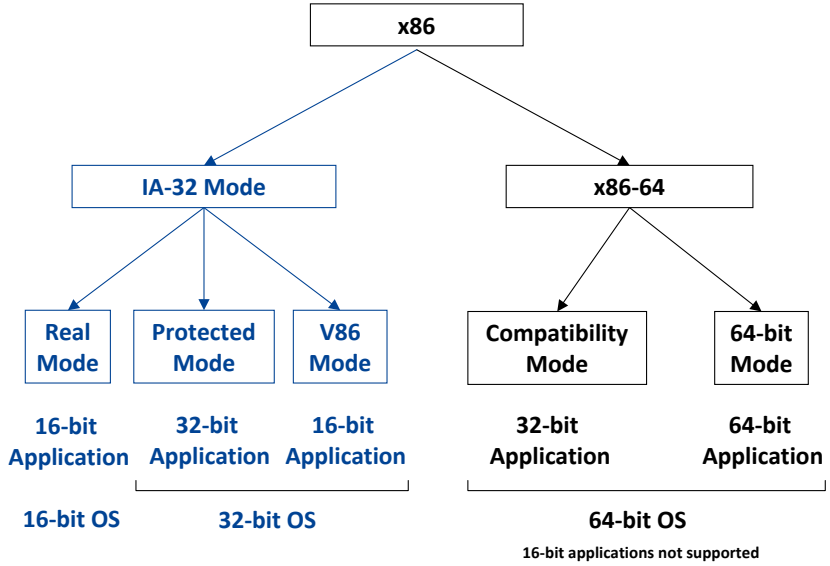
IA-32

Intel 32/64-bit Architecture

Operating Modes for Intel x86 Processors



Intel 32-bit Architecture — IA-32

Instruction Set Architecture for 32-bit Intel processors

- 1985 — now
- 80386 — Core / Xeon / Centrino

Characteristics

- Backward compatible with 8086, 80186, and 80286
- 32-bit integer
- 32-bit physical address
  - 2^{32} Bytes = 4 GB of addressable memory
- 32-bit general purpose registers (GPR)
  - **EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP**
- 6 segment registers (SR)
  - **CS, DS, SS, ES, FS, GS**
- Hardware support for operating system

IA-32 introduced in 1985

- 80386 processor + full Unix implementation

Operating Modes

Real Mode
Start-up mode
8086 features

↓

Protected Mode
Full IA-32 features

IA-32 processors initialize into real mode

- 16-bit integers and address offsets
- 16-bit GPRs
  - **AX, BX, CX, DX, SI, DI, BP, SP, IP**
- 4 segment registers
  - **CS, DS, SS, ES**
- 20-bit physical address
  - Access lowest 1 MB of RAM
- 8086 interrupts

32-bit OS shifts processor into protected mode

- Windows/Linux/Unix/Mac
- 32-bit GPRs + 6 SRs + 8 system registers
- Hardware support for OS
  - Task management
  - Advanced segmentation model
  - Virtual memory and paging management
  - Advanced interrupt mechanism

IA-32 Memory Model

Segmentation

Functional division of address space

Segment defined by type — Data / Code / System

Access restricted by type

LOGICAL ADDRESS = **SEGMENT:OFFSET** = software address

Paging

Virtual division of address space

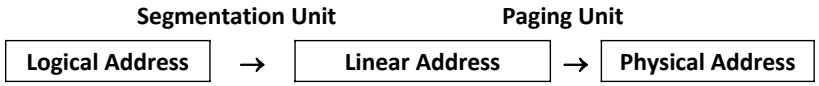
Managed by OS for page swapping + address aliasing

LINEAR ADDRESS = 32-bit address seen by OS

Physical address

PHYSICAL ADDRESS = real address in physical memory

Address translation

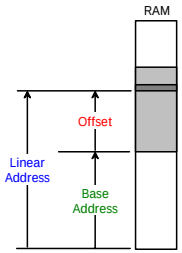


Logical Address Translation

Logical to linear address

Linear address = **base address** + **offset**

Base address = linear address of first byte in segment



Segment register

Holds **SEGMENT** = 16-bit segment **SELECTOR**

8086 segment mapping

SEG × **10h** = 20-bit physical base address

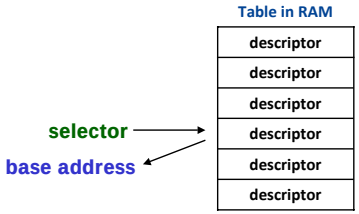
SEG = 16-bit segment selector	0000
segment register	const

IA-32 segment mapping

Selector = index to descriptor table

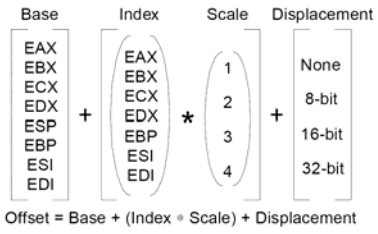
Descriptor = table entry holding

- 32-bit base address
- Segment size
- Segment type + access rights



OFFSET = Effective Address

Effective Address in IA-32

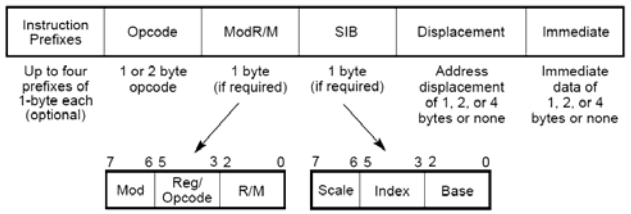


Example

```
mov eax,[eax+4*edi+11223344h]
```

On Pentium+ **scale = 1, 2, 3, 4, 8**

IA-32 instruction encoding



Intel x86 General Register Access

16-bit mode

8-bit access

AL, BL, CL, DL,
AH, BH, CH, DH

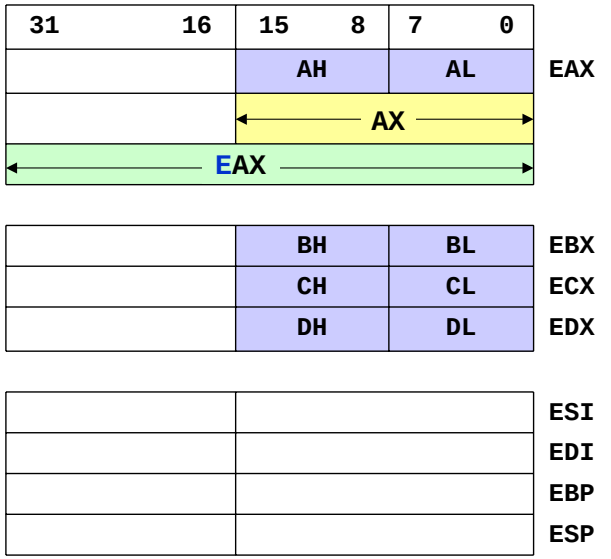
16-bit registers

AX, BX, CX, DX,
SI, DI, BP, SP

32-bit mode

32-bit registers

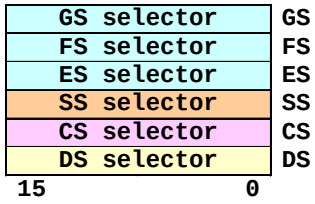
EAX, EBX,
ECX, EDX,
ESI, EDI,
EBP, ESP

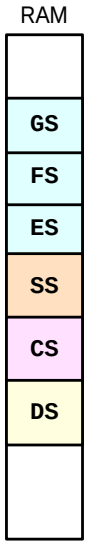


User Segment Registers

Six user segment registers

16-bit **SELECTOR** = pointer to memory segment





Six defined user segments

CS

Code segment

Accessible by instruction fetch

DS, ES, FS, GS

Data segments

Accessible by load / store instructions

SS

Stack segment

Accessible by stack instructions

IA-32 Segmentation Example

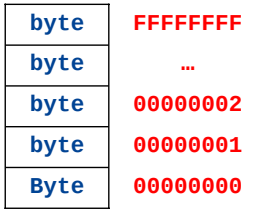
Offset

32-bit number

Combination of registers and immediate values

Offset ∈ {00000000, … , FFFFFFFF}

Maximum segment size = 2^{32} bytes = 4 GB



Linear Address = base address + offset

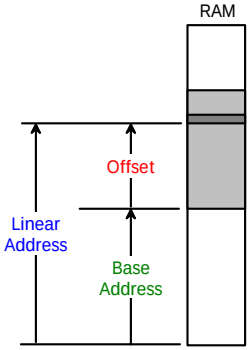
Example

Logical Address = 1234:11223344

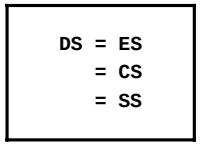
Segment selector = 1234 → descriptor table

Base address = 00000000

Linear Address = 0 + 11223344 = 11223344

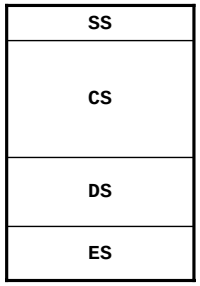


Typical Segment Register Usage



DOS *.com program

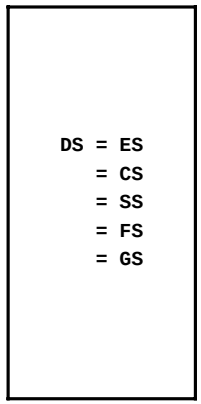
One 64 KB segment



DOS *.exe program

Four defined segments

Segment ≤ 64 KB



Linux software

One 4 GB segment

OS allocates memory to programs

Descriptor Tables

Segment definition

Write 64-bit descriptor → Descriptor Table in RAM

Specify

- Base address — linear address of first byte in segment
- Limit — maximum offset into segment (segment size)
- Access — segment type + access rights

Global Descriptor Table (GDT)

Accessible by any task

Local Descriptor Table (LDT)

Private to task

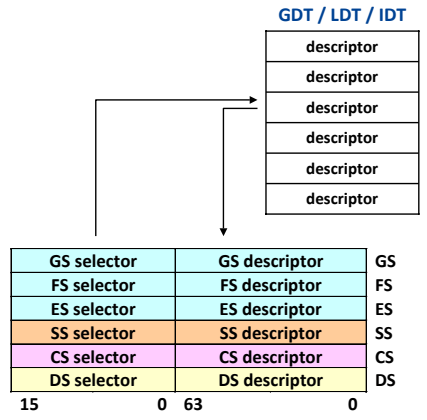
Interrupt Descriptor Table (IDT)

