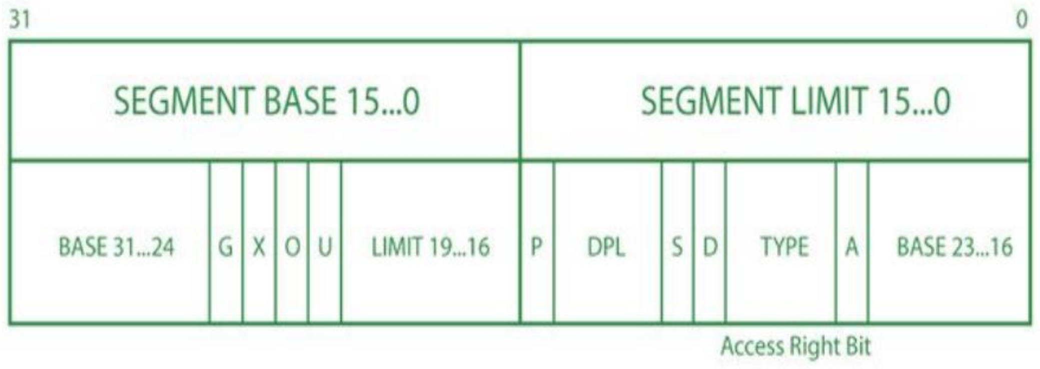
Unit 3

1) **Draw & Explain the general descriptor format available in various descriptor tables.**

General Format of Segment Descriptor

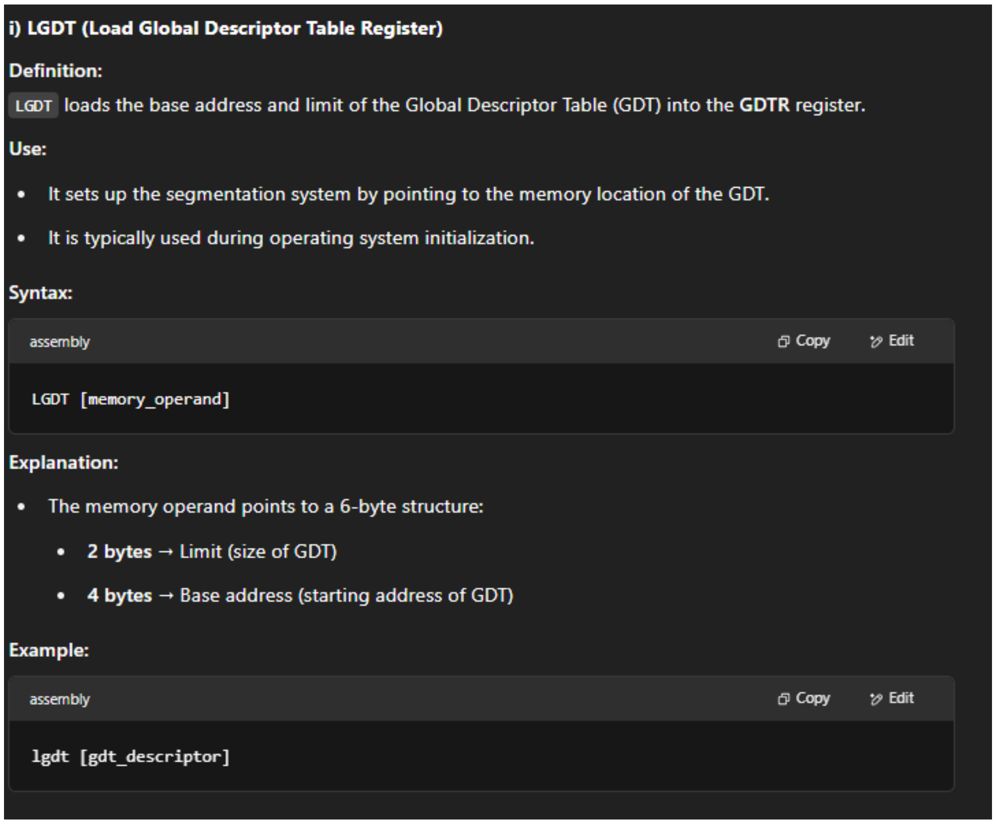


General formate of Descriptor.
(Formate for segment Descriptor)

31	Base Address 15...0						Limit 15...0						0
63	Base 31...24	G	D	0	AVL	Limit 19...16	P	DPL	S	TYPE	A	Base 23...16	32

1] It has total size of 64 bit of descriptor
2] It has 32 bits of base address & 20 bits of limit.
3] It has Info bits. G - ~~B~~Gradulity, D - Default size.
O - Don't care, AVL - A~~a~~vailable for user/os.
1] G - Gradulity bit states 0 = 0, 1 : 4KB is require.
2] D - Default size :- 0 - 16 bit requir, 1 - 32 bit require
3] 0 - Don't care.
4] AVL - Available for user/os :- ~~we~~ user can ~~ch~~ add some features as it required.
4] Now, move on to next we have Access Right bits.
P - Present, DPL - Descriptor Privilege level, S - system Type (Descriptor type). TYPE :- segment type, A - Access bit.
1] P - Present bit :- It shows that segment is present on RAM or not
2] ~~B~~PL - Descriptor Privilege level :- It check or matches privilege level as requested from user.
3] S - Decriptor Type :- It checks type of descriptor as 0 : system, 1 = code/data. If 0 then no access to user.
4] TYPE :- Descriptor segment type :- It ~~chee~~ has 3 bits E, ED & W, to check segment type as code, stack & data ~~as~~ respectively.
5] A :- Accessed bit :- So by CPU when segment is accessed.

2) Explain the use of following Instructions in detail: i) LGDT ii) SGDT iii) SIDT



i) LGDT (Load Global Descriptor Table Register)

Definition:
`LGDT` loads the base address and limit of the Global Descriptor Table (GDT) into the **GDTR** register.

Use:

- It sets up the segmentation system by pointing to the memory location of the GDT.
- It is typically used during operating system initialization.

Syntax:

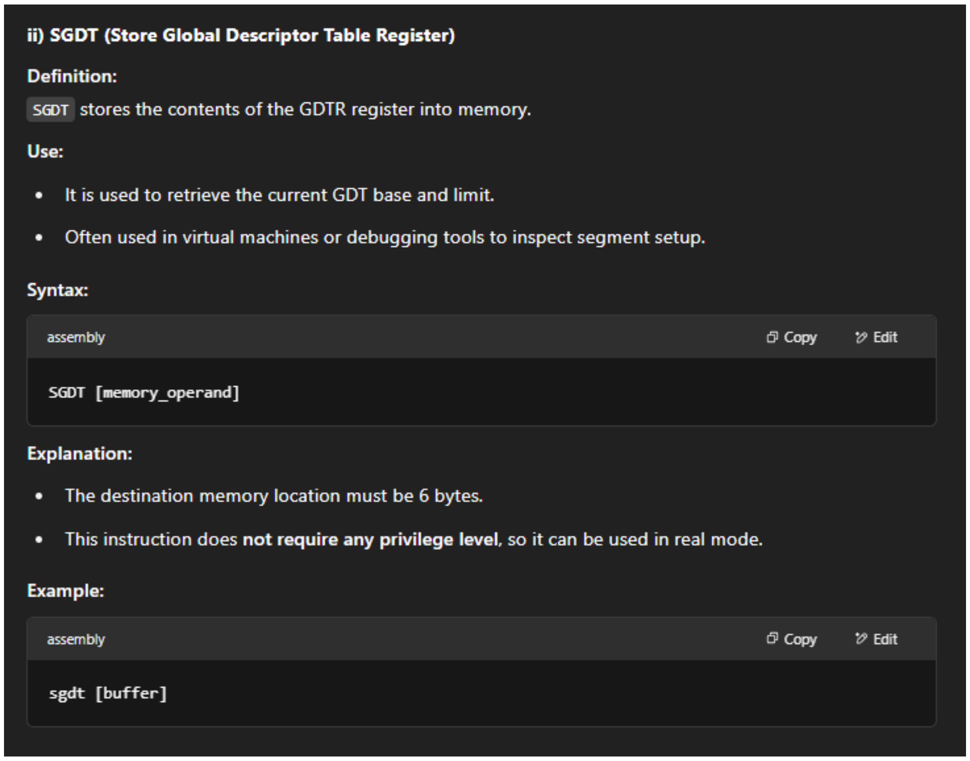
```assembly
LGDT [memory_operand]
```

Explanation:

- The memory operand points to a 6-byte structure:
  - **2 bytes** → Limit (size of GDT)
  - **4 bytes** → Base address (starting address of GDT)

Example:

```assembly
lgdt [gdt_descriptor]
```



ii) SGDT (Store Global Descriptor Table Register)

Definition:
`SGDT` stores the contents of the GDTR register into memory.

Use:

- It is used to retrieve the current GDT base and limit.
- Often used in virtual machines or debugging tools to inspect segment setup.

Syntax:

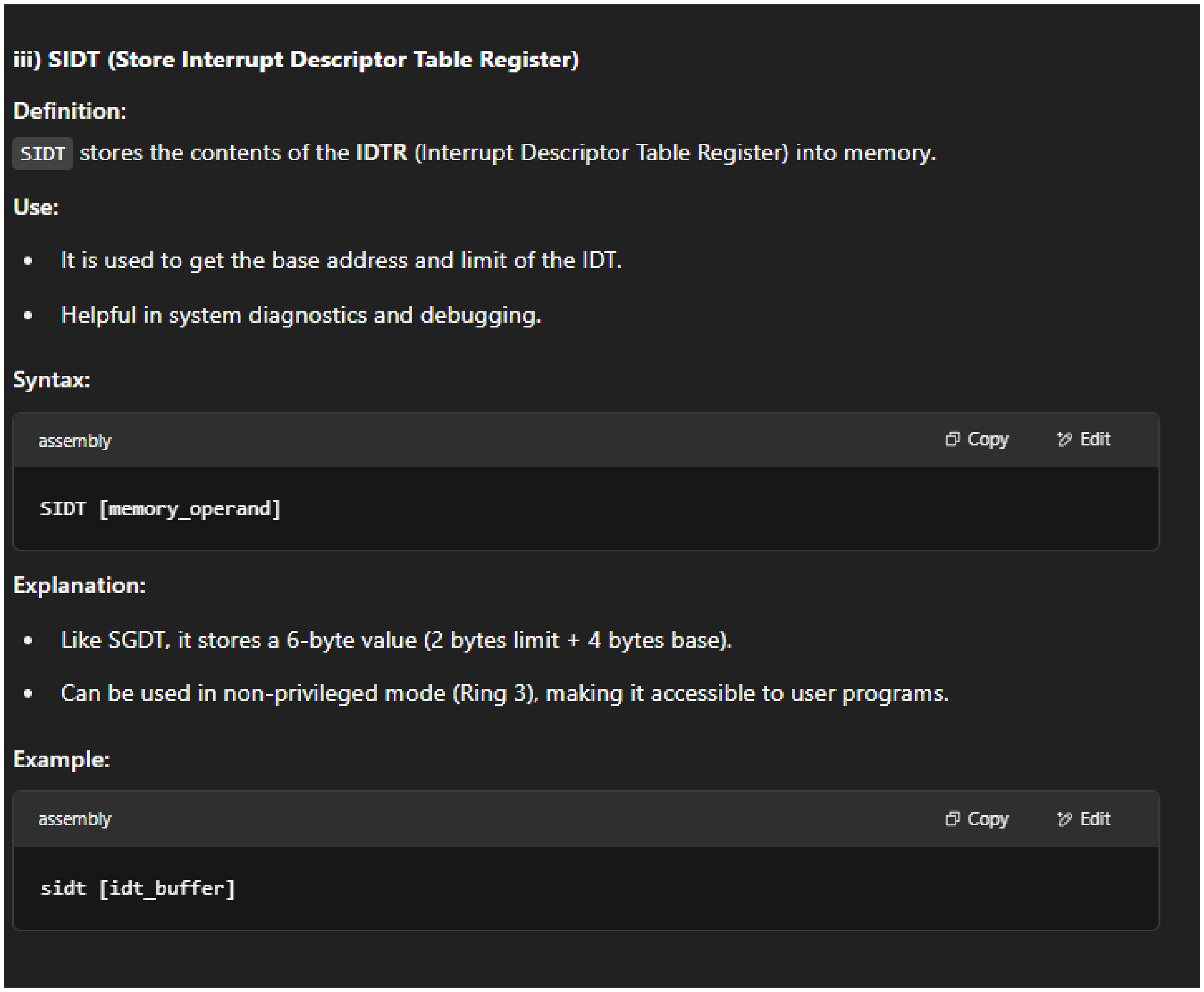
```assembly
SGDT [memory_operand]
```

Explanation:

- The destination memory location must be 6 bytes.
- This instruction does **not require any privilege level**, so it can be used in real mode.

Example:

```assembly
sgdt [buffer]
```


iii) SIDT (Store Interrupt Descriptor Table Register)

Definition:

`SIDT` stores the contents of the **IDTR** (Interrupt Descriptor Table Register) into memory.

Use:

- It is used to get the base address and limit of the IDT.
- Helpful in system diagnostics and debugging.

Syntax:

```assembly
SIDT [memory_operand]
```

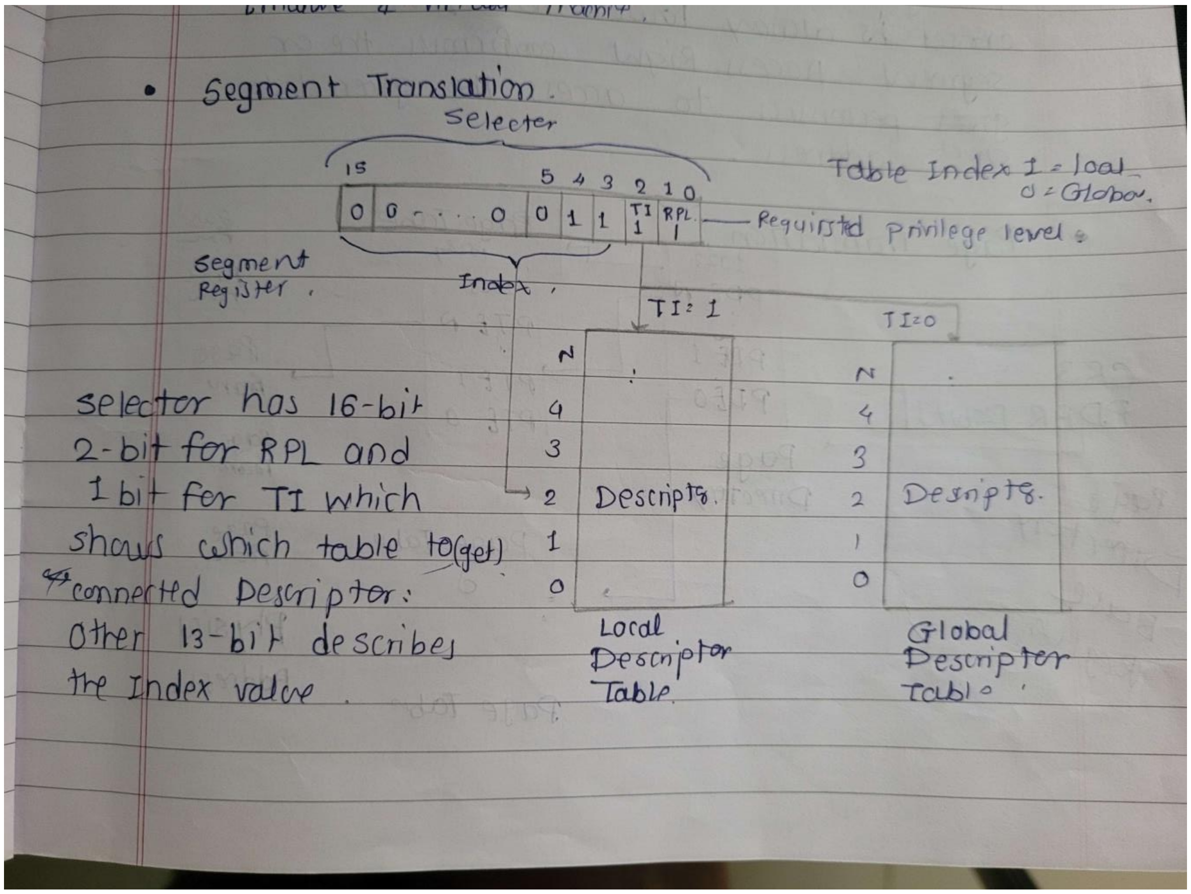
Explanation:

- Like SGDT, it stores a 6-byte value (2 bytes limit + 4 bytes base).
- Can be used in non-privileged mode (Ring 3), making it accessible to user programs.

Example:

```assembly
sidt [idt_buffer]
```

3) Explain the segment Translation process of 80386.



- Segment Translation.

Selecter

15				5	4	3	2	1 0
0	0	- . . .	0	0	1	1	TI 1	RPL 1

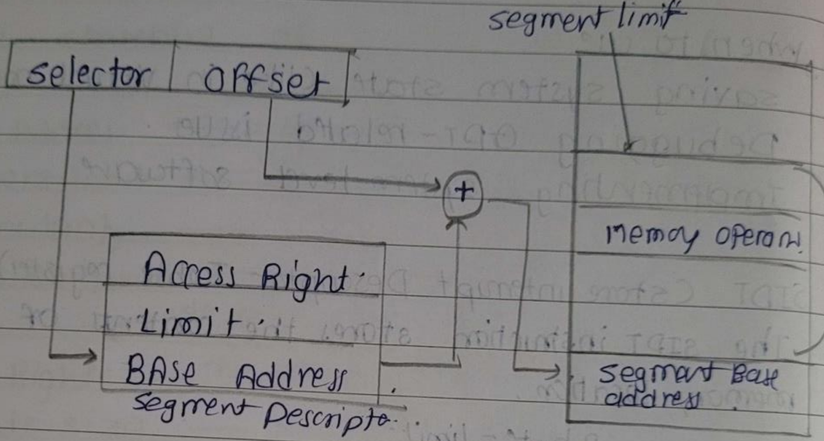
Segment Register, Index, Requisted privilege level.

Table Index 1 = local, 0 = Global.

TI = 1: N, 4, 3, 2 Descripter, 1, 0 — Local Descriptor Table.

TI = 0: N, 4, 3, 2 Descripter, 1, 0 — Global Descripter Table.

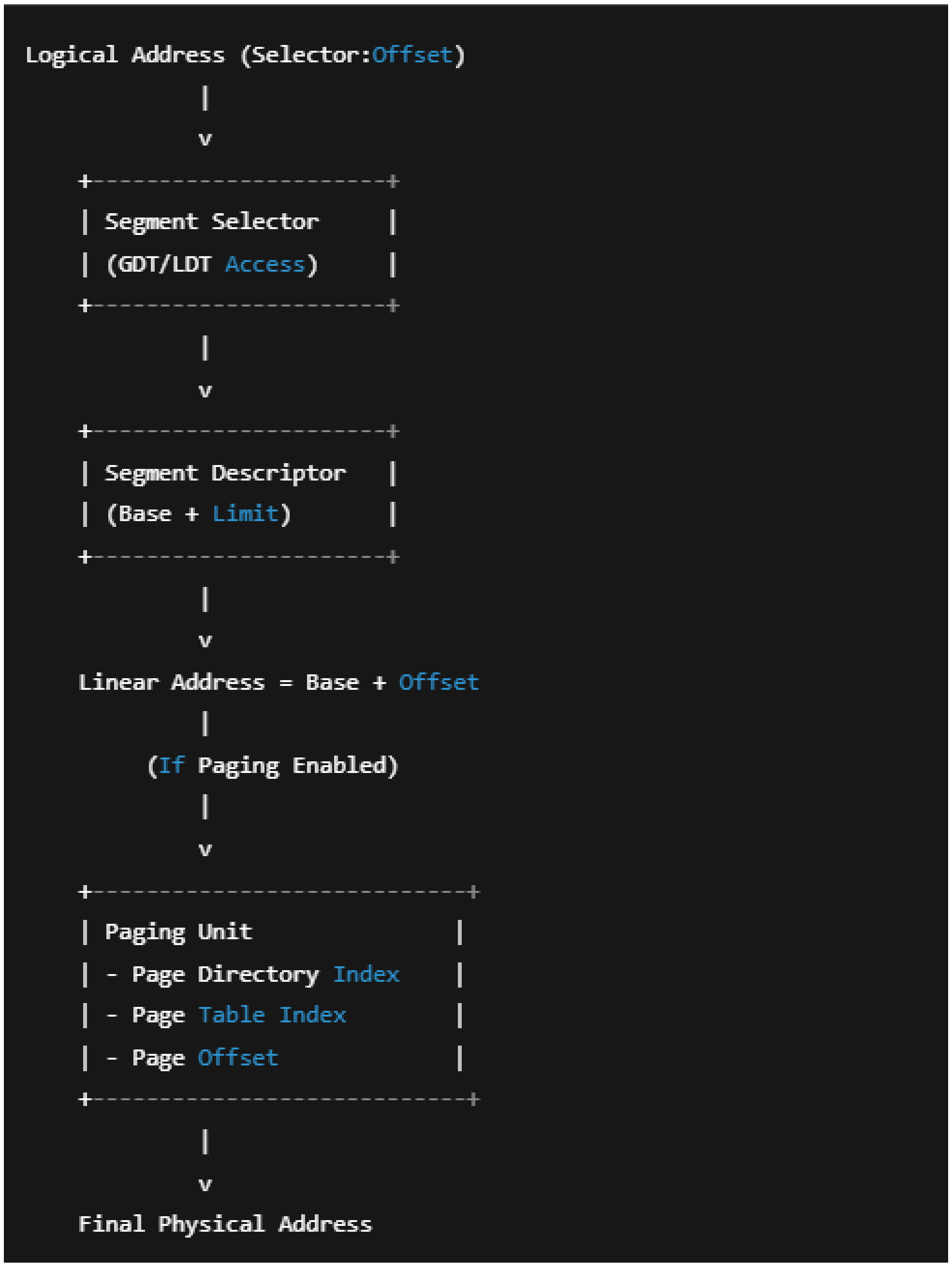
selector has 16-bit
2-bit for RPL and
1 bit for TI which
shows which table to(get)
connected Descriptor:
Other 13-bit describes
the Index value.



selecter get the base address, Limit and Access Right from segment Descripter. ~~an~~ with the help of offset & base address we get the linear address in to the Table. offset is alway lower than Limit of segment. Access Right confirmus the or ~~gives~~ check permission to access the given data address.

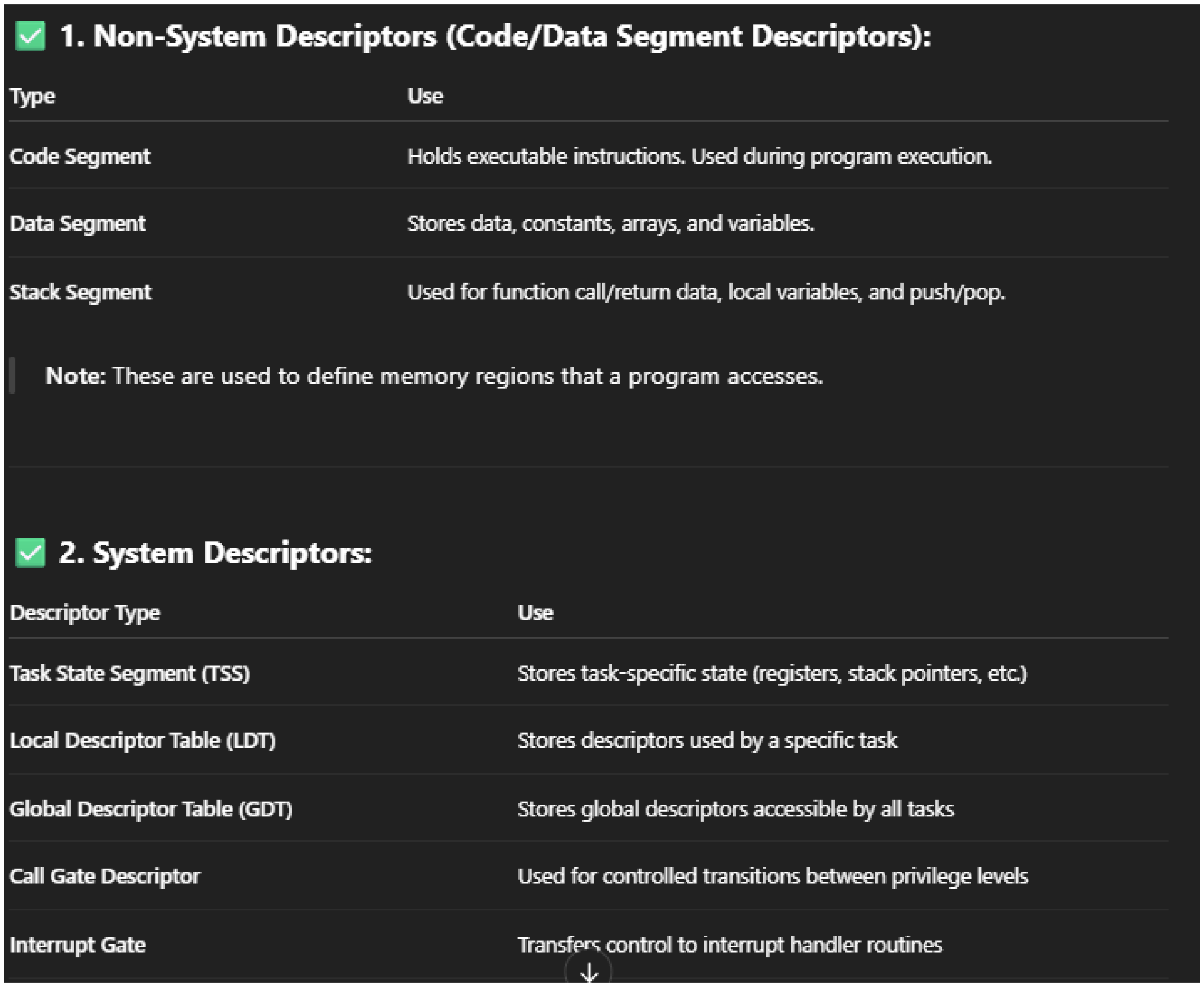
D

4) **With the necessary Diagram, Explain the complete Address Translation process in 80386.**



```
Logical Address (Selector:Offset)
             |
             v
    +----------------------+
    | Segment Selector     |
    | (GDT/LDT Access)     |
    +----------------------+
             |
             v
    +----------------------+
    | Segment Descriptor   |
    | (Base + Limit)       |
    +----------------------+
             |
             v
    Linear Address = Base + Offset
             |
         (If Paging Enabled)
             |
             v
    +----------------------------+
    | Paging Unit                |
    | - Page Directory Index     |
    | - Page Table Index         |
    | - Page Offset              |
    +----------------------------+
             |
             v
    Final Physical Address
```

5) **Enlist various types of system & non-system descriptors in 80386. Explain their use in brief.**



✅ 1. Non-System Descriptors (Code/Data Segment Descriptors):

Type	Use
Code Segment	Holds executable instructions. Used during program execution.
Data Segment	Stores data, constants, arrays, and variables.
Stack Segment	Used for function call/return data, local variables, and push/pop.

> **Note:** These are used to define memory regions that a program accesses.

✅ 2. System Descriptors:

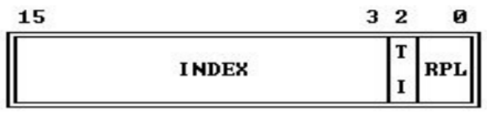
Descriptor Type	Use
Task State Segment (TSS)	Stores task-specific state (registers, stack pointers, etc.)
Local Descriptor Table (LDT)	Stores descriptors used by a specific task
Global Descriptor Table (GDT)	Stores global descriptors accessible by all tasks
Call Gate Descriptor	Used for controlled transitions between privilege levels
Interrupt Gate	Transfers control to interrupt handler routines

6) **Draw & Explain the General Selector Format.**

3.Selectors

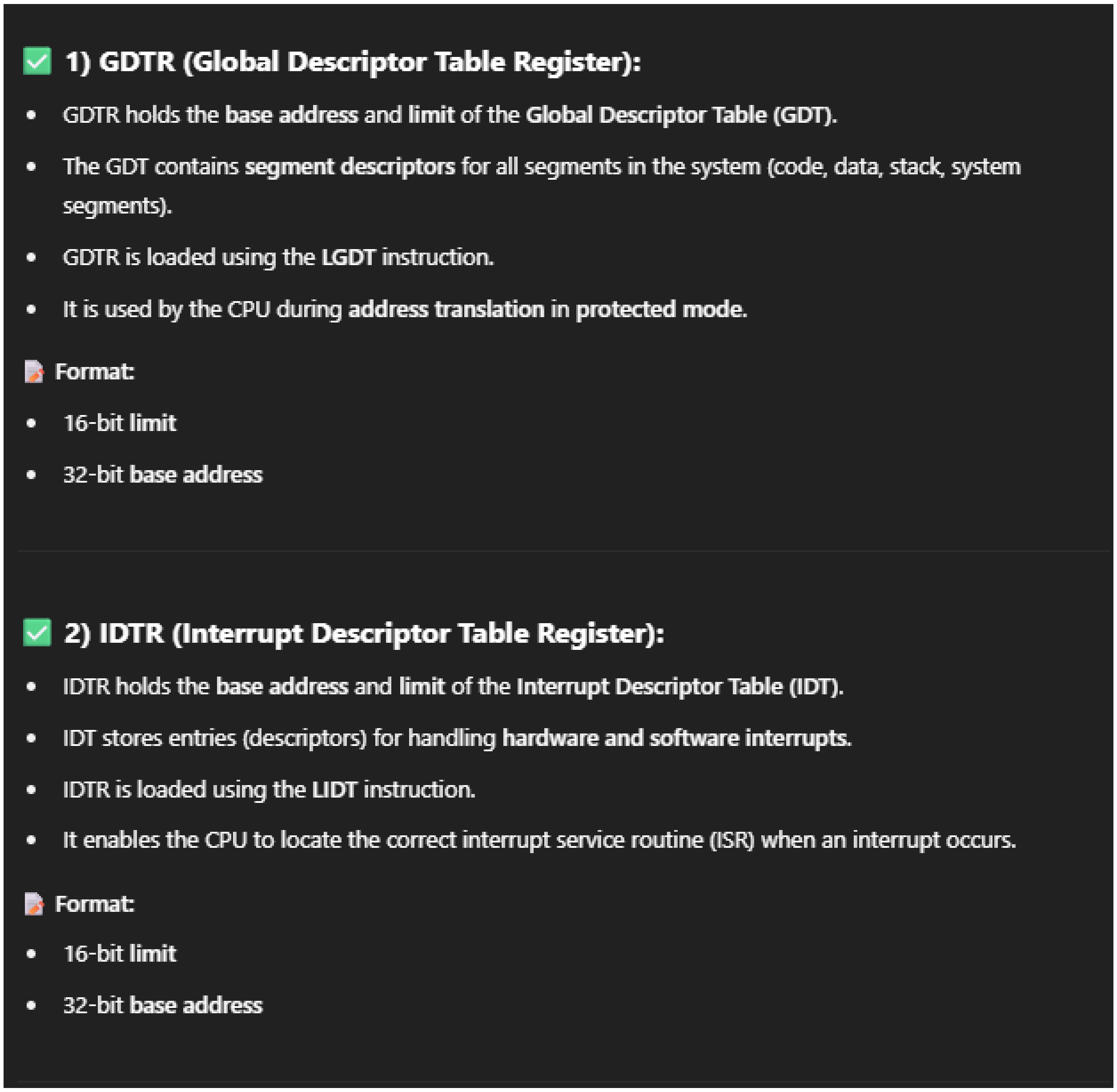
- The selector portion of a logical address identifies a descriptor by specifying a descriptor table and indexing a descriptor within that table. Selectors may be visible to applications programs as a field within a pointer variable, but the values of selectors are usually assigned (fixed up) by linkers or linking loaders.
- Table Indicator: Specifies to which descriptor table the selector refers. A zero indicates the GDT; a one indicates the current LDT.
- Requested Privilege Level: Used by the protection mechanism.

Format of a Selector



TI – TABLE INDICATOR
RPL – REQUESTOR'S PRIVILEGE LEVEL

7) Write a short note on GDTR, IDTR, LDTR



✅ 1) GDTR (Global Descriptor Table Register):

- GDTR holds the **base address** and **limit** of the **Global Descriptor Table (GDT).**
- The GDT contains **segment descriptors** for all segments in the system (code, data, stack, system segments).
- GDTR is loaded using the **LGDT** instruction.
- It is used by the CPU during **address translation** in **protected mode.**

📝 **Format:**

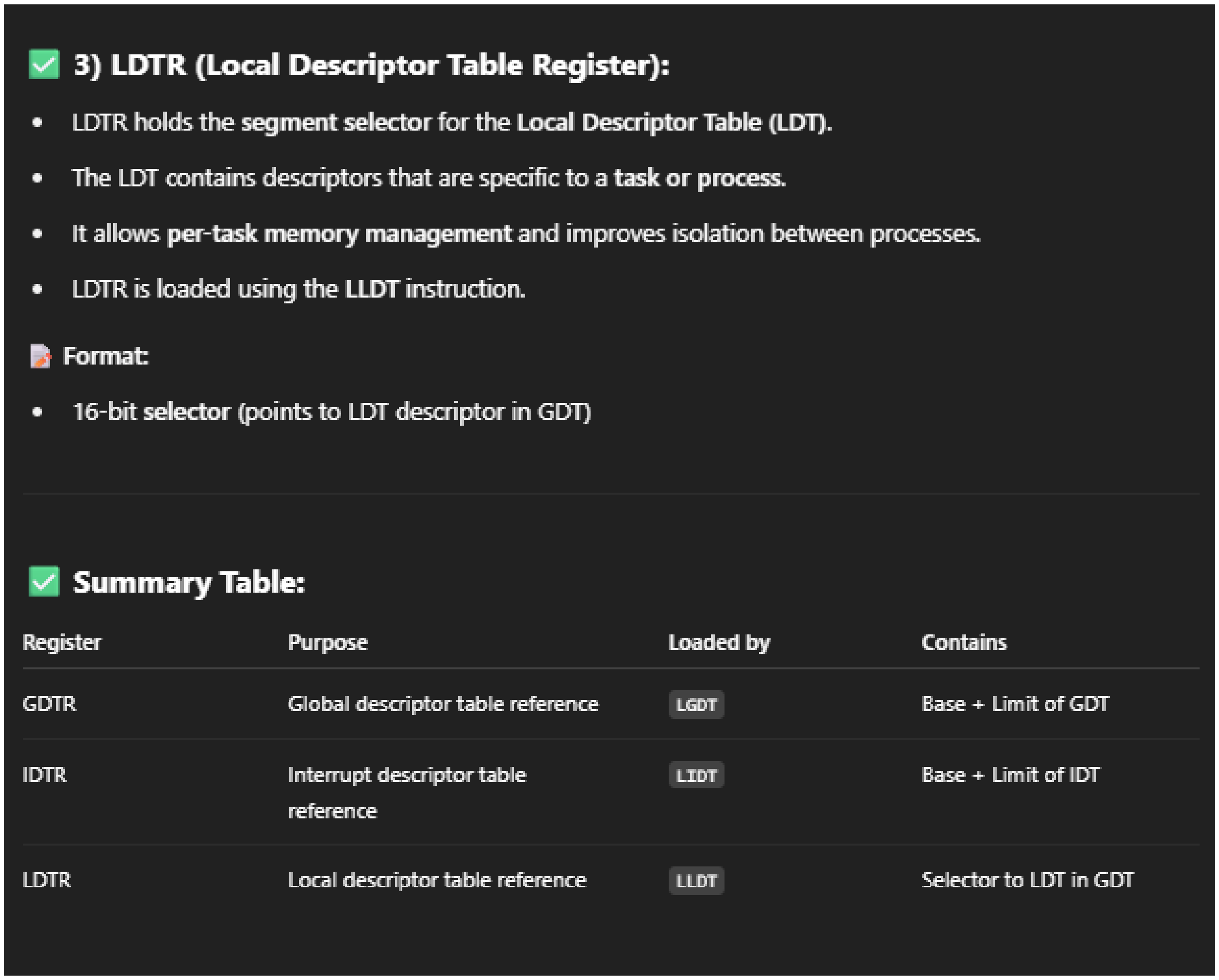
- 16-bit **limit**
- 32-bit **base address**

✅ 2) IDTR (Interrupt Descriptor Table Register):

- IDTR holds the **base address** and **limit** of the **Interrupt Descriptor Table (IDT).**
- IDT stores entries (descriptors) for handling **hardware and software interrupts.**
- IDTR is loaded using the **LIDT** instruction.
- It enables the CPU to locate the correct interrupt service routine (ISR) when an interrupt occurs.

