# *Introduction to Memory Architecture*

Reverse Engineering – Introduction to Memory Architecture

Albert López - newlog@overflowedminds.net

# *Goals*

What can you expect after this module?

## Goals

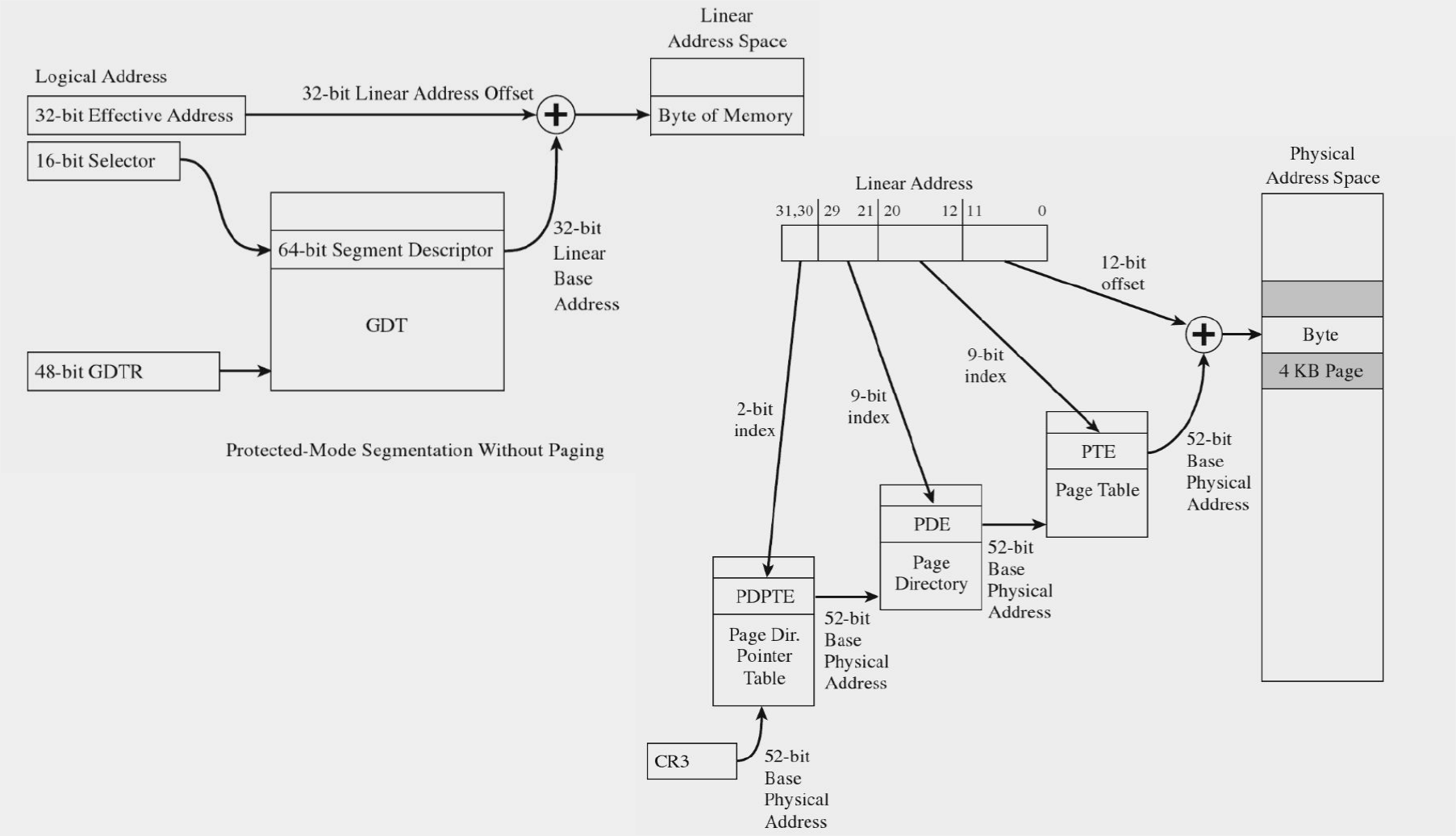
1. Get the big picture around OS and CPU memory management

1. Understand that there is much more happening behind the scenes that we won't see when RE software

1. Get answers to basic questions such as:
   a. Can one process access other processes' memory?
   b. Why two different processes use the same addresses?
   c. How can a process access N GB of memory in x86?
   d. How to map from virtual to physical?

# Goals

At the end of this module, you will understand this:



# *The Basics*

Numerical representations, processors, addresses, etc

# *Numerical Representations*

Hexadecimal Number: 0x**C7**28**32**40 → 16 symbols: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Decimal Number: 3341300288 → 10 symbols: 0,1,2,3,4,5,6,7,8,9

Binary Number: **1100 0111** 0010 1000 **0011 0010** 0100 0000 → 2 symbols: 0,1

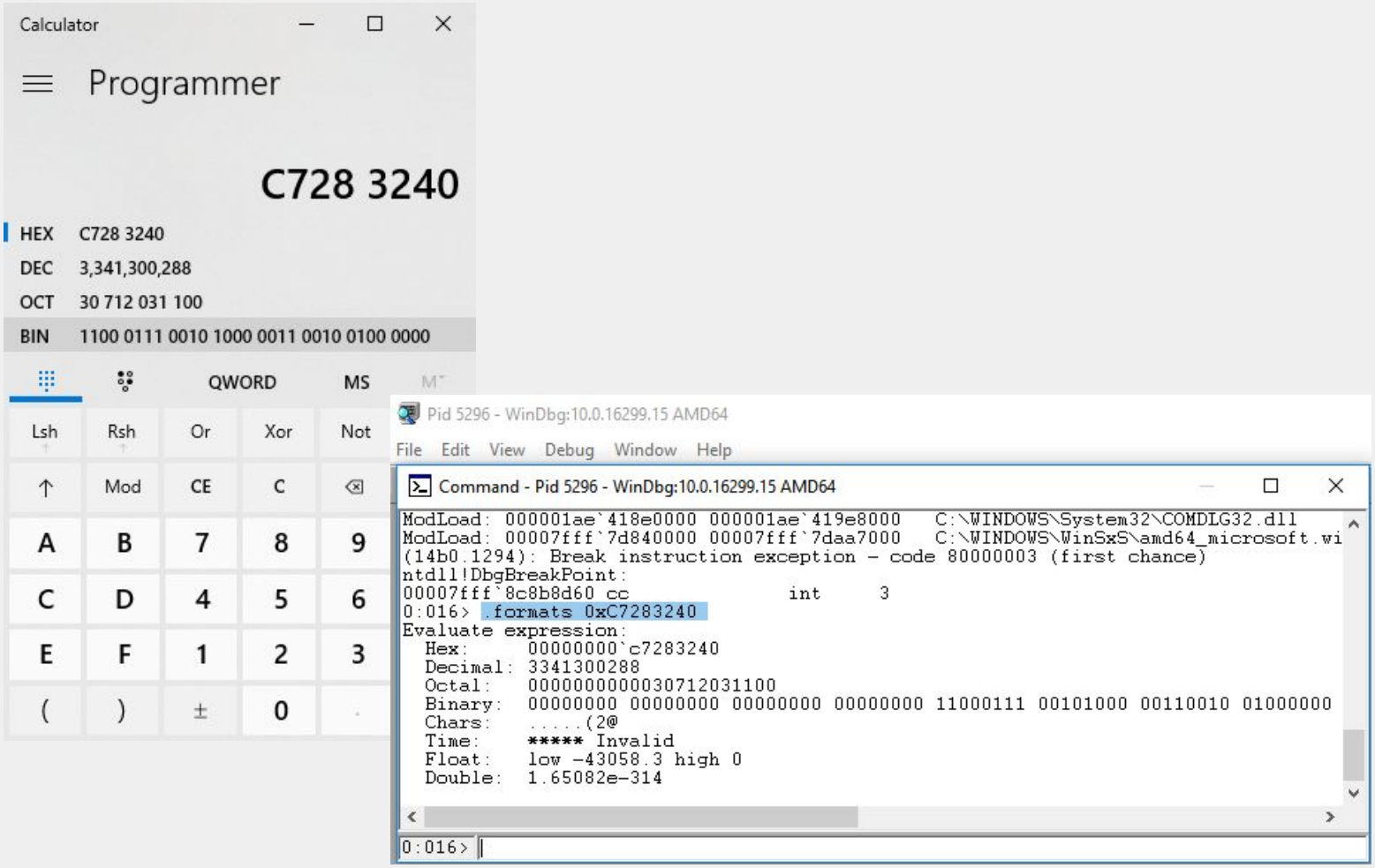
0x C 7 2 8 3 2 4 0

- Software world → Hexadecimal representation

However, many operations related to **memory management** are at the **bit level**.

e.g. Software needs to know if bit N in address X is 1 or 0.

Luckily, you don't need to do that manually!



# *Processors / CPUs*

- **Why we care about the CPU** (Central Processing Unit)?

The CPU is the hardware component within a computer that **executes the instructions** of a computer program.

- Main vendors for desktop systems are **Intel** and **AMD**.
  - Given their market share, both Intel and AMD have compatible **instruction sets**[1].
  - e.g. when source code is compiled, the compiler generates compatible assembler instructions.