Accessed on trap / interrupt

Shadow registers

Descriptor entry

Copied to CPU from RAM table

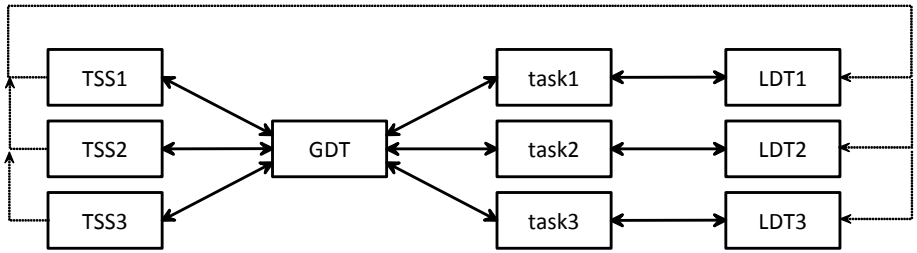


Task / Process Control in IA-32

IA-32 process allocated Task State Segment (TSS)

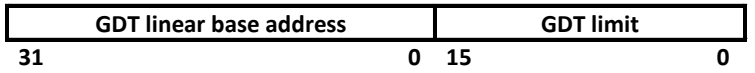
- Context for swapped-out process
  - General register values
  - OS-specific information
- Segment register values
- Pointer (selector) to LDT via GDT
- Status registers

TSS selector points to TSS entry via GDT

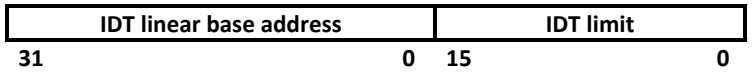


Segmentation Table Registers

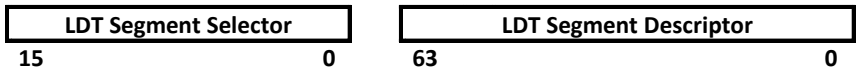
GDT Register (Global Descriptor Table)



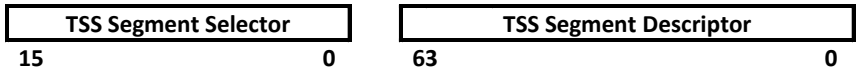
IDT Register (Interrupt Descriptor Table)



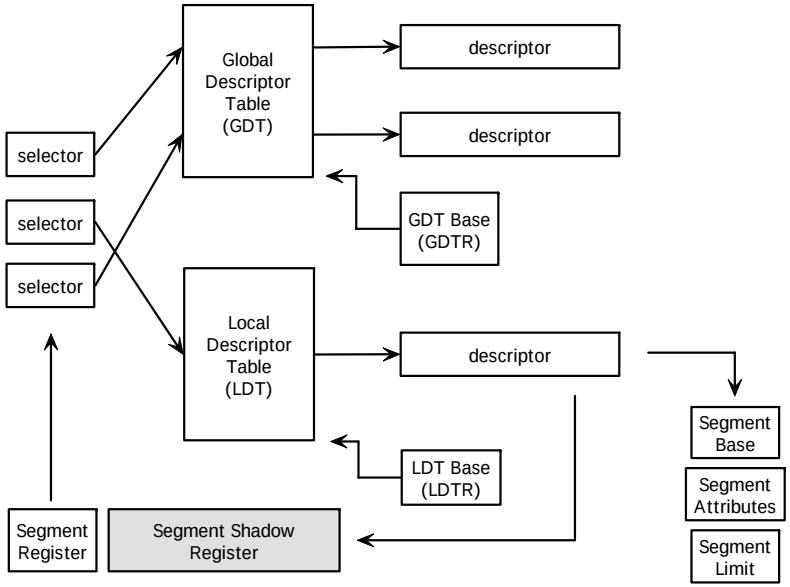
LDT Register (Local Descriptor Table) **Shadow Register**



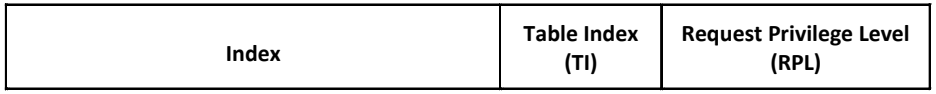
TSS Register (Task State Segment) **Shadow Register**



Segmentation Model



Selector Format



13-bit index to descriptor table

$2^{13} = 2^3 \times 2^{10} = 8 \times 1$ K

8 K (8192) descriptors per table

Descriptor address = Table base address + descriptor offset

Descriptor = 64 bits = 8 bytes

Descriptor offset = Index × 8 = Index × 1000_2 = Selector AND FFF8

8 K Descriptors/table × 8 Bytes/Descriptor = 64 KB/table

Table Index

TI = 0 for GDT

TI = 1 for LDT

Request Privilege Level (RPL)

Copy of user privilege level when selector passed as pointer

Descriptor Format

8	4	4	8	24	16
Base	Access	Limit	Access	Base	Limit
31 ... 24	11 ... 8	19 ... 16	7 ... 0	23 ... 0	15 ... 0
63 56	55 52	51 48	47 40	39 16	15 0

	11	10	9	8	7	6 5	4	3 2 1	0	
Access	G	D	0	0	P	DPL	S = 1	TYPE	A	User Segment
	G	0	0	0	P	DPL	S = 0	TYPE		System Segment

Base	32-bit segment base address
Limit	20-bit offset limit Segment Size = [1 + Limit] × $(4096)^G$ Bytes G= 0 ⇒ Byte Granularity: 2^{20} bytes = 1 MB maximum segment size G= 1 ⇒ 4 KB Granularity: 2^{20} × 4 KB = 4 GB maximum segment size
Code Type	D = 0 ⇒ default 16-bit code + effective address
	D = 1 ⇒ default 32-bit code + effective address
Present	P = 1 ⇒ segment in memory P = 0 ⇒ swapped-out segment
DPL	Descriptor Privilege Level
System	S = 0 ⇒ system segment S = 1 ⇒ user segment
Type	Segment type code
Access	A = 0 ⇒ segment not accessed A = 1 ⇒ access has been accessed

Type Fields

User Segments (S = 1)

	11	10	9	8	7	6 5	4	3 2 1	0	
Access	G	D	0	0	P	DPL	S = 1	TYPE	A	User
	G	0	0	0	P	DPL	S = 0	TYPE		System

Code Segment

- Type = 1 C R
- C = 1 for Conforming code (in protection scheme)
- R = 1 for Readable Code (**`MOV EAX,CS:EA`** legal)

Data Segment

- Type = 0 ED W
- ED = 1 for Expand Down (stack segment)
- W = 0 for Read-Only segment

System Segments (S = 0)

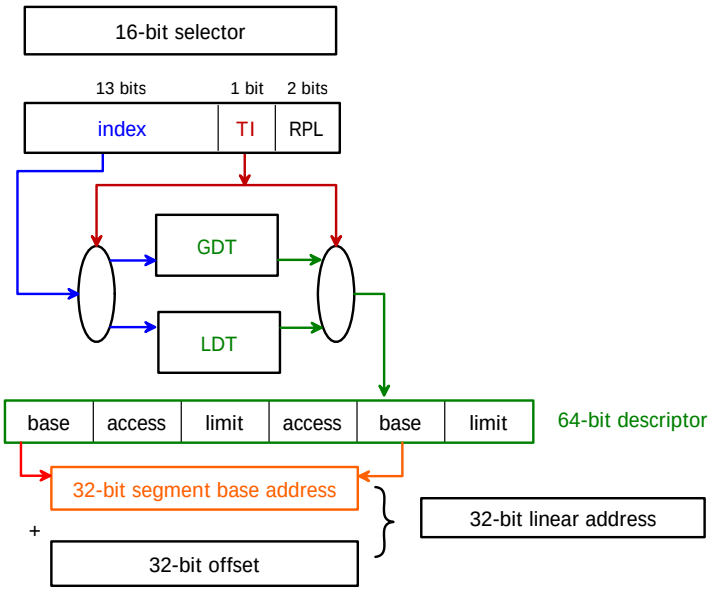
LDT Segment

- Type = 0010

Task State Segment

- Active task: Type = 1011
- Inactive task: Type = 1001

Segment Address Translation

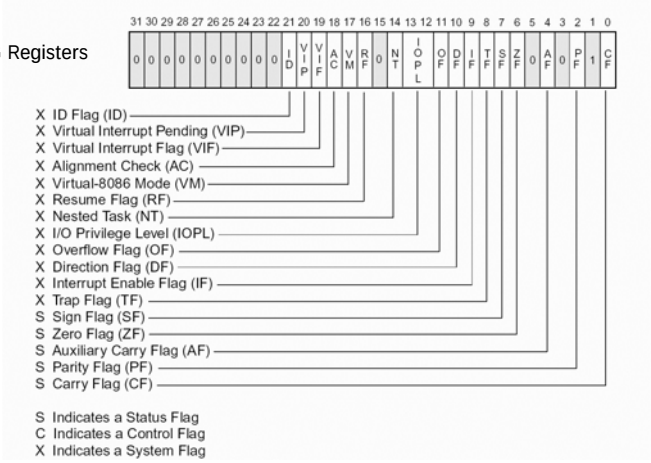


Control + Status Registers

Control Registers

CR0	CR1	CR2	CR3	CR4
Options	Reserved	Last Page Fault	Directory Base Address	Protected Mode Options

EFLAG Registers



Linear Address Translation

10 bits	10 bits	12 bits
directory	page table	offset

directory = index into directory table

2^{10} =1024 = 1K page table entries per directory

4 bytes per entry ⇒ 4 KB per directory

page table = index into selected page table

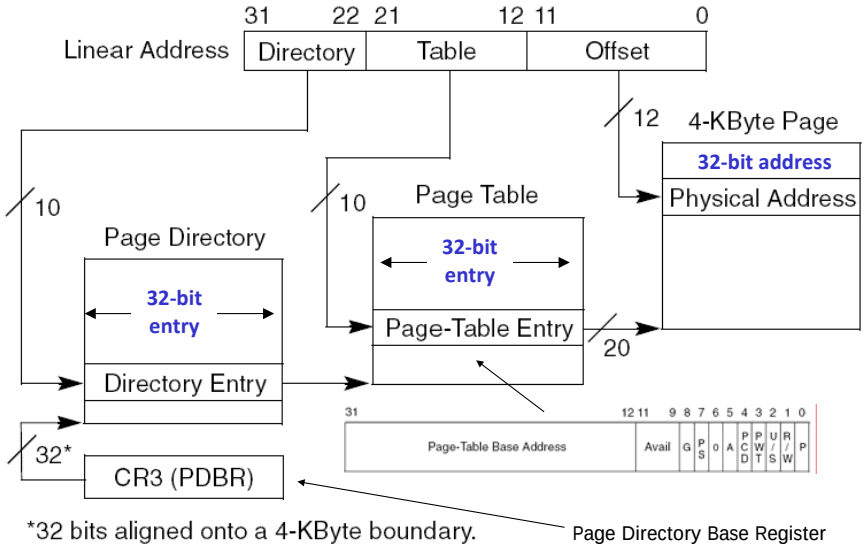
4 bytes per entry ⇒ 4 KB per page table

offset = index (of byte) into selected page

2^{12} = 4096 = 4 KB per page

2^{10} page tables × 2^{10} page/page table × 2^{12} bytes/page = 2^{32} bytes

IA-32 4 KB Paging



Entries in Page Directory and Page Table

Upper 20 bits of Physical Address	OS reserved	G	PS	D	A	0	U/S	R/W	P
31 12	11 9	8	7	6	5	4 3	2	1	0

