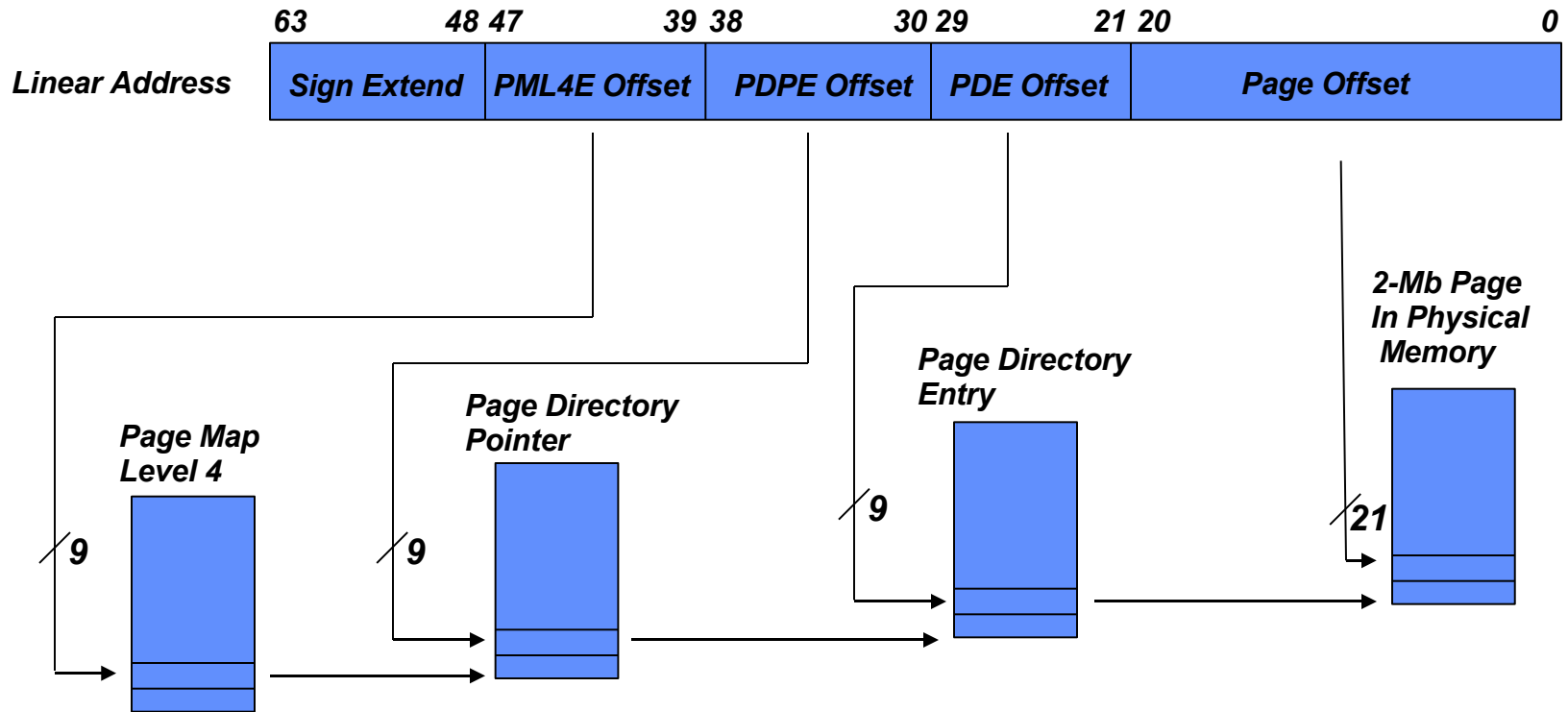
## 4Kb Page Translation



$$512 \text{ PML4E} * 512 \text{ PDPE} * 512 \text{ PDE} * 512 \text{ PTE} = 2^{36} \text{ 4-Kb pages}$$

# Paging

## 2Mb Page Translation



$$512 \text{ PML4E} * 512 \text{ PDPE} * 512 \text{ PDE} = 2^{27} \text{ 2-Mb pages}$$

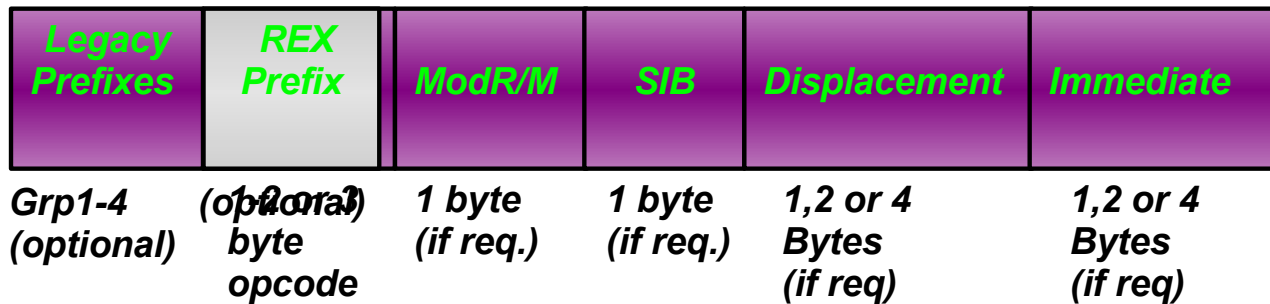
# Instruction Set Changes

- REX – a New family of instruction prefixes used in 64-bit mode.
  - Specify the new GPRs and SSE registers
  - Specify a 64-bit operand size
  - Specify extended control registers
- Not all instructions require a REX prefix
- An instruction can have only one REX prefix
- The legacy instruction size limit of 15 bytes still applies to instructions contain a REX prefix

# Instruction Set Changes

- This is how the REX prefix fits within the byte order of instructions

*Location is critical*



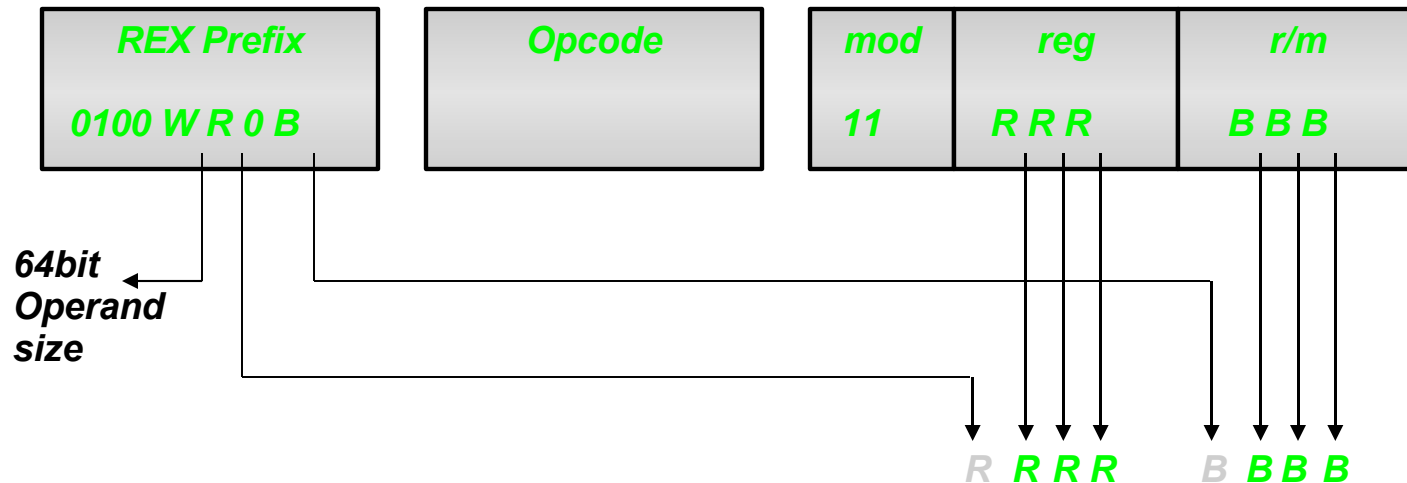
# Instruction Set Changes

| Field Name | Bit Position | Definition                                                           |
|------------|--------------|----------------------------------------------------------------------|
| -          | 7:4          | 0100                                                                 |
| W          | 3            | 0 = Default Operand size<br>1= 64 Bit Operand size                   |
| R          | 2            | Extension of the ModR/M reg field                                    |
| X          | 1            | Extension of the SIB index field                                     |
| B          | 0            | Extension of the ModR/M field,<br>SIB base field or Opcode reg field |

# Instruction Set Changes

- Let's get the feeling of how and why REX prefix is used.

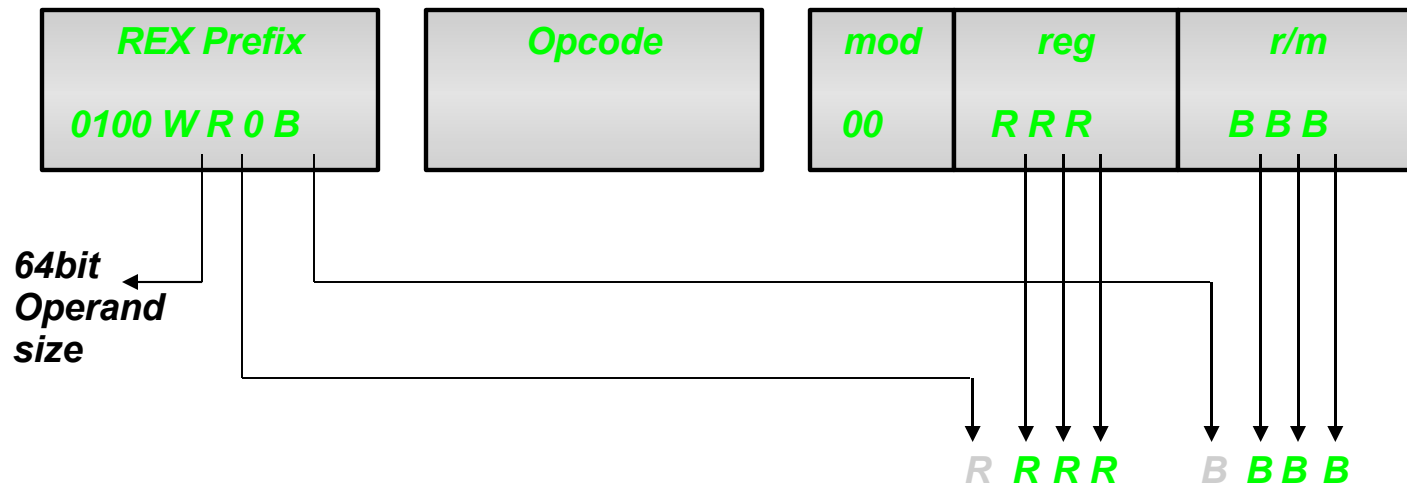
***mov r8, r9    // Register–Register addressing (no memory)***