[1] Differences between AMD and Intel instruction set (architectures):
https://en.wikipedia.org/wiki/X86-64#Differences_between_AMD64_and_Intel_64
https://stackoverflow.com/questions/38516823/what-exactly-is-the-difference-between-intels-and-amds-isa-if-any

This module **focus** on a subset of the **Intel ISA** (Instruction Set Architecture), specifically, the **IA-32 Memory Management**.

We are going to explain for dummies, what you can find here:

- Intel® 64 and IA-32 Architectures, Software Developer's Manual, Volume 3A: System Programming Guide, Part 1
- 462 pages vs ~100 slides!

We will understand:

- Part of what your operating system does behind the scenes.
- What mechanics are used to store software in memory

A little bit of Intel's CPU history[1]:

- Intel 4004. Year 1971. 8-bit CPU. Intel's first CPU.
- Intel 8086. Year 1976. 16-bit CPU. First implementation of **x86**[2] architecture.
- Intel 80386(DX). Year 1985. 32-bit CPU. 32-bit x86 architecture = **IA-32** **= i386**.
- Itanium. Year 2001. 64-bit CPU. ~~IA-64 architecture~~. You won't hear about it anymore.
- Pentium 4F. Year 2006. 64-bit CPU. **Intel 64** architecture. Also known as IA-32e, x86-64, x64 and AMD64. Backwards compatible with x86.

[1] List of Intel microprocessors:
https://en.wikipedia.org/wiki/List_of_Intel_microprocessors
[2]Known as x86 because many other processor names after the 8086, with the same architecture, ended with "86".

# *IA-32 Memory Management*

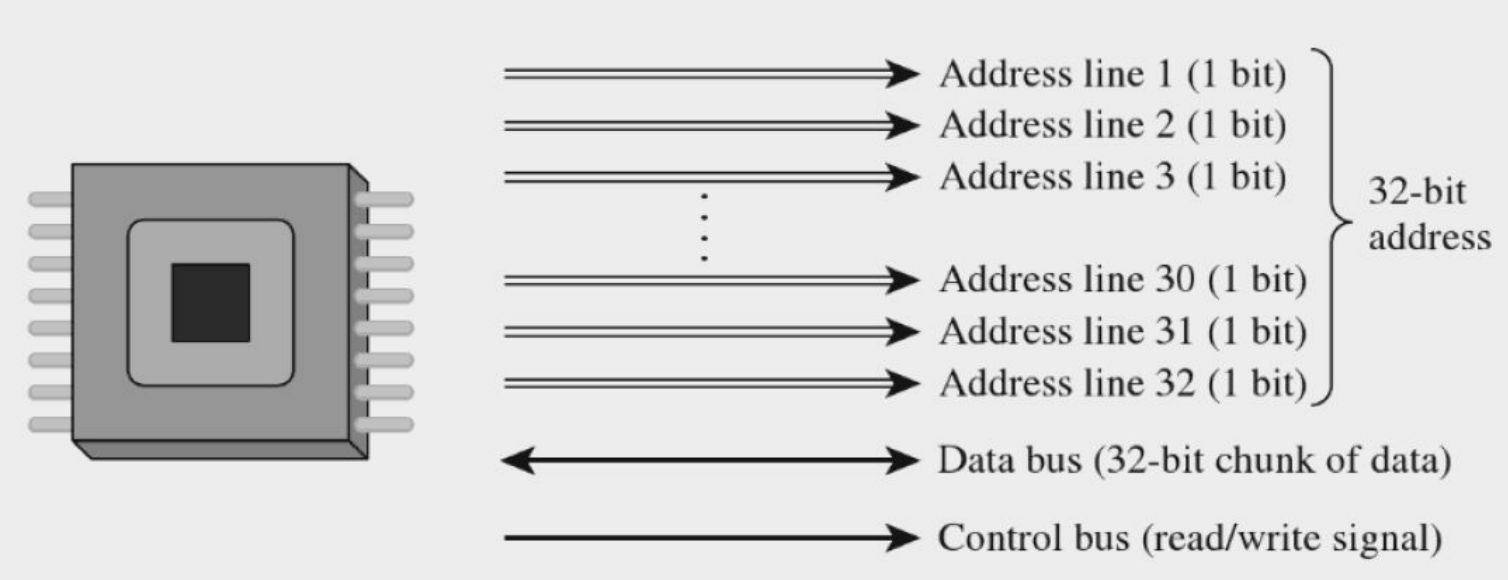
Flat and Segmented Memory Models

# *Addresses*

## So… What's memory?

A computer's memory is built using a hardware component called RAM (Random Access Memory) and it is used to store any type of data while the computer is turned on.

The RAM is connected directly to the motherboard and the CPU by the **Data Bus** which is formed by **address lines**. Each address line represents one bit of the **physical address** where the data is stored in the RAM



# IA-32 Memory Management

An **address is an integer** (e.g 0x80203040 or 2,149,593,152) that points to one byte (8 bits) of information.

Addresses start at 0 and end at $2^{\wedge N}$-1 where N is the number of addresses lines.

If a CPU has a data bus of 32 bits, the **physical address space** of the RAM goes from 0 to $2^{\wedge 32}$-1 → 0 to 4,294,967,295 (or 0xFFFFFFFF)[1].

If each **physical address** points to one **byte** of information, this means that a 32-bit processor can access 4,294,967,295 bytes.

4,294,967,295 bytes → 4,194,303.9 kilobytes[2] → 4,095.9 megabytes → 3.9 gigabytes

~ **4GB**

IA-32/Intel 64 **reads and writes** memory in **chunks** of 4/8 bytes, respectively! That means that the address points to the first bit of that chunk of bytes.

[1] Now it's obvious why it's common to use hexadecimal instead of decimal, right?
[2] Please, don't make me pronounce that...https://en.wikipedia.org/wiki/Kibibyte

Finally, we have these different **address types**:

- Physical Addresses (already explained)
- Logical Addresses
- Linear Addresses

We will understand each of these type of addresses by understanding the different **memory models**:

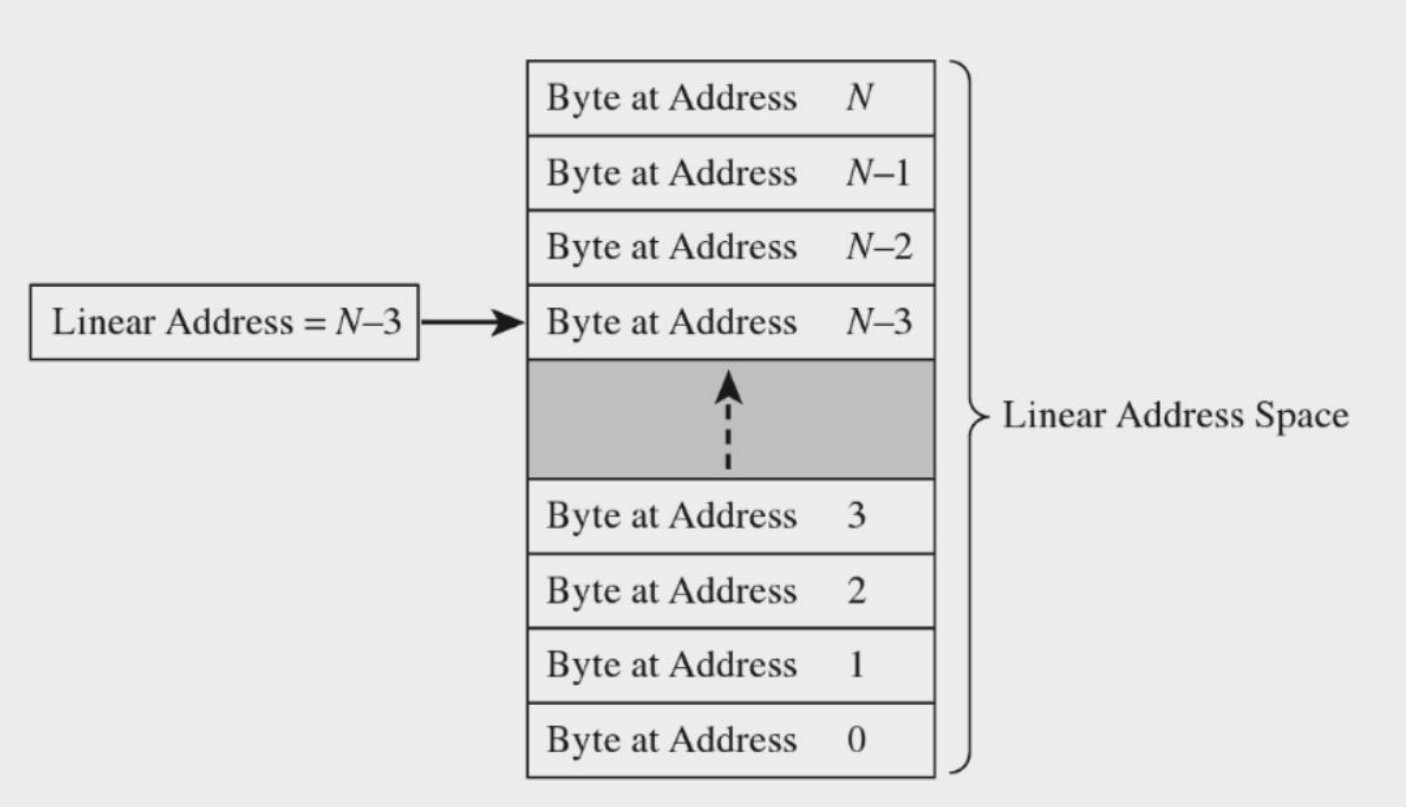
- Flat Memory Model
- Segmented Memory Model

# *Flat Memory Model*

Similar to what we see when talking about physical addresses.