Pages aligned on 4 KB boundaries

Start address = page number × **1000h**

12 lower bits of start address = **000h**

Upper 20 bits of address = Page Number

D — dirty bit **A** — accessed **P** = 0 ⇒ swapped-out **P** = 1 ⇒ present

U/S = 0 ⇒ supervisor (kernel mode) page

R/W permission for user mode pages

R/W permission for supervisor data pages

U/S = 1 ⇒ user mode page

No access to supervisor data pages

Read-only access to data pages with R/W = 0

Read/write access to data pages with R/W = 1

PS — page size option

G — global option

Translation Lookaside Buffer (TLB)

linear address	physical address

Address Cache

Saves 32 linear address → physical address translations

CPU makes two accesses in parallel

Usual Directory/PT/Page translation

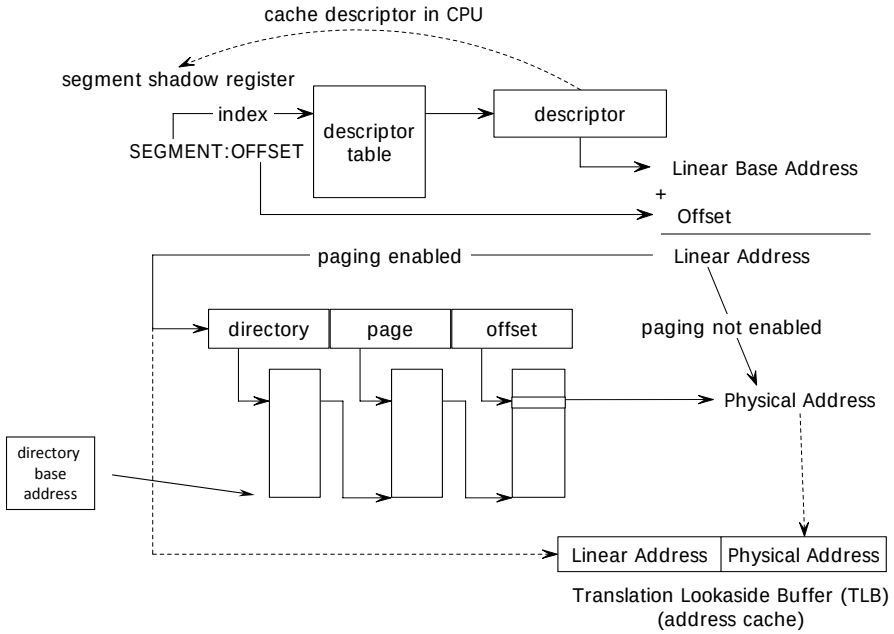
Access to TLB

If linear address in TLB

TLB responds first and cancels translation

TLB catches 98 - 99% of linear address accesses

Logical Address to Physical Address



Building Page Tables

Start with Physical Address = Linear Address

Physical address ≠ linear address after swapping

Build sequential tables — define sequential pages

Page Table 0

- Starts on page boundary at address p_start
- Table entry = 32 bits = 4 bytes
- Entries / table = 1 K = 400h ⇒ table size = 4 KB = 1000h bytes

Page Table PT

- Starts at address p_start + PT × 1000
- Entry n address = p_start + PT × 1000 + n × 4
- Points to page P = PT × 400 + n
  - Page P starts at address P × 1000
- Entry = (PT × 400 + n) × 1000 + 12 bits of OS information

Physical address = (PT × 400 + n) × 1000 + offset

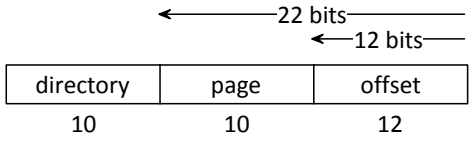
= PT × 400000 + n × 1000 + offset

= PT shifted 22 left + n shifted 12 left + offset

1000h = 2^{12} ⇒ 12 left-shifts
400h = 2^{10} ⇒ 10 left-shifts

Linear Address Format

Linear Address

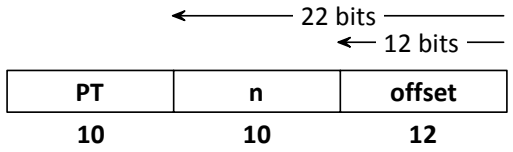


Physical Address from page translation is

A = PT × 400000 + n × 1000 + offset

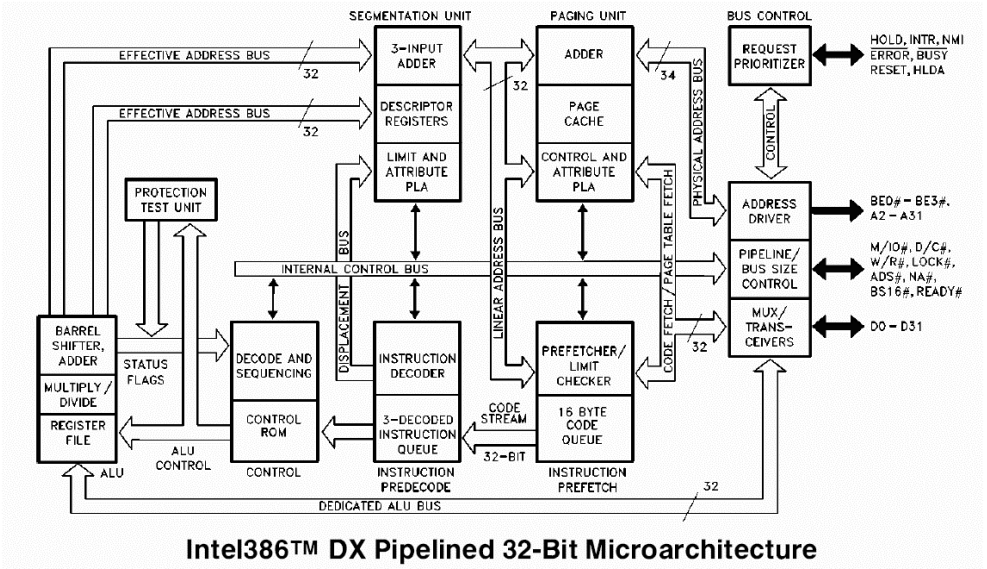
PT × 400000 = PT followed by 22 binary 0's

n × 1000 = n followed by 12 binary 0's



Physical Address ≡ Linear Address

Intel 80386 Microprocessor



Intel386™ DX Pipelined 32-Bit Microarchitecture

Paging Options

PG (paging)

Bit 31 of CR0

Enables IA-32 page translation

32-bit linear address → 32-bit physical address

Page Size Extensions (PSE)

Bit 4 of CR4

Enables large page sizes

4 MB pages

2 MB pages (with PAE flag set)

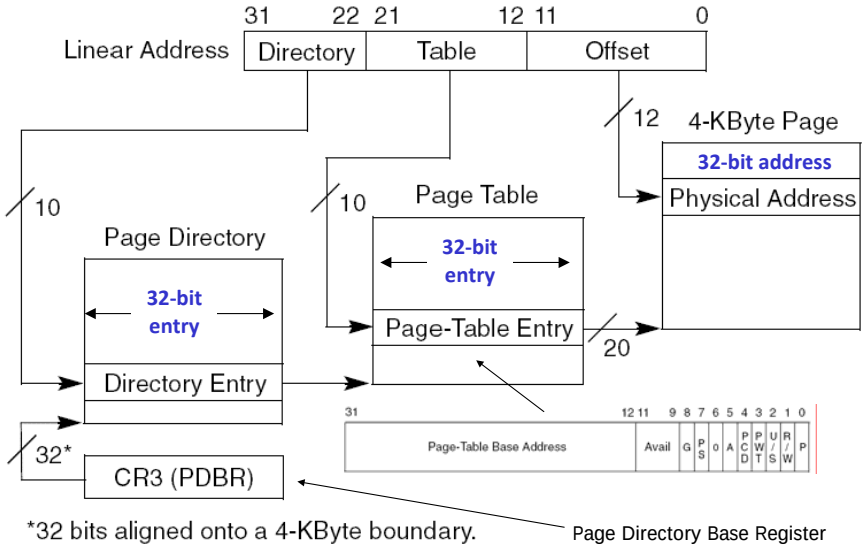
Physical Address Extension (PAE)

Bit 5 of CR4

Enables 36-Bit Physical Addressing

32-bit linear address → 36-bit physical address

IA-32 4 KB Paging

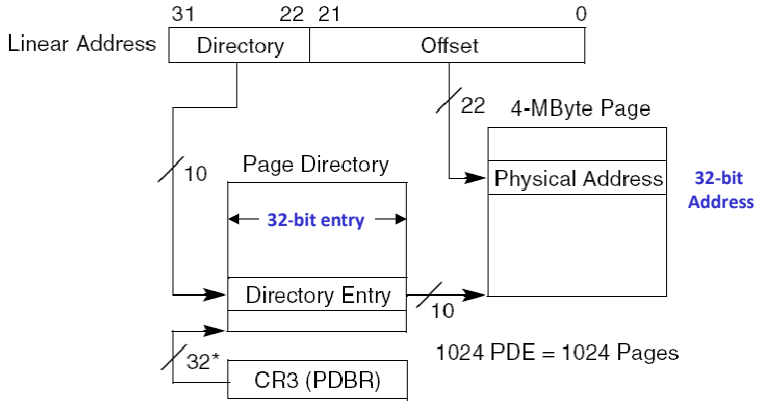


4 MB Paging

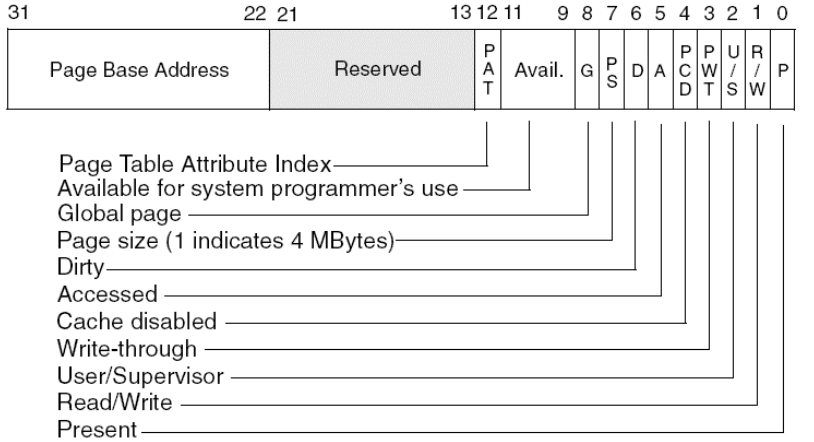
No middle address field

Directory → single page table — 1024 entries

Directory entry → 4 MB page — 22 bit offset into page



4 MB Page Directory Entry



Page Table Attribute Index (PAT)