***In this Case the instruction bytes would be: 0x4d 0x8b 0xc1***

# Instruction Set Changes

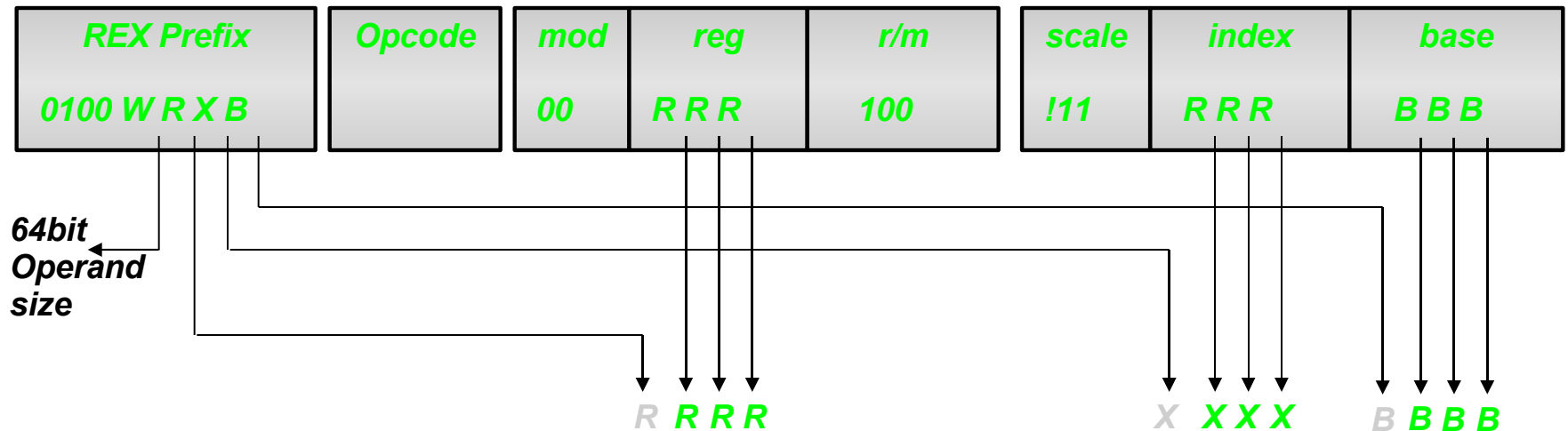
***mov r8, [r9] // Register–Memory addressing (no SIB byte)***



***In this Case the instruction bytes would be: 0x4d 0x8b 0x1***

# Instruction Set Changes

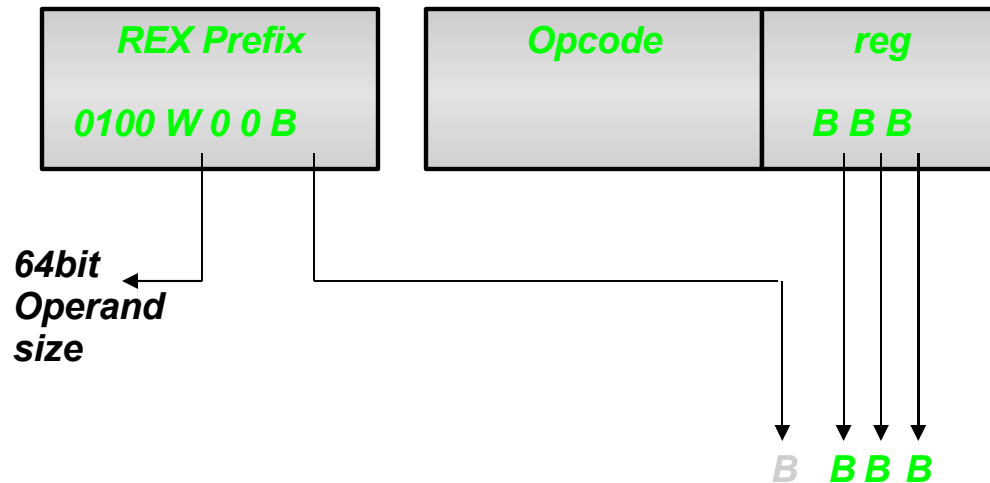
***mov r8, [r9 + r10\*2] // Memory addressing with SIB byte***



***In this Case the instruction bytes would be: 0x4f 0x8b 0x04 0x51***

# Instruction Set Changes

*push r8    // Register operand coded in opcode byte*



*In this Case the instruction bytes would be: **0x49 0x50***

# Instruction Set Changes

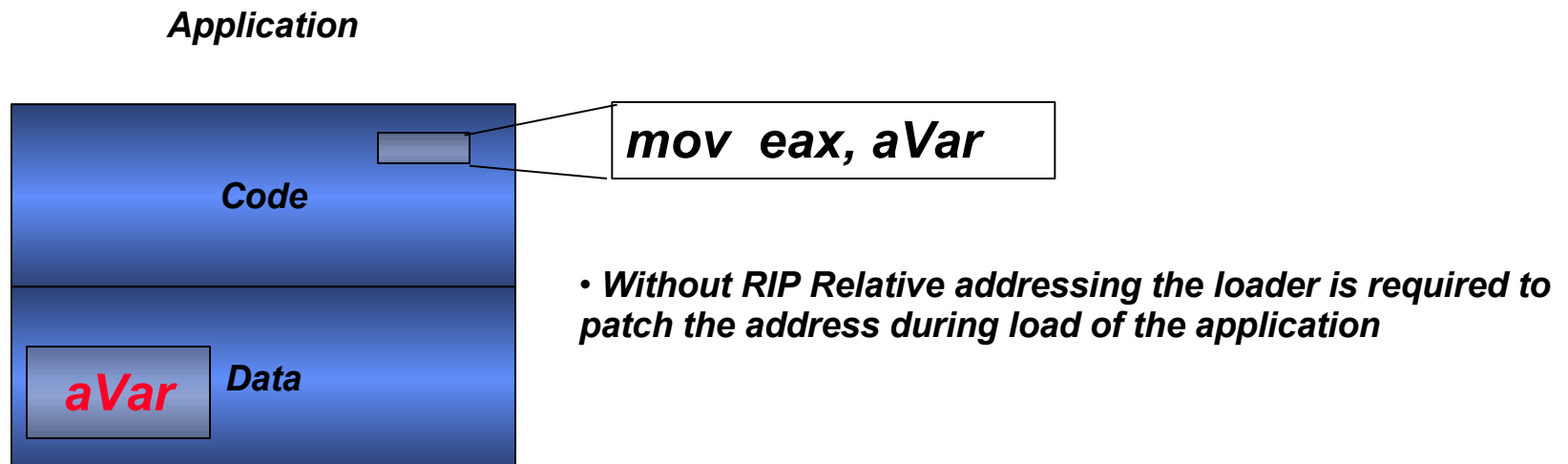
## *Default Address & Operand Size*

| Mode   |               | Default Address Size | Default Operand Size | Register Extension | GPR Width |
|--------|---------------|----------------------|----------------------|--------------------|-----------|
| IA-32e | 64-bit        | 64                   | 32                   | Yes                | 64        |
|        | Compatibility | 32                   | 32                   | No                 | 32        |
|        |               | 16                   | 16                   |                    | 16, 8     |

# Instruction Set Changes

## ● RIP – Relative Addressing

- A new addressing form implemented in 64 bit mode
- Using a signed 32-bit displacement an offset range of  $\pm 2$  GB from the RIP is provided.
- In legacy IA32, addressing relative to the IP is available only with control transfer instructions
- Allows to generate position independent code.



# Instruction Set Changes

## ● Let's assume:

- ◆ CPU in 64-bit mode
- ◆ RAX = 0002\_0001\_8000\_2201
- ◆ RBX = 0002\_0002\_0123\_3301

## ● **Example 1:** 64-bit Add: ADD RBX, RAX

- ◆ RBX = 0004\_0003\_8123\_5502

## ● **Example 2:** 32-bit Add: ADD EBX, EAX

- ◆ RBX = 0000\_0000\_8123\_5502 (32 bit result is zero extended)

## ● **Example 3:** 16-bit Add: ADD BX, AX

- ◆ RBX = 0002\_0002\_0123\_5502 (bits 63:16 are preserved)

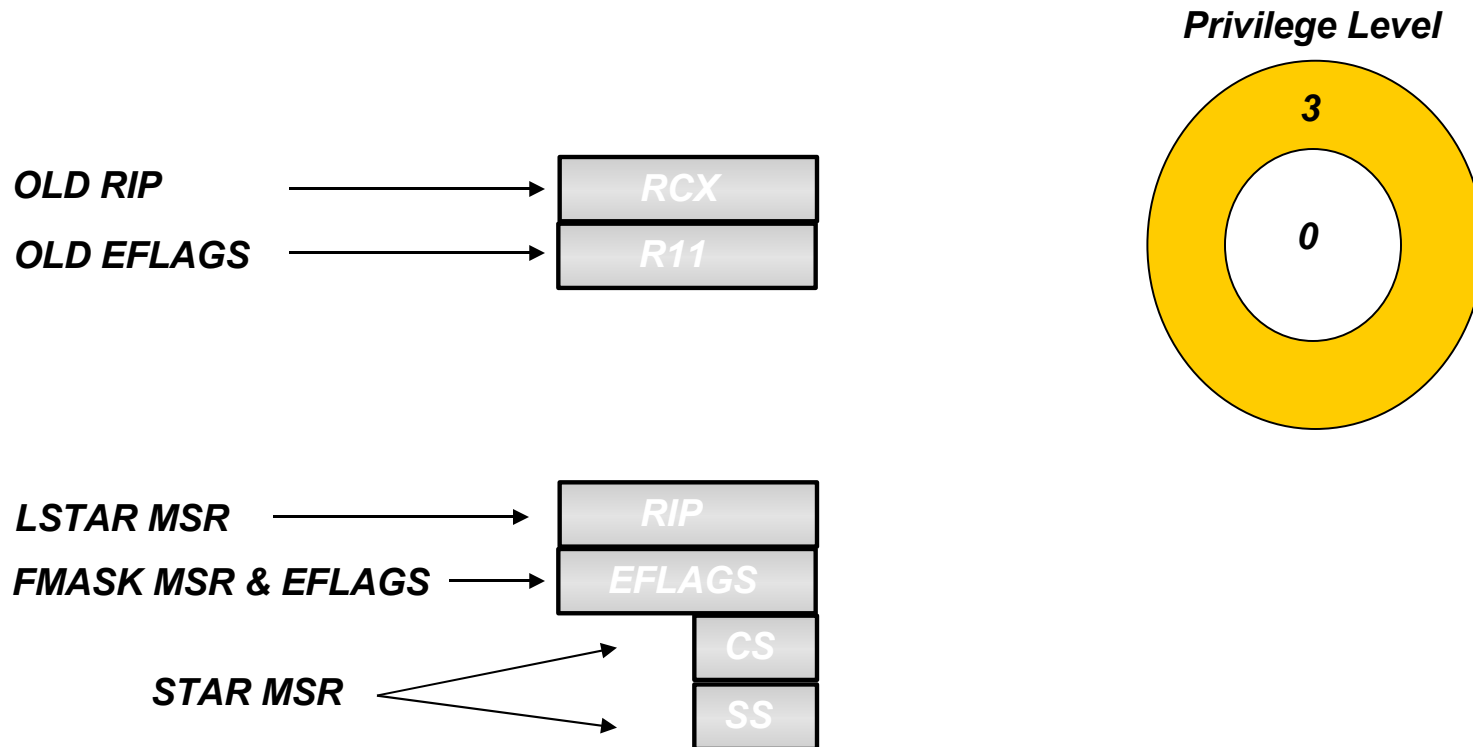
## ● **Example 4:** 8-bit Add: ADD BL, AL

- ◆ RBX = 0002\_0002\_0123\_3302 (bits 63:08 are preserved)