This model is just **an abstraction** useful to introduce **linear addresses**.

The linear address might or might not be the same as the physical address.

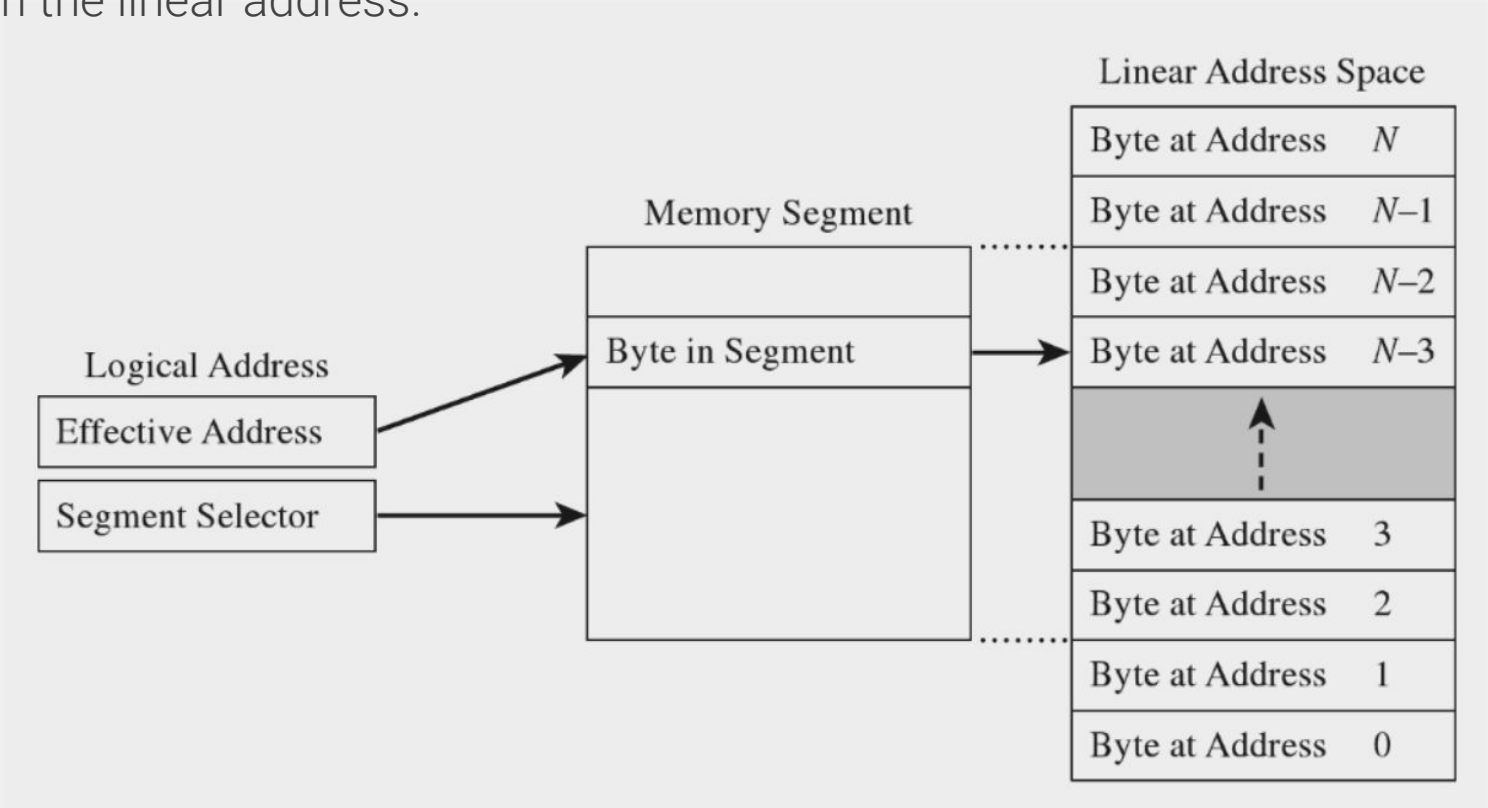


# *Segmented Memory Model*

The **Logical address** (*far pointer*) concept is introduced.

- Segment selector + Effective address (*offset address*)

The logical address is used to find a **memory segment** that will contain the linear address.