Field introduced in Pentium III

Enables reference to a table of detailed attribute definitions

P6 Physical Address Extension (PAE)

36-bit physical address

32-bit linear address → 36-bit physical address

4 added address lines on CPU I/O bus

2^{36} Bytes = 64 GB address space

Modified Directory + Page Table structure

Directory Pointer Table (DPT)

2	9	9	12
Pointer	Dir	Table	Offset

Top of table hierarchy

Defines 4 page table directories (first 2 bits in linear address)

Directory and page tables

64 bit entry → 36 bit physical addresses

2^9 = 512 entries / table × 64 bits / entry = 4 KB / table

Page size option

4 KB or 2 MB

PAE — 4 KB Pages

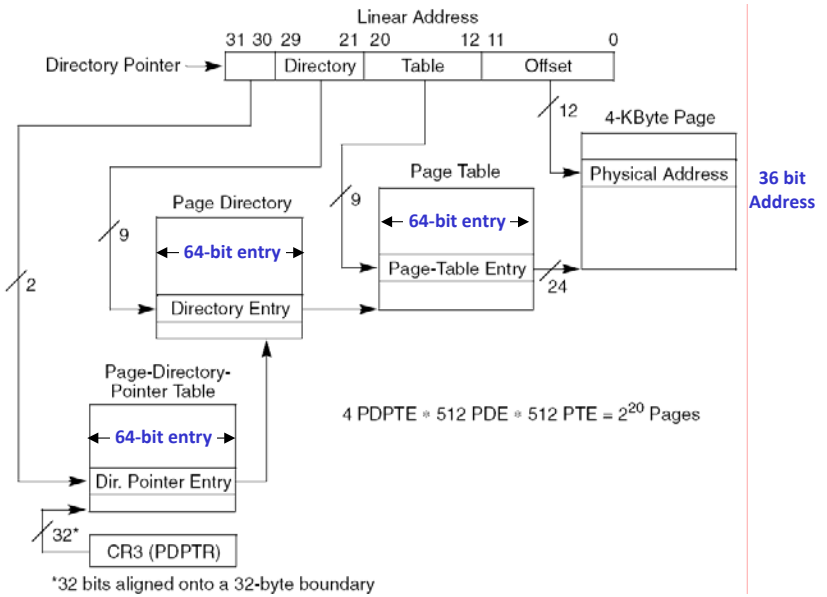
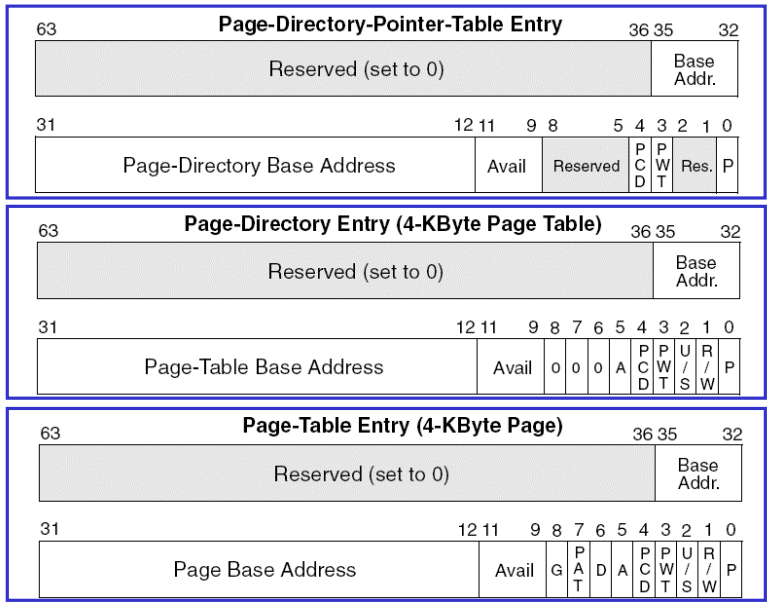


Table Entry — 4 KB Pages



PAE — 2 MB Pages

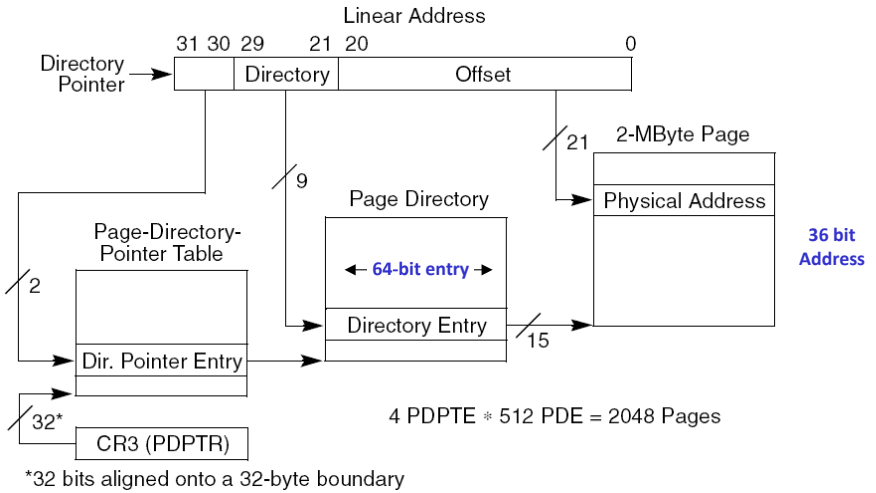
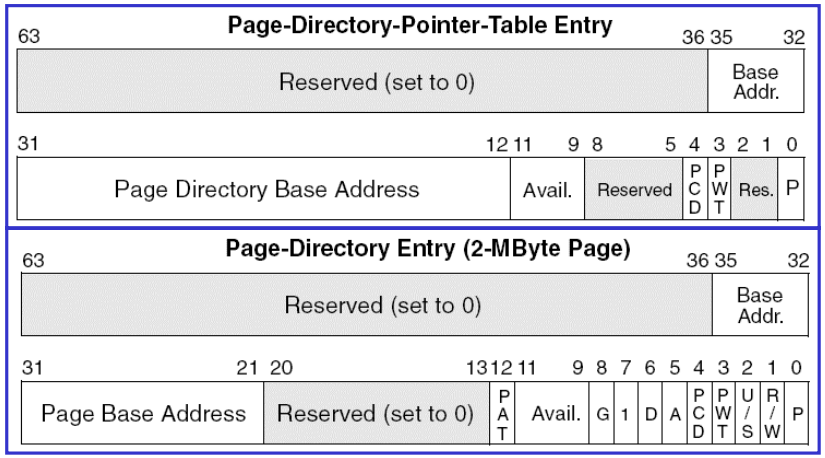


Table Entry — 2 MB Pages



Accessing 64 GB Physical Memory

Page Table structure accesses 4 GB of 64 GB address space

- Directory Pointer Table (DPT) → 2^2 = 4 directories
- Directory → 2^9 = 512 page tables
- Page table → 2^9 = 512 pages
- Page = 2^{12} bytes
- Simultaneous address space = $2^2 \times 2^9 \times 2^9 \times 2^{12}$ bytes = 2^{32} bytes

64 GB address space ⇒ DPT updates

- Address space permits 16 different DPT tables
  - $2^{(36-32)}$ = (64 GB / 4 GB) = 16
- 4 of 64 possible directories "visible" at any time

Accessing additional 4 GB memory sections

- Change base address for DPT
  - New table defines 4 new directories
- Write new entries into DPT
  - Entries point to new directories

New Instructions for IA-32

Instruction	Description
ARPL r/m16,r16	Adjust **RPL** of **r/m16** to not less than **RPL** of **r16**
LAR r16,r/m16	Load Access Rights: **r16** ← **r/m16** masked by **FF00H**
LSL r16,r/m16	Load: **r16** ← segment limit, selector **r/m16**
LSL r32,r/m32	Load: **r32** ← segment limit, selector **r/m32**
SGDT, SIDT m	Store **GDTR** to **m**, Store **IDTR** to **m**
SLDT r/m16	Stores segment selector from **LDTR** in **r/m16**
STR r/m16	Stores segment selector from **Task Register** in **r/m16**
VERR r/m16	Set **ZF=1** if segment specified with **r/m16** can be read
VERW r/m16	Set **ZF=1** if segment specified with **r/m16** can be written
CLTS	Clears **Task Switch** flag in **CR0**
LGDT m16&32	Load **m** into **GDTR**
LIDT m16&32	Load **m** into **IDTR**
LLDT r/m16	Load segment selector **r/m16** into **LDTR**
LTR r/m16	Load **r/m16** into task register

r = register **m** = memory pointer **16/32** = length in bits
r16={AX, CX, DX, BX, SP, BP, SI, DI}
r32={EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI}

Segment-Level Memory Protection

Segment tables "hide" segments from user

- User program knows segment selectors
  - Segment base address hidden in descriptor
- Descriptor hidden in GDT / LDT
  - GDT / LDT access — privileged machine instruction

Local data / code segments

- Defined in LDT
  - LDT selector hidden in Task State Segment (TSS)
- Tasks cannot locate / access
  - TSS, LDT selector, LDT, segment defined in LDT

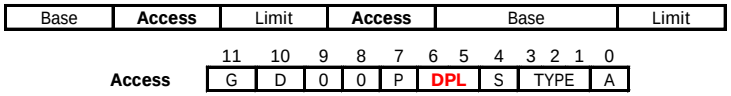
Hardware denies memory access on

- Segment overflow
  - Offset in logical address > segment limit in descriptor
- Action does not match access type in descriptor
  - Write to CS / instruction fetch from DS / user access to system segment
- Insufficient privilege level

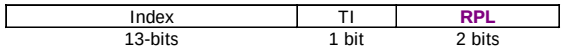
Privilege Rings

Segment access parameters

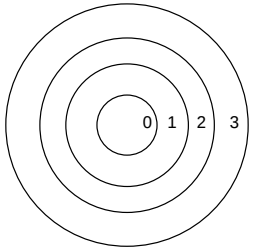
DPL field in segment descriptor



RPL field in segment selector



Ring	Function
0	OS kernel mode
1	Less sensitive OS functions
2	Protected user functions
3	User mode