📝 **Format:**

- 16-bit **limit**
- 32-bit **base address**



✅ 3) LDTR (Local Descriptor Table Register):

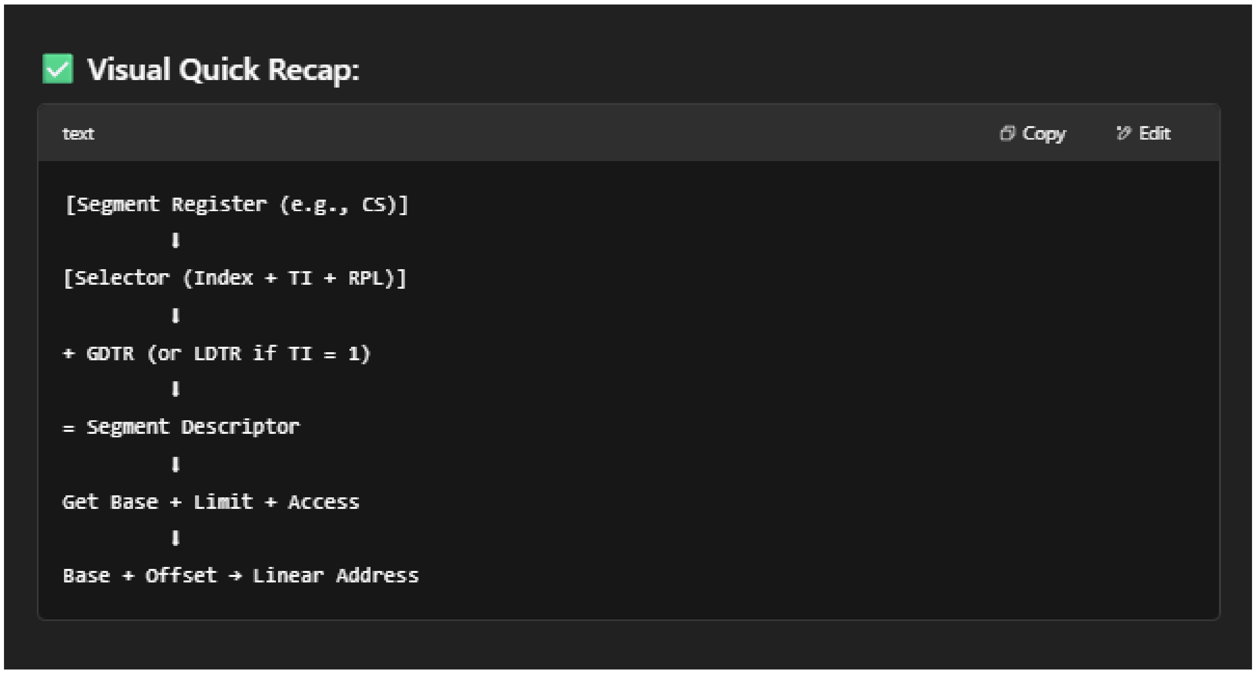
- LDTR holds the **segment selector** for the **Local Descriptor Table (LDT).**
- The LDT contains descriptors that are specific to a **task or process.**
- It allows **per-task memory management** and improves isolation between processes.
- LDTR is loaded using the **LLDT** instruction.

📝 **Format:**

- 16-bit **selector** (points to LDT descriptor in GDT)

✅ Summary Table:

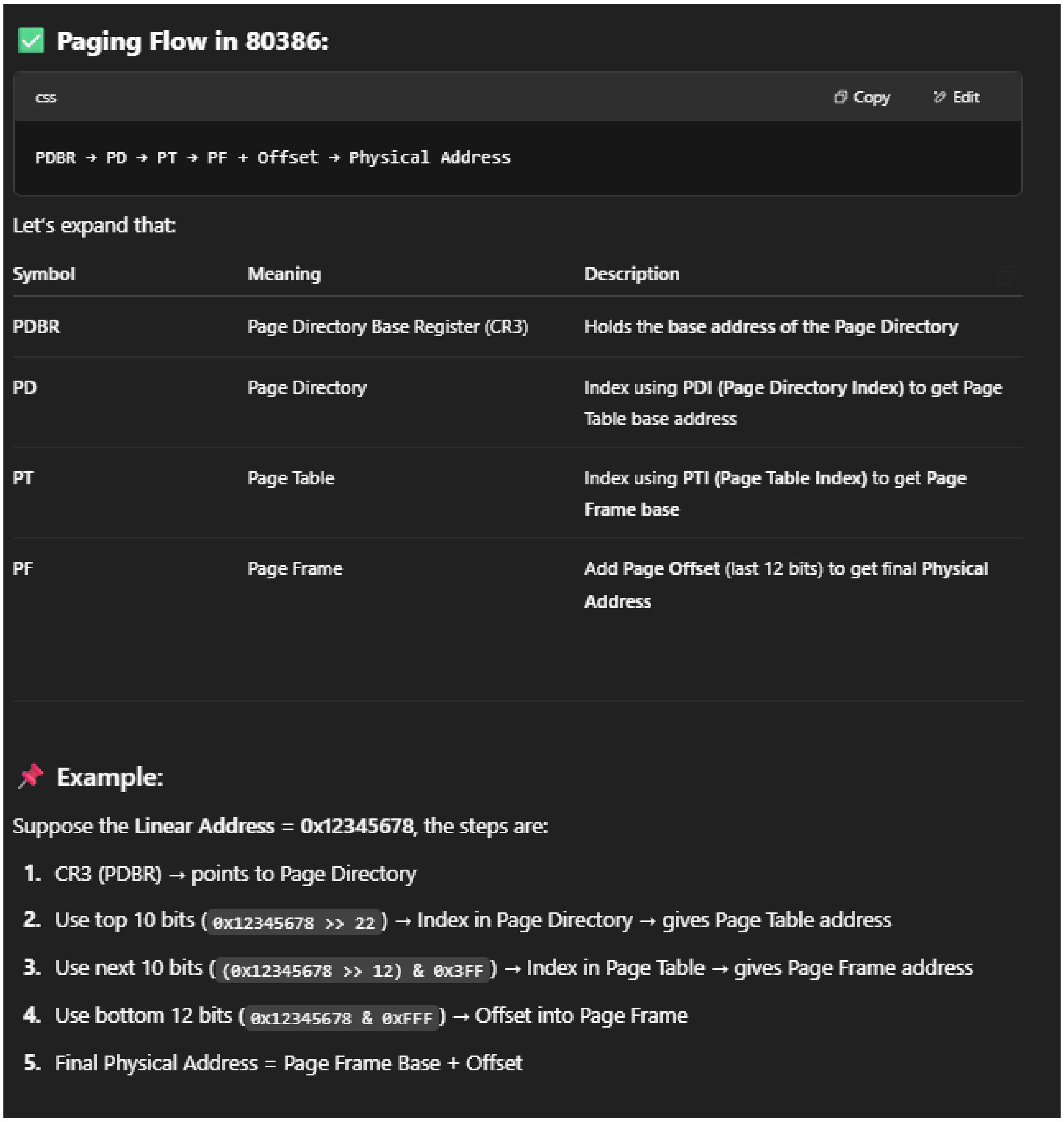
Register	Purpose	Loaded by	Contains
GDTR	Global descriptor table reference	`LGDT`	Base + Limit of GDT
IDTR	Interrupt descriptor table reference	`LIDT`	Base + Limit of IDT
LDTR	Local descriptor table reference	`LLDT`	Selector to LDT in GDT



✅ Visual Quick Recap:

```text
[Segment Register (e.g., CS)]
         ⬇
[Selector (Index + TI + RPL)]
         ⬇
+ GDTR (or LDTR if TI = 1)
         ⬇
= Segment Descriptor
         ⬇
Get Base + Limit + Access
         ⬇
Base + Offset → Linear Address
```

8) Explain the page translation process in 80386.



✅ Paging Flow in 80386:

css Copy Edit

```css
PDBR → PD → PT → PF + Offset → Physical Address
```

Let's expand that:

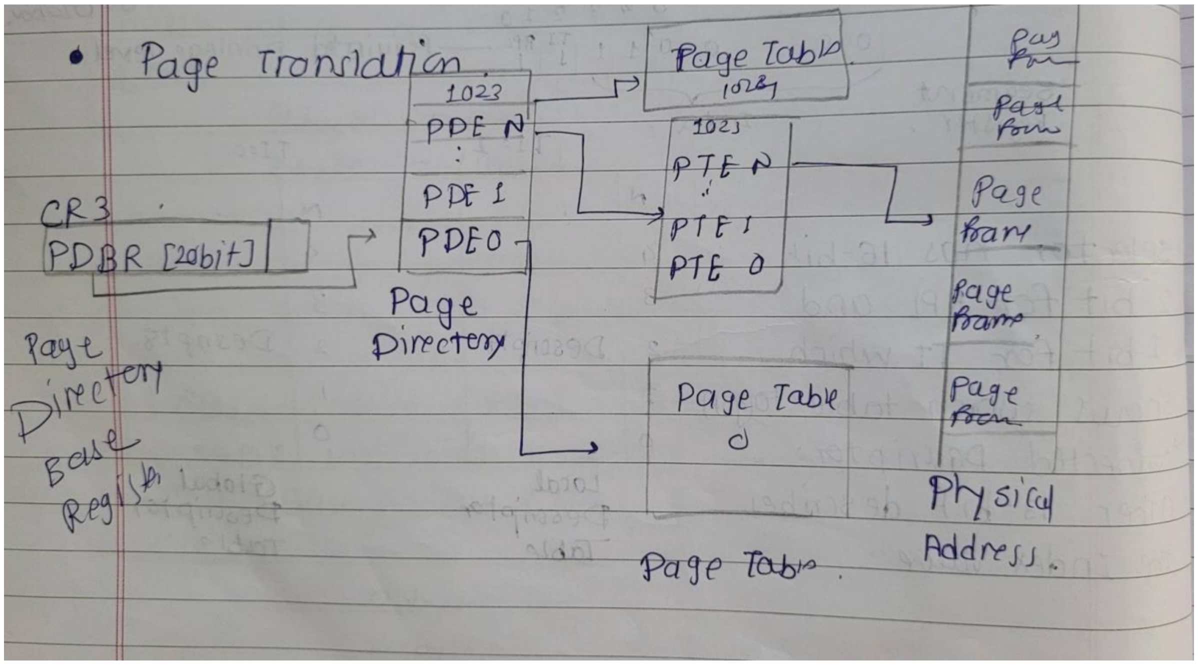
Symbol	Meaning	Description
PDBR	Page Directory Base Register (CR3)	Holds the base address of the Page Directory
PD	Page Directory	Index using PDI (Page Directory Index) to get Page Table base address
PT	Page Table	Index using PTI (Page Table Index) to get Page Frame base
PF	Page Frame	Add Page Offset (last 12 bits) to get final Physical Address

📌 Example:

Suppose the **Linear Address** = **0x12345678**, the steps are:

1. CR3 (PDBR) → points to Page Directory
2. Use top 10 bits (`0x12345678 >> 22`) → Index in Page Directory → gives Page Table address
3. Use next 10 bits (`(0x12345678 >> 12) & 0x3FF`) → Index in Page Table → gives Page Frame address
4. Use bottom 12 bits (`0x12345678 & 0xFFF`) → Offset into Page Frame
5. Final Physical Address = Page Frame Base + Offset

• Page Translation.



1] PDBR of 20bit holds ~~the address of~~ base address of Page Direcoty.
2] Page Directory has Total ~~1~~023 Page Entries. Each PDE has size of 4Bytes.
3] Each PDE holds address of Page Table. Each Page Table entry has 4 Bytes of spaces. ~~&~~ A particular Table contains total 1024 Page Table entries
4] Page Table entry holds base address of page frame
5] Each page frame is physical memory size of 4KB.

$\underbrace{\text{selector(16) offset(32)}}$ — virtual addres.

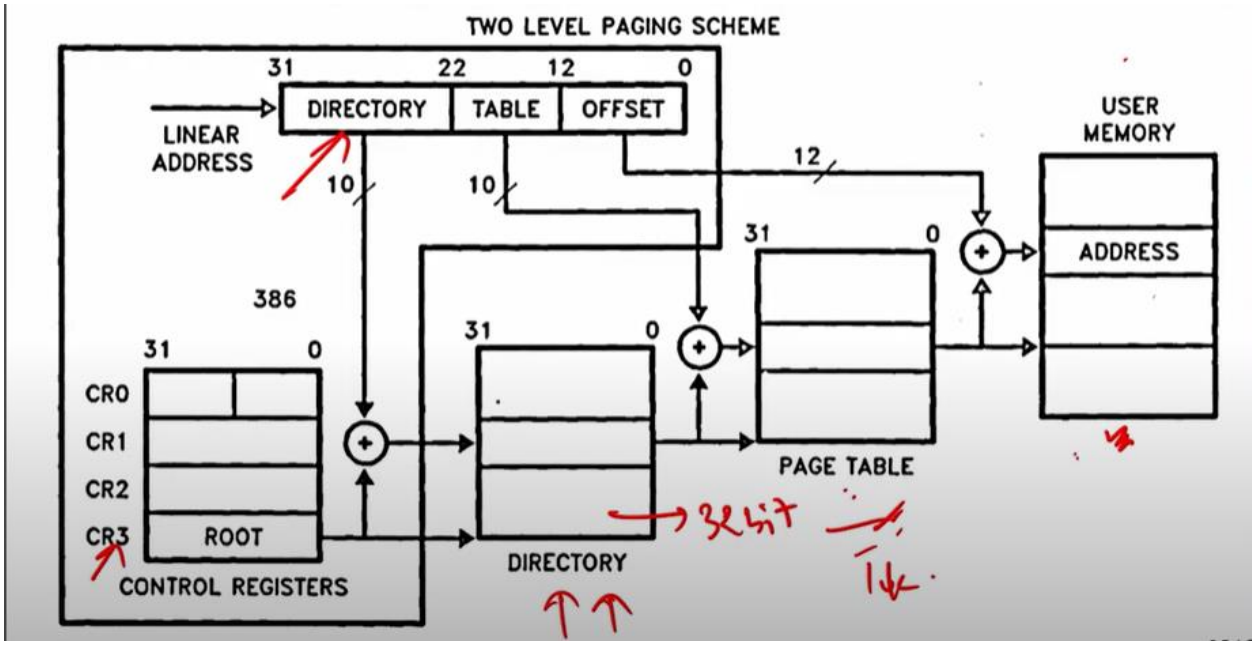
$$\begin{tikzcd}
 & \text{segmentation.} \arrow[d] & \text{— linear address.} \\
 & \text{PG} \arrow[d, "\text{Yes, PG = 1.}"] \arrow[dd, bend right=60, "\text{NO, PG=0}"'] & \\
 & \text{Paging} \arrow[d] & \\
 & \text{Physical address.} &
\end{tikzcd}$$

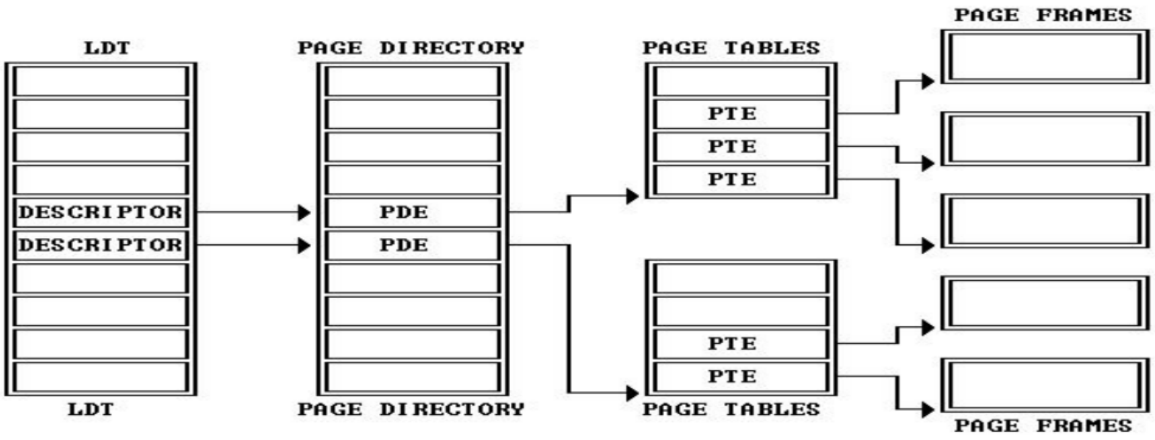
PDE
Page Table address =: Giress address of page Table.
User - 3 bits are not defined & left for use.
A = Access - bit automatically set when PDE is used in address translation.
User/Superviser = If U/S = 0 Page converd by this PDE are accessible only to Supervisor mode.

R/W - Read write Protection :- U/S = 1, then no meaning, U/S = 0, If R/W = 0, only read allowed, if R/W = 1 then both allowed.

P - Preset = P = 1 = Preset.

PTE
Page frame address,
the bit is set 1 whenever the write operation is performed on Page pointed by PTE.

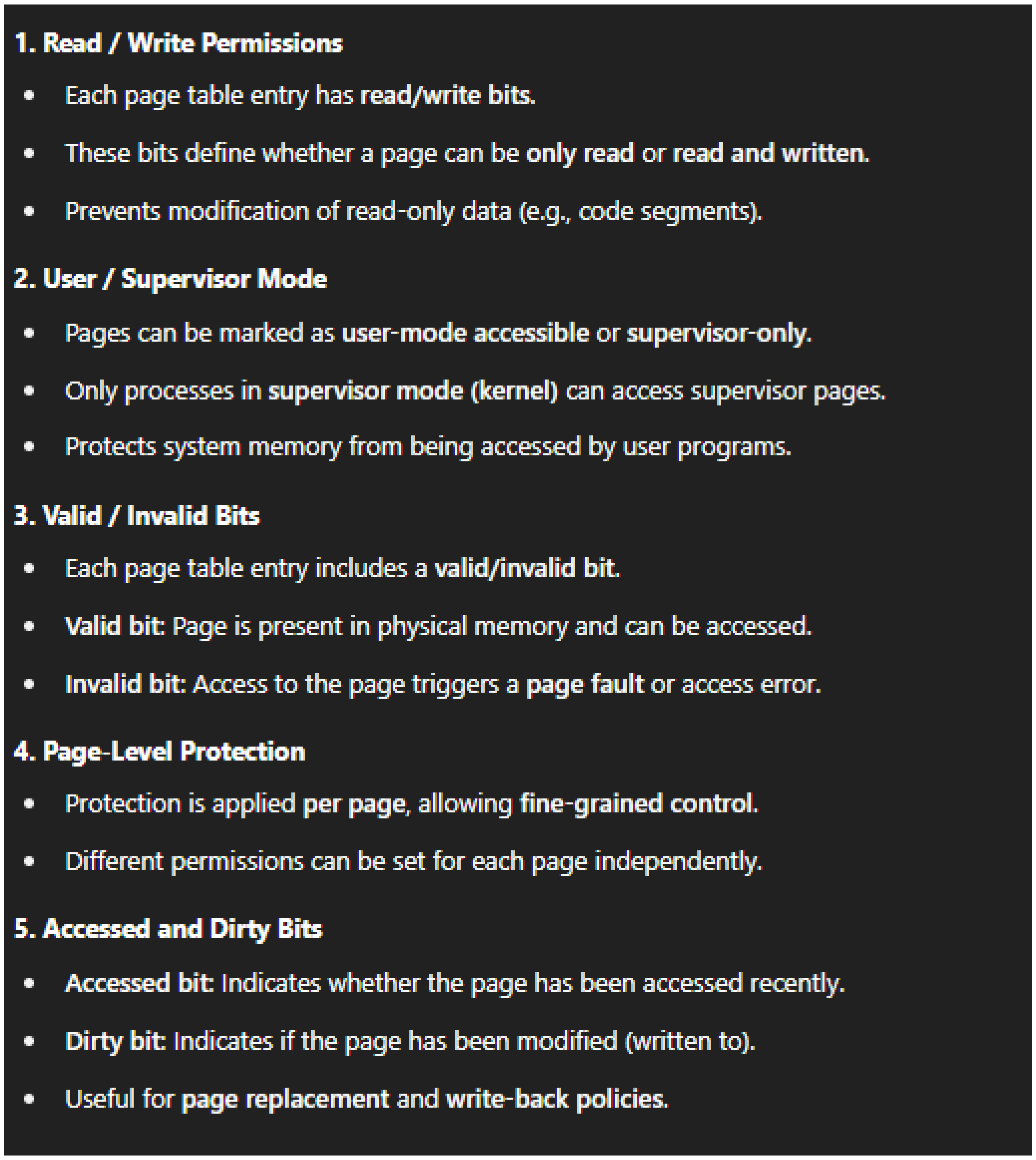




Descriptor per page table

UNIT 4

1) **Explain various Aspects of Protection Mechanism of Paging unit.**



1. Read / Write Permissions

- Each page table entry has **read/write bits.**
- These bits define whether a page can be **only read** or **read and written.**
- Prevents modification of read-only data (e.g., code segments).

2. User / Supervisor Mode

- Pages can be marked as **user-mode accessible** or **supervisor-only.**
- Only processes in **supervisor mode (kernel)** can access supervisor pages.
- Protects system memory from being accessed by user programs.

3. Valid / Invalid Bits

- Each page table entry includes a **valid/invalid bit.**
- **Valid bit:** Page is present in physical memory and can be accessed.
- **Invalid bit:** Access to the page triggers a **page fault** or access error.

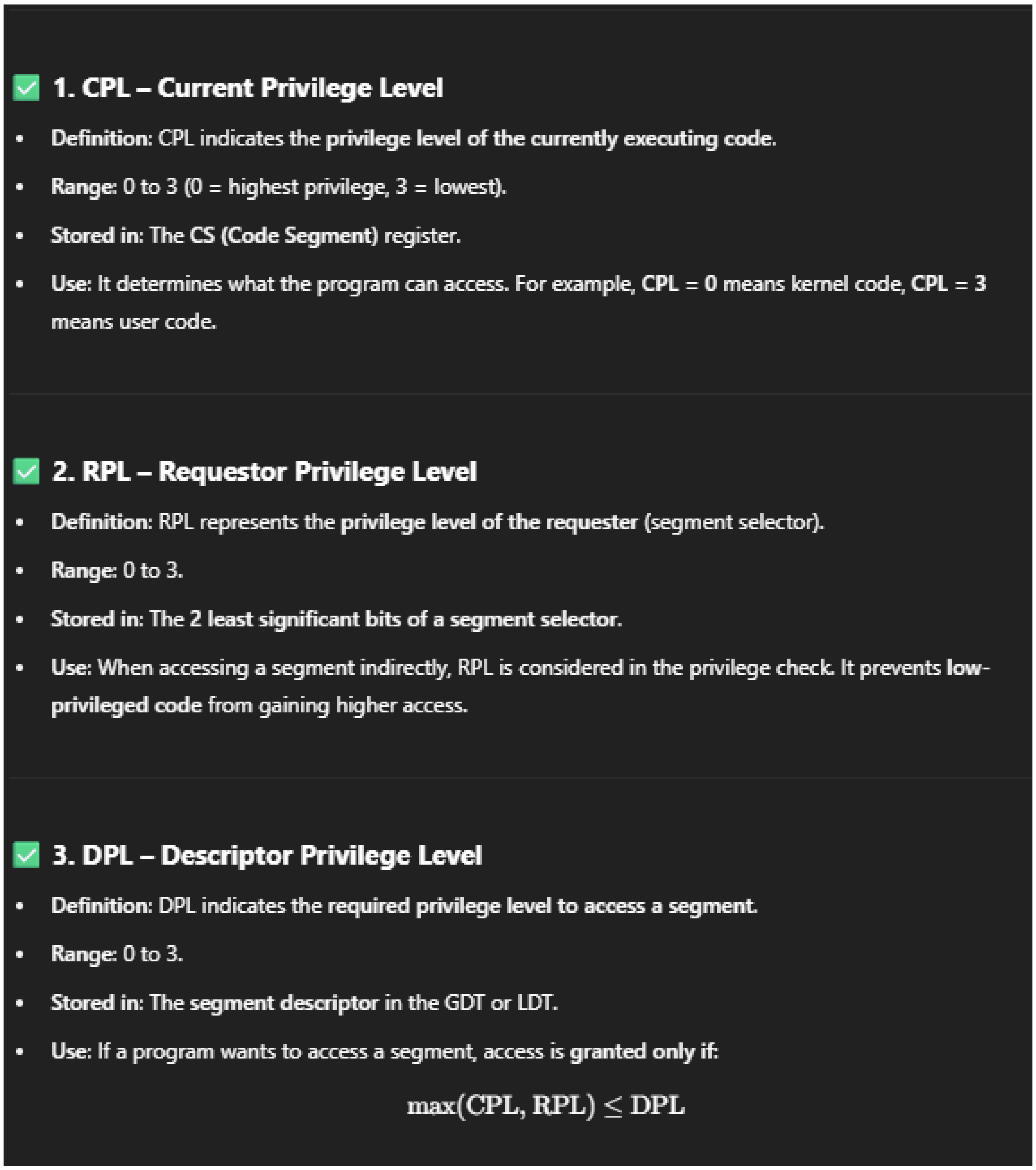
4. Page-Level Protection

- Protection is applied **per page**, allowing **fine-grained control.**
- Different permissions can be set for each page independently.

5. Accessed and Dirty Bits

- **Accessed bit:** Indicates whether the page has been accessed recently.
- **Dirty bit:** Indicates if the page has been modified (written to).
- Useful for **page replacement** and **write-back policies.**

2) What is CPL, EPL, IOPL? Explain in Brief.



✅ 1. CPL – Current Privilege Level

- **Definition:** CPL indicates the **privilege level of the currently executing code.**
- **Range:** 0 to 3 (0 = highest privilege, 3 = lowest).
- **Stored in:** The **CS (Code Segment)** register.
- **Use:** It determines what the program can access. For example, **CPL = 0** means kernel code, **CPL = 3** means user code.

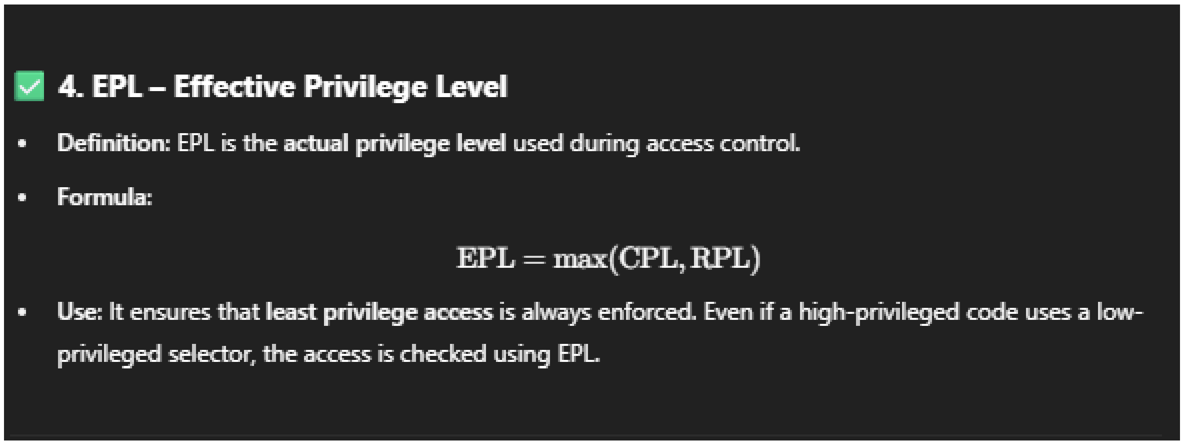
✅ 2. RPL – Requestor Privilege Level

- **Definition:** RPL represents the **privilege level of the requester** (segment selector).
- **Range:** 0 to 3.
- **Stored in:** The **2 least significant bits of a segment selector.**
- **Use:** When accessing a segment indirectly, RPL is considered in the privilege check. It prevents **low-privileged code** from gaining higher access.

✅ 3. DPL – Descriptor Privilege Level

- **Definition:** DPL indicates the **required privilege level to access a segment.**
- **Range:** 0 to 3.
- **Stored in:** The **segment descriptor** in the GDT or LDT.
- **Use:** If a program wants to access a segment, access is **granted only if:**

$$\max(\text{CPL}, \text{RPL}) \leq \text{DPL}$$

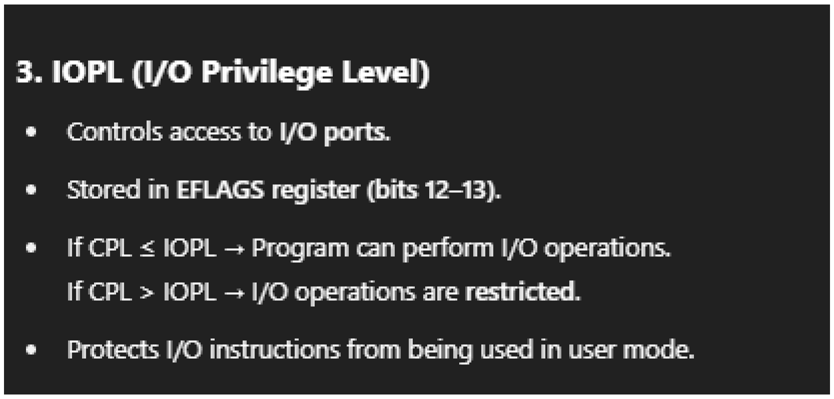


✅ 4. EPL – Effective Privilege Level

- **Definition:** EPL is the **actual privilege level** used during access control.
- **Formula:**

$$\text{EPL} = \max(\text{CPL}, \text{RPL})$$

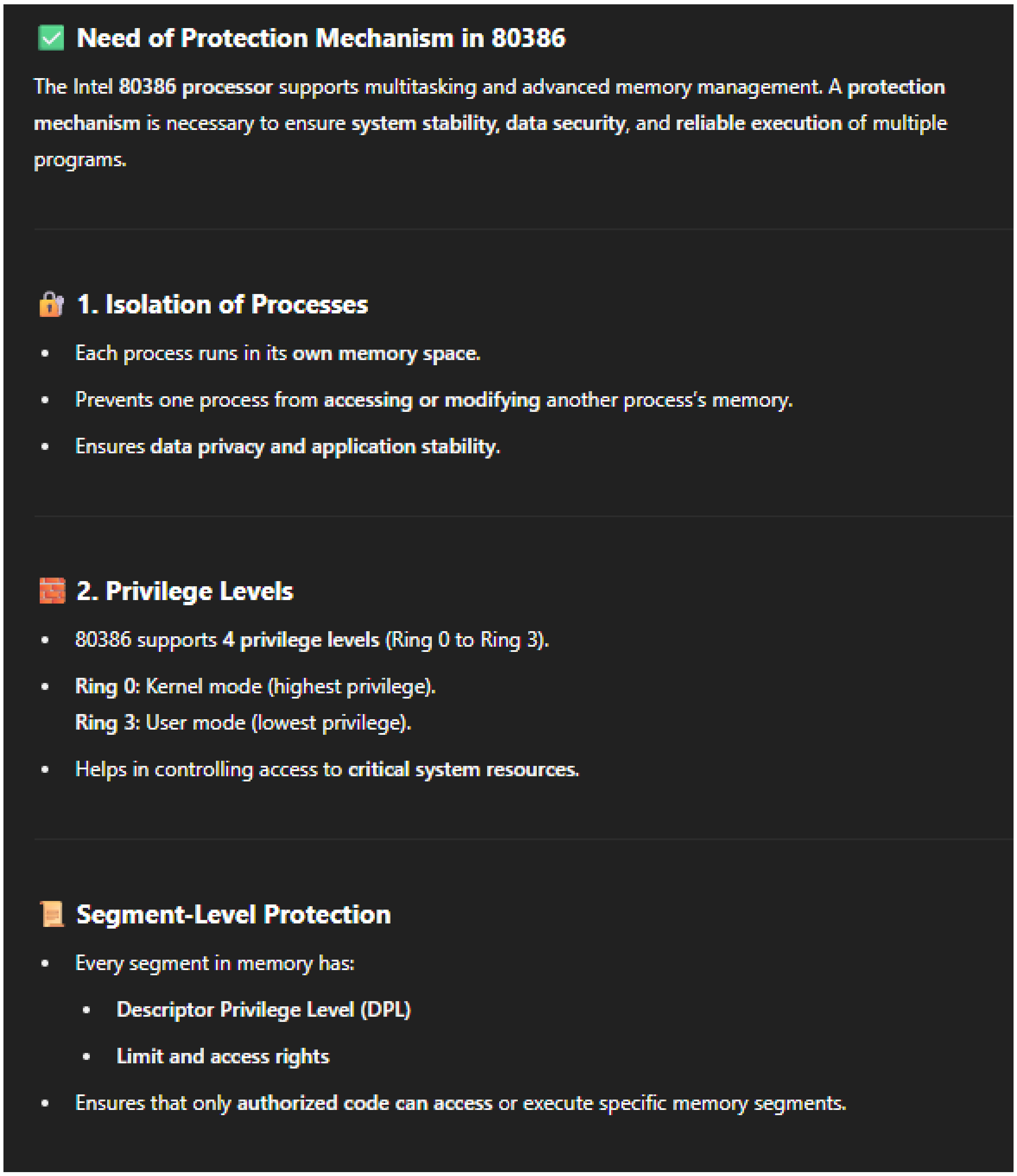
- **Use:** It ensures that **least privilege access** is always enforced. Even if a high-privileged code uses a low-privileged selector, the access is checked using EPL.