# **SYSCALL \ SYSRET instructions**

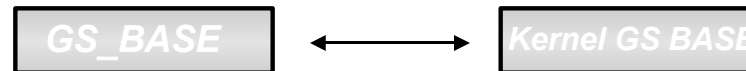
- New Instructions available only in 64-Bit mode
- Provides a fast call to privilege level 0 system procedure.
  - ◆ Relevant to OS's that use a flat memory model with no segmentation. (Windows and Linux)
  - ◆ The simplification comes from eliminating unneeded checks and memory references
- A similar mechanism to SYSENTER \ SYSEXIT instructions

# SYSCALL instruction



# SWAPGS

- New Instruction available only in 64-Bit mode
- Provides a fast method for system software to load a pointer to system data structures.
- The need for such an instruction arises when using SYSCALL to implement system calls.
  - ◆ No kernel stack exists at the OS entry point



# Interrupt Handler in Linux – Example

**common\_interrupt:**

```
 cld
 subq $9*8, %rsp //
 movq %rdi, 8*8(%rsp) //
 movq %rsi, 7*8(%rsp) // Save
 movq %rdx, 6*8(%rsp) // General
 movq %rcx, 5*8(%rsp) // Purpose
 movq %rax, 4*8(%rsp) // Registers
 movq %r8, 3*8(%rsp) //
 movq %r9, 2*8(%rsp) //
 movq %r10, 1*8(%rsp) //
 movq %r11, (%rsp) //
 leaq 9*8(%rsp), %rdi
 testl $3, 8(%rdi) // Check CPL
 je 1f
 swapgs // Swap user & kernel GS
1: addl $1, %gs:pda_irqcount
 call do_IRQ // Call specific handler
```

# Interrupt Handler in Linux – Example

```
cli
subl $1, %gs:pda_irqcount
leaq -9*8(%rdi),%rsp
testl $3, 8(%rdi) // Check CPL
je 2f
swapgs // Swap kernel & user GS
2: movq (%rsp), %r11 //
 movq 1*8(%rsp), %r10 //
 movq 2*8(%rsp), %r9 //
 movq 3*8(%rsp), %r8 // Restore
 movq 4*8(%rsp), %rax // General
 movq 5*8(%rsp), %rcx // Purpose
 movq 6*8(%rsp), %rdx // Registers
 movq 7*8(%rsp), %rsi //
 movq 8*8(%rsp), %rdi //
 addq $9*8, %rsp //
 iretq // Return from interrupt
```

# Other new Instructions

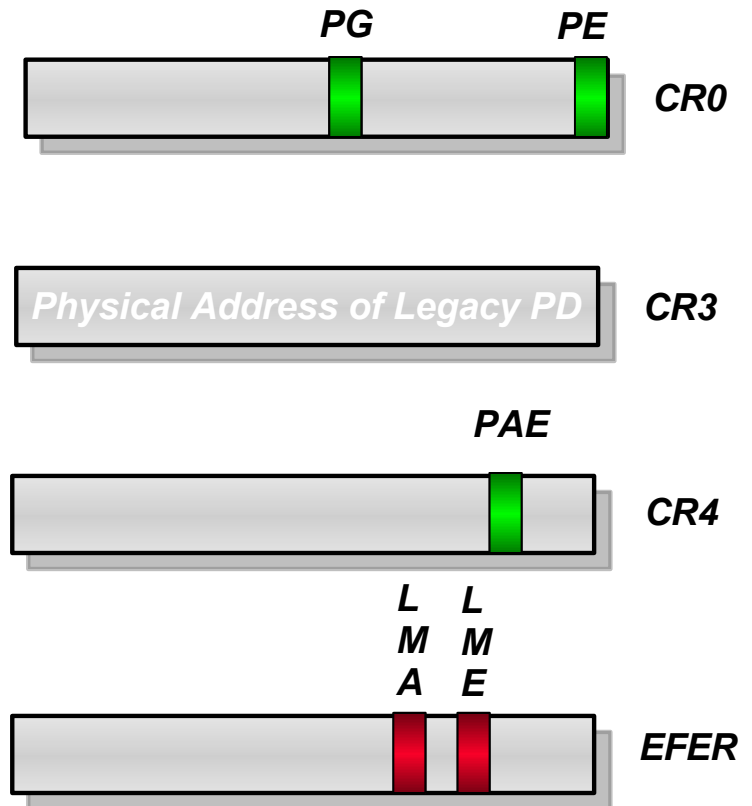
- Many instructions were promoted to 64-Bits such as:
  - ◆ Arithmetic Calculations – ADD, SUB, DIV, MUL
  - ◆ Boolean Bitwise Operations – NOT, OR, XOR, AND
  - ◆ Rotation Instruction – SHL, SHR, SHLD, SHRD
- SSE instructions can access XMM8-XMM15

# Invalid Instructions

- The following instructions are invalid in 64-bit mode:
  - ◆ AAA, AAD, AAM, AAS, DAA, DAS
  - ◆ BOUND – Check Array Bounds
  - ◆ JMP, CALL (far absolute)
  - ◆ INTO
  - ◆ LDS, LES
  - ◆ POP DS, POP ES, POP SS
  - ◆ POPA, POPAD, PUSHA, PUSHAD
  - ◆ PUSH CS,DS,ES,SS
  - ◆ DEC, INC (one byte opcode only) – becomes a REX prefix.

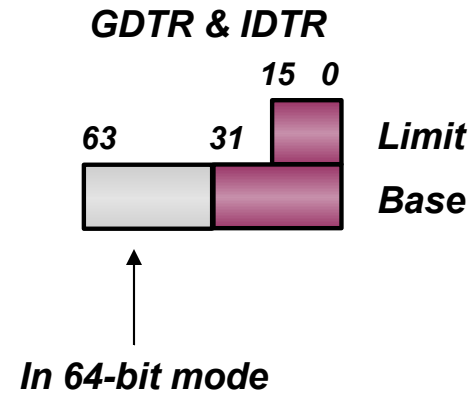
# Deactivating IA32e mode

- Start from Compatibility sub mode
- Disable Paging
- Load CR3 with the physical address of the legacy page table directory base address
- Disable IA32e mode by setting EFER.LME bit
- Enable Paging

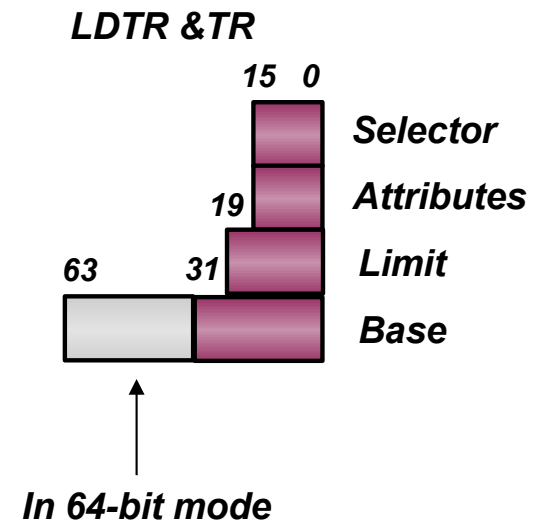


# System Descriptors

● Loading the GDT and IDT :



● Loading the LDT and TR :



# Instruction Set Changes

## *Address Size Prefixes*

| Mode   |               | Default Address Size | Effective Address Size | Address Prefix Required |
|--------|---------------|----------------------|------------------------|-------------------------|
| IA-32e | 64-bit        | 64                   | 32                     | Yes                     |
|        | Compatibility | 32                   | 32                     | No                      |
|        |               | 16                   | 16                     |                         |

# Instruction Set Changes

## ● Displacement in 64 bit mode

- Remain 8 bits or 32 bits and are sign extended to 64 bits

## ● Direct memory-offset MOVs

- extended to specify a 64-bit memory offset immediate absolute address. (called *moffset*)

## ● Immediates in 64 bit mode

- By default remain 32 bits
- When using 64 bits operand size the processor sign-extends all immediates to 64-bits prior to their use.
- Support for 64-bits immediate is accomplished by extending *MOV reg,imm16/32*

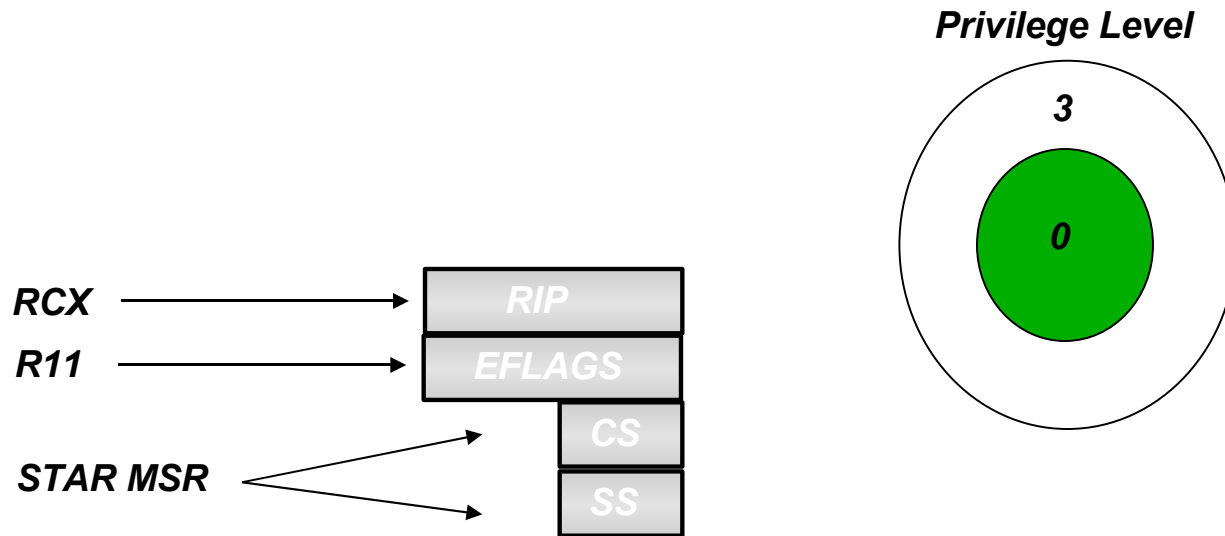
## ● Branches

- Near branch operand size if forced to 64-bit (No need for REX)
- Call Gates are extended to support 64-bit offset

# Instruction Set Changes

- When performing 32-bit operations with a GPR destination in 64-Bit mode, the processor zero extends the 32-bit result into the full 64-bit destination.
- 8-bit and 16-bit operations on GPR's preserve all unwritten upper bits of the destination GPR.
- Software should explicitly sign extend the result of 8-bit, 16-bit and 32 bit operations to the full 64-bit width before using the result in 64-bit address calculations.
- After a processor mode change the high 32-Bits of the 64-bit GPR's becomes undefined.

