

Addressing, Segmentation and Memory Layout

CS 348
Implementation of Programming Languages Lab
Computer Science and Engineering Department
Indian Institute of Technology
Guwahati

Contents

- Address Computation in x86
- Logical Addressing in x86
- Segmentation in 64-bit Mode
- Address Translation Overview
- Program Memory Layout

Why Addressing Matters

- Every instruction touches memory
- Incorrect address will lead to crash, fault, or vulnerability
- Foundation for OS, compilers, and debugging

Address Computation

- Memory operands in x86 use **effective addresses**
- General form:

operation(base, index, scale)

- Effective address computed as:

address = base + (index × scale) + displacement

- All components are optional
- Computation is done by hardware **before** memory access

LEA (Load Effective Address)

- LEA computes an address and stores it in a register
- Syntax:

```
lea src, dest
```

- Source must be a **memory-style operand**
- Destination must be a **register**
- **Does NOT access memory, only computes the address and stores**

```
lea 8(%rax, %rbx, 4), %rcx
```

Why LEA?

- Efficient pointer arithmetic
- Array indexing
- Structure field access
- Arithmetic without affecting flags

Compiler usage:

- Replaces add, mul, shift combinations
- Generates shorter and faster code

MOV vs LEA

MOV

- Transfers **data**
- May **access memory**
- Dereferences the address

mov (%rax), %rbx

MOV moves **values**, LEA computes **addresses**.

LEA

- Computes an **address**
- **Never accesses memory**
- Pure arithmetic operation

lea (%rax), %rbx

Before Virtual Memory

- Early systems did not have virtual memory
- Programs accessed physical memory directly
- No automatic isolation between programs
- Errors could corrupt other programs or the OS

- Need for protection
- Need for structured memory usage

Segmentation

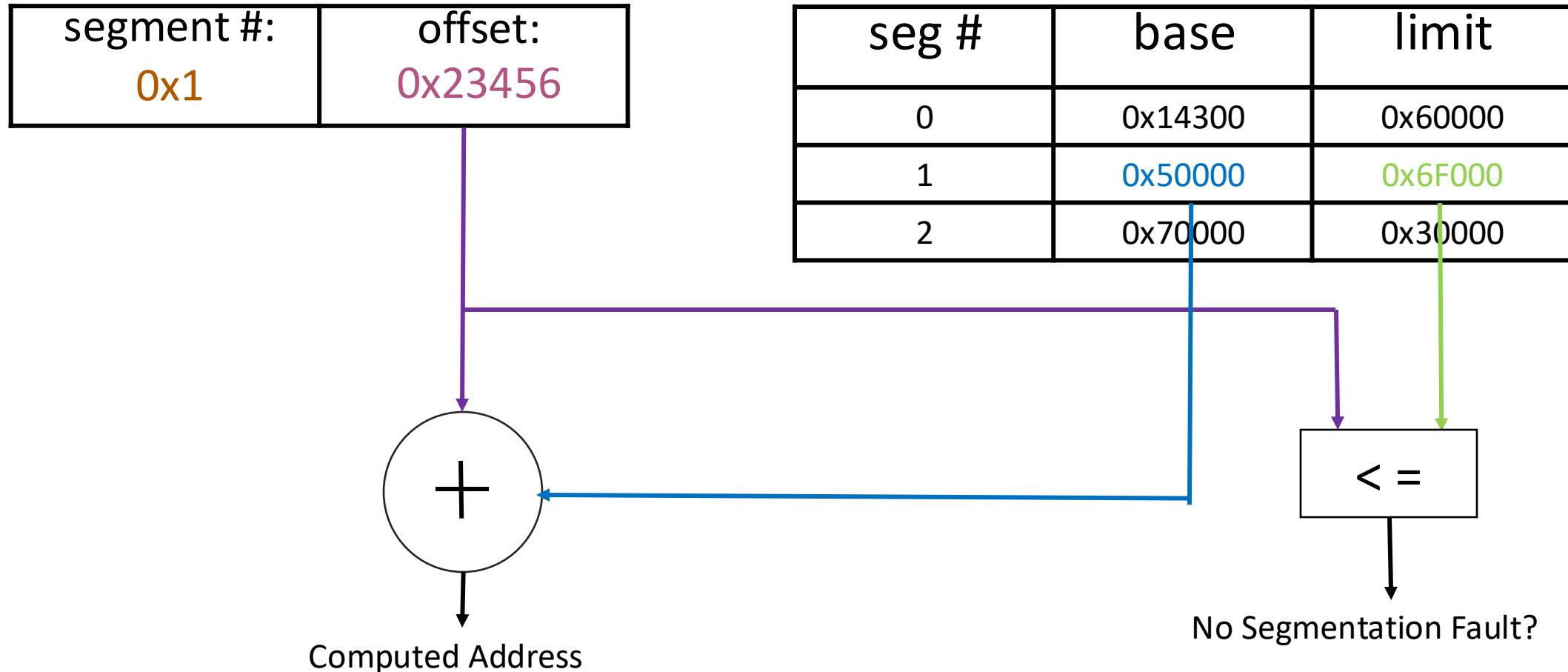
- Memory divided into **segments**
- A logical address consists of:

segment number + offset

- Each segment describes a contiguous memory region
- Segment table maps segment numbers to memory regions

Segmentation

before virtual memory, there was **segmentation**



Segment Registers in x86

Addresses you've seen are the offsets and every access uses a segment number

Segment numbers come from registers

- **CS** – Code Segment number (jump, call, etc.)
- **DS** – Data Segment number (push, pop, etc.)
- **SS** – Stack Segment number (mov, add, etc.)
- **ES** – Extra Data Segment (String instructions)
- **FS, GS** – Additional segments (special use)

Some instructions can have a segment override:

movq \$42, %fs:100(%rsi)

// move 42 to segment (# in FS), offset 100 + RSI

x86_64 segmentation

- Segment limits are ignored
- Segment base addresses are ignored
- Logical addresses behave as a **flat address space**

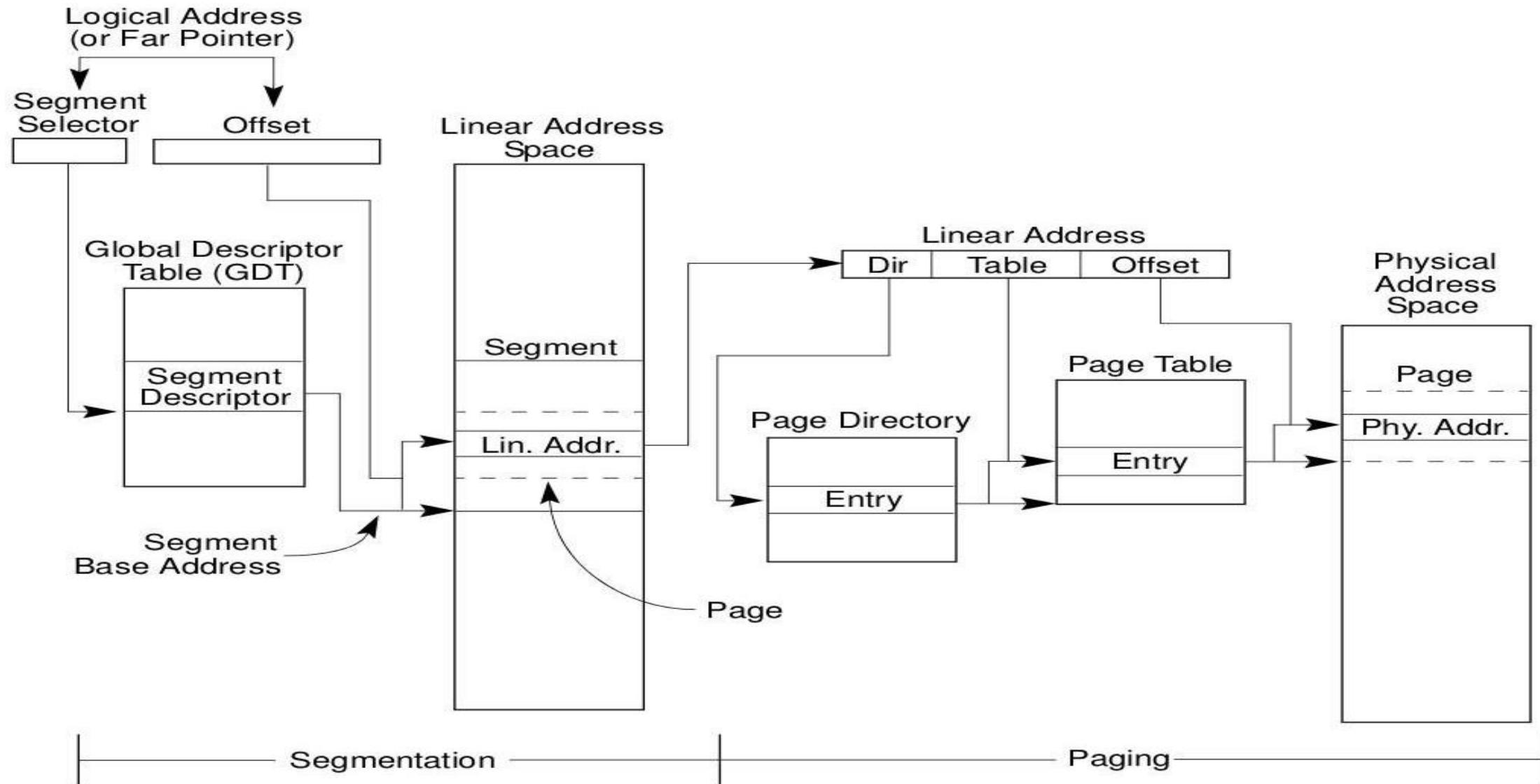
Exceptions:

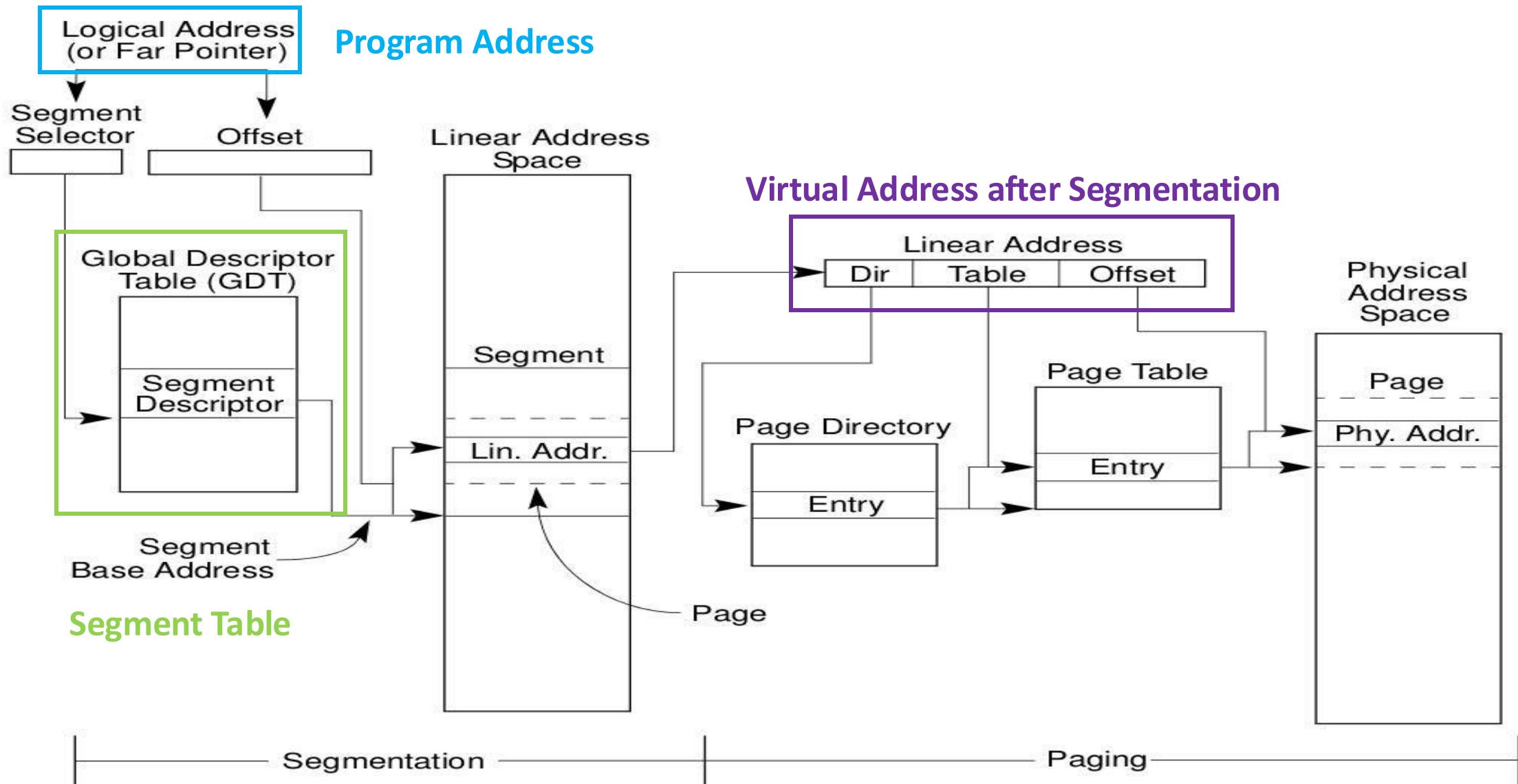
- %fs and %gs retain their base addresses
- Used primarily for thread-local storage (TLS)
- Accessible via explicit segment override