# IA-32 Operation Modes

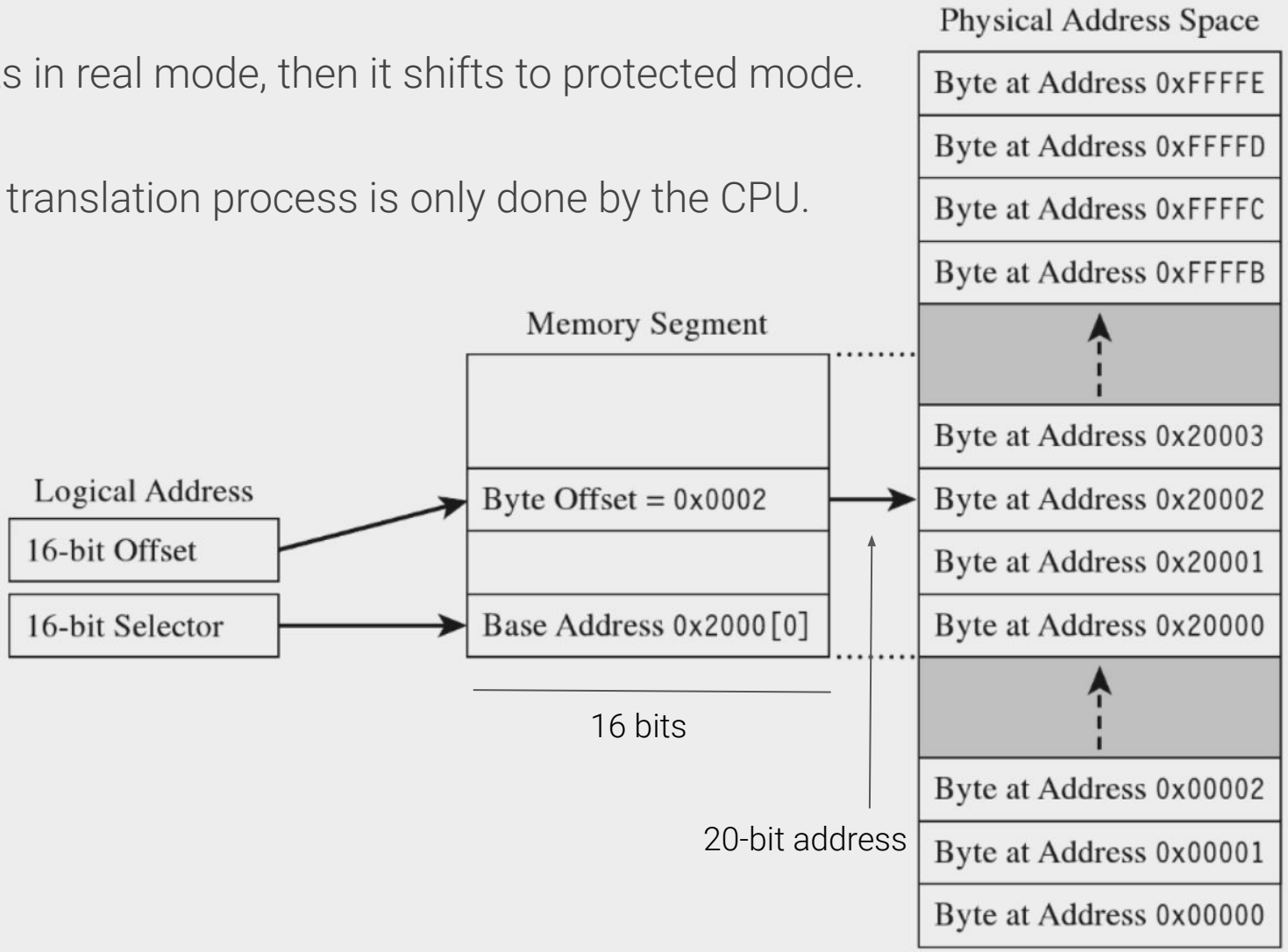
Real and Protected Modes

# *Real Mode*

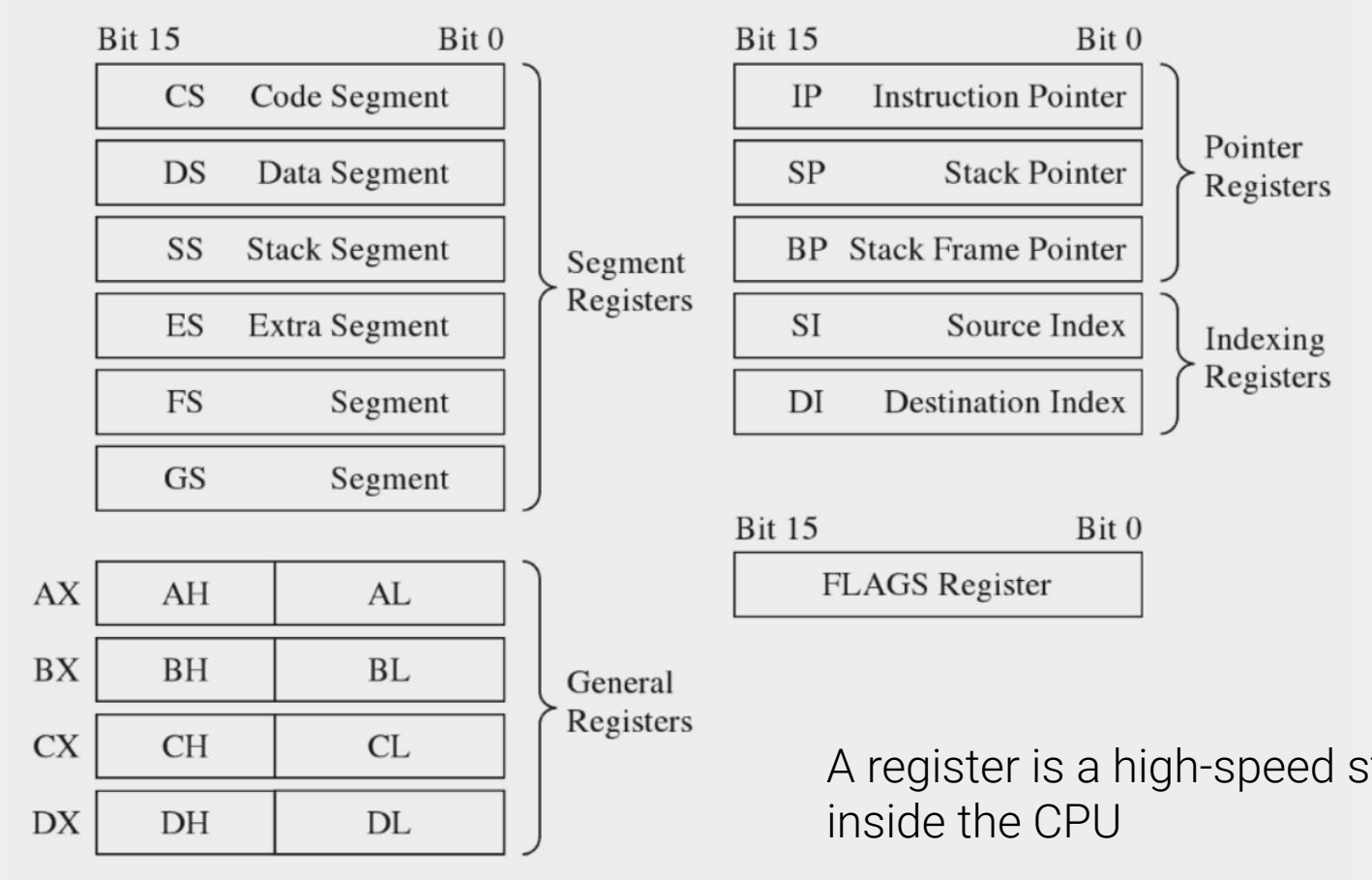
Real mode is an instance of the segmented model, **without protection**.

System boots in real mode, then it shifts to protected mode.

The address translation process is only done by the CPU.