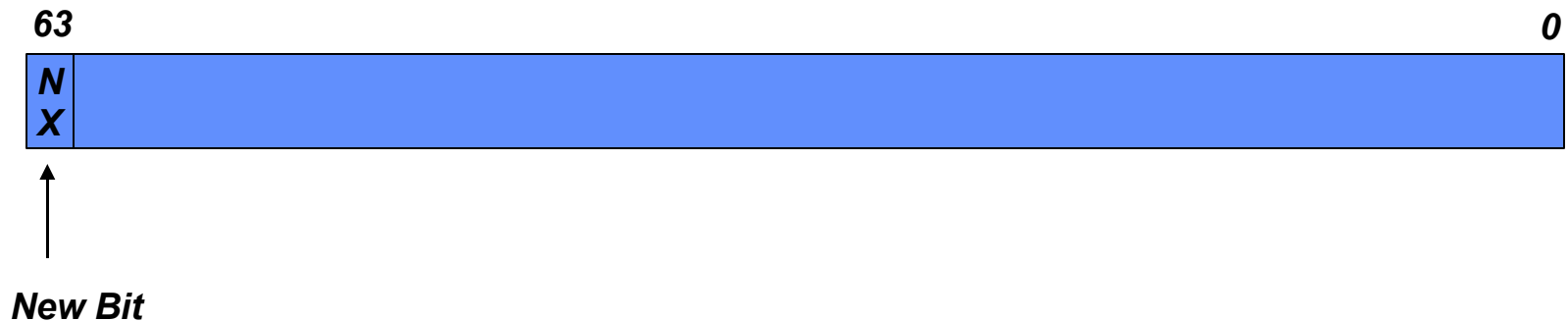
# SYSRET instruction



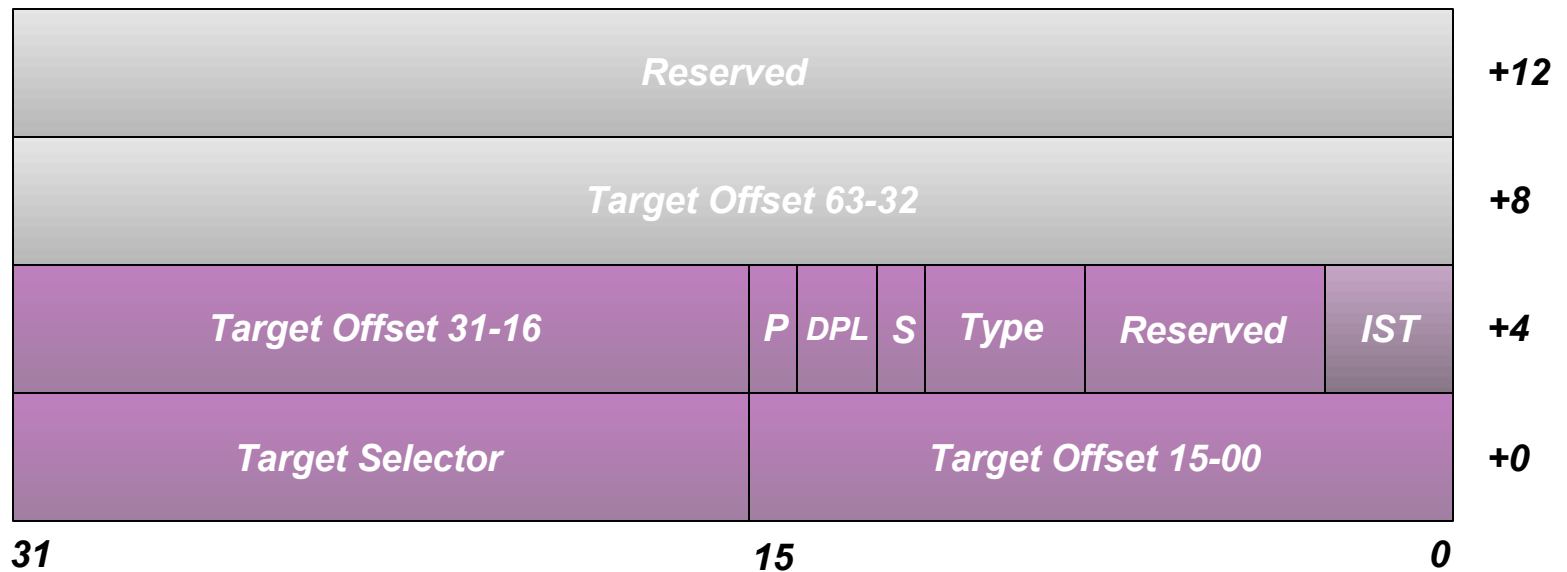
# No Execute (NX) Page Protection

- A new paging protection that prevents data pages from being used by malicious software to execute code.
- Available only in PAE mode (Legacy and IA32e mode)
- Can be activated \ deactivated

*Page Table Entry*



# Interrupt and Trap Gates



# Floating Point

# Floating Point

- Supports *single*, *double* and *extended* memory format (32, 64 and 80 bits), as well as other exotic formats (BCD, integers)
- 8 registers arranged as a stack
- A stack entry (register) is always 80 bits; conversion done during load/store only
- Each operation works on top of stack and optionally on another element (another stack entry of memory). Stack top moves!
- Supports all IEEE-754 operations
  - ◆ fadd, fsub, fmul, fdiv, fsqrt, fcmp...
- and many others
  - trig, transcendentals, etc...
- Separate unit (“co-processor”) for 8086, 286, 386. On-chip since i486
  - ◆ Exception model reflects this history
- Big performance boost in i486, large in Pentium, huge in Pentium-II

# Floating Point Formats

● General Format:  $(-1)^S 2^E (b_0 \Delta b_1 b_2 b_3 \dots b_{p-1})$

◆ Where:

$S = 0 \text{ or } 1$

$E = \text{any integer between } E_{\min} \text{ and } E_{\max}, \text{ inclusive}$

$b_i = 0 \text{ or } 1$

$p = \text{number of bits of precision}$

● Specifics

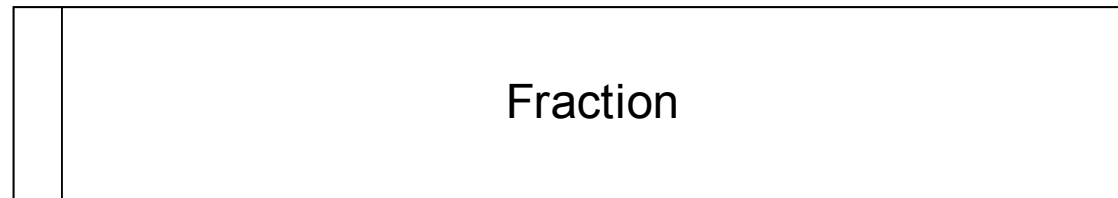
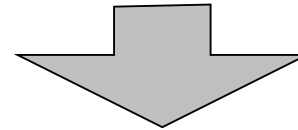
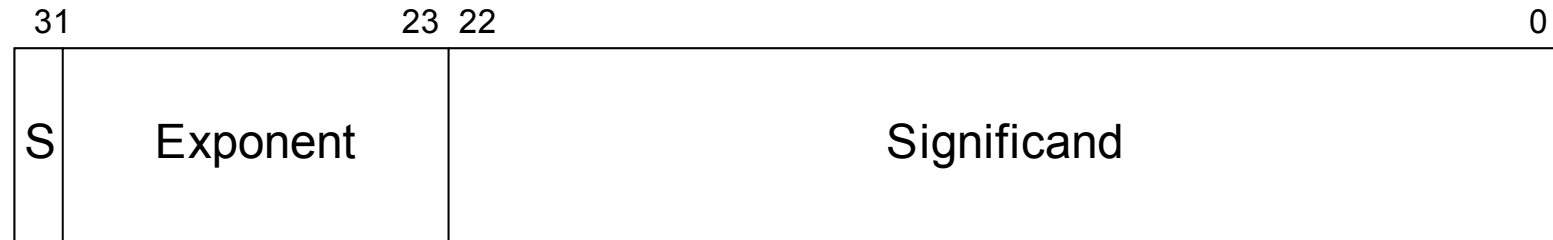
| Parameter               | Single | Double | Extended |                |
|-------------------------|--------|--------|----------|----------------|
| Format width in bits    | 32     | 64     | 80       |                |
| $p$ (bits of precision) | 24     | 53     | 64       | 2's complement |
| Exponent width in bits  | 8      | 11     | 15       | Biased format  |
| $E_{\max}$              | +127   | +1023  | +16383   |                |
| $E_{\min}$              | -126   | -1022  | -16382   |                |
| Exponent bias           | +127   | +1023  | +16383   |                |

● Biased Exponent means that a constant is added to the actual exponent so that the biased exponent is always a positive number.

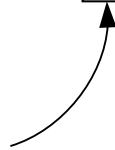
●  $b_0$  is implied in single/double, explicit in extended

# Floating Point Formats

- Example of single precision number:



Integer or J-Bit



# Normalized and De-Normalized Numbers

## ● Normalized Numbers

- ◆ Except for zero, the significand is always made up of an integer of 1 and the following fraction

1.fff..ff

For values less than 1, the leading zeros are eliminated

For each leading zero eliminated, the exponent is decremented by one

Example

$$0.125_{10} = 0.001_2 = 1.0 E_2-3$$

## ● De-Normalized (Tiny) Numbers

- ◆ When the biased exponent is zero, smaller numbers can only be represented by making the integer bit of the significand zero
- ◆ A denormalized number is computed through a technique called gradual underflow

# De-Normalization Process Example

| Operation       | Sign | Exponent<br>(unbiased) | Significand       |
|-----------------|------|------------------------|-------------------|
| True Result     | 0    | -129                   | 1.01011100000..00 |
| Denormalize     | 0    | -128                   | 0.10101110000..00 |
| Denormalize     | 0    | -127                   | 0.01010111000..00 |
| Denormalize     | 0    | -126                   | 0.00101011100..00 |
| Denormal Result | 0    | -126                   | 0.00101011100..00 |

# Real Number Notation

| Notation                               | Value                     |                 |                                          |
|----------------------------------------|---------------------------|-----------------|------------------------------------------|
| Ordinary Decimal                       | 178.125                   |                 |                                          |
| Scientific Decimal                     | $1.78125E_{10}2$          |                 |                                          |
| Ordinary Binary                        | 10110010.001              |                 |                                          |
| Scientific Binary                      | $1.0110010001E_2111$      |                 |                                          |
| Scientific Binary<br>(Biased Exponent) | $1.0110010001E_210000110$ |                 |                                          |
| Single-Real Format                     | Sign                      | Biased Exponent | Normalized Significant                   |
|                                        | 0                         | 10000110        | 011001000100000000000000<br>1. (Implied) |

# Special Numbers

## ● Signed Zeros

- ◆ Exponent and significant are 0
- ◆ Sign: +0 or −0

Both are equal in value

The sign result depends on the operation being performed and the rounding mode being used

## ● Signed Infinities

- ◆  $+\infty$  or  $-\infty$
- ◆ Represent the maximum positive and negative real numbers that can be represented in floating-point format
- ◆ Significant is 0
- ◆ Exponent is all 1's
- ◆ Single precision example

$+\infty = 0\ 11111111\ 000000000000000000000000$

$-\infty = 1\ 11111111\ 000000000000000000000000$

- ◆ Special rules for  $0*\text{inf}$ , etc...

# Special Numbers

## ● NaNs – Not a Number

- ◆ Maximum allowed biased exponent
- ◆ Non-zero fraction
- ◆ Sign bit is ignored
- ◆ Quite NaNs – QNaNs

Most significant fraction bit set

Allowed to propagate through most arithmetic operations without signaling an exception

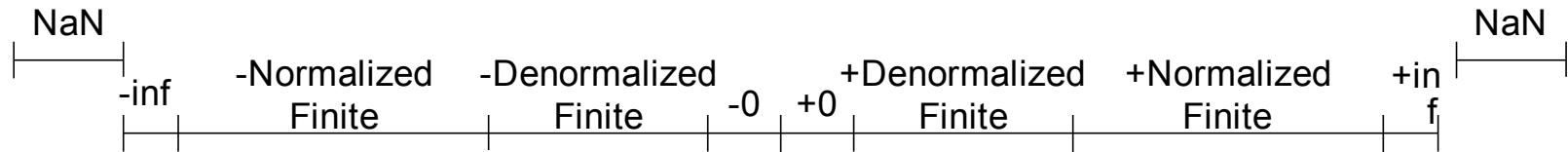
- ◆ Signaling NaNs – SNaNs

Most significant fraction bit clear

Generally signal an invalid operation exception whenever they appear as operands in arithmetic operations.