Most systems use Ring 0 = Kernel Mode and Ring 3 = User Mode

Access Rights by Privilege

Memory access operations by segment type

Data segment access

Code performs load / store instruction

Code segment access

Code performs jump / call / interrupt instruction

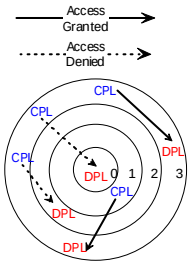
Current code

Selector in CS → descriptor → code segment containing instruction

Access rights

Selector in CS → descriptor → **DPL** in access field

Current Privilege Level (**CPL**) = **DPL** of current CS



Forbidden accesses

Load / store to data segment with **DPL < CPL**

Jump / call to code segment with **DPL < CPL**

System Calls

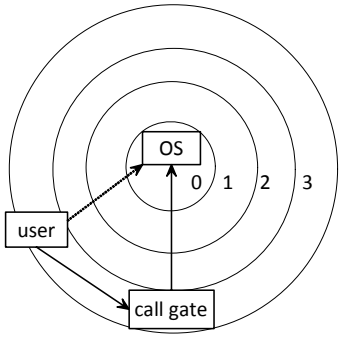
OS code

Runs at **DPL** = 0

User code

Runs at **CPL** = 3

Cannot jump or call OS code directly



Gate mechanism

OS advertises **CS:EIP** for system call

CS call points to special descriptor in GDT

Similar mechanisms for system call / interrupts / task switch

System call

User calls **CS:EIP**

CS = selector → descriptor = Call Gate

Call Gate completes system call

Gate Type System Segments (S = 0)

Defines indirect access

Selector → Gate in descriptor table

Instead of normal descriptor

Gate provides new logical address **SEG:OFFSET**

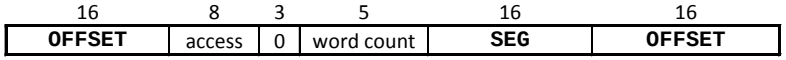
Gate privilege permits ring 0 kernel access

Word Count

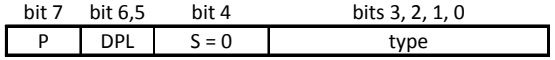
Privileged Stack — user stack segment reserved for system calls

Gate call copies 32-bit words from User Stack to Privileged Stack

Gate Format

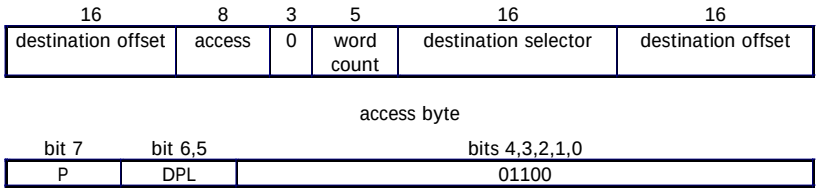


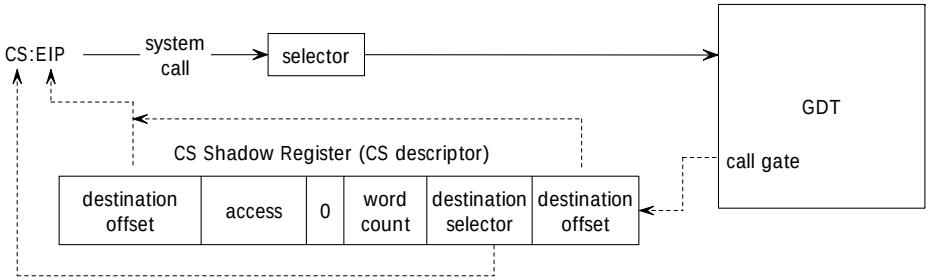
Access



Gate	Type Field
Call Gate	1100
Interrupt Gate	0110
Task Gate	0101

Call Gate





The Trojan Horse Problem

Problem

User program denied access to protected segment

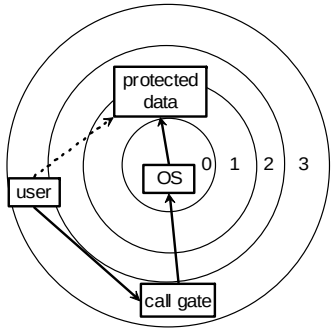
data **DPL** < user **CPL**

User program performs system call

Passes segment selector to OS as pointer

OS accesses protected data segment

(data **DPL** ≥ OS **CPL**)



Solution

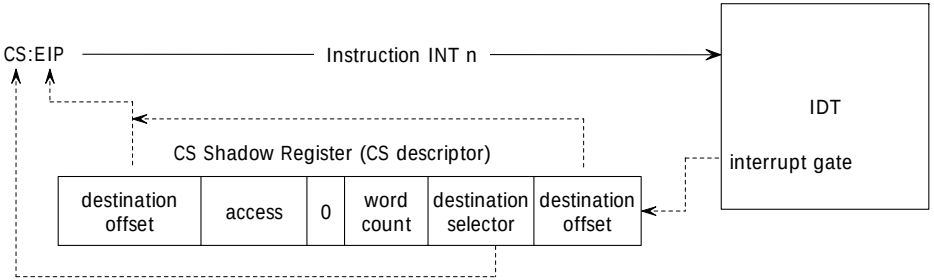
Request Protection Level (**RPL**) field in selector

OS adjusts selector passed by user program

Sets **RPL** = user **CPL**

Access permitted iff **DPL** ≥ max (**CPL** , **RPL**)

Interrupt Service



Execute INT n

CS:EIP of next instruction pushed onto stack

Interrupt Gate

Address = IDT base + n × 8

Loaded to CS Shadow Register

Selector:offset from Interrupt Gate loaded to **CS:EIP**

CS:EIP = address of ISR (interrupt handler)

ISR finishes with **IRET** → pop previous **CS:EIP**

Hardware Task Creation

Write into Task State Segment (TSS)

back link
stacks and stack pointers for CPL = 0, 1, 2 task switch to higher DPL → switch to separate stack
CR3
EIP
EFLAGS
EAX, ECX, EBX, EDX, ESP, EBP, ESI, EDI
ES, CS, SS, DS, FS, GS
LDT Selector
OS specific information

Write TSS descriptor

Normal descriptor entered into GDT / LDT

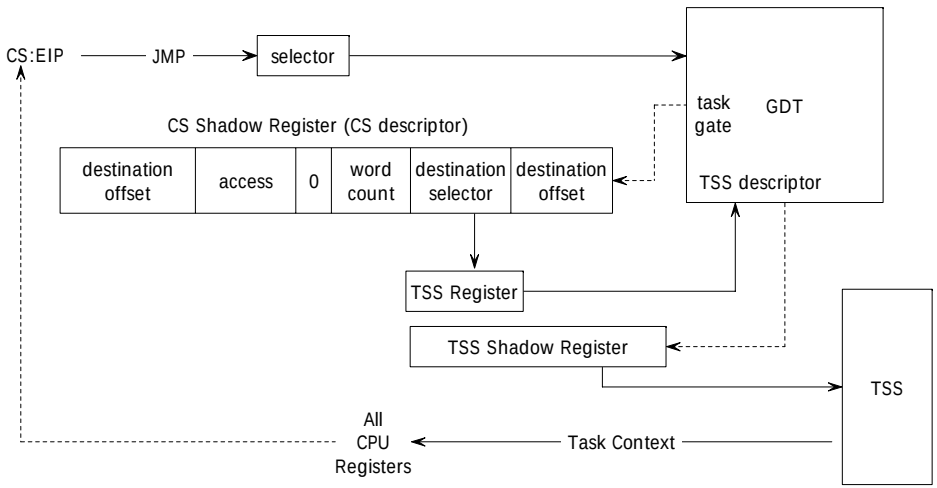
Points to TSS for task

Write Task Gate

Gate descriptor entered into GDT / LDT / IDT

Destination selector points to TSS descriptor in GDT / LDT

Task Switching by Jump



Task Switching by Jump

No nesting

Back link not set

Current code executes JMP to CS:EIP

CS selector → Task Gate in GDT / LDT

Descriptor (Task Gate) loaded to CS Shadow Register

CPU

Recognizes descriptor = Task Gate

- Copies context of old task to old TSS
- Loads Destination Selector from Task Gate → TSS Register

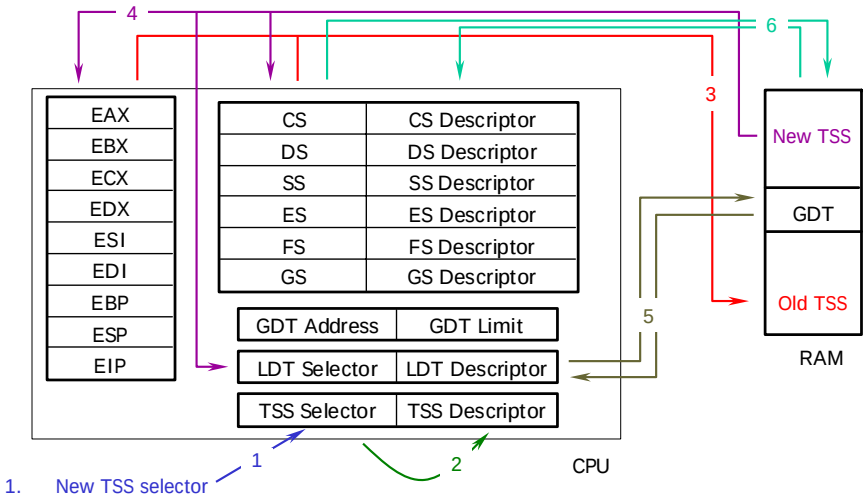
Selector in TSS Register → TSS descriptor

- Loads TSS descriptor to TSS Shadow Register

Loads new context from new TSS

Runs new task from CS:EIP from new task context

Context Switch



1. New TSS selector
2. TSS descriptor auto-update
3. Auto-save old context
4. Auto-load new context (values for LDT, segment registers, general registers)
5. Auto-update shadow registers for LDT
6. Auto-update shadow registers for CS, DS, SS, ES, FS, GS

Task Switching by Call / Return Instruction

Current code executes CALL to CS:EIP

Push CS:EIP of next instruction onto stack

CS selector → Task Gate in GDT / LDT

Descriptor (Task Gate) loaded to CS Shadow Register

CPU

Recognizes descriptor = Task Gate

- Copies context of old task to old TSS
- Writes old TSS Selector → back link of new TSS
- Loads Destination Selector → TSS Register