## Execution Environment

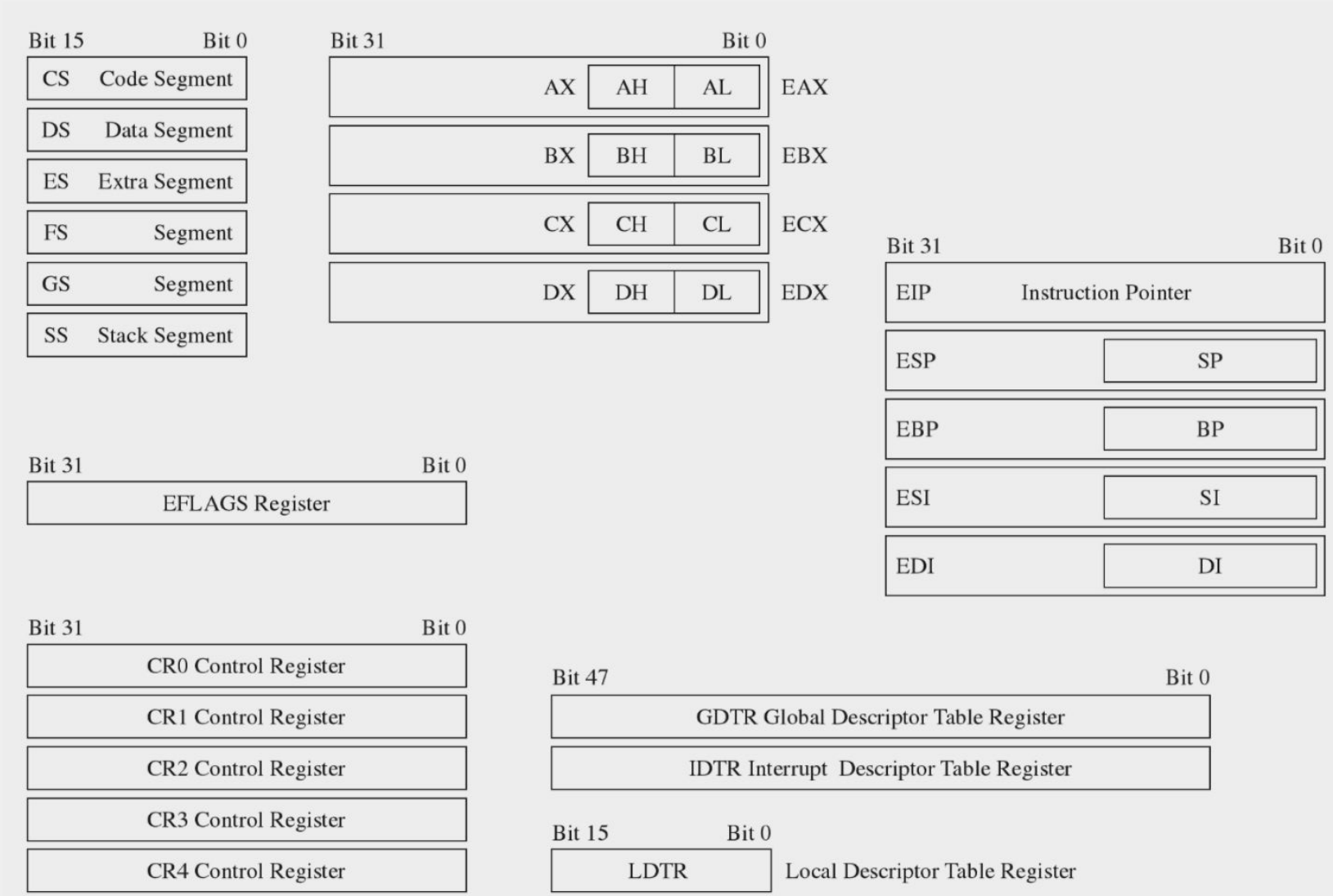



A register is a high-speed storage location inside the CPU

# *Protected Mode*

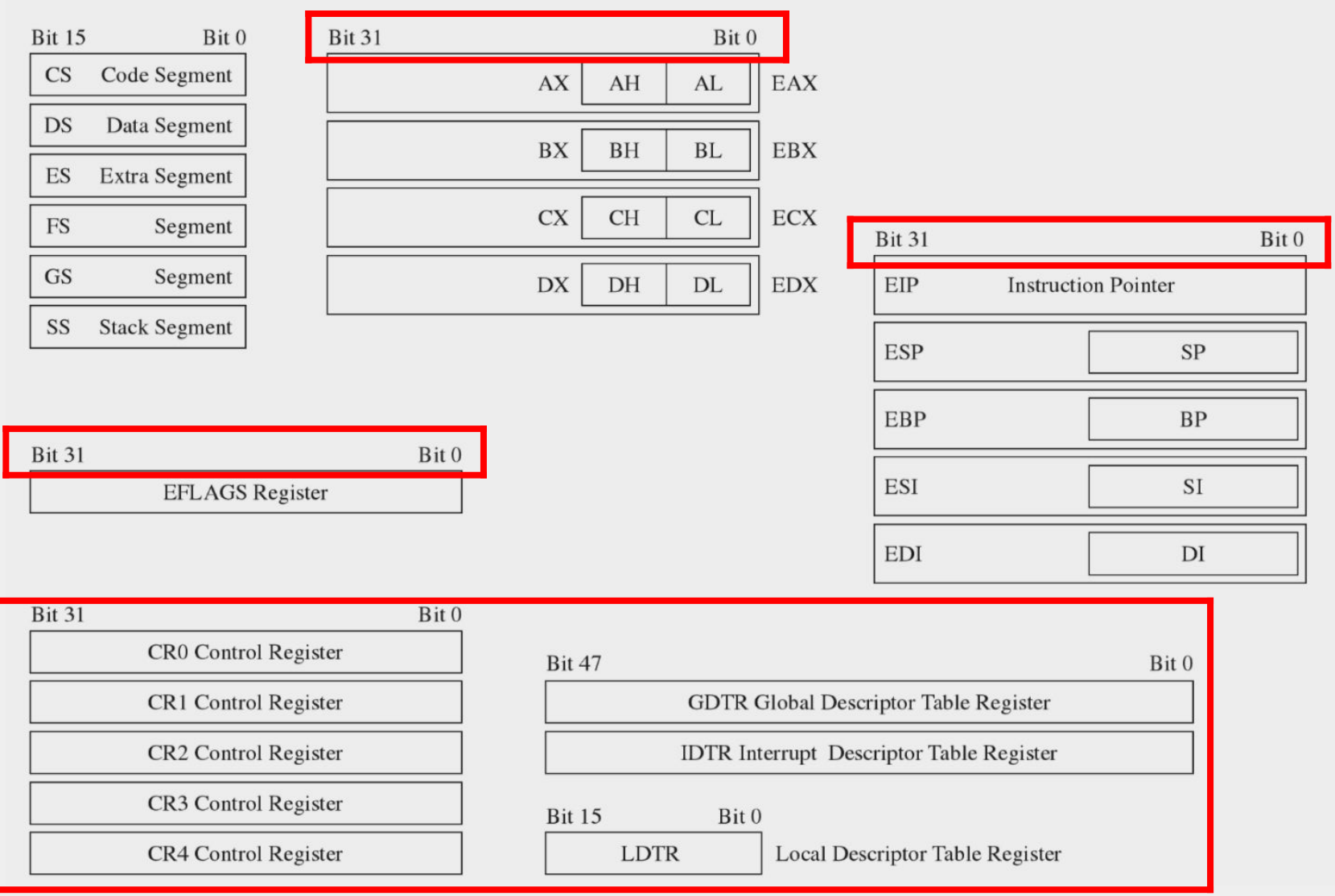
# IA-32 Operation Modes → Protected Mode

Protected mode is an instance of the segmented model, **with protection**.
The address translation process is done by the CPU **and the OS.**



# IA-32 Operation Modes → Protected Mode

Protected mode is an instance of the segmented model, **with protection**.
The address translation process is done by the CPU **and the OS.**



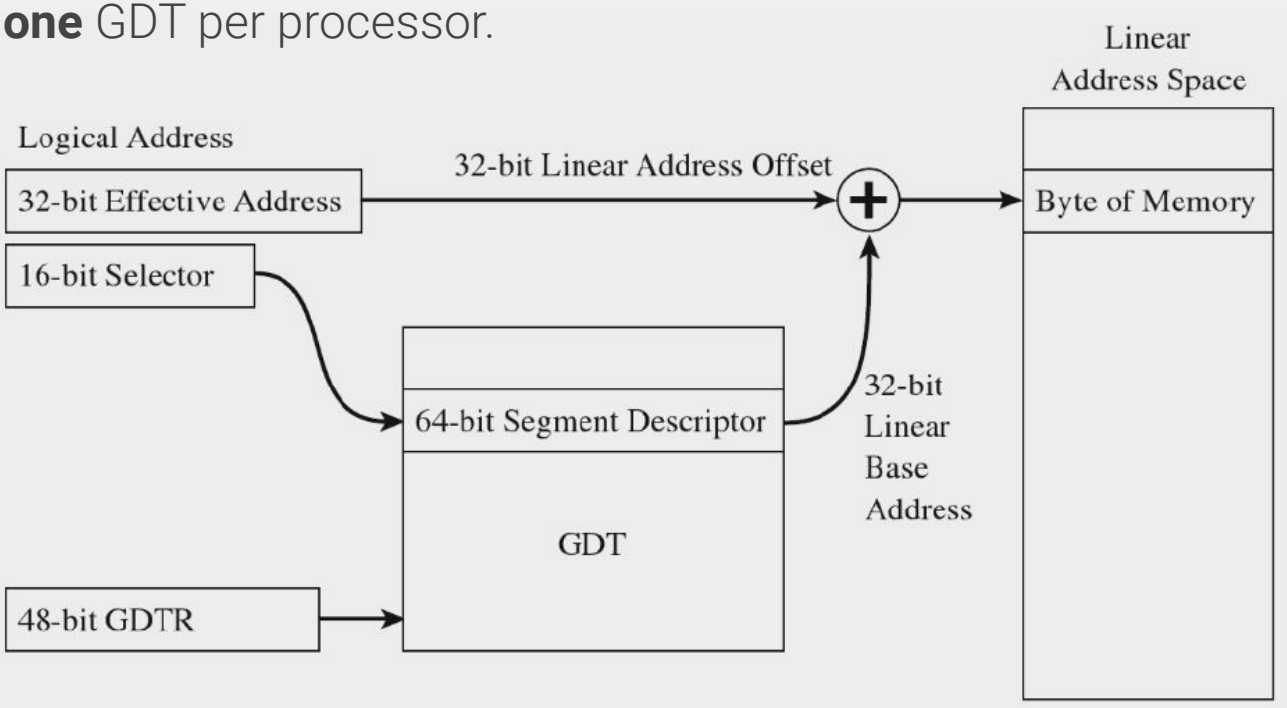
# *IA-32 Segmentation*

# IA-32 Segmentation

The **Global Descriptor Table** (GDT) is a data structure containing **Segment Descriptors**.

A segment descriptor represents the **base linear address for a given logical address** and some **metadata** about that linear address.

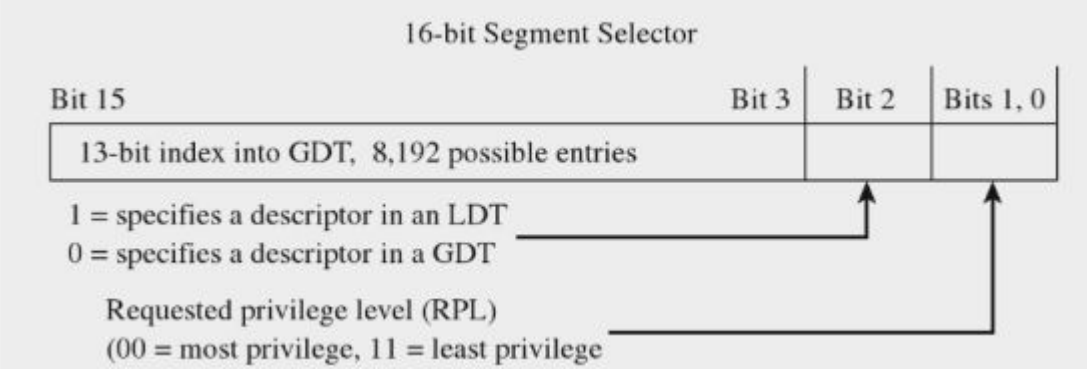
There's only **one** GDT per processor.




Protected-Mode Segmentation Without Paging

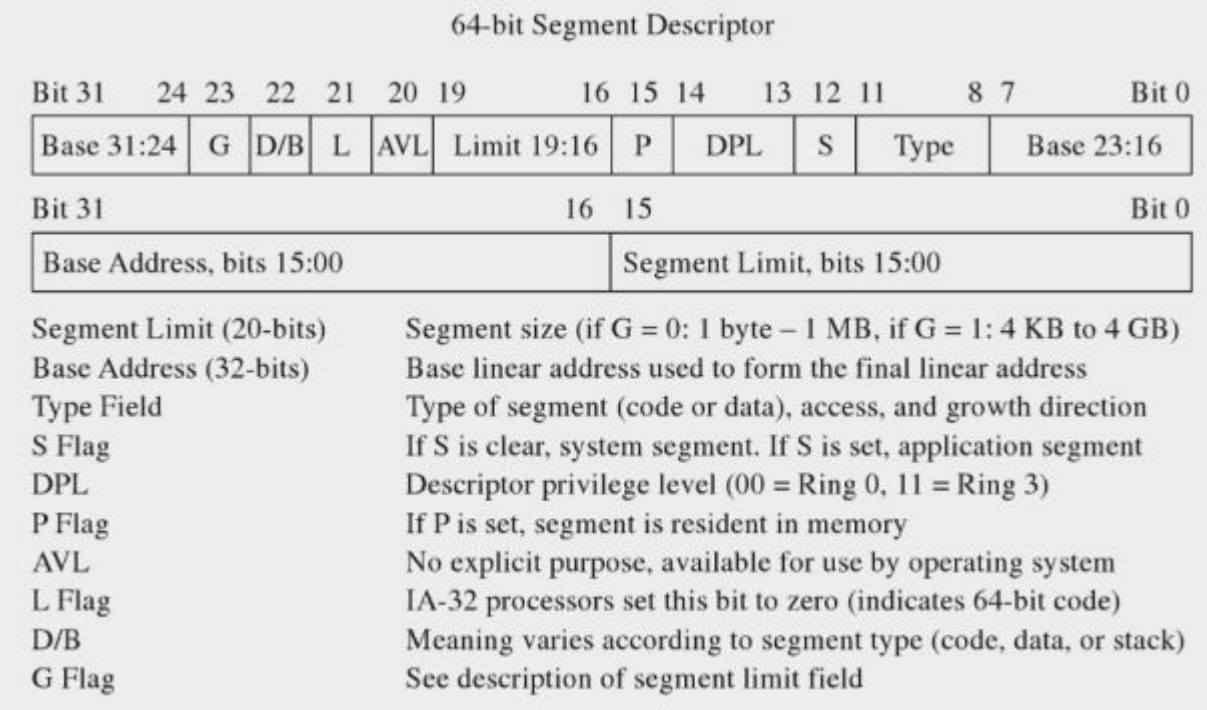
## The Segment Selector

The **Requested Privilege Level (RPL)** is one of the components that builds the **protection** scheme in protected mode segmentation.