- ◆ Used to propagate errors in computations when exceptions are masked

# Normalized and De-Normalized Finite Numbers



| S | E | F |
|---|---|---|
| 1 | 0 | 0 |

-0

+0

| S | E | F |
|---|---|---|
| 0 | 0 | 0 |

|   |   |       |
|---|---|-------|
| 1 | 0 | 0.XXX |
|---|---|-------|

-Denormalized Finite

+Denormalized Finite

|   |   |       |
|---|---|-------|
| 0 | 0 | 0.XXX |
|---|---|-------|

|   |         |           |
|---|---------|-----------|
| 1 | 1...254 | Any Value |
|---|---------|-----------|

-Normalized Finite

+Normalized Finite

|   |         |           |
|---|---------|-----------|
| 0 | 1...254 | Any Value |
|---|---------|-----------|

|   |     |   |
|---|-----|---|
| 1 | 255 | 0 |
|---|-----|---|

-inf

+inf

|   |     |   |
|---|-----|---|
| 0 | 255 | 0 |
|---|-----|---|

|   |     |       |
|---|-----|-------|
| X | 255 | 1.0XX |
|---|-----|-------|

-SNaN

+SNaN

|   |     |       |
|---|-----|-------|
| X | 255 | 1.0XX |
|---|-----|-------|

|   |     |       |
|---|-----|-------|
| X | 255 | 1.1XX |
|---|-----|-------|

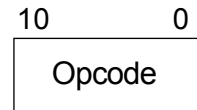
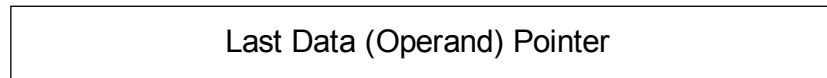
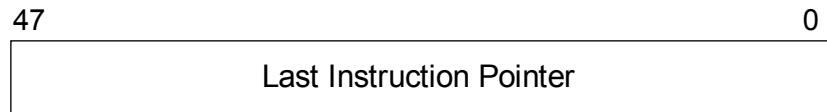
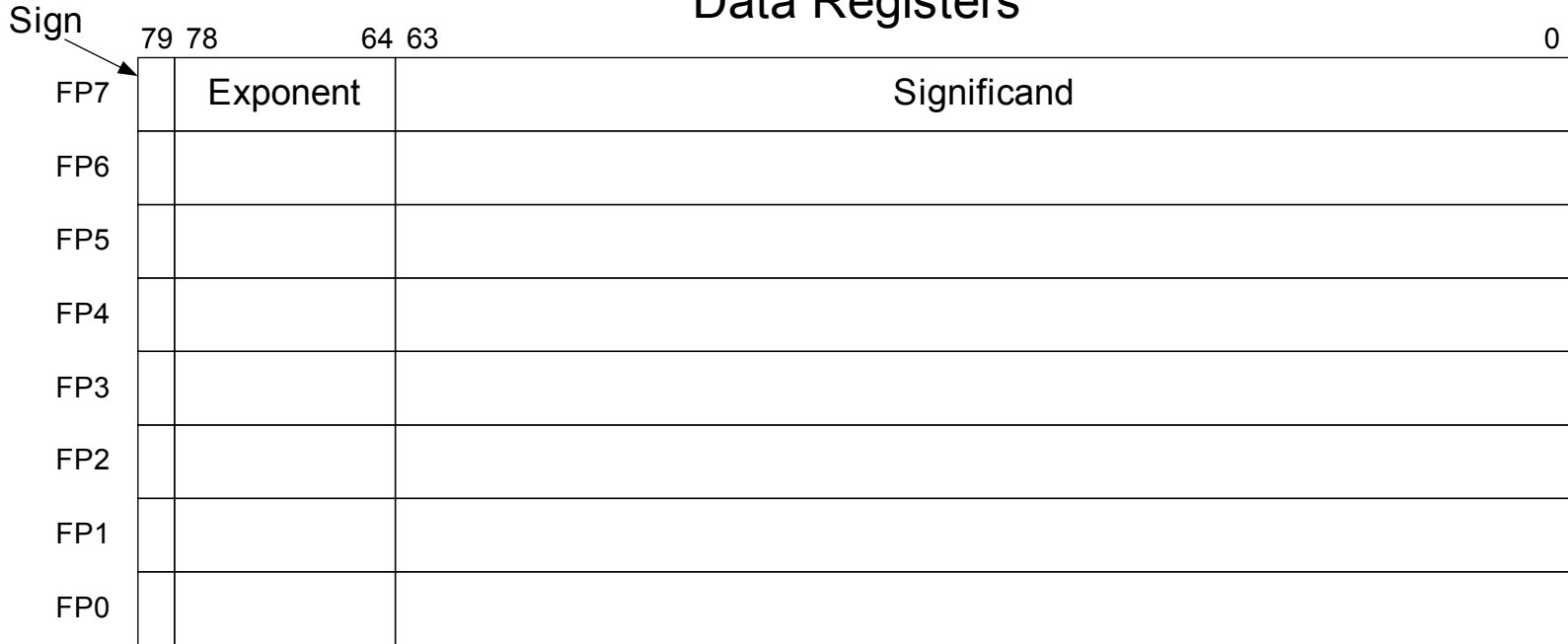
-QNaN

+QNaN

|   |     |       |
|---|-----|-------|
| X | 255 | 1.1XX |
|---|-----|-------|

# Floating Point Registers

# Data Registers



# FPU Data Register Stack

- FPU data registers are treated as a register stack
- All addressing of the data registers is relative to the register on top of stack
- The register number of the current top-of-stack register is stored in the TOP field in the FPU status word
- Load operations decrement TOP by one and load a value into the new top-of-stack register (equivalent to PUSH)
- Store operations store the value from the current TOP register in memory and then increment TOP by one (equivalent to POP)
- If a load operation is performed when TOP is at 0, register wraparound occurs and the new value of TOP is set to 7

# Example FPU Dot Product Computation

**Computation:** Dot Product

$$(5.6, 3.8) \times (2.4, 10.3) = (5.6 \times 2.4) + (3.8 \times 10.3)$$

**Code:**

```

FLD value1; (a) value1 = 5.6
FMUL value2; (b) value2 = 2.4
FLD value3; value3 = 3.8
FMUL value4; (c) value4 = 10.3
FADD ST(1); (d)

```

| (a) |     | (b) |       | (c) |       | (d) |       |
|-----|-----|-----|-------|-----|-------|-----|-------|
| R7  |     | R7  |       | R7  |       | R7  |       |
| R6  |     | R6  |       | R6  |       | R6  |       |
| R5  |     | R5  |       | R5  |       | R5  |       |
| R4  | 5.6 | R4  | 13.44 | R4  | 13.44 | R4  | 13.44 |
| R3  |     | R3  |       | R3  | 39.14 | R3  | 52.58 |
| R2  |     | R2  |       | R2  |       | R2  |       |
| R1  |     | R1  |       | R1  |       | R1  |       |
| R0  |     | R0  |       | R0  |       | R0  |       |

ST(0)

ST(0)

ST(1)

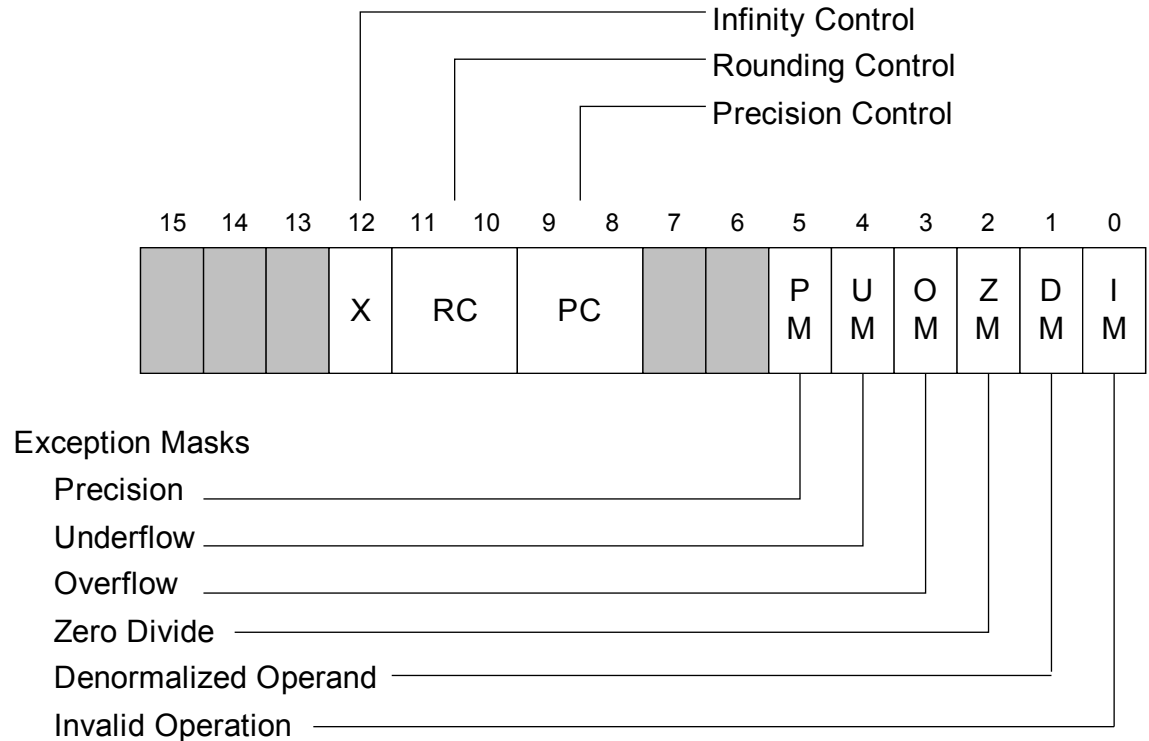
ST(1)

ST(0)

ST(0)

# Floating Point - Control

- Rounding modes
- Precision control
- Exception Masks



## Rounding Control

- 00 - Round to Nearest or Even
- 01 - Round Down (Toward  $-\infty$ )
- 10 - Round Up (Toward  $+\infty$ )
- 11 - Chop (Truncate Toward 0)