3. IOPL (I/O Privilege Level)

- Controls access to **I/O ports.**
- Stored in **EFLAGS register (bits 12–13).**
- If CPL ≤ IOPL → Program can perform I/O operations.
  If CPL > IOPL → I/O operations are **restricted.**
- Protects I/O instructions from being used in user mode.

3) Explain the need of Protection Mechanism in 80386.



✅ Need of Protection Mechanism in 80386

The Intel **80386 processor** supports multitasking and advanced memory management. A **protection mechanism** is necessary to ensure **system stability, data security**, and **reliable execution** of multiple programs.

🔐 1. Isolation of Processes

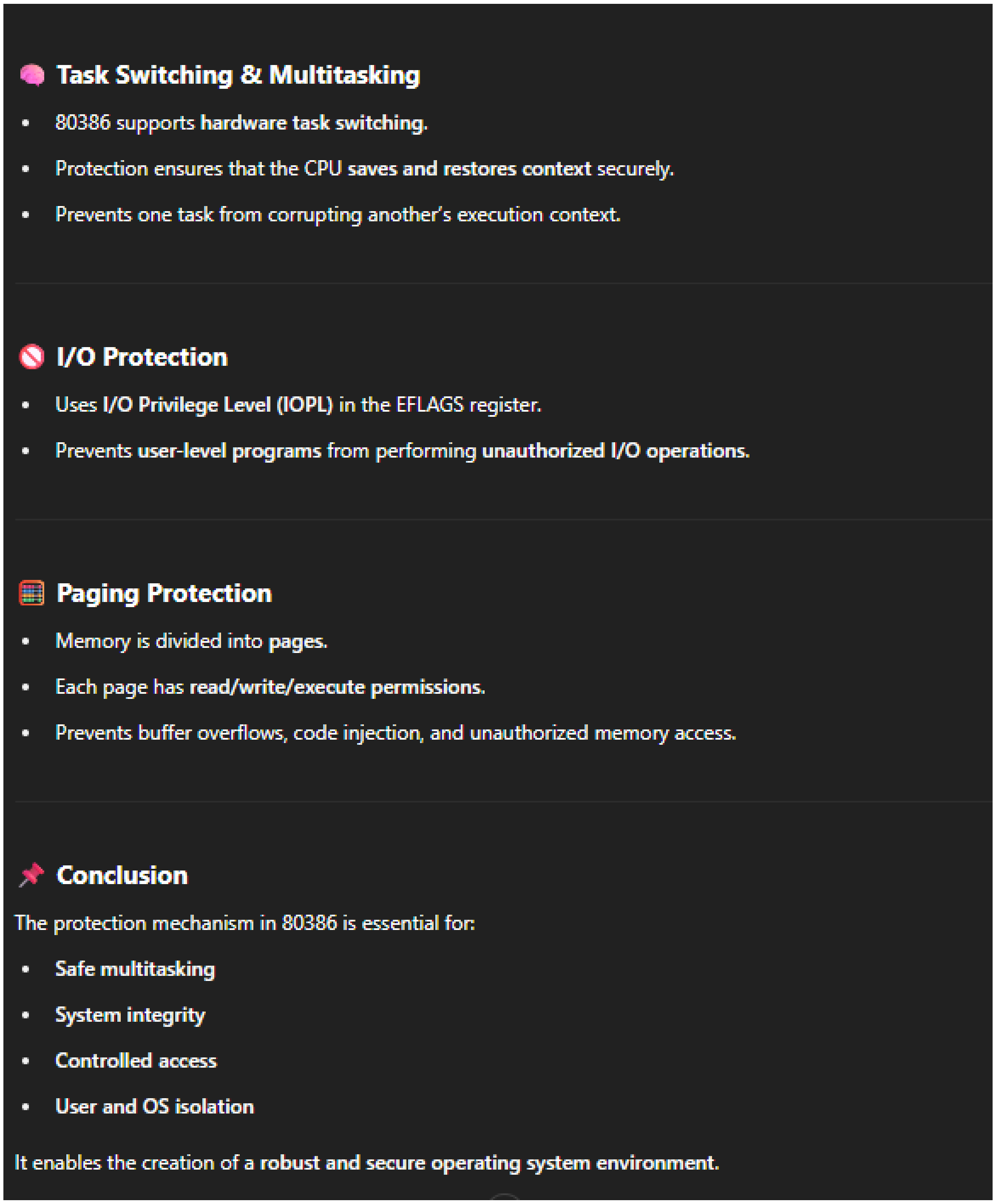
- Each process runs in its **own memory space.**
- Prevents one process from **accessing or modifying** another process's memory.
- Ensures **data privacy and application stability.**

🧱 2. Privilege Levels

- 80386 supports **4 privilege levels** (Ring 0 to Ring 3).
- **Ring 0:** Kernel mode (highest privilege).
  Ring 3: User mode (lowest privilege).
- Helps in controlling access to **critical system resources.**

📜 Segment-Level Protection

- Every segment in memory has:
  - **Descriptor Privilege Level (DPL)**
  - **Limit and access rights**
- Ensures that only **authorized code can access** or execute specific memory segments.



🧠 Task Switching & Multitasking

- 80386 supports **hardware task switching.**
- Protection ensures that the CPU **saves and restores context** securely.
- Prevents one task from corrupting another's execution context.

🚫 I/O Protection

- Uses **I/O Privilege Level (IOPL)** in the EFLAGS register.
- Prevents **user-level programs** from performing **unauthorized I/O operations.**

🧮 Paging Protection

- Memory is divided into **pages.**
- Each page has **read/write/execute permissions.**
- Prevents buffer overflows, code injection, and unauthorized memory access.

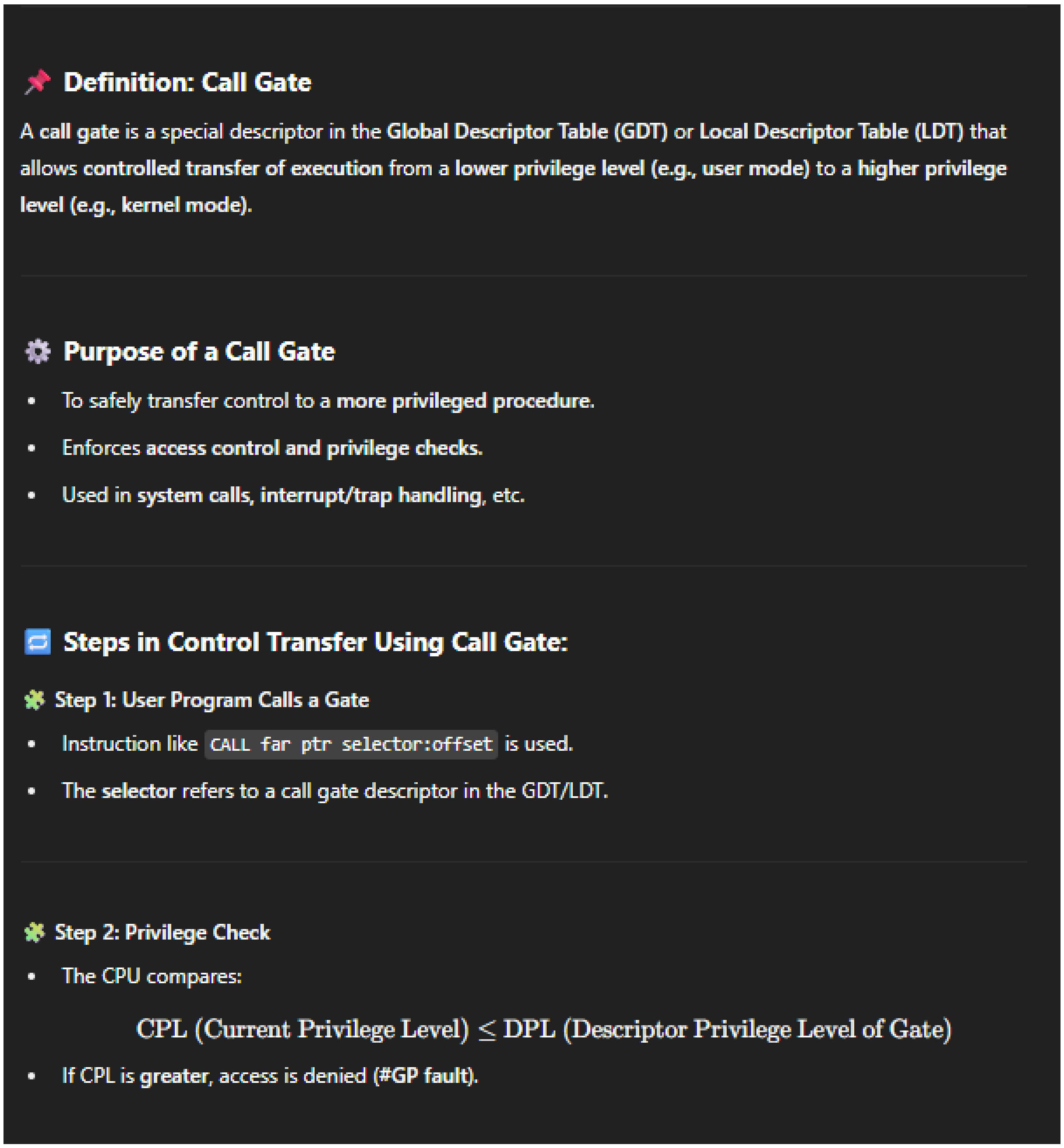
📌 Conclusion

The protection mechanism in 80386 is essential for:

- **Safe multitasking**
- **System integrity**
- **Controlled access**
- **User and OS isolation**

It enables the creation of a **robust and secure operating system environment.**

4) Explain how control transfer Instructions are executed using the call gate in the system.



📌 Definition: Call Gate

A **call gate** is a special descriptor in the **Global Descriptor Table (GDT)** or **Local Descriptor Table (LDT)** that allows **controlled transfer of execution** from a **lower privilege level (e.g., user mode)** to a **higher privilege level (e.g., kernel mode).**

⚙️ Purpose of a Call Gate

- To safely transfer control to a **more privileged procedure.**
- Enforces **access control and privilege checks.**
- Used in **system calls, interrupt/trap handling**, etc.

🔁 Steps in Control Transfer Using Call Gate:

🧩 Step 1: User Program Calls a Gate

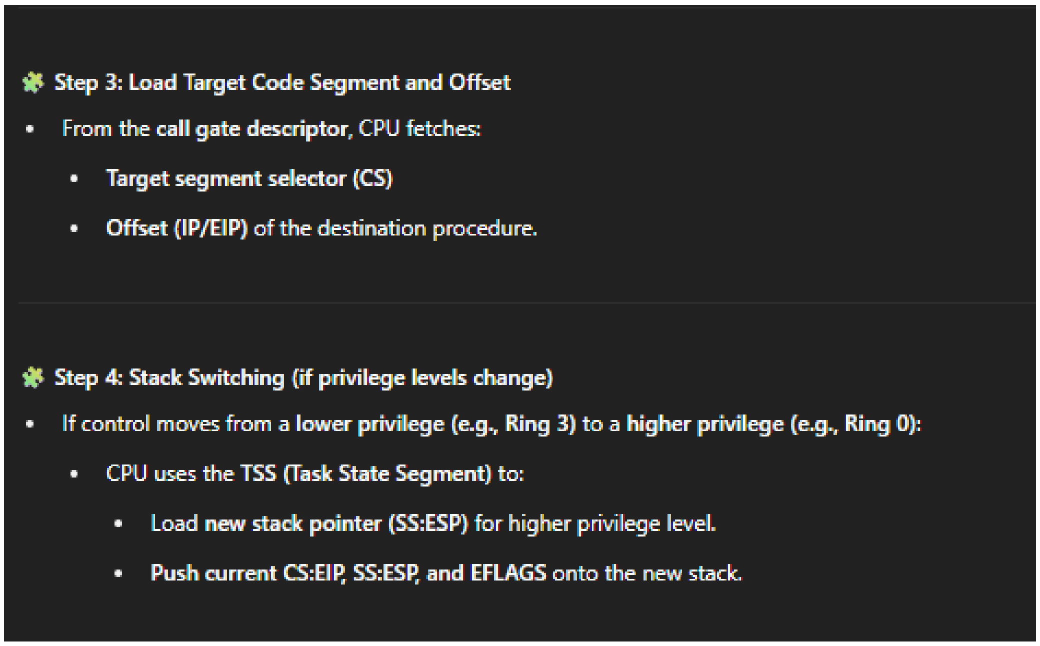
- Instruction like `CALL far ptr selector:offset` is used.
- The **selector** refers to a call gate descriptor in the GDT/LDT.

🧩 Step 2: Privilege Check

- The CPU compares:

$$\text{CPL (Current Privilege Level)} \leq \text{DPL (Descriptor Privilege Level of Gate)}$$

- If CPL is **greater**, access is denied (**#GP fault**).

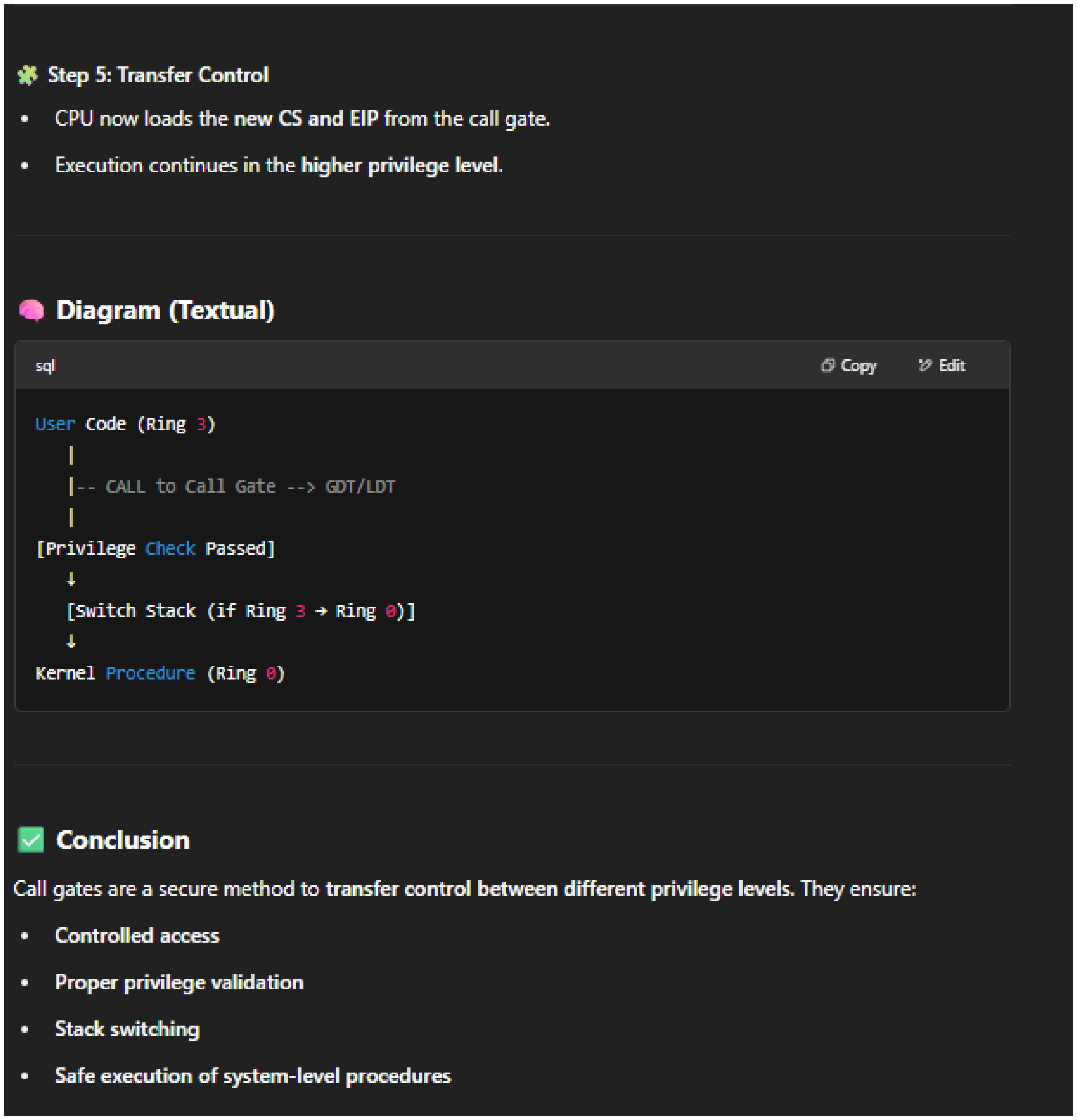


🧩 **Step 3: Load Target Code Segment and Offset**

- From the **call gate descriptor**, CPU fetches:
  - **Target segment selector (CS)**
  - **Offset (IP/EIP)** of the destination procedure.

🧩 **Step 4: Stack Switching (if privilege levels change)**

- If control moves from a **lower privilege (e.g., Ring 3)** to a **higher privilege (e.g., Ring 0)**:
  - CPU uses the **TSS (Task State Segment)** to:
    - Load **new stack pointer (SS:ESP)** for higher privilege level.
    - **Push current CS:EIP, SS:ESP, and EFLAGS** onto the new stack.



🧩 **Step 5: Transfer Control**

- CPU now loads the **new CS and EIP** from the call gate.
- Execution continues in the **higher privilege level**.

🧠 Diagram (Textual)

sql

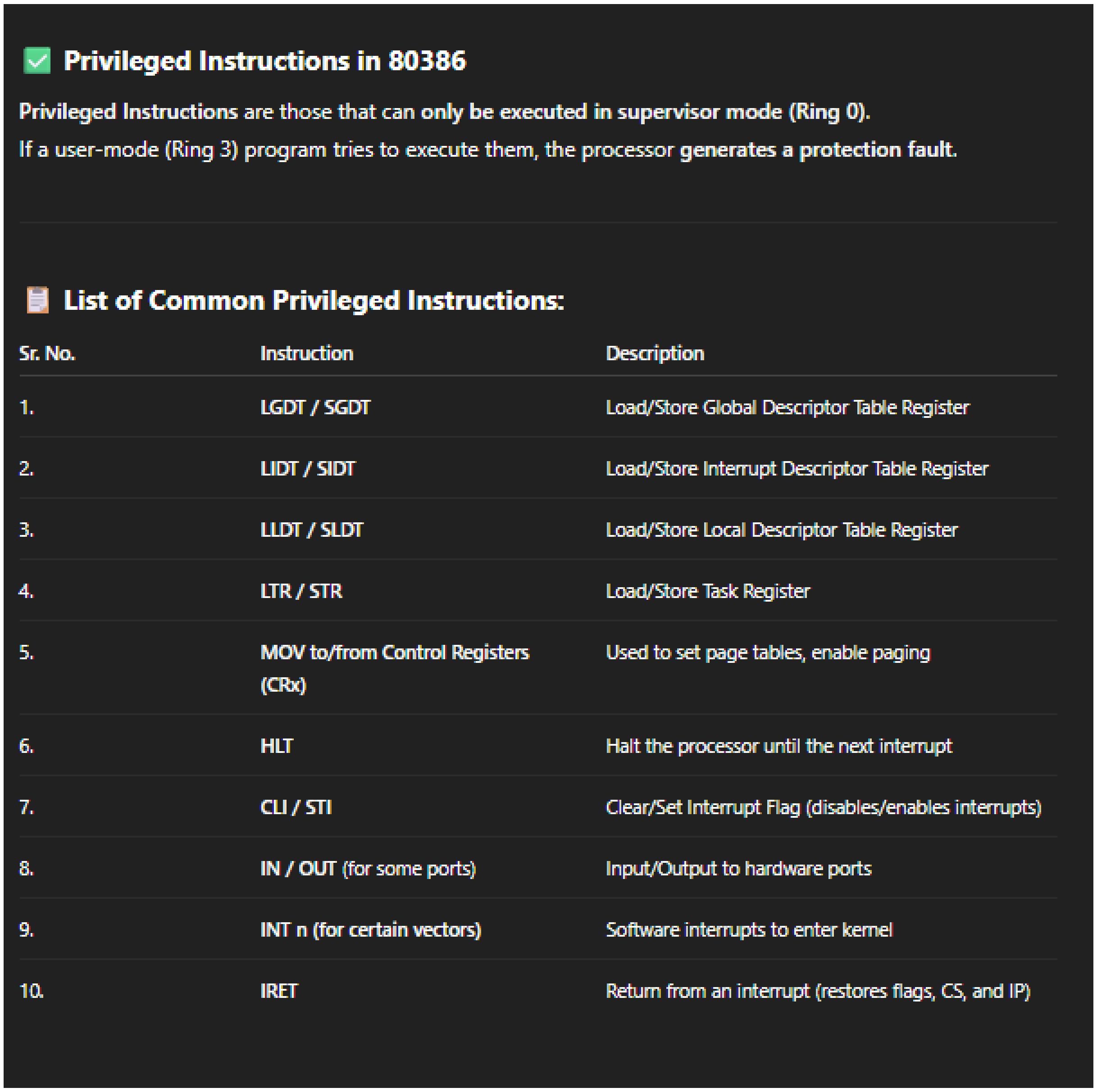
```sql
User Code (Ring 3)
   |
   |-- CALL to Call Gate --> GDT/LDT
   |
[Privilege Check Passed]
   ↓
   [Switch Stack (if Ring 3 → Ring 0)]
   ↓
Kernel Procedure (Ring 0)
```

✅ Conclusion

Call gates are a secure method to **transfer control between different privilege levels.** They ensure:

- **Controlled access**
- **Proper privilege validation**
- **Stack switching**
- **Safe execution of system-level procedures**

5) **List & Explain various Privilege Instructions.**

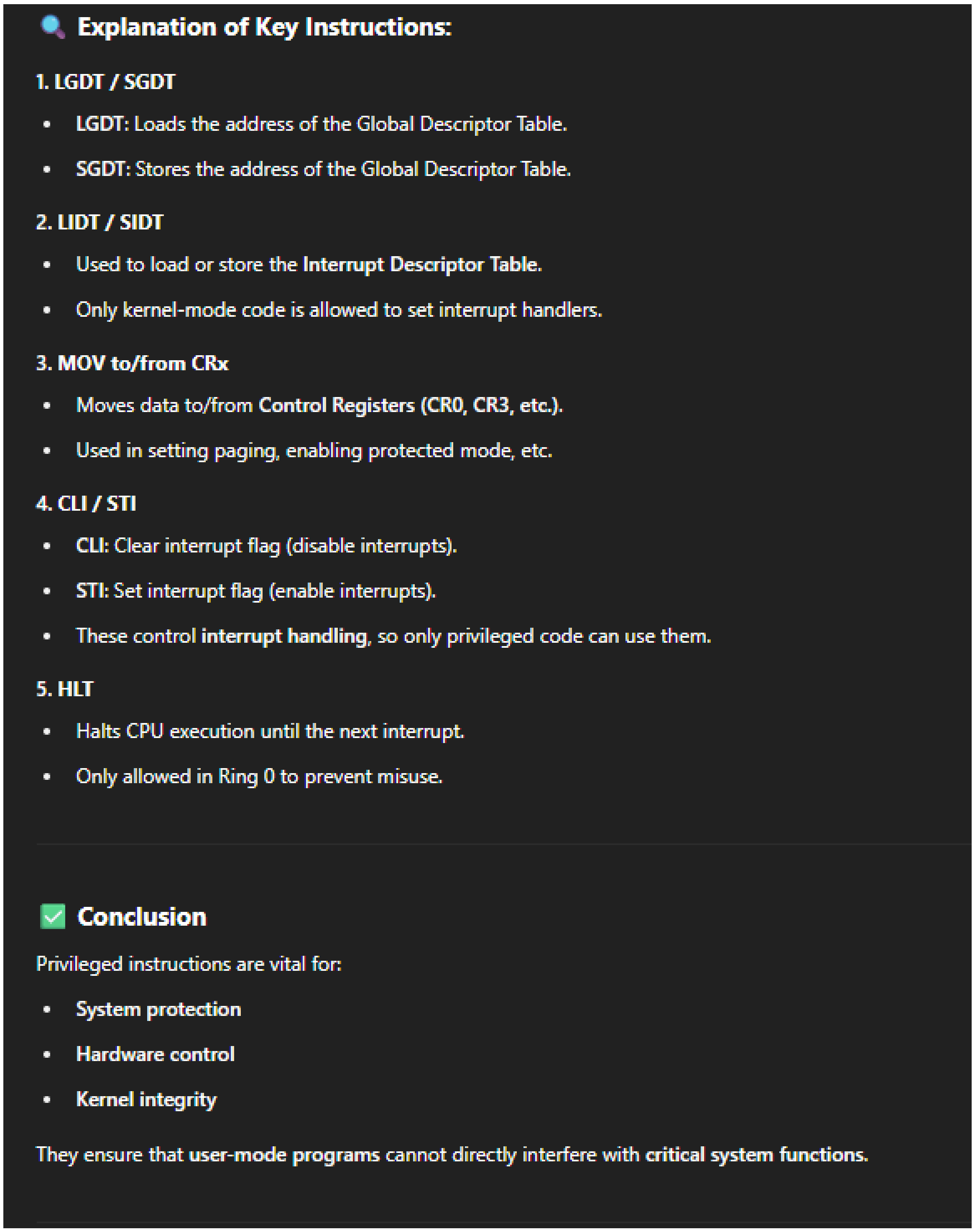


✅ Privileged Instructions in 80386

Privileged Instructions are those that can **only be executed in supervisor mode (Ring 0).**
If a user-mode (Ring 3) program tries to execute them, the processor **generates a protection fault.**

📋 List of Common Privileged Instructions:

Sr. No.	Instruction	Description
1.	**LGDT / SGDT**	Load/Store Global Descriptor Table Register
2.	**LIDT / SIDT**	Load/Store Interrupt Descriptor Table Register
3.	**LLDT / SLDT**	Load/Store Local Descriptor Table Register
4.	**LTR / STR**	Load/Store Task Register
5.	**MOV to/from Control Registers (CRx)**	Used to set page tables, enable paging
6.	**HLT**	Halt the processor until the next interrupt
7.	**CLI / STI**	Clear/Set Interrupt Flag (disables/enables interrupts)
8.	**IN / OUT** (for some ports)	Input/Output to hardware ports
9.	**INT n** (for certain vectors)	Software interrupts to enter kernel
10.	**IRET**	Return from an interrupt (restores flags, CS, and IP)



🔍 Explanation of Key Instructions:

1. LGDT / SGDT

- **LGDT**: Loads the address of the Global Descriptor Table.
- **SGDT**: Stores the address of the Global Descriptor Table.

2. LIDT / SIDT

- Used to load or store the **Interrupt Descriptor Table.**
- Only kernel-mode code is allowed to set interrupt handlers.

3. MOV to/from CRx

- Moves data to/from **Control Registers (CR0, CR3, etc.).**
- Used in setting paging, enabling protected mode, etc.

4. CLI / STI

- **CLI**: Clear interrupt flag (disable interrupts).
- **STI**: Set interrupt flag (enable interrupts).
- These control **interrupt handling**, so only privileged code can use them.

5. HLT

- Halts CPU execution until the next interrupt.
- Only allowed in Ring 0 to prevent misuse.

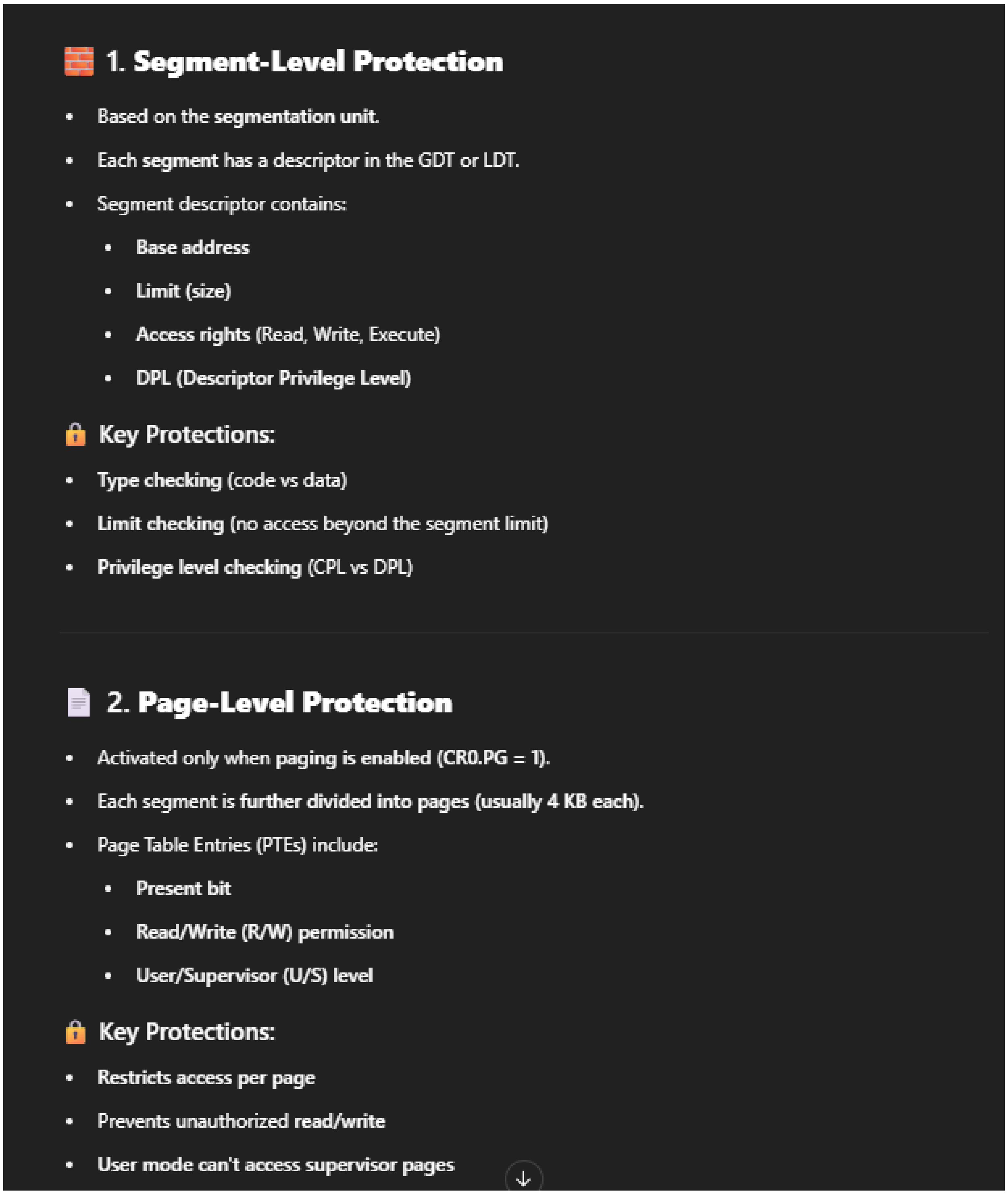
✅ Conclusion

Privileged instructions are vital for:

- **System protection**
- **Hardware control**
- **Kernel integrity**

They ensure that **user-mode programs** cannot directly interfere with **critical system functions.**

6) Elaborate the concept of combining segment Protection & Page level protection in 80386



🧱 1. **Segment-Level Protection**

- Based on the **segmentation unit**.
- Each segment has a descriptor in the GDT or LDT.
- Segment descriptor contains:
    - **Base address**
    - **Limit (size)**
    - **Access rights** (Read, Write, Execute)
    - **DPL (Descriptor Privilege Level)**

🔒 Key Protections:

- **Type checking** (code vs data)
- **Limit checking** (no access beyond the segment limit)
- **Privilege level checking** (CPL vs DPL)

📄 2. **Page-Level Protection**

- Activated only when **paging is enabled (CR0.PG = 1).**
- Each segment is **further divided into pages (usually 4 KB each).**
- Page Table Entries (PTEs) include:
    - **Present bit**
    - **Read/Write (R/W) permission**
    - **User/Supervisor (U/S) level**

🔒 Key Protections:

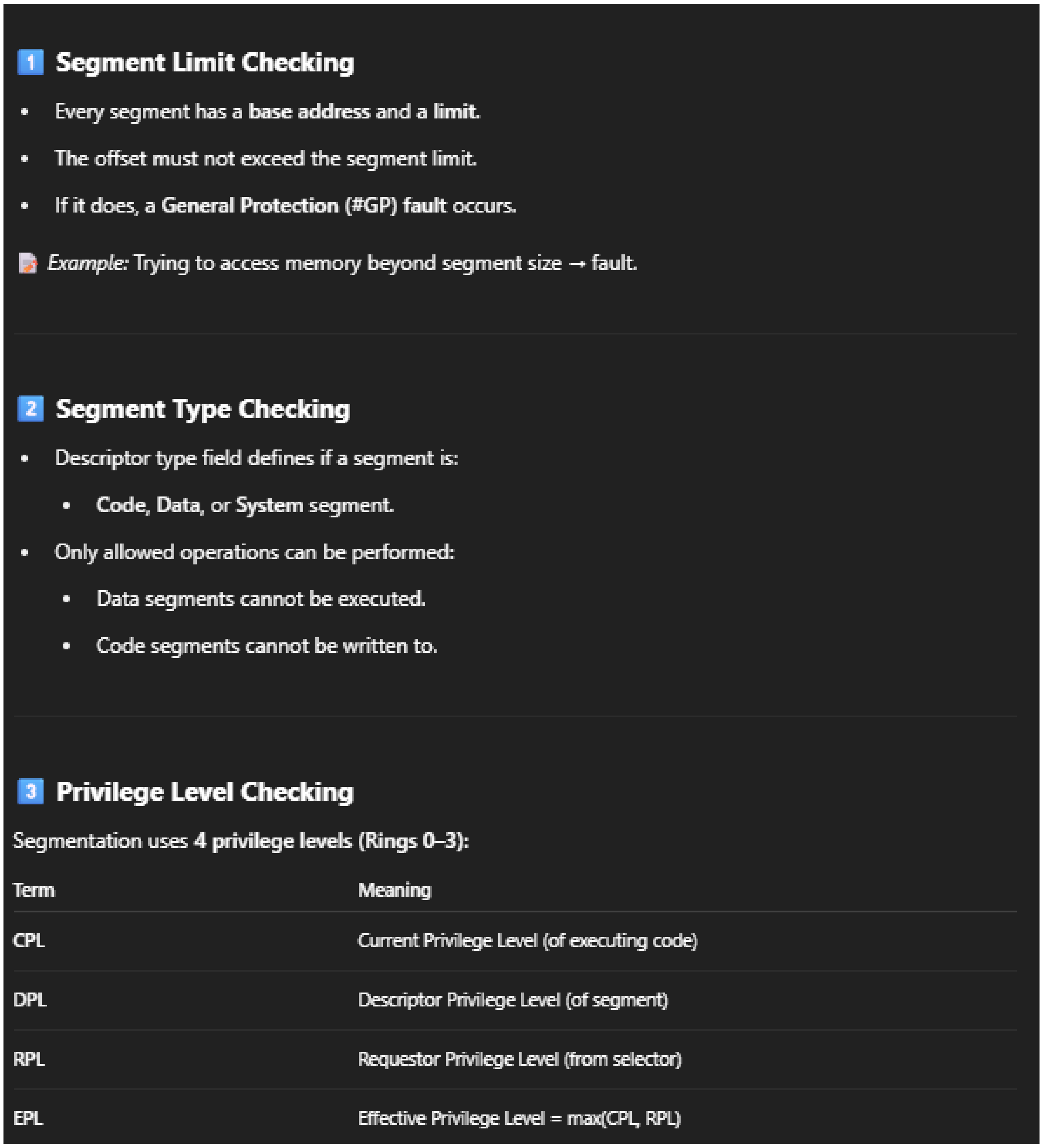
- **Restricts access per page**
- Prevents unauthorized **read/write**
- **User mode can't access supervisor pages**

- combining Page level protection & segment level protection.