## The Segment Descriptor

The **Descriptor Privilege Level (DPL)** is another of the components that builds the **protection** scheme in protected mode segmentation.



The approach that Intel CPU provides around segmentation is all founded on these bit level metadata.

Each time that code is executed (e.g. a JMP instruction is executed) internally there are algorithms handling things like:

- What segments should be used?
- Where are those segments?
- Can those segments be used by the current process?
- etc

All this happens behind the scenes without the user (the reverse engineer) caring too much.

# IA-32 Segmentation

The introduction of segmentation is how Intel handled, among others, the protection mechanisms around code access/execution (e.g. a regular user should not be able to tamper OS code).

→ Through RPL, DPL, CPL (Current Privilege Level) metadata

However, OS developers decided they would not use these capabilities provided by the processor…

That's when **Paging/Pagination** comes into play!

# *IA-32 Paging*

## IA-32 Paging

Paging is optional.

If no paging, then linear address → physical address.

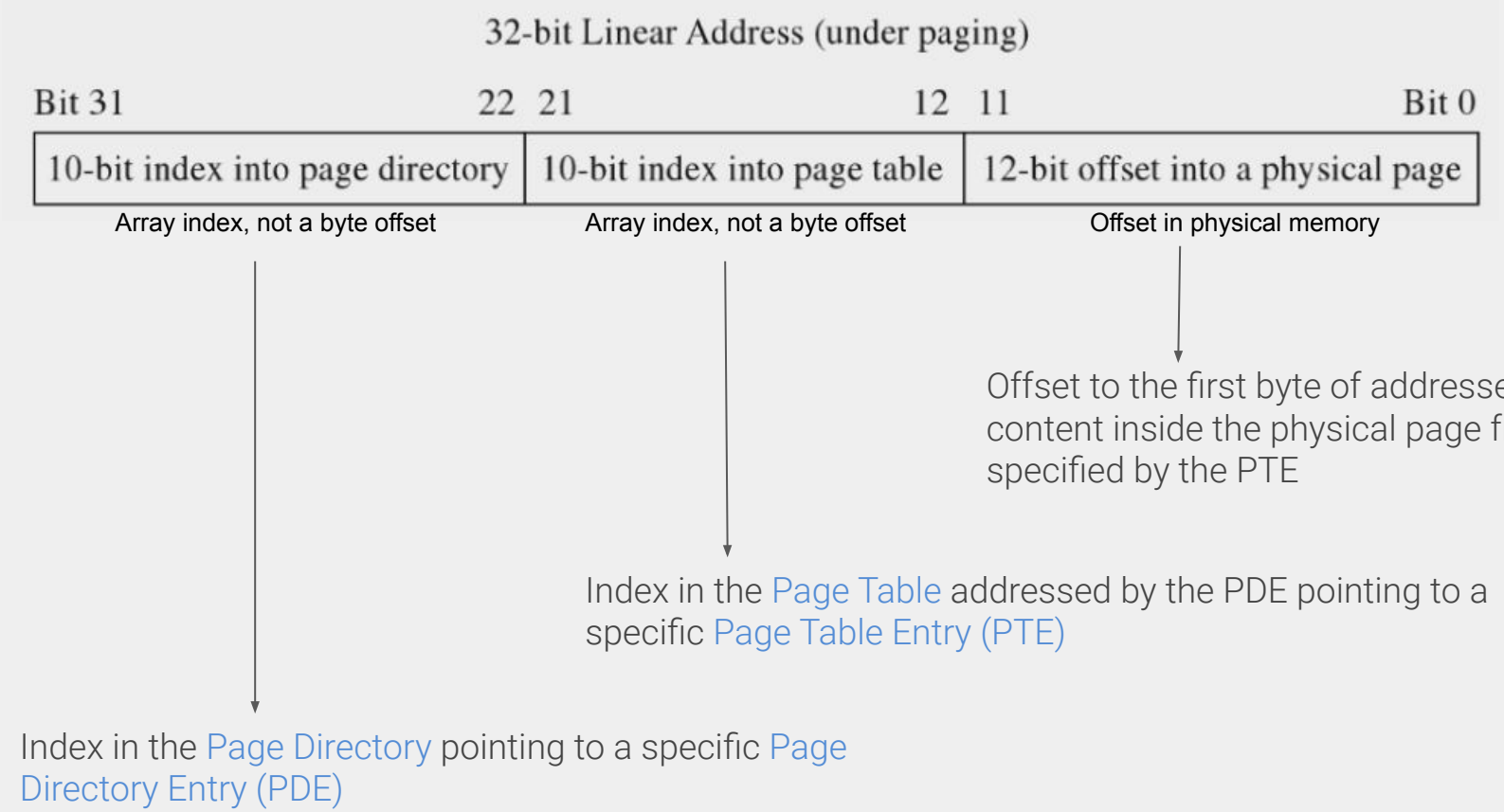
If no paging, **maximum addressable physical memory is 4GB** given that linear addresses are 32 bits ($2^{32}$).

***Page Frames*** are slots in physical memory that contain ***Pages***, which contents can be both in memory or disk (loaded to memory on access).

Pages can be 4KB (usually), 2MB (with PAE) or 4MB (with and without PAE).

# IA-32 Paging

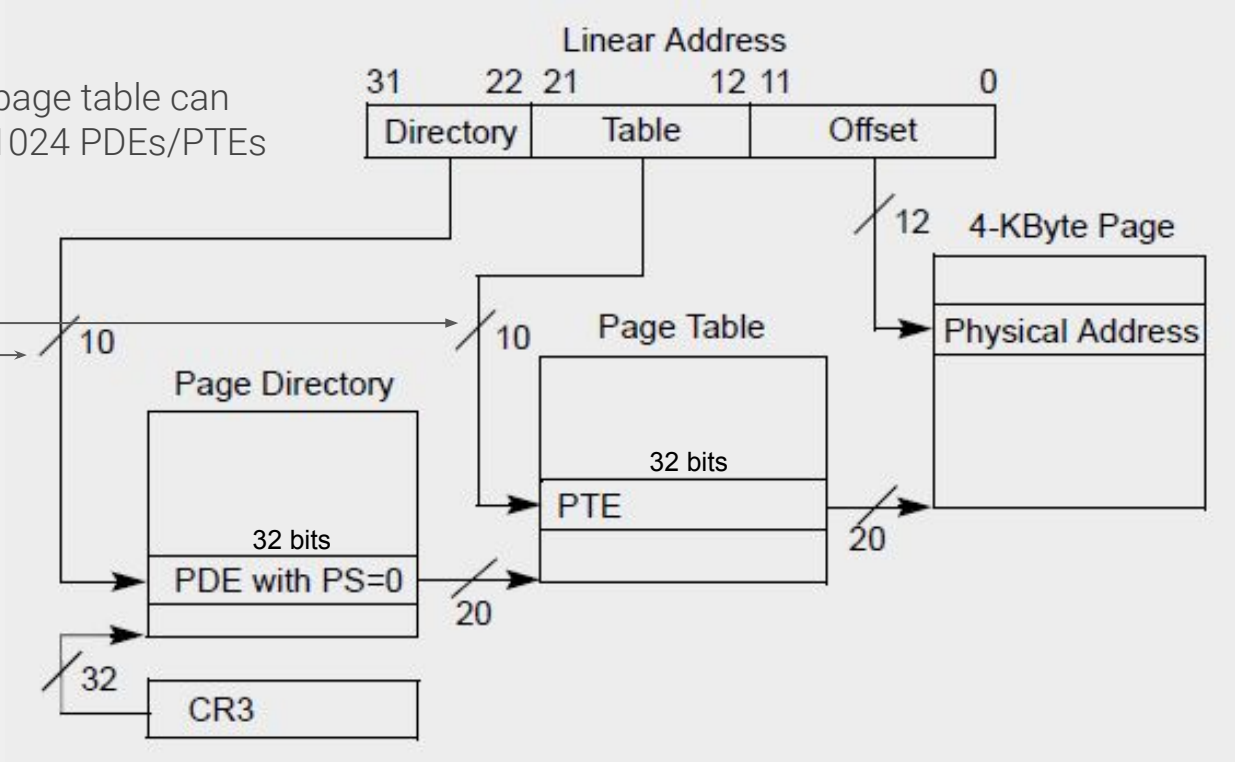
With paging enabled[1], a linear address is interpreted as follows (4KB pages):




[1] Bit PG=1 in CR0 and bit PAE=0 in CR4

## Paging diagram

A page directory / page table can only contain $2^{10}$ = 1024 PDEs/PTEs




- CR3 (the Page Directory Base Register (PDBR)) stores the **physical address** of the first byte of the Page Directory structure. Each process has its own CR3 pointing to its own Page Directory[1].
- Each PDE stores the **physical address** of the first byte of the Page Table structure.
- Each PTE stores the **physical address** of the first byte of the Page Frame.
- The linear address offset is added to the content of the PTE to get the final physical address.