Selector in TSS Register → TSS descriptor

- Loads TSS descriptor to TSS Shadow Register
- Loads context from new TSS to run called task

Called task ends with IRET (or preemption)

- Copies context of new task to new TSS
- Loads back link → TSS Register

Selector in TSS Register → old TSS descriptor

- Loads old TSS descriptor → TSS Shadow Register
- Loads context from old TSS → restore old task

Real Mode

Start up mode for IA 32 processor

Processor runs like fast 8086

Access only lowest 1 MB of memory

OS boot code must be in low memory

Create pseudo-descriptors in shadow registers

Base Address field ← Selector × 10h

Limit ← FFFFh

CS access word

G	D	0	0	P	DPL	S	CODE	C	R	A
0	0	0	0	1	00	1	1	0	1	1

DS, ES, FS, GS access word

G	0	0	0	P	DPL	S	CODE	ED	W	A
0	0	0	0	1	00	1	0	0	1	1

SS access word

G	0	0	0	P	DPL	S	CODE	ED	W	A
0	0	0	0	1	00	1	0	1	1	1

Before Switching To Protected Mode

OS starts in real mode

Uses 8086 mechanisms

Build GDT

At least one Data Segment descriptor

At least one Code Segment descriptor

Build IDT

Convert 32-bit 8086 ISR vectors to 64-bit ISR descriptors

Build TSS for OS scheduler

Put Task Gate for TSS into GDT

Build Page Tables and Directory

Linear Address = Physical Addresses

Write Directory Physical Address into TSS

Load GDT register and IDT register to CPU

Entering Protected Mode

Set flag PE in CR0

Enter protected mode

JMP to Task Gate in GDT

Loads Task Register

Selector points to TSS Descriptor

CPU loads scheduler context from TSS

Set flag PG in CR0

Enable paging (optional)

OS scheduler now running in protected mode with paging

OS creates processes by writing

TSS

GDT entries

Instruction Types

New instruction encoding for IA-32

Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate
Up to four prefixes of 1-byte each (optional)	1 or 2 byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes or none	Immediate data of 1, 2, or 4 bytes or none

Instruction prefix changes width of default instruction

Code Type	Operand Width		Address Width	
16-bit code	No Prefix	0x66	No Prefix	0x67
	16 bits	32 bits	16 bits	32 bits
32-bit code	No Prefix	0x66	No Prefix	0x67
	32-bits	16 bits	32-bits	16 bits

Example for 16-bit code

With prefix **66B844332211** → `mov eax,0x11223344`

Without prefix **B844332211** → **B84433** → `mov ax,0x3344`

1122 → `and dl,[bx+di]`

Example Code

```
      ORG 0x100
section .text
      mov eax,11223344h
      push eax
      pop ebx
      call disp32
      mov ebx,55667788h
      call disp32
      mov ax,4C00h
      int 21h

disp32:
      mov cx,08h  ; counter = 8
      mov ah,02h  ; DOS function is print byte
nibble:
      rol ebx,4   ; move most significant nibble to least
      mov dl,bl   ; load BL to print buffer
      and dl,0fh  ; zero upper nibble
      add dl,30h  ; ASCII digit range
      cmp dl,39h  ; is nibble in [A-F]
      jle go      ; if not > 9 print
      add dl,7h   ; if > 9 ASCII letter range
go:   int 21h     ; print the byte
      loop nibble ; CX-- and continue
      mov dl, 0dh ; CR
      int 21h
      mov dl, 0ah ; LF
      int 21h
      ret
```


Disassemble

```
00000000  66B844332211  mov eax,0x11223344
00000006  6650          push eax
00000008  665B          pop ebx
0000000A  E80E00        call 0x1b
0000000D  66BB88776655  mov ebx,0x55667788
00000013  E80500        call 0x1b
00000016  B8004C        mov ax,0x4c00
00000019  CD21          int 0x21
0000001B  B90800        mov cx,0x8
0000001E  B402          mov ah,0x2
00000020  66C1C304      rol ebx,0x4
00000024  88DA          mov dl,bl
00000026  80E20F        and dl,0xf
00000029  80C230        add dl,0x30
0000002C  80FA39        cmp dl,0x39
0000002F  7E03          jng 0x34
00000031  80C207        add dl,0x7
00000034  CD21          int 0x21
00000036  E2E8          loop 0x20
00000038  B20D          mov dl,0xd
0000003A  CD21          int 0x21
0000003C  B20A          mov dl,0xa
0000003E  CD21          int 0x21
00000040  C3            ret
```


Example of 32-bit Address Overrides

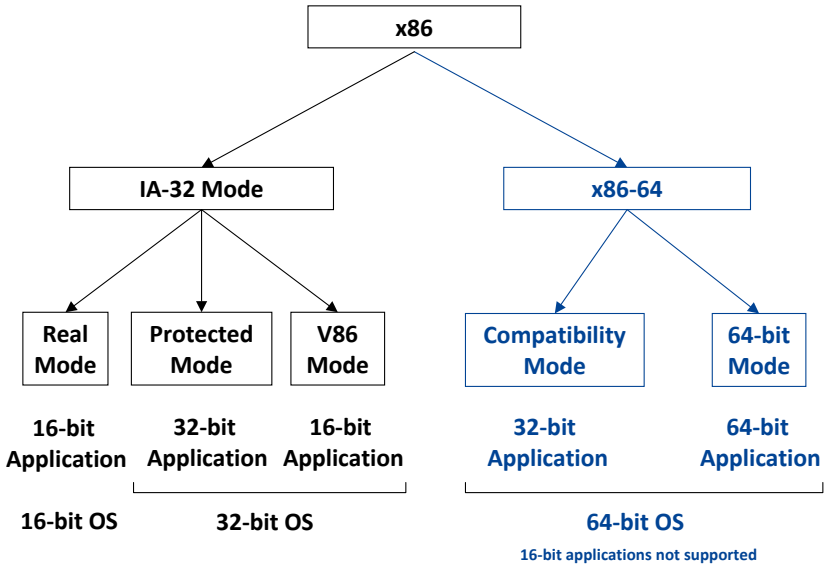
```
            ORG 0x100
        section .data
            filename        db "test.txt",0
        section .bss
            handle          resw 1
        section .text
            mov eax,'abcd'
            mov ebx,'ABCD'
            mov ebp,2000h
            mov [ebp],eax
            mov [ebp+4],ebx
create:     mov dx,filename         ; point to file name
            mov cx,0h               ; default attributes
            mov ah,3ch              ; DOS create file
            int 21h                 ; DOS system call
            jc end                  ; stop on error
            mov [handle],ax         ; store file handle
write:      mov bx,[handle]         ; copy file handle to BX
            mov cx,8h               ; write 8 bytes to file
            mov edx,ebp             ; point EDX to buffer
            mov ah,40h              ; DOS write to file
            int 21h                 ; DOS system call
            jc end                  ; stop on error
close:      mov bx,[handle]         ; copy file handle to BX
            mov ah,3eh              ; DOS close file
            int 21h                 ; DOS system call
end:        mov ax,4C00h            ; return to DOS
            int 21h                 ; DOS system call
```


Assembler Output

```
00000100  66B861626364      mov eax,0x64636261
00000106  66BB41424344      mov ebx,0x44434241
0000010C  66BD00200000      mov ebp,0x2000
00000112  6667894500        mov [ebp+0x0],eax
00000117  6667895D04        mov [ebp+0x4],ebx
0000011C  BA4801            mov dx,0x148
0000011F  B90000            mov cx,0x0
00000122  B43C              mov ah,0x3c
00000124  CD21              int 0x21
00000126  721B              jc 0x43
00000128  A35401            mov [0x154],ax
0000012B  8B1E5401          mov bx,[0x154]
0000012F  B90800            mov cx,0x8
00000132  6689EA            mov edx,ebp
00000135  B440              mov ah,0x40
00000137  CD21              int 0x21
00000139  7208              jc 0x43
0000013B  8B1E5401          mov bx,[0x154]
0000013F  B43E              mov ah,0x3e
00000141  CD21              int 0x21
00000143  B8004C            mov ax,0x4c00
00000146  CD21              int 0x21
00000148  7465              jz 0xaf
0000014A  7374              jnc 0xc0
0000014C  2E7478            cs jz 0xc7
0000014F  7400              jz 0x51
```

```
C:\nasm\programs>type TEST.TXT
abcdABCD
```

Operating Modes for Intel x86 Processors



Why 64 bits?

Features of true 64-bit architecture

- 64-bit ALU integer operands
- 64-bit general purpose register set
- 64-bit flat virtual address space

Advantages of 64-bit architecture

- Huge virtual address space
  - 2^{64} Bytes = $2^4 \times (2^{30})^2$ = 16 Giga-GB = 16 Exa-Bytes
  - Serve many users accessing huge data bases
- Perform high precision arithmetic efficiently
  - 64-bit integer ALU and 128-bit long ALU operations
  - Perform scientific and CAD/CAM/CAE calculations

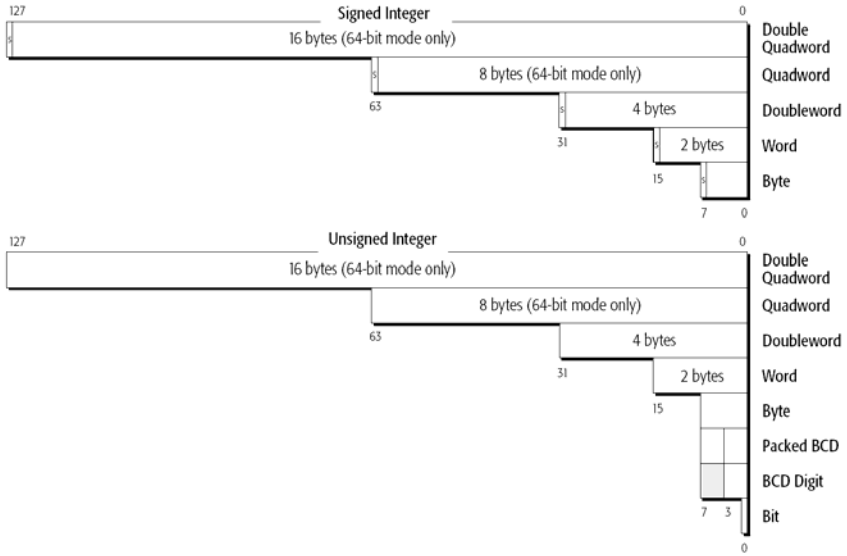
Examples of true 64-bit architecture

PowerPC, Sparc, Alpha, IA-64 (Itanium)