1) when paging is enable, first segment level protection is evaluated.
2) If everything good, then only page level protection is evaluated.
3) If process detects a protection violation at any stage, the requested operation can't proceed & an exception occurs.

Page
4KB.

7) Explain various aspects of protection mechanism of segmentation unit.



1️⃣ Segment Limit Checking

- Every segment has a **base address** and a **limit**.
- The offset must not exceed the segment limit.
- If it does, a **General Protection (#GP) fault** occurs.

📝 *Example:* Trying to access memory beyond segment size → fault.

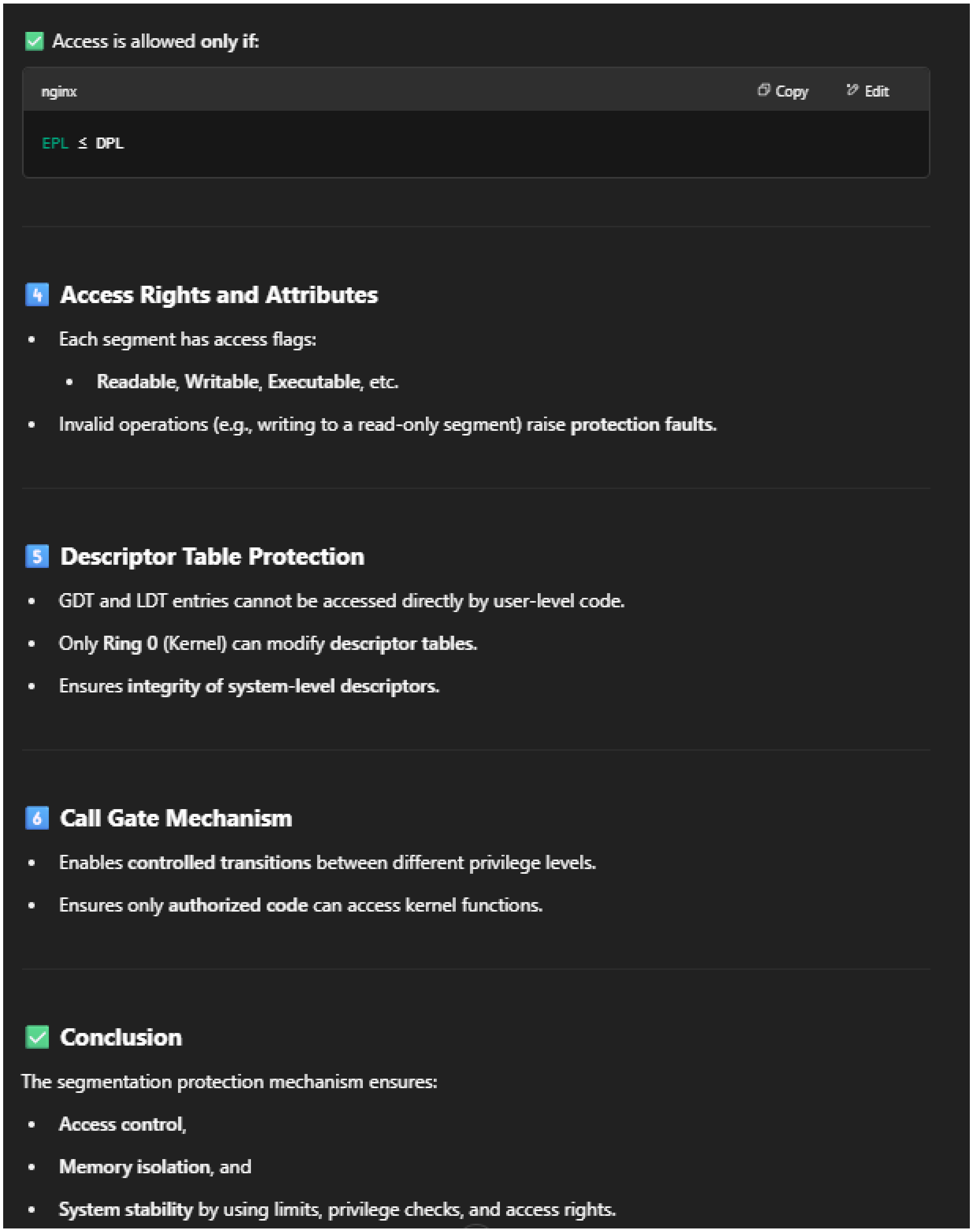
2️⃣ Segment Type Checking

- Descriptor type field defines if a segment is:
  - **Code, Data,** or **System** segment.
- Only allowed operations can be performed:
  - Data segments cannot be executed.
  - Code segments cannot be written to.

3️⃣ Privilege Level Checking

Segmentation uses **4 privilege levels (Rings 0–3):**

Term	Meaning
CPL	Current Privilege Level (of executing code)
DPL	Descriptor Privilege Level (of segment)
RPL	Requestor Privilege Level (from selector)
EPL	Effective Privilege Level = max(CPL, RPL)



✅ Access is allowed **only if:**

```nginx
EPL ≤ DPL
```

4️⃣ Access Rights and Attributes

- Each segment has access flags:
  - **Readable, Writable, Executable**, etc.
- Invalid operations (e.g., writing to a read-only segment) raise **protection faults.**

5️⃣ Descriptor Table Protection

- GDT and LDT entries cannot be accessed directly by user-level code.
- Only **Ring 0** (Kernel) can modify **descriptor tables.**
- Ensures **integrity of system-level descriptors.**

6️⃣ Call Gate Mechanism

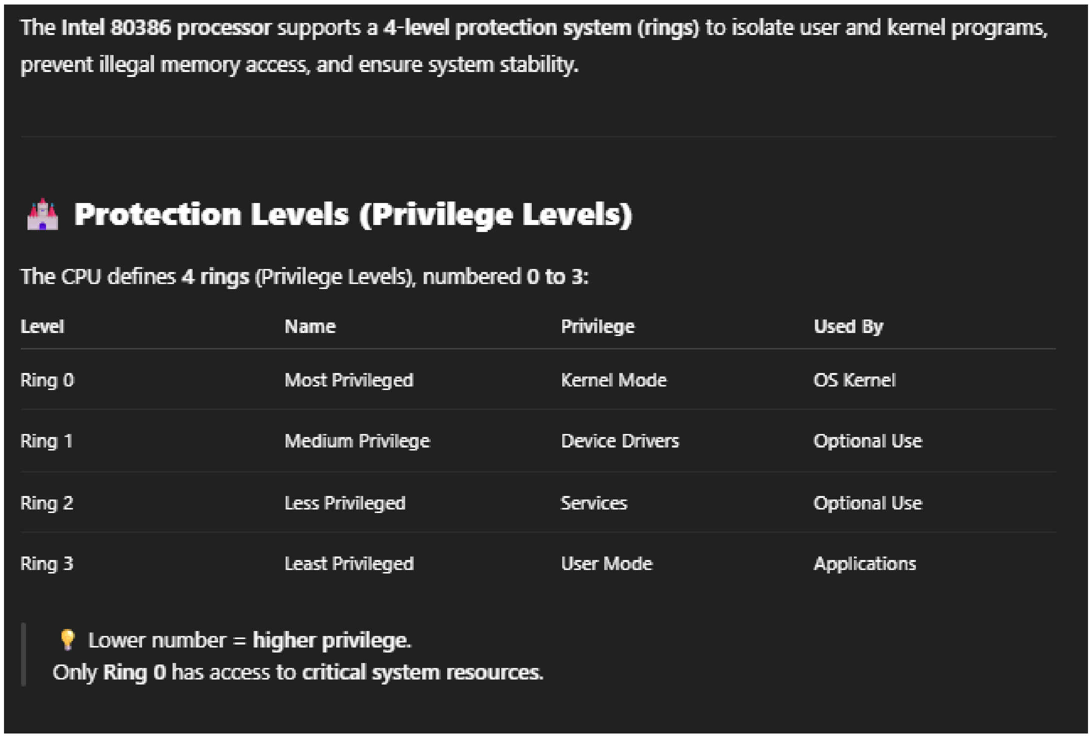
- Enables **controlled transitions** between different privilege levels.
- Ensures only **authorized code** can access kernel functions.

✅ Conclusion

The segmentation protection mechanism ensures:

- **Access control,**
- **Memory isolation,** and
- **System stability** by using limits, privilege checks, and access rights.

8) **With the help of neat diagram explain various levels of protection and rules for protection check.**


The **Intel 80386 processor** supports a **4-level protection system (rings)** to isolate user and kernel programs, prevent illegal memory access, and ensure system stability.

🏰 Protection Levels (Privilege Levels)

The CPU defines **4 rings** (Privilege Levels), numbered **0 to 3:**

Level	**Name**	**Privilege**	**Used By**
Ring 0	Most Privileged	Kernel Mode	OS Kernel
Ring 1	Medium Privilege	Device Drivers	Optional Use
Ring 2	Less Privileged	Services	Optional Use
Ring 3	Least Privileged	User Mode	Applications

> 💡 Lower number = **higher privilege.**
> Only **Ring 0** has access to **critical system resources.**

levels of protection, called **privilege levels (PL).**

2. They are designed to support the needs of multitasking OS to isolate and protect user programs from each other and the OS from unauthorized access.

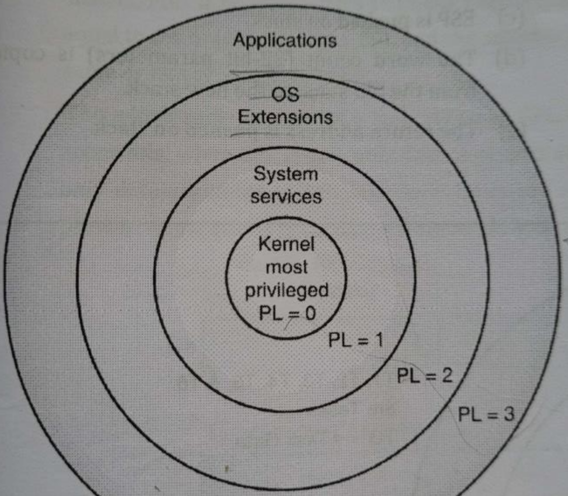


Fig. 6.3.2 : Privilege levels of the tasks in x86 architecture

University Questions

Explain CPL, RPL and DPL concepts.

SPPU - May 12, 3 Marks

What is CPL and RPL ?

SPPU - Dec. 17, May 18, 2 Marks

With appropriate diagram explain the concept of privilege level in 80386.

SPPU - May 19, 4 Marks

Define RPL and CPL ?

SPPU - Dec. 19, 2 Marks

Most processors have only two protection levels (user and supervisor), but x86 architecture features four levels of protection, called **privilege levels (PL).** They are designed to support the needs of multitasking OS to isolate and protect user programs from each other and the OS from unauthorized access.

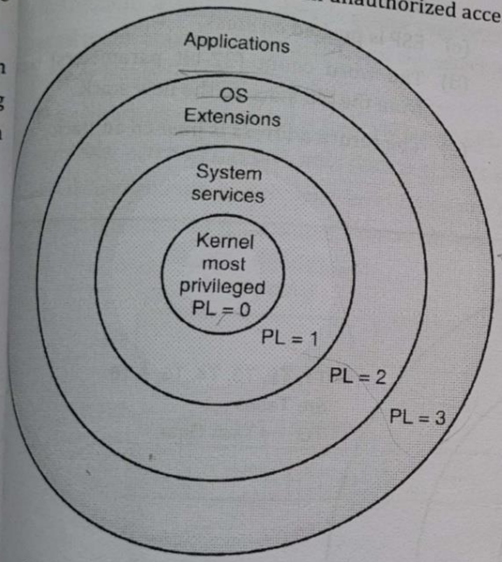


Fig. 6.3.2 : Privilege levels of the tasks in x86 architecture

The privilege levels control the use of privileged instructions, I/O instructions, and access to segments and segment descriptors. The x86 architecture offers an additional type of protection on a page basis, when paging is enabled.

Requested	PL	⇒ RPL
Descriptor	PL	⇒ DPL
Current	PL	⇒ CPL
Effective	PL	⇒ EPL

EPL = max (RPL, CPL)

4. The PLs are numbered 0, 1, 2, and 3. Level 0 is the most privileged level. Level 3 is the least privileged. As shown in Fig. 6.3.2, level 3 is used for user application, level 2 is used for OS extensions, level 1 for system services, and the most privileged level 0 is used for kernel.

5. The x86 architecture controls access to both data and code between levels of task, according to the following rules of privilege:

   (a) Data stored in a segment with PL = p can be accessed only by code executing at a PL, numerically, at least as privileged as p.

   (b) A code segment (a procedure) with PL = p can be called only by a task executing at, numerically, the same or lower PL than p.

   (c) A stack segment with PL = p can be used only by a task executing at the same PL.

6. The following PLs are used to maintain privilege level check :

   (a) Requestor PL, RPL, the PL of the original task that supplies the selector. RPL is determined by the two LSBs of the selector.

   (b) Descriptor PL, DPL, the PL (according to the above rules) at which a task may access that descriptor and the segment associated with that descriptor. Bits 6 and 5 of the access rights byte (ARB) of a descriptor determine the DPL.

   (c) Current PL, CPL, the PL at which a task is currently executing, i.e. at which the code segment is being executed. CPL is stored in the processor, but not accessible to the programmer.

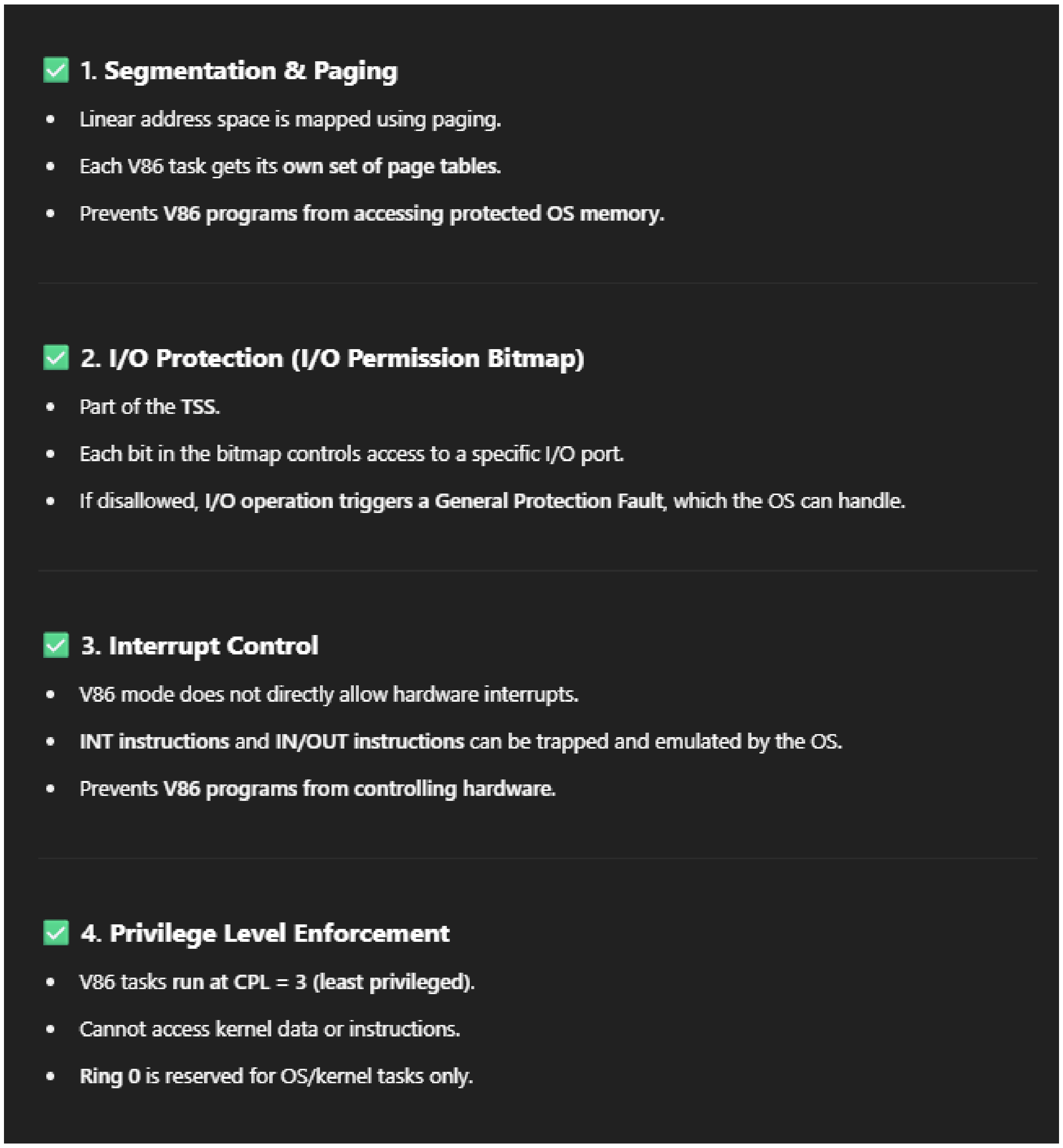
   (d) Effective PL, EPL, the least privileged of the RPL and the CPL. Since smaller PL values indicate greater privilege, EPL is the numerical maximum of RPL and CPL, i.e. EPL = max (RPL, CPL). EPL is not stored anywhere, but is immediately copied into CPL.

7. System segments describe information about tasks, interrupts, subroutines etc.

   The different types (the lower four bits of ARB for a system descriptor) of system descriptor are as listed below:

UNIT 5

1) **Explain the structure of a V86 Task in detail. How is protection provided within the V86 task?**



✅ 1. Segmentation & Paging

- Linear address space is mapped using paging.
- Each V86 task gets its **own set of page tables.**
- Prevents **V86 programs from accessing protected OS memory.**

✅ 2. I/O Protection (I/O Permission Bitmap)

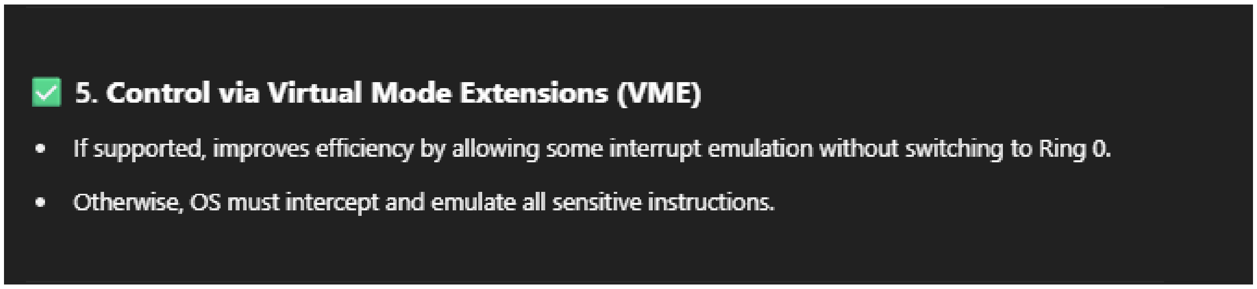
- Part of the **TSS.**
- Each bit in the bitmap controls access to a specific I/O port.
- If disallowed, **I/O operation triggers a General Protection Fault**, which the OS can handle.

✅ 3. Interrupt Control

- V86 mode does not directly allow hardware interrupts.
- **INT instructions** and **IN/OUT instructions** can be trapped and emulated by the OS.
- Prevents **V86 programs from controlling hardware.**

✅ 4. Privilege Level Enforcement

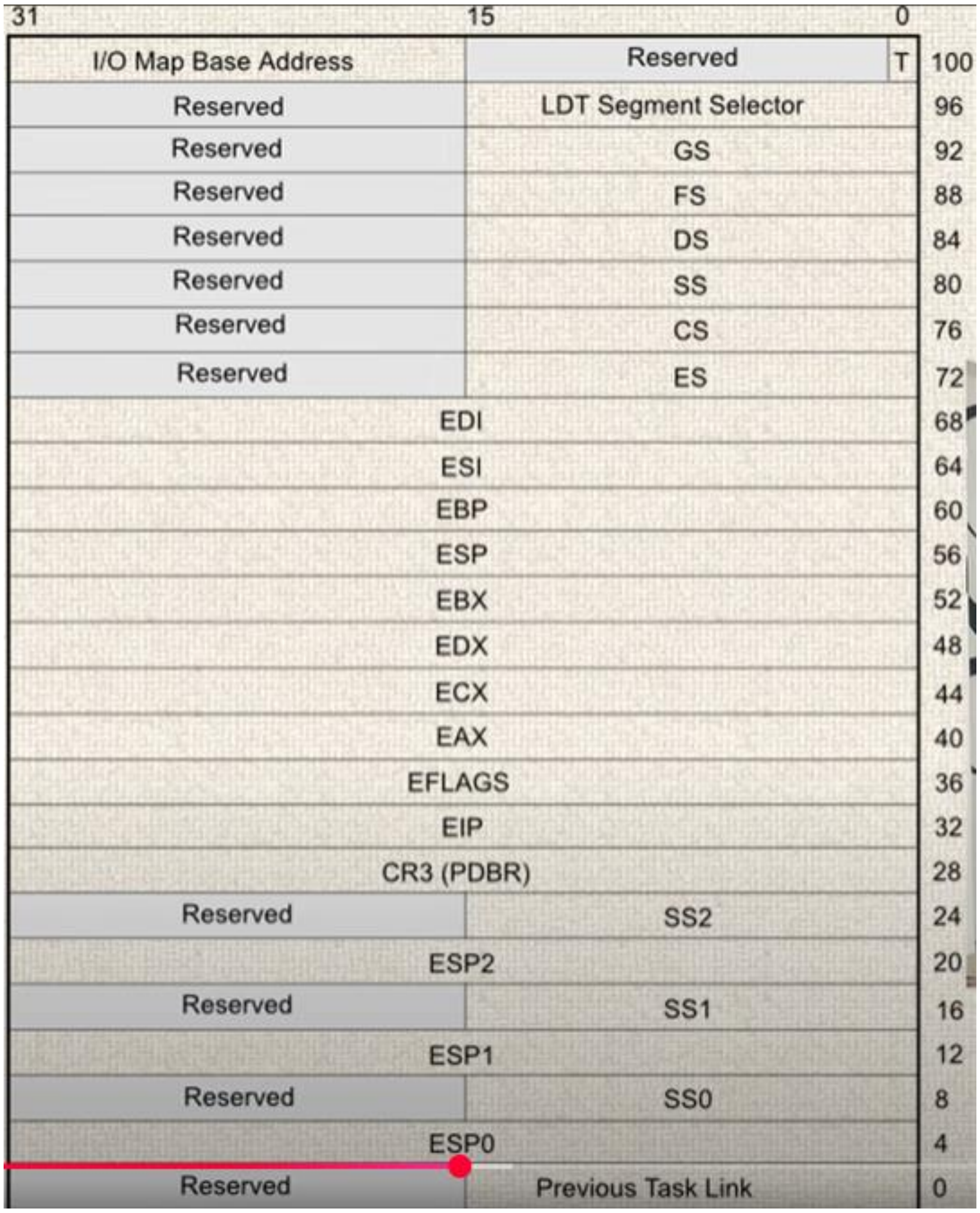
- V86 tasks **run at CPL = 3 (least privileged).**
- Cannot access kernel data or instructions.
- **Ring 0** is reserved for OS/kernel tasks only.



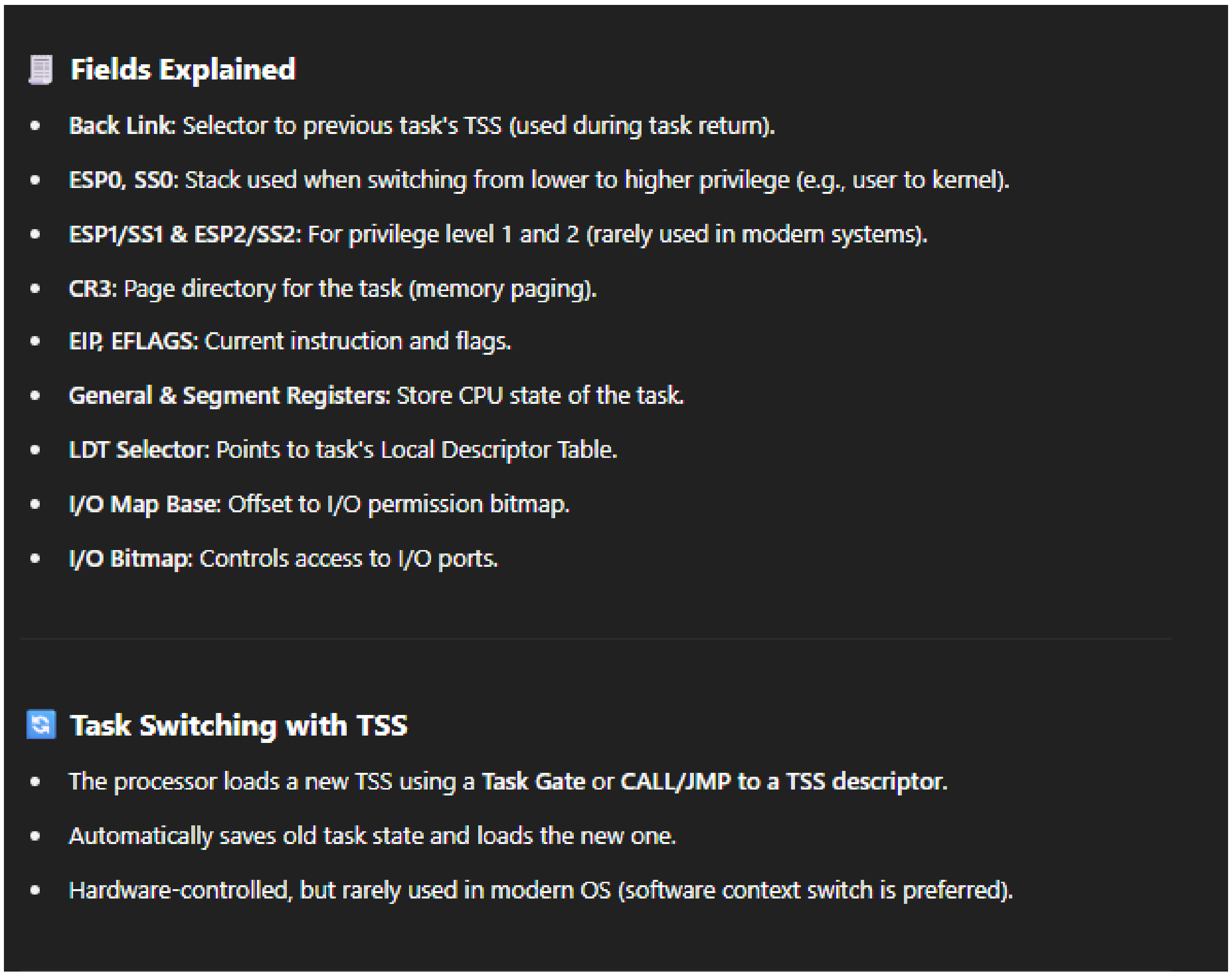
✅ 5. Control via Virtual Mode Extensions (VME)

- If supported, improves efficiency by allowing some interrupt emulation without switching to Ring 0.
- Otherwise, OS must intercept and emulate all sensitive instructions.

2) Draw & Explain the Task state segment of 80386.



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



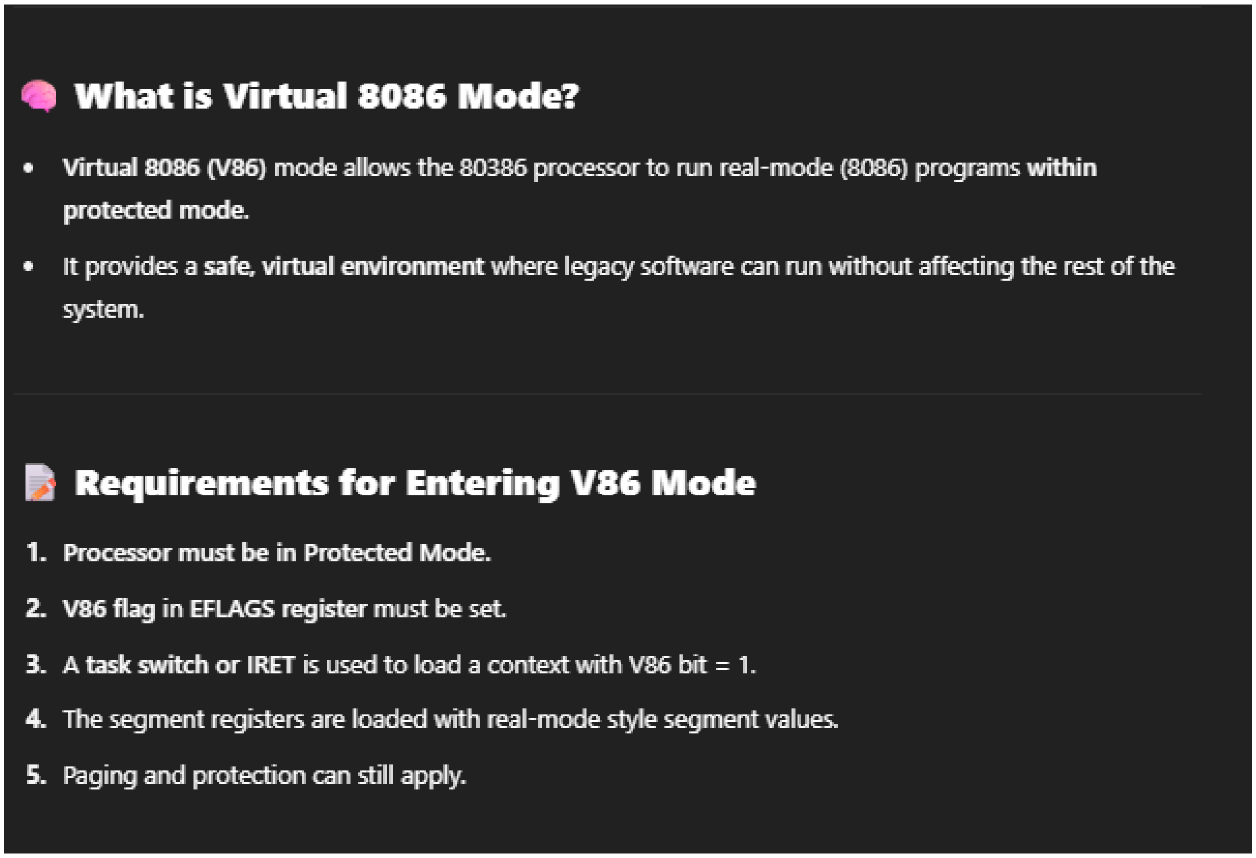
🧾 Fields Explained

- **Back Link:** Selector to previous task's TSS (used during task return).
- **ESP0, SS0**: Stack used when switching from lower to higher privilege (e.g., user to kernel).
- **ESP1/SS1 & ESP2/SS2**: For privilege level 1 and 2 (rarely used in modern systems).
- **CR3**: Page directory for the task (memory paging).
- **EIP, EFLAGS**: Current instruction and flags.
- **General & Segment Registers:** Store CPU state of the task.
- **LDT Selector**: Points to task's Local Descriptor Table.
- **I/O Map Base**: Offset to I/O permission bitmap.
- **I/O Bitmap**: Controls access to I/O ports.

🔄 Task Switching with TSS

- The processor loads a new TSS using a **Task Gate** or **CALL/JMP to a TSS descriptor**.
- Automatically saves old task state and loads the new one.
- Hardware-controlled, but rarely used in modern OS (software context switch is preferred).

3) With the necessary diagram, Explain entering & leaving the virtual mode of 80386.