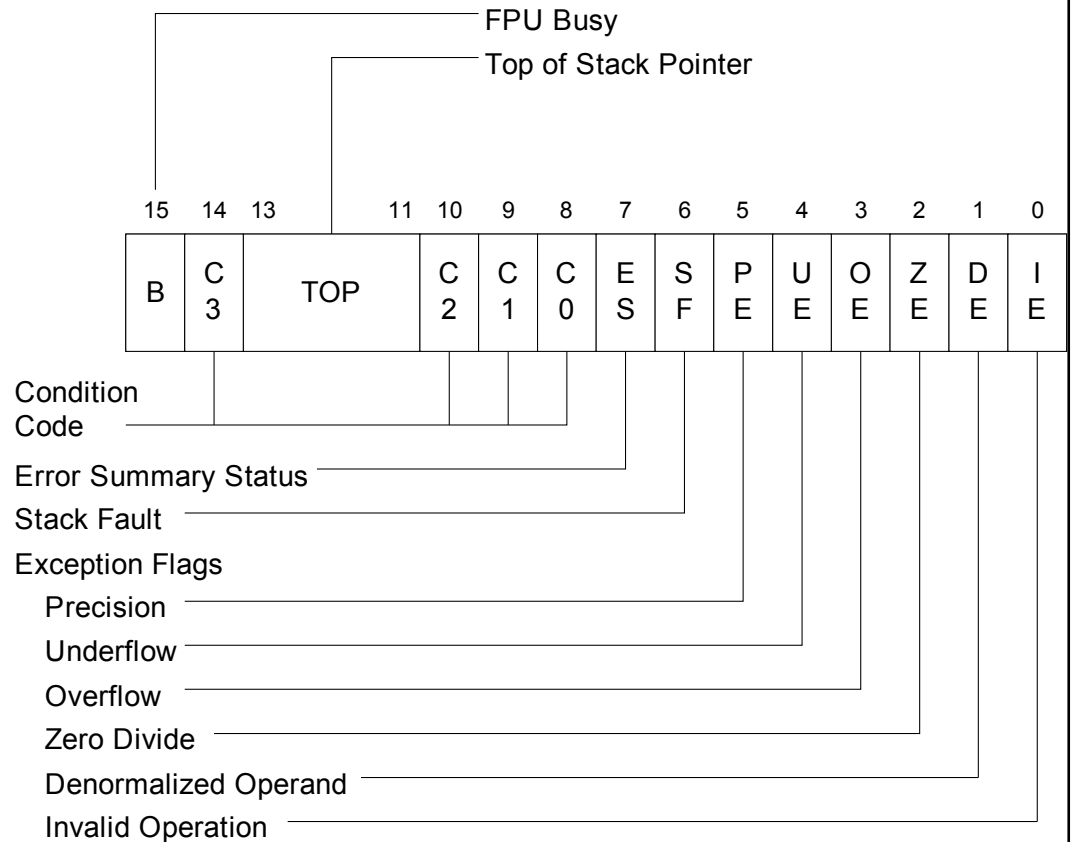
## Precision control

- 00 - 24 Bits (Single Precision)
- 01 - (Reserved)
- 10 - 53 Bits (Double Precision)
- 11 - 64 Bits (Extended Precision)

\* Infinity Control has no effect on 386 and on.  
Used for compatibility with 8086 & 286

# Floating Point – Status Word

- Exception Flags
- Top Of Stack position
- Complex exception model and handling.  
Not discussed.
- ES is set if any unmasked exception bit is set;  
Cleared otherwise
- Top values:  
000 = Register 0 is top of stack  
001 = Register 1 is top of stack  
....  
111 = Register 7 is top of stack



# Floating Point – Tag Word

- Flag each stack value as Valid, Empty, Zero, Special

|        |    |        |    |        |    |        |   |        |   |        |   |        |   |        |   |
|--------|----|--------|----|--------|----|--------|---|--------|---|--------|---|--------|---|--------|---|
| 15     | 14 | 13     | 12 | 11     | 10 | 9      | 8 | 7      | 6 | 5      | 4 | 3      | 2 | 1      | 0 |
| Tag(7) |    | Tag(6) |    | Tag(5) |    | Tag(4) |   | Tag(3) |   | Tag(2) |   | Tag(1) |   | Tag(0) |   |

## Tag Values

00 = Valid

01 = Zero

10 = Special: Invalid (NaN, Unsupported, Infinity, or Denormal)

11 = Empty

# Another Example – Quadratic Equation

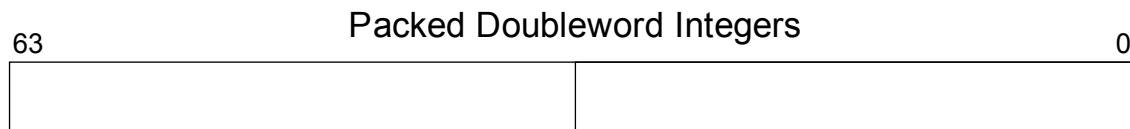
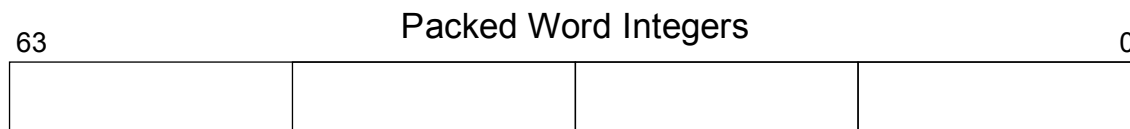
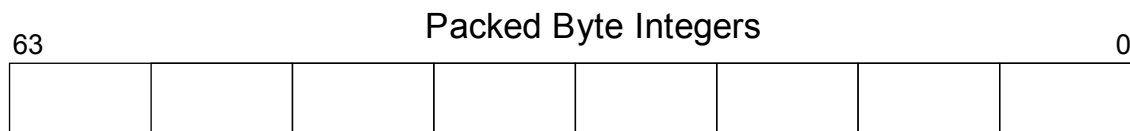
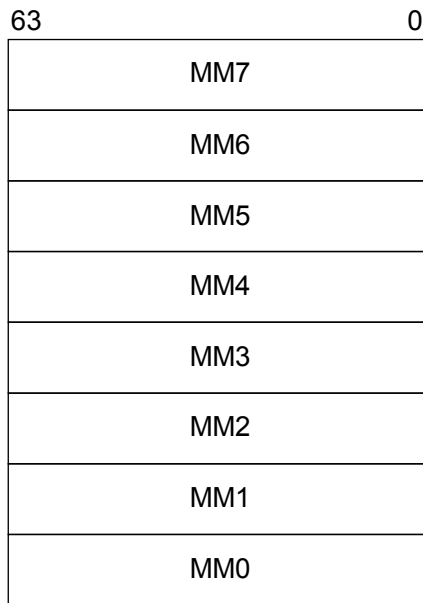
|        |      |            |        |   |    |          |     |
|--------|------|------------|--------|---|----|----------|-----|
|        | COMM | _x1:QWORD  | ;      | S | E  | P        |     |
| _a     | DD   | 040b00000r | ; 5.5  | 0 | 81 | (1.)6000 |     |
| _b     | DD   | 040800000r | ; 4    | 0 | 81 | (1.)0    |     |
| _c     | DD   | 03e800000r | ; 0.25 | 0 | 7B | (1.)0    |     |
| _i2    | DD   | 02H        |        |   |    |          |     |
| _\$T32 | DD   | 040800000r | ; 4    |   |    |          | TOS |
|        |      |            |        |   |    |          | 0   |
|        |      |            |        |   |    |          | 7   |
|        |      |            |        |   |    |          | 6   |
|        |      |            |        |   |    |          | 5   |
|        |      |            |        |   |    |          | 6   |
|        |      |            |        |   |    |          | 7   |
|        |      |            |        |   |    |          | 6   |
|        |      |            |        |   |    |          | 7   |
|        |      |            |        |   |    |          | 6   |
|        |      |            |        |   |    |          | 7   |
|        |      |            |        |   |    |          | 0   |

# MMX

# MMX

- An extension to the X86 Architecture
- Useful mainly for multimedia applications
- Technique: Single Instruction, Multiple Data (SIMD) with existing registers
- New data types - packed 64 bit consists of either
  - ◆ 8 bytes
  - ◆ 4 words
  - ◆ 2 doublewords
- Instructions to operate on this data
- Software model to allow easy migration with existing OS

# MMX Registers and Data Types



# MMX Instructions

## ● Operations (57 instructions)

### ◆ Arithmetic (saturating & wrap-around)

Add, Subtract, Multiply, Multiply-add (for fast multiply accumulate)

### ◆ Logical

And, Or, Shifts, Compares (including arithmetic)

### ◆ Conversions

Pack from large to smaller words

Unpack from small to large words

### ◆ Transfers

Register to register

Load from and store to memory

## ● Integration into Intel Architecture

### ◆ 8 new registers - MM0- MM7

### ◆ Uses previously reserved opcodes

### ◆ Uses same addressing modes

# Example of Operations

## Packed Add

**PADDW:** Add Packed word

|       |       |       |       |
|-------|-------|-------|-------|
| a3    | a2    | a1    | FFFFh |
| +     | +     | +     | +     |
| b3    | b2    | b1    | 8000h |
| <hr/> |       |       |       |
| a3+b3 | a2+b2 | a1+b1 | 7FFFh |

**PADDUSW:** Add Unsigned Packed word with Saturate

|       |       |       |       |
|-------|-------|-------|-------|
| a3    | a2    | a1    | FFFFh |
| +     | +     | +     | +     |
| b3    | b2    | b1    | 8000h |
| <hr/> |       |       |       |
| a3+b3 | a2+b2 | a1+b1 | FFFFh |

## Multiply-Add

**PMADDWD:** multiply and add

|             |    |             |    |
|-------------|----|-------------|----|
| a3          | a2 | a1          | a0 |
| *           | *  | *           | *  |
| b3          | b2 | b1          | b0 |
| <hr/>       |    |             |    |
| a3*b3+a2*b2 |    | a1*b1+a0*b0 |    |

## Shift

**t**

**PSLLQ:** Shift left logical all 64-bit

|       |    |    |       |
|-------|----|----|-------|
| a3    | a2 | a1 | a0    |
|       |    |    | 0010h |
| <hr/> |    |    |       |
| a2    | a1 | a0 | 0000h |

# Example Operations

## Multiply-Add

**PMADDWD:** multiply and add

|    |    |    |    |
|----|----|----|----|
| a3 | a2 | a1 | a0 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| *  | *  | *  | *  |
| b3 | b2 | b1 | b0 |

|               |               |
|---------------|---------------|
| $a3*b3+a2*b2$ | $a1*b1+a0*b0$ |
|---------------|---------------|

## Multiply-Low

**PMULLW:** multiply low

|    |    |    |    |
|----|----|----|----|
| a3 | a2 | a1 | a0 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| b3 | b2 | b1 | b0 |
|----|----|----|----|

|         |         |         |         |
|---------|---------|---------|---------|
| $a3*b3$ | $a2*b2$ | $a1*b1$ | $a0*b0$ |
|---------|---------|---------|---------|

|    |    |    |    |
|----|----|----|----|
| c3 | c2 | c1 | c0 |
|----|----|----|----|

**low order 16 bits**

## Multiply-High

**PMULLHW:** multiply high

|    |    |    |    |
|----|----|----|----|
| a3 | a2 | a1 | a0 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| b3 | b2 | b1 | b0 |
|----|----|----|----|

|         |         |         |         |
|---------|---------|---------|---------|
| $a3*b3$ | $a2*b2$ | $a1*b1$ | $a0*b0$ |
|---------|---------|---------|---------|

|    |    |    |    |
|----|----|----|----|
| c3 | c2 | c1 | c0 |
|----|----|----|----|