[1] https://technet.microsoft.com/en-us/library/cc736309

## Page Directory and Page Table entries

32-Bit Page Directory Entry (PDE)

| Bit 31 — 12 | 11 — 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 20-bit Page-Table Base Address | Avail | G | PS | 0 | A | PCD | PWT | U/S | W | P |

| | |
|---|---|
| Avail | Available for OS use |
| G | Global page (ignored) |
| PS | Page size (0 indicates 4 KB page size) |
| 0 | Set to zero |
| A | Accessed (this page/page table has been accessed, when set) |
| PCD | Cache disabled (when this flag is set, this page/page table cannot be cached) |
| PWT | Write-through (if set, write-through caching is enabled for this page/page table) |
| U/S | User/Supervisor (when this flag is clear, the page has supervisor privileges ) |
| W | Read/Write (if this flag is clear, the pages pointed to by this entry are read-only) |
| P | Present (if this flag is set, the 4 KB page/page table is currently loaded into memory) |

32-Bit Page Table Entry (PTE)

| Bit 31 — 12 | 11 — 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 20-bit Page Base Address | Avail | G | PAT | D | A | PCD | PWT | U/S | W | P |

| | |
|---|---|
| G | Global flag (helps keep frequently accessed pages in the TLB) |
| PAT | Page attribute table index |
| D | Dirty bit (if set, the page pointed to has been written to) |

1. A Page Table Entry can use 20 bits to point to each Page Frame. $2^{20}$ = 1,048,576 Page Frames
2. Each Page Frame is $2^{12}$ = 4KB.

Therefore, **maximum addressable physical memory** is 1,048,576 x 4KB = 4GB

# *Page Address Extension (PAE)*

# IA-32 Paging → Page Address Extension (PAE)

As we saw before, paging allowed the OS to address the same amount of memory than without paging (4GB).

**Physically**, the address space accessible by a processor is limited by the address lines connected to the processor.

To increase address space, Intel introduced the Pentium Pro (1995) that instead of 32 address lines, it had 36 so it could address $2^{36}$ = 64GB.

**Virtually**, something was needed to handle this physical memory extension…

…That's why **PAE** was created.

Aside from handling this memory extension, PAE also introduced security features such as No-eXecute (NX bit).

[1] Bit PAE=1 in CR4

PAE introduces a preliminary table with 4 entries called **Page Directory Pointer Table (PDPT)**.

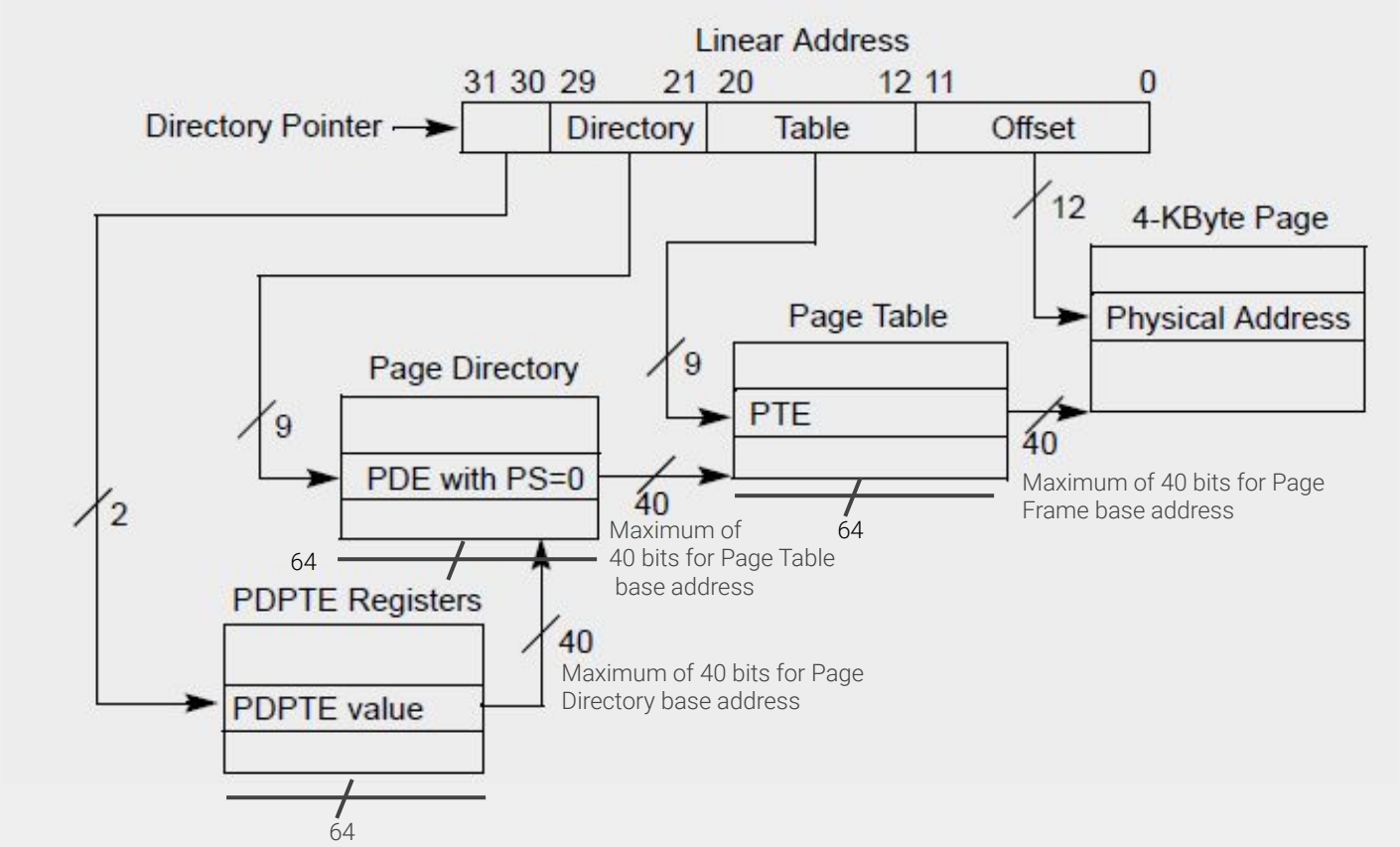
With PAE enabled, a linear address is interpreted as follows:

32-bit Linear Address (under paging with PAE)

| Bit 31, 30 | 29 … 21 | 20 … 12 | 11 … Bit 0 |
|---|---|---|---|
| Index into PDPT | 9-bit index into page directory | 9-bit index into page table | 12-bit offset into physical page |

The most significant two bits are the index to the PDPT. Both the Page Directory and Page Table pointers are one bit less.

## Paging diagram under PAE




52 is the maximum address lines that a x86 processor can have [1]. Therefore, maximum bits that can be used for the base address lines for both PDEs and PTEs is 40 (next slide).

[1] Intel® 64 and IA-32 Architectures, Software Developer's Manual, Volume 3A, System Programming Guide, Part 1. Table 4-8.

## Page Directory and Page Table entries under PAE

64-Bit PDPTE

| Bit 63 M | M-1 12 | 11 9 | 8 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Reserved (i.e., zero) | Base Physical Address of Page Directory | Ignored | Reserved (zero) | PCD | PWT | 0 | 0 | P |

64-Bit Page Directory Entry (PDE)

| Bit 63 M | M-1 12 | 11 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| Reserved (i.e., zero) | Base Physical Address of Page Table | Ignored | PS | 0 | A | PCD | PWT | U/S | W | P |

64-Bit Page Table Entry (PTE)

| Bit 63 M | M-1 12 | 11 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved (i.e., zero) | Base Physical Address of Page | Avail | G | PAT | D | A | PCD | PWT | U/S | W | P |

1. M stands for MAXPHYADDR, which is the maximum number of address lines of a CPU.
2. If a CPU has 36 address lines (as the Pentium Pro example before):
   a. A Page Table Entry can use 24 bits (36 - 1 - 11 = 24) to point to each Page Frame. $2^{24}$ = 16,777,216 Page Frames
   b. Each Page Frame is $2^{12}$ = 4KB.

Therefore, **maximum addressable physical memory** is 16,777,216 x 4KB = 64GB

# *Address Translation Process*

For the **IA-32** architecture

# Address Translation Process

Let’s check the **Kernel Debugging for Segmentation and Paging Analysis in IA32** document!