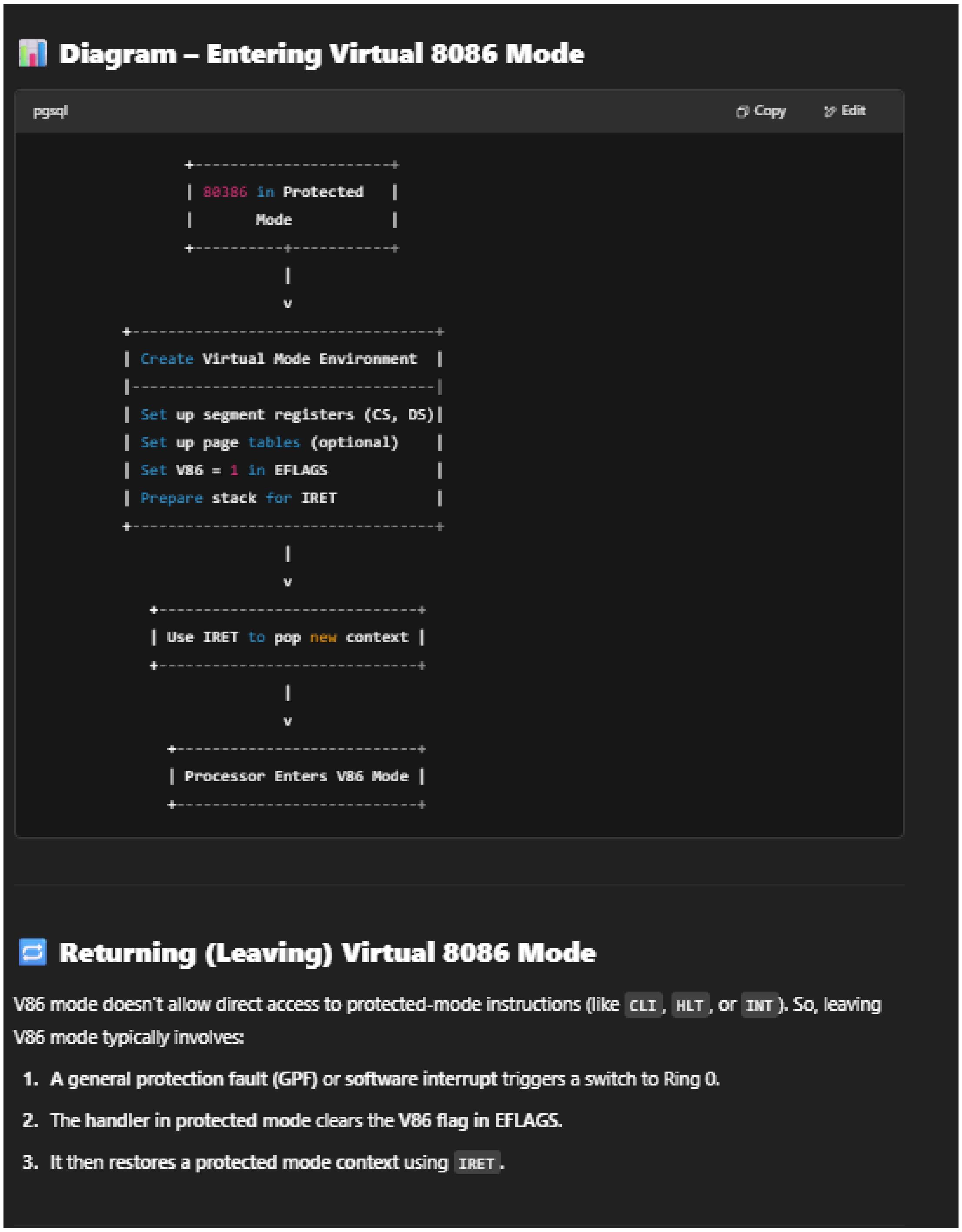


What is Virtual 8086 Mode?

- **Virtual 8086 (V86)** mode allows the 80386 processor to run real-mode (8086) programs **within protected mode.**
- It provides a **safe, virtual environment** where legacy software can run without affecting the rest of the system.

Requirements for Entering V86 Mode

1. Processor must be in Protected Mode.
2. **V86 flag** in **EFLAGS register** must be set.
3. A **task switch or IRET** is used to load a context with V86 bit = 1.
4. The segment registers are loaded with real-mode style segment values.
5. Paging and protection can still apply.



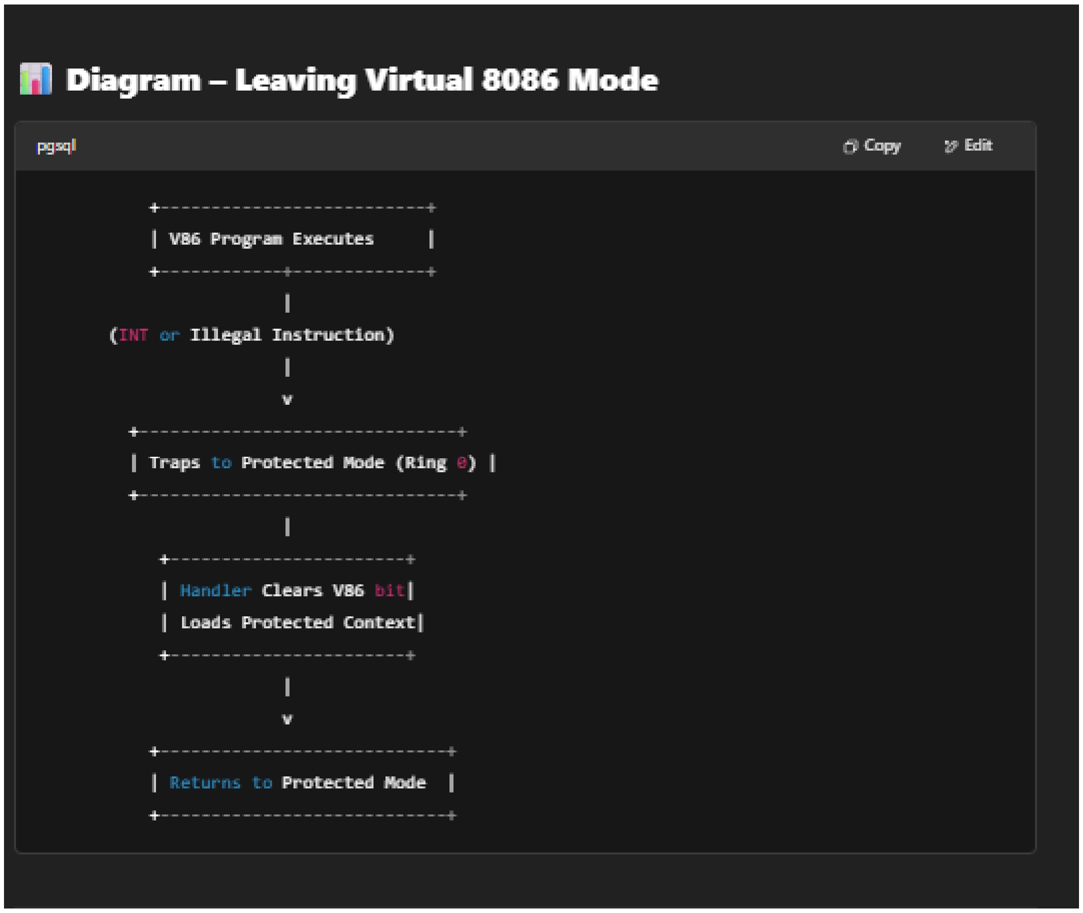
📊 Diagram – Entering Virtual 8086 Mode

```
           +-----------------------+
           | 80386 in Protected    |
           |      Mode             |
           +-----------+-----------+
                       |
                       v
    +-----------------------------------+
    | Create Virtual Mode Environment   |
    |-----------------------------------|
    | Set up segment registers (CS, DS)|
    | Set up page tables (optional)     |
    | Set V86 = 1 in EFLAGS             |
    | Prepare stack for IRET            |
    +-----------------------------------+
                       |
                       v
       +-----------------------------+
       | Use IRET to pop new context |
       +-----------------------------+
                       |
                       v
         +---------------------------+
         | Processor Enters V86 Mode |
         +---------------------------+
```

🔁 Returning (Leaving) Virtual 8086 Mode

V86 mode doesn't allow direct access to protected-mode instructions (like `CLI`, `HLT`, or `INT`). So, leaving V86 mode typically involves:

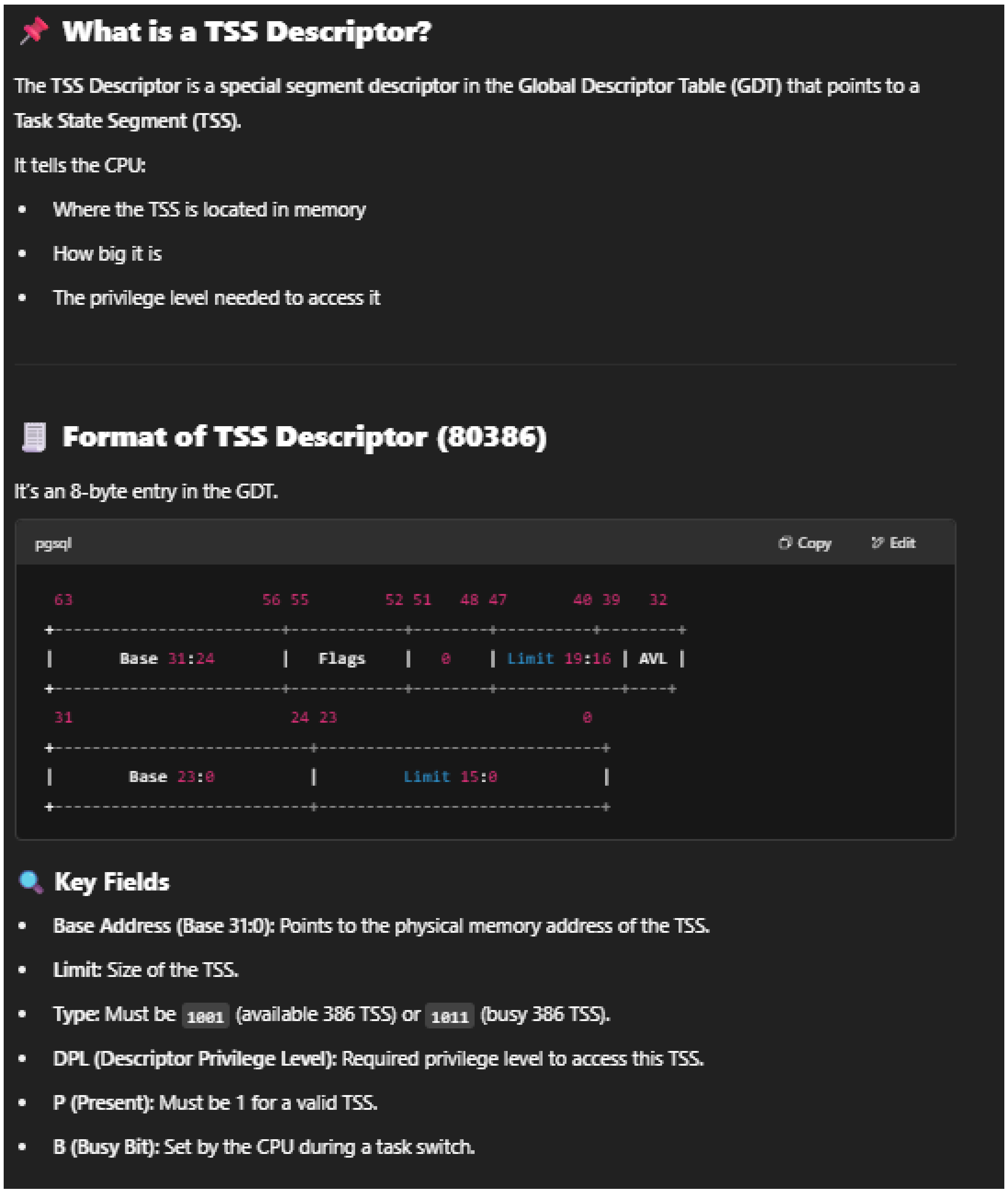
1. A general protection fault (GPF) or software interrupt triggers a switch to Ring 0.
2. The handler in protected mode clears the V86 flag in EFLAGS.
3. It then restores a protected mode context using `IRET`.



📊 Diagram – Leaving Virtual 8086 Mode

```
           +--------------------------+
           | V86 Program Executes     |
           +------------+-------------+
                        |
       (INT or Illegal Instruction)
                        |
                        v
         +-------------------------------+
         | Traps to Protected Mode (Ring 0) |
         +-------------------------------+
                        |
            +----------------------+
            | Handler Clears V86 bit|
            | Loads Protected Context|
            +----------------------+
                        |
                        v
           +---------------------------+
           | Returns to Protected Mode  |
           +---------------------------+
```

4) Explain the TSS descriptor & its role in multitasking.



📌 What is a TSS Descriptor?

The TSS Descriptor is a special segment descriptor in the Global Descriptor Table (GDT) that points to a Task State Segment (TSS).

It tells the CPU:

- Where the TSS is located in memory
- How big it is
- The privilege level needed to access it

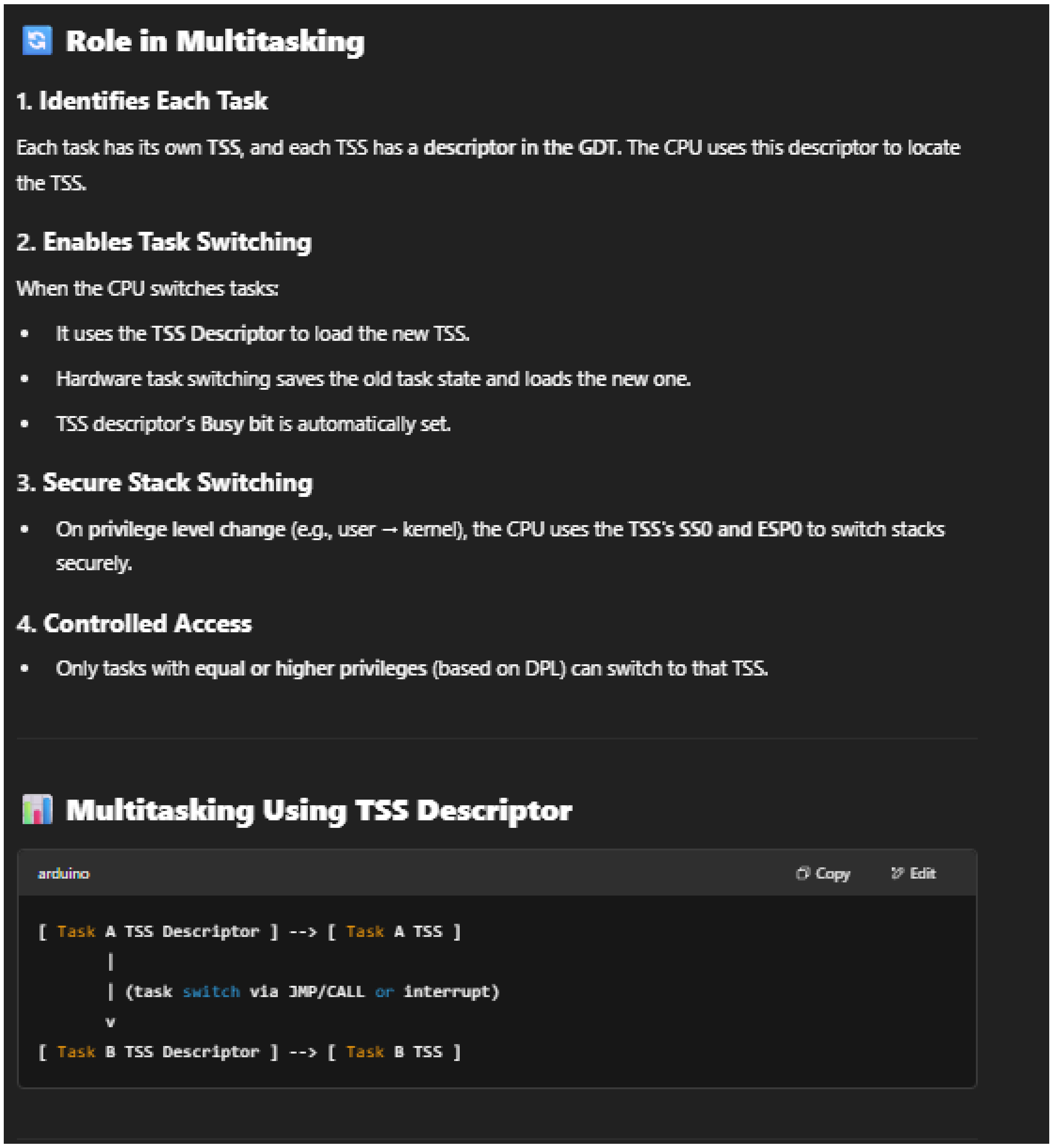
🧾 Format of TSS Descriptor (80386)

It's an 8-byte entry in the GDT.

```pgsql
 63                    56 55        52 51   48 47         40 39   32
+------------------------+------------+--------+-----------+--------+
|       Base 31:24       |   Flags    |   0    | Limit 19:16 | AVL |
+------------------------+------------+--------+-------------+----+
 31                         24 23                            0
+---------------------------+-----------------------------+
|        Base 23:0          |         Limit 15:0          |
+---------------------------+-----------------------------+
```

🔍 Key Fields

- Base Address (Base 31:0): Points to the physical memory address of the TSS.
- Limit: Size of the TSS.
- Type: Must be `1001` (available 386 TSS) or `1011` (busy 386 TSS).
- DPL (Descriptor Privilege Level): Required privilege level to access this TSS.
- P (Present): Must be 1 for a valid TSS.
- B (Busy Bit): Set by the CPU during a task switch.



🔄 Role in Multitasking

1. Identifies Each Task

Each task has its own TSS, and each TSS has a **descriptor in the GDT**. The CPU uses this descriptor to locate the TSS.

2. Enables Task Switching

When the CPU switches tasks:

- It uses the **TSS Descriptor** to load the new TSS.
- Hardware task switching saves the old task state and loads the new one.
- TSS descriptor's **Busy bit** is automatically set.

3. Secure Stack Switching

- On **privilege level change** (e.g., user → kernel), the CPU uses the **TSS's SS0 and ESP0** to switch stacks securely.

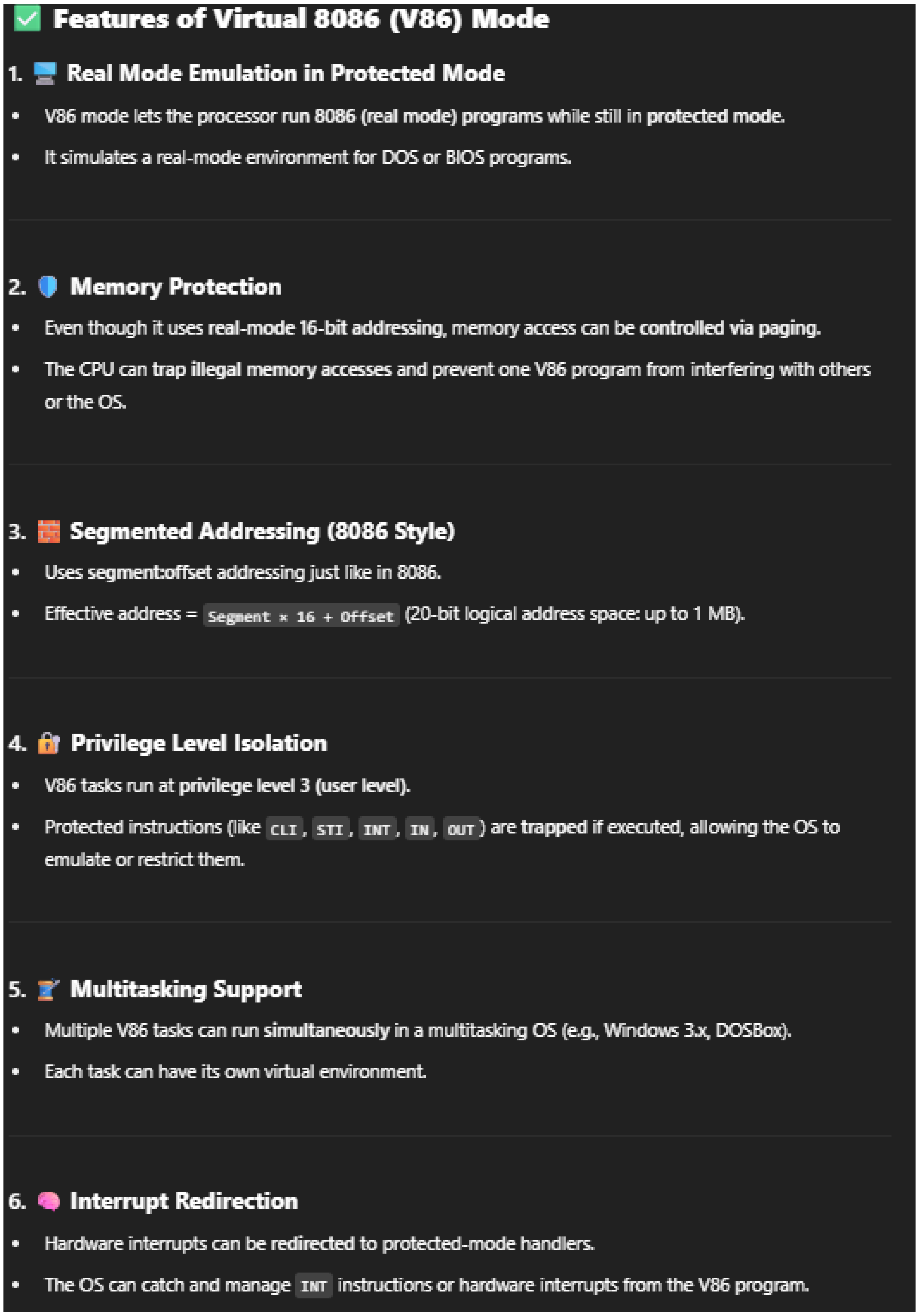
4. Controlled Access

- Only tasks with **equal or higher privileges** (based on DPL) can switch to that TSS.

📊 Multitasking Using TSS Descriptor

```arduino
[ Task A TSS Descriptor ] --> [ Task A TSS ]
       |
       | (task switch via JMP/CALL or interrupt)
       v
[ Task B TSS Descriptor ] --> [ Task B TSS ]
```

5) List & Explain various features of virtual 8086 Mode.



✅ Features of Virtual 8086 (V86) Mode

1. 🖥️ Real Mode Emulation in Protected Mode

- V86 mode lets the processor run 8086 (real mode) programs while still in protected mode.
- It simulates a real-mode environment for DOS or BIOS programs.

2. 🛡️ Memory Protection

- Even though it uses real-mode 16-bit addressing, memory access can be controlled via paging.
- The CPU can trap illegal memory accesses and prevent one V86 program from interfering with others or the OS.

3. 🧱 Segmented Addressing (8086 Style)

- Uses segment:offset addressing just like in 8086.
- Effective address = `Segment × 16 + Offset` (20-bit logical address space: up to 1 MB).

4. 🔐 Privilege Level Isolation

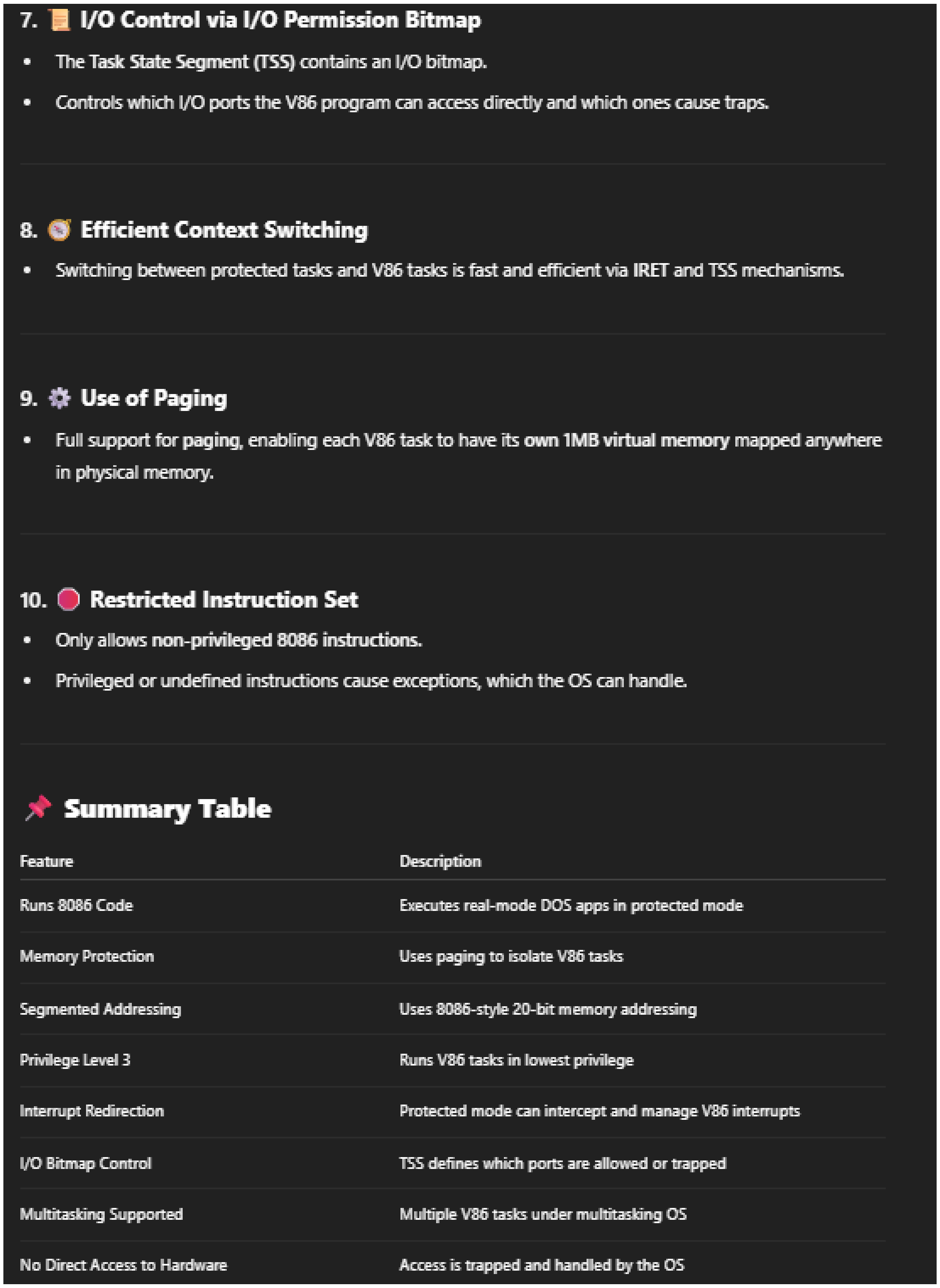
- V86 tasks run at privilege level 3 (user level).
- Protected instructions (like `CLI`, `STI`, `INT`, `IN`, `OUT`) are trapped if executed, allowing the OS to emulate or restrict them.

5. 🧵 Multitasking Support

- Multiple V86 tasks can run simultaneously in a multitasking OS (e.g., Windows 3.x, DOSBox).
- Each task can have its own virtual environment.

6. 🧠 Interrupt Redirection

- Hardware interrupts can be redirected to protected-mode handlers.
- The OS can catch and manage `INT` instructions or hardware interrupts from the V86 program.



7. 📜 I/O Control via I/O Permission Bitmap

- The Task State Segment (TSS) contains an I/O bitmap.
- Controls which I/O ports the V86 program can access directly and which ones cause traps.

---

8. 🧭 Efficient Context Switching

- Switching between protected tasks and V86 tasks is fast and efficient via IRET and TSS mechanisms.

---

9. ⚙️ Use of Paging

- Full support for **paging**, enabling each V86 task to have its **own 1MB virtual memory** mapped anywhere in physical memory.

---

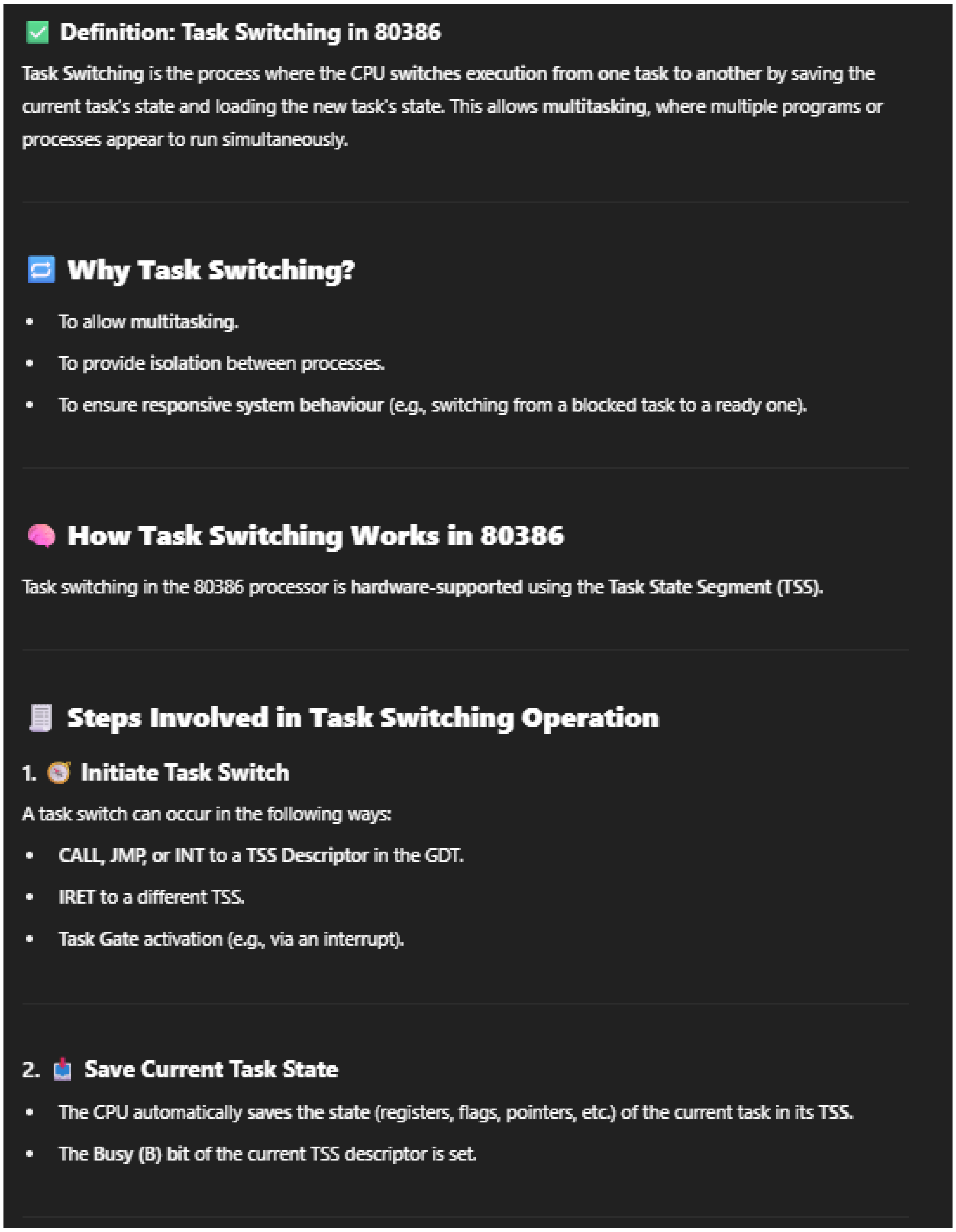
10. 🛑 Restricted Instruction Set

- Only allows **non-privileged 8086 instructions.**
- Privileged or undefined instructions cause exceptions, which the OS can handle.

📌 Summary Table

Feature	Description
Runs 8086 Code	Executes real-mode DOS apps in protected mode
Memory Protection	Uses paging to isolate V86 tasks
Segmented Addressing	Uses 8086-style 20-bit memory addressing
Privilege Level 3	Runs V86 tasks in lowest privilege
Interrupt Redirection	Protected mode can intercept and manage V86 interrupts
I/O Bitmap Control	TSS defines which ports are allowed or trapped
Multitasking Supported	Multiple V86 tasks under multitasking OS
No Direct Access to Hardware	Access is trapped and handled by the OS

6) **Define Task switching & Explain the steps involved in task switching operation.**



✅ Definition: Task Switching in 80386

Task Switching is the process where the CPU **switches execution from one task to another** by saving the current task's state and loading the new task's state. This allows **multitasking**, where multiple programs or processes appear to run simultaneously.

🔁 Why Task Switching?

- To allow **multitasking**.
- To provide **isolation** between processes.
- To ensure **responsive system behaviour** (e.g., switching from a blocked task to a ready one).

🧠 How Task Switching Works in 80386

Task switching in the 80386 processor is **hardware-supported** using the **Task State Segment (TSS)**.