**High order 16 bits**

# FP/MMX view of registers

- MMX registers MM0-MM7 overlap FP registers

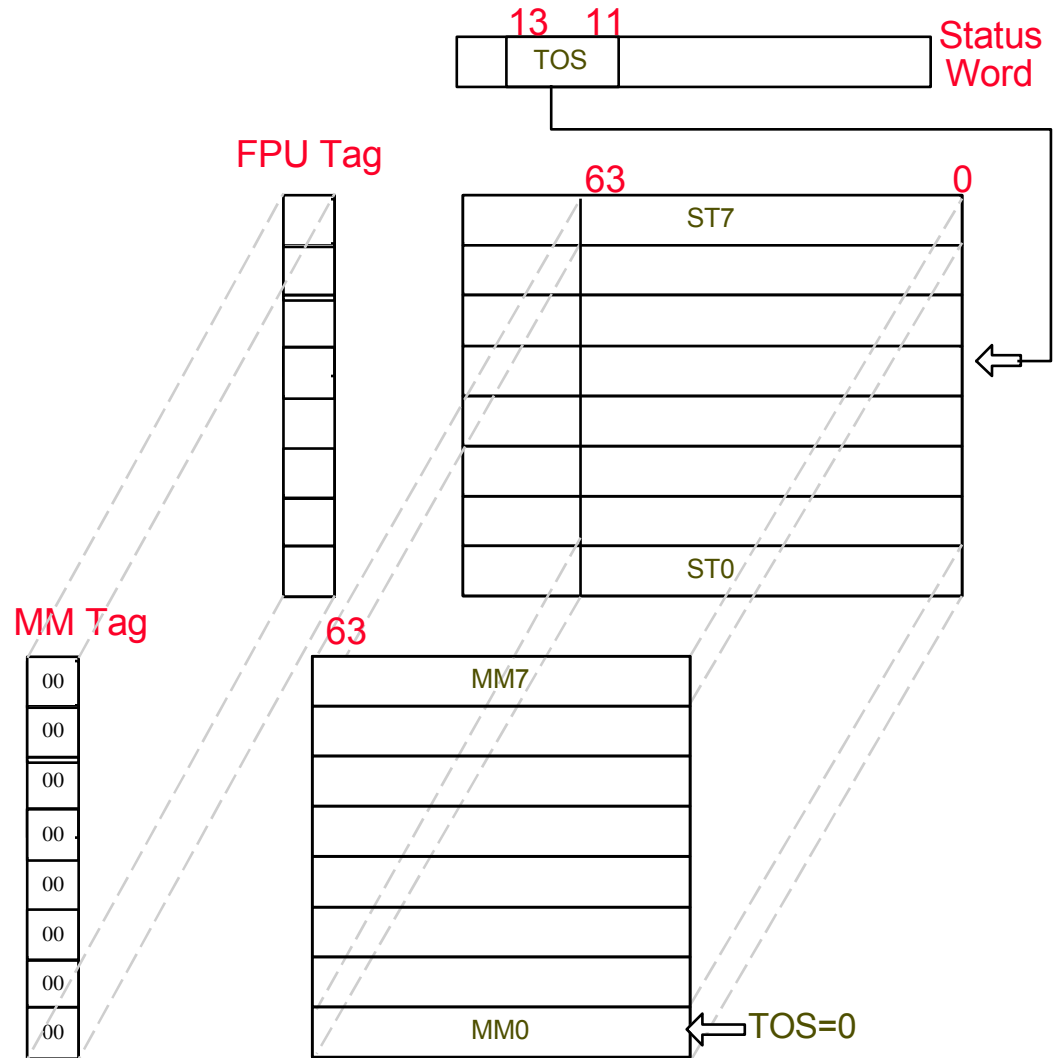
- ◆ All tags are valid w/ MMX
- ◆ TOS is 0

- Pros

- ◆ No change of context
- ◆ Easy OS migration

- Cons

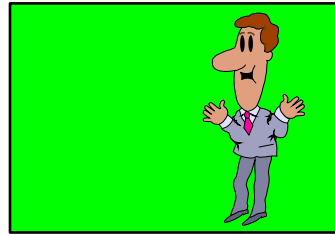
- ◆ Difficult to Intermix MMX & FP
- ◆ Need to signal MMX to FP move (EMMS: clear tags)



# Example: Conditional Select / Branch Removal

- Packed Comparison (PCMPEQW) and the logical instructions enable conditional select operations in parallel and without data dependent branches.

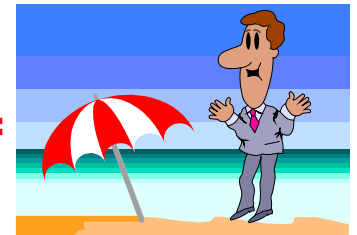
X



Y



new\_image



X1=green X2 != green X3=green X4 !=green

PCMPEQW

green green green green

if (X[i] != green) then  
new\_image[i] = X[i]  
else  
new\_image[i] = Y[i]

PANDN

|        |        |        |        |
|--------|--------|--------|--------|
| 0xFFFF | 0x0000 | 0xFFFF | 0x0000 |
| X1     | X2     | X3     | X4     |

PAND

|        |        |        |        |
|--------|--------|--------|--------|
| 0xFFFF | 0x0000 | 0xFFFF | 0x0000 |
| Y1     | Y2     | Y3     | Y4     |

```
MOVQ MM1, X
PCMPEQW MM1, GREEN
MOVQ MM2, MM1
PANDN MM1, X
PAND MM2, Y
POR MM1, MM2
MOVQ New, MM1
```

POR

finished pixels

|        |        |        |        |
|--------|--------|--------|--------|
| 0x0000 | X2     | 0x0000 | X4     |
| Y1     | 0x0000 | Y3     | 0x0000 |
| Y1     | X2     | Y3     | X4     |

# Streaming SIMD Extensions

- New technology to exploit parallelism in FP and INT applications
- Key capabilities
  - ◆ Packed Operations
  - ◆ Branch Removal/Compression
  - ◆ Data Movement/Hints
  - ◆ FP/INT Type Conversion
- Application Domains...
  - ◆ 3D Graphics: geometry, lighting
  - ◆ signal processing
  - ◆ high precision simulation & modeling
  - ◆ video encoding/decoding
  - ◆ other apps requiring streaming input and output

# SIMD FP instructions types

- arithmetic
- square root
- approximation instructions
- min & max
- loads & stores
- move mask
- compare & set mask
- logical
- compare & set eflags
- conversion

## Instruction Categories

SIMD FP - 4 wide, single-precision, 2 wide double precision

SIMD INT - extensions to MMX™ technology capabilities,

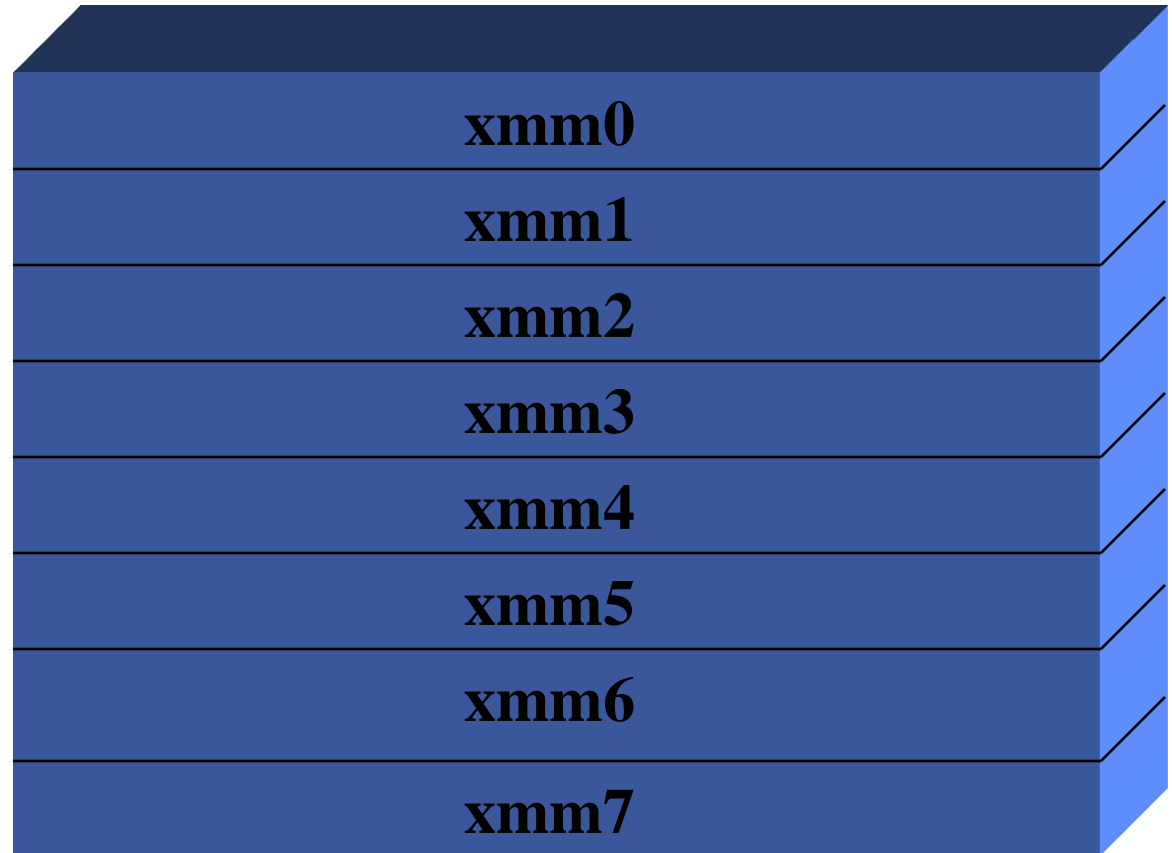
Extending all the MMX instruction to 128 bit using the new XMM registers

Cacheability control

State management

# New SIMD FP Registers

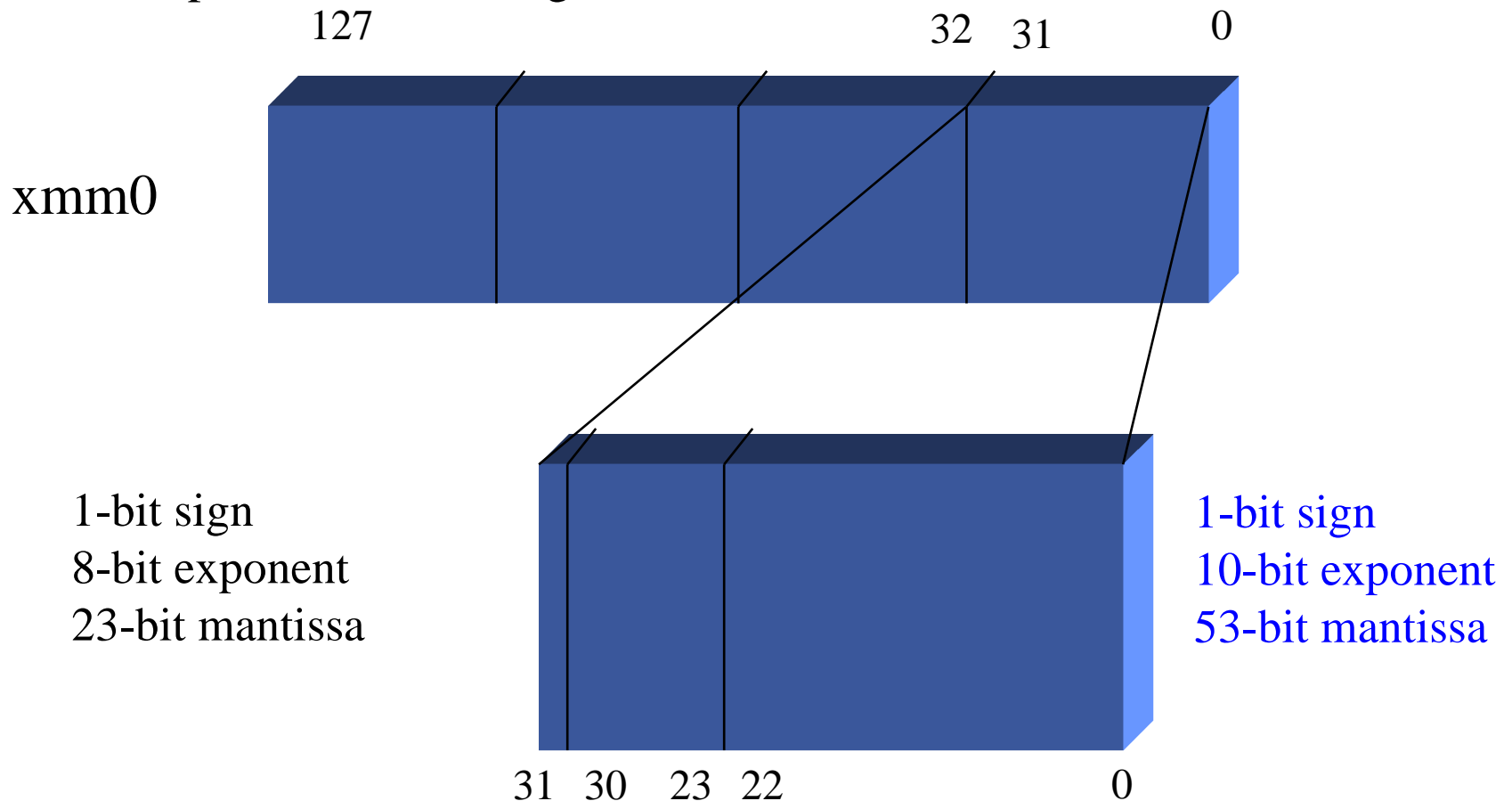
- New set of registers!
- Direct access
- Used for data only
- Extended processor state



eight 128-bit, 4x32bit single-precision FP  
2x64bit single-precision FP, 128 bit MMX