Practical Effect:

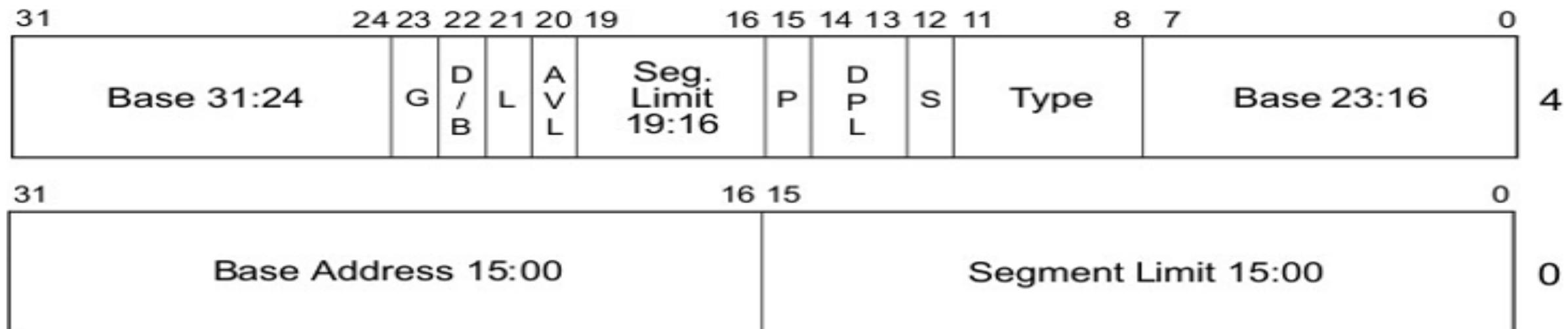
- %fs / %gs act as **implicit pointer registers**
- Commonly used by the OS and runtime libraries

Segmentation and Paging





x86 segment descriptor



L — 64-bit code segment (IA-32e mode only)

AVL — Available for use by system software

BASE — Segment base address

D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)