📃 Steps Involved in Task Switching Operation

1. 🧭 Initiate Task Switch

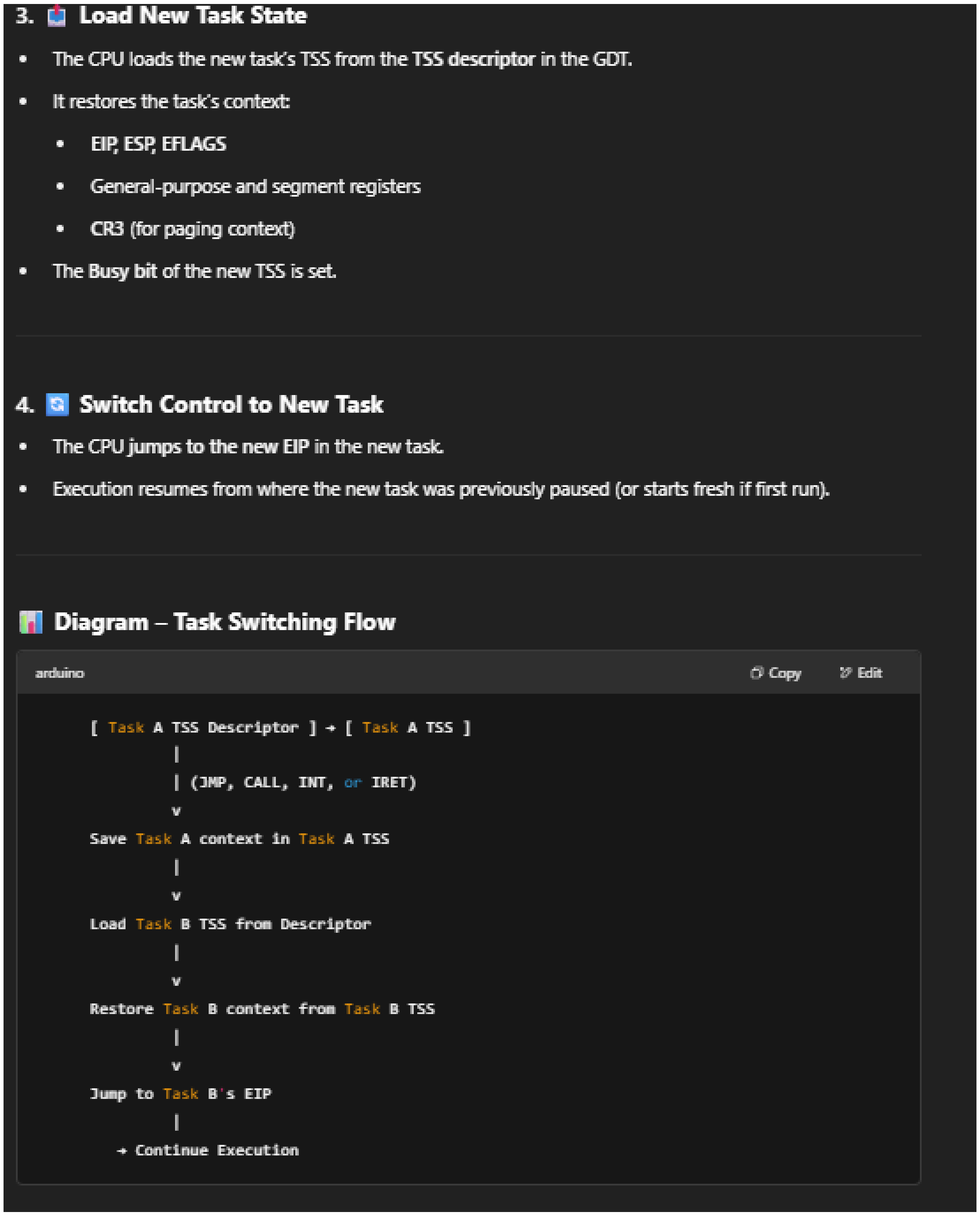
A task switch can occur in the following ways:

- **CALL, JMP, or INT** to a **TSS Descriptor** in the GDT.
- **IRET** to a different TSS.
- **Task Gate** activation (e.g., via an interrupt).

---

2. 📥 Save Current Task State

- The CPU automatically saves the state (registers, flags, pointers, etc.) of the current task in its **TSS**.
- The **Busy (B) bit** of the current TSS descriptor is set.



3. 📥 Load New Task State

- The CPU loads the new task's TSS from the **TSS descriptor** in the GDT.
- It restores the task's context:
  - EIP, ESP, EFLAGS
  - General-purpose and segment registers
  - CR3 (for paging context)
- The **Busy bit** of the new TSS is set.

---

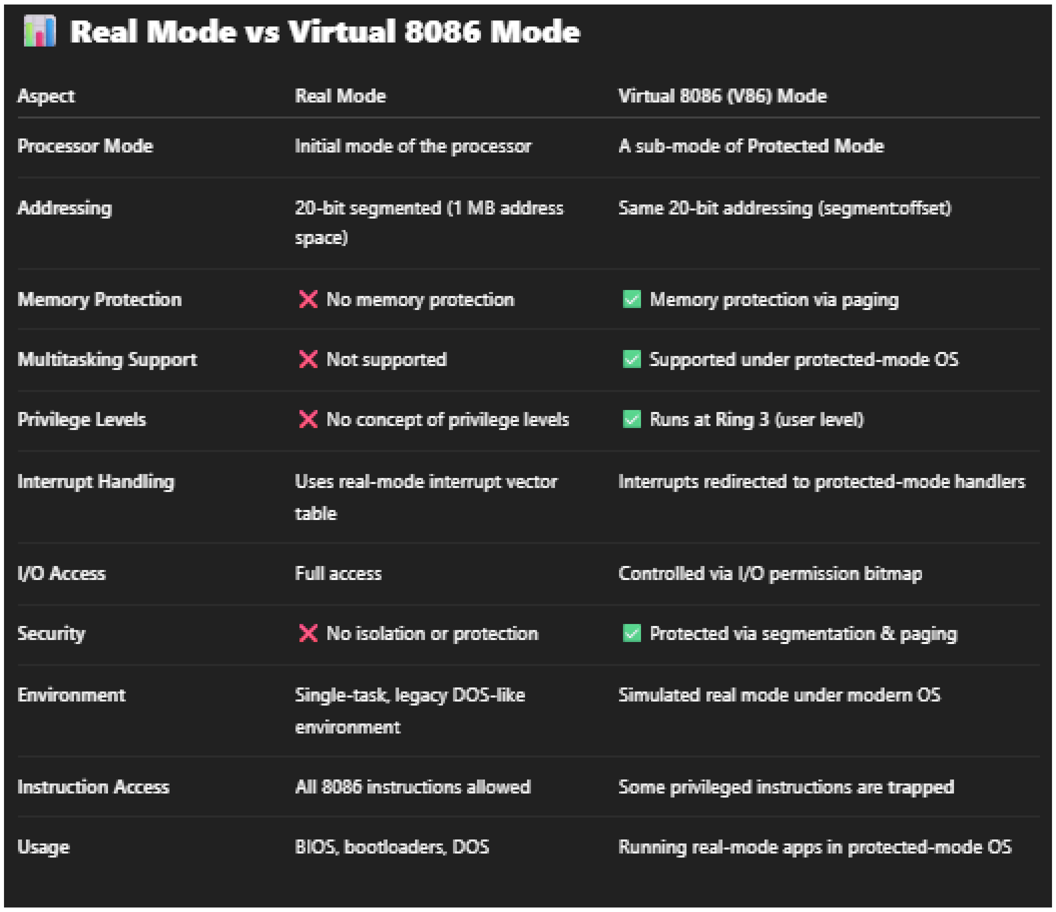
4. 🔄 Switch Control to New Task

- The CPU **jumps to the new EIP** in the new task.
- Execution resumes from where the new task was previously paused (or starts fresh if first run).

📊 Diagram – Task Switching Flow

```arduino
[ Task A TSS Descriptor ] → [ Task A TSS ]
         |
         | (JMP, CALL, INT, or IRET)
         v
Save Task A context in Task A TSS
         |
         v
Load Task B TSS from Descriptor
         |
         v
Restore Task B context from Task B TSS
         |
         v
Jump to Task B's EIP
         |
   → Continue Execution
```

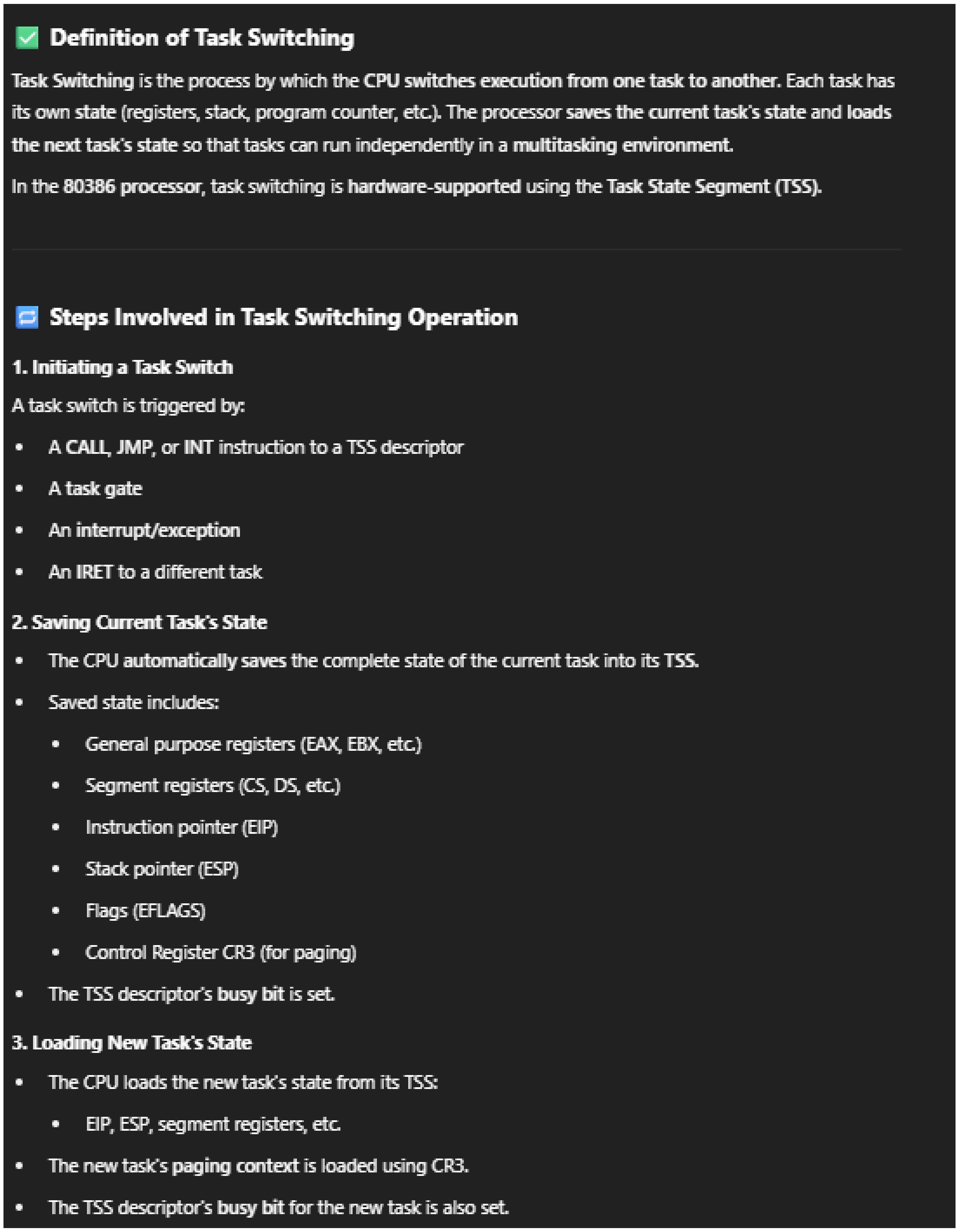
7) **Differentiate between real mode and virtual mode.**



📊 Real Mode vs Virtual 8086 Mode

Aspect	Real Mode	Virtual 8086 (V86) Mode
Processor Mode	Initial mode of the processor	A sub-mode of Protected Mode
Addressing	20-bit segmented (1 MB address space)	Same 20-bit addressing (segment:offset)
Memory Protection	❌ No memory protection	✅ Memory protection via paging
Multitasking Support	❌ Not supported	✅ Supported under protected-mode OS
Privilege Levels	❌ No concept of privilege levels	✅ Runs at Ring 3 (user level)
Interrupt Handling	Uses real-mode interrupt vector table	Interrupts redirected to protected-mode handlers
I/O Access	Full access	Controlled via I/O permission bitmap
Security	❌ No isolation or protection	✅ Protected via segmentation & paging
Environment	Single-task, legacy DOS-like environment	Simulated real mode under modern OS
Instruction Access	All 8086 instructions allowed	Some privileged instructions are trapped
Usage	BIOS, bootloaders, DOS	Running real-mode apps in protected-mode OS

8) Define task switching and explain the steps involved in task switching operation.



✅ Definition of Task Switching

Task Switching is the process by which the **CPU switches execution from one task to another.** Each task has its own state (registers, stack, program counter, etc.). The processor **saves the current task's state and loads the next task's state** so that tasks can run independently in a **multitasking environment.**

In the **80386 processor**, task switching is **hardware-supported** using the **Task State Segment (TSS).**

🔁 Steps Involved in Task Switching Operation

1. Initiating a Task Switch

A task switch is triggered by:

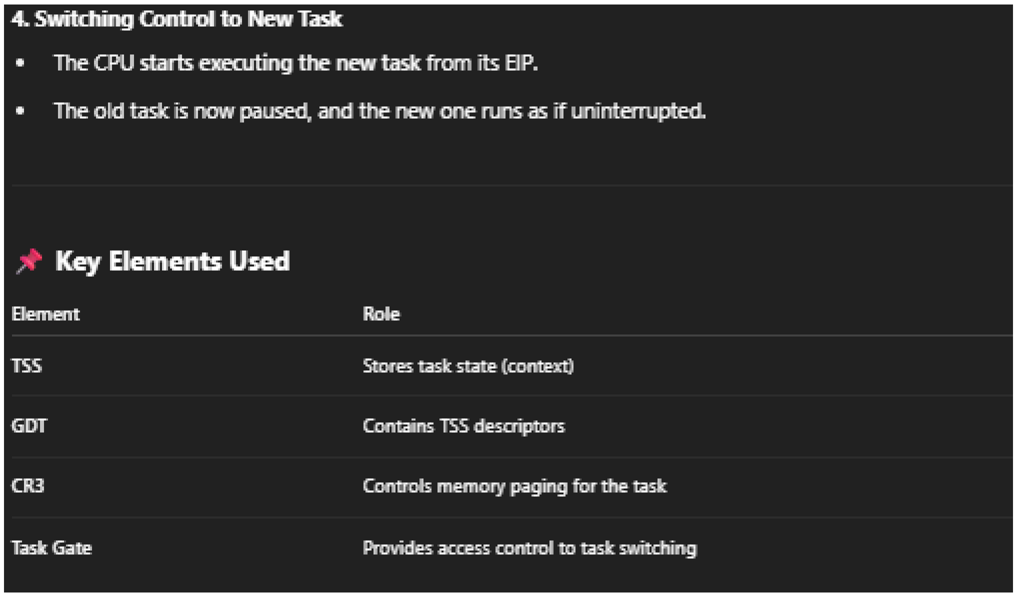
- A **CALL, JMP,** or **INT** instruction to a TSS descriptor
- A **task gate**
- An **interrupt/exception**
- An **IRET** to a different task

2. Saving Current Task's State

- The CPU **automatically saves** the complete state of the current task into its TSS.
- Saved state includes:
  - General purpose registers (EAX, EBX, etc.)
  - Segment registers (CS, DS, etc.)
  - Instruction pointer (EIP)
  - Stack pointer (ESP)
  - Flags (EFLAGS)
  - Control Register CR3 (for paging)
- The TSS descriptor's **busy bit** is set.

3. Loading New Task's State

- The CPU loads the new task's state from its TSS:
  - EIP, ESP, segment registers, etc.
- The new task's **paging context** is loaded using CR3.
- The TSS descriptor's **busy bit** for the new task is also set.



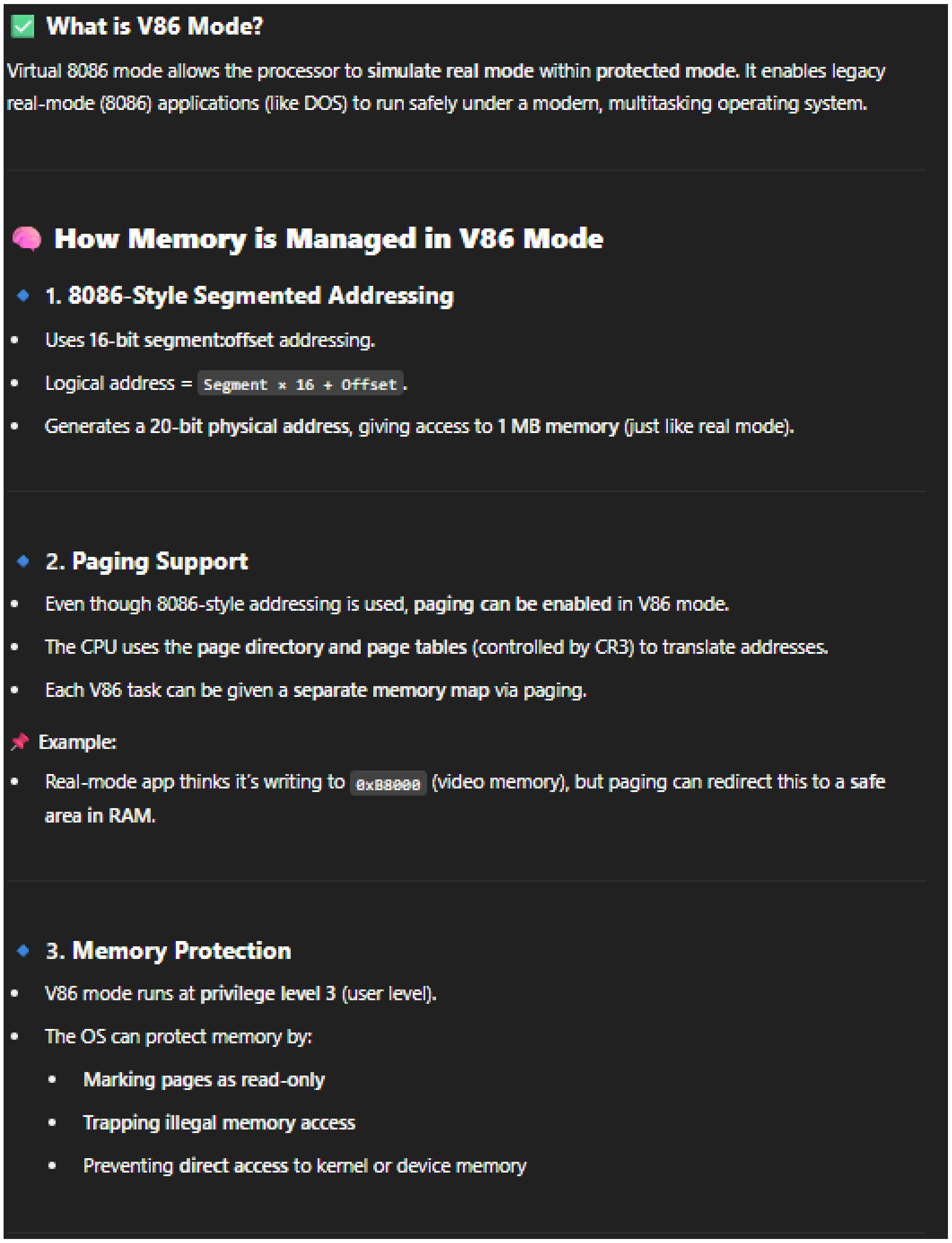
4. Switching Control to New Task

- The CPU starts executing the new task from its EIP.
- The old task is now paused, and the new one runs as if uninterrupted.

📌 Key Elements Used

Element	Role
TSS	Stores task state (context)
GDT	Contains TSS descriptors
CR3	Controls memory paging for the task
Task Gate	Provides access control to task switching

9) **Explore memory management in the Virtual 8086 Mode**



✅ What is V86 Mode?

Virtual 8086 mode allows the processor to **simulate real mode** within **protected mode**. It enables legacy real-mode (8086) applications (like DOS) to run safely under a modern, multitasking operating system.

🧠 How Memory is Managed in V86 Mode

🔹 1. 8086-Style Segmented Addressing

- Uses **16-bit segment:offset** addressing.
- Logical address = `Segment × 16 + Offset`.
- Generates a **20-bit physical address**, giving access to **1 MB memory** (just like real mode).

🔹 2. Paging Support

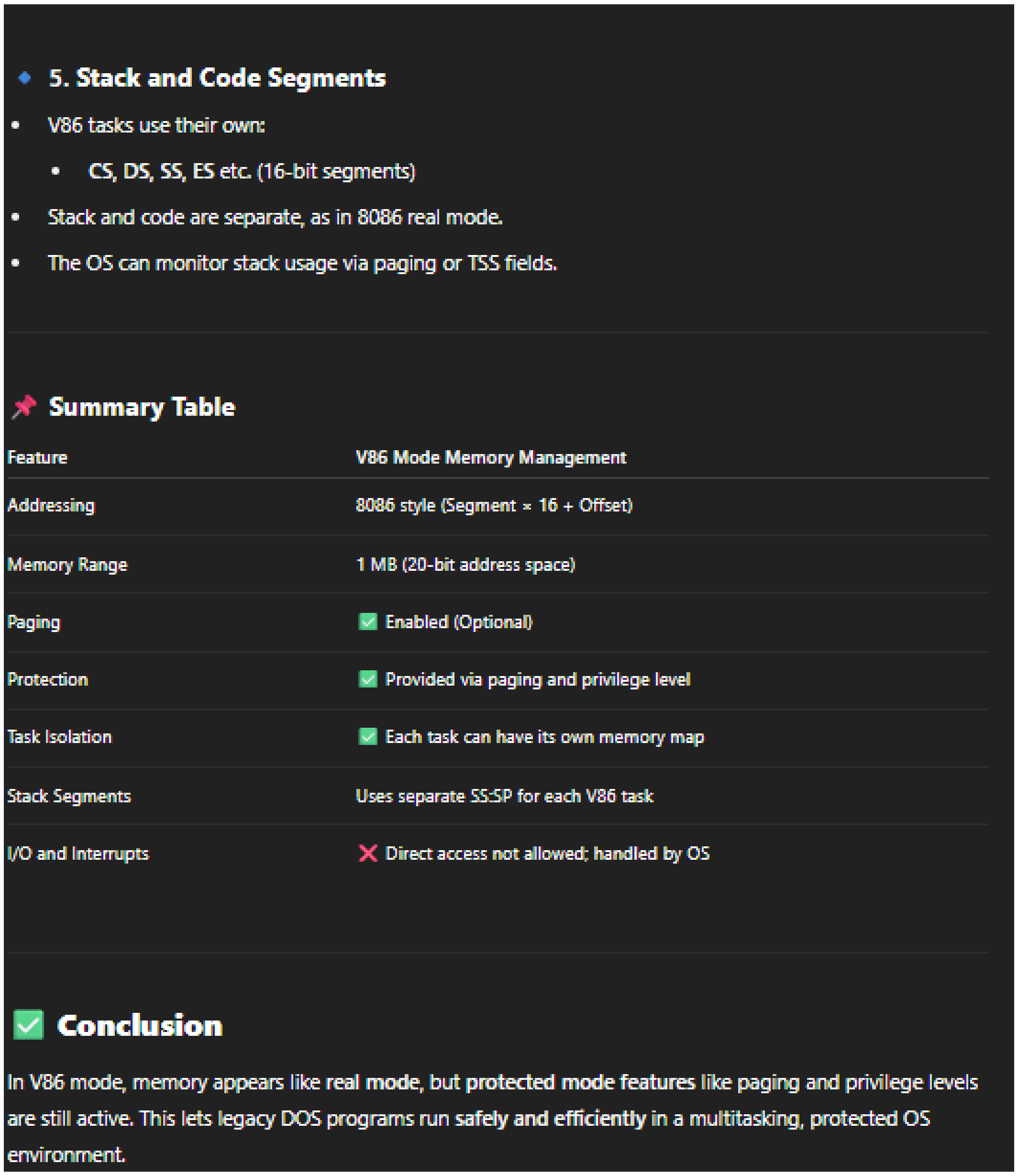
- Even though 8086-style addressing is used, **paging can be enabled** in V86 mode.
- The CPU uses the **page directory and page tables** (controlled by CR3) to translate addresses.
- Each V86 task can be given a **separate memory map** via paging.

📌 Example:

- Real-mode app thinks it's writing to `0xB8000` (video memory), but paging can redirect this to a safe area in RAM.

🔹 3. Memory Protection

- V86 mode runs at **privilege level 3** (user level).
- The OS can protect memory by:
  - **Marking pages as read-only**
  - **Trapping illegal memory access**
  - Preventing **direct access** to kernel or device memory



🔹 5. Stack and Code Segments

- V86 tasks use their own:
  - CS, DS, SS, ES etc. (16-bit segments)
- Stack and code are separate, as in 8086 real mode.
- The OS can monitor stack usage via paging or TSS fields.

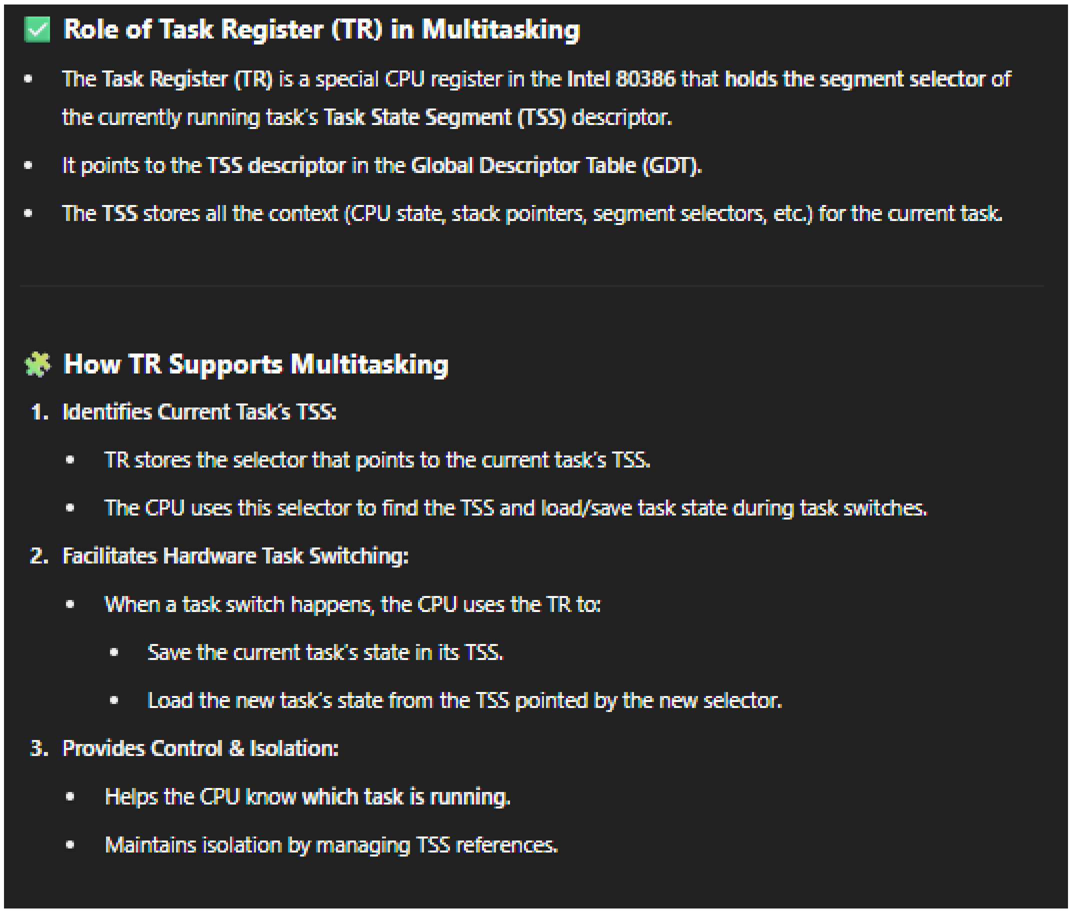
📌 Summary Table

Feature	V86 Mode Memory Management
Addressing	8086 style (Segment × 16 + Offset)
Memory Range	1 MB (20-bit address space)
Paging	✅ Enabled (Optional)
Protection	✅ Provided via paging and privilege level
Task Isolation	✅ Each task can have its own memory map
Stack Segments	Uses separate SS:SP for each V86 task
I/O and Interrupts	❌ Direct access not allowed; handled by OS

✅ Conclusion

In V86 mode, memory appears like **real mode**, but **protected mode features** like paging and privilege levels are still active. This lets legacy DOS programs run **safely and efficiently** in a multitasking, protected OS environment.

10) **Explore the role of Task Register in multitasking and the instructions used to modify and read Task Register.**

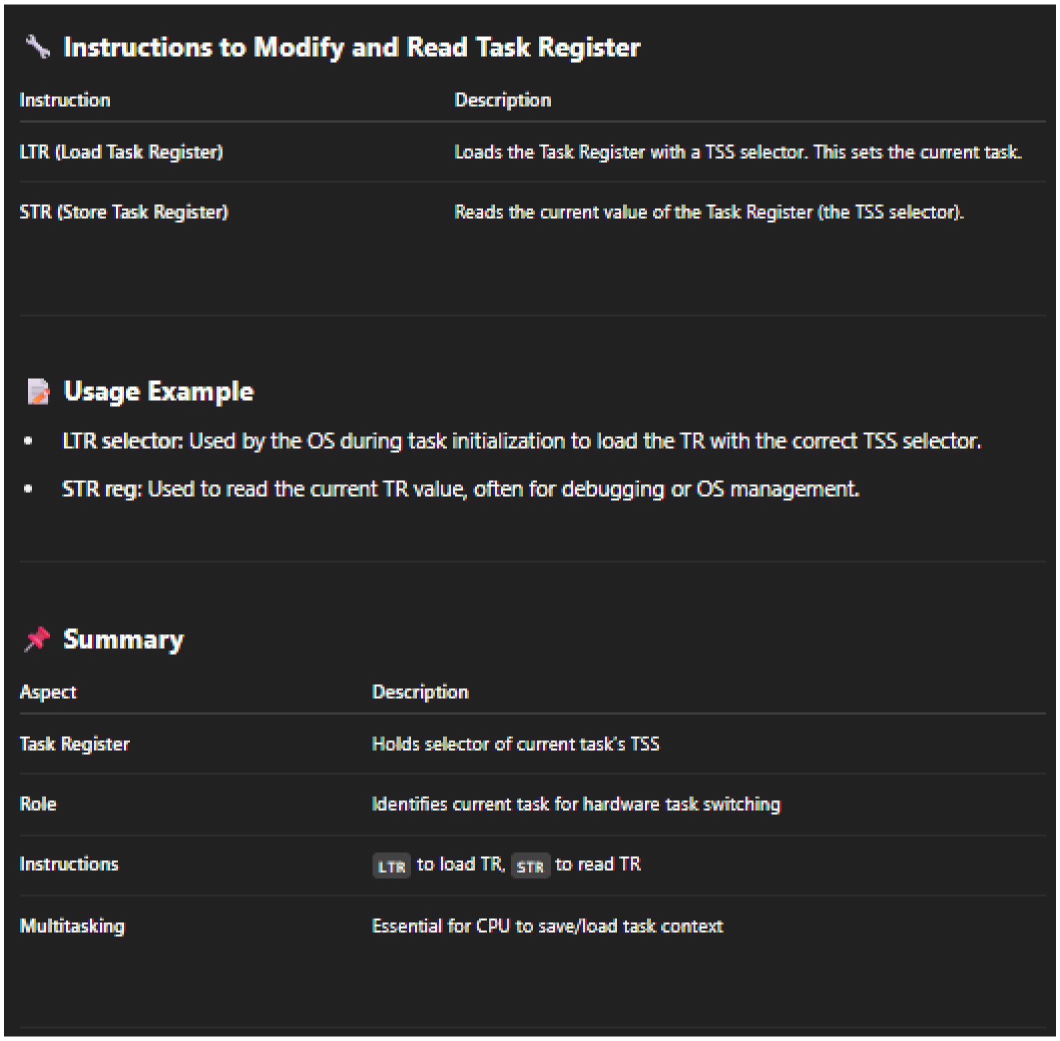


✅ Role of Task Register (TR) in Multitasking

- The Task Register (TR) is a special CPU register in the **Intel 80386** that **holds the segment selector** of the currently running task's **Task State Segment (TSS)** descriptor.
- It points to the **TSS descriptor** in the **Global Descriptor Table (GDT)**.
- The **TSS** stores all the context (CPU state, stack pointers, segment selectors, etc.) for the current task.

How TR Supports Multitasking

1. **Identifies Current Task's TSS:**
   - TR stores the selector that points to the current task's TSS.
   - The CPU uses this selector to find the TSS and load/save task state during task switches.
2. **Facilitates Hardware Task Switching:**
   - When a task switch happens, the CPU uses the TR to:
     - Save the current task's state in its TSS.
     - Load the new task's state from the TSS pointed by the new selector.
3. **Provides Control & Isolation:**
   - Helps the CPU know **which task is running**.
   - Maintains isolation by managing TSS references.



🔧 Instructions to Modify and Read Task Register

Instruction	Description
LTR (Load Task Register)	Loads the Task Register with a TSS selector. This sets the current task.
STR (Store Task Register)	Reads the current value of the Task Register (the TSS selector).

📝 Usage Example

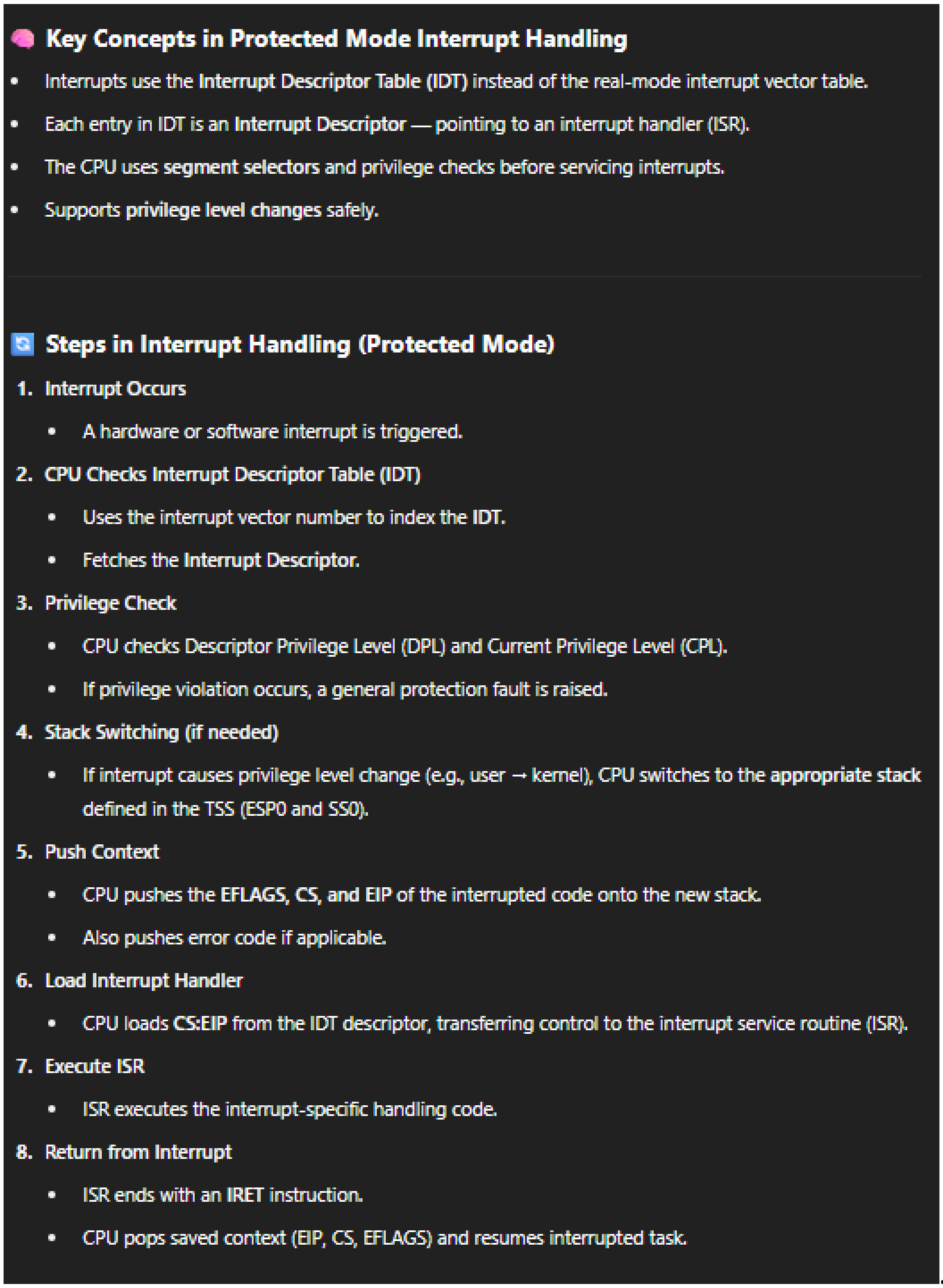
- **LTR selector:** Used by the OS during task initialization to load the TR with the correct TSS selector.
- **STR reg:** Used to read the current TR value, often for debugging or OS management.