Metric Prefixes

Kilo	K	10^3	2^{10}	1,024
Mega	M	10^6	2^{20}	1,048,576
Giga	G	10^9	2^{30}	1,073,741,824
Terra	T	10^{12}	2^{40}	1,099,511,627,776
Peta	P	10^{15}	2^{50}	1,125,899,906,842,624
Exa	E	10^{18}	2^{60}	1,152,921,504,606,846,976

Data Types



Operand Ranges

Data Type	Byte	Word	Doubleword	Quadword	Double Quadword
Signed Integers	-2^7 to $+(2^7-1)$	-2^{15} to $+(2^{15}-1)$	-2^{31} to $+(2^{31}-1)$	-2^{63} to $+(2^{63}-1)$	-2^{127} to $+(2^{127}-1)$
Unsigned Integers	0 to $+2^8-1$ (0 to 255)	0 to $+2^{16}-1$ (0 to 65,535)	0 to $+2^{32}-1$ (0 to 4.29×10^9)	0 to $+2^{64}-1$ (0 to 1.84×10^{19})	0 to $+2^{128}-1$ (0 to 3.40×10^{38})
Packed BCD Digits	00 to 99	multiple packed BCD-digit bytes			
BCD Digit	0 to 9	multiple BCD-digit bytes			

How to Think About x86-64 ?

x86-64 not true 64-bit architecture

Optimized for default 32-bit integer

- 64-bit integer operations by override

Optimized for default 32-bit register accesses

- 64-bit register accesses by override

64-bit virtual address space

- "Tricks" standard IA-32 segmentation system

Why x86-64 — Intel

Easy migration path from IA-32 to 64-bits

Provides some 64-bit features

Preserves IA-32 Instruction Set Architecture (ISA)

Preserves most IA-32 software in most circumstances

Preserves IA-32 "knowledge base"

What Intel Said

The move toward 64-bit computing for mainstream applications will initially focus on applications that are already **constrained** by 32-bit **memory limitations**.

The challenge for IT organizations is to determine the best architecture for specific solutions, while taking into account total **cost** and **value** within the broader IT and business environments.

Itanium architecture remains the platform of **choice** for the **most demanding**, business-critical data tier applications, such as **high-end database** and business intelligence solutions.

Platforms based on the Intel **Xeon** processor with Intel **EM64T** are preferable for general purpose applications, such as **Web** and **mail** infrastructure, digital **content** creation, mechanical **computer aided design**, and electronic design **automation**; and for mixed environments in which **optimized 32-bit performance** remains critical.

The 64-bit Tipping Point, September 2004

64-bit Extensions

Operands

IA-32 registers → 64-bit width

EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP →
RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, RIP

8 new general purpose registers (GPR)

R8, R9, ... , R15

Default 32-bit integer operand

- Override → 64-bit integer ALU operations

Address

64-bit flat linear address space

- Logical address = SEG:OFFSET
- SEG CS, DS, ES, SS → physical base address = 0
- 64-bit OFFSET = Linear Address
- Segmentation enforces protection

64-bit paging

52-bit physical address

Summary of Operating Modes

Operating Mode		Operating System	Defaults		Register Extensions	Typical
			Address Size (bits)	Operand Size (bits)		GPR Width (bits)
x86-64	64-Bit Mode	64-bit OS	64	32	yes	64
	Compatibility Mode		32		no	32
			16	16		16
IA-32	Protected Mode	32-bit OS	32	32	no	32
			16	16		16
	Real Mode	16-bit OS				

Booting 64-bit OS

Initialize CPU into real mode
Switch CPU to 32-bit protected mode
Switch CPU to 64-bit mode
OS runs
- 64-bit applications
- 32-bit applications (in compatibility mode)

64-bit Applications — Typical Features

Code fetch

CS → code segment base address = 0
64-bit instruction pointer **RIP**
Linear Address = **RIP**

IA-32 instruction syntax

PUSH, POP, MOV, ADD, SUB, ... work as usual
Most instructions use 32-bit operands

```
MOV EAX, 11223344h
ADD EAX, [EBX+ESI+11223344]
```

64-bit virtual address

ADD EAX, [EBX+ESI+11223344]
Logical Address = **DS:EBX+ESI+11223344**
- **DS** → data segment base address = 0
- Sign extend EA = **EBX+ESI+11223344** → 64-bit EA_{64}

Linear Address = EA_{64}

IA-32 Prefixes

IA-32 code prefixes — override default parameters

66H
- 16-bit code — 16-bit operand → 32-bit operand
- 32-bit code — 32-bit operand → 16-bit operand

67H
- 16-bit code — 16-bit address offset → 32-bit address offset
- 32-bit code — 32-bit address offset → 16-bit address offset

Example

16-bit code fragment

```
66B861626364        mov eax,0x64636261
6667894500          mov [ebp+0x0],eax
6667895D04          mov [ebp+0x4],ebx
```

32-bit code fragment

```
66B86162            mov ax,0x6261
```


Prefixes for Instruction Encoding

General instruction encoding

Legacy prefixes	REX prefix	Opcode	ModR/M	SIB	Displacement	Immediate

Legacy prefix = IA-32 prefix
ModR/M = mod-reg-r/m
SIB = scale-index-base
- Effective Address = base + scale * index + displacement

REX prefixes

Override default operand / address size
Combined with 66H and 67H codes
- W — operand width
- R — register (in ModR/M)
- X — index (in ModR/M)
- B — base (in ModR/M)

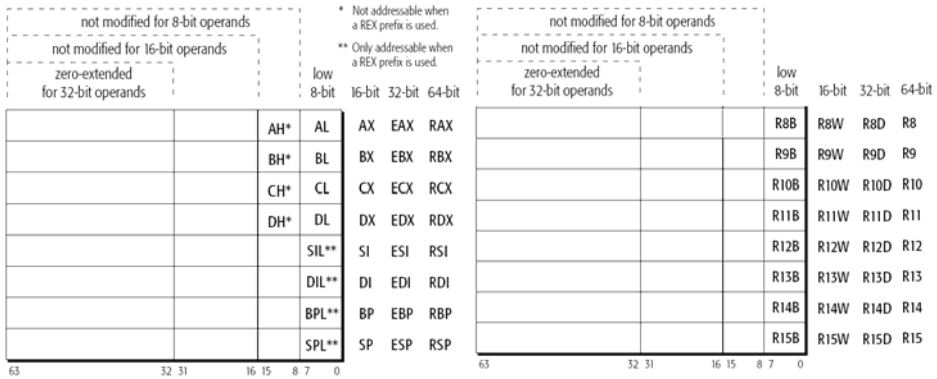
4	1	1	1	1
0100	W	R	X	B

IA-32 and REX Overrides

Mode	Default Address Size (bits)	Linear Address Size (bits)	Prefix
64-bit	64	64	—
		32	0x67
Compatibility	32	32	—
		16	0x67
	16	32	0x67
		16	—

Mode	Default Operand Size (bits)	Effective Operand Size (bits)	Instruction Prefix	
			IA-32 Prefix	REX.W
64-bit	32	64	Ignore	1
		32	—	0
		16	66H	
Compatibility	32	32	—	—
		16	66H	
	16	32	66H	
		16	—	

Register Access



Register accesses

- 64-bit operations access entire register
- 32-bit operations access lower 32-bits of 64-bit registers (default)
- 16-bit operations access lower 16-bits of 64-bit registers (where permitted)
- 8-bit operations access lower 8-bits of 64-bit registers
  - Access lower 8-bits of legacy base/index registers

Instructions and Operands

Default effective address is 64 bits

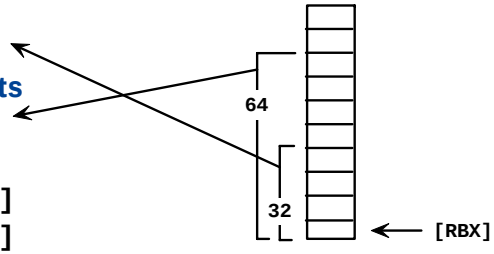
```
EA = Base + Scale * Index + Displacement
```

Default operand — 32 bits

```
ADD EAX,[RBX]
```

REX override operand → 64 bits

```
ADD RAX,[RBX]
```



Most displacements — 32 bits

```
ADD EAX,[RSI+11223344]
ADD RAX,[RSI+11223344]
```

Special form of MOV — 64-bit absolute address

```
MOV RAX,[1122334455667788]
```

Most immediates — 32 bits

```
ADD EAX,11223344
```

Sign extended immediates for 64-bit operation

```
ADD RAX,11223344
```

Special form of MOV — 64-bit immediate

```
MOV RAX,1122334455667788
```


Other Instructions

Branch

- Near branch — default target address = 64 bits `JMP targ`
- Far branch — use indirect target `JMP [pointer]`
- Loop instructions check **RCX**

Stack instructions

- Operand — 64 bits

```
PUSH RAX
```

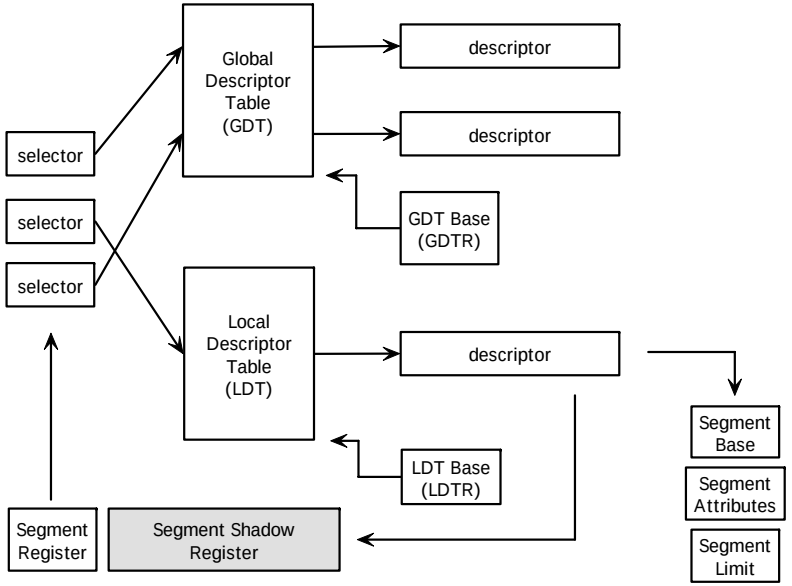
New addressing mode (in kernel mode)

- **RIP**-relative
- **EA** = 64-bit **RIP** + 32-bit displacement

String instructions

```
LODSQ          ; RAX ← [DS:RSI] , RSI ← RSI + 8
STOSQ          ; [ES:RDI] ← RAX , RDI ← RDI + 8
MOVSQ
CMPSQ
```