DPL — Descriptor privilege level

G — Granularity

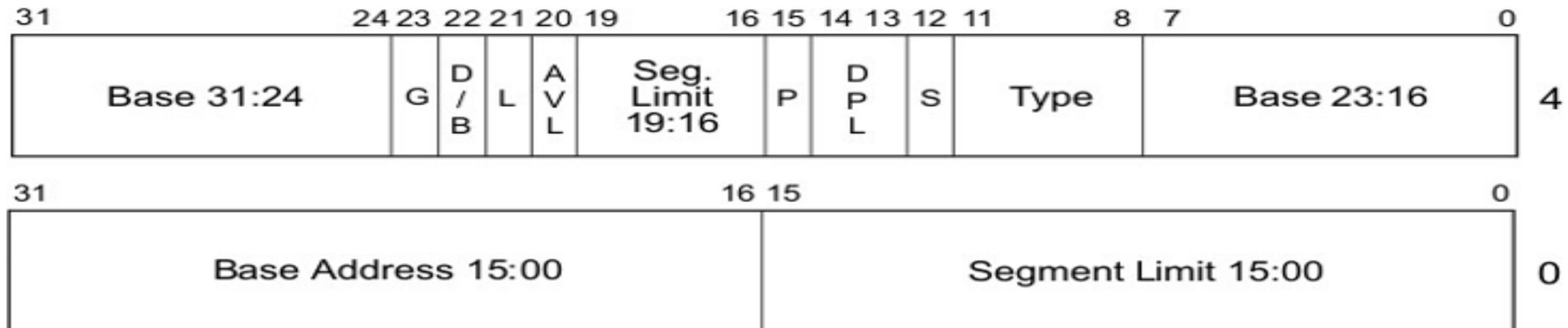
LIMIT — Segment Limit

P — Segment present

S — Descriptor type (0 = system; 1 = code or data)

TYPE — Segment type

x86 segment descriptor



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

Base Address

The **Base Address** defines the **starting linear address** of a segment.
Any memory access using a segment register is **relative to this base**.

Allows programs and OS to **relocate code/data** without changing instructions.
Same code can run at different memory locations by changing only the base.

Split across the descriptor:

Base 15:0

Base 23:16

Base 31:24

CPU internally **reconstructs** the full 32-bit base.

Segment Limit Field (20 bits)

Size Enforcement

Defines the **maximum offset** allowed within a segment.
Prevents access beyond segment boundary.

Provides **hardware-enforced bounds checking**

Stops buffer overflows from crossing segment limits (legacy protection)

Granularity and Maximum Sizes

Granularity interaction

- If $G = 0$:
 - Limit is in **bytes**
- If $G = 1$:
 - Limit $\times 4$ KB

Maximum sizes

- $G = 0 \rightarrow 1$ MB
- $G = 1 \rightarrow \sim 4$ GB

Example

- Limit = $0xFFFF$, $G = 1$
- Effective size ≈ 4 GB

Access Byte Permission & Safety Control

Present Bit (P)

Indicates whether the segment exists in memory.

P = 0 → Access causes **segment not present fault**

Used for:

- Demand loading

- Swapping

Descriptor Privilege Level (DPL)

Values: 0 (kernel) to 3 (user)

Defines **who is allowed to access the segment**

Checked against:

- CPL (Current Privilege Level)

- RPL (Requested Privilege Level)

S Bit (System vs Code/Data)

S = 1 → Code or Data segment

S = 0 → System segment

- TSS (Task State Segment)

- LDT (Local Descriptor Table)

- Call Gates

Why important

System segments follow **different validation rules**

Misuse causes protection fault

D/B (Default Operand Size)

Determines:

Register width

Stack pointer width

D/B = 0 → 16-bit

D/B = 1 → 32-bit

AVL Bit

Ignored by CPU

Reserved for OS use

Often used for:

- Debugging

- Custom state tracking

How CPU Uses the Descriptor (Execution Flow)

- Instruction references memory
- Segment selector is read
- Descriptor fetched from GDT/LDT
- CPU checks:
 - Present bit
 - Privilege level
 - Type
- Offset checked against limit
- Base added to offset
- Linear address produced
- Paging applies (if enabled)