📌 Summary

Aspect	Description
Task Register	Holds selector of current task's TSS
Role	Identifies current task for hardware task switching
Instructions	`LTR` to load TR, `STR` to read TR
Multitasking	Essential for CPU to save/load task context

UNIT 6

1) **With the help of neat diagram Explain the Process of handling Interrupts in Protected mode**



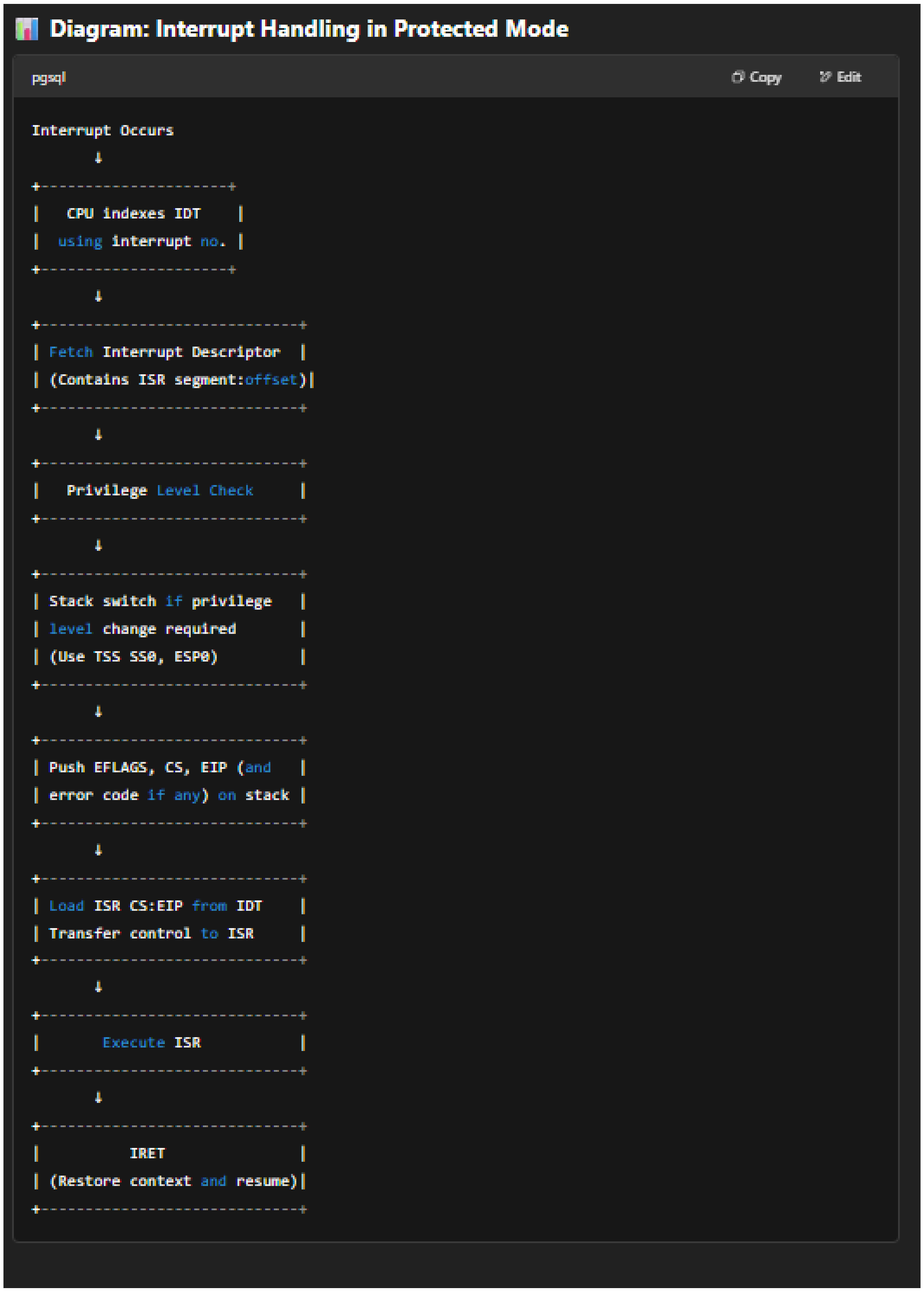
🧠 Key Concepts in Protected Mode Interrupt Handling

- Interrupts use the **Interrupt Descriptor Table (IDT)** instead of the real-mode interrupt vector table.
- Each entry in IDT is an **Interrupt Descriptor** — pointing to an interrupt handler (ISR).
- The CPU uses **segment selectors** and **privilege checks** before servicing interrupts.
- Supports **privilege level changes** safely.

🔄 Steps in Interrupt Handling (Protected Mode)

1. **Interrupt Occurs**
   - A hardware or software interrupt is triggered.
2. **CPU Checks Interrupt Descriptor Table (IDT)**
   - Uses the interrupt vector number to index the **IDT**.
   - Fetches the **Interrupt Descriptor**.
3. **Privilege Check**
   - CPU checks Descriptor Privilege Level (DPL) and Current Privilege Level (CPL).
   - If privilege violation occurs, a general protection fault is raised.
4. **Stack Switching (if needed)**
   - If interrupt causes privilege level change (e.g., user → kernel), CPU switches to the **appropriate stack** defined in the TSS (ESP0 and SS0).
5. **Push Context**
   - CPU pushes the **EFLAGS, CS, and EIP** of the interrupted code onto the new stack.
   - Also pushes error code if applicable.
6. **Load Interrupt Handler**
   - CPU loads **CS:EIP** from the IDT descriptor, transferring control to the interrupt service routine (ISR).
7. **Execute ISR**
   - ISR executes the interrupt-specific handling code.
8. **Return from Interrupt**
   - ISR ends with an **IRET** instruction.
   - CPU pops saved context (EIP, CS, EFLAGS) and resumes interrupted task.

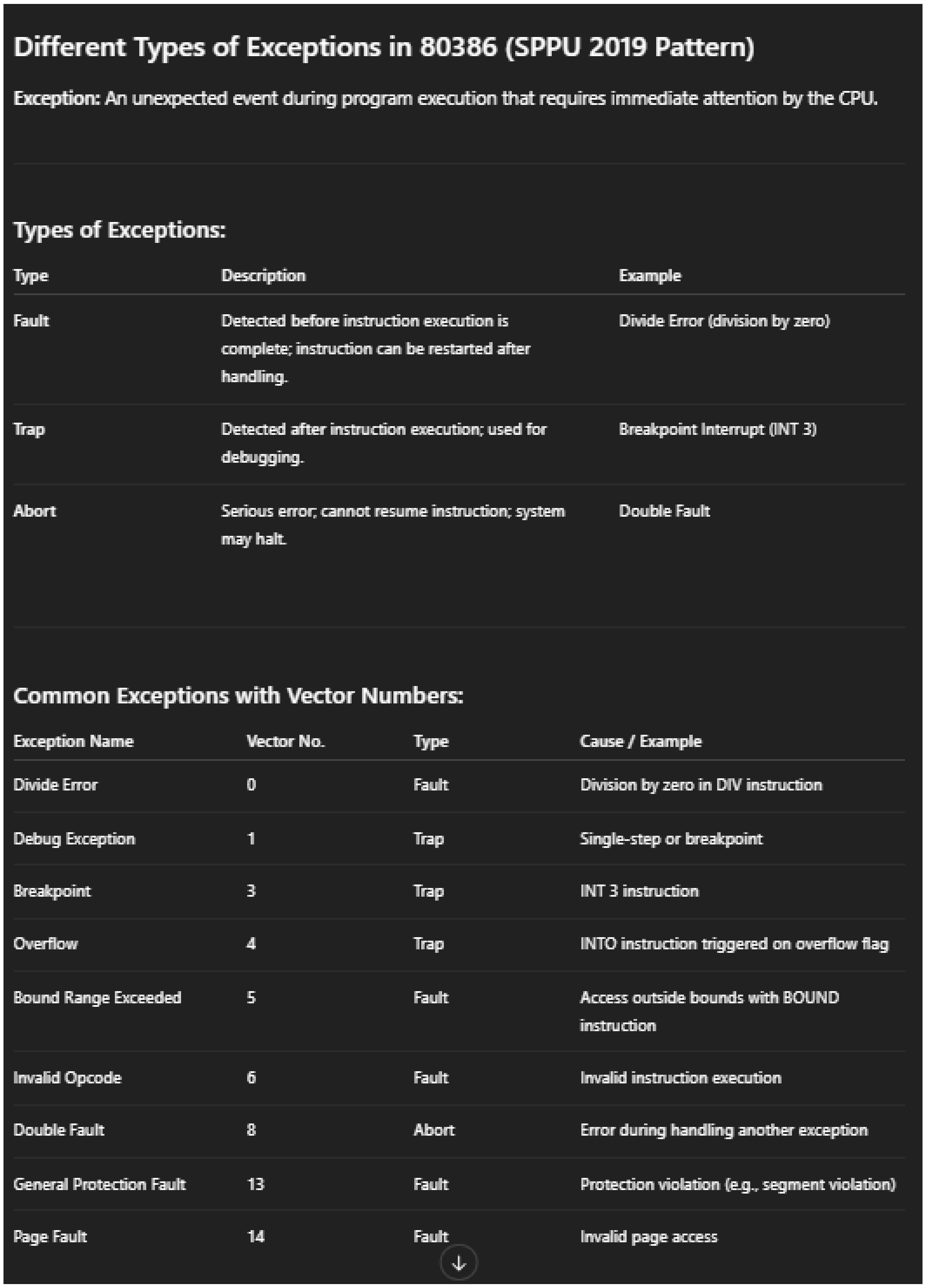
.



📊 Diagram: Interrupt Handling in Protected Mode

```
Interrupt Occurs
       ↓
+---------------------+
|   CPU indexes IDT    |
|  using interrupt no. |
+---------------------+
       ↓
+-----------------------------+
| Fetch Interrupt Descriptor  |
| (Contains ISR segment:offset)|
+-----------------------------+
       ↓
+-----------------------------+
|   Privilege Level Check     |
+-----------------------------+
       ↓
+-----------------------------+
| Stack switch if privilege   |
| level change required       |
| (Use TSS SS0, ESP0)         |
+-----------------------------+
       ↓
+-----------------------------+
| Push EFLAGS, CS, EIP (and   |
| error code if any) on stack |
+-----------------------------+
       ↓
+-----------------------------+
| Load ISR CS:EIP from IDT    |
| Transfer control to ISR     |
+-----------------------------+
       ↓
+-----------------------------+
|       Execute ISR           |
+-----------------------------+
       ↓
+-----------------------------+
|          IRET               |
| (Restore context and resume)|
+-----------------------------+
```

2) Explain the different types of exceptions in 80386 with suitable example



Different Types of Exceptions in 80386 (SPPU 2019 Pattern)

Exception: An unexpected event during program execution that requires immediate attention by the CPU.

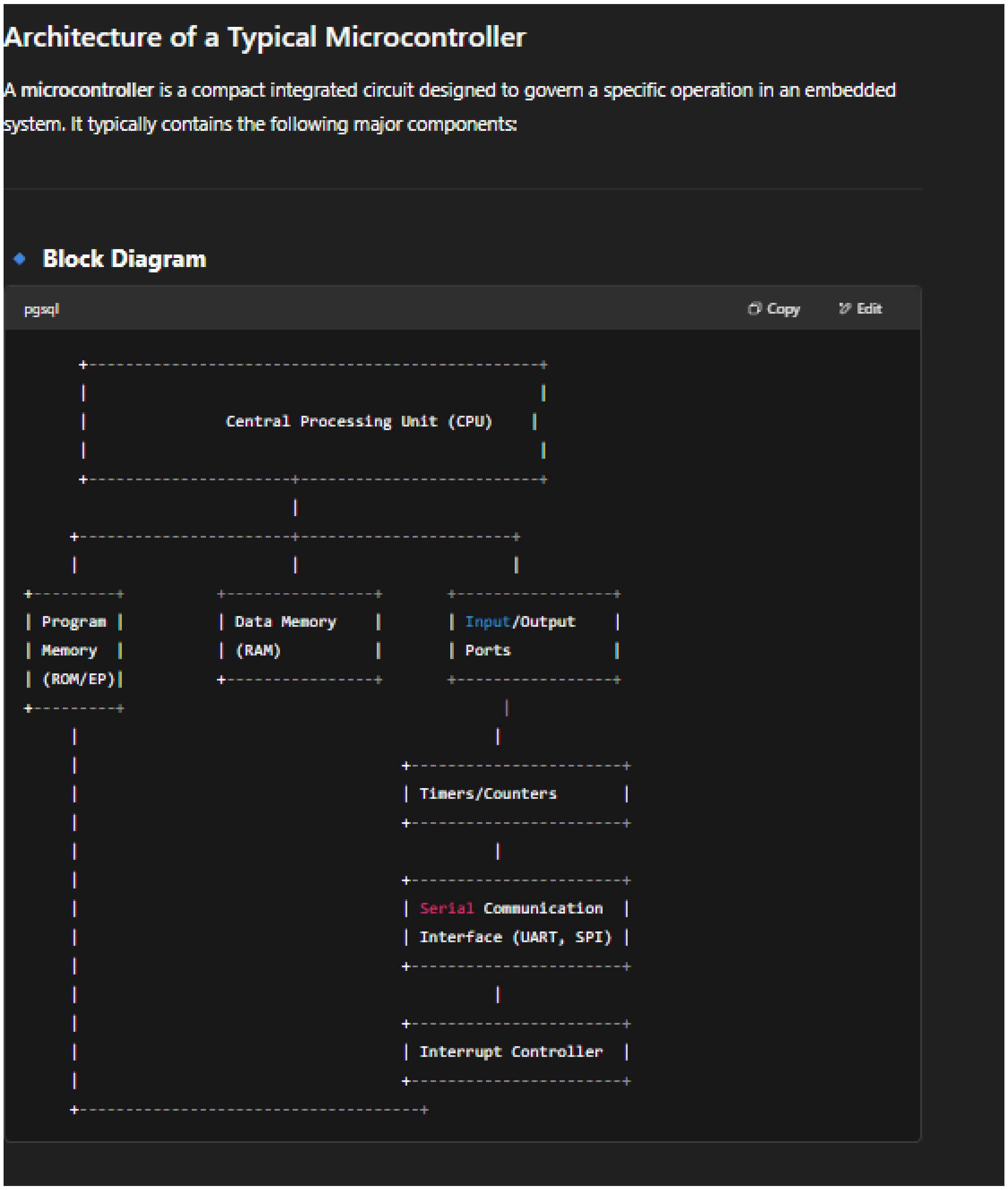
Types of Exceptions:

Type	Description	Example
Fault	Detected before instruction execution is complete; instruction can be restarted after handling.	Divide Error (division by zero)
Trap	Detected after instruction execution; used for debugging.	Breakpoint Interrupt (INT 3)
Abort	Serious error; cannot resume instruction; system may halt.	Double Fault

Common Exceptions with Vector Numbers:

Exception Name	Vector No.	Type	Cause / Example
Divide Error	0	Fault	Division by zero in DIV instruction
Debug Exception	1	Trap	Single-step or breakpoint
Breakpoint	3	Trap	INT 3 instruction
Overflow	4	Trap	INTO instruction triggered on overflow flag
Bound Range Exceeded	5	Fault	Access outside bounds with BOUND instruction
Invalid Opcode	6	Fault	Invalid instruction execution
Double Fault	8	Abort	Error during handling another exception
General Protection Fault	13	Fault	Protection violation (e.g., segment violation)
Page Fault	14	Fault	Invalid page access

3) With the help of neat diagram explain the architecture of typical Microcontroller.



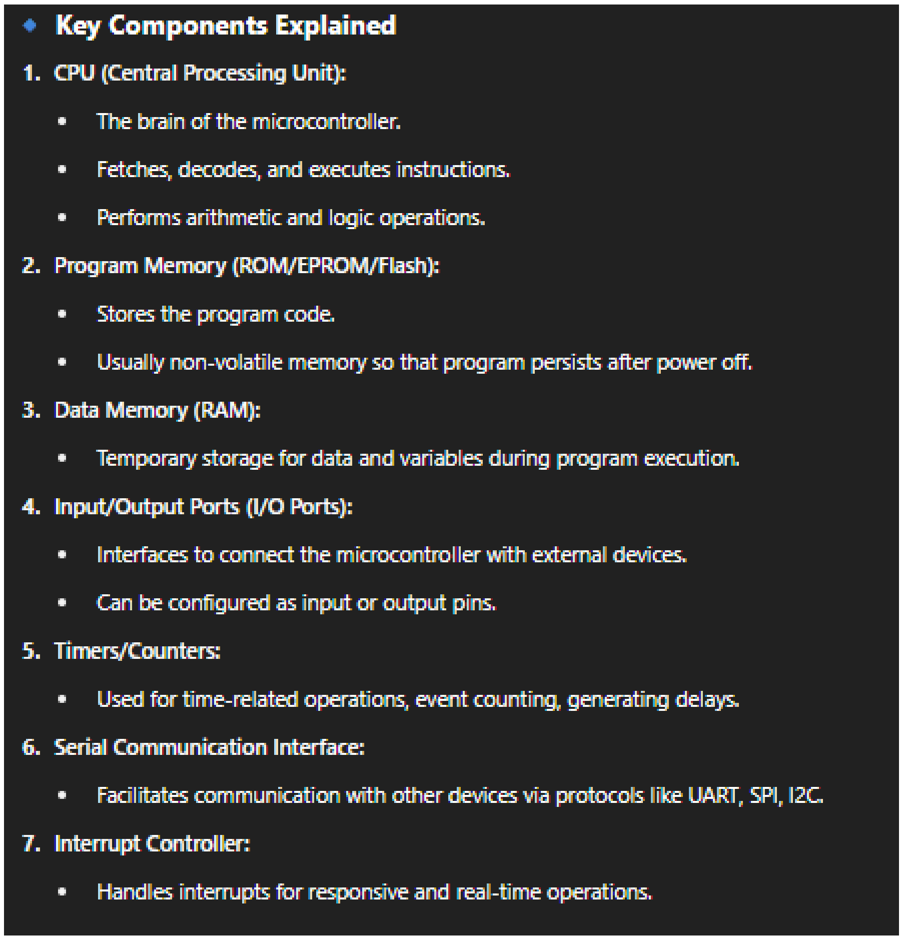
Architecture of a Typical Microcontroller

A **microcontroller** is a compact integrated circuit designed to govern a specific operation in an embedded system. It typically contains the following major components:

🔹 Block Diagram

pgsql

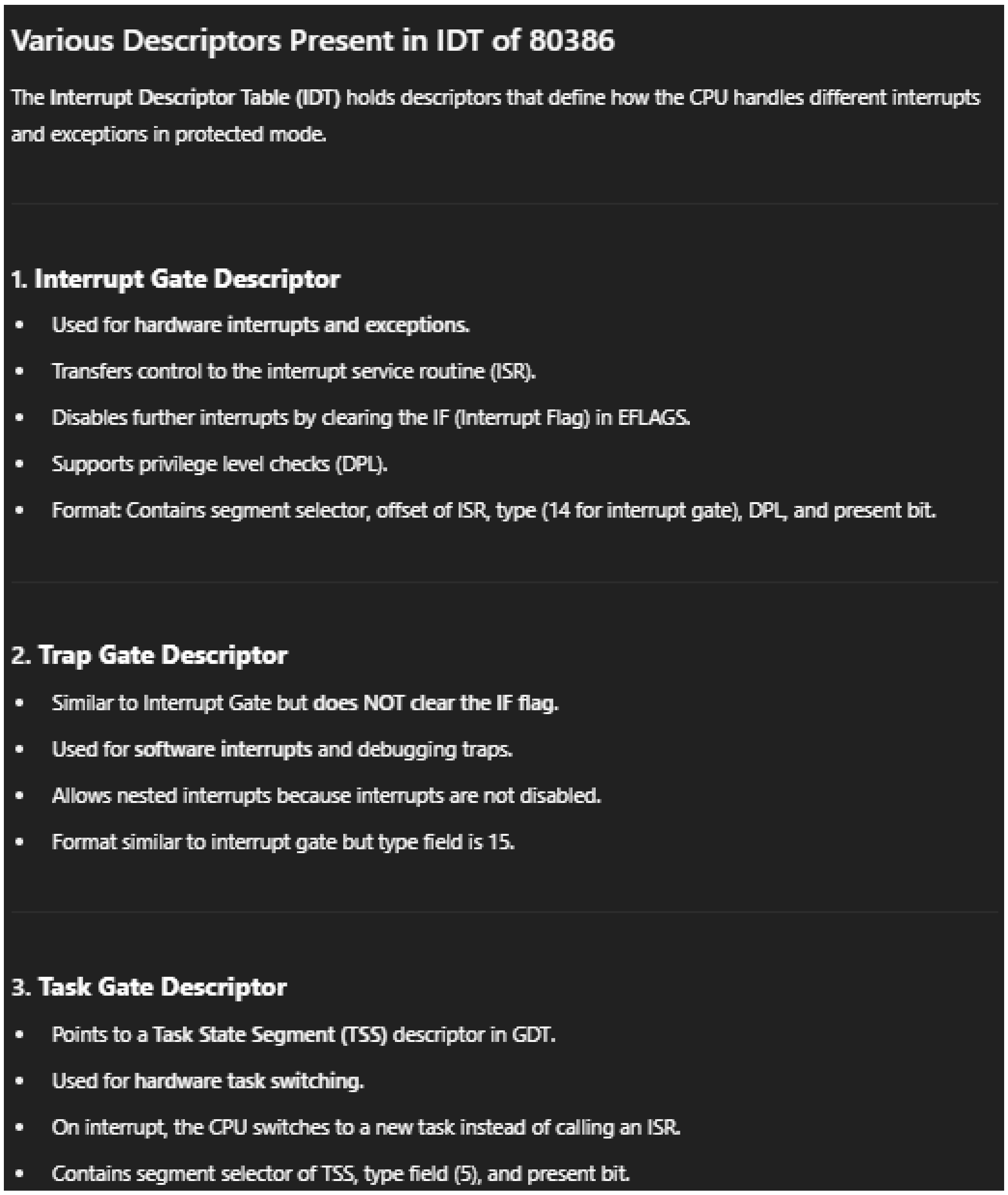
```
      +-------------------------------------------------+
      |                                                 |
      |               Central Processing Unit (CPU)    |
      |                                                 |
      +----------------------+--------------------------+
                             |
     +-----------------------+-----------------------+
     |                       |                       |
+---------+          +----------------+       +-----------------+
| Program |          | Data Memory    |       | Input/Output    |
| Memory  |          | (RAM)          |       | Ports           |
| (ROM/EP)|          +----------------+       +-----------------+
+---------+                                          |
     |                                             |
     |                                   +-----------------------+
     |                                   | Timers/Counters       |
     |                                   +-----------------------+
     |                                             |
     |                                   +-----------------------+
     |                                   | Serial Communication  |
     |                                   | Interface (UART, SPI) |
     |                                   +-----------------------+
     |                                             |
     |                                   +-----------------------+
     |                                   | Interrupt Controller  |
     |                                   +-----------------------+
     +-------------------------------------+
```



🔹 **Key Components Explained**

1. **CPU (Central Processing Unit):**
   - The brain of the microcontroller.
   - Fetches, decodes, and executes instructions.
   - Performs arithmetic and logic operations.
2. **Program Memory (ROM/EPROM/Flash):**
   - Stores the program code.
   - Usually non-volatile memory so that program persists after power off.
3. **Data Memory (RAM):**
   - Temporary storage for data and variables during program execution.
4. **Input/Output Ports (I/O Ports):**
   - Interfaces to connect the microcontroller with external devices.
   - Can be configured as input or output pins.
5. **Timers/Counters:**
   - Used for time-related operations, event counting, generating delays.
6. **Serial Communication Interface:**
   - Facilitates communication with other devices via protocols like UART, SPI, I2C.
7. **Interrupt Controller:**
   - Handles interrupts for responsive and real-time operations.

4) Explain various Descriptors present in IDT of 80386



Various Descriptors Present in IDT of 80386

The **Interrupt Descriptor Table (IDT)** holds descriptors that define how the CPU handles different interrupts and exceptions in protected mode.

1. Interrupt Gate Descriptor

- Used for **hardware interrupts and exceptions.**
- Transfers control to the interrupt service routine (ISR).
- Disables further interrupts by clearing the IF (Interrupt Flag) in EFLAGS.
- Supports privilege level checks (DPL).
- Format: Contains segment selector, offset of ISR, type (14 for interrupt gate), DPL, and present bit.

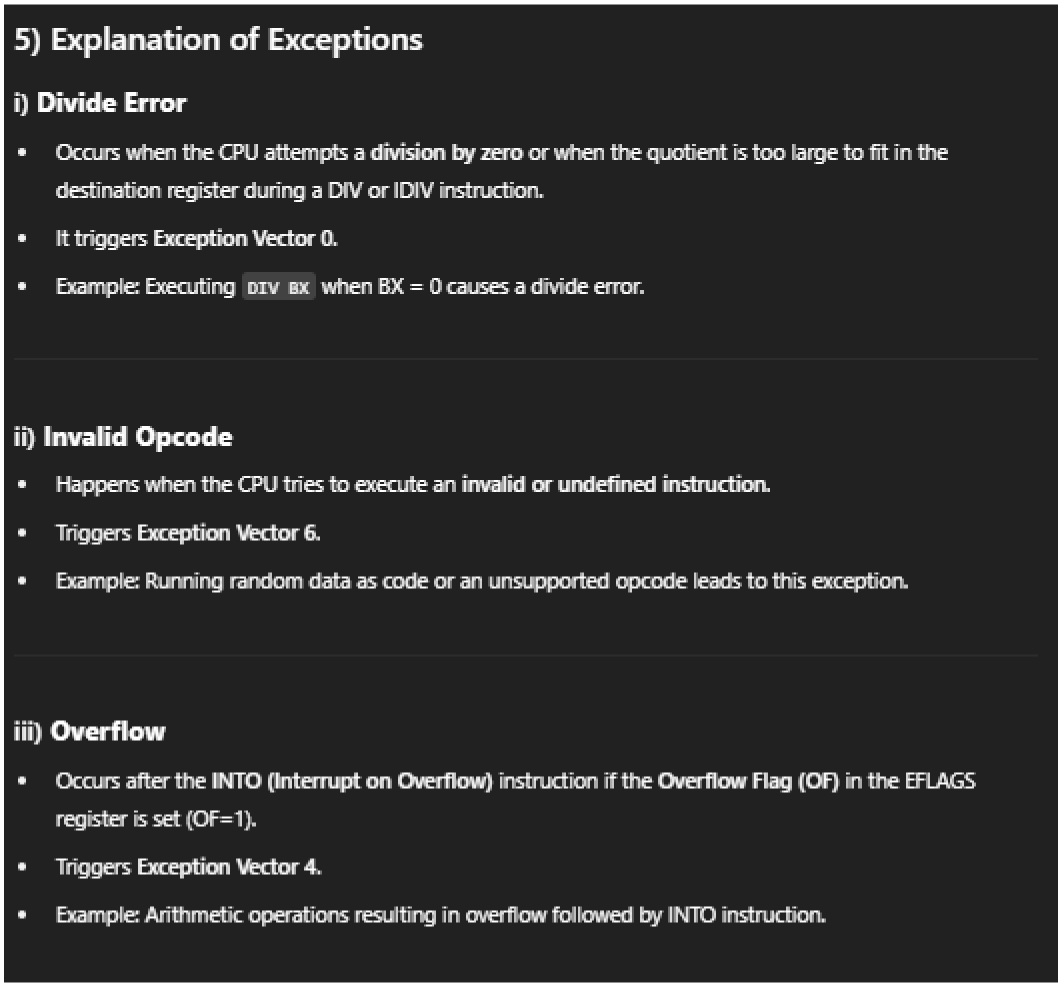
2. Trap Gate Descriptor

- Similar to Interrupt Gate but **does NOT clear the IF flag.**
- Used for **software interrupts** and debugging traps.
- Allows nested interrupts because interrupts are not disabled.
- Format similar to interrupt gate but type field is 15.

3. Task Gate Descriptor

- Points to a **Task State Segment** (TSS) descriptor in GDT.
- Used for **hardware task switching.**
- On interrupt, the CPU switches to a new task instead of calling an ISR.
- Contains segment selector of TSS, type field (5), and present bit.

5) Explain the following exceptions in brief. i) Divide Error ii) Invalid op code iii) Overflow



5) Explanation of Exceptions

i) **Divide Error**

- Occurs when the CPU attempts a **division by zero** or when the quotient is too large to fit in the destination register during a DIV or IDIV instruction.
- It triggers **Exception Vector 0.**
- Example: Executing `DIV BX` when BX = 0 causes a divide error.

---

ii) **Invalid Opcode**

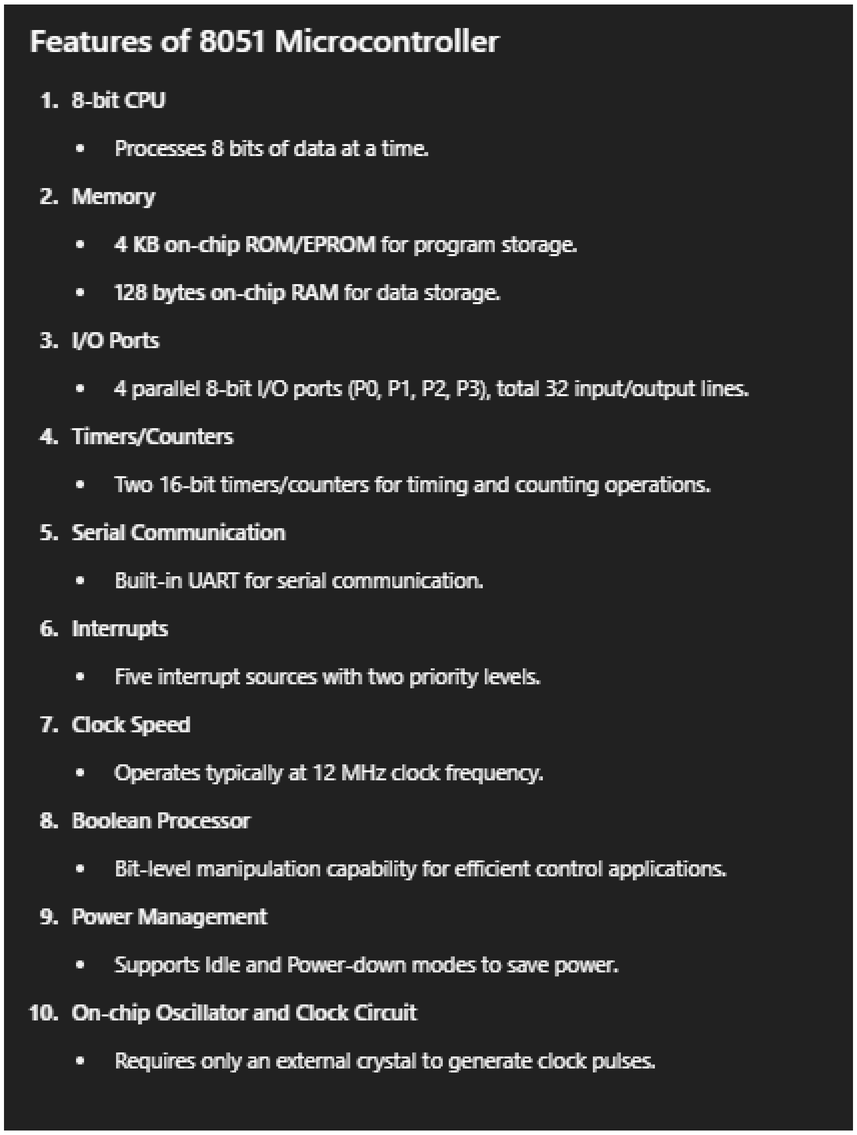
- Happens when the CPU tries to execute an **invalid or undefined instruction.**
- Triggers **Exception Vector 6.**
- Example: Running random data as code or an unsupported opcode leads to this exception.

---

iii) **Overflow**

- Occurs after the **INTO (Interrupt on Overflow)** instruction if the **Overflow Flag (OF)** in the EFLAGS register is set (OF=1).
- Triggers **Exception Vector 4.**
- Example: Arithmetic operations resulting in overflow followed by INTO instruction.