Segmentation Model



Segmentation in x86-64

Segment selectors

As in IA-32

Descriptor tables

Direct descriptor table registers
- Global Descriptor Table Register (GDTR)
- Local Descriptor Table Register (LDTR)
- Interrupt Descriptor Table Register (IDTR)

Located at 64-bit linear base address

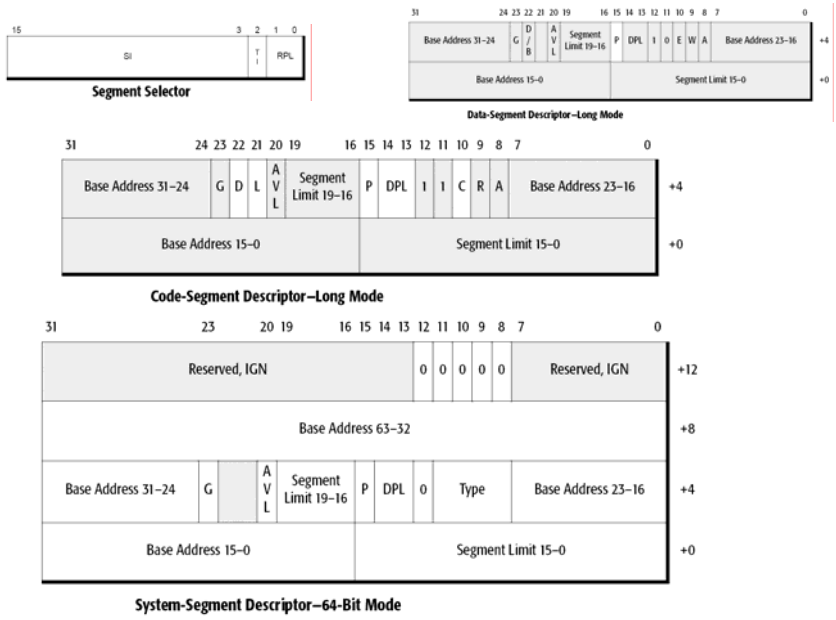
10-byte registers
- 64-bit table base address (8 bytes)
- 16-bit table limit (2 bytes)

Descriptors

User descriptor structure identical to IA-32

System descriptors expanded to 128 bits

Selector/Descriptor Formats



Segmentation Process in 64-bit Mode

Code segment

Load selector to CS register — selector points to descriptor
- Load descriptor to shadow register

Bit L = 1 ⇒ 64-bit mode (L = 0 ⇒ compatibility mode)

Check DPL = current privilege level

Other descriptor fields ignored

Data, stack, and extra segments

No selector or descriptor loaded

No attribute checking

FS and GS

Load selector to FS/GS register — selector points to descriptor
- Load descriptor to shadow register
- Shadow register expanded → 64-bit segment base address

Other descriptor fields ignored

Canonical Virtual Address

Virtual (linear) address

Maximum address length = 64 bits

Minimum implemented address length = 48 bits

2^{48} bytes = 256 × 2^{10} × 2^{30} = 256 Mega-GB = 256 TB

Procedures for address longer than 48 bits are defined by Intel or AMD

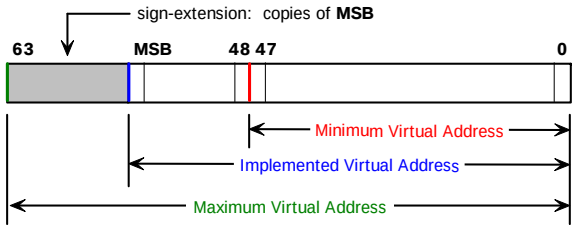
Canonical Form

Bit MSB+1 to bit 63 = copy of MSB

Sign-extended format

Splits address space into "positive" and "negative" sectors

Used by OS for system management



Physical Address Extension (PAE) Paging

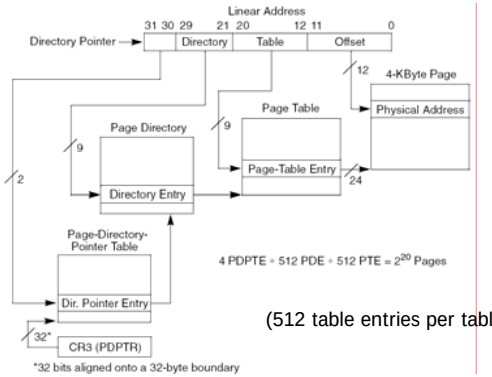
4 level paging system for 36-bit physical address

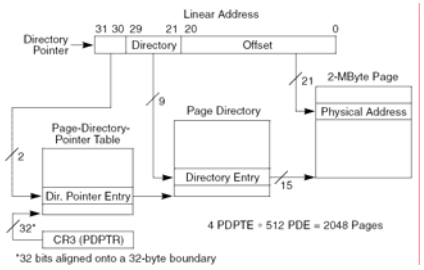
64-bit Directory Pointer Table entry points to Directory

64-bit Directory entry points to Page Table / Page

64-bit Page Table entry points to Page

12/21-bit Offset points to byte in Page





(512 table entries per table) × (64-bits = 8 bytes per entry) = 4 KB / table

64-bit Paging

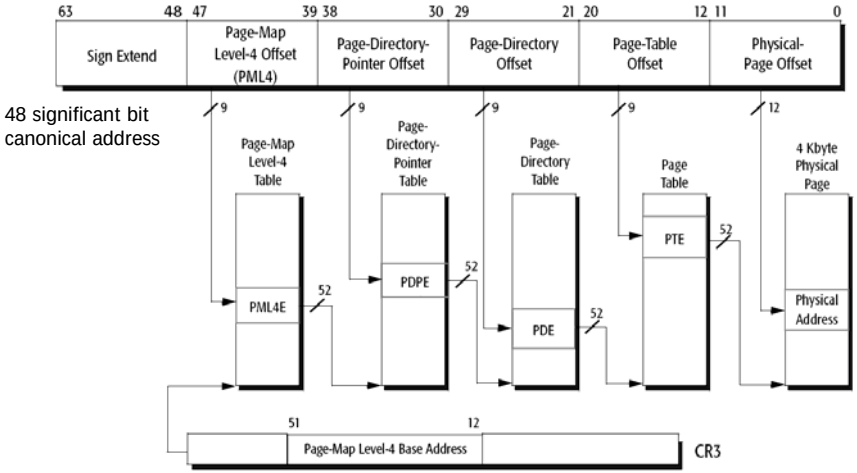


Table structure

Entries as in PAE

512 entries in Pointer Table (4 in PAE)

Added **Page-Map Table** at top of hierarchy defines 512 Pointer Tables

2 MB Page Size in 64-bit Mode

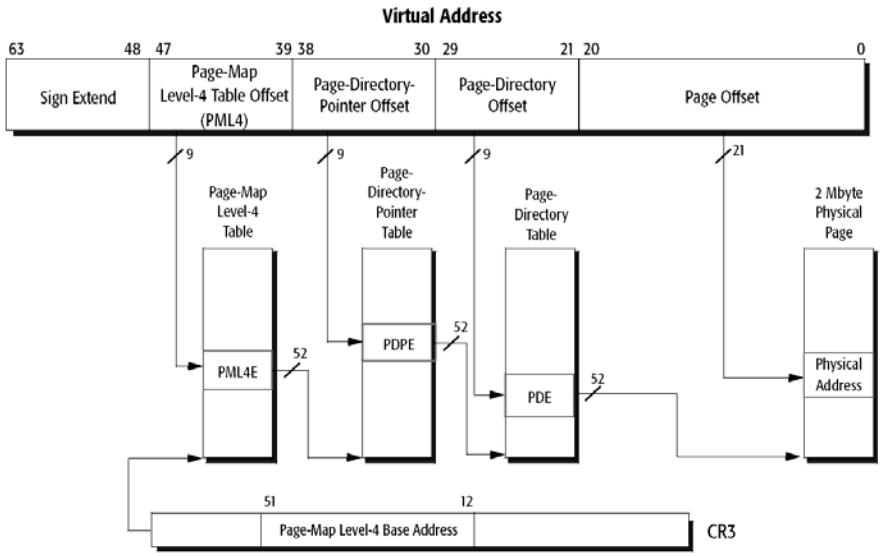


Table Entries

Bit	Page Map	Directory Pointer	Directory (4 KB Page)	Directory (2 MB Page)	Page Table
0	P — Present / Not Present				
1	R/W — Read Only / Writeable				
2	U/S — User / Supervisor				
3	PWT — Page Level Write Through				
4	PCD — Page Level Cache Disable				
5	A — Accessed / Not Accessed				
6	Reserved			D — Dirty	
7	Reserved		0	1	PAT
8	Reserved			G — Global	
9 – 11	Available to OS				
12	Directory Pointer Address	Directory Address	Page Table Address	PAT	Page Address
13				Reserved	
21 – 39				Page Address	
40 – 51	Reserved				
52 – 62	Available to OS				
63	Reserved				

Entering 64-bit Mode

OS running IA-32 protected mode with paging enabled

- Disable paging
- Enable physical address extensions (PAE)
  - Allows 52-bit physical addresses
- Load physical base address of PML4 (top paging table) to CR3
- Enable x86-64 mode
- Enable 64-bit paging

OS now running in 64-bit mode with 64-bit paging

- GDTR, LDTR, IDTR, TR still point to IA-32 descriptor tables
- Disable exceptions and interrupts
- Execute LGDT, LLDT, LIDT, and LTR
  - Load physical base addresses to 64-bit descriptor tables
- Enable exceptions and interrupts

64-bit Mode → Compatibility Mode

No change to

- Segment registers / segment shadow registers
- Descriptor table physical base registers
- Physical base address of PML4 (top paging table)

CPU creates "virtual protected mode" environment

- **CS** descriptor checked for bit L
  - 1 — 64-bit mode
  - 0 — indicates compatibility mode
- Other descriptor fields treated as in IA-32

IA-32 segmentation and paging enabled

- 16-bit / 32-bit address and operand sizes
- Access to lower 4 GB of linear address space

IA-32 instruction prefixes and registers

- 32-bit registers and memory accesses
- REX prefixes ignored

Set Up Compatibility Mode for Application

In 64-bit mode

- Load **DS, ES, SS** with selectors
  - **MOV SREG, source** / **POP SREG, source**
- CPU loads descriptor from GDT / LDT
  - Descriptor base, limit, and attribute loaded to shadow registers

64-bit mode ignores

- Contents of data and stack segment selectors
- Descriptor shadow registers

Call / jump / interrupt / task switch to compatibility mode CS

- CPU loads selector to **CS**
- CPU loads CS descriptor from GDT / LDT
  - Descriptor base, limit, and attribute loaded to shadow register
- **CS.L = 0** ⇒ compatibility mode code segment
  - CPU runs code in compatibility mode

Address Translation in x86-64