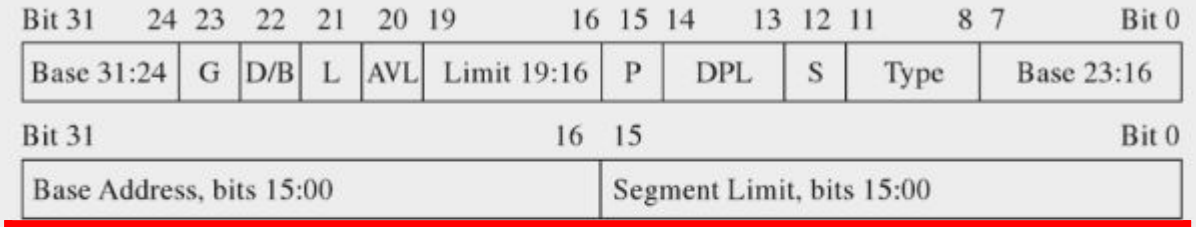
# *Introduction to Memory Protection*

Segmentation and Paging Protections

# *Protection Through Segmentation*

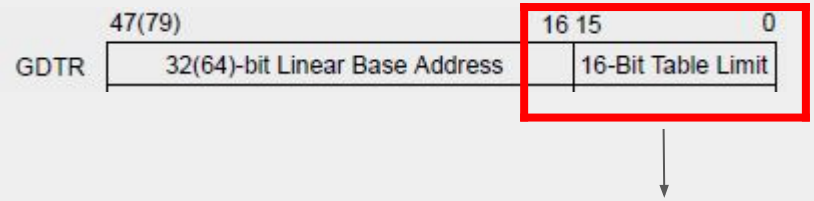
## Examples of Limit check protections

64-bit Segment Descriptor




| | |
|---|---|
| Segment Limit (20-bits) | Segment size (if G = 0: 1 byte – 1 MB, if G = 1: 4 KB to 4 GB) |
| Base Address (32-bits) | Base linear address used to form the final linear address |
| Type Field | Type of segment (code or data), access, and growth direction |
| S Flag | If S is clear, system segment. If S is set, application segment |
| DPL | Descriptor privilege level (00 = Ring 0, 11 = Ring 3) |
| P Flag | If P is set, segment is resident in memory |
| AVL | No explicit purpose, available for use by operating system |
| L Flag | IA-32 processors set this bit to zero (indicates 64-bit code) |
| D/B | Meaning varies according to segment type (code, data, or stack) |
| G Flag | See description of segment limit field |

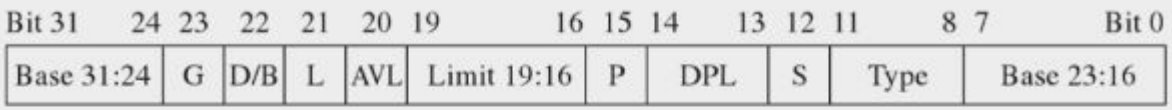
To check if accesses to the Segment are out of bounds




To check if accesses to the GDT are out of bounds

## Examples of Type check protections

64-bit Segment Descriptor




| | |
|---|---|
| Segment Limit (20-bits) | Segment size (if G = 0: 1 byte – 1 MB, if G = 1: 4 KB to 4 GB) |
| Base Address (32-bits) | Base linear address used to form the final linear address |
| Type Field | Type of segment (code or data), access, and growth direction |
| S Flag | If S is clear, system segment. If S is set, application segment |
| DPL | Descriptor privilege level (00 = Ring 0, 11 = Ring 3) |
| P Flag | If P is set, segment is resident in memory |
| AVL | No explicit purpose, available for use by operating system |
| L Flag | IA-32 processors set this bit to zero (indicates 64-bit code) |
| D/B | Meaning varies according to segment type (code, data, or stack) |
| G Flag | See description of segment limit field |

Depending on the type of segment, some instruction might rise exceptions. Examples:

- Instructions cannot write to code segments
- Far jumps can only jump to code segments (or call gates)

## Examples of Privilege check protections

Four privilege levels. From 0 (most) to 3 (least). Often called rings.

Processes running in less privileged rings should not access more privileged segments

Players in this game:

- CPL (Current Privilege Level) ← In the Segment Registers. Bits 0, 1.
  - **Privilege level of the process being executed**
- DPL (Descriptor Privilege Level) ← In Segment Descriptor. Bits 13, 14.
  - **Privilege level of a segment**
- RPL (Requestor Privilege Level) ← In the Segment Selector. Bits 0, 1.
  - An (override) privilege level associated to segment selectors.

## Privilege level protections workflows

Privilege checks are executed when:

- The segment selector of a segment descriptor is going to be loaded into a segment register **(privilege checks happen before)**

This happens, for example, when:

- Instructions change execution flow to other than the current code segments
- Instructions read/write data in other than the current data segments

Depending on what type of action happens, **checks will be different**:

- Intel® 64 and IA-32 Architectures, Software Developer's Manual, Volume 3A: System Programming Guide, Part 1. Section 5.5
- 15 pages defining different types of checks...

## E.g. Privilege level checks with direct (far) call or jumps to code segments

1. Get CPL from current Code Segment

1. Get DPL from the Segment Selector of the destination Code Segment

1. Check if CPL == DPL (RPL is not used)
   a. True → Jump to that code segment
   b. False → Trigger exception

This basically means that **a process** executing code in one segment **has to have the same privileges** as the privilege level set for the **destination** code to be executed.

Code segments can be conforming or non-conforming. Mentioned workflow works for non-conforming segments.
Conforming code segments were introduced by Intel in order to allow the Kernel to share code with less privileged applications

## Examples of Restricted Instruction check protections

Checks are made to make sure that only privileged applications (CPL = 0) are able to execute specific instructions.

Examples:

- LGDT - Load GDT register value
- HLT - Halt processor until the next external interrupt is fired[1]
- etc...

[1]Interrupts are signals sent by hardware devices to the CPU alerting it that an event occurred to which it should react.

The funny thing...

The funny thing…

… these protections are barely used

Because segments are not really used by OS…

Only to differentiate between kernel and user code through using segments covering the whole memory space

# *Protection Through Paging*

# Introduction to Memory Protection → Protection Through Paging

64-Bit PDPTE

| Bit 63 M | M-1 12 | 11 9 | 8 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Reserved (i.e., zero) | Base Physical Address of Page Directory | Ignored | Reserved (zero) | PCD | PWT | 0 | 0 | P |

64-Bit Page Directory Entry (PDE)

| Bit 63 M | M-1 12 | 11 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| Reserved (i.e., zero) | Base Physical Address of Page Table | Ignored | PS | 0 | A | PCD | PWT | U/S | W | P |

64-Bit Page Table Entry (PTE)

| Bit 63 M | M-1 12 | 11 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved (i.e., zero) | Base Physical Address of Page | Avail | G | PAT | D | A | PCD | PWT | U/S | W | P |

Those bits define privileges for a page table or all the page tables under a page directory.

- U/S bit: If 1 → User mode. If 0 → Supervisor mode
- W/R bit: If 1 → Read write. If 0 → Read-only

U/S bit is tied to CPL of segment:

- If CPL = 0 → U/S bit = 0
- If CPL = 1 → U/S bit = 1

If the U/S flag (bit 2) is 0 in at least one of the paging-structure-entries, the address is a supervisor-mode address. Otherwise, the address is a user-mode address

*Reality*

Nowadays OSes implement a “**flat” segmentation model**.

OSes usually have 2 different segments covering the whole address space:

- Code and Data segments with DPL = 0 ← Kernel and drivers
- Code and Data segments with DPL = 3 ← User applications

This means that paging features are used extensively and segmentation features are used collaterally given that the OS is forced by the processor.

There are other segments used for different purposes

# *Homework*

## Homework

1. **Find examples of real-world usage of logical addresses.**

During this module, we have discussed both logical and linear addresses. This exercise is about you going ahead of the course contents and find out what's going on with logical addresses.

You can find examples in the internet or you can disassemble software to provide examples of logical address usage.

1. **Record a video with the real-world process for address translation.**

    1. Spin up your lab, setup the virtual machine, debug in kernel-mode, pick a random logical/linear address and read what it contains (not zeros…=).

    1. Then convert that logical/linear address into a physical address. Read the content of that physical address.

    1. The value it contains should match what you previously showed.

# *Bibliography*

## Bibliography


- The Rootkit Arsenal, 2nd Edition. Chapter 3, Hardware Briefing.
- Understanding the Linux Kernel, 2nd Edition. Chapter 2, Memory Addressing
- Intel® 64 and IA-32 Architectures, Software Developer's Manual, Volume 3A: System Programming Guide, Part 1. Chapter 2, System Architecture Overview
- Open Security Training, Intermediate x86
- Linus Torvalds rant on PAE
- How PAE X86 Works
- Difference between DPL and RPL in x86
- Privilege level checking when accessing code segment

## Index of Figures


- Slide 3, The Rootkit Arsenal, 2nd Edition. Page 91, 95.
- Slide 15, The Rootkit Arsenal, 2nd Edition. Page 56.
- Slide 19, The Rootkit Arsenal, 2nd Edition. Page 58.
- Slide 20, The Rootkit Arsenal, 2nd Edition. Page 59.
- Slide 24, The Rootkit Arsenal, 2nd Edition. Page 61.
- Slide 25, The Rootkit Arsenal, 2nd Edition. Page 65.
- Slide 26, 27, The Rootkit Arsenal, 2nd Edition. Page 88.
- Slide 37, The Rootkit Arsenal, 2nd Edition. Page 94.
- Slide 38. Intel® 64 and IA-32 Architectures, Software Developer's Manual, Volume 3A, System Programming Guide, Part 1. Page Vol. 3A 4-9