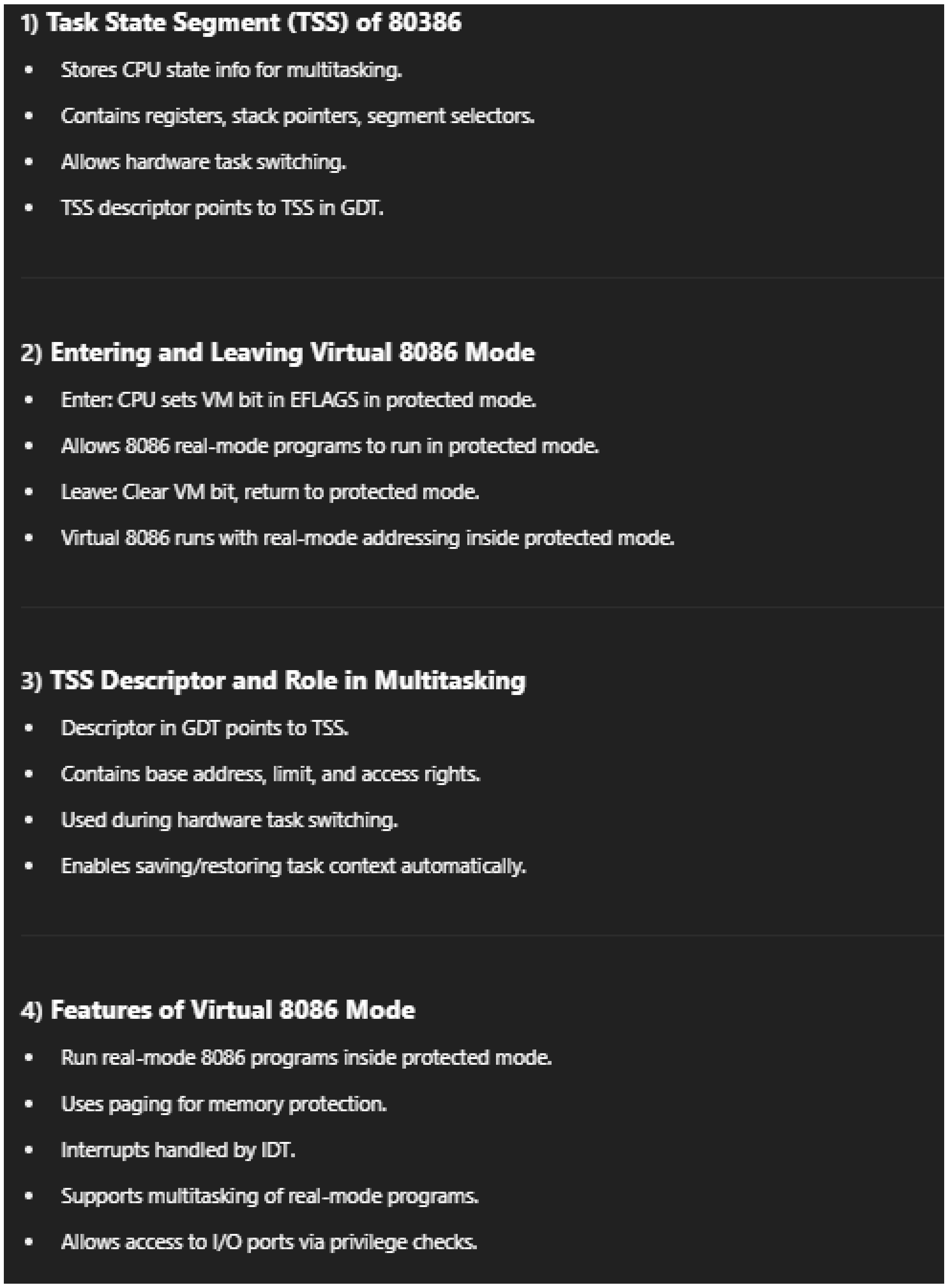
6) **Explain various features of the 8051 Microcontroller.**



Features of 8051 Microcontroller

1. **8-bit CPU**
   - Processes 8 bits of data at a time.
2. **Memory**
   - 4 KB on-chip ROM/EPROM for program storage.
   - 128 bytes on-chip RAM for data storage.
3. **I/O Ports**
   - 4 parallel 8-bit I/O ports (P0, P1, P2, P3), total 32 input/output lines.
4. **Timers/Counters**
   - Two 16-bit timers/counters for timing and counting operations.
5. **Serial Communication**
   - Built-in UART for serial communication.
6. **Interrupts**
   - Five interrupt sources with two priority levels.
7. **Clock Speed**
   - Operates typically at 12 MHz clock frequency.
8. **Boolean Processor**
   - Bit-level manipulation capability for efficient control applications.
9. **Power Management**
   - Supports Idle and Power-down modes to save power.
10. **On-chip Oscillator and Clock Circuit**
    - Requires only an external crystal to generate clock pulses.

7) **Elaborate about enabling and disabling interrupts in 80386.**



1) Task State Segment (TSS) of 80386

- Stores CPU state info for multitasking.
- Contains registers, stack pointers, segment selectors.
- Allows hardware task switching.
- TSS descriptor points to TSS in GDT.

2) Entering and Leaving Virtual 8086 Mode

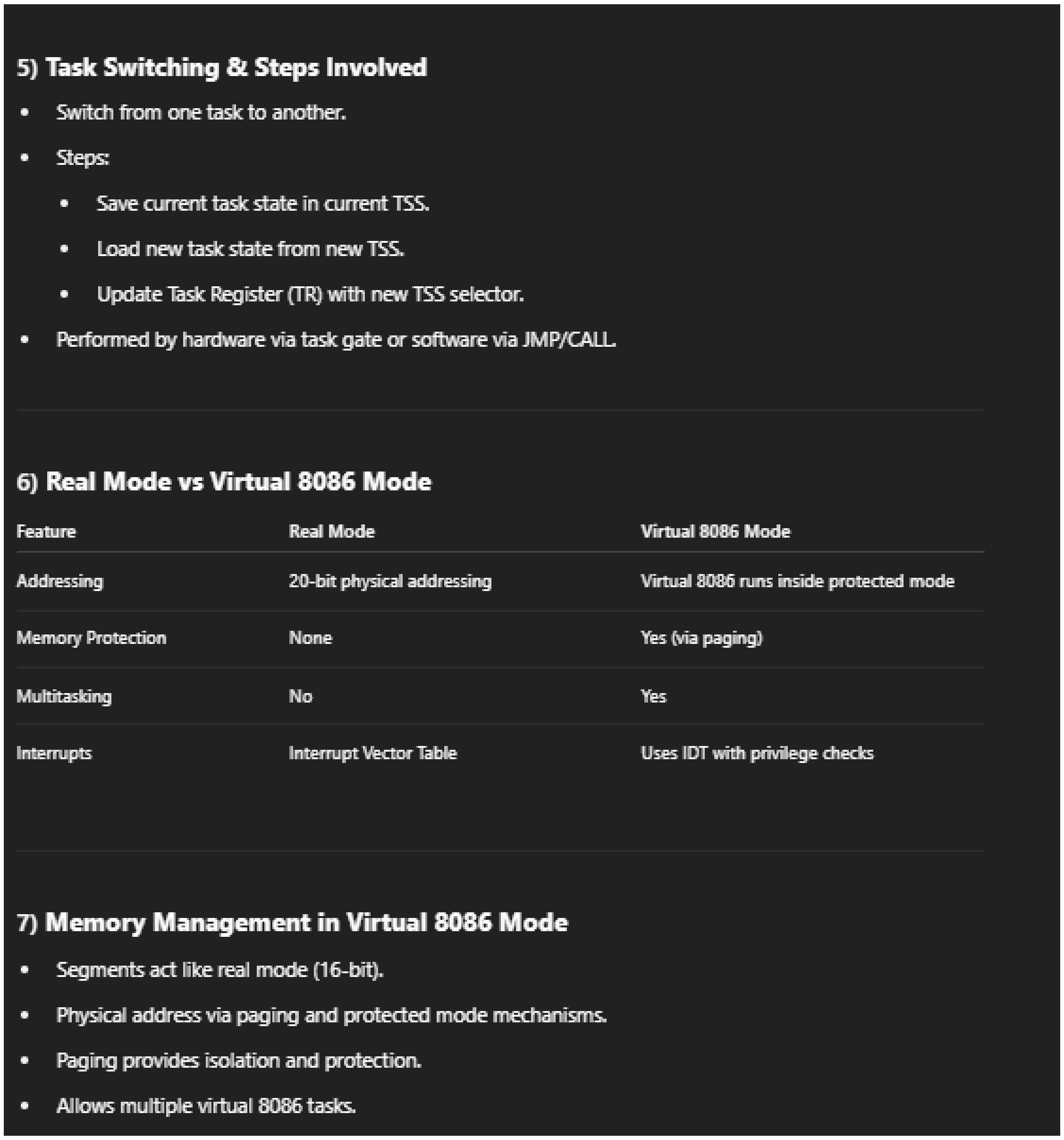
- Enter: CPU sets VM bit in EFLAGS in protected mode.
- Allows 8086 real-mode programs to run in protected mode.
- Leave: Clear VM bit, return to protected mode.
- Virtual 8086 runs with real-mode addressing inside protected mode.

3) TSS Descriptor and Role in Multitasking

- Descriptor in GDT points to TSS.
- Contains base address, limit, and access rights.
- Used during hardware task switching.
- Enables saving/restoring task context automatically.

4) Features of Virtual 8086 Mode

- Run real-mode 8086 programs inside protected mode.
- Uses paging for memory protection.
- Interrupts handled by IDT.
- Supports multitasking of real-mode programs.
- Allows access to I/O ports via privilege checks.



5) Task Switching & Steps Involved

- Switch from one task to another.
- Steps:
  - Save current task state in current TSS.
  - Load new task state from new TSS.
  - Update Task Register (TR) with new TSS selector.
- Performed by hardware via task gate or software via JMP/CALL.

---

6) Real Mode vs Virtual 8086 Mode

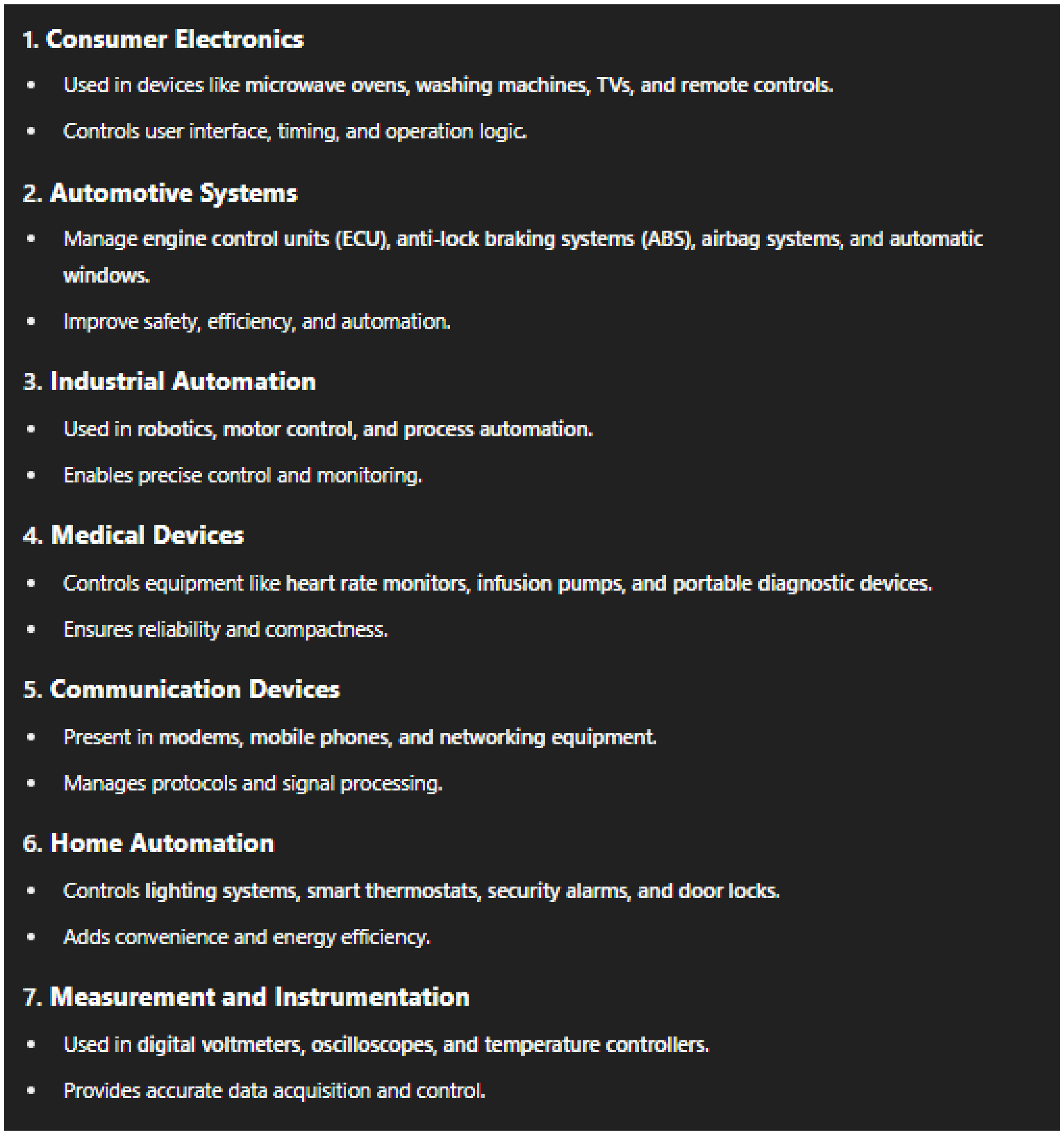
Feature	Real Mode	Virtual 8086 Mode
Addressing	20-bit physical addressing	Virtual 8086 runs inside protected mode
Memory Protection	None	Yes (via paging)
Multitasking	No	Yes
Interrupts	Interrupt Vector Table	Uses IDT with privilege checks

---

7) Memory Management in Virtual 8086 Mode

- Segments act like real mode (16-bit).
- Physical address via paging and protected mode mechanisms.
- Paging provides isolation and protection.
- Allows multiple virtual 8086 tasks.

8) List and elaborate on different applications of microcontrollers



1. Consumer Electronics

- Used in devices like **microwave ovens, washing machines, TVs, and remote controls.**
- Controls user interface, timing, and operation logic.

2. Automotive Systems

- Manage **engine control units (ECU), anti-lock braking systems (ABS), airbag systems, and automatic windows.**
- Improve safety, efficiency, and automation.

3. Industrial Automation

- Used in **robotics, motor control, and process automation.**
- Enables precise control and monitoring.

4. Medical Devices

- Controls equipment like **heart rate monitors, infusion pumps, and portable diagnostic devices.**
- Ensures reliability and compactness.

5. Communication Devices

- Present in **modems, mobile phones, and networking equipment.**
- Manages protocols and signal processing.

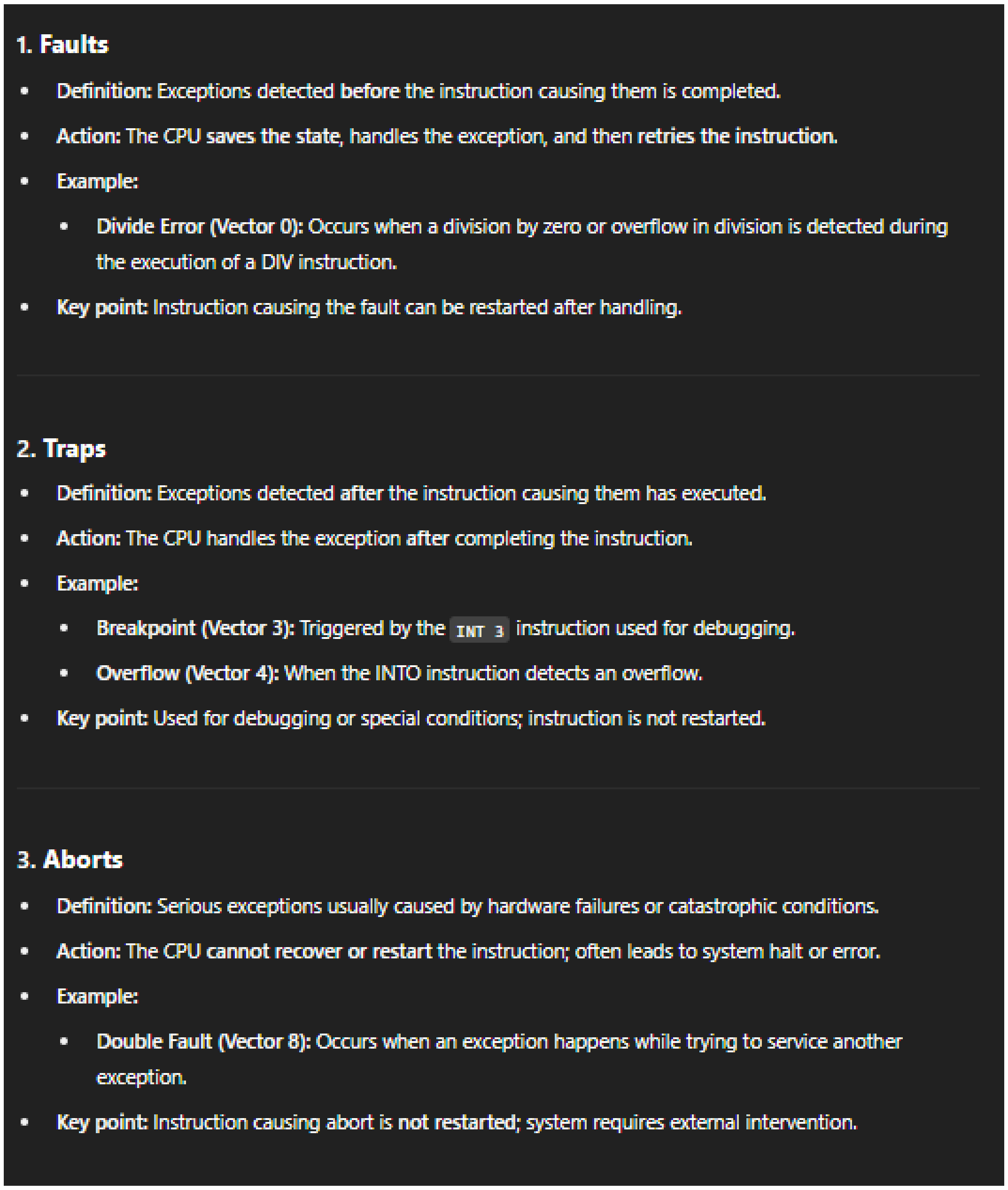
6. Home Automation

- Controls **lighting systems, smart thermostats, security alarms, and door locks.**
- Adds convenience and energy efficiency.

7. Measurement and Instrumentation

- Used in **digital voltmeters, oscilloscopes, and temperature controllers.**
- Provides accurate data acquisition and control.

9) Explain the following exception conditions with an example: Faults, Traps, and Aborts.



1. Faults

- **Definition:** Exceptions detected **before** the instruction causing them is completed.
- **Action:** The CPU **saves the state**, handles the exception, and then **retries the instruction**.
- **Example:**
  - **Divide Error (Vector 0):** Occurs when a division by zero or overflow in division is detected during the execution of a DIV instruction.
- **Key point:** Instruction causing the fault can be restarted after handling.

---

2. Traps

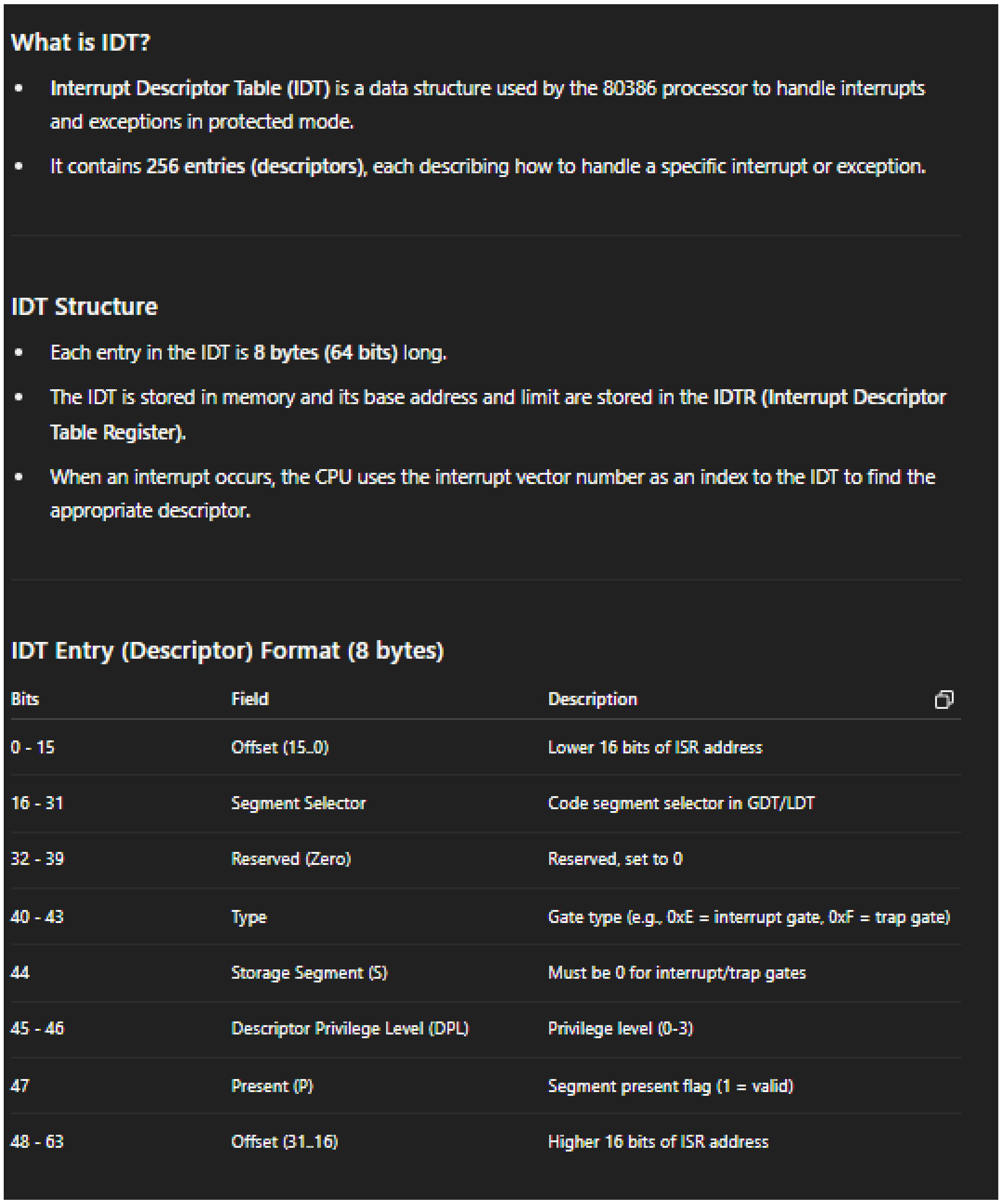
- **Definition:** Exceptions detected **after** the instruction causing them has executed.
- **Action:** The CPU handles the exception **after completing** the instruction.
- **Example:**
  - **Breakpoint (Vector 3):** Triggered by the `INT 3` instruction used for debugging.
  - **Overflow (Vector 4):** When the INTO instruction detects an overflow.
- **Key point:** Used for debugging or special conditions; instruction is not restarted.

---

3. Aborts

- **Definition:** Serious exceptions usually caused by hardware failures or catastrophic conditions.
- **Action:** The CPU **cannot recover or restart** the instruction; often leads to system halt or error.
- **Example:**
  - **Double Fault (Vector 8):** Occurs when an exception happens while trying to service another exception.
- **Key point:** Instruction causing abort is **not restarted**; system requires external intervention.

10) With the help of the necessary diagram, explain the structure of IDT in 80386.



What is IDT?

- **Interrupt Descriptor Table (IDT)** is a data structure used by the 80386 processor to handle interrupts and exceptions in protected mode.
- It contains **256 entries (descriptors)**, each describing how to handle a specific interrupt or exception.

IDT Structure

- Each entry in the IDT is **8 bytes (64 bits)** long.
- The IDT is stored in memory and its base address and limit are stored in the **IDTR (Interrupt Descriptor Table Register)**.
- When an interrupt occurs, the CPU uses the interrupt vector number as an index to the IDT to find the appropriate descriptor.

IDT Entry (Descriptor) Format (8 bytes)

Bits	Field	Description
0 - 15	Offset (15..0)	Lower 16 bits of ISR address
16 - 31	Segment Selector	Code segment selector in GDT/LDT
32 - 39	Reserved (Zero)	Reserved, set to 0
40 - 43	Type	Gate type (e.g., 0xE = interrupt gate, 0xF = trap gate)
44	Storage Segment (S)	Must be 0 for interrupt/trap gates
45 - 46	Descriptor Privilege Level (DPL)	Privilege level (0-3)
47	Present (P)	Segment present flag (1 = valid)
48 - 63	Offset (31..16)	Higher 16 bits of ISR address

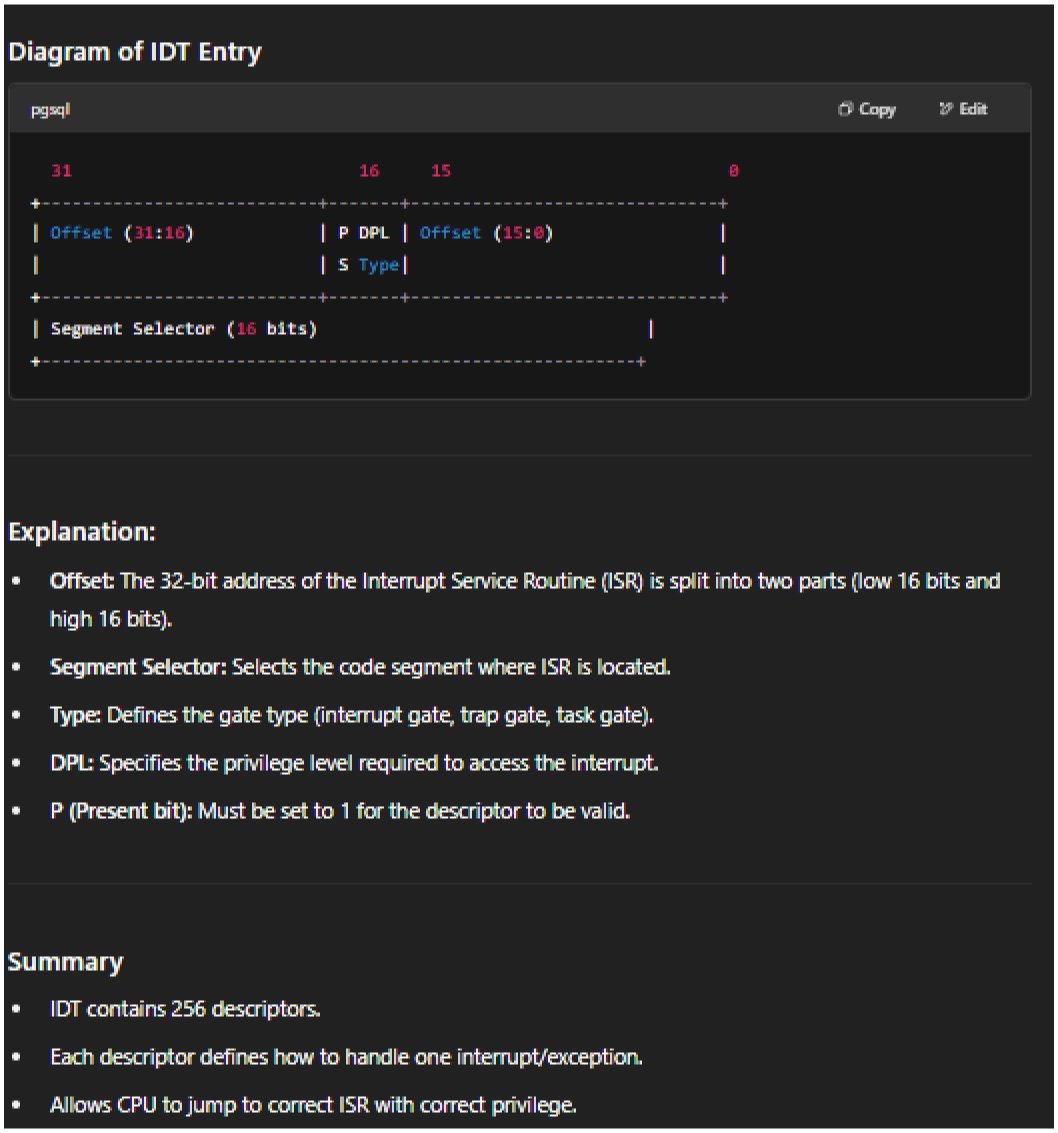


Diagram of IDT Entry

pgsql

Copy Edit

```
  31                            16     15                           0
+---------------------------+-------+------------------------------+
| Offset (31:16)            | P DPL | Offset (15:0)                |
|                           | S Type|                              |
+---------------------------+-------+------------------------------+
| Segment Selector (16 bits)                                |
+----------------------------------------------------------+
```

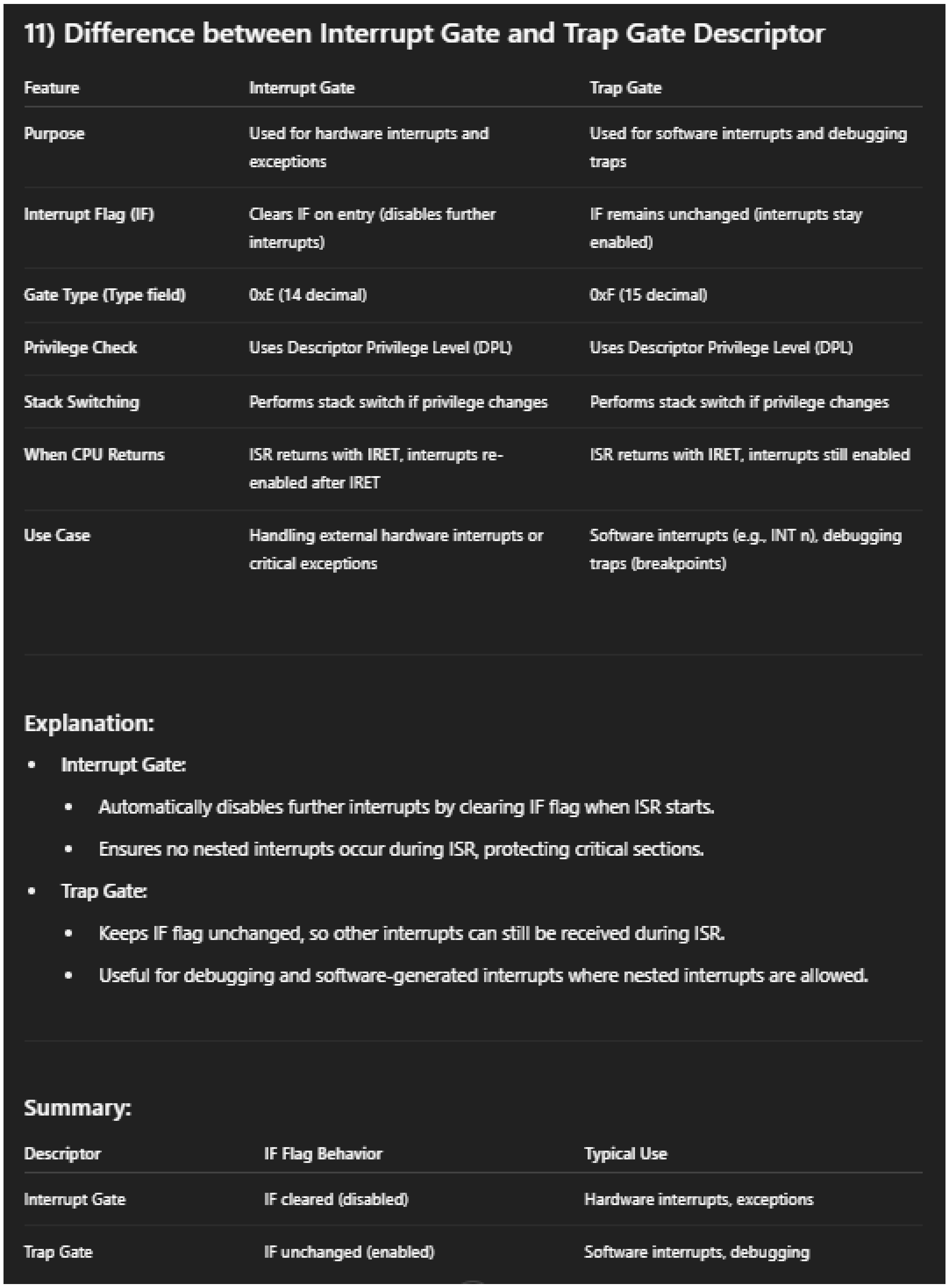
Explanation:

- **Offset:** The 32-bit address of the Interrupt Service Routine (ISR) is split into two parts (low 16 bits and high 16 bits).
- **Segment Selector:** Selects the code segment where ISR is located.
- **Type:** Defines the gate type (interrupt gate, trap gate, task gate).
- **DPL:** Specifies the privilege level required to access the interrupt.
- **P (Present bit):** Must be set to 1 for the descriptor to be valid.

Summary

- IDT contains 256 descriptors.
- Each descriptor defines how to handle one interrupt/exception.
- Allows CPU to jump to correct ISR with correct privilege.

11) Differentiate and Explain the Interrupt gate and Trap gate descriptor



11) Difference between Interrupt Gate and Trap Gate Descriptor

Feature	Interrupt Gate	Trap Gate
Purpose	Used for hardware interrupts and exceptions	Used for software interrupts and debugging traps
Interrupt Flag (IF)	Clears IF on entry (disables further interrupts)	IF remains unchanged (interrupts stay enabled)
Gate Type (Type field)	0xE (14 decimal)	0xF (15 decimal)
Privilege Check	Uses Descriptor Privilege Level (DPL)	Uses Descriptor Privilege Level (DPL)
Stack Switching	Performs stack switch if privilege changes	Performs stack switch if privilege changes
When CPU Returns	ISR returns with IRET, interrupts re-enabled after IRET	ISR returns with IRET, interrupts still enabled
Use Case	Handling external hardware interrupts or critical exceptions	Software interrupts (e.g., INT n), debugging traps (breakpoints)

Explanation:

- **Interrupt Gate:**
  - Automatically disables further interrupts by clearing IF flag when ISR starts.
  - Ensures no nested interrupts occur during ISR, protecting critical sections.
- **Trap Gate:**
  - Keeps IF flag unchanged, so other interrupts can still be received during ISR.
  - Useful for debugging and software-generated interrupts where nested interrupts are allowed.

Summary:

Descriptor	IF Flag Behavior	Typical Use
Interrupt Gate	IF cleared (disabled)	Hardware interrupts, exceptions
Trap Gate	IF unchanged (enabled)	Software interrupts, debugging

All The Best!!
- Anish Joshi