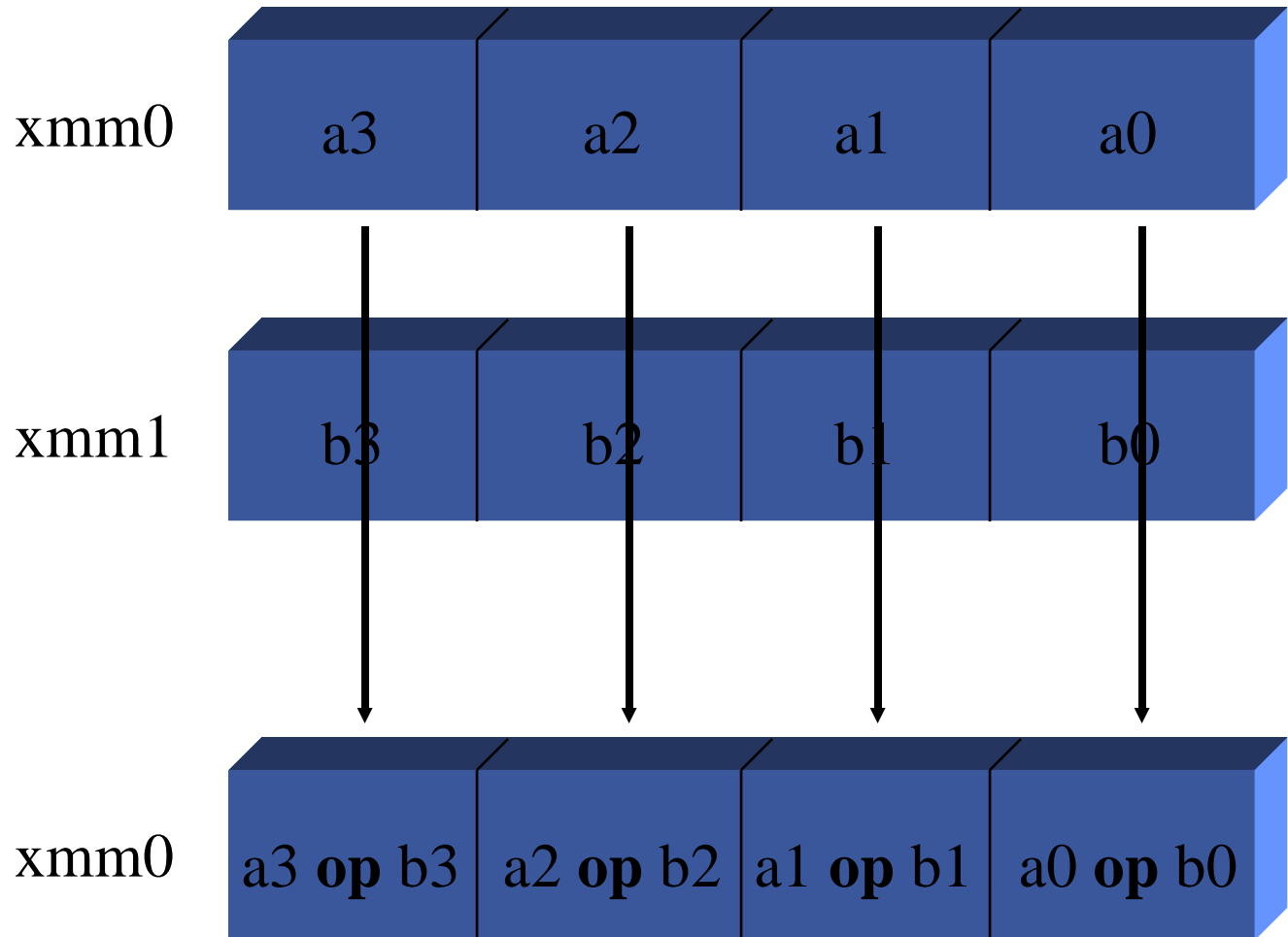
# Packed SP Data Type

- Each register holds 4 single-precision FP values or 2 double precision
- IEEE-754 compatible
- Scalar operates on least-significant number



# Packed Operations

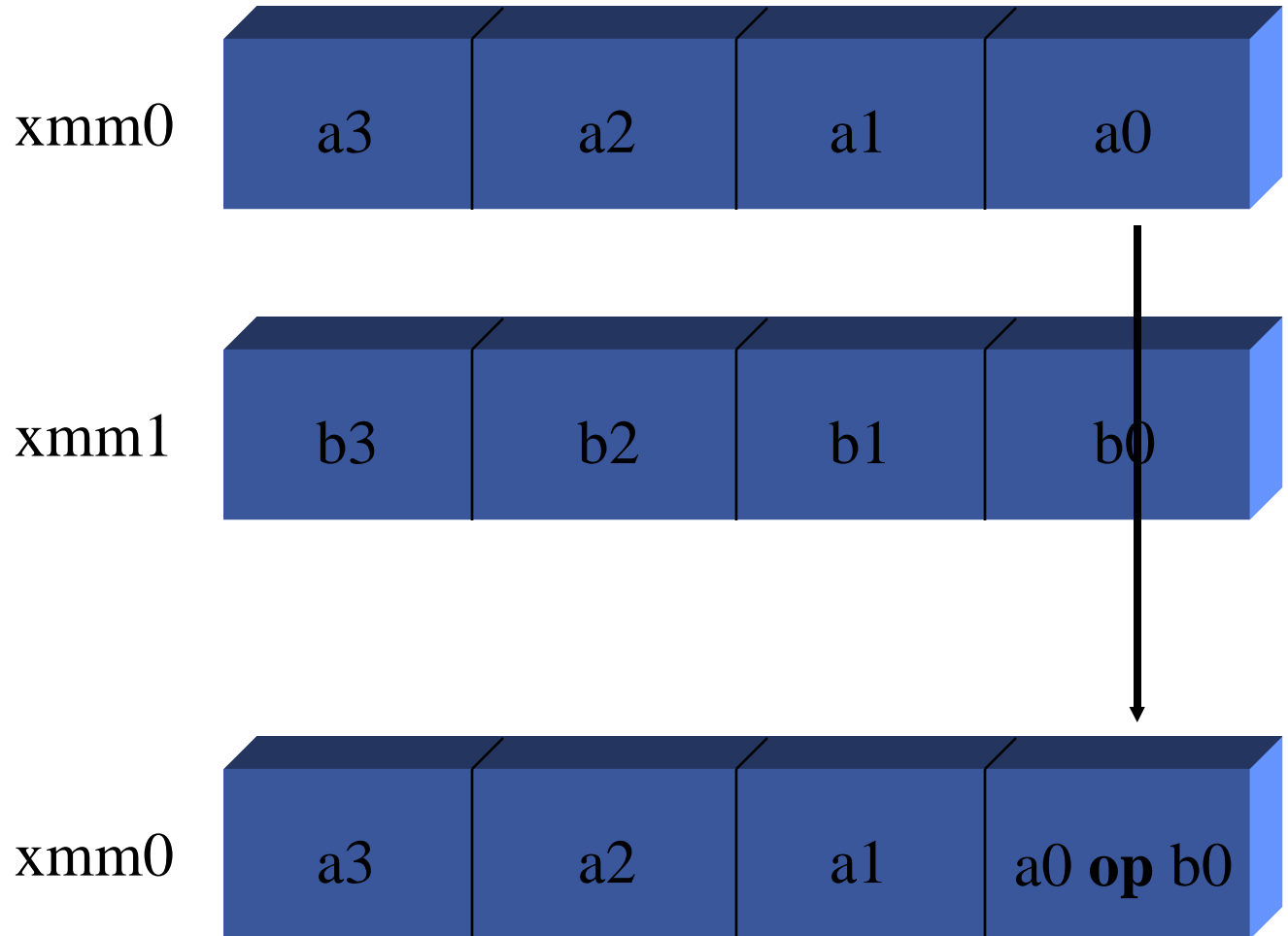
- op** is one of
- addps **addpd**
  - subps **subpd**
  - mulps **mulpd**
  - divps **divpd**



# Scalar Operations

**op** is one of

- Addss **addsd**
- subss **subsd**
- mulss **mulsd**
- divss **divsd**



# SIMD Data Organization

- Exploit vertical parallelism
- SOA versus AOS
  - ◆ Array of Structures
  - ◆ Structure of Arrays

$X_0, Y_0, Z_0, X_1, Y_1, Z_1, X_2, Y_2, Z_2, \dots$

*Better cacheability  
Better SIMD calculations*

$X_0, X_1, X_2, X_3, \dots Y_0, Y_1, Y_2, Y_3, \dots Z_0, Z_1, Z_2, Z_3, \dots$

# Matrix Vector Multiply example

- Typical 3D operation

- Load values in SOA format

  - ◆ xxxx..., yyyy..., zzzz...

- Follow with multiply and add operations

```
movaps xmm0, [list+X+ecx] ;load X components
movaps xmm2, [list+Y+ecx] ;load Y components
movaps xmm3, [list+Z+ecx] ;load Z components
movaps xmm1, [esi+m00] ;m00 m00 m00 m00
movaps xmm4, [esi+m01] ;m01 m01 m01 m01
```

Loop back and pick up next 4 vertices...

```
mulps xmm1, xmm0 ;x*m00 x*m00 x*m00 x*m00
mulps xmm4, xmm2 ;y*m01 y*m01 y*m01 y*m01
addps xmm4, xmm1 ;add the 2 results
movaps xmm1, [esi+m02] ;load matrix element m02 (x4)
mulps xmm1, xmm3 ;z*m02 z*m02 z*m02 z*m02
addps xmm4, xmm1 ;add results
addps xmm4, [esi+m03] ;add last element of matrix
```

# SIMD Integer Instructions

- Extensions to MMX™ technology instructions
- Operate on same 64-bit registers as previous MMX technology instructions, Or on the new 128 bit XMM registers
- Instructions:
  - ◆ extract, insert, min/max, byte mask → integer, multiply high unsigned, shuffle multiply unsigned double, add and sub qword, average